Git

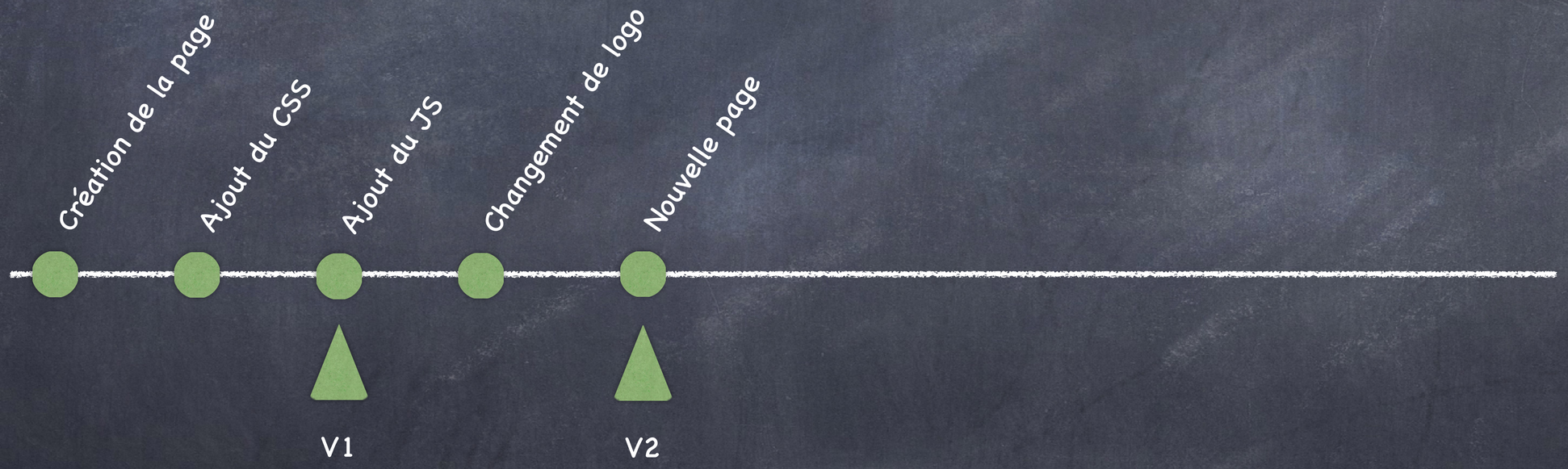
et

Github

Introduction

Définition

Outil permettant de contrôler
les différentes versions
d'un ou de plusieurs documents

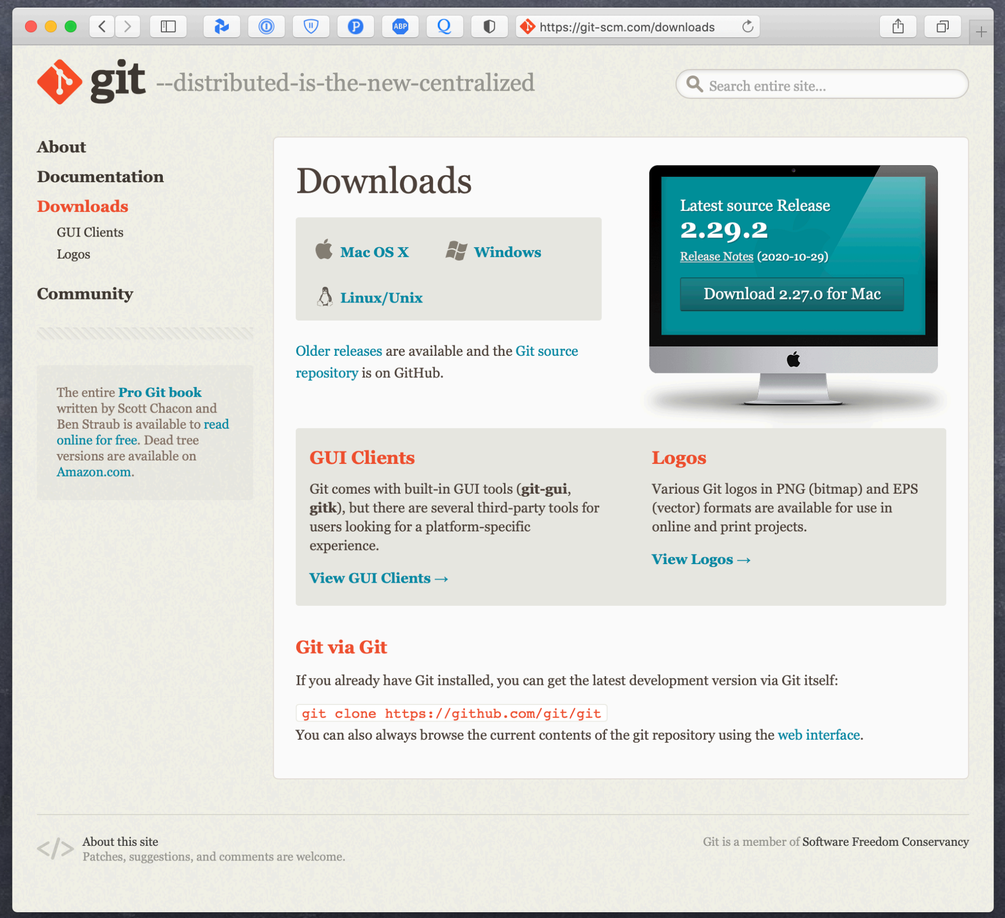


A quoi ça sert

- ✓ Historique des modifications sur nos fichiers
- ✓ Archive locales et distantes (possibilité de travailler en équipe)
- ✓ Versions multiples (tags et branches)
- ✓ Suivi de projet
- ✓ Déploiement de projet
- ✓ Filet de sécurité

Installation

https://git-scm.com/downloads



Mise en place


```
Default (bash)                                ⌥⌘1
mo-ghra-leictreonach:git-test seb$ git init
Initialized empty Git repository in /Users/s
eb/Documents/formation/git/git-test/.git/
mo-ghra-leictreonach:git-test seb$
```

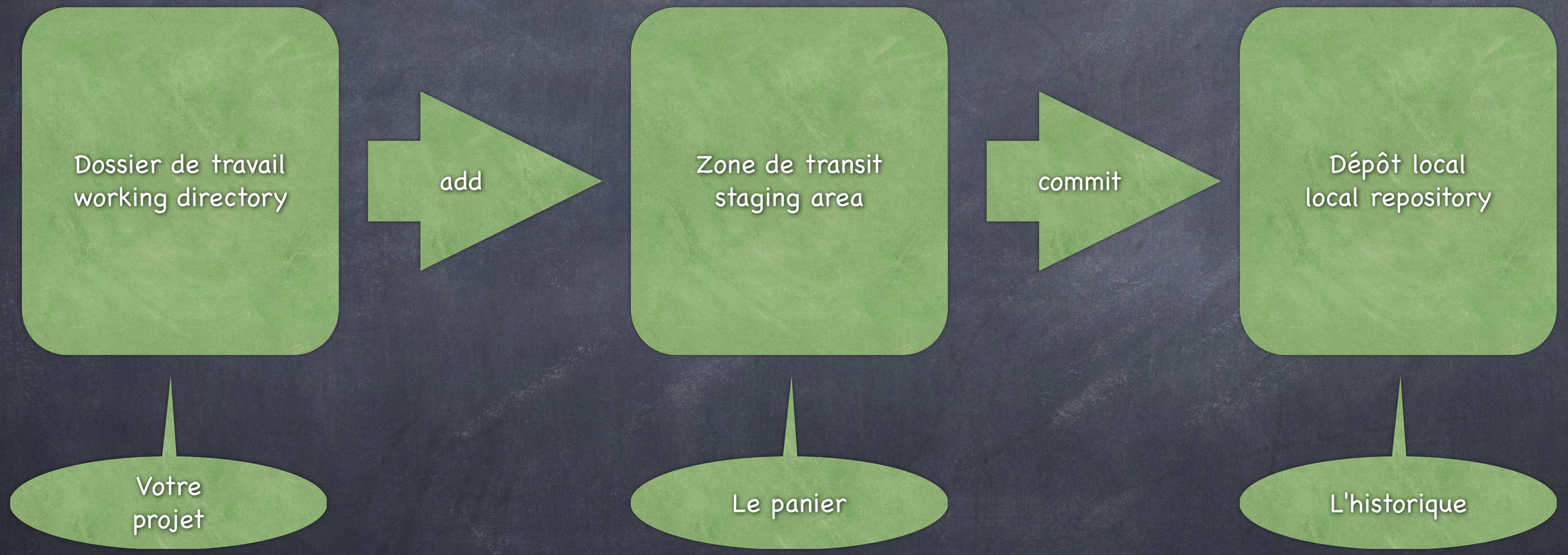
La commande **git init** constitue la première étape, elle initialise un projet git dans le dossier dans lequel elle est exécutée

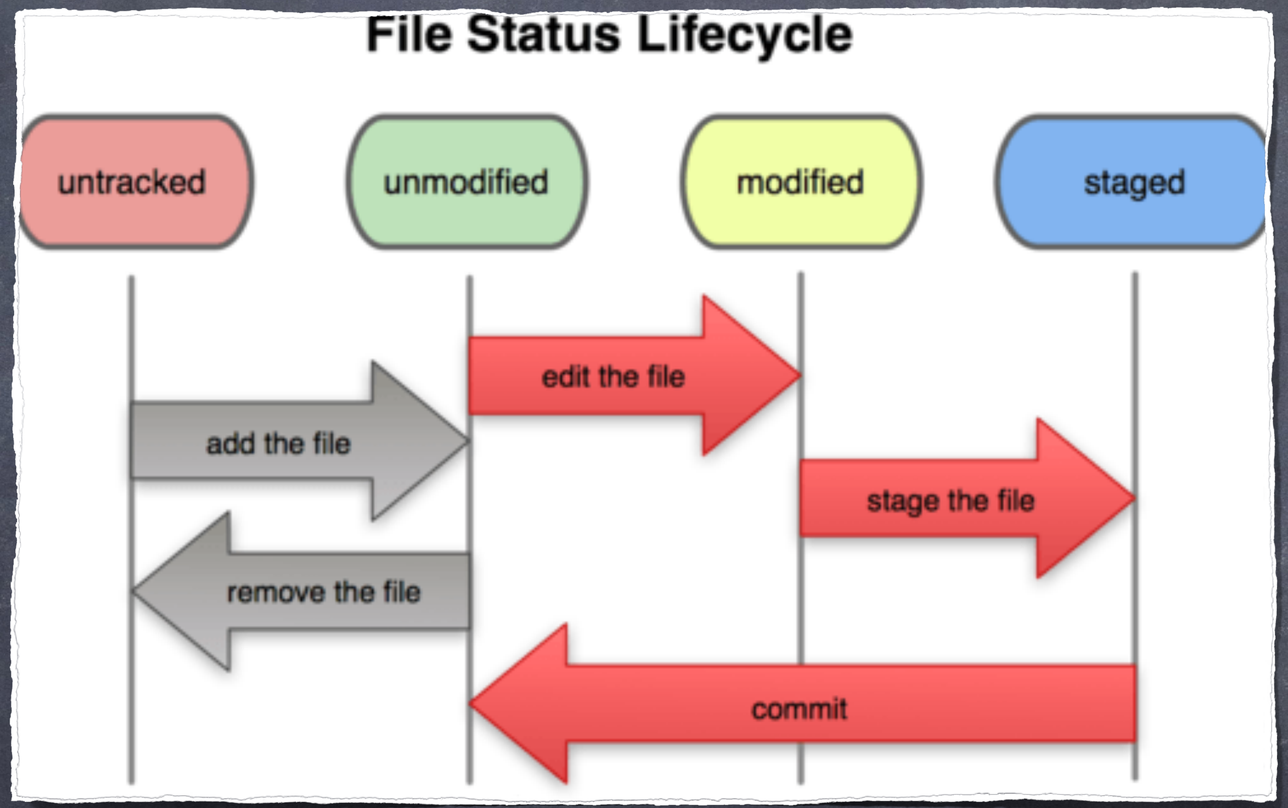
L'historique des fichier
est stocké dans un dossier **.git**

Configuration


```
git config --global user.name seb
git config --global user.email seb@mail.com
```

Principes


Dossier de travail
working directory
add
Zone de transit
staging area
commit
Dépôt local
local repository
Votre projet
Le panier
L'historique


File Status Lifecycle
untracked
unmodified
modified
staged
edit the file
add the file
stage the file
remove the file
commit

Ajouter à la zone de transit


```
git add index.html
```

ajouter le fichier index.html

```
git add images/*.jpg
```

ajouter tous les fichiers jpeg qui sont dans le dossier images

```
git add .
```

ajouter tous le fichiers

Créer un commit


```
git commit -m "un message de commit"
```

```
git commit --message="un message de commit"
```

```
git commit --all --message="un message de commit"
```

Équivalent à un git add . suivi d'un commit

Comment rédiger un message de commit

- ✓ Préférer l'impératif "ajoute le titre de la page"
  plutôt que "ajout du titre de la page"
- ✓ Commencer par un verbe d'action
- ✓ Faire un message court et précis (pas plus de 50 caractères)
- ✓ On peut ajouter une description plus longue
  - ✓ dans ce cas il faut utiliser deux options -m ou --message
  - ✓ ou ne pas utiliser d'option et rédiger le message avec Vi
    (laisser une ligne vide entre le titre et la description)
- ✓ Mentionner les références :
  id de la user story, du bug ou de la tâche
- ✓ Faire des commits atomiques (une seule modification par commit)
- ✓ Indiquer la motivation, le pourquoi dans le message

Commandes Vi en bref

passer en mode insertion de texte	i	copier une ligne	yy
ajouter du texte en fin de ligne	A	copier un mot	yw
ajouter du texte au curseur	a	copier depuis le curseur jusqu'à la fin de ligne	y$
revenir au mode commande	ESC	coller	p
supprimer un caractère	x	annuler la dernière opération	u
supprimer une ligne	dd	quitter en sauvegardant	:x
créer une ligne en dessous	o	quitter sans sauvegarder	:q!
créer une ligne au dessus	O	sauvegarder sans quitter	:w

Le statut de notre gestion de version

git status

```
Default (bash)                                                    ⌥⌘1
mo-ghra-leictreonach:git-test seb$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

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

no changes added to commit (use "git add" and/or "git commit -a")
mo-ghra-leictreonach:git-test seb$
```

Historique des commits


```
git log
```

Quelques options

-n <nombre>	Limite le nombre de commits affichés
--oneline	Affiche chaque commit sur une seule ligne
--decorate	Affiche les références (tags, branches..)
--stat	Affiche des fichiers et actions du commit
--graph	Affiche un graphe des branches

Options de recherche

--author="<nom>"	N'affiche que les commits d'un auteur spécifié
--grep="<text>"	Recherche et affiche les messages de commits qui contiennent <text>
--before="<date>" --after="<date>"	Recherche en fonction des dates de commit (--since et --until sont également possibles)
-- <fichier>	Affiche les commits qui concernent un fichier particulier
-S "<text>"	Affiche les commits qui ont ajoutés ou supprimés <text>

Exemples


```
git log -n 3
```

```
git log --oneline --graph --decorate
```

```
git log --since="2 weeks ago" --author="Author Name"
```

Enregistrer le résultat dans un fichier


```
git log > log.txt
```

Création ou remplacement du contenu

```
git log >> log.txt
```

Ajout à un fichier existant

Quand on s'est trompé


```
git commit --ammend -m "un message de commit"
```

- ✓ on veut faire une modification
- ✓ on a oublié un fichier
- ✓ on veut changer le message du commit

Ne fonctionne que sur le dernier commit

Exercice

- ✓ Créer un dossier my-webapp
- ✓ Dans ce dossier initialiser un projet Git
- ✓ Créer un dossier src et un fichier index.html avec un titre h1
- ✓ Faire un commit de ce fichier
- ✓ Ajouter un fichier styles.css avec un sélecteur pour body
- ✓ Faire un lien vers styles.css dans index.html
- ✓ Faire un nouveau commit avec ces deux fichiers
- ✓ Afficher le log des commits et l'enregistrer dans un fichier

Revenir en arrière sur un fichier


```
git restore [fichier]
```

restaure un fichier à sa version contenue dans l'historique, par défaut cet historique est de dernier commit

Attention, on perd alors toutes modifications non enregistrées dans un commit

Revient à la version d'un commit spécifique

```
git restore index.html
```

Revient à la version trois commits en arrière sur la branche master

```
git restore --source master~2 index.html
```

```
git restore --source e3213f25017c5354cbf35dcd47ca93ecc33610df index.html
```

Comparer


```
git diff [fichier]
```

Compare la version d'un fichier
dans le répertoire de travail
avec celle du dernier commit

```
git diff master
```

Compare la version de tous les fichiers
dans le répertoire de travail
avec celle du dernier commit
de la branche master

Comparaison
avec un commit particulier

```
git diff e3213f25017c5354cbf35dcd47ca93ecc33610df index.html
```

Enlever un fichier de la zone de transit


```
git rm --cached [fichier]
```

```
git rm --cached -r .
```

Vide toute la zone de transit

Revenir en arrière dans l'historique


```
git revert [commit]
```

restaure tous les fichiers à leur version contenue dans l'historique, indiquée par un numéro de commit

Cette opération effectue un nouveau commit placé en haut de la pile (HEAD)

En ligne de commande cette opération ouvre l'éditeur Vi pour nous permettre d'écrire le message de commit.

Exercice

- ✓ Ajouter un fichier nav.html au projet
- ✓ Le placer dans la zone de transit
- ✓ Le retirer de la zone de transit
- ✓ Faire un lien vers nav.html dans index.html
- ✓ Faire un commit "ajout de la navigation"
- ✓ Revenir en arrière (revert du dernier commit) et faire un git log --oneline
- ✓ Annuler le revert et faire un git log --oneline
- ✓ Comparer index html avec sa version dans le premier commit

Petit test

- ✓ Faire un git log --oneline sur le projet
- ✓ Noter les id des premier commit et dernier commit
- ✓ Faire un git checkout <id du premier commit>
  - ✓ You are in 'detached HEAD' state, ça doit faire mal
- ✓ Faire un git log --oneline sur le projet
  - ✓ On a perdu l'historique
- ✓ Faire un git checkout <id du dernier commit>
  - ✓ on retour l'historique
- ✓ Comparer cette opération avec la commande git revert

Git Récapitulatif

Action	Commande
Ajouter des fichiers pour préparer un commit	`git add [fichier]`
Retirer un fichier de la zone de transit	`git rm --cached [fichier]`
Réaliser un commit	`git commit --message="message de commit"`
Connaître l'état du projet git	`git status`
Lister l'ensemble des comités	`git log`
Revenir à la version de l'historique	`git restore [fichier]` `git restore .` (tous les fichiers)

Les branches

GIT Branch and its Operations.

Créer une branche


```
git branch [nom de la branche]
```

Attention
Un nom de branche est unique sur le projet
Un nom de branche ne peut contenir
ni espaces ni caractères spéciaux

Opération sur les branches

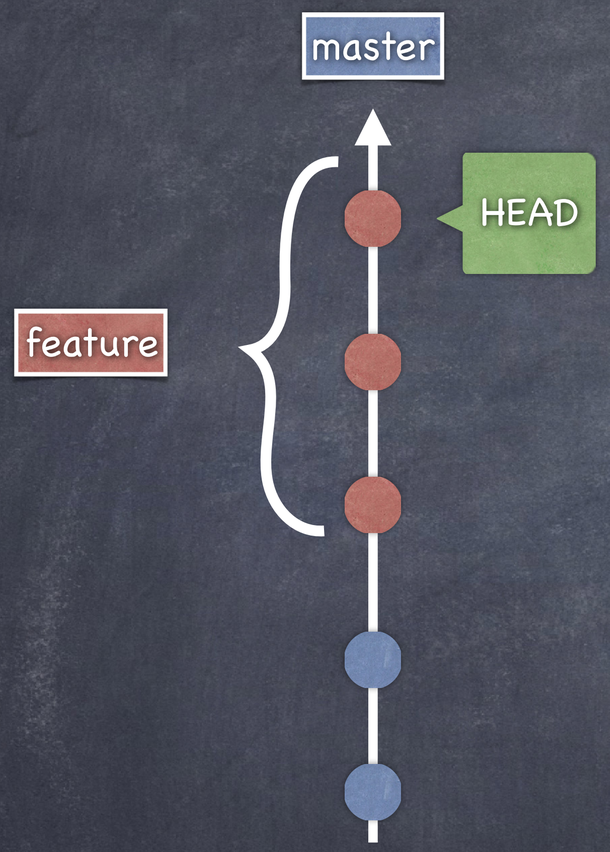
Action	Commande
Aller sur une branche	`git switch [branche]` `git checkout [branche]`
Supprimer une branche	`git branch --delete [branche]` `git branch -d [branche]`
Forcer la suppression d'une branche non fusionnée	`git branch -D [branche]`
Lister les branches d'un projet	`git branch`
Renommer une branche	`git branch --move [ancien nom] [nouveau nom]`
Copier une branche	`git branch --copy [source] [nouvelle branche]`
Fusionner deux branches (en étant sur la branche cible)	`git merge [branche source]`

Une astuce


```
git checkout -b [nom de la branche]
```

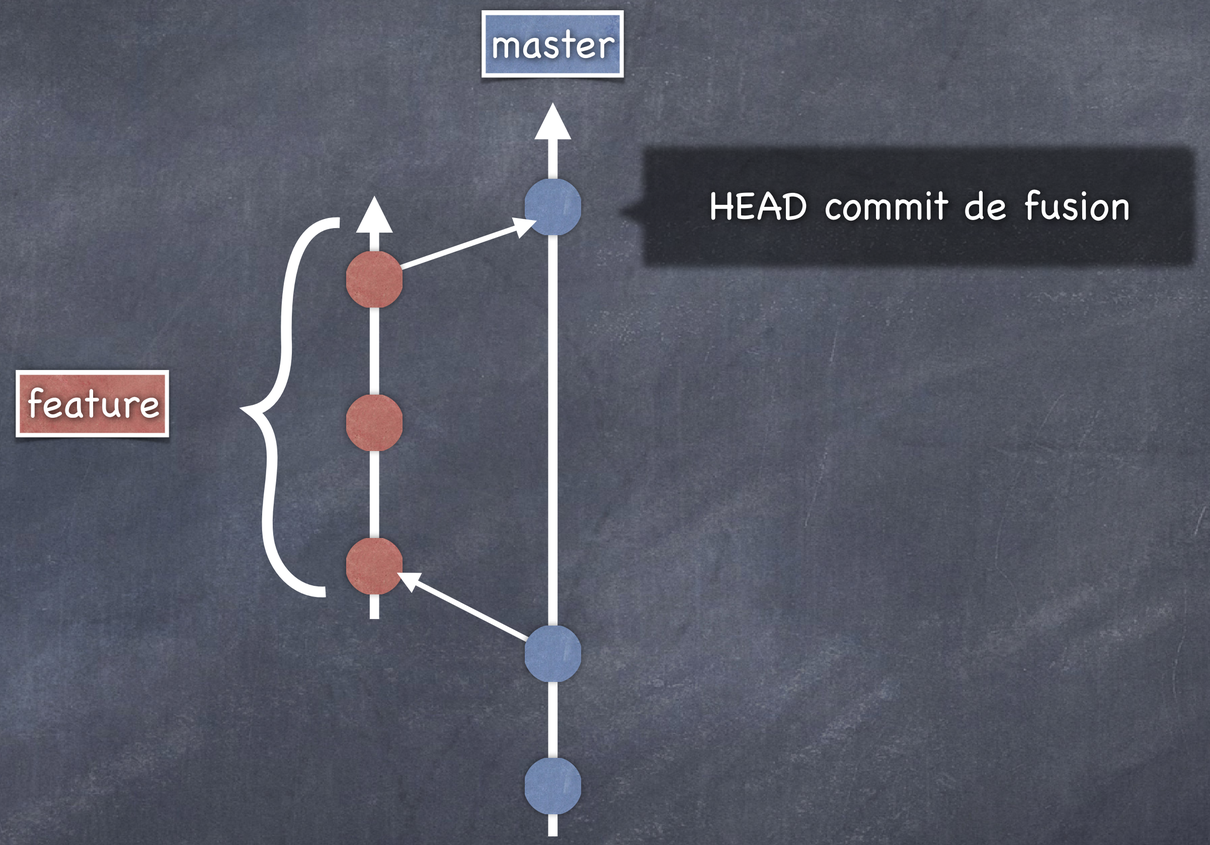
Combine git branch et switch
cette commande va créer une branche
et nous positionner sur celle-ci

Git merge



`git merge`

fast forward



`git merge --no-ff`

no fast forward

TP Branches

Sur le projet my-webapp

- ✓ Créer une branche nommée feature-searchbar et se déplacer sur cette branche
  - ✓ Créer une fichier src/search.js
  - ✓ Créer un lien vers ce fichier dans index.html
  - ✓ Faire un commit
- ✓ Se déplacer sur master et faire un merge feature-searchbar
- ✓ Constater le résultat avec git log et gitk dans un terminal

TP no FF

Sur le projet my-webapp

- ✓ Créer une branche nommée feature-gallery et se déplacer sur cette branche
  - ✓ Créer une fichier src/gallery.js
  - ✓ Créer un lien vers ce fichier dans index.html
  - ✓ Faire un commit
- ✓ Se déplacer sur master et faire un git merge --no-ff feature-searchbar
- ✓ Constater le résultat avec git log et gitk dans un terminal

TP création d'un conflit

Sur le projet my-webapp

- ✓ Sur la branche master modifier le titre h1 dans index.html et faire un commit
- ✓ Créer une branche new-title et se déplacer sur cette branche
  - ✓ Modifier le titre h1 dans index.html
  - ✓ Ajouter un titre h2 en dessous
  - ✓ Faire un commit
- ✓ Se déplacer sur master et faire un git merge new-title
- ✓ Constater le résultat (gitk dans un terminal)

Gestion des conflits


```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Accueil</title>
    <link rel="stylesheet" href="styles.css">
</head>

<body>
    <h1>Première page</h1>

    <p>Un ajout</p>

<<<<<<< HEAD
    <p>Texte sur la démarche</p>
=======
    <p>Barre de navigation</p>
>>>>>>> test

</body>

</html>
```

Git traçant les modifications lignes à lignes, les conflits sont plus rares

En cas de conflits, Git va intégrer les deux versions des lignes en conflit dans le fichier
Il faudra alors choisir ce que l'on conserve et supprimer le reste

Ensuite il faudra faire un nouveau commit avec ces modifications. Tant que le commit ne sera pas terminé ou qu'il restera des conflits Git restera en mode merge et on ne pourra changer de branche

Dès lors qu'il y a conflit, il n'est plus possible de faire un fast forward merge, il y aura forcément un commit de fusion sur la branche accueillante

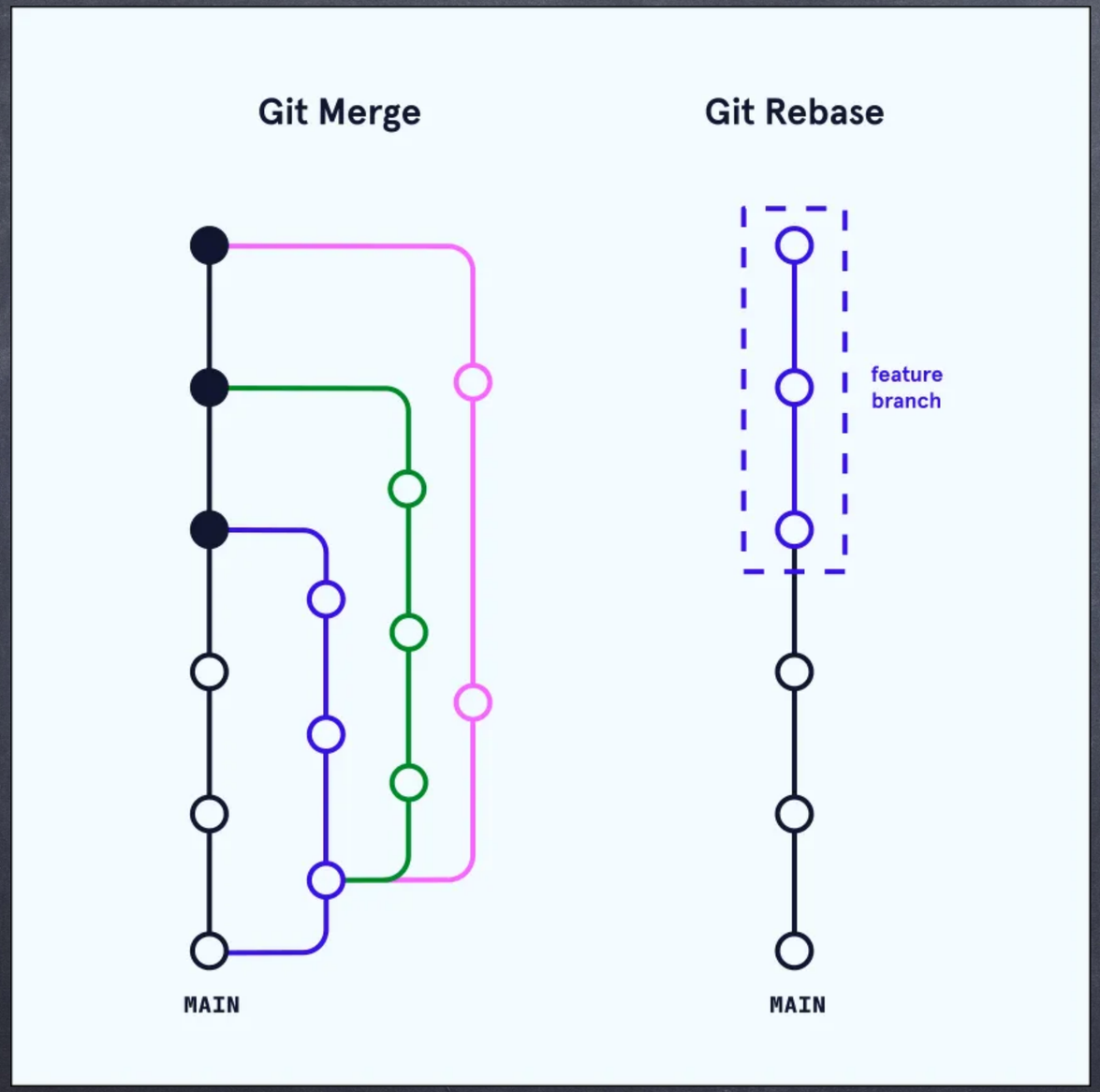
```
git merge --abort
```

annule la fusion
et sort du mode merge

A quoi servent les branches

- ✓ Travailler sur une nouvelle feature sans casser la branche master
- ✓ Faire un test et revenir à un projet propre
- ✓ Définir un point de sauvegarde pour un déploiement
- ✓ Marquer les différentes étapes d'un projet

Rebase




```
git rebase [nom de la branche]
```

Ressemble beaucoup à un fast forward merge

Principe

- ✓ Rejoue les commits d'une branche sur un autre branche
- ✓ Permet d'obtenir un historique plus propre en évitant les commits de fusion

TP

- ✓ Depuis la branche master créer une branche feature-1
- ✓ Dans cette nouvelle branche ajouter un fichier feature-1.txt et faire un commit
- ✓ Créer une nouvelle branche feature-2 avec un fichier feature-2.txt, faire un commit
- ✓ Vous travaillez sur feature-1 et vous voulez changer la base de votre branche (git rebase feature-2), constater le résultat
- ✓ Modifier la branche master en ajoutant un fichier bug-fix-1.txt, faire un commit
- ✓ git rebase master sur la branche feature-1

Quand utiliser rebase

- ✓ Vous avez créé une branche mais vous vous êtes trompé, vous voulez baser cette nouvelle branche sur une autre branche
- ✓ Vous travaillez sur une branche et vous voulez intégrer les modifications d'une autre branche sans faire de fusion
- ✓ Vous travaillez sur une branche depuis longtemps et vous voulez résoudre les conflits sur cette branche avant de faire un merge fast forward sur la branche principale afin de conserver un historique propre sur cette dernière

Quand ne surtout pas utiliser rebase

Dès lors que vous êtes sur une branche partagée par d'autres développeurs, rebase change les id des commits

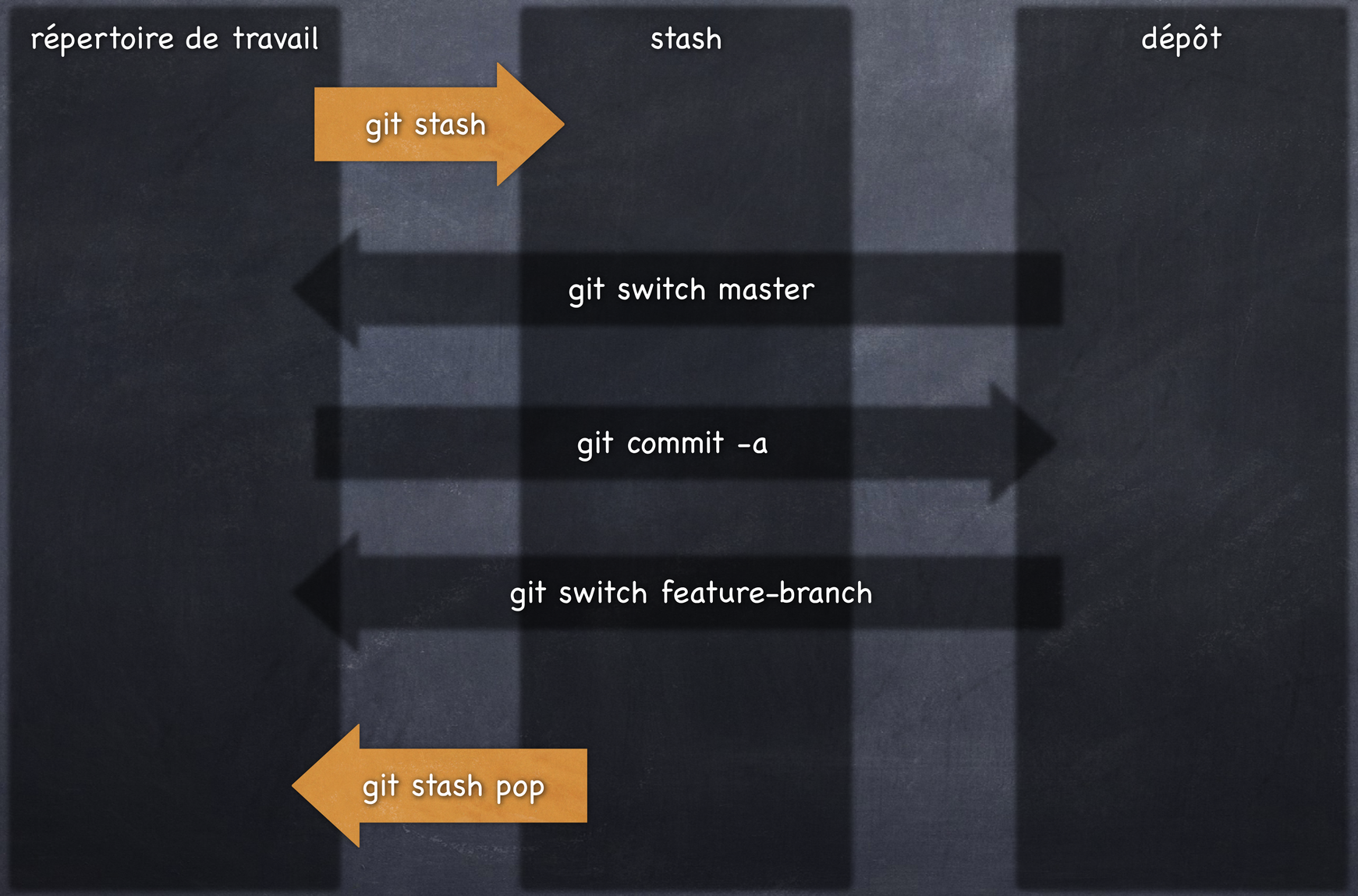
pas de rebase
sur une branche distante

Remisage (stashing)

On met de côté des modifications quand on doit intervenir sur une autre branche alors que l'on est pas encore prêt à faire un commit

On peut ensuite récupérer les modifications mises de côté et continuer à travailler

Git stash workflow



Commandes

Action	Commande
Mettre de côté	`git stash ou git stash save [nom]` `-u pour également ajouter les fichiers non encore tracés`
Voir la liste des remisages	`git stash list`
Récupérer les derniers éléments remisés (en supprimant le remisage)	`git stash pop [nom ou numéro de remisage]`
Récupérer les derniers éléments remisés (en conservant le remisage)	`git stash apply [nom ou numéro de remisage]`
Supprimer un remisage	`git stash drop [nom ou numéro de remisage]`
Supprimer tous les remisages	`git stash clear`

Exemple

- ✓ Je suis sur une branche et je dois intervenir en urgence sur master, je met de côté mes modifications avec `git stash -u "message"`
- ✓ Je fais ce que j'ai à faire sur master et je retourne à ma branche de travail
- ✓ Je vérifie l'état de mon contenu mis de côté avec `git stash list`
- ✓ Je vois que je n'ai qu'un stash dans la liste, je récupère ce que j'ai mis de côté avec `git stash pop stash@{0}` (ou simplement `git stash pop` vu que je n'ai qu'un seul stash)

Les tags

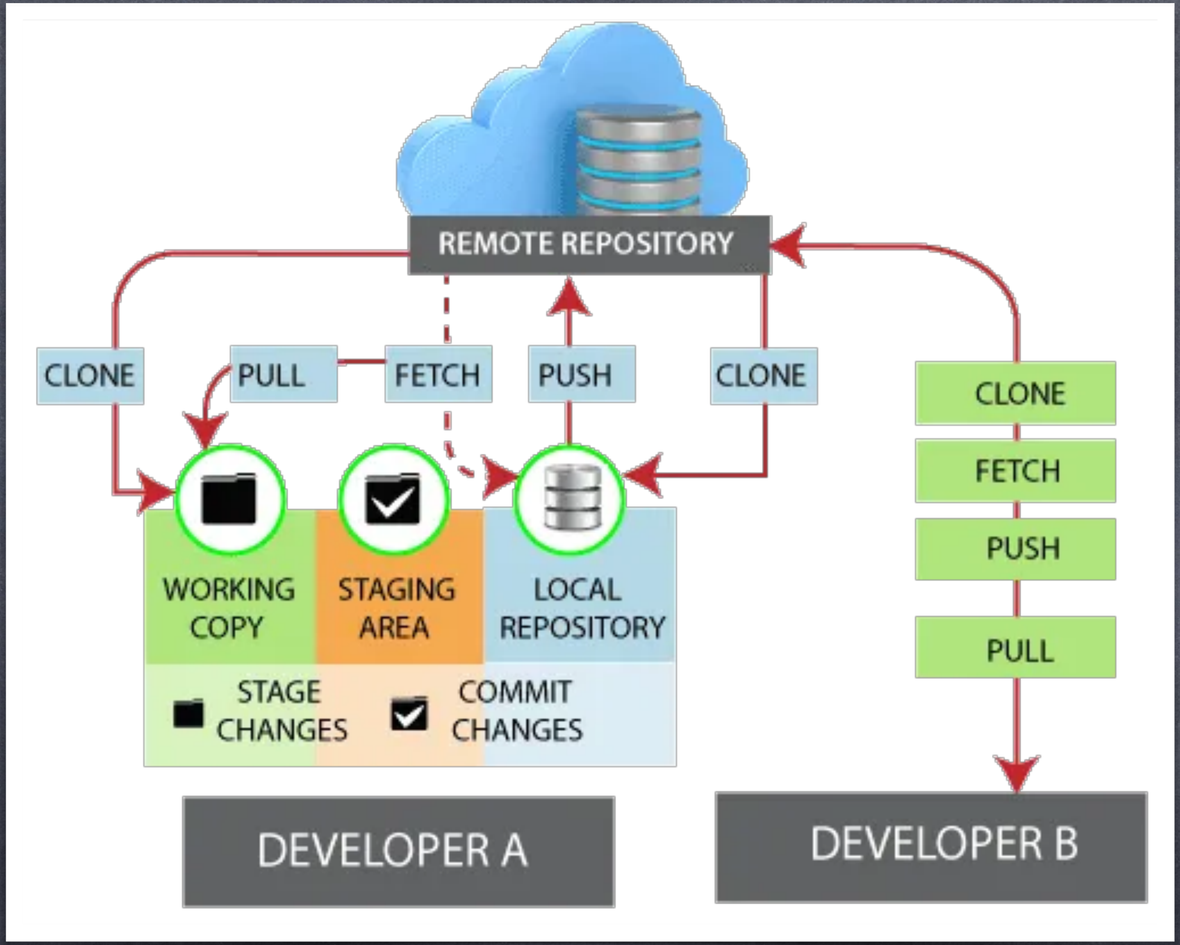
Un moyen pour donner un nom à une position dans l'historique

Attention
Un nom de tag et de branche est unique sur le projet

Un nom de tag ne peut contenir ni espaces ni caractères spéciaux

`git tag v1`	associe le tag "v1" au commit courant
`git switch v1`	Positionne l'historique au tag "v1"
`git tag`	Liste tous les tags
`git tag -d v1`	Supprime le tag "v1"
`git tag v1 [commit]`	associe le tag "v1" à un numéro de commit spécifique

Travailler avec un dépôt distant


REMOTE REPOSITORY
CLONE
PULL
FETCH
PUSH
CLONE
CLONE
FETCH
PUSH
PULL
WORKING COPY
STAGING AREA
LOCAL REPOSITORY
STAGE CHANGES
COMMIT CHANGES
DEVELOPER A
DEVELOPER B

Principes

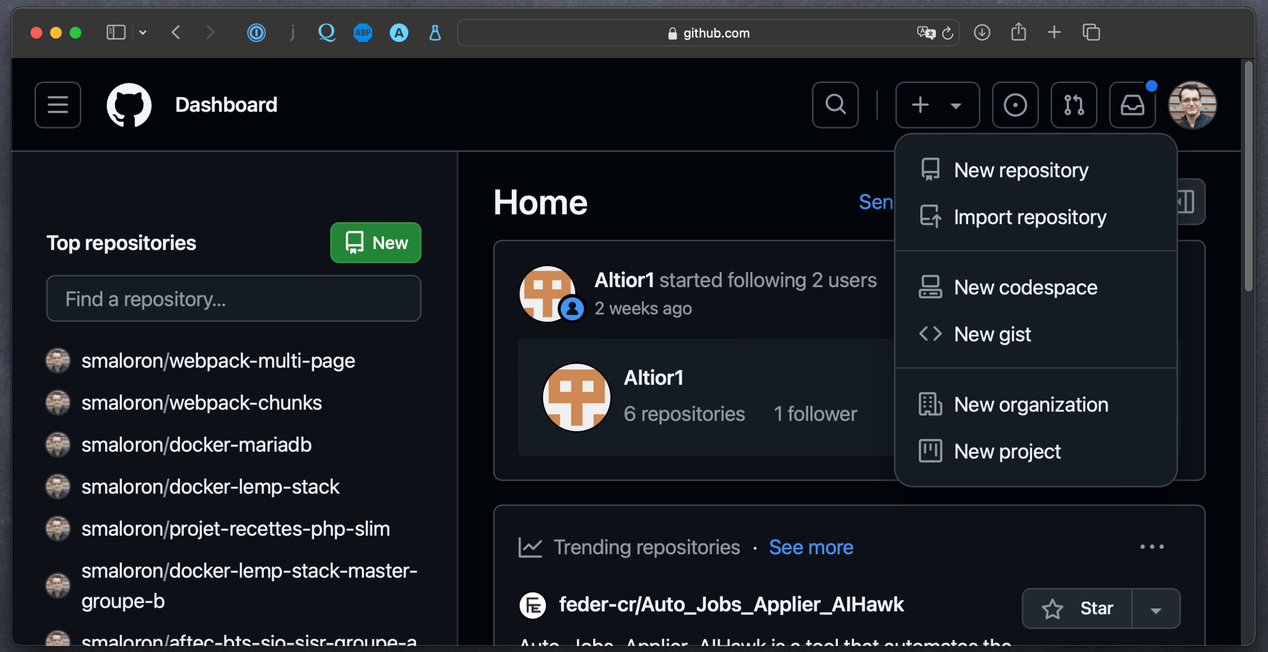
- ✓ Le dépôt distant possède un historique, des branches et des tags comme un dépôt local
- ✓ Chaque développeur possède dans son dépôt local une copie du projet plus ou moins synchronisée avec le dépôt distant

Les service de dépôt distant

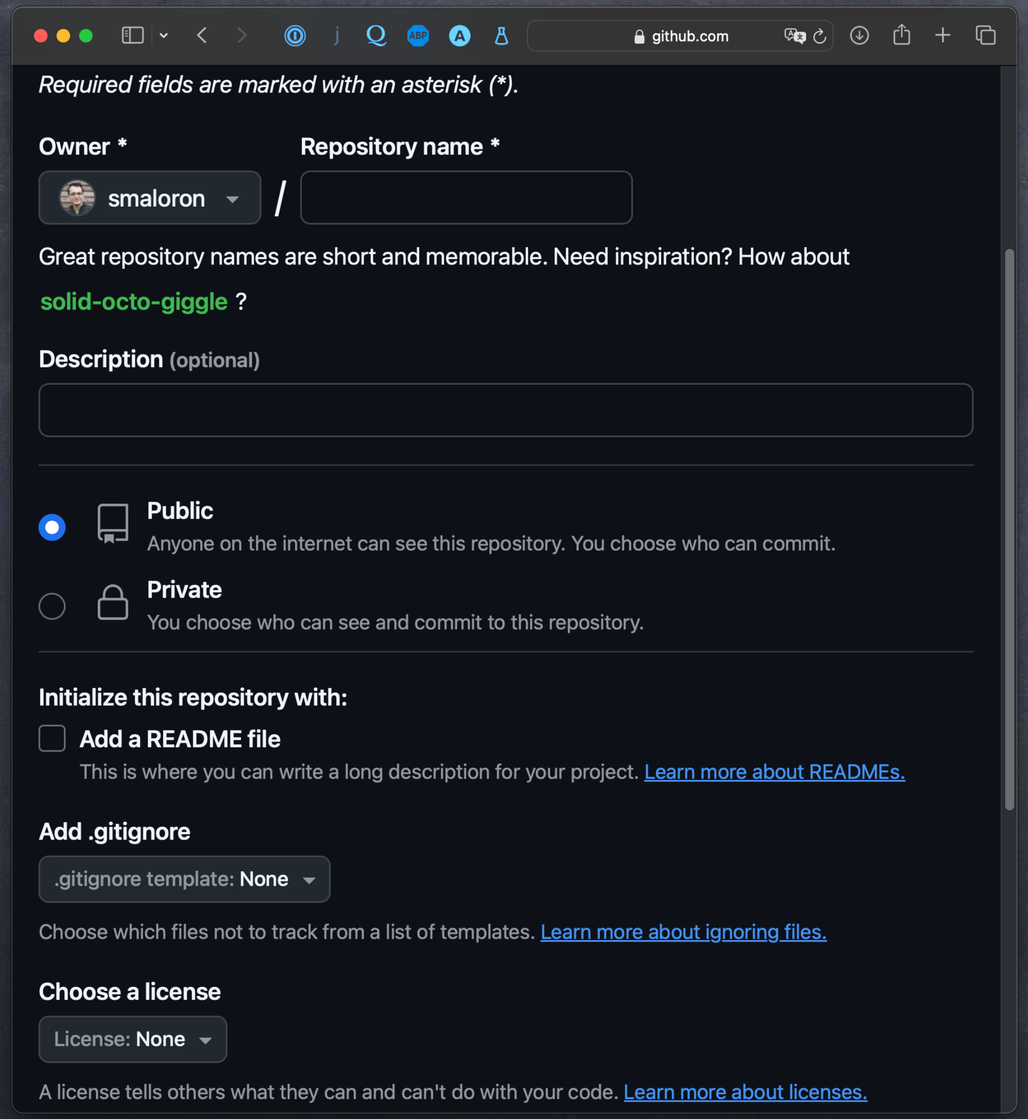
- ✓ Github
- ✓ GitLab
- ✓ Bitbucket
- ✓ OneDev
- ✓ Gogs
  (auto hébergement)
- ✓ Codeberg
  (pour projets open source uniquement)

Github

- ✓ Créer un compte sur Github
- ✓ Créer un dépôt distant

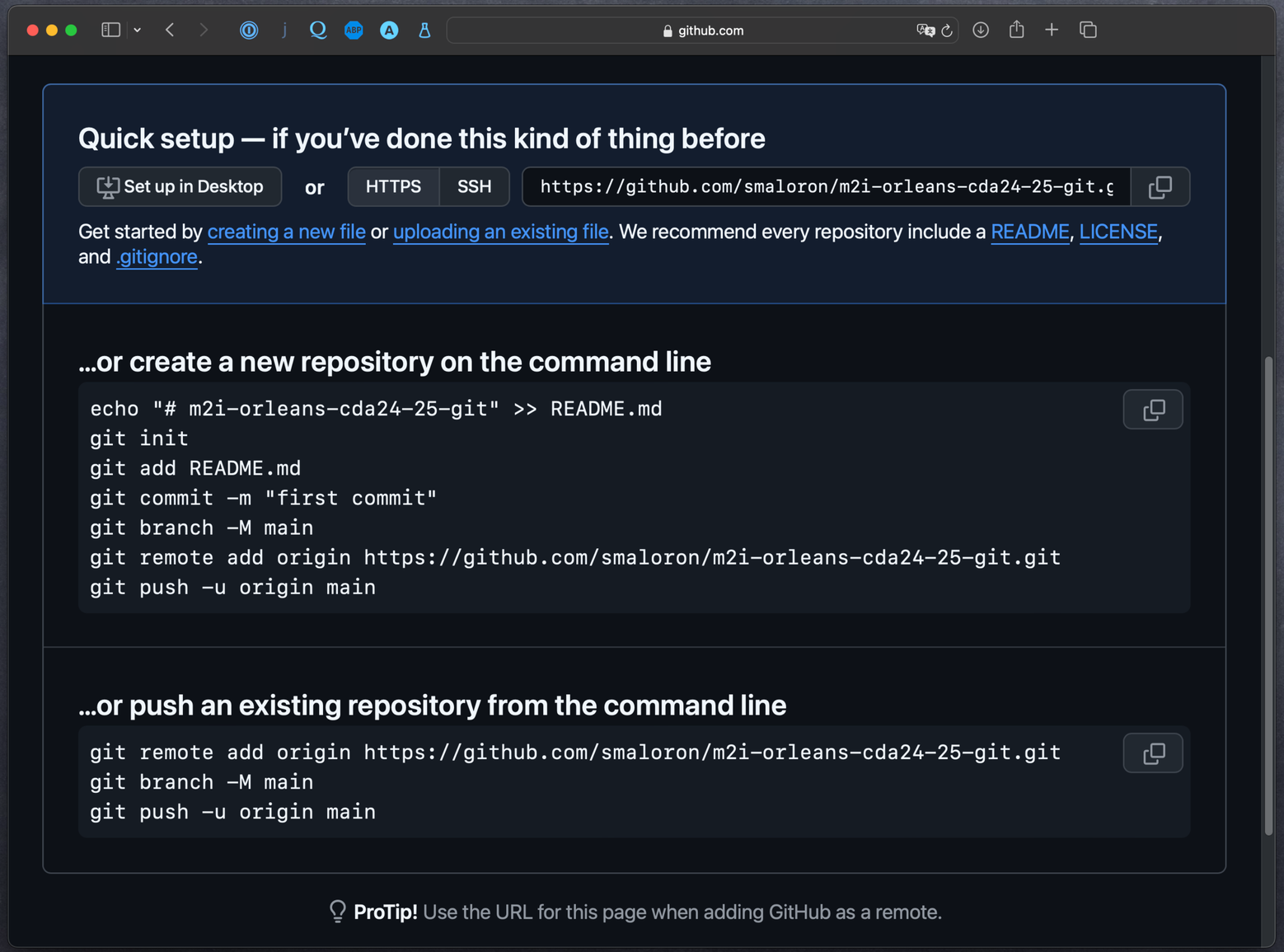


Dépôt Github



Attention
Ne pas ajouter de fichier
README et .gitignore

Fin de la création du dépôt



git branch -M

change le nom
de la branche en cours

Sur Github, la branche principale par défaut s'appelle "main"

git push -u

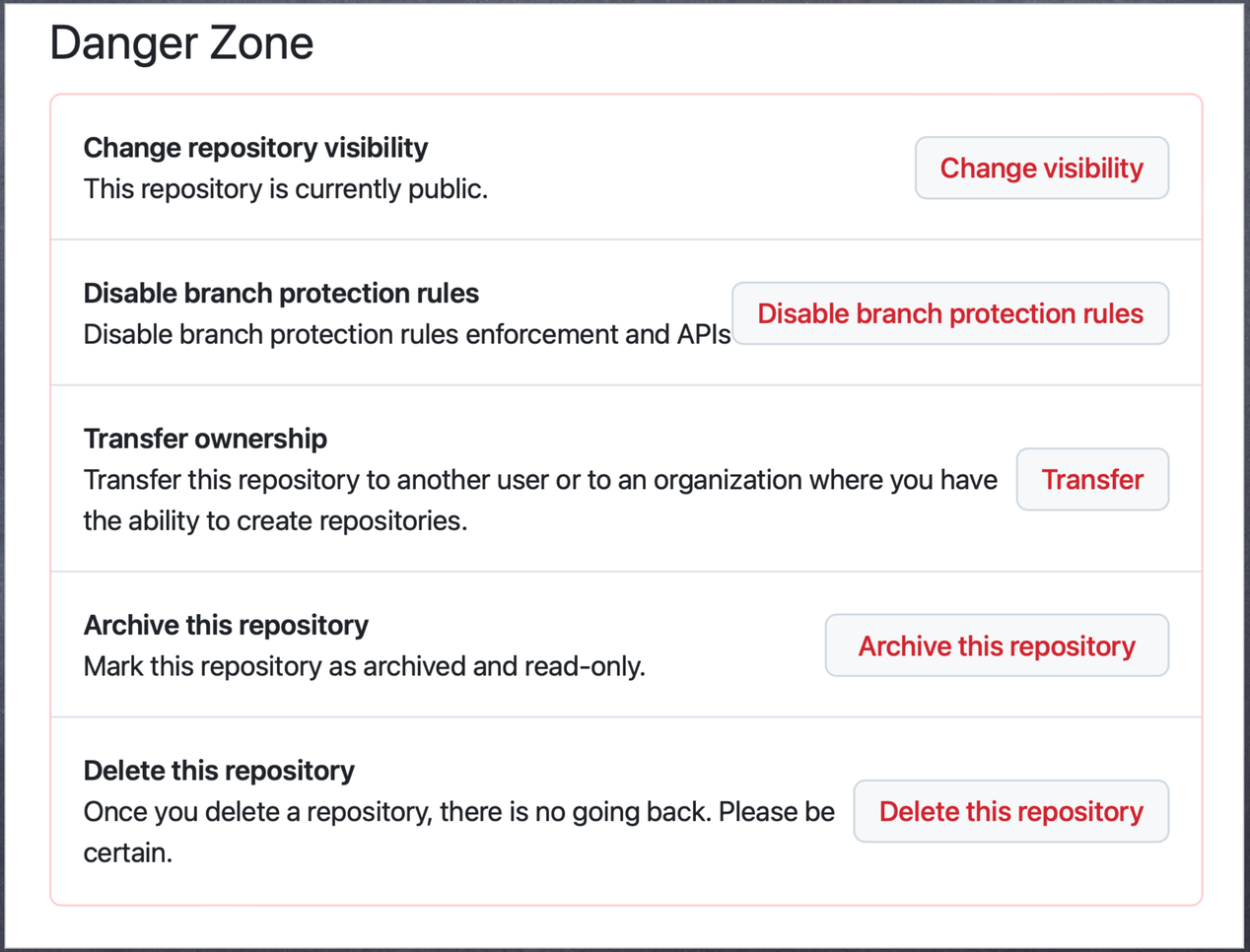
déclare que la branche distante sera "upstream", c'est à dire la branche distante par défaut pour cette branche locale.

Permet donc de lier une branche distante avec une branche locale

Supprimer un dépôt Github

Aller dans `settings`, onglet `general`

Descendre tout en bas de la page



A vous

- ✓ Faites quelques commits sur votre projet local
- ✓ Faites un `push -u origin main` pour envoyer ces commits sur le dépôt distant
- ✓ Constater le résultat sur Github

Félicitations vous venez de créer votre premier dépôt sur Github

Syntaxe de push

nom du remote

push de la branche locale active

```
git push -u origin main
```

nom de la branche distante

push d'une branche locale

```
git push -u origin main:prod
```

nom de la branche distante

nom de la branche locale

```
git push origin :prod
```

supprime la branche distante

Syntaxe de push simplifié


```
git push
```

conditions

Il n'y a qu'un seul remote sur le projet
La branche poussé est upstream sur le dépôt distant

Syntaxe de push sur les tags

nom du remote

```
git push origin tag
```

nom du tag

nom du remote

```
git push origin --tags
```

envoie tous les tags locaux sur le remote

Git Récupérer un dépôt distant en local

```
git clone adresse-du-depot [nom du répertoire cible]
```

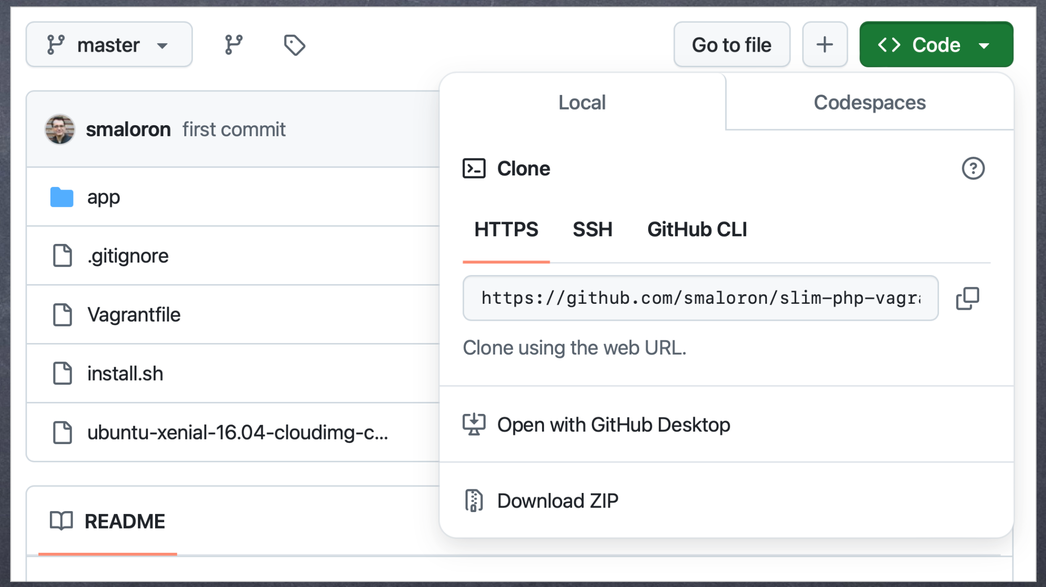
https://github.com/nom-du-compte/nom-du-projet

Aller sur Github et visualiser le dépôt d'un de vos camarade

Récupérer l'adresse de ce dépôt

Déplacez-vous dans le répertoire de vos projets et faites un clone

Faites un git remote pour constater que le projet local est lié au dépôt distant



Mettre à jour le dépôt local


```
git pull [remote] [branche distante]
```

Récupère les dernières mises à jour du dépôt distant sur la France active dans le répertoire de travail

1. Demandez à votre voisin de faire une modification sur son projet et de faire un `git push` pour modifier son dépôt distant
2. Faites de même sur votre propre projet pour que votre voisin puisse tester
3. Faites un `git pull` et constater le résultat sur votre code en local

Git pull avec commit de fusion

Récupère les dernières mises à jour du dépôt distant sur la France active dans le répertoire de travail

1. Demandez à votre voisin de faire une modification sur son projet (dans la branche main) et de faire un `git push` pour modifier son dépôt distant

2. Faites de même sur votre propre projet pour que votre voisin puisse tester

3. Sur le projet cloné de votre voisin faites vous même une modification suivie d'un commit (votre historique et celui du dépôt distant seront donc différents)

4. Faites un `git pull` et constater le résultat sur votre code en local

Récupérer les nouvelles branches distantes


```
git fetch [nom du remote]
```

Récupère les branches du dépôt distant
et les place dans le dépôt local (remote/branche)

1. Demandez à votre voisin de faire une nouvelle branche et de la pousser sur le dépôt distant

2. Faites de même sur votre propre projet pour que votre voisin puisse tester

3. Faites un `git fetch` suivi d'un `git branch` pour voir la nouvelle branche et d'un `git switch nom-de-la branche` pour récupérer cette branche dans votre répertoire de travail

Débat

Quelles sont les différences entre pull et fetch ?

Alternative à pull

Permet d'avoir un historique plus propre

1. Demandez à votre voisin de faire une modification sur sa branche main et de la pousser sur le dépôt distant
2. Faites de même sur votre propre projet pour que votre voisin puisse tester
3. Faites un `git fetch` suivi d'un `git rebase origin/main` en étant positionné sur votre branche main en local

Mise à jour avant push

S'il y a eu des changement sur le dépôt distant depuis la dernière mise à jour sur le dépôt local il est impossible de faire un push, il faut avant cela récupérer les mise à jour :

- ✓ avec un pull
- ✓ avec un fetch suivi d'un rebase

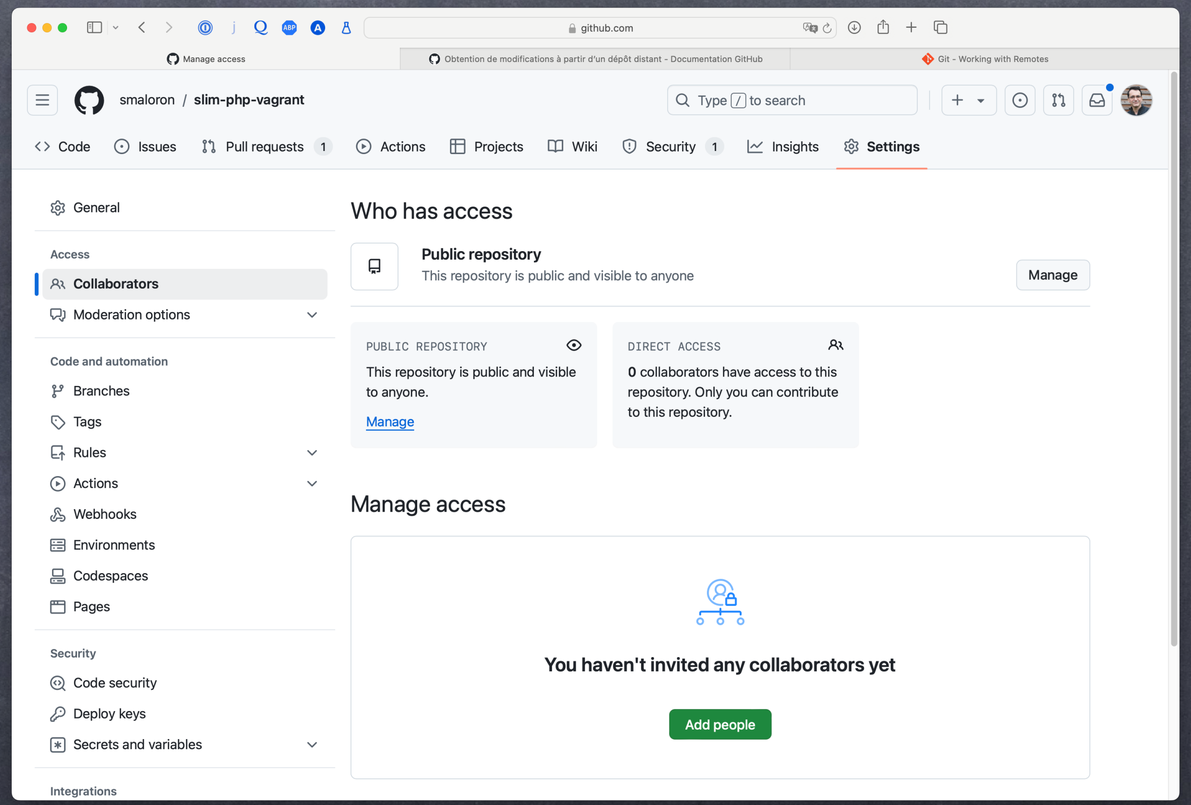
Test

Essayez de faire un commit sur un projet cloné et de faire ensuite un push

Que ce passe-t-il ?

Git Inviter des collaborateurs

Le propriétaire du dépôt doit inviter des développeurs pour que ceux-ci aient le droit de faire des push sur son projet



Invitez-vous

Récupère les branches du dépôt distant
et les place dans le dépôt local

1. Invitez vos collègues (avec le nom de leurs comptes Github)
2. Faites vous inviter par vous collègues
3. Cloner les dépôts distants et faites des push et des pull

Toujours faire un pull avant un push

Gestion des remotes

ajout d'un remote

```
git remote add nom-remote adresse
```

adresse du dépôt sur github

```
git remote add origin https://github.com/smaloron/test.git
```

nom du remote en local, par convention
le premier remote est nommé origin

liste des remotes
-v affiche l'url

```
git remote -v
```

renomme un remote

```
git remote rename old-name new-name
```

supprime un remote

```
git remote remove name
```

Gitignore

Certains fichier ou répertoires n'ont pas vocation à être versionnés,
d'autres ne doivent surtout pas apparaitre dans un dépôt distant surtout s'il est public

- ✓ Le répertoires des dépendances
  (node_modules par exemple)
- ✓ Les fichiers système de votre OS
  (.DS_Store, desktop.ini, Thumbs.db...)
- ✓ Les fichiers contenant des informations sensibles
- ✓ Le dossier de configuration provenant de votre IDE
- ✓ Le répertoire de build
- ✓ Les logs et les fichiers temporaires
- ✓ Les fichiers compilés et les archives

Exemple de fichier


```
# System files
.DS_Store
Thumbs.db

# IDE files
.vscode/
.idea/
*.sublime-*

# Dependencies
node_modules/
vendor/
__pycache__/
*.py[cod]

# Build outputs
dist/
build/
bin/
target/

# Environment files
.env
.env.local

# Logs
*.log
```

Enregistrer ce fichier
à la racine du projet
et le nommer `.gitignore`

Le caractère # indique un commentaire

Le caractère * remplace n'importe quelle suite de caractères

Les fichiers et répertoires de .gitignore
ne seront pas tracés par Git

Outil pour générer le contenu de .gitignore
https://www.toptal.com/developers/gitignore/

Mini projet

L'un d'entre vous créer un dépôt distant sur Github

Il invite ses camarades à collaborer sur le projet

Il créé la page index.html et un fichier styles.css qu'il pousse sur le dépôt distant

Chacun réalise une page html, il peut ajouter des images et du css spécifique, il doit utiliser le fichier css général définit par le créateur du projet et ajouter un lien vers sa page dans index.html. Une fois elle travail terminé, chacun pousse sa modification sur Github

Vous pouvez et devez vous concerter

Connection avec SSH

- ✓ Plus sécurisée que le mot de passe
- ✓ Plus simple
- ✓ Facilite l'automatisation (pas besoin de saisie de mot de passe)
- ✓ Accès multi comptes
- ✓ Ne passe pas par HTTPS

Création d'une clef SSH

1. Création de la clef

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

2. Ajout de l'identité à ssh

```
eval "$(ssh-agent -s)" ssh-add ~/.ssh/id_ed25519
```

nom de la clef

3. Copie de la clef ssh

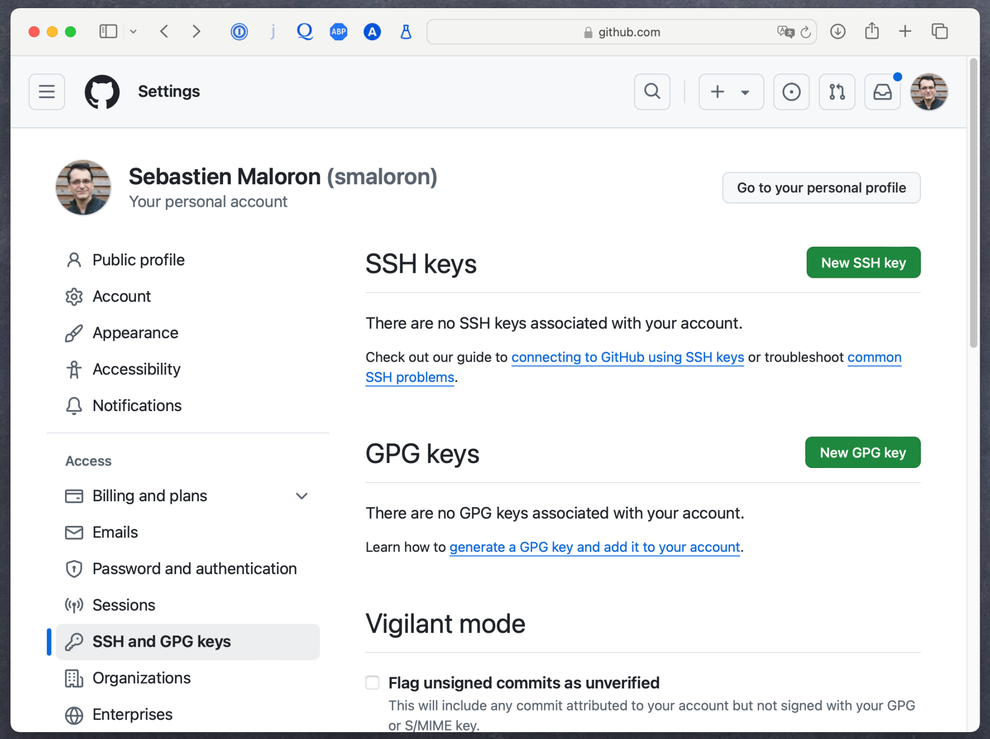
```
pbcopy < ~/.ssh/id_ed25519.pub  # macOS
xclip -sel clip < ~/.ssh/id_ed25519.pub  # Linux
clip < ~/.ssh/id_ed25519.pub  # Windows (Git Bash)
```

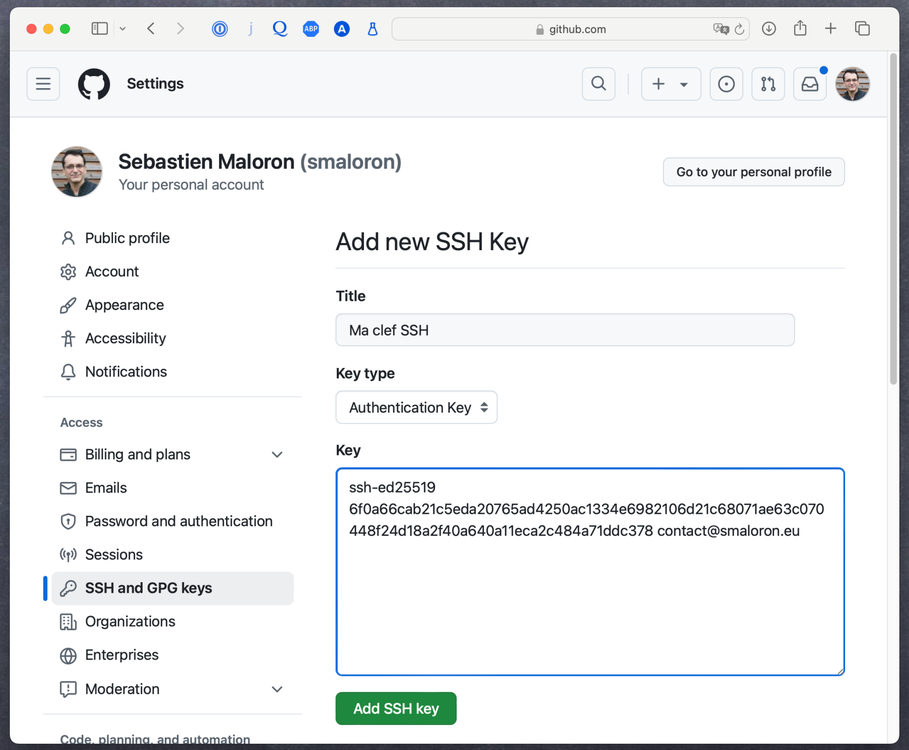
```
cat ~/.ssh/id_ed25519.pub
```

alternative

Ajout d'une clef SSH sur Github

https://github.com/settings/keys





Test de la connexion

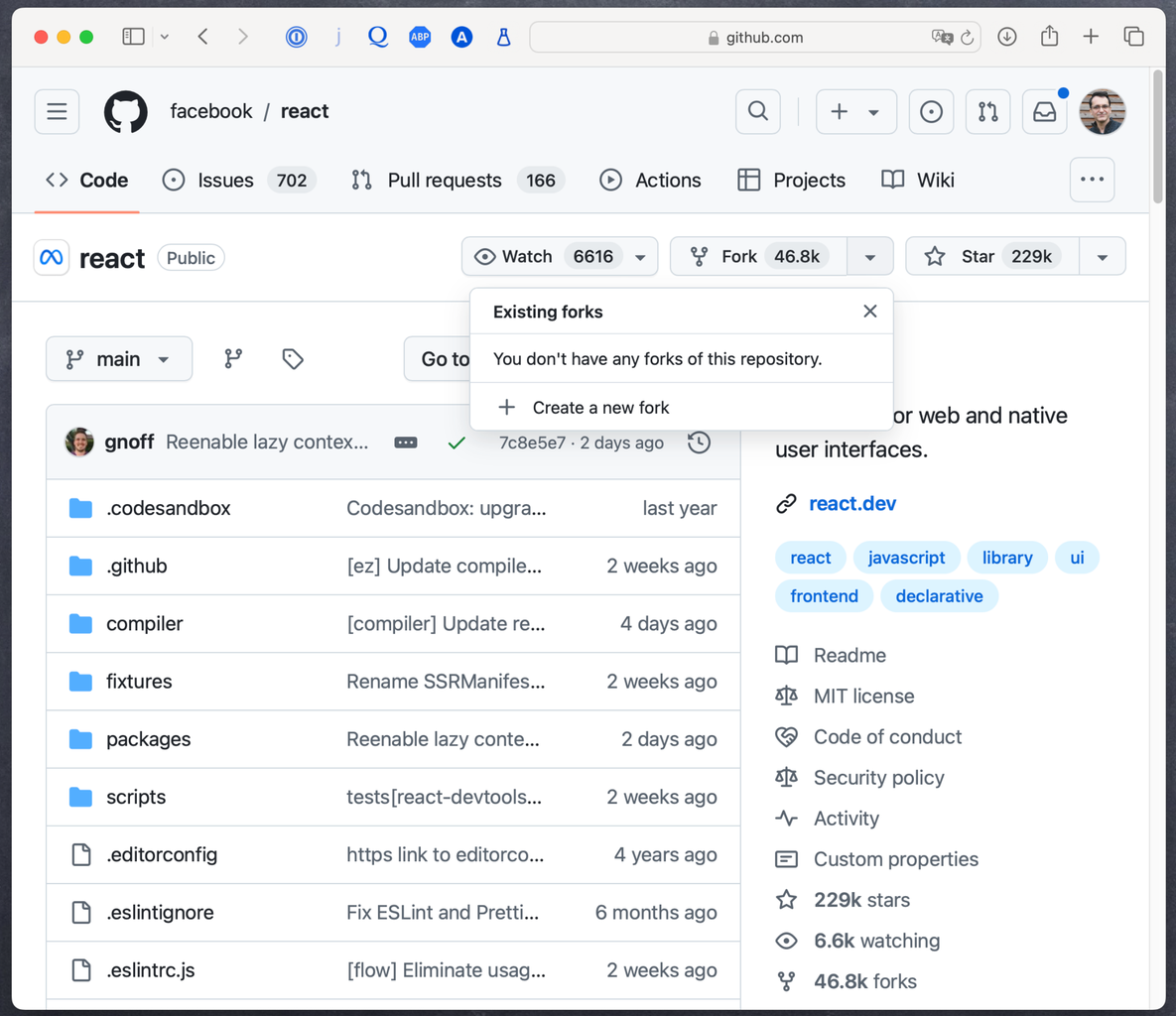
```
ssh -T git@github.com
```

A la première connexion il faut ajouter Github
à la liste des known hosts (répondre yes à la question)

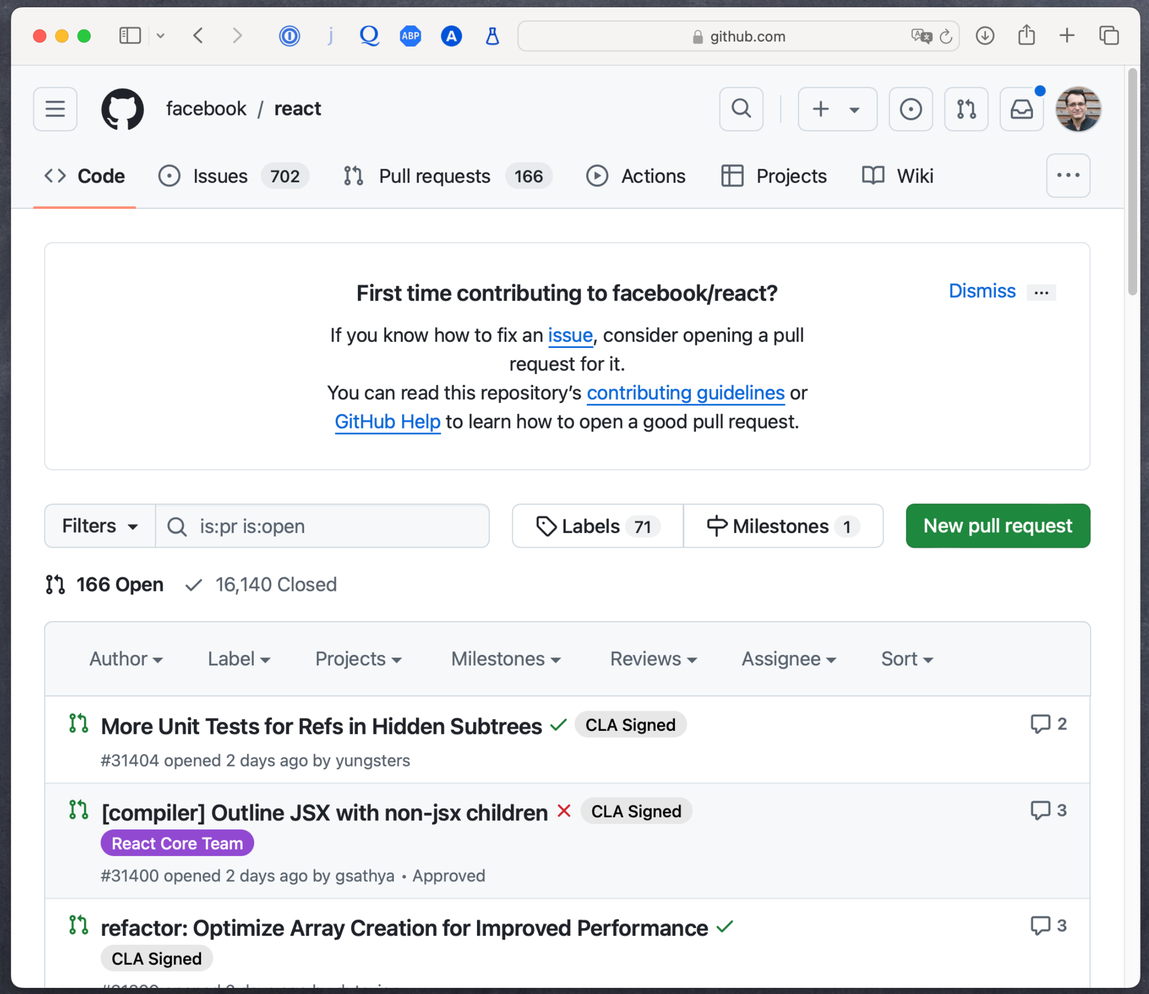
Collaborer à un projet en asynchrone

- ✓ Faire un fork d'un projet sur Github
- ✓ Faire un clone en local de votre fork
- ✓ Faire une modification sur une branche et la pousser sur votre fork
- ✓ Aller sur le projet d'origine sur Github et faire un Pull Request
  indiquer la branche source (chez vous)
  et la branche cible (sur le repo d'origine)

Faire un fork



Faire un Pull Request



Exercice

- ✓ Faites un fork du projet d'un camarade
- ✓ Faites une modification et un pull request
- ✓ Votre camarade pourra revoir le pull request et l'accepter

Git workflow

Trunk based

- ✓ Une branche principale unique et stable :

  Tout le développement converge vers une seule branche principale (main ou trunk), qui est toujours dans un état déployable. Les développeurs intègrent leurs modifications directement dans cette branche.

- ✓ Branches de courte durée (si utilisées) :

  Si des branches de fonctionnalité sont créées, elles sont très courtes (généralement quelques heures ou jours maximum) et sont fusionnées dans la branche principale dès qu'elles sont prêtes.

  L'objectif est de minimiser le temps de divergence de la branche principale et de réduire les conflits de fusion.

- ✓ Intégration fréquente :

  Les développeurs fusionnent leurs changements plusieurs fois par jour dans main, ce qui permet de détecter rapidement les conflits ou les problèmes de compatibilité et de les résoudre sans attendre.

Trunk based

- ✓ Feature Flags pour les fonctionnalités non finalisées :

  Pour les fonctionnalités nécessitant plusieurs jours de développement, les équipes utilisent des "feature flags" (drapeaux de fonctionnalité) pour activer ou désactiver la fonctionnalité en production sans devoir la retirer du code.

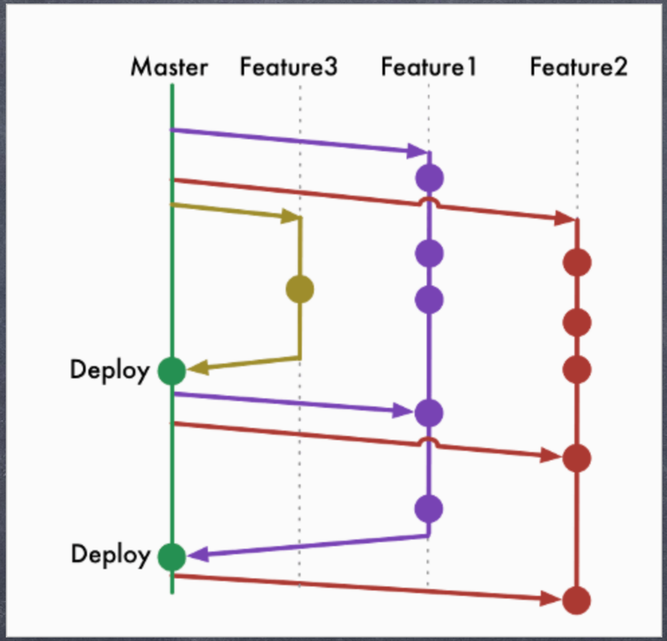
  Cela permet de fusionner du code non finalisé dans main sans le rendre visible aux utilisateurs finaux.

- ✓ Automatisation des tests et CI/CD :

  L'automatisation des tests est cruciale pour garantir que chaque changement fusionné dans main est stable et prêt pour le déploiement.

  Les pipelines CI/CD automatisés permettent de tester et de déployer chaque modification rapidement et de manière fiable.

Trunk Based



Avantages

- ✓ Conflits de fusion réduits.
- ✓ Déploiements rapides et continus.
- ✓ Simplicité et transparence.
- ✓ Feedback rapide.

Inconvénients

- ✓ Nécessite une discipline rigoureuse.
- ✓ Plus vulnérable aux bugs, erreurs et catastrophes.
- ✓ Gestion complexe des fonctionnalités longues.
- ✓ Forte dépendance aux tests automatisés pour assurer la stabilité de la branche main.
- ✓ Peu adapté aux grandes équipes sans outils devops très robustes
- ✓ Les feature flags (maintenance, tests, migration BD)

Github flow

- ✓ Une branche principale stable :

  La branche main (ou master) est toujours dans un état déployable. Toute modification effectuée sur cette branche doit être prête pour la production.

- ✓ Branches de fonctionnalités courtes :

  Chaque nouvelle fonctionnalité, correctif ou amélioration est développée dans une branche de courte durée créée à partir de main.

  Ces branches sont généralement nommées en fonction de la tâche ou de la fonctionnalité (feature/nouvelle-fonctionnalite ou fix/correctif-bug).

- ✓ Pull Requests (PR) et révisions :

  Une fois le travail terminé sur une branche de fonctionnalité, le développeur ouvre une Pull Request (PR) vers main.

  Les PR permettent aux autres membres de l'équipe de réviser le code, de commenter, et de s'assurer que la fonctionnalité est prête pour la fusion.

Github flow

- ✓ Tests et validation avant la fusion :

  Avant de fusionner la PR dans main, le code doit passer les tests automatisés et répondre aux critères de qualité.

  GitHub Flow encourage l'utilisation de pipelines CI/CD pour exécuter automatiquement des tests sur chaque PR.

- ✓ Fusion et déploiement :

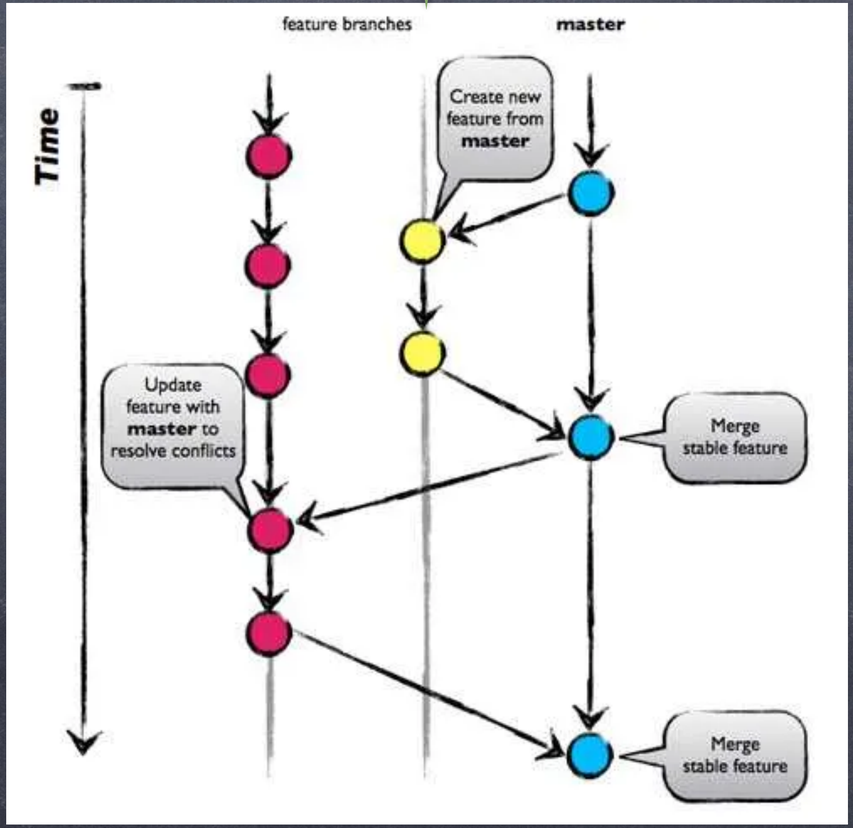
  Une fois la PR approuvée, la branche est fusionnée dans main.

  Dans un workflow CI/CD complet, une fusion dans main déclenche automatiquement un déploiement en production ou sur un environnement intermédiaire (staging) pour validation finale.

Github flow

features

Pull Request



Avantages

- ✓ Simplicité et agilité.
- ✓ Automatisation et CI/CD sur une seule branche.
- ✓ Formalisation de la validation avec le Pull Request.
- ✓ Adapté aux projets open source et aux équipes hétérogènes

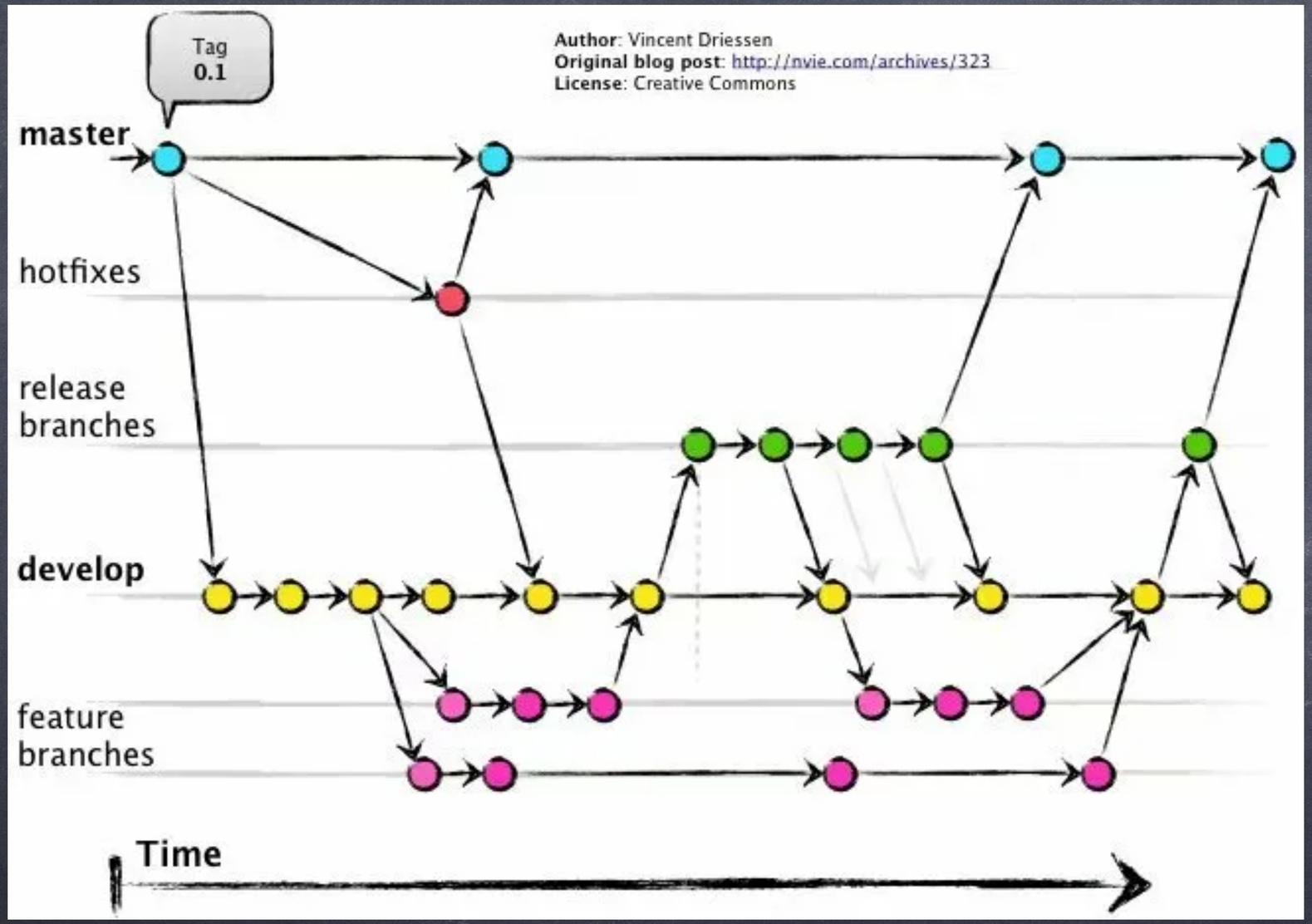
Inconvénients

- ✓ Pas adapté si on doit maintenir différentes versions du code en production.
- ✓ Plus vulnérable aux bugs, erreurs et catastrophes.
- ✓ Difficulté avec les grosses features qui restent longtemps sans être fusionnées.
- ✓ Forte dépendance aux tests automatisés pour assurer la stabilité de la branche main

Git flow

- ✓ main : branche principale qui contient le code prêt pour le déploiement
- ✓ develop : branche contenant les dernières modifications prêtes pour la prochaine version. Les nouvelles fonctionnalités et les correctifs y sont fusionnés avant de passer en production.
- ✓ Branches de support :
  - ✓ Branches de fonctionnalité (feature/*) : Pour travailler sur de nouvelles fonctionnalités.
  - ✓ Branches de version (release/*) : Pour préparer une nouvelle version en production.
  - ✓ Branches de correctif (hotfix/*) : Pour les correctifs urgents en production.

Git flow


Tag
0.1
Author: Vincent Driessen
Original blog post: http://nvie.com/archives/323
License: Creative Commons
master
hotfixes
release
branches
develop
feature
branches
Time

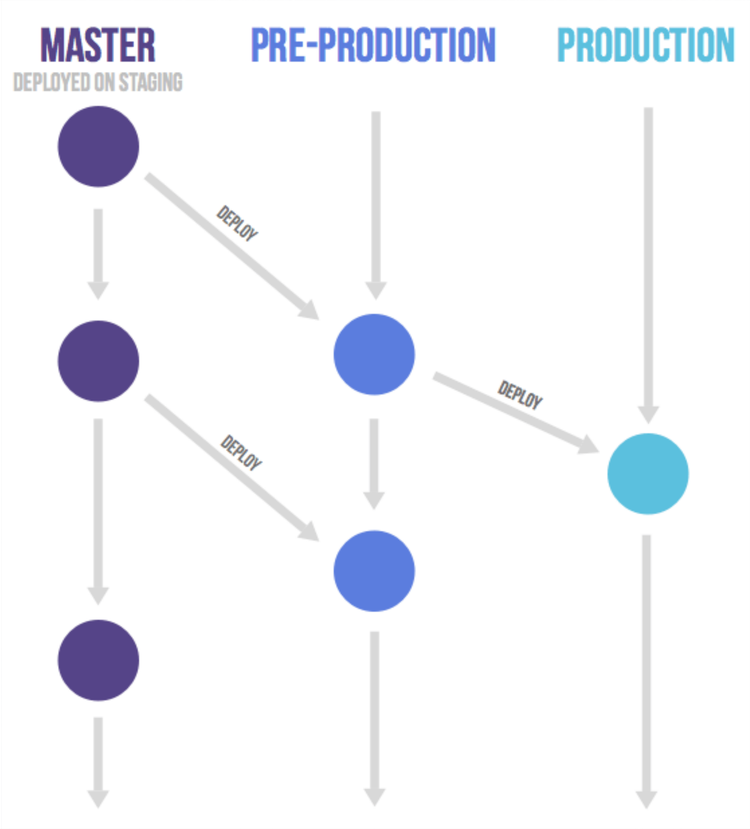
Avantages

- ✓ Structure claire : Facilite le suivi du statut des développements et des branches de fonctionnalités.
- ✓ Développement isolé : Garde les fonctionnalités et les correctifs séparés jusqu'à ce qu'ils soient stables et prêts à être fusionnés.
- ✓ Processus de version : Permet un processus structuré avec des tests approfondis.
- ✓ Support des correctifs : Permet des corrections rapides en production sans perturber le développement en cours.

Inconvénients

- ✓ Complexité excessive pour les petits projets ou les équipes agiles
- ✓ Mise en production lente
- ✓ Complique le pipeline CI/CD (intégration et déploiement continu)
- ✓ Risque de divergence entre main et develop et donc fusion délicate
- ✓ Correctifs appliqués sur les branches main et develop (double travail de fusion)
- ✓ Historique peu lisible

Gitlab Flow


MASTER
DEPLOYED ON STAGING
PRE-PRODUCTION
PRODUCTION
DEPLOY
DEPLOY
DEPLOY

Récapitulatif

Type de produit et méthode de diffusion	Taille de l'équipe	Maturité de la collaboration	Stratégie
Tout	Petites équipes	Haute	Trunk based
Produits permettant un déploiement et une mise à jour continus, tels que les produits SaaS.	Moyenne	Moyenne	GitHub-Flow et Trunk based
Produits avec une fenêtre de release définie et une cadence de publication périodique, tels que les applications iOS.	Moyenne	Moyenne	Git-Flow et GitLab-Flow avec branche de release
Produits exigeants en termes de qualité et supportant un déploiement et des releases en continu.	Moyenne	Moyenne	GitLab-Flow
Produits exigeants en termes de qualité et ayant un long cycle de maintenance pour les versions publiées.	Grande	Moyenne	Git-Flow