Bonne pratiques et mauvaises pratiques Git

ESGI - 3ème année IW 2020

Introduction

Git est un outil puissant pour travailler en équipe sur un code commun, et il y a des mauvaises pratiques qui peuvent pénaliser votre équipe ou/et vous-même, vous faire perdre du code, ouvrir des failles de sécurités…. Et des bonnes pratiques pour éviter ça.

git push --force (ou -f)

Sert à réécrire l’historique d’une branche distante en l’écrasant avec l’historique local

Utile dans le cas où une branche distante doit être corrigée en revenant à un commit fonctionnel par exemple

Risque:

- On risque de perdre du code poussé par un collègue qu’on a pas récupéré
- On force les développeurs utilisant la branche à resynchroniser leurs historiques locaux

git commit --amend

Sert à modifier le dernier commit, que ce soit les modifications du code ou le message de commit

Utile pour corriger un commit:

- lorsque le message est incomplet ou stupide (style “WIP”, “Done”) ou ne correspond pas aux modifications
- quand on a laissé une ligne de debug, l’affichage de message de log, ou des informations sensibles

Risque:

- Si le commit est déjà poussé sur le remote, il faut aussi réécrire l’historique avec un git push --force, avec les mêmes risques et conséquences

Secrets

Les applications ont très souvent besoin de clefs de sécurité, mots de passe, clef de chiffrements etc => **Données sensibles**

Il faut être très vigilants pour ne jamais commit de données sensibles

Si cela survenait, il faut s'assurer de supprimer le commit de l'historique et ne pas juste ajouter un commit supprimant la donnée sensible, elle apparaîtrait alors deux fois dans l'historique, donc deux fois pire

Lorsqu'une application nécessite l'utilisation de secrets, on utilise des variables d'environnement, des "placeholders" dans le code remplacés à l'exécution.

.env

En général, les secrets sont chargés depuis l'environnement, mais dans l'environnement local on utilise un ou plusieurs fichiers .env contenant nos variables d'environnement et qui sont alors chargés au lancement.

Bonne pratiques:

- Ajouter une ligne ".env" aux fichiers ignorés par git dans un .gitignore
- Commit un fichier ".env.dist" (ou autre nommage) contenant les noms des variables sans les valeurs sensibles, exemple:

```
APP_SECRET=
DB_KEY=
ROOT_SALT=
```

.gitignore

Fichier contenant des chemins de fichiers, dossiers, ou pattern de chemins pour ignorer des fichiers.

git add et git status ignoreront les chemins listés, évitant des commits malheureux

Ils est nécessaire d’ignorer les fichiers sensibles, mais il est souvent utile d’ignorer les fichiers générés lors de votre travail: binaires et exécutables, fichiers temporaires, fichiers générés par votre IDE, et tout ce qui n’a pas intérêt à se retrouver partagés pour éviter de polluer l’historique et le dossier de travail.

Branche par défaut

Sauf au début et autant que possible, il faut éviter (voir interdire) de commit directement sur la branche par défaut du projet

Risque:
Pousser du code mal conçu, non validés par l’équipe, souvent source de bug

Branche de travail

Une branche de travail, que ce soit pour une nouvelle fonctionnalité ou un bug, ne devrait concerner qu'une seule tâche.

Si vous devez travailler sur 3 tâches simultanéments, ouvrez 3 branches différentes

Risque:

- Si vous utiliser une même branche pour traiter plusieurs tickets, vous prenez le risque de devoir modifier l'historique si une des tâches est modifiée ou supprimée
- En ouvrant une Pull/Merge request, vous compliquez le travail de review en ne traitant pas qu'un seul ticket dedans, et vous risquez de devoir corriger autant de code que de retours fait concernant plusieurs problématiques, donc aussi vous surcharger de corrections

git reset

Sert à restaurer l’historique vis-à-vis d’un commit en particulier. Utile lorsqu’il faut revenir à un point donné pour faire une review locale, revenir à un code fonctionnel pour comparer, changer de branche etc

Conséquence, on perd le code indexé dans ces fichiers suivi.

Bonne pratique:
Ne pas hésiter à commit ou même à stash son code pour éviter de le perdre

Git lfs

Notamment dans le jeu vidéo mais aussi dans le web, il peut arriver de devoir commit des fichiers très volumineux. Ça l'est d'autant plus pour git qui ajoute à l'historique autant de contenu, en particulier pour les fichiers binaires, ce qui alourdit fortement l'historique et peut rendre très long les futur git clone, git pull et push, forcément.

Bonne pratique:
Si les fichiers ne peuvent pas être stocké en dehors de git, vous pouvez utiliser git-lfs qui gèrent le tracking de fichiers volumineux indépendamment (il y a souvent un espace de stockage limité à un ou deux Go)