Git au service de l'intégration continue

Sandy Ingram

Plan du cours

- Intégration Continue
  - Définition de l'Intégration Continue
  - Lien avec le Déploiement Continu
  - Lien (ou rôle de Git)

- Zoom sur Git
  - Commandes de base Git
  - Git "stages"
  - **"Conventionnels commits"**
  - Gestion des conflits
  - GitFlow vs Développement "Trunk-based"

Intégration Continue (CI)

- CI est une pratique de développement incitant une équipe de développement logiciel de mettre en commun (intégrer) son travail fréquemment (plusieurs fois par jour)
- Grâce à des tests automatiques, le code “intégré” est automatiquement vérifié par détecter les erreurs relatives et les corriger au plus vite.

Avantages de l'intégration Continue (CI)

- La CI permet de maintenir une base de code stable et fiable.
- Elle facilite la livraison agile de code: rapide et de haute qualité.
- Elle favorise la collaboration et la communication fréquente au sein d'une équipe de développement

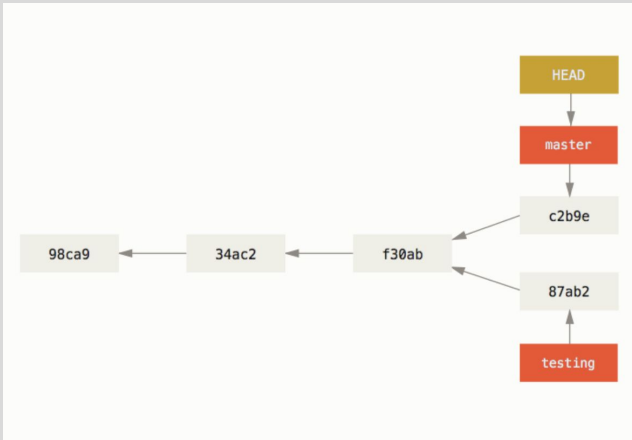
Et Déploiement Continu (CD)?

- Le CD se concentre le déploiement automatique des applications en “production”.
- L’objectif est de fluidifier et automatiser (le plus possible) le processus de mise en production.
- La mise en production n’a lieu que si les tests d’intégration sont réussis suite à une mise à jour du code (CI) en assurant les conditions de sécurité.
- Avantages: agilité, mise à jour fréquente, et satisfaction des utilisateurs finaux.

Définition de GIT

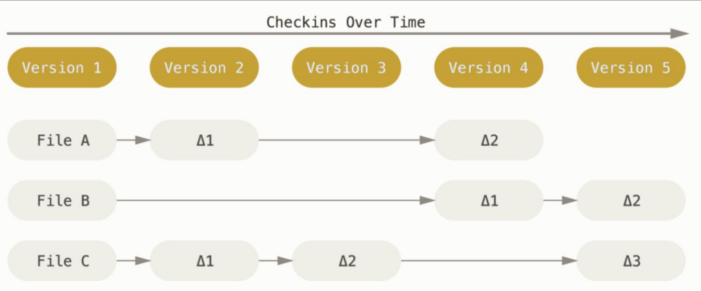
- Git est gratuit et “**open source**”: il offre un système puissant de contrôle des versions du code (CVS: Code Versioning System); Il facilite la gestion et la mise à jour du codebase partagé.

Pour illustrer, à droite: on voit deux “branches” (“testing” et “master”) d’un répertoire git,
et leurs commits communs et divergents, en série “temporelle”).

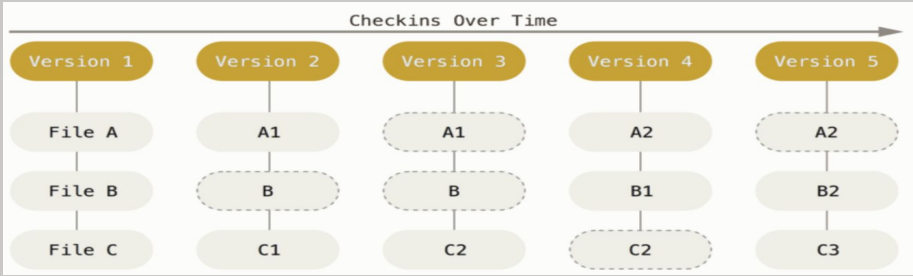


Définition de GIT

- Contrairement à d'autres CVS, Git prend une "snapshot" de l'état des fichiers après une mise à jour et stocke une référence à ce état. Si un fichier n'a pas changé, GIT garde juste un lien vers la version précédente déjà stockée.



"Delta-based Version Control": sauvegarde les changements dans chaque version



Git sauve l'état des fichiers (ou mini file système) après chaque version. B, A1, A2, et C2 n'ayant pas changé, on n'en garde qu'un lien vers la version précédente.

Git joue un rôle fondamental pour le CI/CD

- Git permet le **traçage** des changements du codebase:
  - en **local** pour permettre des développements parallèles
  - et en **ligne** (partagé) pour assurer la synchronisation fréquente => CI
- Il intègre des fonctionnalités puissantes de **branchement et "merge"** (fusion") des branches, facilitate la mise à jour fréquente de code partagé.
- Il intègre également des **pipelines CI/CD** facilement et configurables permettant des vérifications et des déploiements automatiques, chaque fois qu'un code est "poussé" dans le répertoire en ligne.
- Il offre aussi la possibilité de "rollback" ou "**revert**": annuler une mise à jour ou retourner sur une version précédente du code.
- GIT facilite aussi la revue de code ("code review") : les développeurs peuvent faire des "pulls requests".

Zoom sur git

Commandes git de base (Cheat Sheet)

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch.
`git reset <file>`	Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto `<base>`. `<base>` can be a commit ID, branch name, a tag, or a relative reference to HEAD.
`git reflog`	Show a log of changes to the local repository's HEAD. Add `--relative-date` flag to show date info or `--all` to show all refs.

git reflog permet de récupérer des commits perdus et n'est pas affecté par "git reset –hard"

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`.
`git checkout -b <branch>`	Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch.
`git merge <branch>`	Merge `<branch>` into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.
`git fetch <remote> <branch>`	Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

Source: Atlassian.com

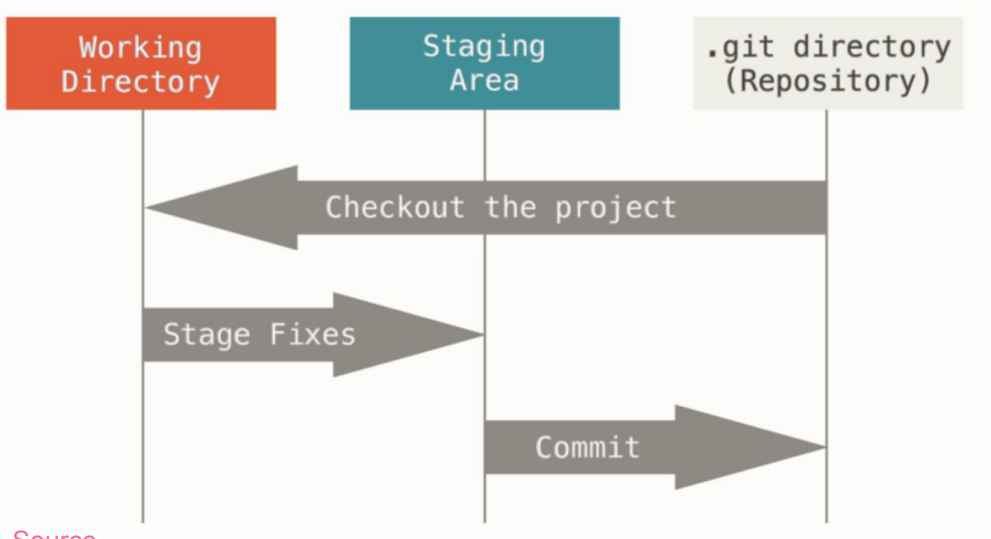
- (En option) Github education propose une GIT Cheat Sheet plus "avancée"

Pour l’examen:

- Il faut pouvoir expliquer et différencier toutes les commandes du cheatsheet d’Atlassian (slide précédent).

- Il faut pouvoir expliquer la différence entre git reset soft et git reset hard.

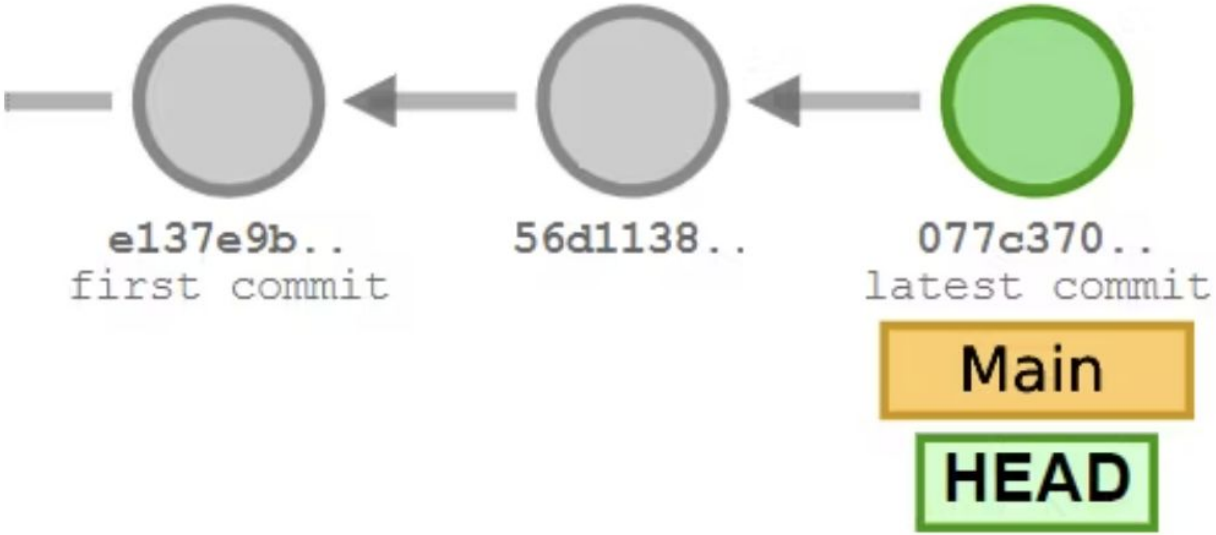
Les 3 états principaux de GIT (Three States)



Source

1. Un fichier **modifié** est dans votre “working directory” et n’est pas encore intégré à votre base de données git.
2. Un fichier “**staged**” a été sélectionné pour faire partie du prochain commit.
3. Un fichier “**commited**” est stocké dans la base de données **localement**.
4. Uniquement après un git **push**, le code sur le repertoire en ligne est mise à jour.

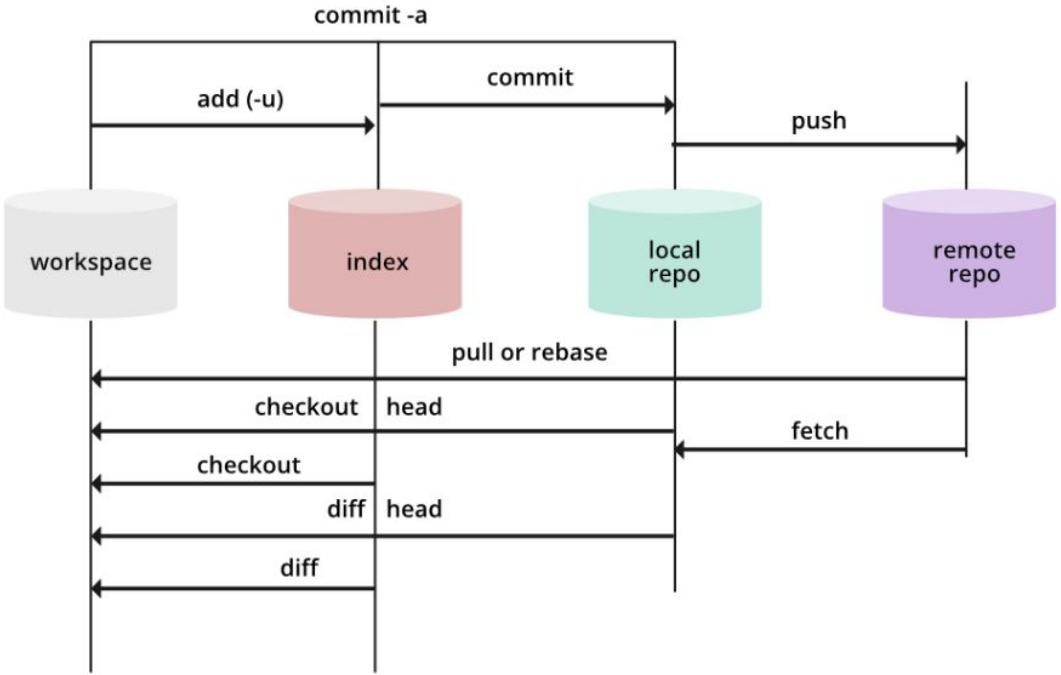
GIT “HEAD”



Source

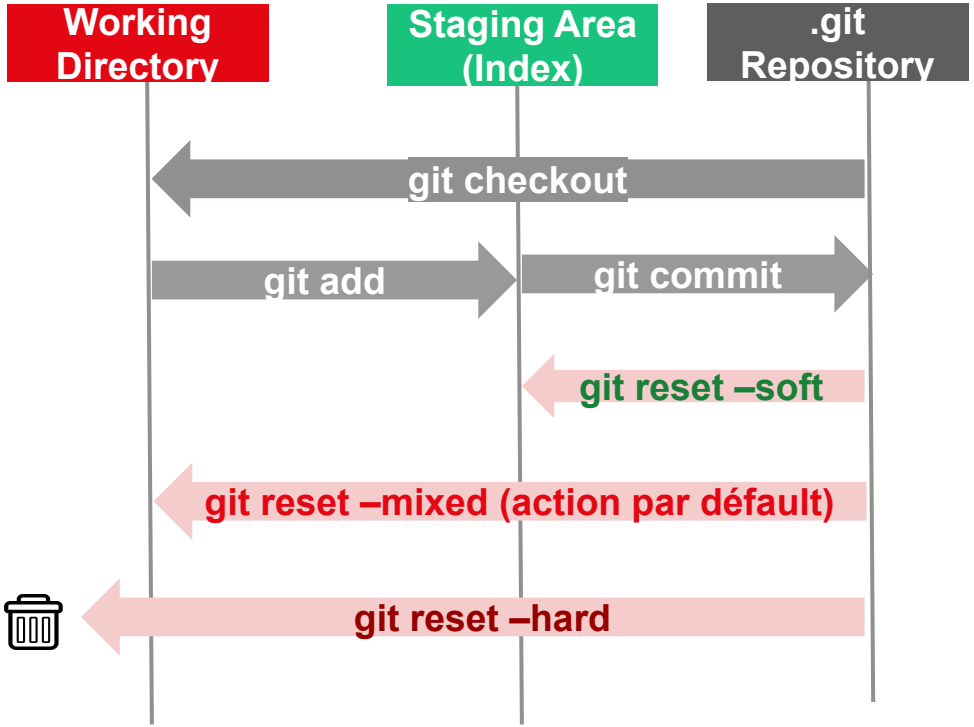
- HEAD se réfère normalement au dernier commit de la branche “active” (celle “checked out”) ou la branche principale.

Les états de Git représentés dans un autre schéma



Source

Git Reset



- **git reset --soft:** (“uncommit”): bouge HEAD au commit spécifié (par défaut celui précédent le dernier commit) mais **garde le commit “annulé”** dans **l’index** (facilitant typiquement un autre commit plus sélectif par exemple).
- **git reset --mixed:** bouge HEAD au commit spécifié **et** modifie l’état de l’index pour représenter le commit **spécifié**.
- **git reset --hard:** supprime complètement le commit à annuler, bouge HEAD au commit spécifié **et** remet **aussi l’état de l’index** et du **working directory** à celui spécifié. Tout changement dans le “working directory” pas “staged”, sera perdu.

Spécification d'un ou plusieurs avec git reset

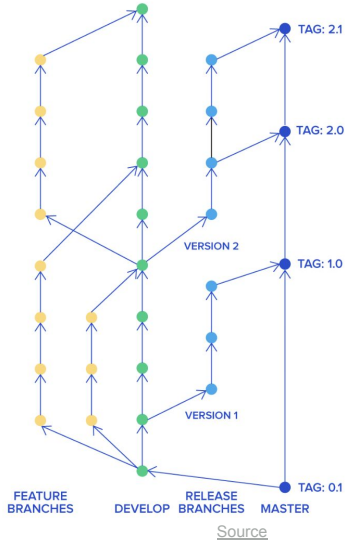
- ➢ Par défaut c'est le dernier commit (HEAD) qui est annulé et HEAD référencé le précédent, mais on peut tout à fait annuler plusieurs commits et spécifier le nouveau commit à prendre en compte en utilisant HEAD~n pour revenir en arrière jusqu'à n commits.
- ➢ Par exemple avec git reset --mixed HEAD ~3, l'état de l'index (Staging area) correspond à son état avant les 3 derniers commits précédent HEAD, et le "working directory n'est pas touché.
- ❖ Une autre option serait d'utiliser git reset HEAD^, HEAD^^ ou HEAD^2 pour indiquer respectivement le parent direct, le parent du parent, et le deuxième parent (en cas de fusion de branches).
- ➢ Selon vous, comment serait la syntaxe de l'option par défaut (le dernier commit?)

Slide Optionnel

Demo avec GIT Reset et réécriture de “l’histoire”

Demo Live :Supposer une situation où un fichier avec des vrais mot de passe a été poussé dans répertoire git partagé. Après quelques commits, on le remarque; le fichier était effacé, et un nouveau commit est poussé sans ce fichier. Toute personne qui clone/pull le répertoire à ce moment là, n’a pas accès à ce fichier, mais la trace dans les logs (historique) des commits est toujours là, on peut même y voir le contenu du fichier! Voici une démo de comment on peut résoudre ce problème en ré-écrivant l’historique de git.

Développement Trunk-based vs GitFlow

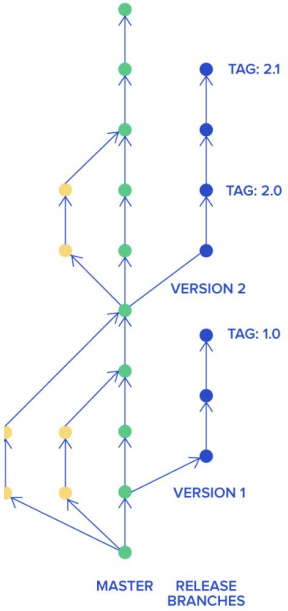


Source

GitFlow

Gitflow est pratique pour les gros projets “open source”, les produits complexes, et avec des développeurs “**juniors**”. Les revues de code s’effectuent sur les changements des branches concernées.

Ici on remarque des **“features branches” de longue durée**, un processus plus formel de déploiement avec aussi de “**release branches**” constituant une étape intermédiaire de test avant le déploiement définitif sur la branche master.



Trunk-Based

Avec moins de “formalisme” que gitflow, le dév. Trunk-based est adapté aux équipes avec développeurs “**seniors**”, aux situations nécessitant des itérations rapides, et en début de projet.

Ici il n’y a plus vraiment de branches “develop” ni “release branches” formelles. Ce qu’on représente par “release branches” un marquage ou tag du commit poussé sur master.

* Dans les deux cas, le tag ou marquage du commit se fait au moment de la mise à jour de la branche master.

Revue de codes avec git

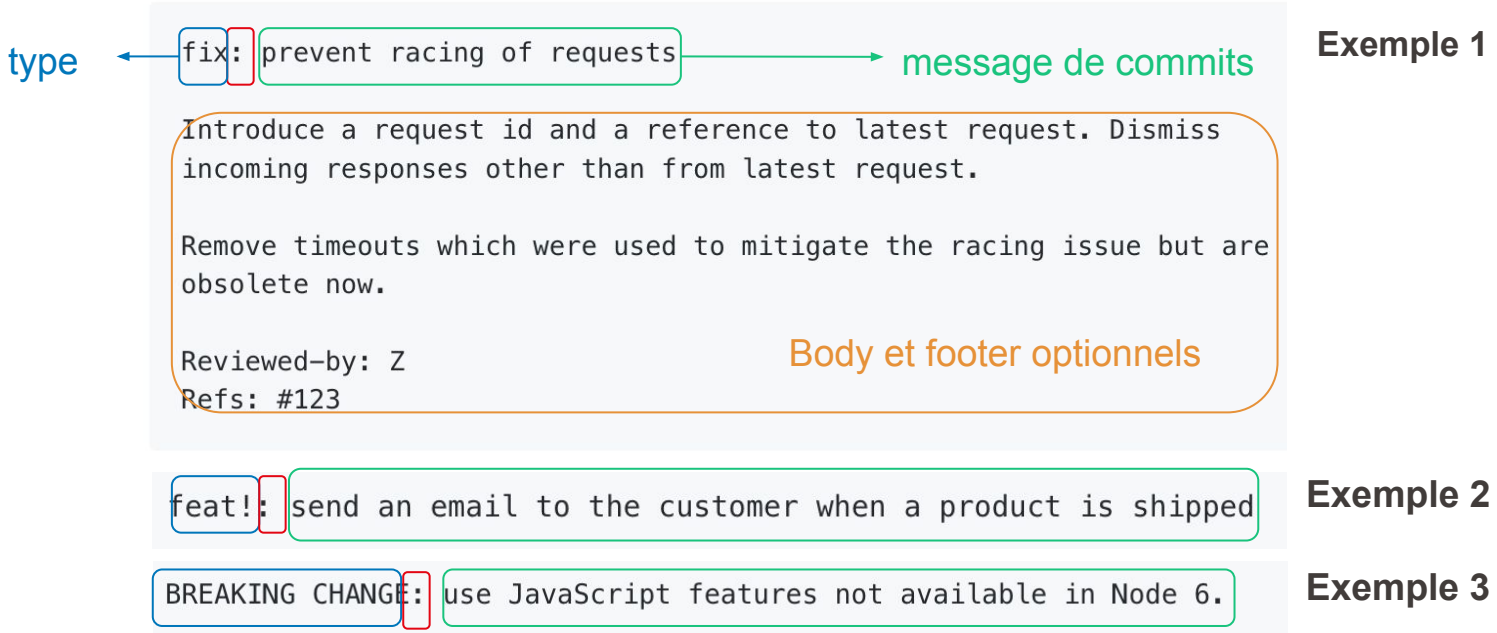
1. Réaliser un commit et un push vers le répertoire git “remote”.
2. Ouvrir un “Pull Request” (ou PR):
    a. Sur gitlab, on ouvre un MR “Merge Request” (sachant que la commande pull = fetch + merge!)
    b. L’idée est de faire une demande formelle de revue de code et approbation pour intégrer son code d’une branche de développement dans la branche principale du projet (master).
3. Un autre développeur réalise une revue de code, fournit un feedback ni nécessaire, des changements pourront être nécessaires
4. Une fois la demande approuvée, la merge request est approuvée et le code est intégré.

Gestion des conflits

- Lors de la fusion (“merge”) de code entre deux branches, ou la mise à jour d’une branche locale (avec un git pull), des changements effectués dans les mêmes fichiers, peuvent générer des “conflits”
  - on peut déléguer la résolution de ces conflits à git, en précisant quelle version privilégier (--ours vs --theirs)
  - ou on les résout manuellement avec l’aide d’un éditeur de texte: en ouvrant les deux versions du même fichier “side-by-side” pour comparer la différence, sachant que git marque la différence avec (des"<<<<<<<", "=======", ">>>>>>>").
  - L’option “Blame” permet également de voir à quel moment une ligne dans un fichier a été modifié, par qui et dans quel commit.

“Conventional commits”

- Des spécifications pour rendre les messages de commits, plus “propres” et faciles à lire pour les humains et les machines.



Liens entre gitlab board “issues” et commits

- Utiliser dans le message du commit un croisillon (#) suivi du numéro de l’issue pour faire un lien entre le commit et le “issue”. Normalement, un log de cette action sera automatiquement ajouté dans le “issue” en question.

- Dans les “issues” fermés en cours, laisser un commentaire en spécifiant un lien (URL) vers le commit réalisés, tagger les membres concernés du groupe en expliquant brièvement ce qui a été réalisé.

- Suivre la démo en classe en guise d’exemple.

Attentes pour le projet

1. Message de commits selon “conventionnel commits”, historique des commits “propres”
2. Gestion des branches avec choix clair en trunk-based development ou git flow (ou autre approche à justifier)
3. Code review (pull request)
4. Utilisation du gitlab board pour la documentation et planification des tâches avec lien aux commits pour tâche accomplies.

Références

- https://git-scm.com/
- https://git-scm.com/docs/git-reset
- https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development
- https://acompiler.com/git-head/
- Extra: https://circleci.com/blog/git-detached-head-state/
- Extra: Rebase vs Merge: https://acompiler.com/git-head/#tve-jump-17716bdd851