Git workflows - Module 3 : Git *essentials*

Paul Schuhmacher

contact@pschuhmacher.com

version: 1.0

Correction des exercices du cours

Prérequis:

- installer git sur sa machine

Exercice 1

1. **Créer** un dépôt local qui contiendra vos notes sur ce cours

```
mkdir git-workflows
cd git-workflows
git init
```

2. **Créer** un fichier `README.md` et renseigner le titre

```
touch README.md
echo "Git workflows - M1" >> README.md
```

3. **Ajouter** ce fichier au *stage*

```
git add README.md
```

4. **Faire** un commit

```
git commit -m "premier commit"
```

5. **Inspecter** les commits avec `git log`
6. **Créer** un fichier `foo.md`
7. **Ajouter** ce fichier au *stage*
8. Finalement nous avons décidé de le supprimer. **Supprimer** le avec `git rm`. Que remarquez-vous ? Quelles sont vos deux options ? **Finaliser** le commit **puis supprimer le fichier dans un nouveau commit**. Inspecter le contenu *du dossier*

```
touch foo.md
git add foo.md
git rm foo.md
```

`git rm` permet de supprimer un fichier du *working tree* et de l'*index*. Comme le fichier a des changements indexés, on ne peut pas le supprimer comme cela. Nous avons deux options:

- soit le supprimer définitivement avec l'option `--force` (du working tree et de l'index)
- soit le supprimer uniquement de l'index avec l'option `--cached` (de l'index uniquement)

Comme indiqué dans la suite de l'exercice, on le supprimera ensuite dans un nouveau commit.

```
git commit -m "ajout fichier foo"
git rm foo.md
git commit -m "supp fichier foo"
```

> *On aurait pu supprimer le fichier en dehors de git avec un `rm foo.md`, puis commit la suppression du fichier. L'avantage de `git rm foo.md` c'est que le fichier est supprimé et le changement est stagé directement dans l'index, il n'y a plus qu'à commit (plus concis).*

9. À l'aide de `checkout`, **se déplacer** dans l'état précédent (où le fichier `foo.md` existe)

```
#retrouver le hash du dernier commit
$ git log
commit a3998060092dcef6c920cb56e452529528b69f79 (HEAD -> main)
Author: websealevel <contact@websealevel.com>
Date:   Thu Feb 9 15:58:40 2023 +0100

    suppr foo

commit 16def2403118dd98ecea67549336160593e78b4b
Author: websealevel <contact@websealevel.com>
Date:   Thu Feb 9 15:57:15 2023 +0100

    ajout foo
#ici je vois que le commit où le fichier existe encore est le commit 16def2...
#je peux renseigner que quelques caractères du hash, git comprend
git checkout 16def
```

10. **Revenir** au dernier commit réalisé avec `checkout`

On revient sur le dernier commit en retournant sur la branche `main`

```
git checkout main
```

> *`git reflog` permet d'inspecter tous les mouvements des références (HEAD et branches), et permet de retrouver les commits qui peuvent plus être indiqués par `git log`. C'est un outil d'inspection plus complet, mais plus bas niveau.*

Exercice 2

1. **Créer** une branche de développement appelée `dev`.

```
git branch dev
git checkout dev
# ou en une ligne
git checkout -b dev
```

2. **Créer** les fichiers `index.html`, `index.js` et `style.css`.

```
touch index.html index.js style.css
```

3. On souhaiterait merger la branche dev avec la branche sur `main`, mais on souhaiterait créer un commit de merge. Or dans ce cas, le `merge` peut s'en passer (voir *fast-forward*). À l'aide de `git merge -h`, **trouver** l'option qui permet de le faire.

Une inspection de la doc avec `git merge -h` ou `man git merge` nous apprend qu'il existe une option `--no-ff: With --no-ff, create a merge commit in all cases, even when the merge could instead be resolved as a fast-forward`. L'option `--no-ff` ou *no fast-forward* permet de créer un commit même si les commits de la branche `dev` à fusionner sont *des descendants directs* de la branche `main` (cad, que la branche `main` peut simplement `être avancée` des commits car elle ne comporte aucun commit depuis la création de la branche `dev`).

1. **Merger** localement cette branche dans `main` en forçant la création d'un commit de merge avec l'option trouvée précédemment.

```
git checkout main
git merge dev --no-ff
```

2. **Lister** les commits et identifier le commit de `merge`.

```
git log
```

3. **Lister** les branches locales et la branche courante.

On appelle la commande branch avec l'option -vv pour `verbose`

```
git branch -vv
```

Exercice 3

1. **Lire** la documentation de merge: `man git merge`
2. **Lire** la documentation de rebase: `man git rebase`
3. **Quelle** est la différence fondamentale entre les deux ?

Les deux commandes merge et rebase existent pour résoudre le même problème: rapatrier du travail d'une branche dans une autre. Elles le font chacune d'une façon très différente.

`git merge` fusionne les deux branches en créant un commit parent aux deux derniers commits de chaque branche. merge est non destructif, il ne modifie pas les branches existantes. Mais si la branche sur laquelle on rapatrie les changements (ici `main`) a été active, alors de potentiels

conflits doivent être réglés. L'historique peut alors vite devenir compliqué à lire, mais chaque *merge commit* apporte du contexte.

`git rebase` fonctionne davantage comme une copie: les commits dans la branche à fusionner sont enregistrés dans une zone temporaire puis appliqués, un par un, à la branche courante. L'historique est beaucoup plus propre: pas de commit de merge et l'historique est linéaire (on réécrit l'histoire en quelque sorte, contrairement à merge). L'inconvénient, c'est qu'en réécrivant l'historique vous pouvez, si vous ne prenez pas garde, faire des choses catastrophiques pour le travail en collaboration sur un même dépôt. Pour éviter cela lisez cet article sur les règles d'or à respecter en utilisant rebase.

4. **Refaire** l'exercice précédent avec `rebase`. Pour cela, **supprimer** le dernier commit de merge avec `reset`.

Notre dernier commit était le commit de merge. Il nous suffit donc de le supprimer avec `reset`, puis de `rebase dev` sur `main`

```
git checkout main
git reset --hard HEAD~1
git rebase dev
```

Exercice 4 : cherry-pick et rebase interactif

1. Sur la branche `dev`, **modifier** les fichiers suivants:
    1. **Ajouter** `<!DOCTYPE HTML><HTML><head><link rel="stylesheet" href="style.css"></head><body><h1>Git workflows</h1></body></html>` dans `index.html` (commit *contenu de la page web*)
    2. **Ajouter** `console.log('hello git')` dans `index.js` (commit *js implementation*)
    3. **Ajouter** `html{color:red}` dans `style.css` (commit *style html red*)

```
git checkout dev
echo "<!DOCTYPE HTML><html><head><link rel="stylesheet" href="style.css">
</head><body><h1>Git workflows</h1></body></html>" > index.html
echo "console.log('hello git')" >> index.js
echo "html{color:red;}" >> style.css
```

2. **Créer** un commit **pour chaque changement** (3 commits)

```
git add index.html
git commit -m "contenu de la page web"
git add index.js
git commit -m "js implementation"
git add style.css
git commit -m "style html red"
```

3. Pour notre site web, on souhaiterait récupérer le nouveau code HTML et css, **mais pas le js** pour le moment, sur la branche `main` . Utiliser `cherry-pick` pour le faire **en une commande**.

```
git checkout main
# faire un git log pour retrouver les hash des commits recherché
git cherry-pick <hash commit contenu de la page web> <hash commit style HTML
red>
# inspecter
git log
```

4. Sur la branche `dev`, **corriger** le document HTML pour intégrer le script `index.js`. (commit *integration js*)

```
git checkout dev
# ajouter <script src="index.js"></script> juste avant la balise fermante de
body
git add index.js
git commit -m "integration js"
```

5. Récupérer ce travail sur la branche main en utilisant `rebase`.

```
git checkout main
git rebase dev
```

6. En utilisant le git rebase interactif, **réorganiser** les commits dans l'ordre suivant:
    1. *contenu de la page web*
    2. *integration js*
    3. *style html red*
    4. *js implementation*

```
git checkout main
git rebase -i HEAD~7
# Dans nano, changer l'ordre suivant la consigne puis appliquer les changements
en quittant l'éditeur
```

7. Toujours à l'aide de rebase interfactif, **fusionner** les deux commits *contenu de la page web* et *integration js* avec comme nouveau message *markup init*

```
git checkout main
git rebase -i HEAD~7
# Dans nano, utiliser s pour squash devant le commit integration js pour le
fusionner avec le précédent
```

8. Toujours à l'aide de rebase interfactif, **réécrire** le commentaire du commit *style HTML red* en *HTML red color*.

```
git checkout main
git rebase -i HEAD~7
# Dans nano, utiliser r pour rewor le commit style HTML red
# Quitter nano, un deuxième éditeur s'ouvre, changer le commentaire
# inspecter
git log
```

Exercice 5: dépôt distant

1. **Créer** un compte sur Github (si vous n'en avez pas déjà un)
2. **Créer** un *dépôt* distant pour héberger le contenu de ce cours
3. **Utiliser** votre précédent dépôt (Exercice 1) pour le faire pointer sur le dépôt distant avec `git remote add origin <url du dépôt distant>`

```
git remote add origin <url du dépot>
```

4. **Pousser** vos précédents *commit* sur le *dépôt* distant, avec `git push --set-upstream origin main`

   Vous allez devoir vous identifier, soit par mot de passe/token (HTTPS), soit par clé SSH (SSH)

```
git push --set-upstream origin main
```

5. **Inspecter** les branches *locales*

```
git branch -vv
```

6. **Inspecter** les branches *sur le dépôt distant*

```
git branch -a
```