



Formation git - avancé

Comprendre git pour l'utiliser encore mieux tous les jours



Agenda

Basique

- 1 - Découverte de git
- 2 - Hands-on : premier pas avec git
- 3 - De SVN vers git
- 4 - Hands-on : comprendre les basiques de git
- 5 - Hands-on : travailler avec un repo. Distant
- 6 – Hands-on : git stash, blame
- 7 - Hands-on : utiliser un workflow
- 8 - Hands-on : ré-écrire l'histoire

Avancé

- 1 - Git sous le capot
- 2 - Hands-on : rebaser pour changer l'histoire
- 3 - Merge vs rebase
- 4 - Hands-on : git cherry-pick, bisect
- 5 - Hands-on : revenir en arrière
- 6 - Hands-on : git hook
- 7 - Hands-on : travailler avec gitlab





— Comprendre le fonctionnement interne ?

- Lors de cette formation, nous allons parler d'opérations qui modifient l'historique de git comme le rebase. Nous sommes convaincus que comprendre les fondamentaux sur lesquels s'appuient git facilitent la compréhension de ces opérations complexes
- Comprendre les 4 concepts sur lesquels git est bâti, aide à comprendre les commandes :
 - > Architecture d'un dépôt serveur
 - > Architecture d'un dépôt de travail
 - > Signature de contenu & Stockage sous git
 - > Structure de l'historique git



Architecture d'un dépôt serveur

- Create a bare repository
 - `git init --bare`

```
.  
├── branches  
├── config  
├── description  
├── HEAD  
├── hooks  
│   ├── applypatch-msg.sample  
│   ├── info  
│   └── exclude  
├── objects  
│   ├── info  
│   └── pack  
└── refs  
    ├── heads  
    └── tags
```

- Un dépôt “bare” ne contient que les données brutes, vous ne pouvez pas travailler dessus.
- Ce dossier contient tout l'historique git
- Tout ce qui se produit sur le dépôt est contenu dans ces fichiers
- Ce type de dossier correspond à un serveur git. Vous pouvez le cloner pour construire un dossier de travail
- Le dossier hook contient des scripts shell à exécuter sur des événements (avant un commit, après un commit, ...)
- Je peux sauvegarder mon dépôt en copiant ce dossier
- Je peux cloner ce dépôt si j'y ai accès



Architecture d'un dépôt de travail

- Create a repository
 - `git init`
- Sur le poste développeur, le dépôt meta data est stocké dans le dossier `.git` directory

```
.
├── file
└── .git
    ├── branches
    ├── COMMIT_EDITMSG
    ├── config
    ├── description
    ├── HEAD
    ├── hooks
    └── ...
```



Signature de contenu

- git utilise le sha1 pour hasher un contenu et calculer une signature sur 40 caractères

```
echo "Fabien Arcellier" > author.txt  
git hash-object author.txt > author.txt.sha1
```

4a8c44c065f8184306fd345350efe150a848f53e

- Cette signature est unique (enfin presque) et dépend exclusivement du contenu
 - > Elle ne change pas dans le temps
- Cette signature permet de valider :
 - > Identité : chaque objet a sa propre identité
 - > Intégrité : vérifier si un fichier n'est pas corrompu sur le disque
 - > Checksum : vérifier lors d'un transfert de fichier que le fichier reçu est identique à celui envoyé



Comment git stocke l'information ?

```
.  
├── ...  
├── HEAD  
├── hooks  
│   ├── applypatch-msg.sample  
│   └── ...  
├── index  
├── info  
│   └── exclude  
├── logs  
│   ├── HEAD  
│   └── refs  
│       └── heads  
└── master
```

```
├── objects  
│   ├── df  
│   │   └── 2b8fc99e1c1d4dbc0a854d9f72157f1d6ea078  
│   ├── e6  
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
│   ├── e8  
│   │   └── 3a550e898efe663f57b8d77d58d35dbf3cbaba  
│   └── info  
└── pack  
└── refs  
    ├── heads  
    │   └── master  
    └── tags
```

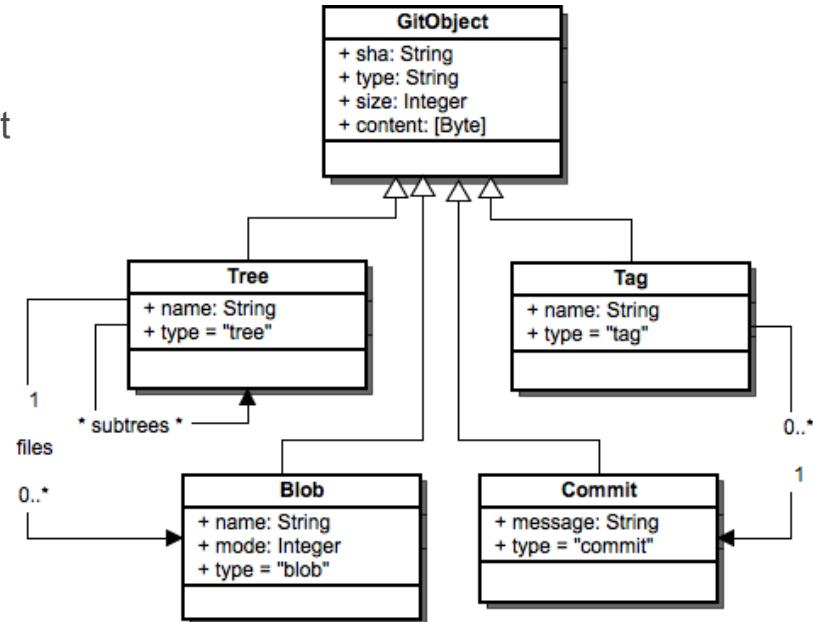
- Chaque objet est stocké dans le dépôt avec son sha-1
 - C'est par ce moyen que git assure l'intégrité et la consistance d'un dépôt
 - C'est par ce moyen que git peut se synchroniser avec un dépôt distant
 - Les objets sont immuables
- Les objets sont compressés par la suite sous la forme d'un fichier pack

S. Potter, "Git," in The Architecture of Open Source Applications (Volume 2), vol. 2, 2015, pp. 89–101

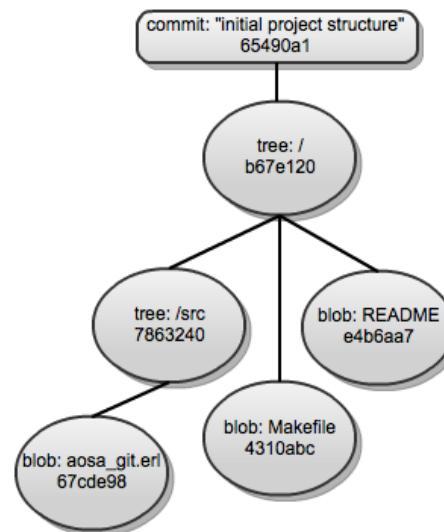


Objet interne de git

- Il y a 4 primitives internes sur lesquels repose git
 - Blob : contenu du fichier
 - Tree
 - Tag
 - Commit
- Tous les objets sont identifiés par un sha1



- Représentation d'un commit sous git

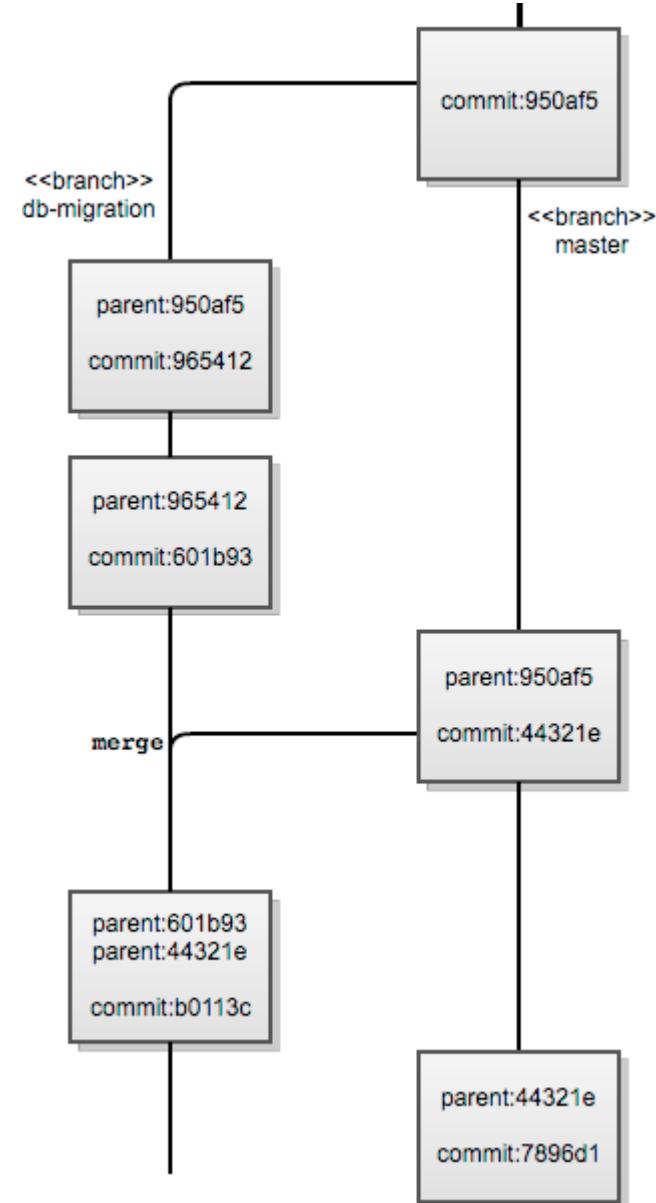


S. Potter, "Git," in The Architecture of Open Source Applications (Volume 2), vol. 2, 2015, pp. 89–101



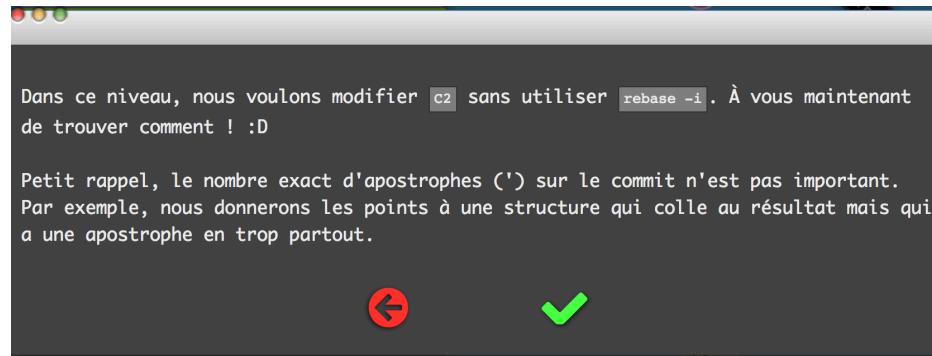
Structure de l'historique git

- Une branche ou un tag dans git représente un label sur un commit. Leur coût est négligeable.
- L'historique git est implémenté sous la forme d'un directed acyclic graph
 - C'est pourquoi vous changez l'origine d'un commit (l'action rebase)
 - git est un toolkit pour manipuler ce graph, c'est l'une des raisons pour lesquelles, il peut être difficile d'accès pour un débutant



Pour aller plus loin dans la manipulation des graphes

<http://learngitbranching.js.org>



Apprenez Git Branching

```
$ git checkout -b feat_br1
$ git commit
$ git commit
$ git checkout master
$ git commit
$ git checkout feat_br1
$ git rebase master
```

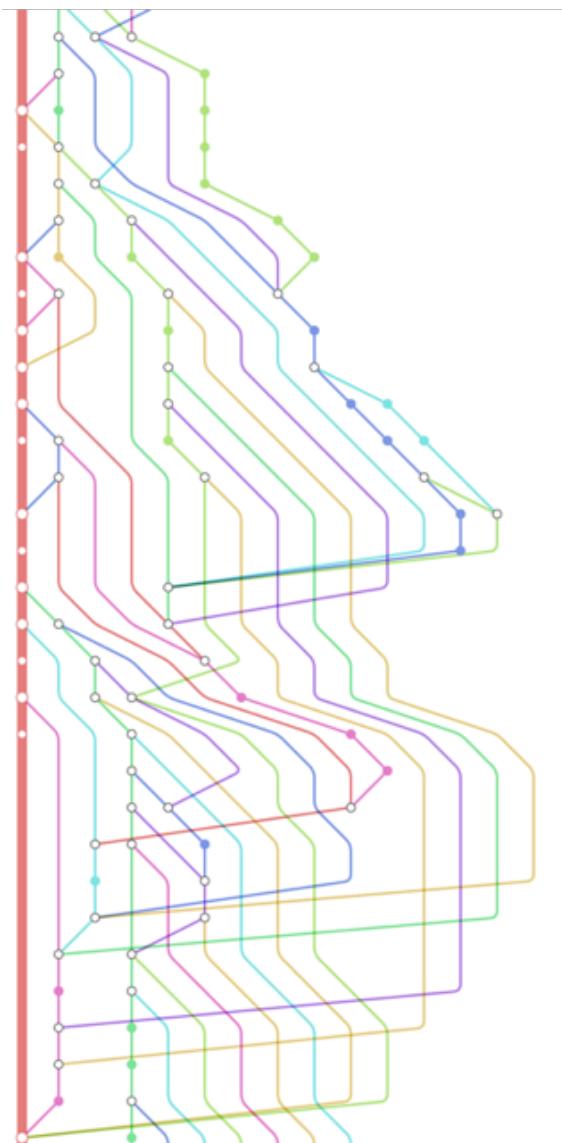
The diagram illustrates a git history graph. It shows several commits: C0 (top), C1, C2, C3, C2', C3', and C4 (bottom). A grey arrow connects C3 to C2. A black arrow connects C2 to C1. A black arrow connects C1 to C4. A pink arrow labeled "master" connects C4 to C2'. A pink arrow labeled "feat_br1*" connects C3' to C2'. A diagonal banner on the right says "Fork me on GitHub".

Hands-On

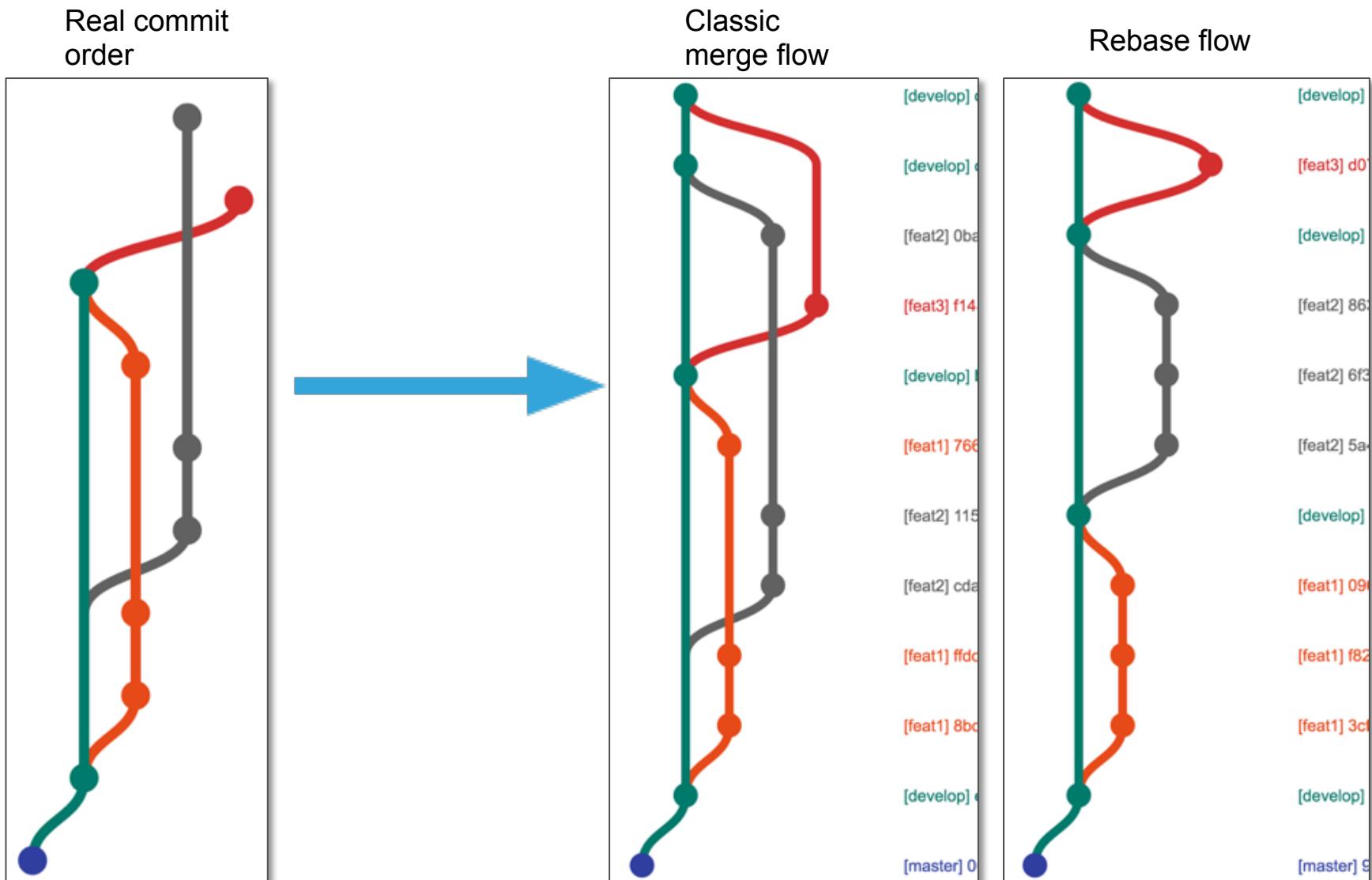
Rebaser pour changer l'histoire



Pourquoi faire un rebase ?

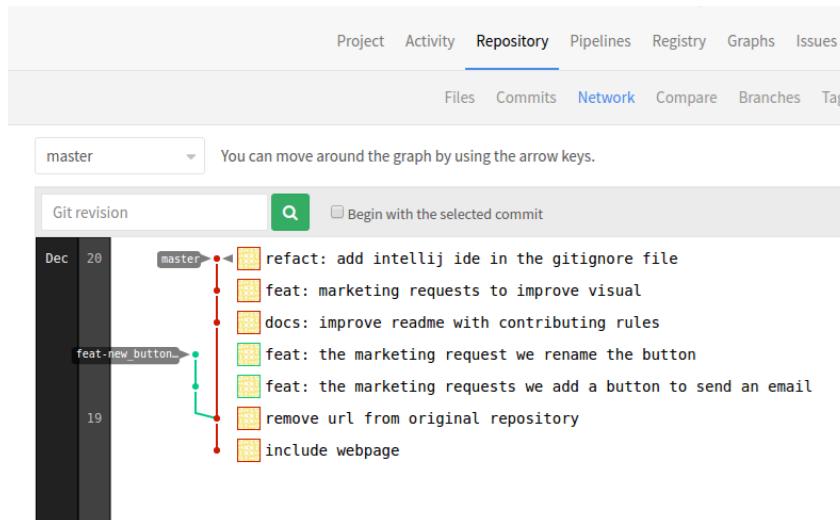


Pourquoi faire un rebase ?



Préparer un dépôt pour travailler le rebase

- Forkez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase/>
- Le visualiser sur gitlab

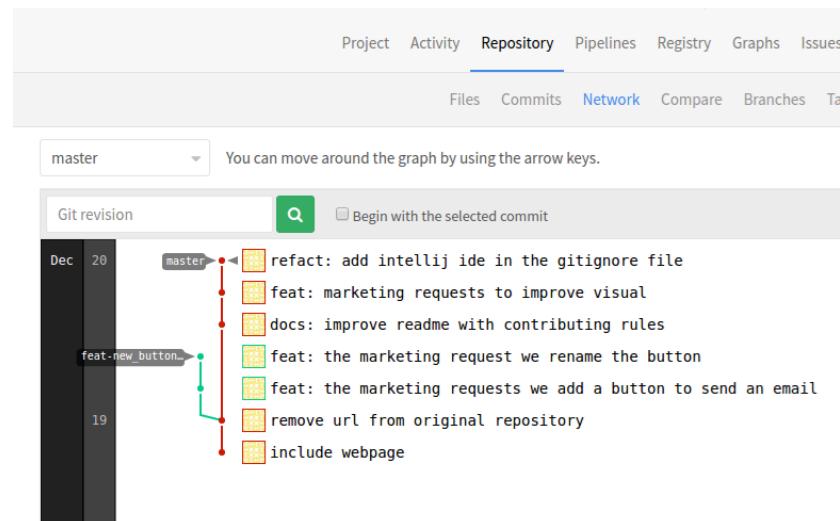


Affichez en console les 2 branches qui divergent



Préparer un dépôt pour travailler le rebase (réponse)

- Forkez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase/>
- Le visualiser sur gitlab



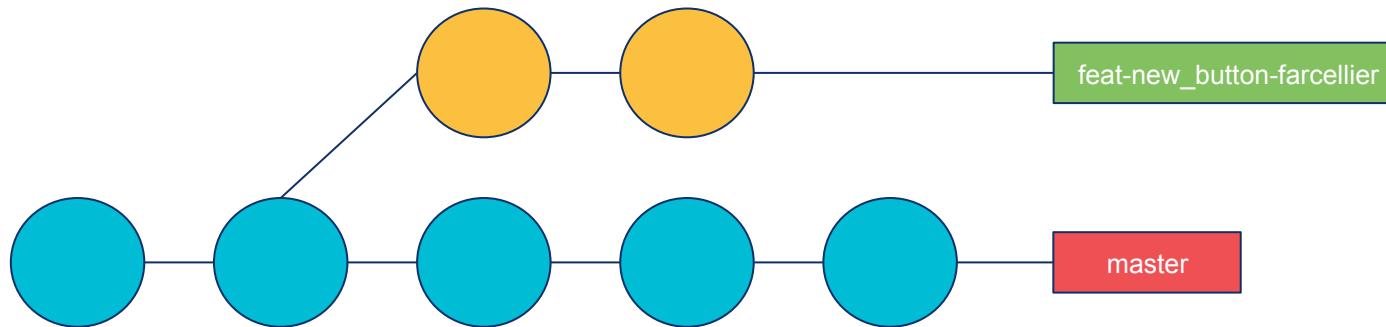
Affichez en console les 2 branches qui divergent
> git log --graph --oneline --decorate origin/master origin/feat-new_button-farcellier



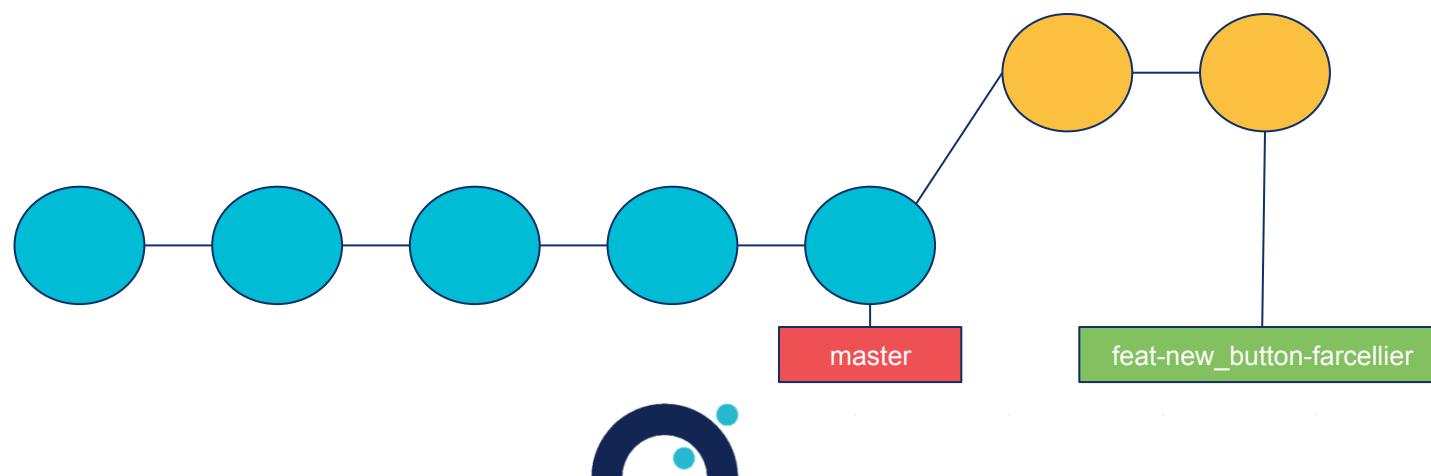
Rebase (1/2)

Un rebase consiste à détacher une branche d'un commit de notre graph git, pour le recoller sur un autre commit.

En interne, git reconstruit les commits et les rejoue. **Leur hash va changer.**



```
?> git checkout feat-new_button-farcellier  
?> git rebase master
```



— Rebase (2/2)

Le rebase est utilisé de façon classique pour intégrer les changements de la branche de référence à votre branche de travail.

Un raccourci consiste à exécuter depuis votre branche de travail (`feat-new_button-farcellier`) un git pull

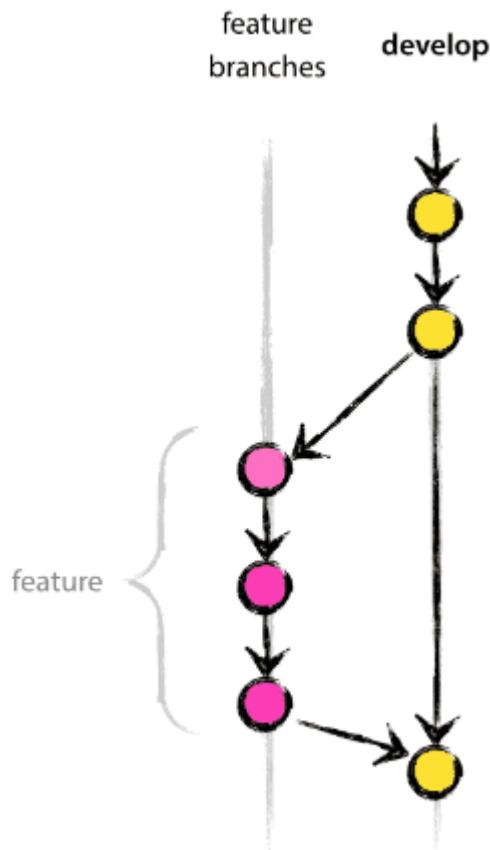
```
?> git pull --rebase origin master
```

Hands-on

- > Clonez le dépôt une deuxième fois
- > Depuis le premier dépôt cloné, effectuez un commit sur le master et poussez le sur le gitlab
- > Depuis le second dépôt cloné, sur la branche `feat-new_button-farcellier` effectuez un rebase direct



Feature branch - au quotidien



- créer une branche de travail
 > git checkout -b ma_branche
- faire vos commits régulièrement
- synchroniser votre branche régulièrement
 > git pull --rebase origin master
- pousser vers gitlab régulièrement
 > git push origin ma_branche

Si vous avez ce message sur votre branche lors d'un push
(après un pull --rebase)

```
Cannot write over Jane's commit
To git@github.com:mikrob/testgit.git
! [rejected]      ma_branche -> ma_branche (non-fast-
forward)
....
```

```
git push -f origin ma_branche
```



Le rebase en cas de conflit

Hands-on

- > Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase-conflict>
- > Effectuez un rebase de la branche *feat-new_button-farcellier*

- Pour résoudre ce conflit :
 - > Vous devez obtenir 3 boutons sur la page (Find out more, our product, contact us)
 - > Ajoutez le fichier corrigé dans la zone de staging (ne pas commiter)
 - > Pour achever le rebase, git rebase --continue

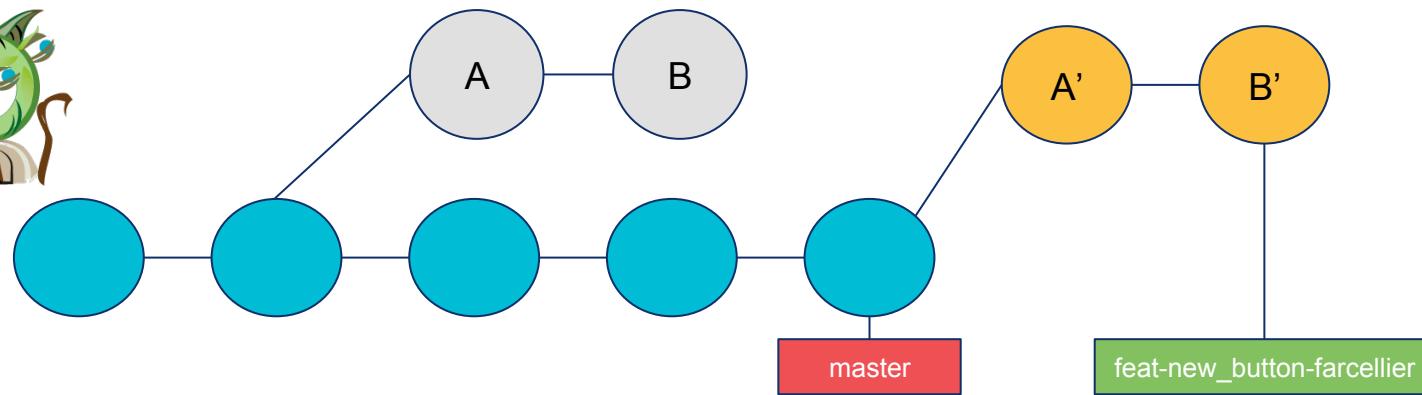
- Vous êtes perdu ou vous avez commis malencontreusement
 - > Abandonnez le rebase, git rebase --abort



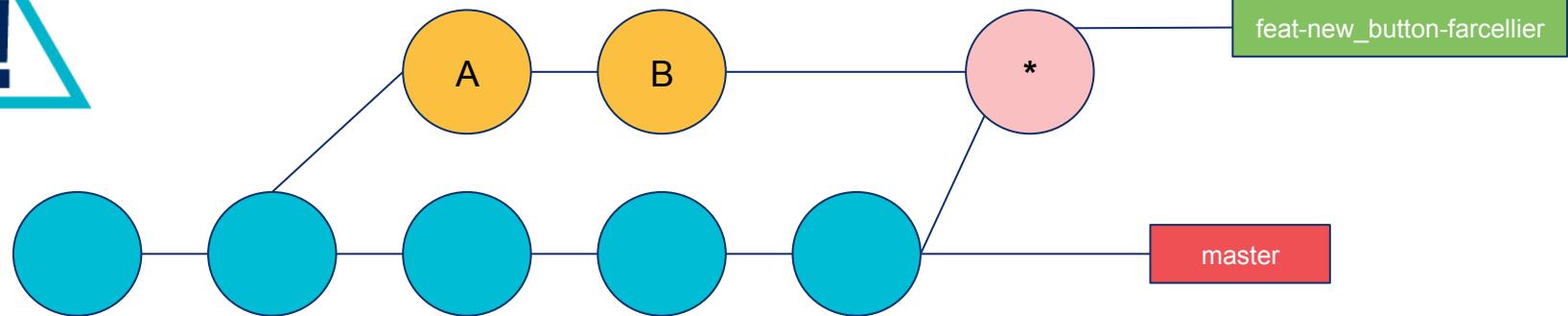


Merge vs rebase

Merge et rebase en une diapo. sur une feature branch



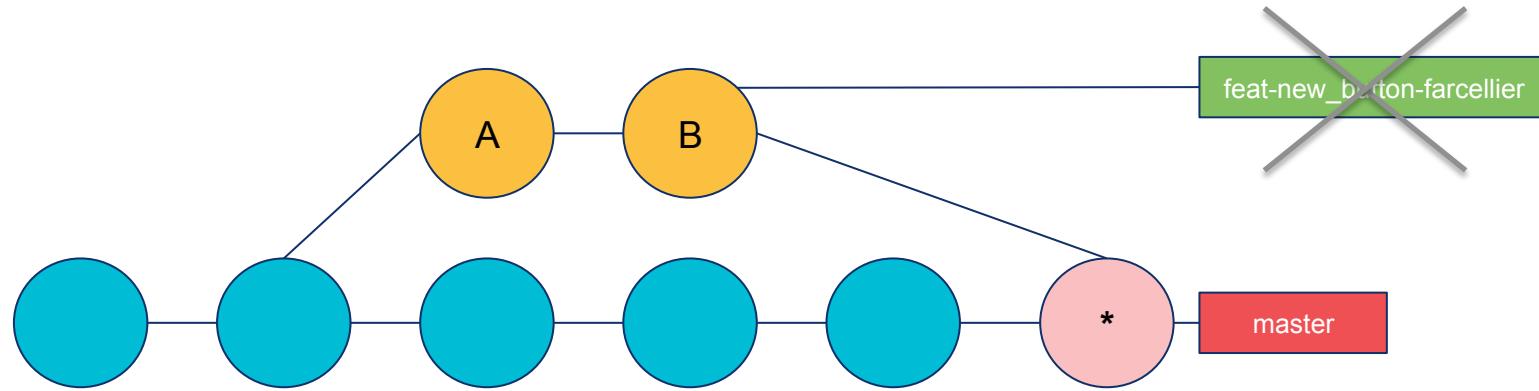
Un rebase est adapté pour mettre à jour sa branche de travail après que le master ait évolué.
Le master va continuer à vivre sa vie.



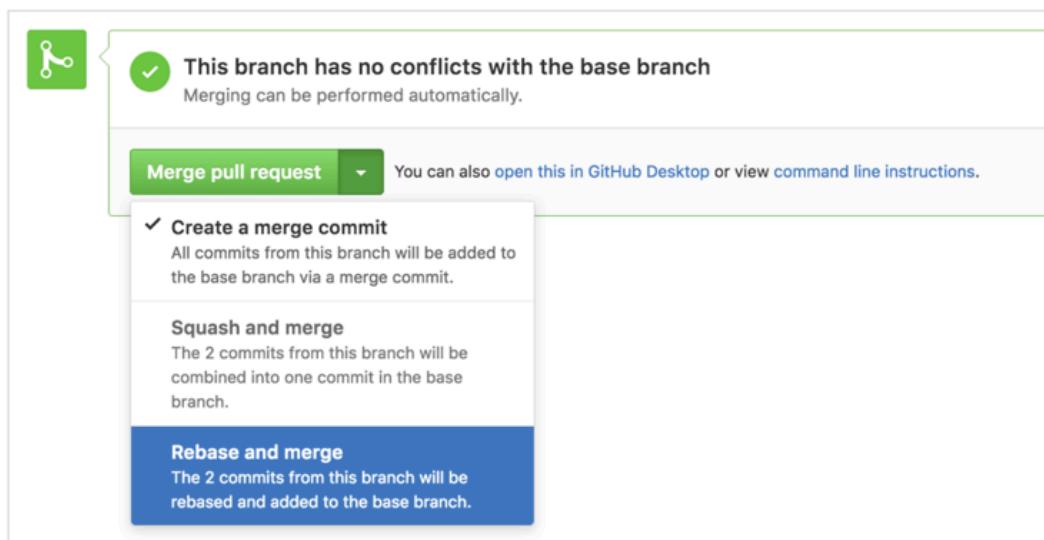
Une feature branch mise à jour avec des merges régulier verra son historique pollué par autant de merge technique.



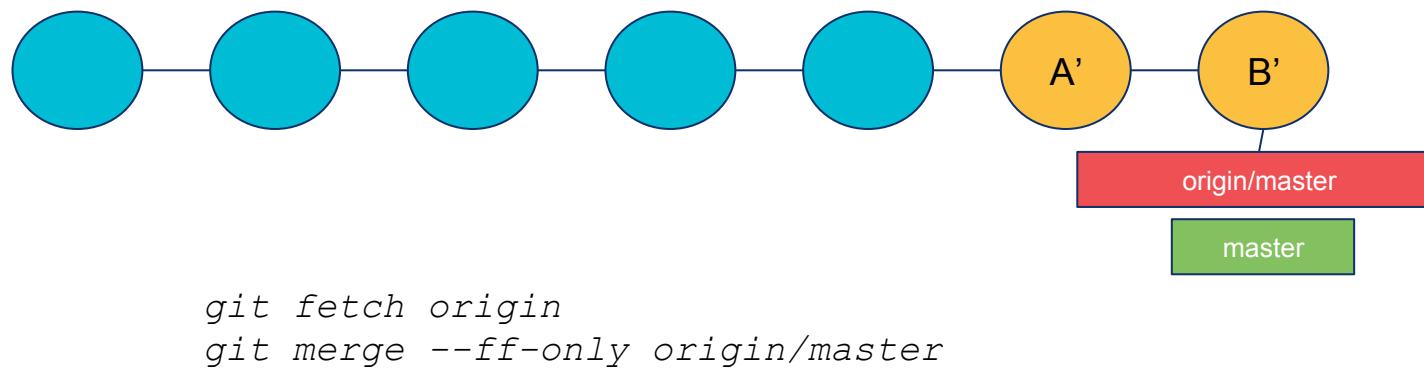
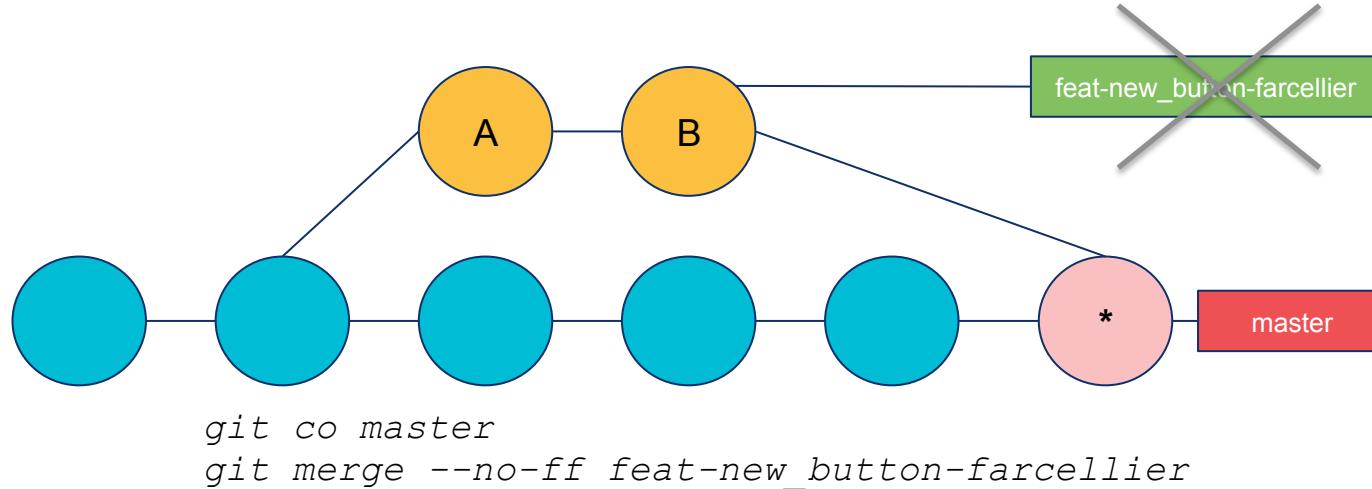
Merge Request d'une feature branch sur le master



Dans le cas d'une merge request / pull request, gitlab effectue un merge "no-fast-forward". Ce merge consolide les 2 branches en un seul commit. La branche de feature peut être supprimée. Le merge "no-fast-forward" est le comportement par défaut de GitLab



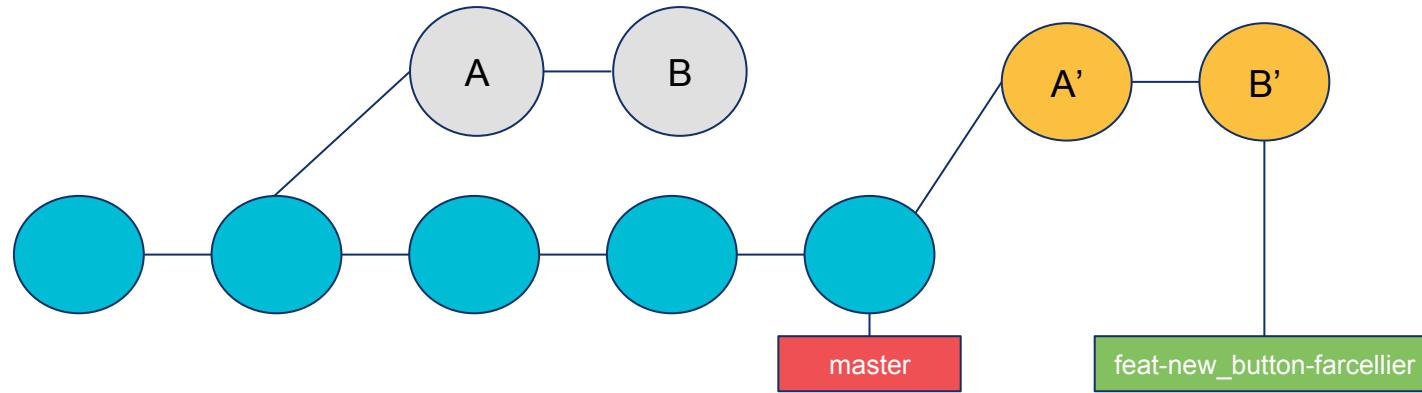
Merge no fast forward et fast forward



Hands-on

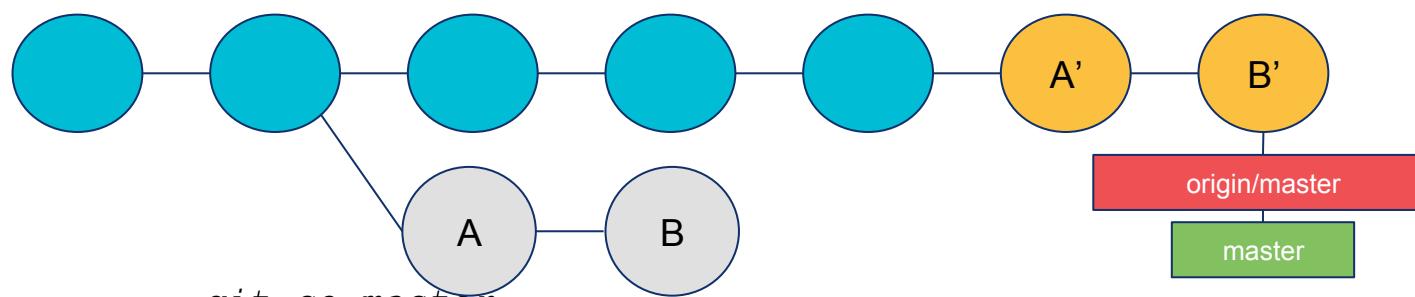
> Quel est l'intérêt des 2 solutions ?

Différence entre git rebase et git merge (fast forward)



```
git co feat-new_button-farcellier  
git rebase master
```

...



```
git co master  
git merge --ff-only feat-new_button-farcellier
```

Hands-on

Quel est la différence entre les 2 commandes ?

Hands-On

git bisect et cherry pick



— Quand utiliser git stash ?

- Vous souhaitez mettre votre travail de côté pour travailler sur une autre branche et le reprendre plus tard.
- J'ai un environnement de travail propre avec un commit initial.
- git stash est le bon candidat pour conserver du travail temporaire, et évite :
 - > git commit -m "my temp feature"
 - > git commit –amend
- *Peut-être remplacé avantageusement par un branche locale temporaire*



git stash

- git stash
 - mettre son espace de staging de côté
- git stash list
 - voir l'ensemble des stashes sauvées
- git stash apply
 - restaurer un stash
- git stash apply stash@{0}
 - pour appliquer uniquement le stash 0
- git stash drop
 - supprimer le dernier stash
- git stash pop
 - correspond à apply+drop

Saved working directory and index
state WIP on master: 77c3d0e first
commit

HEAD is now at 77c3d0e first commit

stash@{0}: WIP on master: 77c3d0e
first commit

stash@{1}: WIP on master: 77c3d0e
first commit

Hands-on

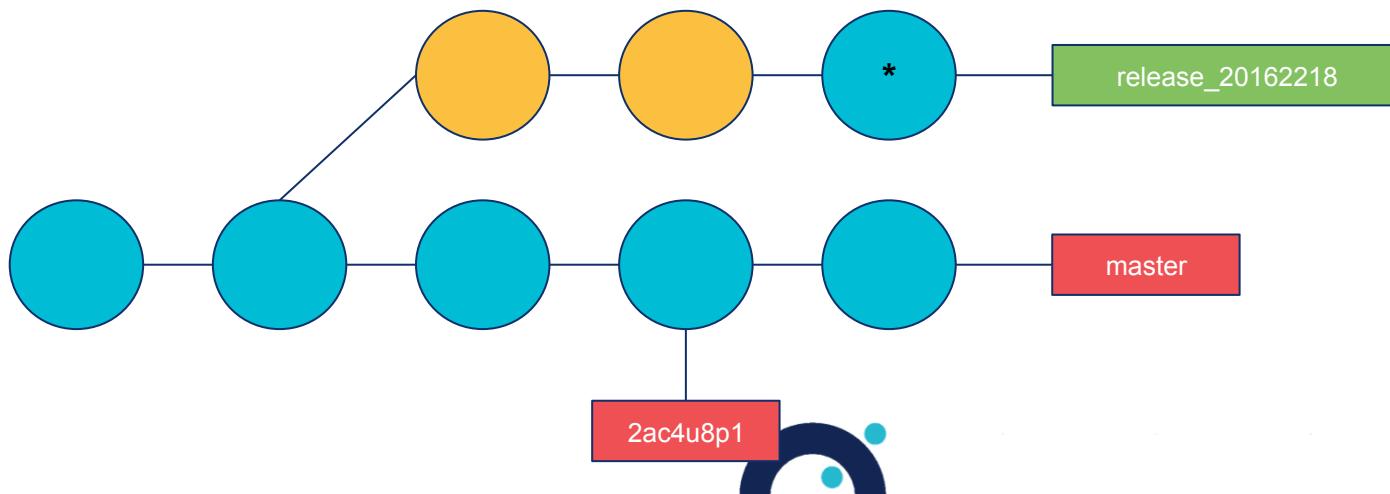
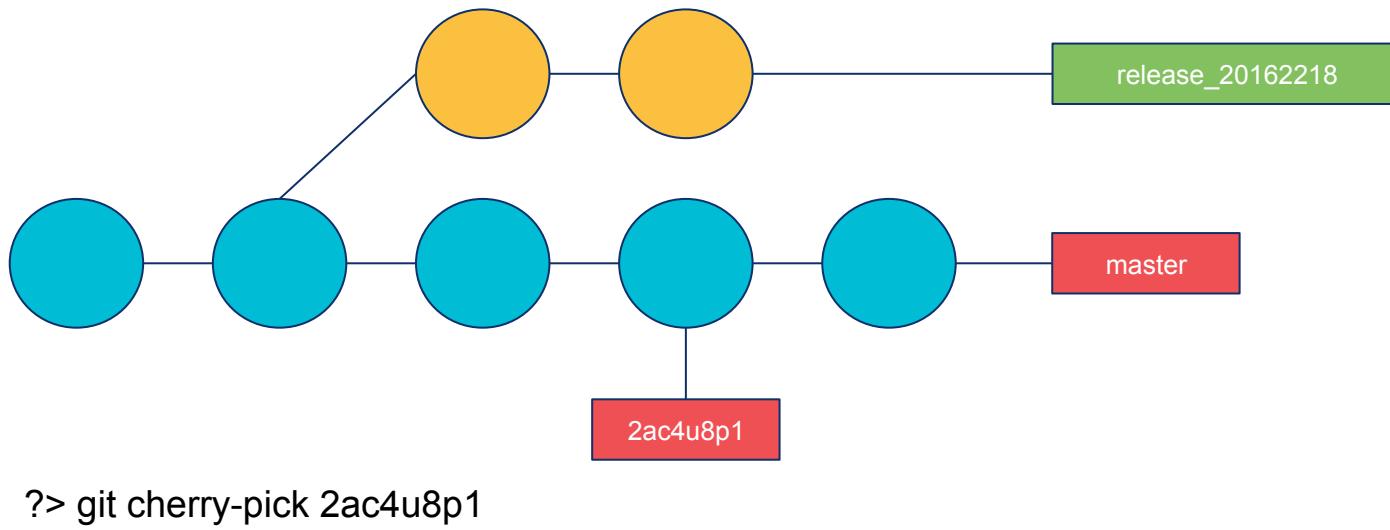
- > Créez un nouveau dépôt en local
- > Ajoutez un fichier “stashme.txt” à votre repo. et l’ajoute au staging
- > Faire un premier commit
- > Ajoutez un autre fichier “stashmeagain.txt” à votre repo. et le stasher.
- > Ajoutez un autre fichier “stashmeagain2.txt” à votre repo. et le stasher.
- > Listez les stashes, puis appliquez le 1er stash.
- > Comment créer une branche avec le 2ème stash ?



Git cherry-pick

- Copie un commit ou des commits d'une branche à une autre

> `git cherry-pick {hash}...{hash n}`



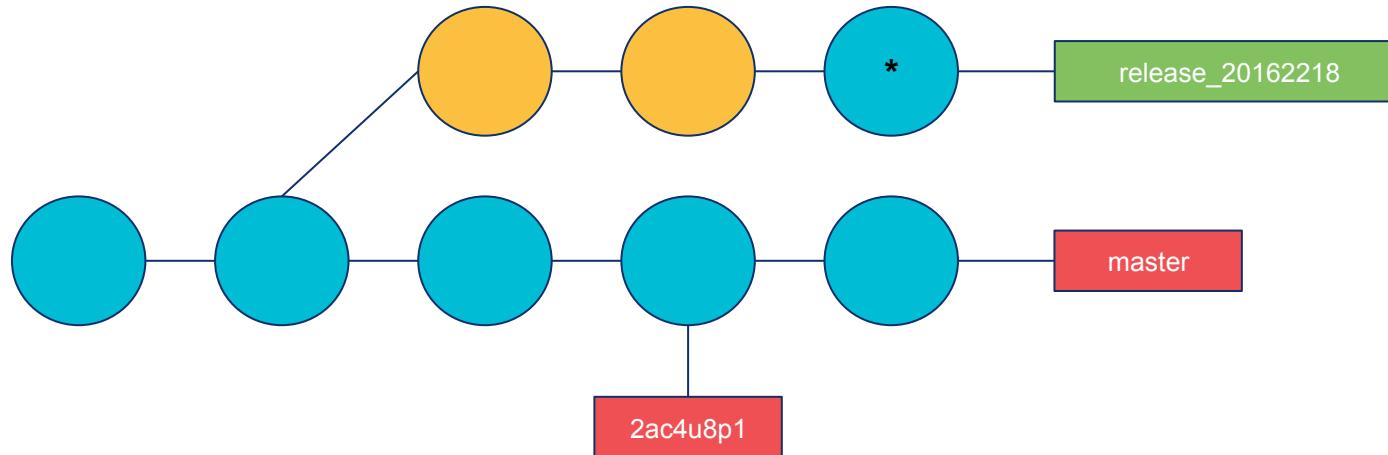
Git cherry-pick

Hands-on

- > Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>
- > Effectuez un cherry pick du commit “*refact: add intellij ide in the gitignore file*” dans la branche *release_20161222*
- > Observez votre log et son graph pour voir l'impact sur le graph
?> `git log --graph --oneline --decorate master release_20161222`

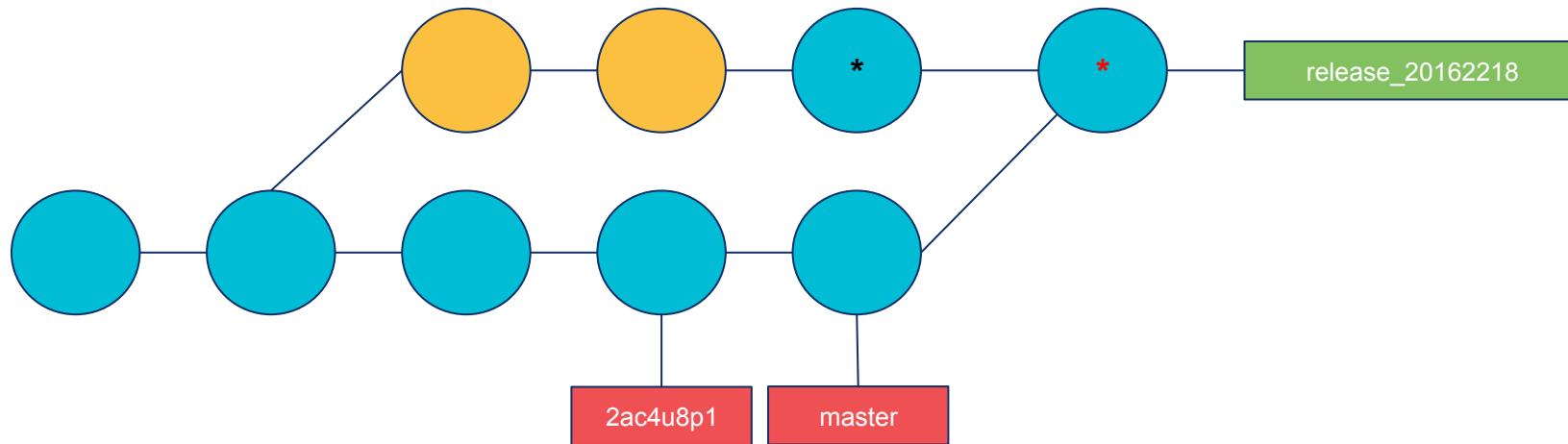


Git cherry-pick



> Lors de quelles opérations l'usage de cherry-pick déclenchera des conflits a posteriori ?

Git cherry-pick



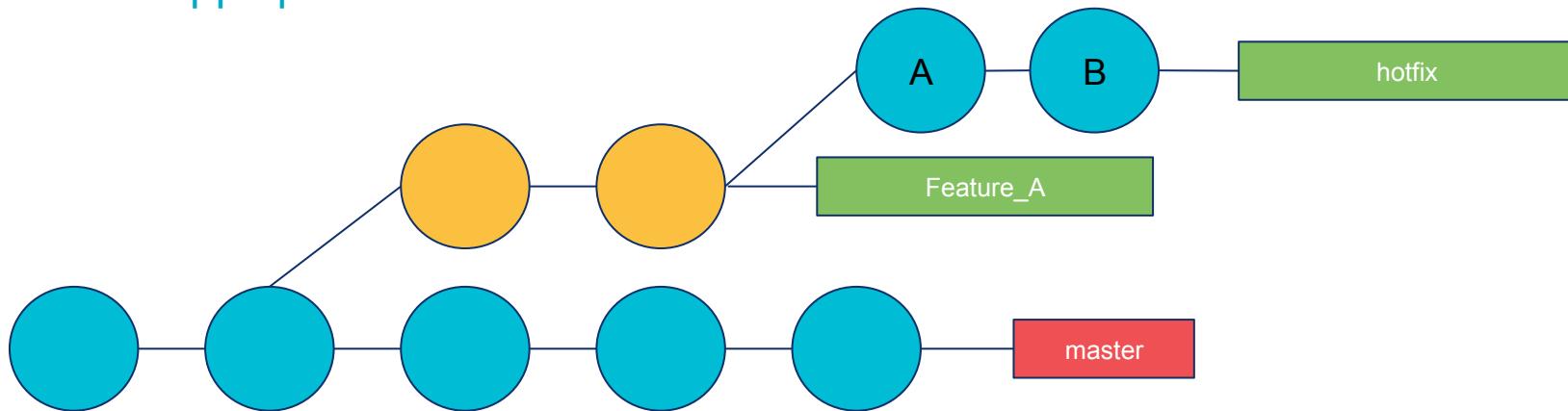
> Lors de quelles opérations, l'usage de cherry-pick déclenchera des conflits a posteriori ?

En cas de merge ou rebase de la branche. Un commit copié à l'aide de la commande cherry-pick déclenchera un conflit. Son usage est inadapté si la branche depuis laquelle nous copions le commit n'est pas indépendante et est voué à être rebasée ou mergée avec la branche cible.

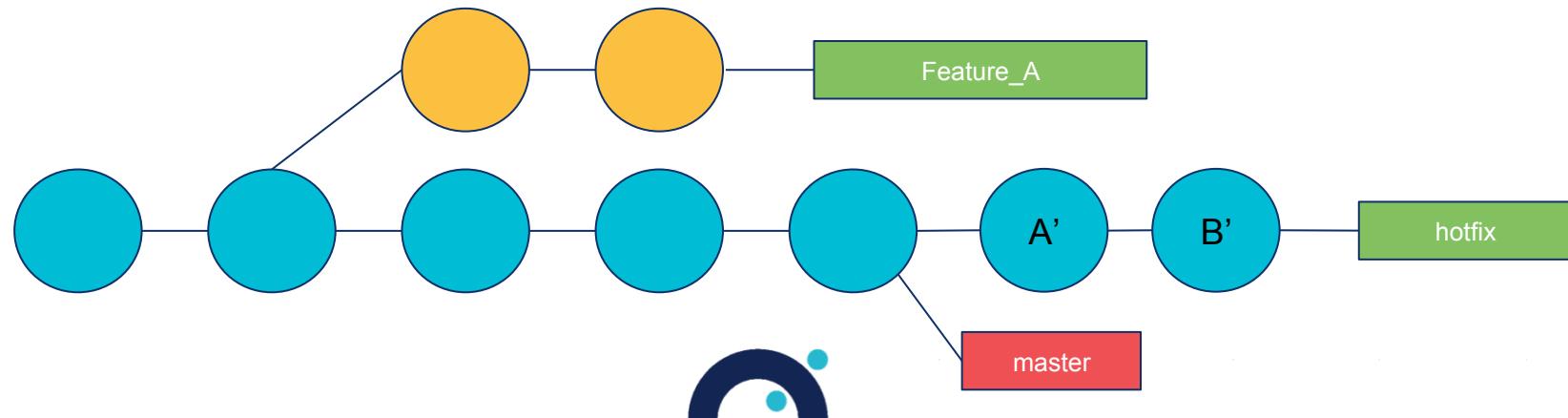


Git rebase onto

Permet d'appliquer un interval de commit sur une branche

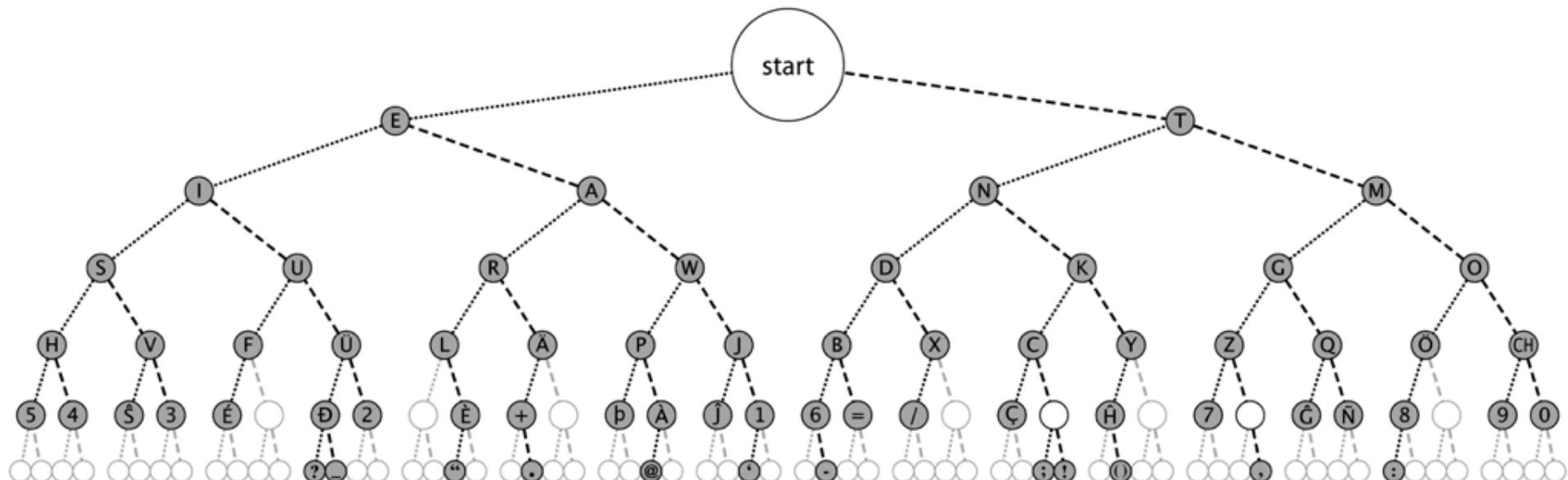


- `git rebase --onto target start what`
- Ici :
 - `git rebase --onto master Feature_A hotfix`



Git bisect

- Pour faire une dicotomie sur vos commit afin de trouver le commit qui a introduit une regression
 - > Manuel : `git bisect start`
 - > Automatique : `git bisect run my_script arguments`



Hands-On

Revenir en arrière



Checkout

- Amener votre environnement sur une version connue
 - > `git checkout {shorthash}`

Lorsque vous revenez à un commit antérieur, votre environnement de travail est placé sur un “DETACHED HEAD”. Si vous commitez du contenu à partir de là, il créera une branche virtuelle référencée nulle part.

Ne commitez pas de contenu après un checkout sur un commit spécifique.

Hands-on

- > Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>
- > Checkoutez sur le commit “feat: marketing requests to improve visual”



Revert

- Construit un nouveau commit qui annule un commit précédent. C'est l'opération à utiliser pour faire disparaître explicitement un commit
 - > `git revert {shorthash}`

Hands-on

- > Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>
- > Revertez le commit “docs: improve readme with contributing rules”

- La commande revert peut annuler n'importe quel commit du dépôt, contrairement à reset qui réinitialise l'historique à proprement parlé
- Pour que la commande fonctionne, votre staging et votre environnement de travail ne doivent pas contenir de modifications en cours



Reset

- Reset permet de jouer sur l'historique. Vous pouvez déplacer le HEAD de votre branche active sur un commit antérieur.

> `git reset {shorthash}`

```
far@far-octo:~/projects/20161220_0813__decathlon/page-web-participative-rebase$ git reset 160ddec
Unstaged changes after reset:
M      .gitignore
M      README.md
M      css/creative.css
M      index.html
```

Hands-on

> Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>

> Resettez votre dépôt sur le commit “remove url from original repository”

Les changements sont remis en zone de staging. Vous pouvez reconstruire pas à pas vos commits.



— Reset - Réinitialiser le dépôt au niveau du dépôt origin

- Vous pouvez déplacer le HEAD de votre branche active au niveau du HEAD de la branche du dépôt origin.
 - > `git reset origin/master`
- Cette opération est très puissante si malgré vos efforts, vous n'arrivez pas à merger proprement vos modifications en cours

Hands-on

> Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>

> Modifiez un fichier, commitez le

> Revenez à l'état de la branche origin/master

Comment revenir à l'état de la branche origin/master en oubliant vos modifications sur votre dépôt local ?

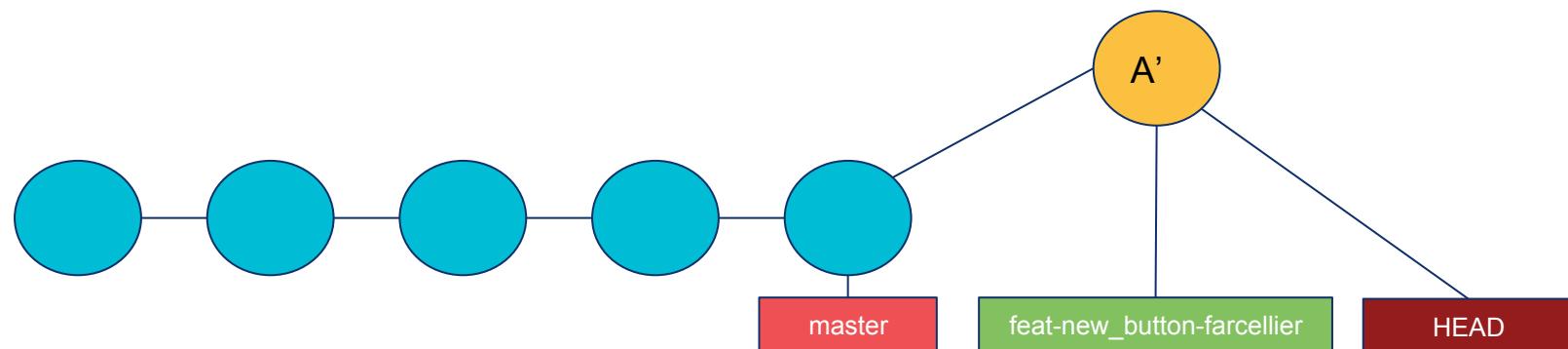
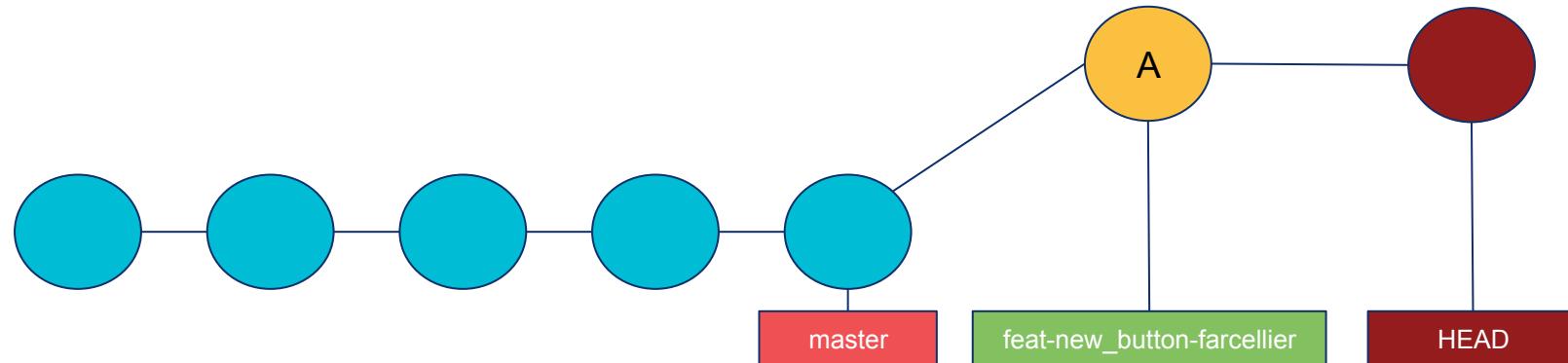


— Ré-écrire l'histoire...



Ré-écrire l'histoire : 'git amend'

Rappel

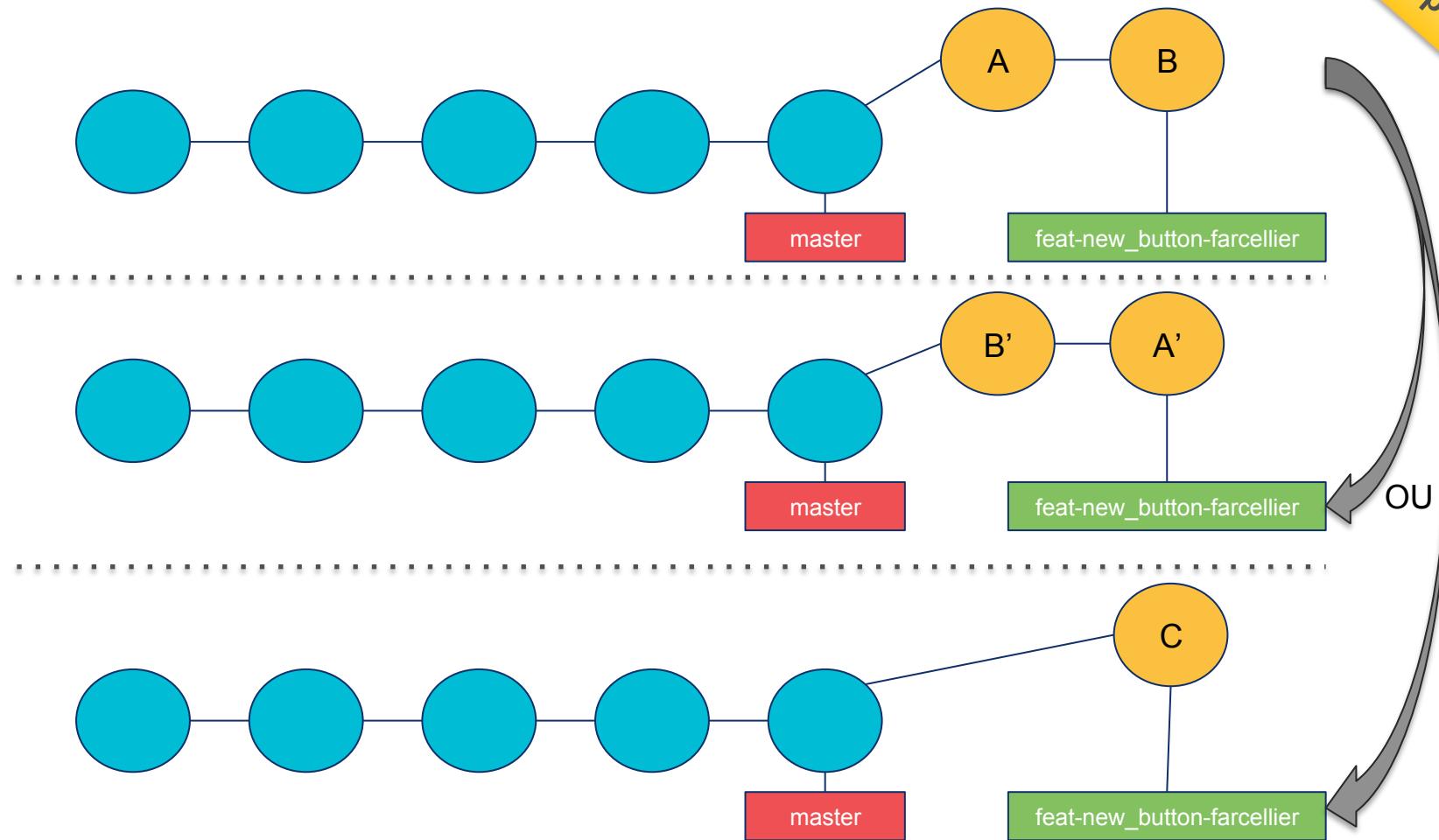


Hands-on

- 'Amendez' le dernier commit avec votre index
 - `git commit --amend`

Ré-écrire l'histoire : Rebase interactif

Rappel



Hands-on

> Fusionnez les 2 derniers commits avec un rebase interactif
git rebase -i {short-hash-du-commit-master}

Refaire son historique : splitter des commit

- Lorsqu'on se rend compte a posteriori qu'un commit est trop gros il est possible de fractionner
 - > Exemple avec le commit 645870b

```
* 241b670 (HEAD -> master, origin/master, origin/HEAD) refact: add intellij ide in the gitignore file
* 645870b feat: marketing requests to improve visual
* fdf9b8b docs: improve readme with contributing rules
| * 200637f (origin/feat-new_button-farcellier) feat: the marketing request we rename the button
| * 0da9a54 feat: the marketing requests we add a button to send an email
```

Hands-on

- Forkez le dépôt <https://gitlab.com/octoformationgit/page-web-participative>
- Faire un rebase interactif sur le commit d'avant
 - *git rebase -i 645870b^*
- Editer le commit 645870b
 - *git rebase -i 645870b^*
- Restaurons l'état initial de ce commit
 - *git reset HEAD~*
- Ajoutons uniquement la première modification du fichier css/creative.css
 - *git add -p css/creative.css*
- Créer le premier commit
 - *git ci -m "feat: Split1"*
- Créer l2 deuxième avec les autres changements
 - *git ci -am "feat: Split2"*
- Terminer le rebase
 - *git rebase --continue*
- Visualiser le changement
 - *git log --graph --oneline --decorate origin/master origin/feat-new_button-farcellier master*

— Refaire l'histoire : supprimer un fichier de tout l'historique

- Tous les commit concernés par le fichier seront ré-écrits
- Pensez à l'exclure ensuite avec .gitignore

Hands-on

- Clonez le dépôt <https://gitlab.com/octoformationgit/page-web-participative-rebase>
- Tracez le graph des commits
 - `git log --graph --oneline --decorate origin/master origin/feat-new_button-farcellier master`
- Supprimer toute trace du fichier
 - `git filter-branch --force --index-filter 'git rm --cached --ignore-unmatchindex.html' '\--prune-empty --tag-name-filter cat -- --all'`
- Tracez le graph des commits
 - `git log --graph --oneline --decorate origin/master origin/feat-new_button-farcellier master`
- Que constatons-nous ?



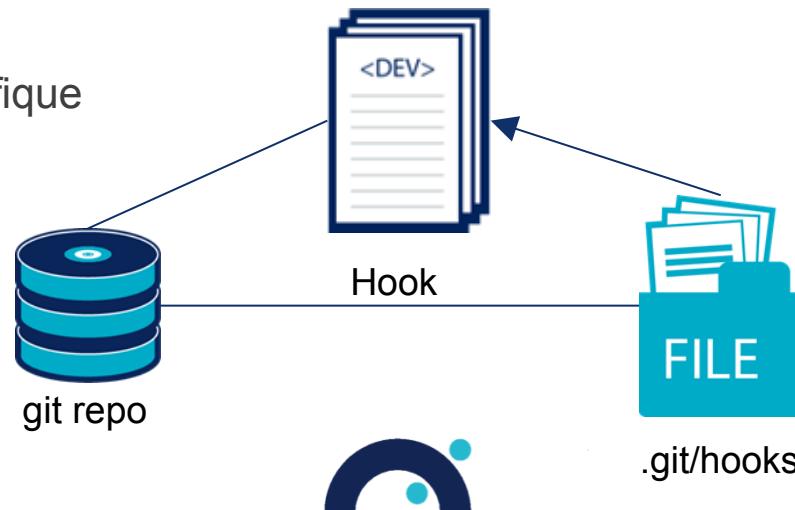
Hands-On

git hook



— Un hook pourquoi faire ?

- Principe d'hollywood : *Ne nous appelez pas, c'est nous qui vous appellerons !*
- Un hook est un script qui s'exécute dès qu'un événement particulier se produit dans un repo. git.
 - > Dans gitlab, ce sont les webhooks qui sont responsables de lancer une action.
- Cas d'utilisation :
 - > hooks côté serveur
 - + mise en place d'une stratégie de commit
 - + workflows d'intégration continue
 - + publication des notifications sur un channel slack ou hipchat
 - > hooks en local
 - + script spécifique



— Les types de hooks

Locaux :

- pre-commit
 - > avant chaque git commit
 - prepare-commit-msg
 - > pour indiquer un message de commit dans l'éditeur de texte
 - commit-msg
 - > après que l'utilisateur a saisi un message de commit
 - post-commit
 - > utilisé à des fins de notification
 - post-checkout
 - > à chaque git checkout, utile pour faire le ménage sur notre solution
 - pre-rebase
 - > avant que git rebase n'apporte le moindre changement
- Des hooks côté serveur existent : pre-receive, update, post-receive...



Jouons avec les hooks

- Langage de scripts :

- > Shell
- > Perl
- > Python
- > ...

- Installer un hook :

- > .git/hooks

Hands-on

> Clonez le repos https://gitlab.com/octoformationgit/template_java

> Ecrivez un hook qui aura pour but d'effectuer d'exécuter la commande “mvn clean test” avant chaque commit

exemple : <https://gist.github.com/arnobroekhof/9454645>

> Ajoutez un test qui casse et essayez de commiter



— Et voilà...

