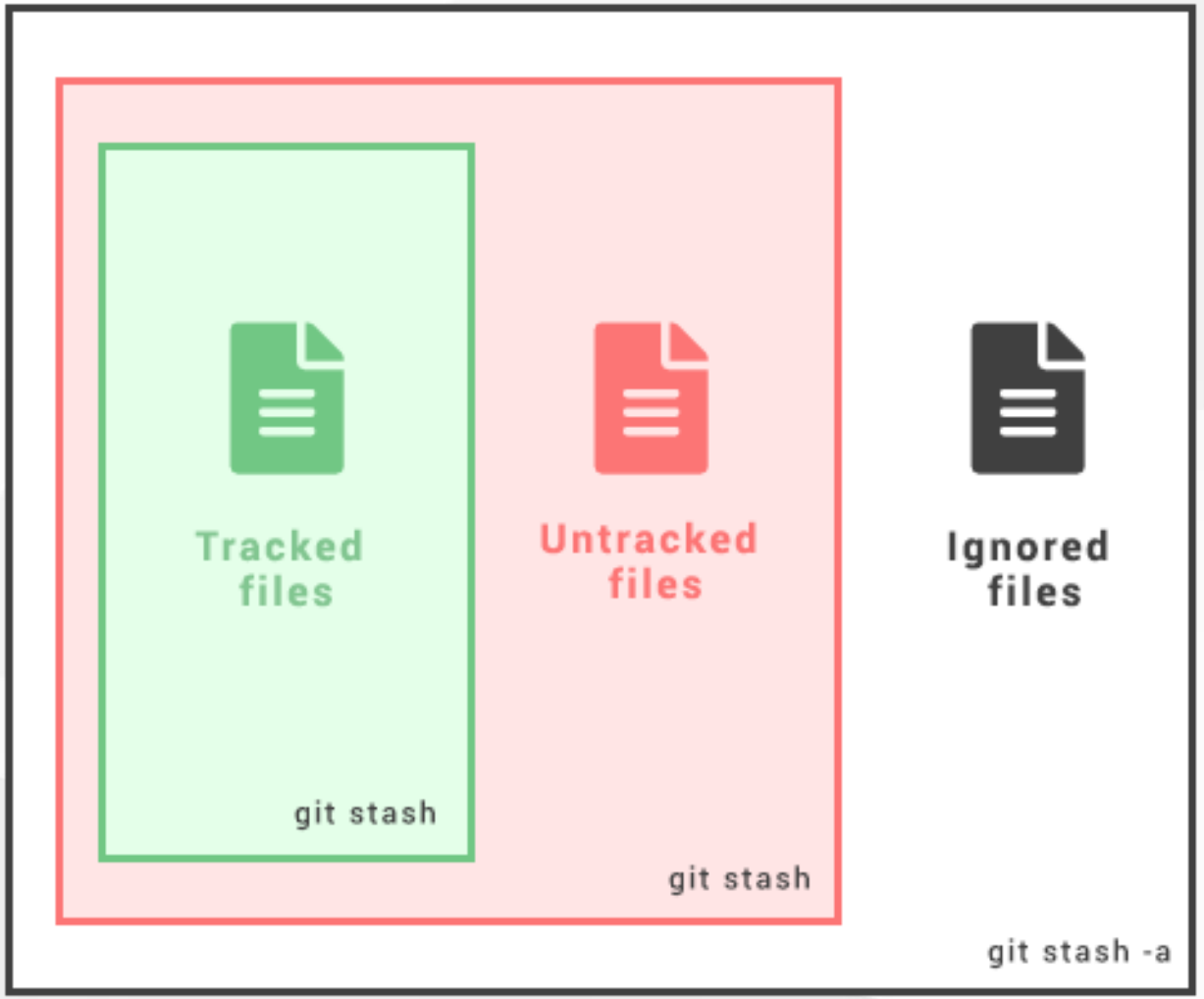
Git

git stash


Tracked files
Untracked files
Ignored files
git stash
git stash
git stash -a

git stash

La commmande `git stash` prend vos changements de non-comité (stagés et non stagés), les enregistre pour une utilisation utlérieure, puis les remplace dans votre copie de travail.

Notez que le stash est local pour votre dépôt Git. Les stashs ne sont pas transférés au serveur lors d'un push

```
$ git status
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html

$ git stash
Saved working directory and index state WIP on main: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage

$ git status
On branch main
nothing to commit, working tree clean
```

Appliquer à nouveau vos changements stashés

Vous pouvez réappliquer vos changements stashés en utilisant la commande: `git stash pop`.
L'apparition de votre stash **supprime** les changements qu'il contient et les réapplique à votre copie de travail.

Vous pouvez également réappliquer les changements à votre copie de travail et les **conserver** dans votre stash avec la commande:
`git stash apply`

git stash pop

```
$ git status
On branch main
nothing to commit, working tree clean
$ git stash pop
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

git stash apply

```
$ git stash apply
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html
```

> C'est utile si vous souhaitez appliquer les mêmes changements stashés à plusieurs branches.

Faire un stash des fichiers non trackés ou ignorés

Par défaut, la commande `git stash` ignore les fichiers non trackés et les fichiers ignorés.

Il faut ajouter l'option `-u` (ou `--include-untracked`) pour que la commande `git stash` inclue les fichiers non trackés.

Vous pouvez aussi ajouter l'option `-a` (ou `--all`) pour que la commande `git stash` inclue tous les fichiers.

Gérer plusieurs stashes

Pour consulter la liste des stashes, utilisez la commande `git stash list`.

Pour donner plus de contexte, il peut être utile d'annoter vos stashes avec une descripton comme ceci: `git stash save "My stash"`

Par défaut, `git stash pop` réapplique les changements du dernier stash (`stash@{0}`).
Vous pouvez utiliser la commande `git stash apply stash@{n}` pour réappliquer le stash n.

Nettoyez votre stash

Si vous décidez que vous n'avez plus besoin d'un stash spécifique, vous pouvez le supprimer avec la commande `git stash drop`.

```
$ git stash drop stash@{1}
Dropped stash@{1} (17e2697fd8251df6163117cb3d58c1f62a5e7cdb)
```

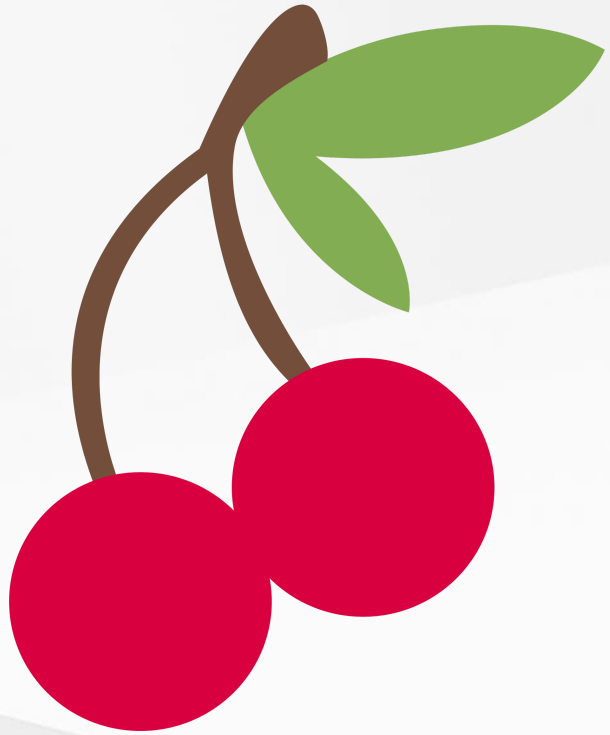
Vous pouvez également supprimer tous les stashes avec la commande `git stash drop --all` ou `git stash clear`.

Affichage des comparaisons entre stashes

Vous pouvez afficher les différences entre deux stashes avec la commande `git stash show stash@{n}` ou `git stash show stash@{n} stash@{m}`.

Ou ajoutez l'option `-p` (ou `--patch`) pour afficher les différences entre deux stashes en format patch.

git cherry-pick



git cherry-pick

La commande `git cherry-pick` permet de récupérer les changements d'une branche à l'autre.

Et pour annuler un cherry-pick, utilisez la commande `git cherry-pick --abort`.

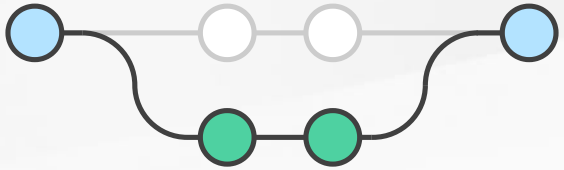
```
a - b - c - d   Main
     \
      e - f - g  Feature
```

```
$ git switch main
$ git cherry-pick f
```

Nous avons récupéré le commit `f` de la branche `Feature` et l'ajouté à la branche `Main`.

```
a - b - c - d - f   Main
     \
      e - f - g  Feature
```

git merge



git merge

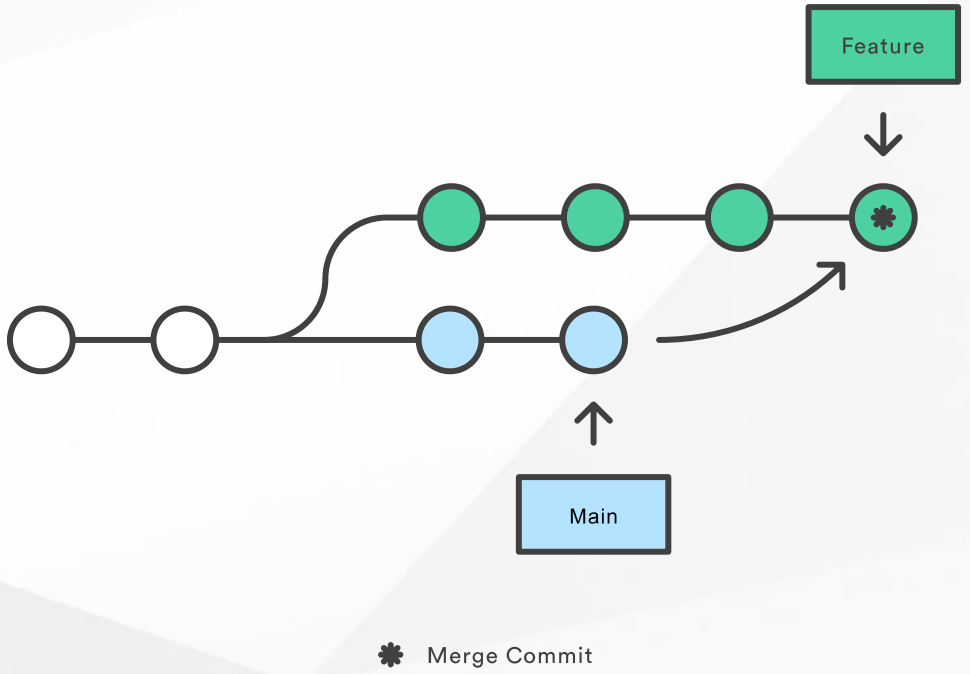
La solution la plus simple consiste à faire un merge de la branche principal (main) dans la branch feature.

```
$ git switch feature
$ git merge main
```

Vous pouvez également condenser cette commande en une ligne:

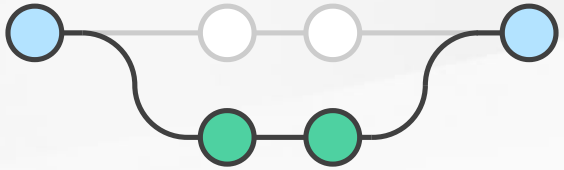
```
$ git merge feature main
```

Merging main into the feature branch



Le merge est une opération intéressante, car elle est non destructive. Les branches existantes ne sont pas modifiées. Cela permet d'éviter les pièges potentiels du rebase.

git rebase



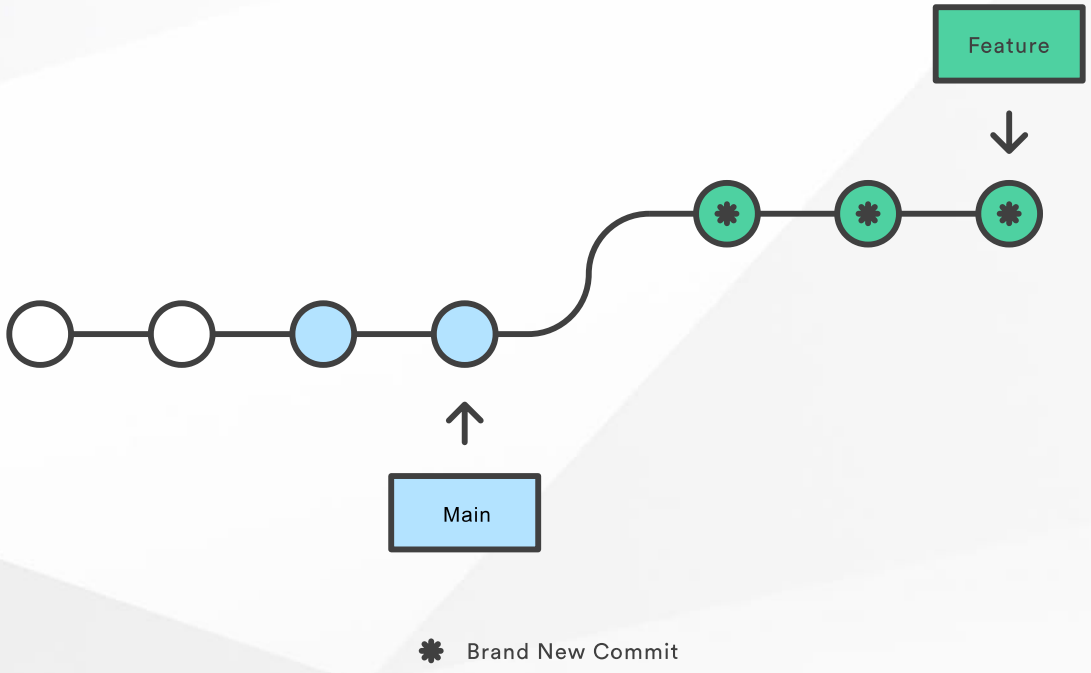
git rebase

Plutôt que de faire un merge, vous pouvez faire un rebase de la branche de feature vers la branche main.

```
$ git checkout feature
$ git rebase main
```

Toute la branche de feature sera ainsi déplacée sur la pointe de la branche main, et tous les nouveaux commits seront intégrés à la branche main.

Rebasing the feature branch onto main



Cependant, au lieu d'utiliser un commit de merge, le rebase consiste à réécrire l'historique du projet en créant de nouveaux commmits pour chaque commit de la branche d'origine.

Avantages

Le principal avantage du rebase est que l'historique de votre projet sera nettement plus propre.

Premièrement, cette opération permet de supprimer les commits de merge superflus requis par la commande git merge.

Deuxièmement, comme l'illustre le schéma ci-dessus, vous obtenez un historique de projet parfaitement linéaire, qui vous permettra de suivre la pointe de la branche de fonctionnalité à toutes les étapes du projet à partir de son commencement.

Rebase interactif

Le rebase interactif vous donne la possibilité de modifier des commits lorsqu'ils sont déplacés vers la nouvelle branche.

Cette opération est plus efficace qu'un rebase automatique, puisqu'elle permet de contrôler l'intégralité de l'historique des commits de la branche.

Le plus souvent, le rebase interactif est utilisé pour nettoyer un historique désordonné avant de faire un merge d'une branche de fonctionnalité dans main.

Pour lancer une session de rebase interactif, ajoutez l'option i à la commande git rebase :

```
$ git checkout feature
$ git rebase -i main
```

Cette commande ouvre un éditeur de texte répertoriant tous les commits qui seront déplacés.

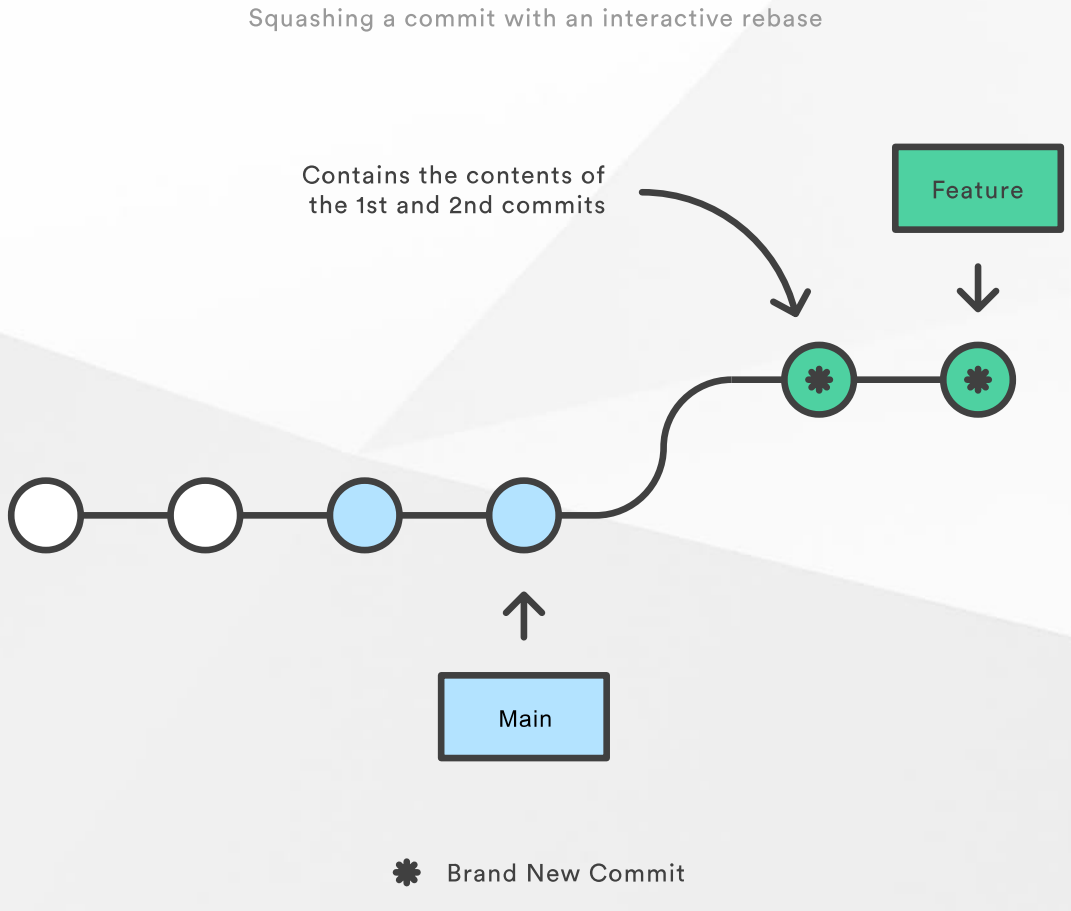
```
pick 33d5b7a Message for commit #1
pick 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

Cette liste détermine précisément quelle sera la structure de la branche après le rebase. En modifiant la commande pick et/ou en réorganisant les entrées, vous pouvez agencer l'historique de la branche comme bon vous semble.

Par exemple, si le deuxième commit résout un petit problème dans le premier, vous pouvez les merger en un seul grâce à la commande fixup :

```
pick 33d5b7a Message for commit #1
fixup 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

Lorsque vous enregistrez et fermez le fichier, Git effectue le rebase conformément à vos instructions et vous obtenez un historique de projet semblable au suivant :

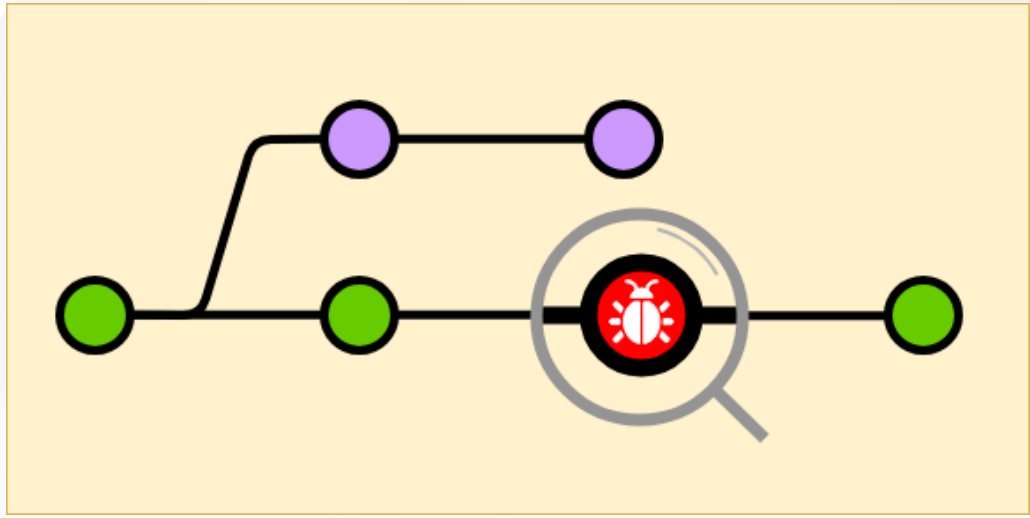


Force-push

Si vous tentez de pusher à nouveau la branche principale (main) rebasé vers un dépôt distant, Git vous en empêchera, car elle entre en conflit avec la branche principale (main) distante. Vous pouvez toutefois forcer ce push en utilisant l'argument --force :

```
# Soyez très prudent avec cete commande!
$ git push --force
```

git bisect



git bisect

La commande `git bisect` permet de déterminer quel commit pécis a causé une anomalie au moyen d'une recherche dichotomique.

En pratique, il faut donner à Git un minimum de deux commits, et il vous permettra de déterminer quel commit a causé l'anomalie.

- le dernier commit où l'anomalie n'est pas présente
- un commit où il a été remarqué que ça ne fonctionne plus, c'est-à-dire que l'anomalie que vous cherchez est apparue (souvent le commit actuel, **HEAD**)

Démarrer la "bisection"

```
## Démarrage de la "bisection"
$ git bisect start

## Donner à git un commit là où il n'y a pas de bug
$ git bisect good <commit>

## Donner à git un commit là où il y a un bug
$ git bisect bad <commit>
```

Ou en une seule ligne :

```
$ git bisect start HEAD <commit>
```

Déterminer si le commit contient l'anomalie

Maintenant que git nous a mis le commit qui n'a pas l'anomalie, nous pouvons tester si le commit contient l'anomalie.

Nous déterminons manuellement si le commit contient l'anomalie (bab) ou non (good), par exemple via l'exécution de tests automatisés ou bien via un test manuel de l'application.

```
$ git bisect good # ou
$ git bisect bad
```

Et répéter, jusqu'à trouver le commmit fautif.

Terminer

Git a donc déterminé que le commit *f5d930f* contient l'anomalie.

Pour retourner à un état normal, nous terminons la procédure par:

```
$ git bisect reset
```

Pour aller plus loin, regarder la documentation

```
## déterminer automatiquement si le commit est bon ou pas
$ git bisect run ./test.sh

## restreindre la recherche (seuls les commits affectant ce chemain seront évalués)
$ git bisect start – src/main/java/
```

Référence

- Documentation officielle de Git
  - Git - Documentation (en)
- Atlassian
  - Devenez un expert de Git
  - Become a git guru (en)
- Grafikart
- GitLab
  - Git
  - Git Cheat Sheet