



La gestion de versions avec Git

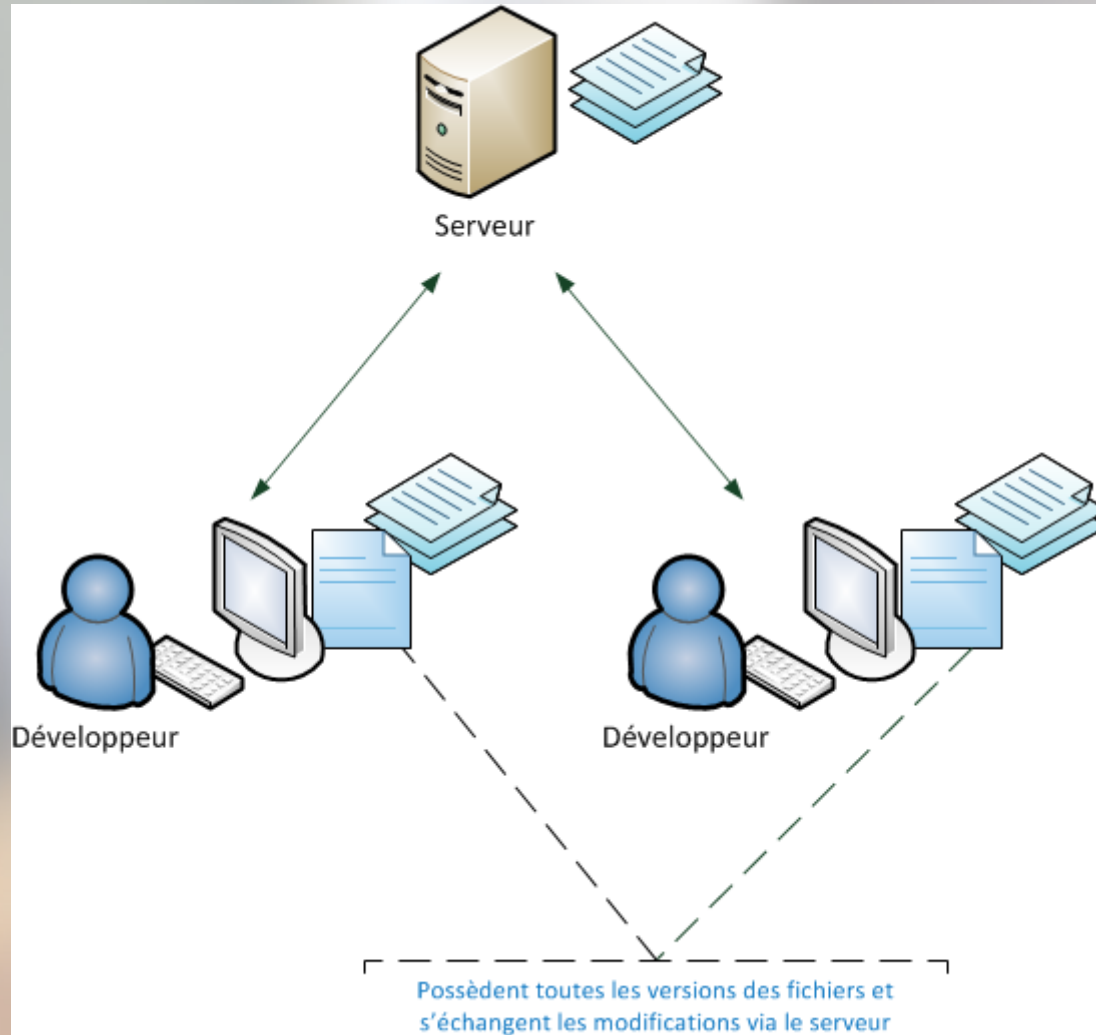
Formation avancée

Au programme

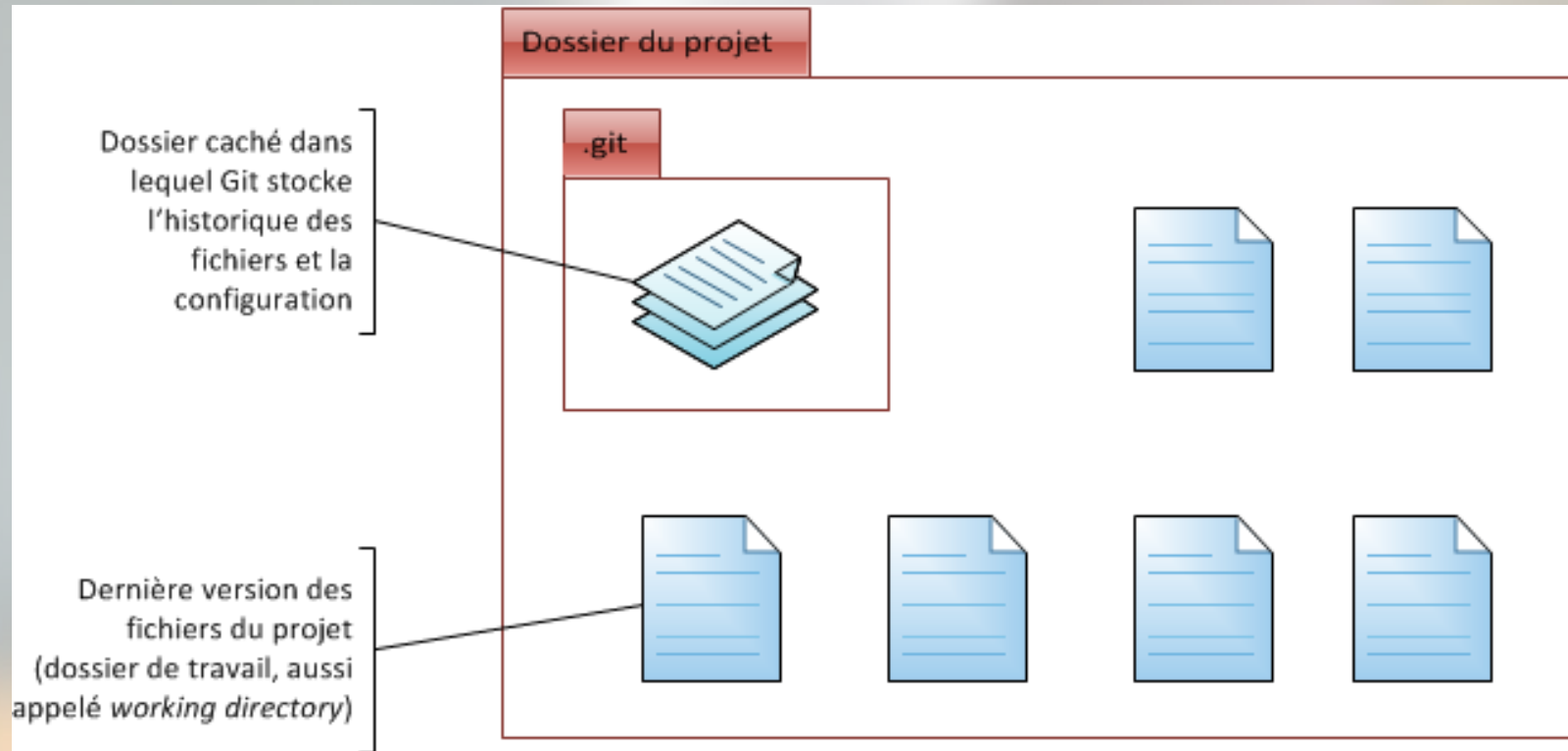
- Rappels
- Un modèle de branches efficace
- Merge vs Rebase
- Résoudre les conflits
- Modifier son historique
- Travaux Pratiques

Rappels

Rappels – Architecture Git



Rappels – Dépôt Git



Rappels – Lexique

Repository : C'est le lieu contenant les fichiers du projet, avec tout son historique. Chaque développeur du projet a son Repository local. Il y a aussi un Repository nommé *Origin* faisant office de serveur, qui est un Repository distant (*remote*).

Commit : un commit est un ensemble de modifications apportées et validées. Avec Git on commit d'abord sur son ordinateur, et non sur un serveur.

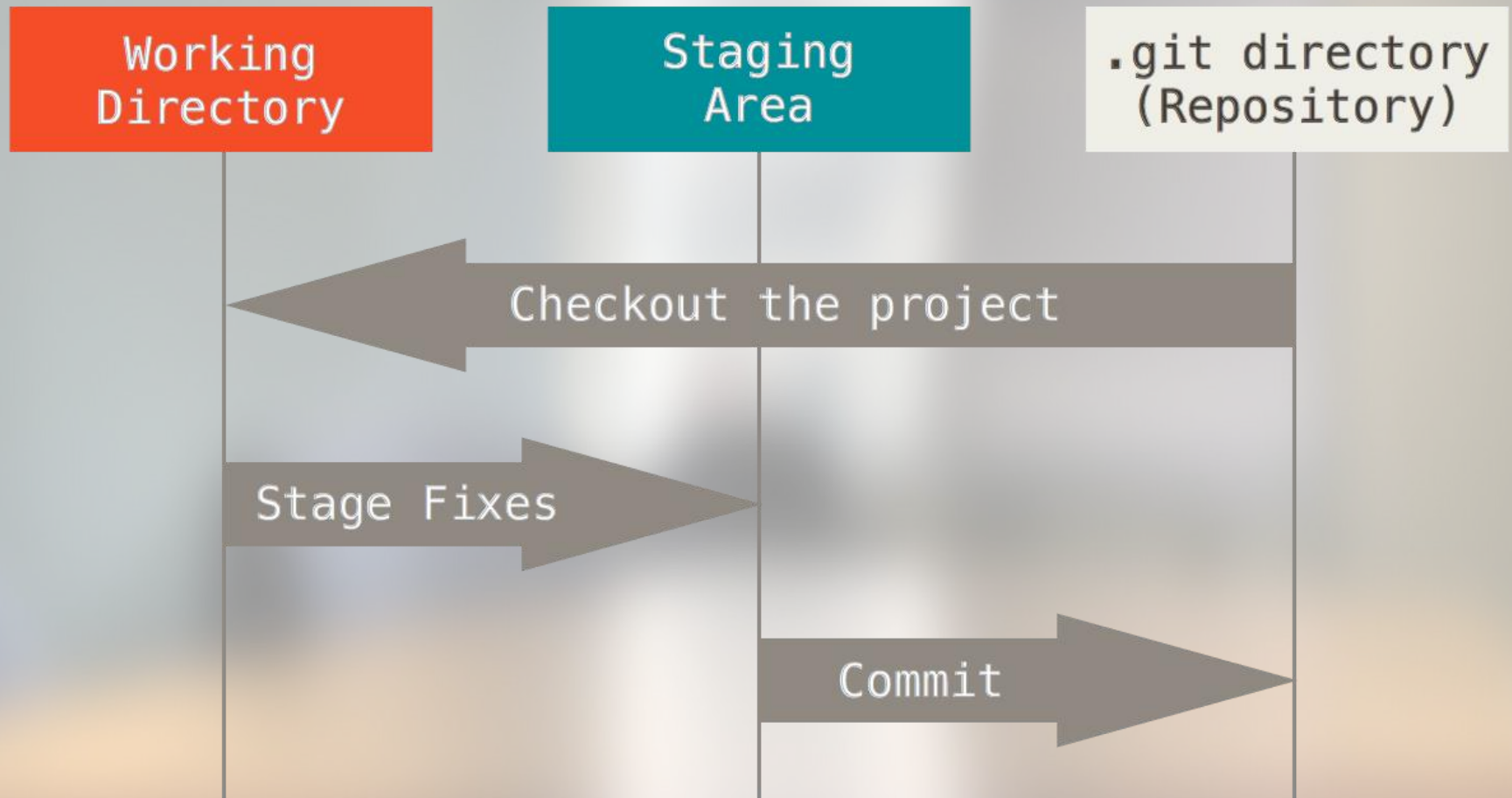
Historique : Permet de retrouver toutes les modifications apportées au projet. Il est construit à partir de la liste des commits.

Tag : Un tag est un pointeur vers un commit spécifique. Il est en général utilisé pour marquer les releases.

Branche : Les branches décrivent un développement en parallèle de votre projet, une dérivation qui va résulter à l'ajout de nouvelles fonctionnalités. La branche principale est la branche appelée *Master*.

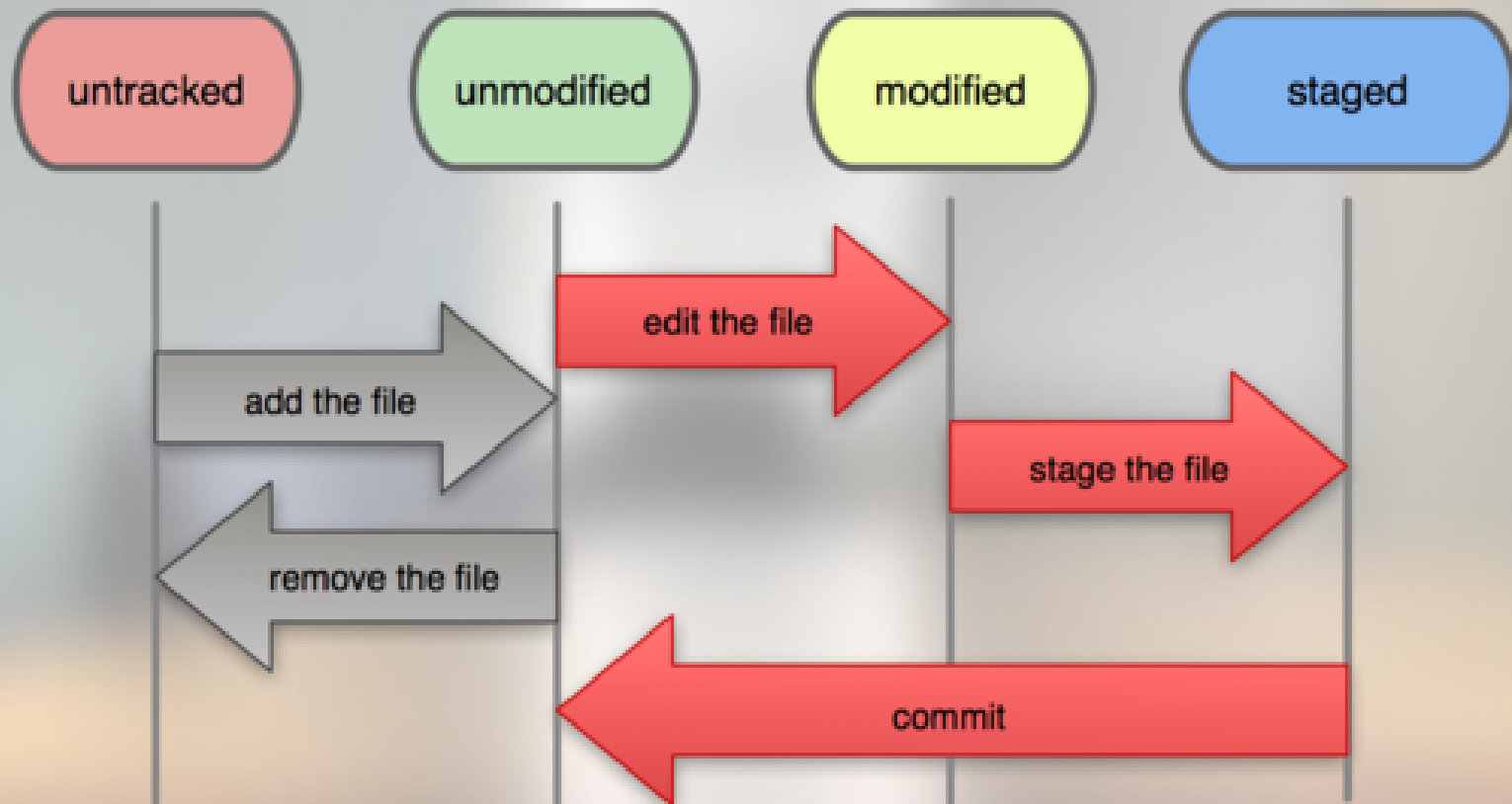
HEAD : La référence HEAD pointe vers le commit qui sera le parent du prochain commit.

Rappels – Staging Area

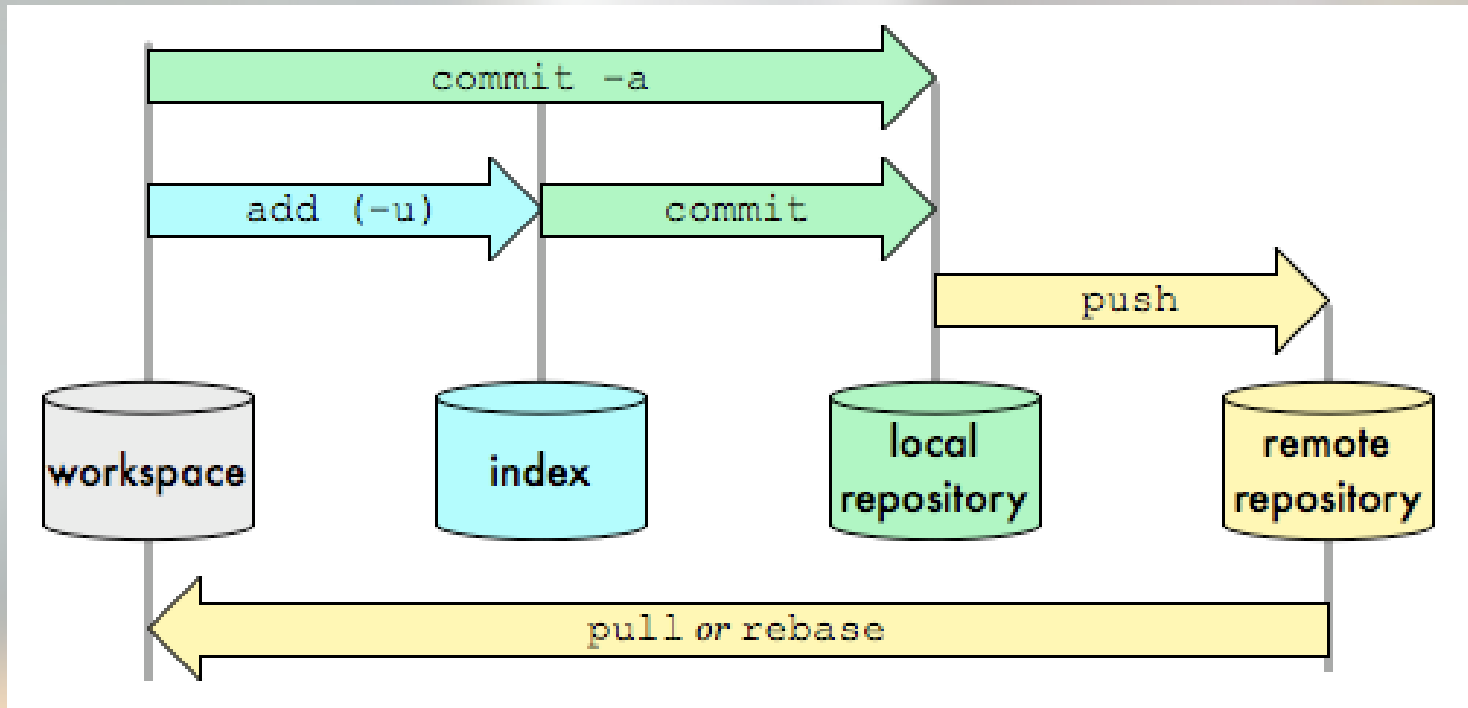


Rappels – Statuts des fichiers

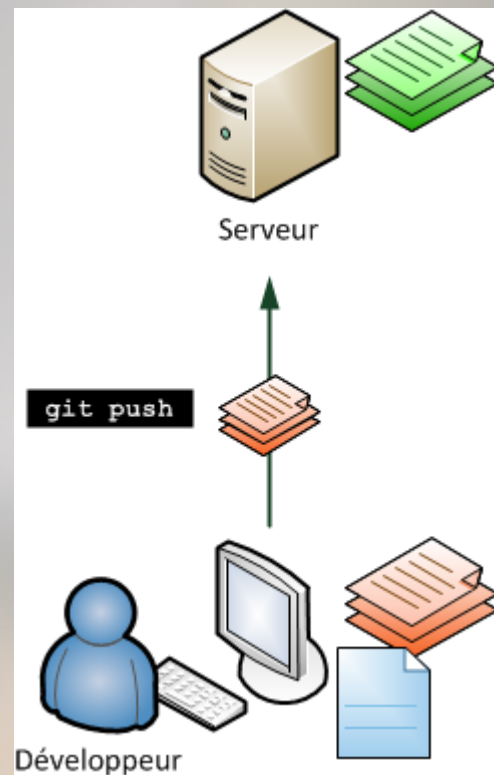
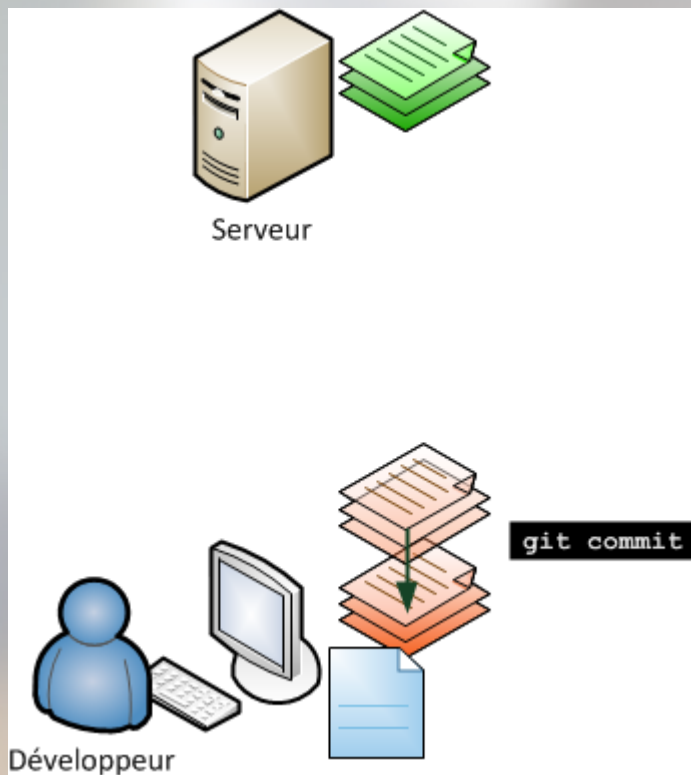
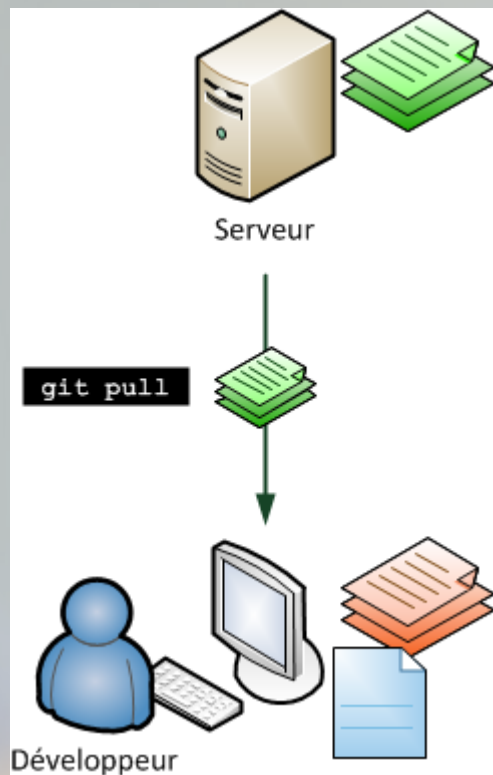
File Status Lifecycle



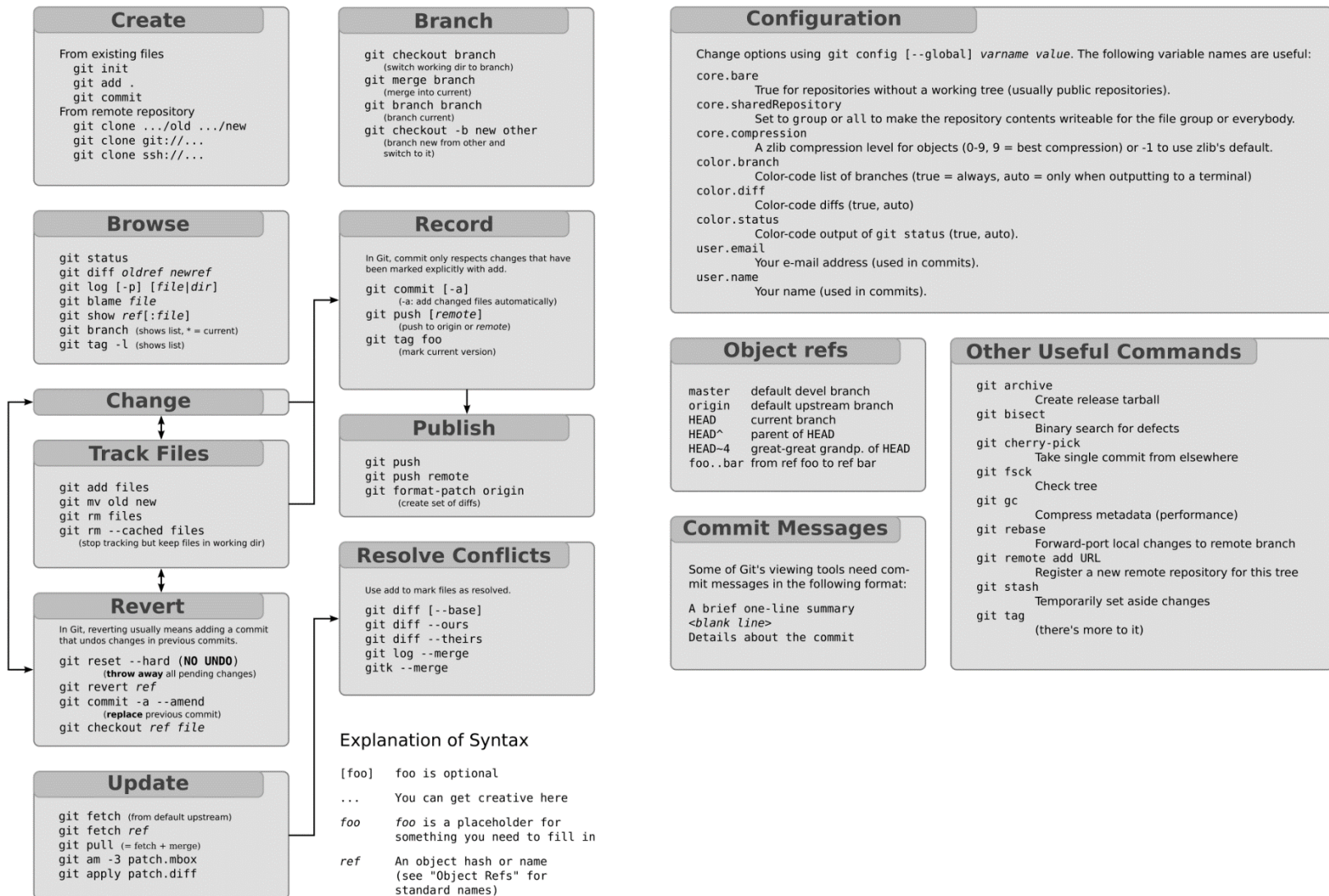
Rappels – Données Git



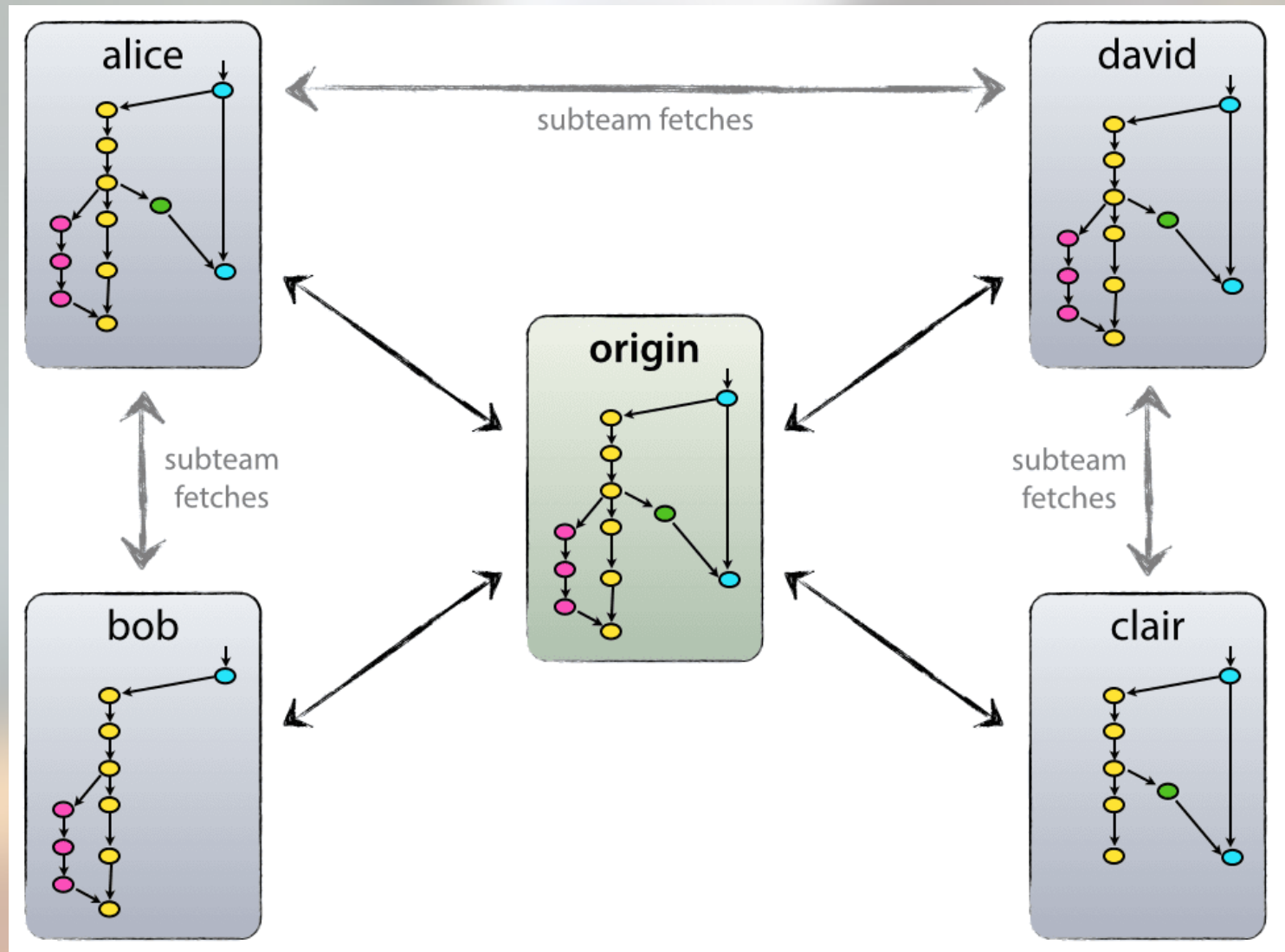
Rappels – Commandes principales



Rappels – Autres commandes



Un modèle de branches efficace



Les branches principales :

- **master**
- **develop**

origin/master est la branche principale où le code source de HEAD reflète l'état prêt à être déployé en production.

origin/develop est la branche principale où le code source de HEAD reflète les derniers changements livrés pour la prochaine version. Certains l'appelleraient « branche d'intégration ». C'est à partir de cet emplacement que sont compilées les versions quotidiennes.

Les branches de support:

- les branches pour les **fonctionnalités**
- les branches pour les **versions**
- les branches de **correctifs**

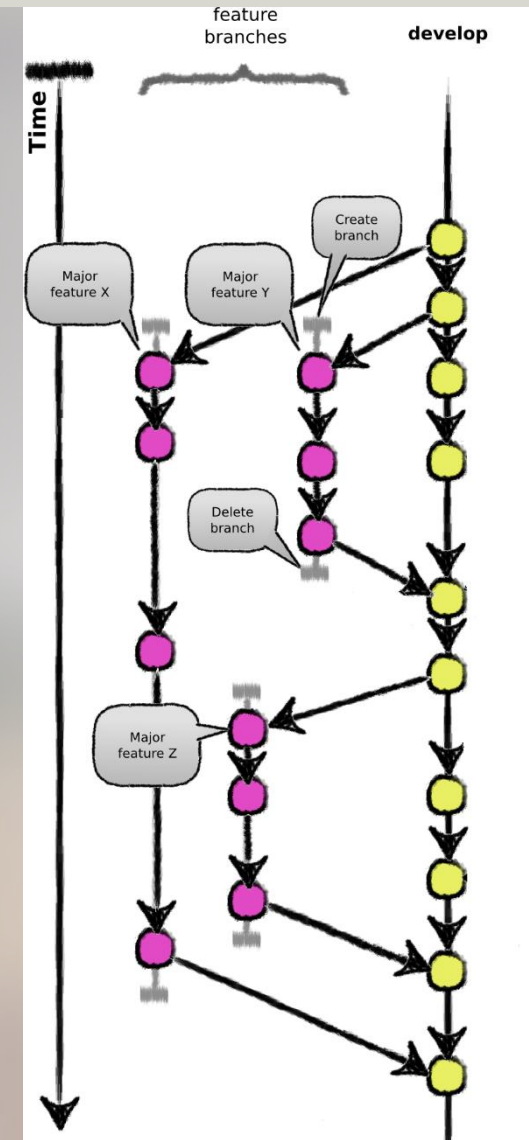
Contrairement aux branches principales, ces branches ont toujours une durée de vie limitée, puisqu'elles seront effacées au final.

Les branches de **fonctionnalité**

Peuvent provenir de :
develop

Doivent être fusionnées dans :
develop

Convention de nommage de la branche :
feature/*



Les branches de **fonctionnalité**

```
$ git checkout -b feature/myFeature develop
```

Développement de la fonctionnalité (commits)

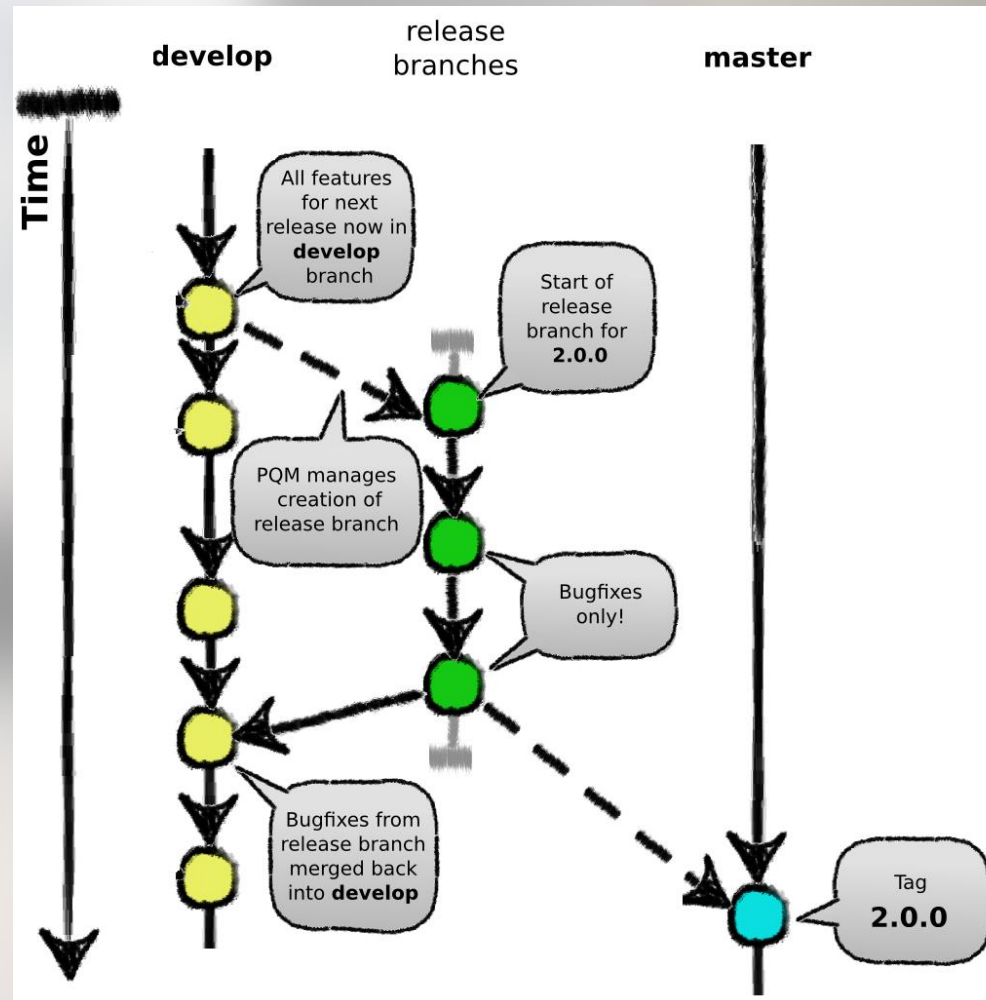
```
$ git checkout develop  
$ git merge --no-ff feature/myFeature  
$ git push origin develop  
$ git branch -d feature/myFeature
```

Les branches de **version**

Peuvent provenir de :
develop

Doivent être fusionnées dans :
develop et **master**

Convention de nommage de la branche :
release/*



Les branches de **version**

```
$ git checkout -b release/v1.2 develop
```

Mise à jour des scripts/documentation (commits)

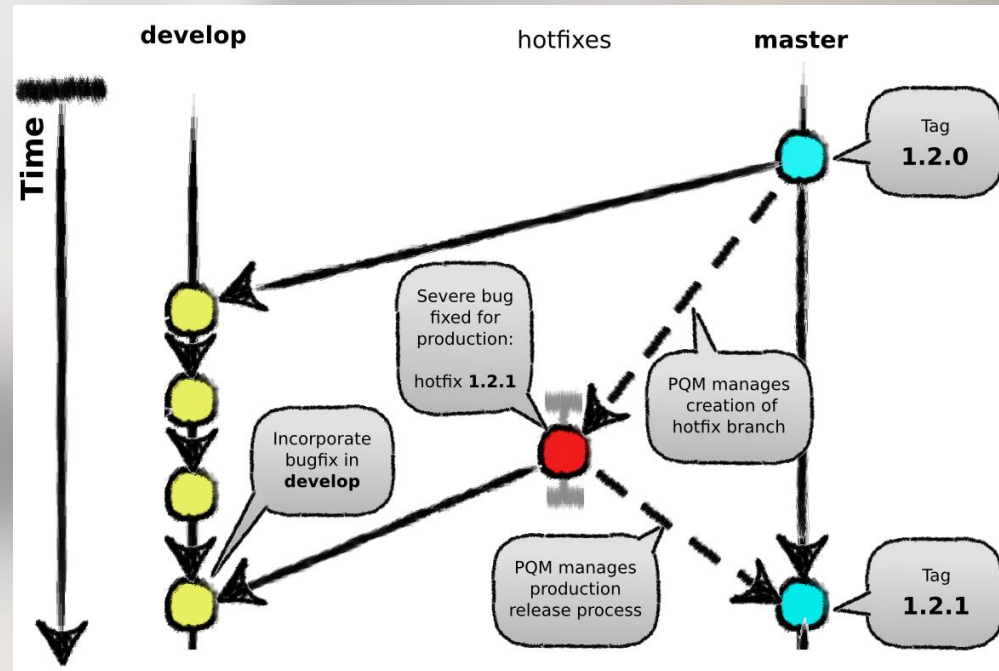
```
$ git checkout master  
$ git merge --no-ff release/v1.2  
$ git tag v1.2  
$ git push origin master  
$ git checkout develop  
$ git merge --no-ff release/v1.2  
$ git push origin develop  
$ git branch -d release/v1.2
```

Les branches de **correctifs**

Peuvent provenir de :
master

Doivent être fusionnées dans :
develop et **master**

Convention de nommage de la branche :
hotfix/*



Les branches de **correctifs**

```
$ git checkout -b hotfix/v1.2.1 master
```

Correction de l'anomalie + mise à jour des scripts/documentation (commits)

```
$ git checkout master  
$ git merge --no-ff hotfix/v1.2.1  
$ git tag v1.2.1  
$ git push --tags origin master  
$ git checkout develop  
$ git merge --no-ff hotfix/v1.2.1  
$ git push --tags origin develop  
$ git branch -d hotfix/v1.2.1
```

Git-Flow

Initialisation :

```
$ git flow init
```

Fonctionnalité :

```
$ git flow feature start myFeature
```

```
$ git flow feature finish myFeature
```

Release :

```
$ git flow release start v1.0.0
```

```
$ git flow release finish v1.0.0
```

Correctif:

```
$ git flow hotfix start myFix
```

```
$ git flow hotfix finish myFix
```

Partage:

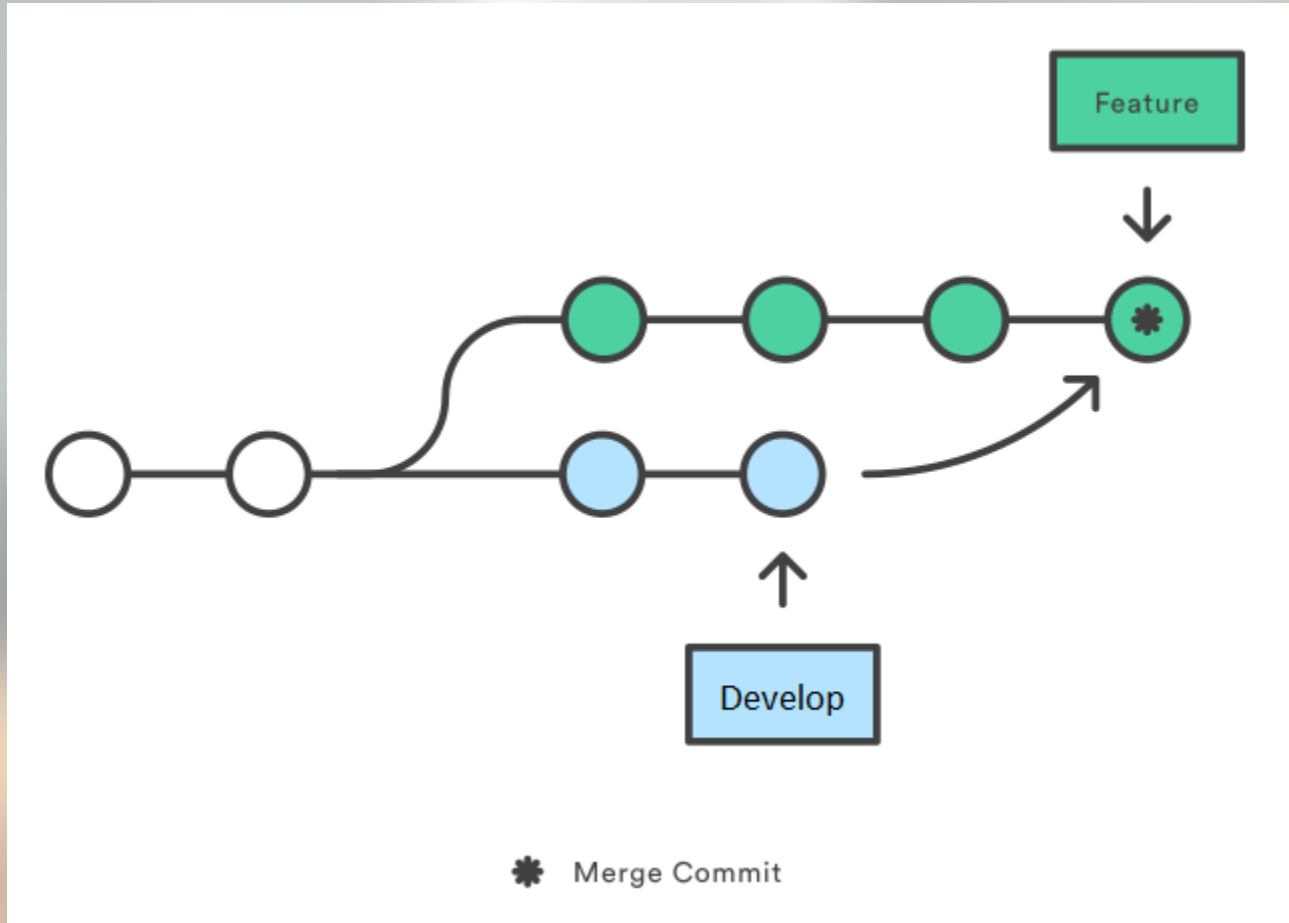
```
$ git flow feature publish myFeature
```

```
$ git flow feature pull myFeature
```

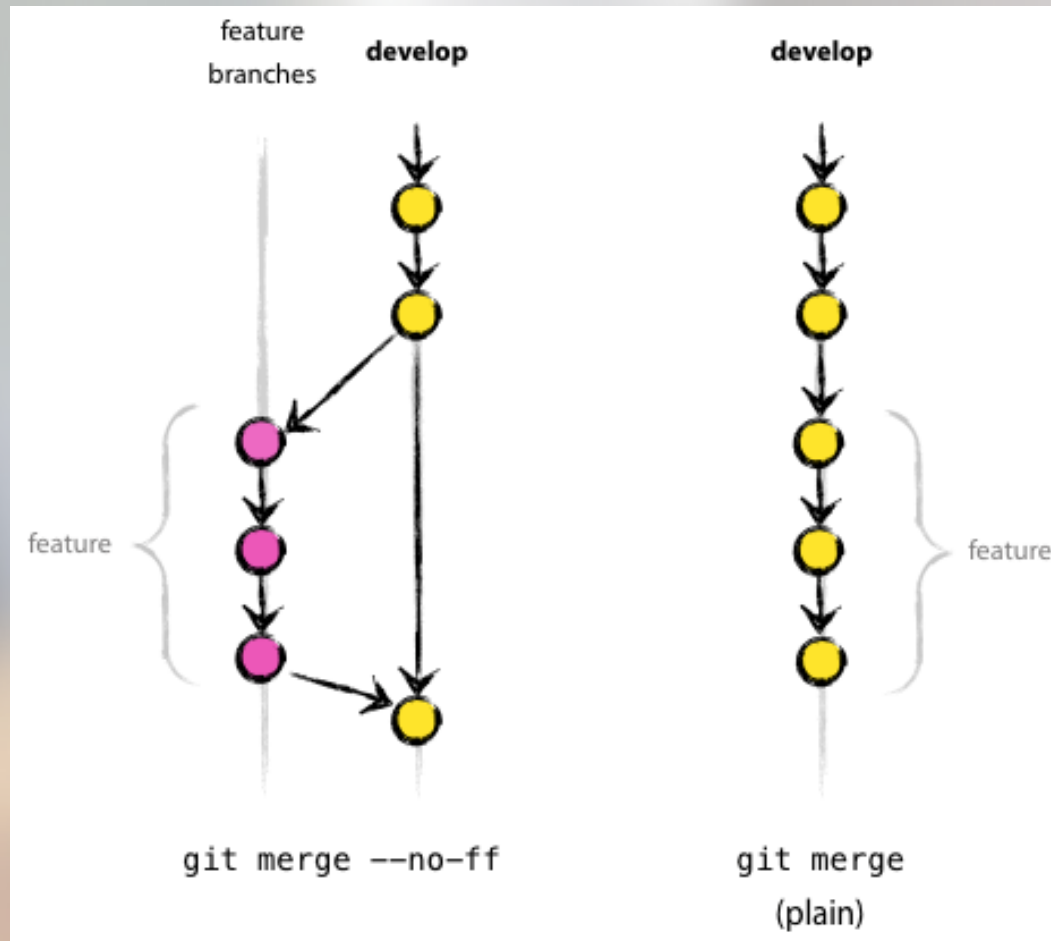

Merge vs Rebase

The diagram illustrates a branching strategy. It shows a main branch (blue) and a feature branch (green). The main branch has two commits. A feature branch is created from the main branch and has three commits. A 'Feature' box points to the feature branch, and a 'Develop' box points to the main branch.

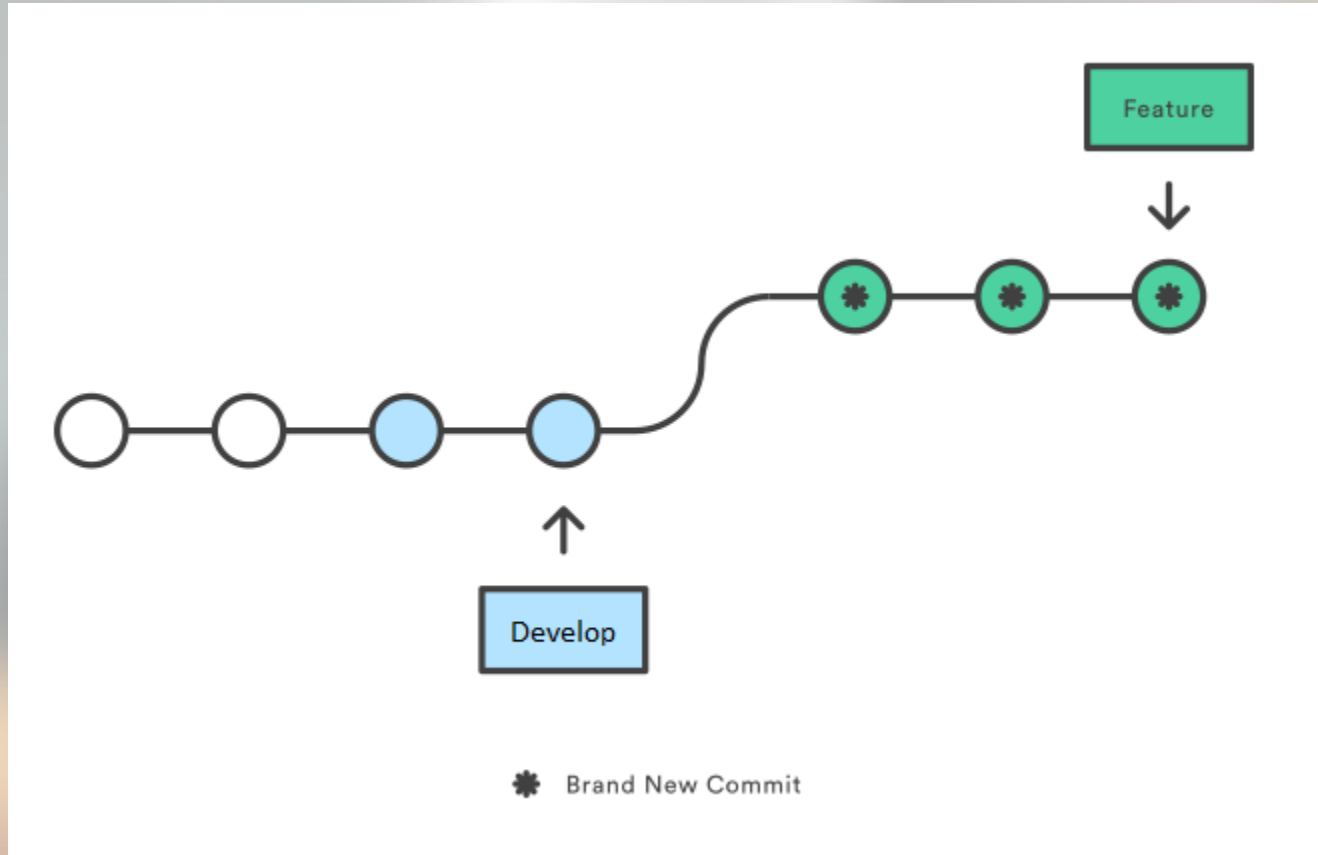
Merge



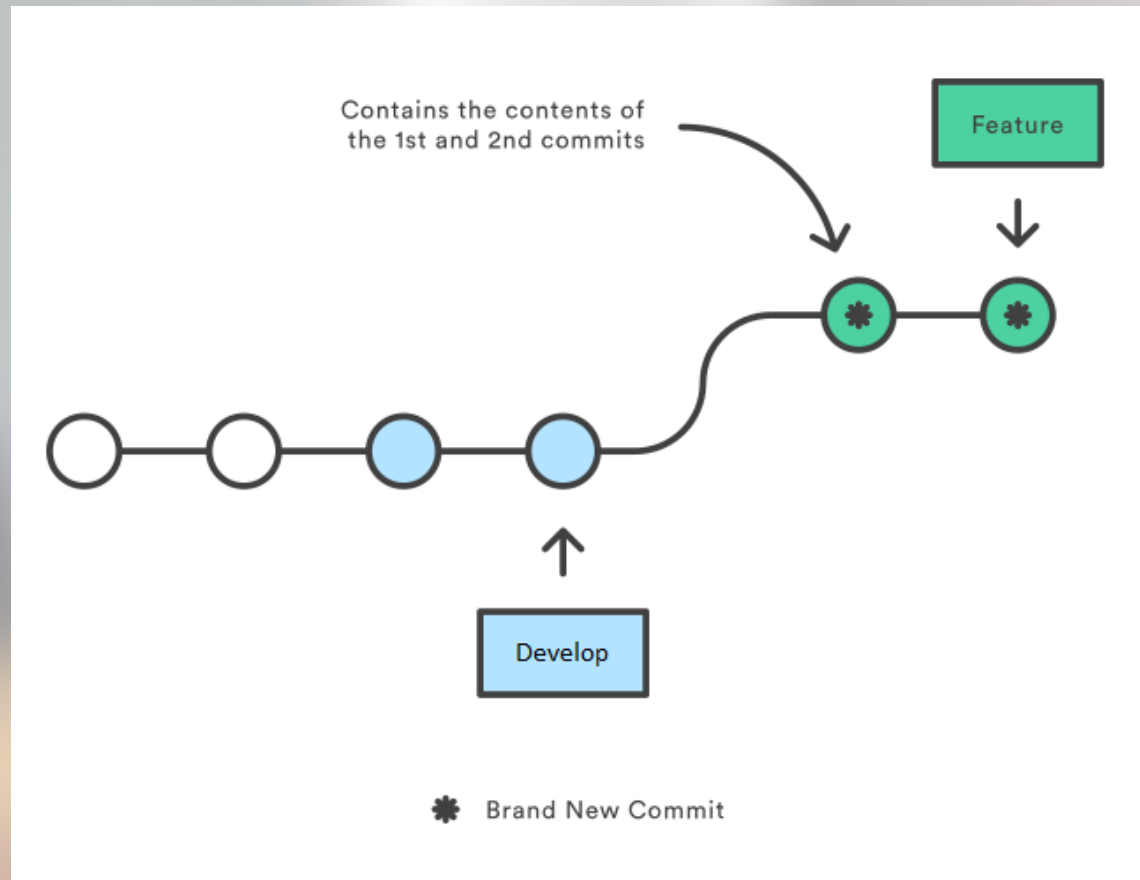
Merge – Fast Forward



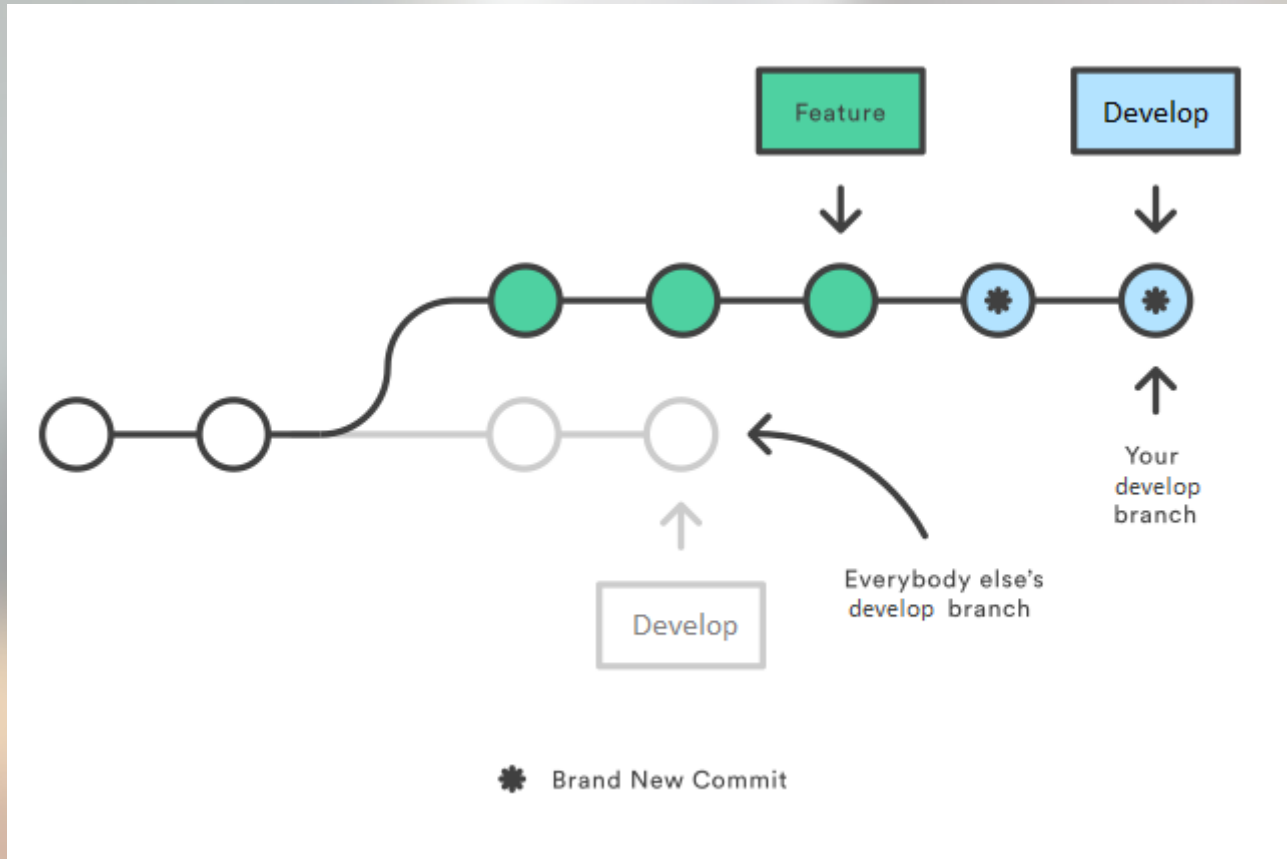
Rebase



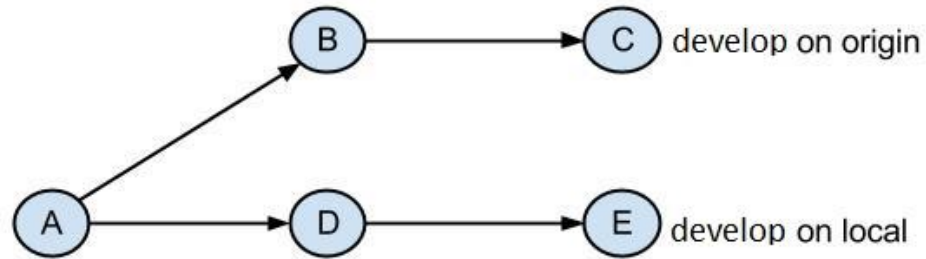
Rebase interactif



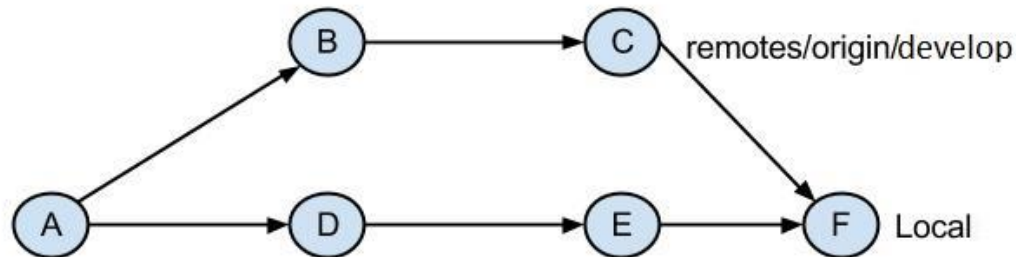
Rebase - règle d'or : **ne jamais l'utiliser sur des branches publiques**



git pull = git fetch + git merge

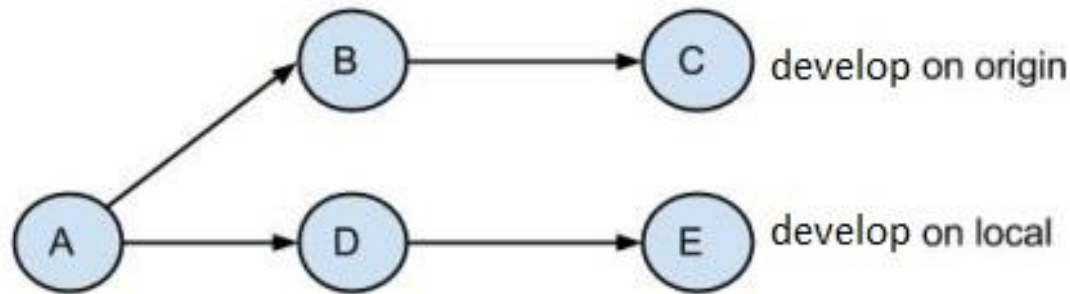


Before git pull

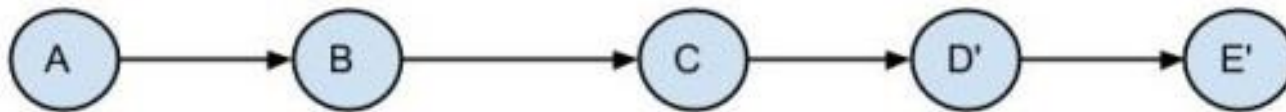


After git pull

`git pull --rebase = git fetch + git rebase`



Before git pull --rebase



After git pull --rebase

Merge : dans quels cas l'utiliser ?

- Pour avoir une branche identifiable dans le graphe
- Pour visualiser une branche « connue », identifiée par l'équipe, le bugtracker ou le gestionnaire de projet (sprint, story, bug...)
- Pour conserver la date de départ de la tâche

Rebase : dans quels cas l'utiliser ?

- Pour avoir un graphe linéaire
- Dans le cas d'une branche locale temporaire partant d'une base obsolète
- Pour nettoyer mon historique local et le mettre au propre avant de le partager
- Quand la date de départ de la tâche n'a pas d'importance

Résoudre les conflits

```
[ETMB]: ~/Documents/test$ git merge master
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Recorded preimage for 'README.txt'
Automatic merge failed; fix conflicts and then commit the result.
[ETMB]: ~/Documents/test$ git status
# On branch knock-knock
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
[ETMB]: ~/Documents/test$
```

```
README.txt  x
1 Hello World!
2
3 Knock, knock.
4 Who's there?
5 Git.
6 Git who?
7 <<<<<< HEAD
8 Git this joke over with.
9 =====
10 Git on with the assignment!
11 >>>>>> master
12
```


editCase.js (Base) <-> editCase.js (Local) <-> editCase.js (Remote) - KDiff3

File Edit Directory Movement Diffview Merge Window Settings Help

A (Base): [ce\Agent.Support\content\scripts\app\support\case\editCase.js (Base)] ... B: [source\Agent.Support\content\scripts\app\support\case\editCase.js (Local)] ... C: [source\Agent.Support\content\scripts\app\support\case\editCase.js (Remote)] ...

Top line 110 Encoding: System Line end style: DOS Top line 225 Encoding: System Line end style: DOS Top line 114 Encoding: System Line end style: DOS

```
...var Controller := Marionette.Controller.extend(
  edit: function (kase, region) {
    region := region || app.mainRegion;

    ...var view := new EditView({ model: kase });

    region.show(view);
  }
);

...var caseId;
...var Controller := Marionette.Controller.extend(
  edit: function (kase, region) {
    region := region || app.mainRegion;

    //var dt := new Date(kase.get('orderDate'));
    //dt := dt.toUTCString().replace('UTC', 'Z');
    //kase.set('orderDate', dt);

    ...var view := new EditView({ model: kase });
    caseId := kase.id;

    region.show(view);

    ...var tabSettings := [
      {
        name: 'Recipient',
        displayName: 'Recipient Information',
        onLoad: function (t) {
          ShowRecipient.show(t.region, ...
        }
      }
    ];
  }
);

...var Controller := Marionette.Controller.extend(
  show: function (region, id) {
    ...var editCase := new EditCase({ id: id });

    //var dt := new Date(kase.get('orderDate'));
    //dt := dt.toUTCString().replace('UTC', 'Z');
    //kase.set('orderDate', dt);

    editCase.fetch({
      success: function () {
        ...var view := new EditCaseView({ model: editCase });

        region.show(view);
      }
    });

    ...var tabSettings := [
      {
        name: 'Recipient',
        displayName: 'Recipient Information',
        onLoad: function (t) {
          ShowRecipient.show(t.region, ...
        }
      }
    ];
  }
);
```

Output: C:\projects\agent\source\Agent.Support\content\scripts\app\support\case\editCase.js [Modified] Encoding for saving: Codec from C: System Line end style: DOS (A, B, C)

Number of remaining unsolved conflicts: 0 (of which 0 are whitespace)

Annuler un merge/rebase :

git merge/rebase --abort

Voir qui sont les dernières personnes à avoir modifié les différentes lignes d'un fichier :

git blame *myFile*

Rechercher tous les fichiers qui contiennent le mot *TODO* dans le code source :

git grep "*TODO*"

Voir l'historique d'un fichier :

git log -p *myFile*

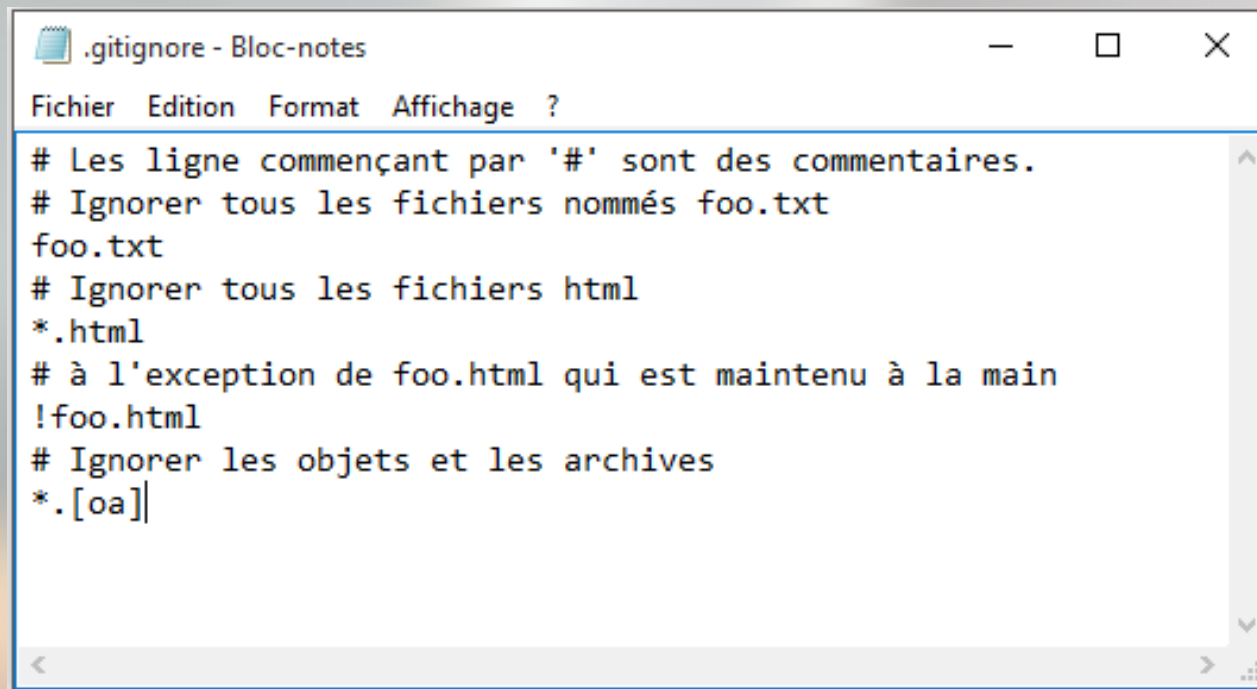
Voir les modifications sur un fichier entre deux commits :

git diff *oldId newId*

Voir les détails d'un commit :

git show *id*

Pour que Git ignore certains fichiers (binaires, fichiers de configuration...), il faut créer un fichier **.gitignore** à la racine de votre dépôt puis ajouter vos chemins à ignorer via des expressions régulières.

A screenshot of a text editor window titled ".gitignore - Bloc-notes". The window has a menu bar with "Fichier", "Edition", "Format", "Affichage", and "?". The text inside the editor is as follows:

```
# Les ligne commençant par '#' sont des commentaires.  
# Ignorer tous les fichiers nommés foo.txt  
foo.txt  
# Ignorer tous les fichiers html  
*.html  
# à l'exception de foo.html qui est maintenu à la main  
!foo.html  
# Ignorer les objets et les archives  
*.[oa]
```

Pour minimiser les conflits :

- Faire des commits réguliers et de petite taille
- Eviter les branches qui vivent trop longtemps (voir si il est possible de répartir la fonctionnalité de cette branche en plusieurs sous fonctionnalités et donc en plusieurs branches dont la durée de vie sera moins importante)
- Régulièrement se remettre à jour avec la branche distante ou de merge (après chaque commit, avec **git pull**)
- Respecter un système de branche
- Utiliser les bons outils pour vous aider

Modifier son historique

Modifier son historique - pourquoi ?

Les raisons de modifier son historique :

- Vous avez navigué entre plusieurs sujets, qui finissent par être entrelacés dans votre historique au lieu d'être regroupés
- Vous vous y êtes repris à plusieurs fois pour véritablement corriger un bug ou apporter une modification
- Vous avez fait du code pour finalement, plus tard, revenir en arrière en annulant le commit concerné, qui s'est avéré être une impasse ou une fausse bonne idée
- Vous avez fait des fautes de frappe dans vos messages de commit
- Vous n'avez pas découpé votre travail dans plusieurs commits

Tout ceci engendre un historique compliqué, peu agréable à lire, comprendre ou exploiter par les autres (ou vous dans plusieurs semaines)

Modifier son historique - stash

Mettre de côté ses modifications avec **git stash**

Mettre ses modification de coté :

git stash ou **git stash save "mon message"**

Récupérer les modifications du dernier stash (le stash sera supprimé!) :

git stash pop

Voir tous les stash sauvegardés:

git stash list

```
$ git stash list
stash@{0}: WIP on master: 6936d57 third test commit
stash@{1}: WIP on master: a5d84ed modified this test3 file
stash@{2}: WIP on master: 232dfec added test files
```

Récupérer les modifications d'un stash particulier :

git stash pop stash@{2}

Modifier son historique - commandes 1

Créer un nouveau commit qui sera fusionné avec le dernier commit :

git commit -am --amend "mon message"

« Unstage » un fichier :

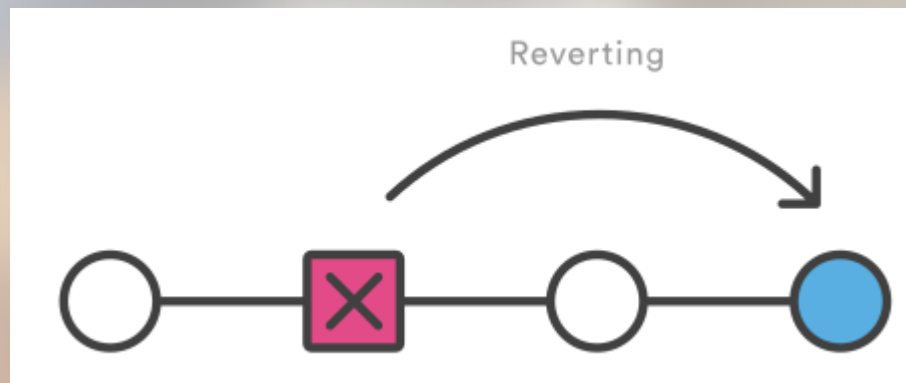
git reset nomfichier

Annuler les modifications d'un fichier quand elles ne sont pas « staged » :

git checkout nomfichier

Annuler un commit envoyé (crée un commit qui effectue l'inverse des modifications) :

git revert 6261cc2



Modifier son historique - commandes 2

Sélectionner un commit et l'appliquer sur la branche actuelle :

git cherry-pick *d42c389f*

Supprimer les fichiers « non trackés » :

git clean -f

Afficher les fichiers qui seront supprimés :

git clean -n

Modifier son historique - reset

Annuler le dernier commit (soft : seul le commit est retiré de Git; vos fichiers, eux, restent modifiés) :

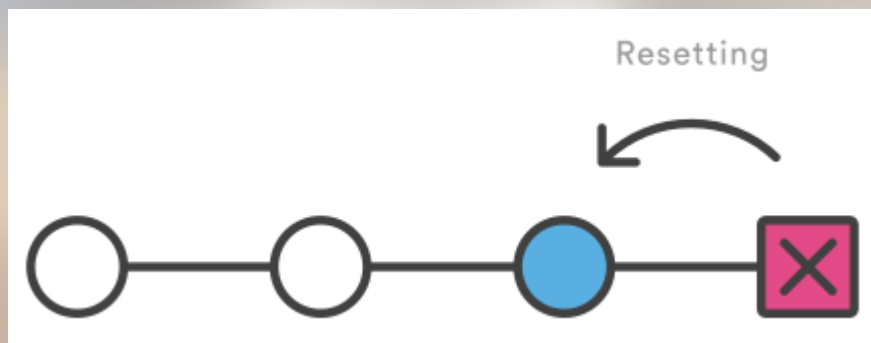
git reset HEAD^

Annuler tous les changements du dernier commit (hard : annule les commits et perd tous les changements) :

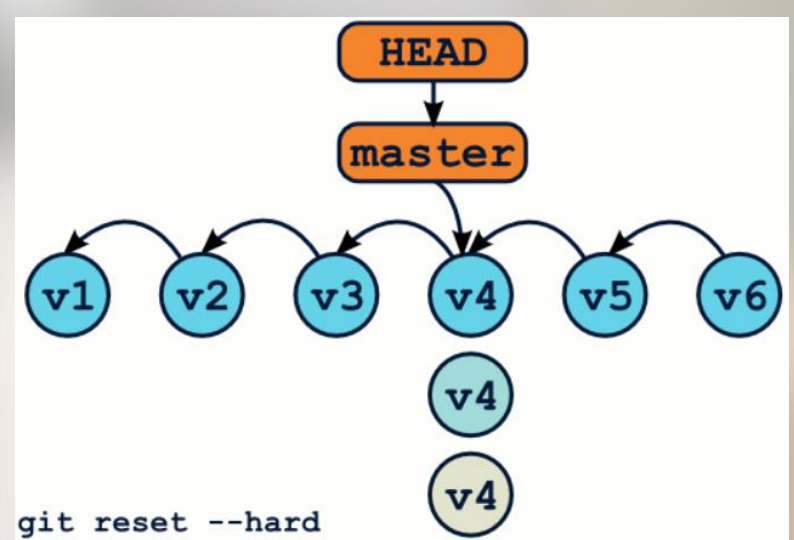
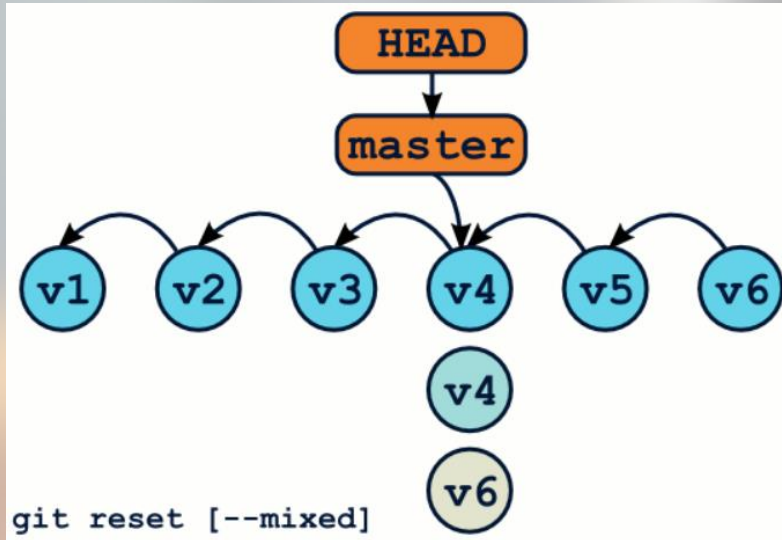
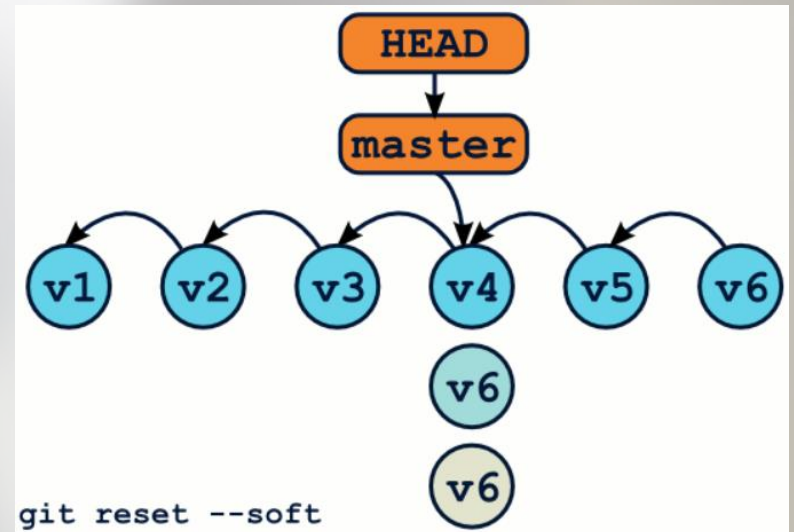
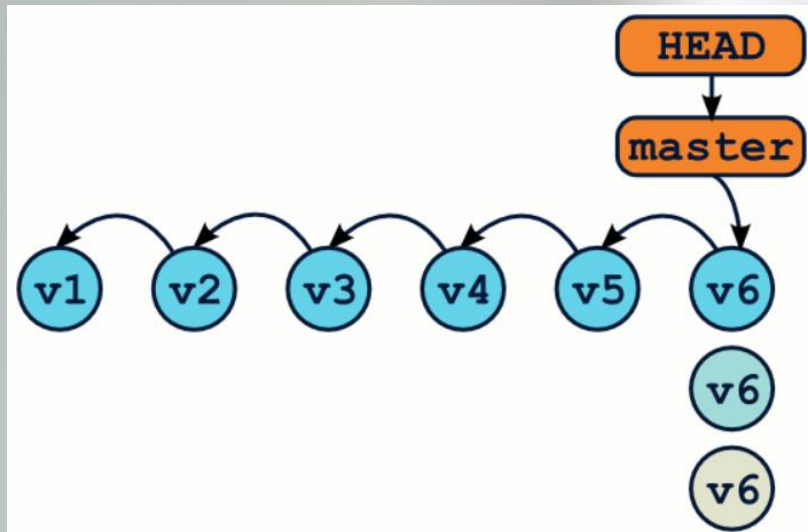
git reset --hard HEAD^

Retirer un fichier qui avait été ajouté pour être « commité » :

git reset HEAD -- *fichierAsupprimer*



Modifier son historique - reset



Modifier son historique - HEAD

HEAD : dernier commit

HEAD^ : avant-dernier commit

HEAD^^ : avant-avant-dernier commit

HEAD~2 : avant-avant-dernier commit (notation équivalente)

d6d989238685 : indique un numéro de commit précis

Modifier son historique - rebase interactif

Le *rebase* interactif fonctionne comme un *rebase* classique, à ceci près qu'au lieu de suivre un script simple (« je cherry-picke tous les commits un par un, en laissant tomber ceux qui font désormais doublon »), il vous permet d'éditer le script en amont.

Ce *rebase* ne changera pas, en fait, le commit de départ. Il se contentera de réécrire l'historique depuis ce commit, il est donc possible de :

- Réordonner les commits
- Les fusionner
- Les découper
- En supprimer
- Reformuler leurs messages de commit

git rebase -i 55646fd17

git rebase -i origin/sprint

git rebase -i HEAD~5

Modifier son historique - rebase interactif

```
pick 057ad88 Locale fr-FR
pick ef61830 Opinion bien tranchée
pick 8993c57 ML dans le footer + rewording Interactive Rebasing
pick dbb7f53 Locale plus générique (fr)
pick c591fd7 Revert "Opinion bien tranchée"
pick 2863a46 MàJ .gitignore

# Rebase 34aelae..2863a46 onto 34aelae
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

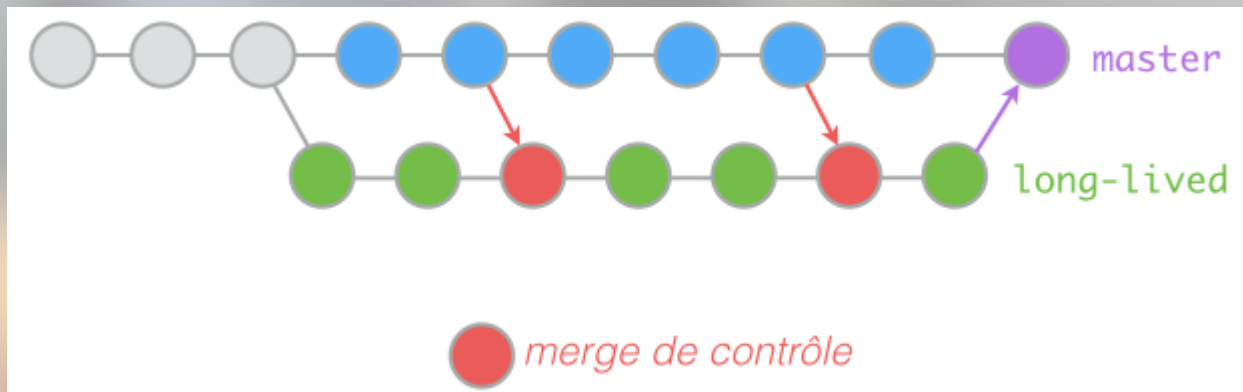

Modifier son historique - rerere

Rerere : reuse recorderd resolution

Cette commande permet à Git de se rappeler de quelle façon un conflit a été résolu et le résoudre automatiquement de la même façon la prochaine que ce conflit se présente.

Pour l'activer :

git config --global rerere.enabled true



Modifier son historique - git bisect

Pour isoler au plus vite l'origine d'un bug, même très loin en arrière dans l'historique

Pour démarrer : **git bisect start**

Identifier le premier commit problématique connu (si rien n'est ajouté c'est HEAD qui est utilisé) : **git bisect bad**

Identifier un commit qui n'avait pas le problème (le plus proche possible de nous si possible) : **git bisect good**

Git va ensuite automatiquement faire des checkout à différents commits dans le temps (par dichotomie), à nous de lui dire si le bug est présent sur cette version :

git bisect good/bad/skip

Pour enregistrer la progression effectuée : **git bisect log > *monfichier***

Pour terminer : **git bisect reset**

Pour reprendre là où vous en étiez : **git bisect replay *monfichier***

Références

<http://git-scm.com/doc>

<https://fr.atlassian.com/git/tutorials/>

<http://nvie.com/posts/a-successful-git-branching-model/>

<https://github.com/nvie/gitflow>

<https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git>

Travaux Pratiques



metrixware

<http://www.metrixware.com>



<http://www.tocea.fr>



<http://www.echoes.fr>

Site de Rennes

16c, rue Jouanet
35700 Rennes, France
Tél : +33 2 99 38 00 44

Site de Nanterre

18-22, rue d'Arras
92000 Nanterre, France
Tél : +33 1 55 69 32 20

contact@metrixware.com