Git comment ça marche ?

Pourquoi la gestion de versions ?

J'ai fait plein de modifications et supprimé du code.
Plus rien ne fonctionne, je veux revenir en arrière, je fais comment ?

→ Avoir un historique

Nous bossons en équipe, pendant que je fais un fix, mon collègue a changé plusieurs choses dans le même fichier dont le nom de la méthode. Comment faire ?

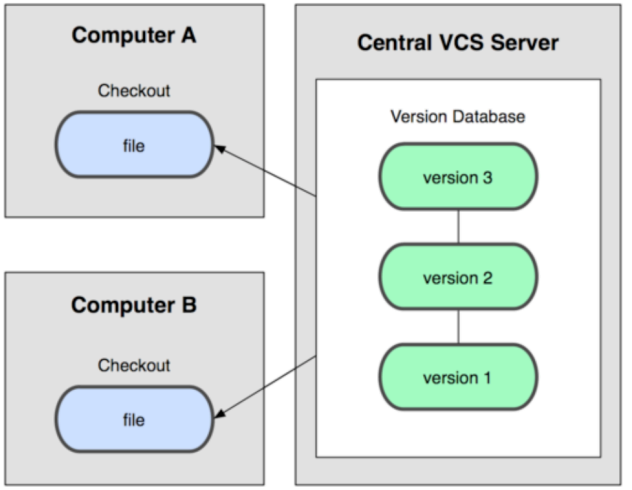
→ Aider à la collaboration

Le client veut corriger un bug sur la version de production alors que l'équipe a changé plein de choses sur la prochaine version.
Evidemment la prochaine version devra également bénéficier du correctif.
Comment faire ?

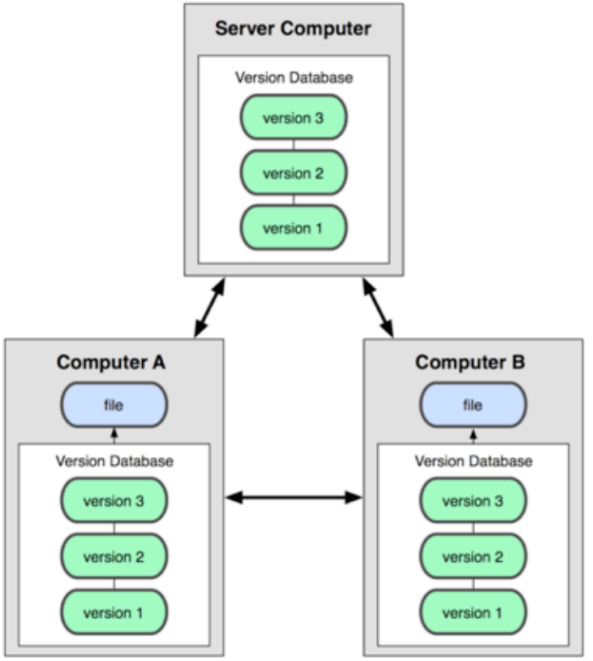
→ Gérer les versions 😃

Différence entre Git et SVN ?

SVN



GIT



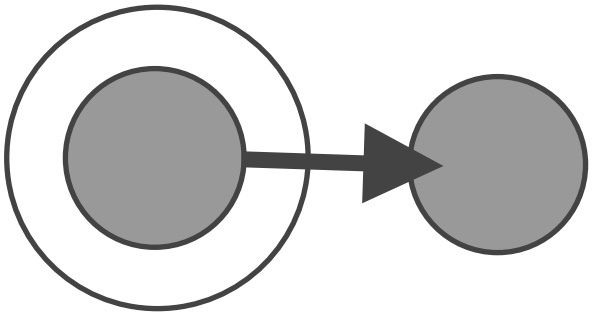
Gestion de versionning en local !

Vocabulaire git

git commit

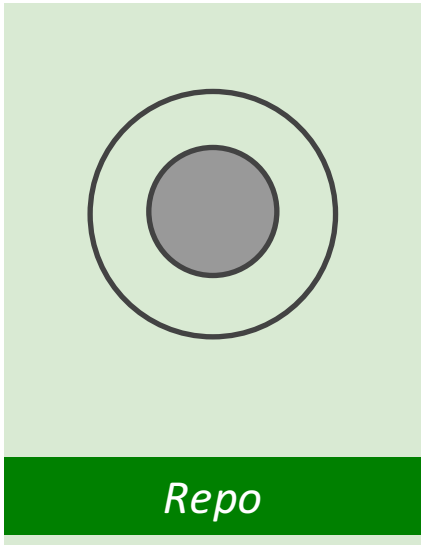
Un *commit* est un ensemble d'un ou plusieurs fichiers ayant subi des modifications. **Il reçoit un identifiant unique à la création (sha).**

Quand on publie des modifications de fichier(s), on le fait depuis un *commit* parent vers un *commit* enfant. Le parent est donc la version précédente des fichiers.



git commit

NB : La modification du premier *commit* est la création de fichier.
Elle se fait dans un répertoire que l'on appelle le *repository*, le "*repo*", ou “dépôt” en français.

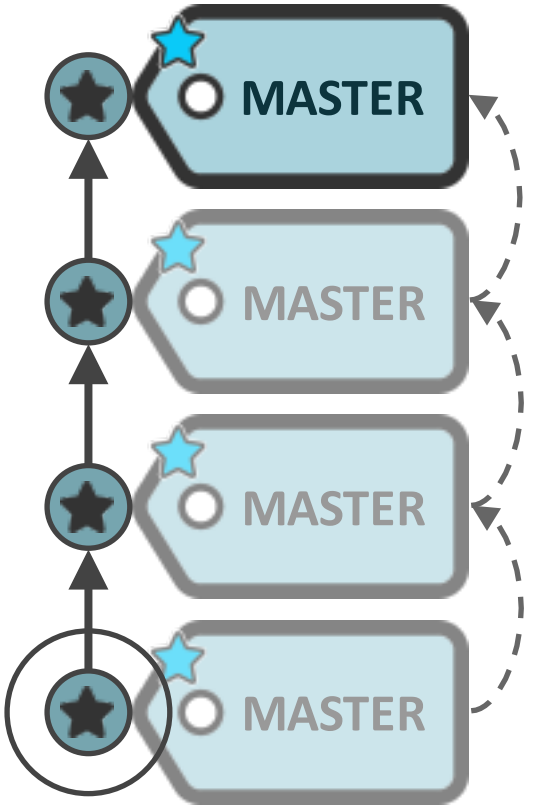


git branch

Une branche représente un ensemble de commits regroupé en une seule étiquette.

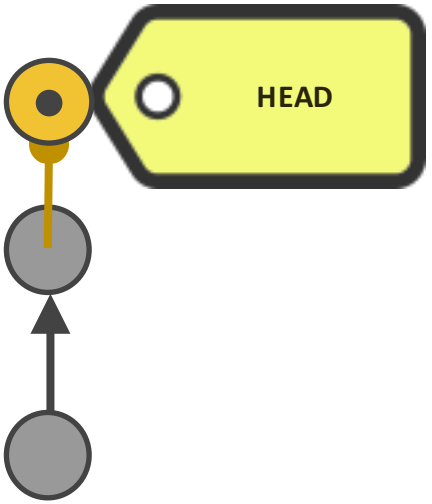
Les commits étant regroupé dans le temps, une branche possède un historique, dans lequel on peux naviguer.

Par convention, la première branche est la "*Master*".

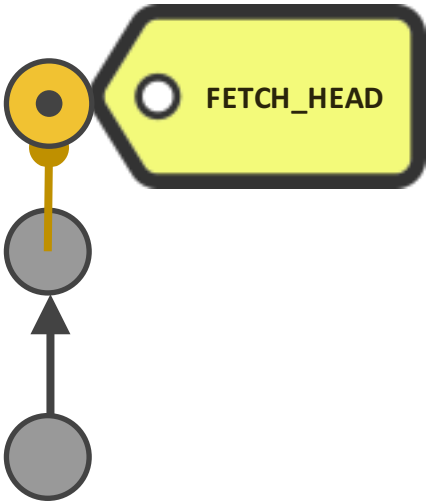


HEAD

On désigne par "*head*" le dernier *commit* ***local*** quelle que soit la branche où l'on est en train de travailler.

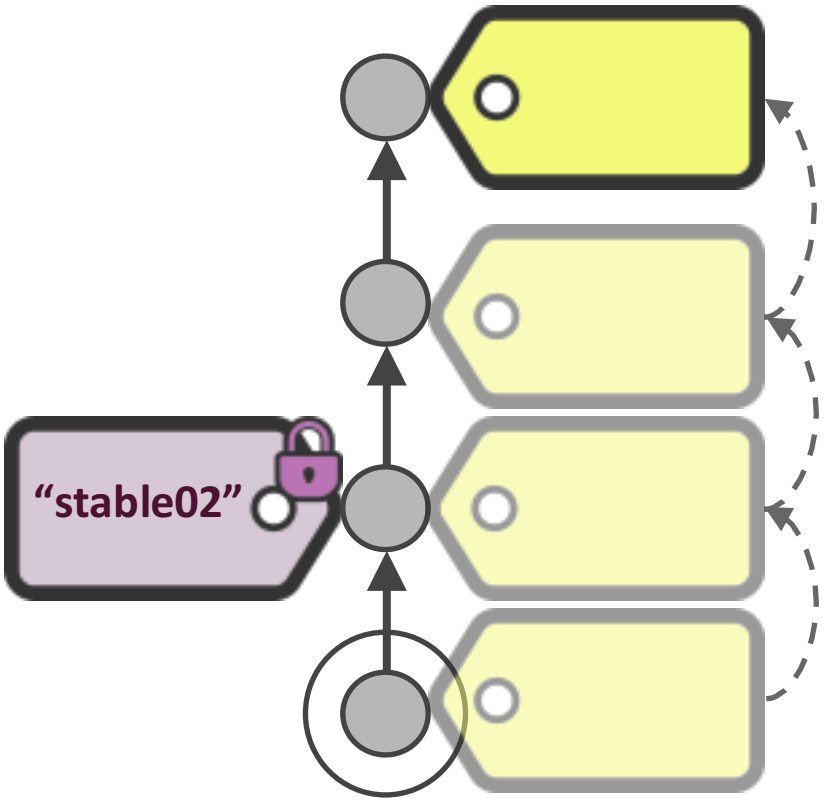


NB : On désigne par "fetch_*head*" le dernier *commit* ***remote*** quelle que soit la branche où l'on est en train de travailler.



Tag

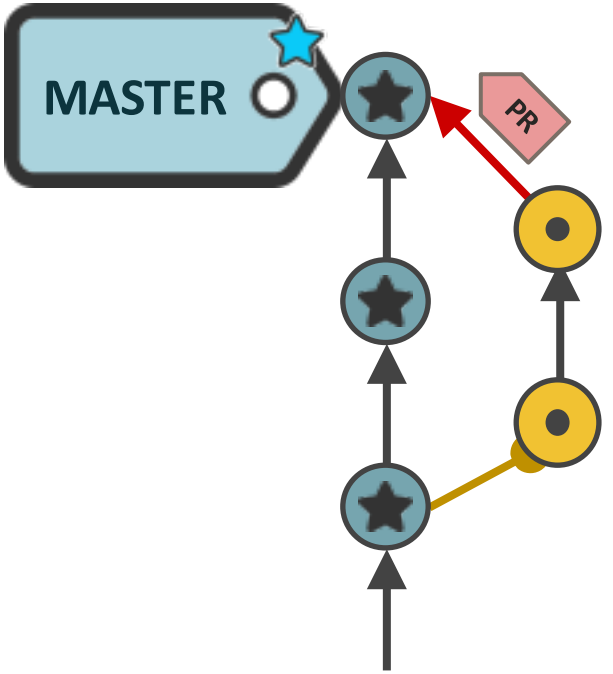
On peut également attribuer une étiquette non-mobile à un *commit*. C'est un *tag*.
Un *tag* est attaché à un *commit* (une version des fichiers) et ne "bouge" pas.



git merge

Si on est satisfait de notre travail sur une nouvelle branche issue du *master*, on reverse les modifications dans le *master* : on *merge*.

On peut faire passer ce *merge* par une étape de validation qu'on appelle *Pull Request* (PR).



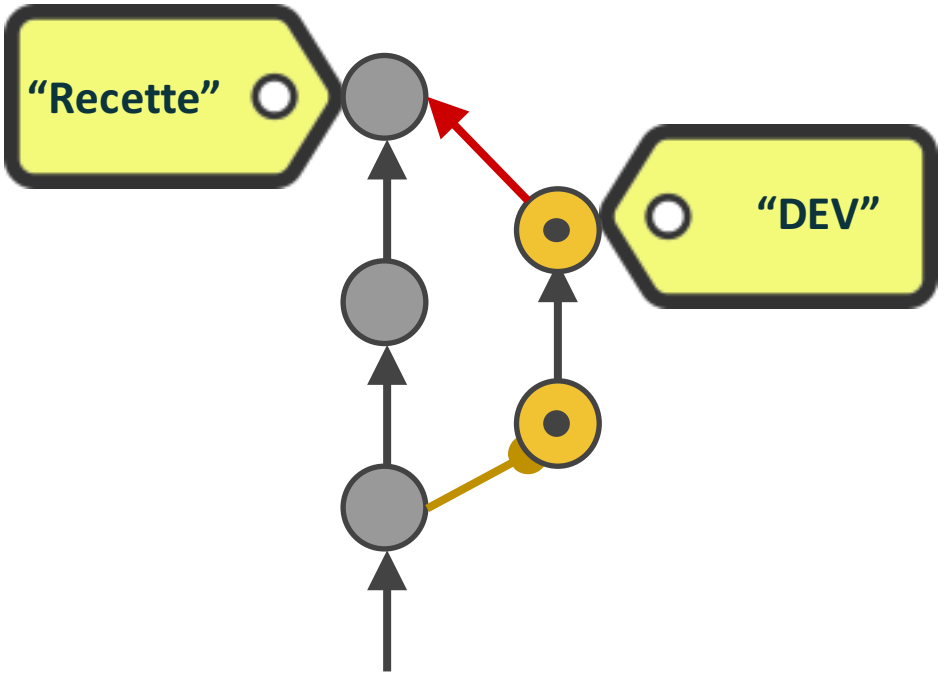
Conflit

Situation empêchant une fusion automatique ; se résout manuellement

workflow

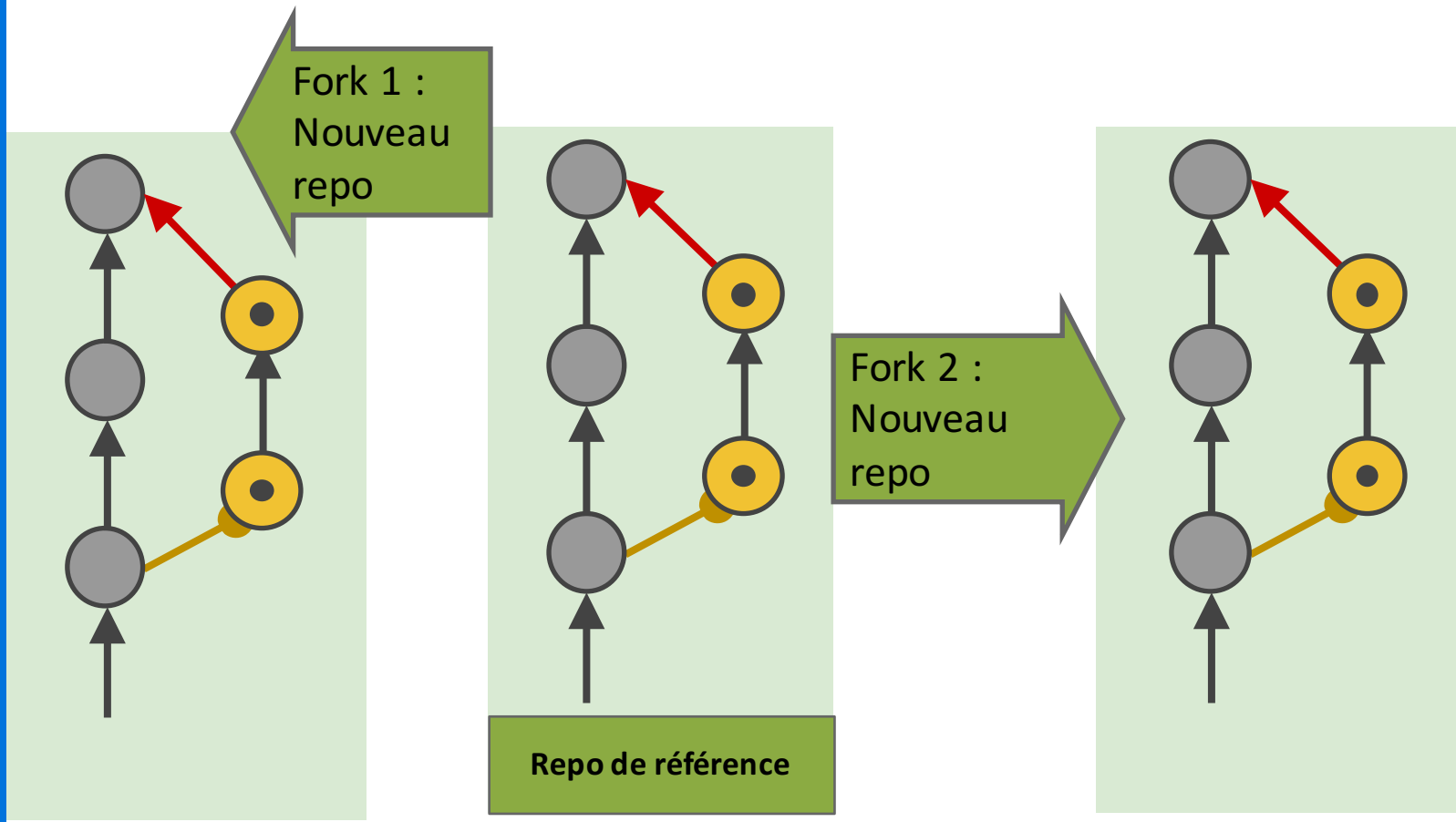
Il n'y a pas que depuis le *master* qu'on peut créer des branches. On peut créer des branches à partir de toutes les branches.

On peut donc également merger dans n'importe quelle branche ; pas uniquement le *master*.



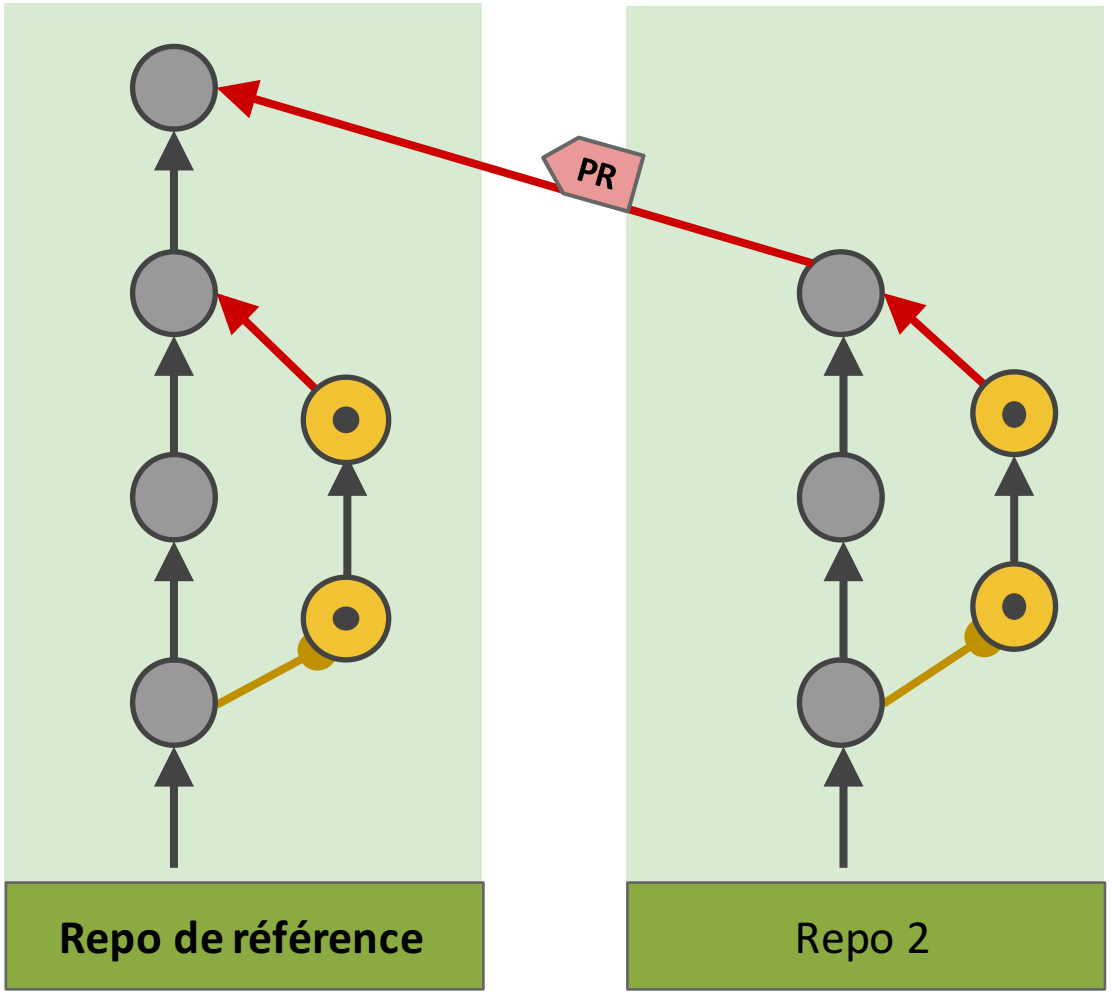
Fork

Quand on souhaite travailler à plusieurs sur un même projet (c'est-à-dire sur un ensemble de branches), chacun crée un clone du projet dans son propre espace de travail.
C'est un *fork* du projet (du *repository*).

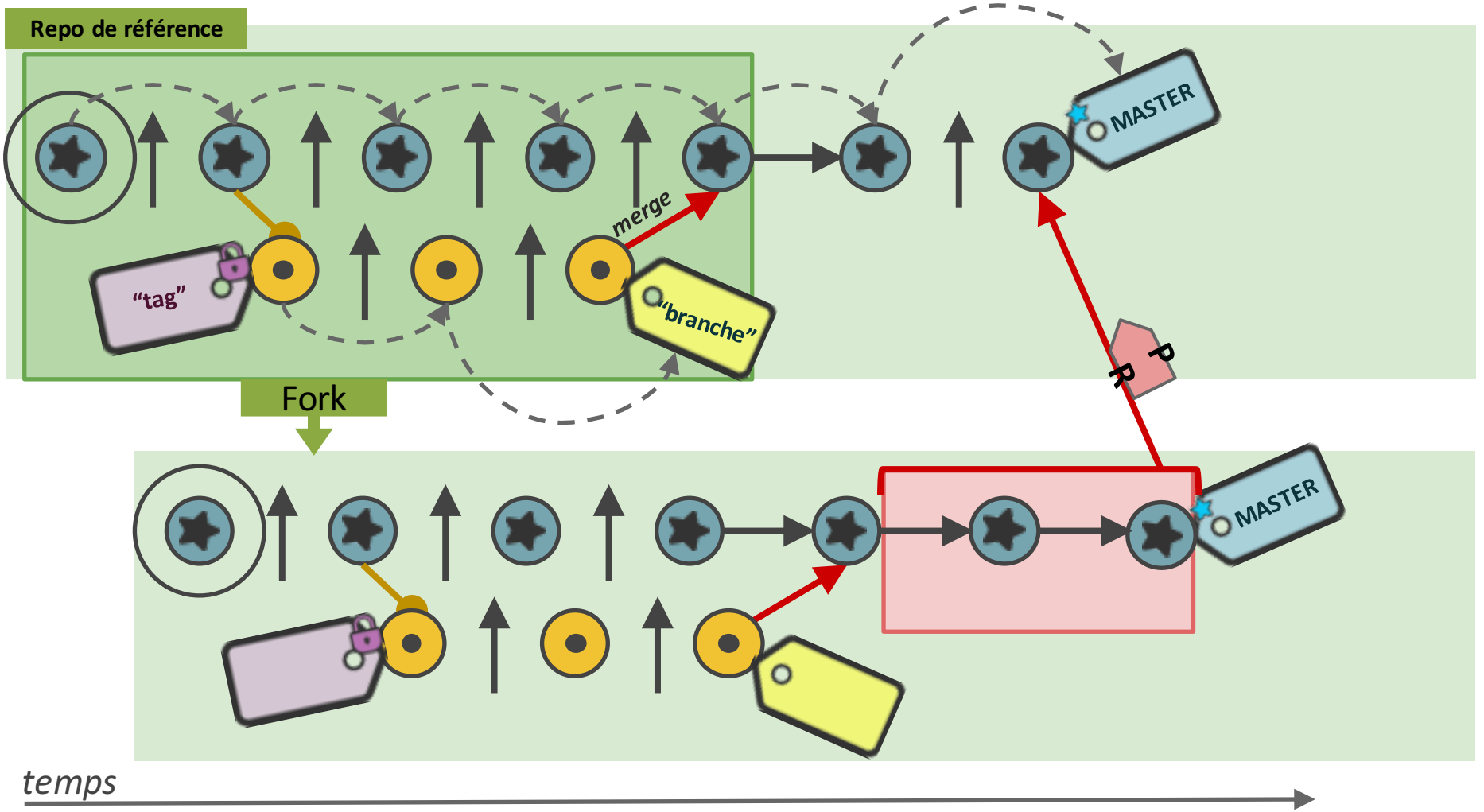


Fork

Dans chacun des *repository*, le projet évolue (commits, branches, *pull requests*). On peut faire une PR (*Pull Request*) depuis un *repository* forké vers le *repository* de référence.



Récap


Repo de référence
MASTER
merge
"tag"
"branche"
Fork
P
R
P
MASTER
temps

Configuration Git

Sous mac

Déjà fait :)

Sous Windows

http://git-scm.com/download/win

Informations Git

- **.gitconfig**

Informations utilisateur, configuration des branches, dépots distants etc.

- **.gitattributes**

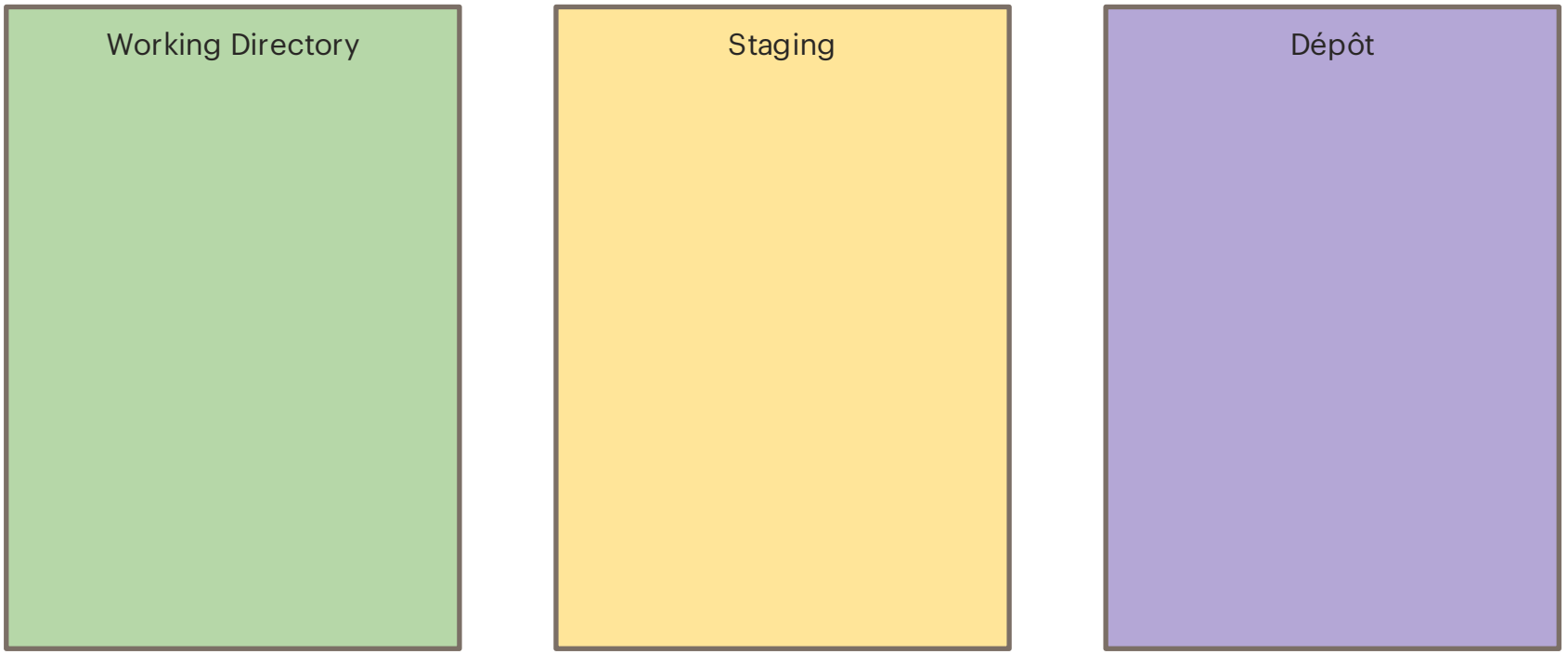
Information et comportement à adopter sur types de fichiers (texte, binaire, outil à utiliser, etc.)

- **.gitignore**

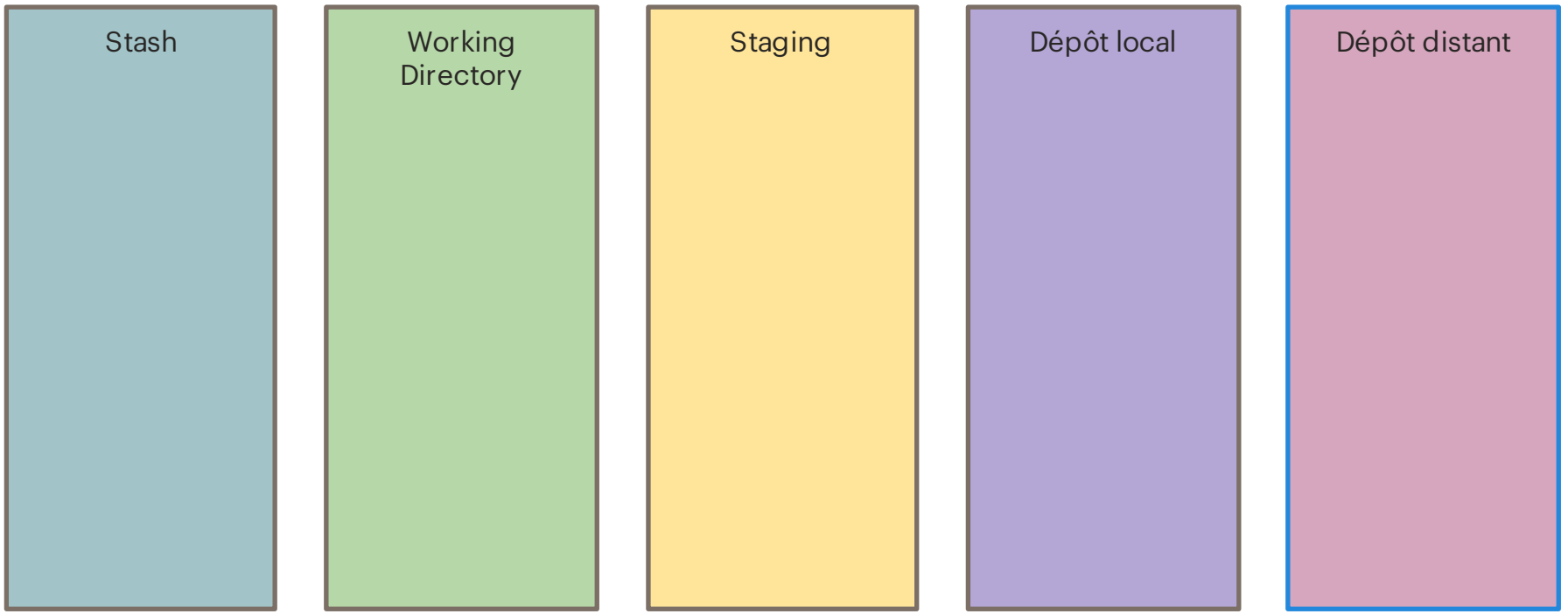
Fichiers et chemins à ignorer (ne seront pas sauvegardés/versionnés)

Avant de démarrer

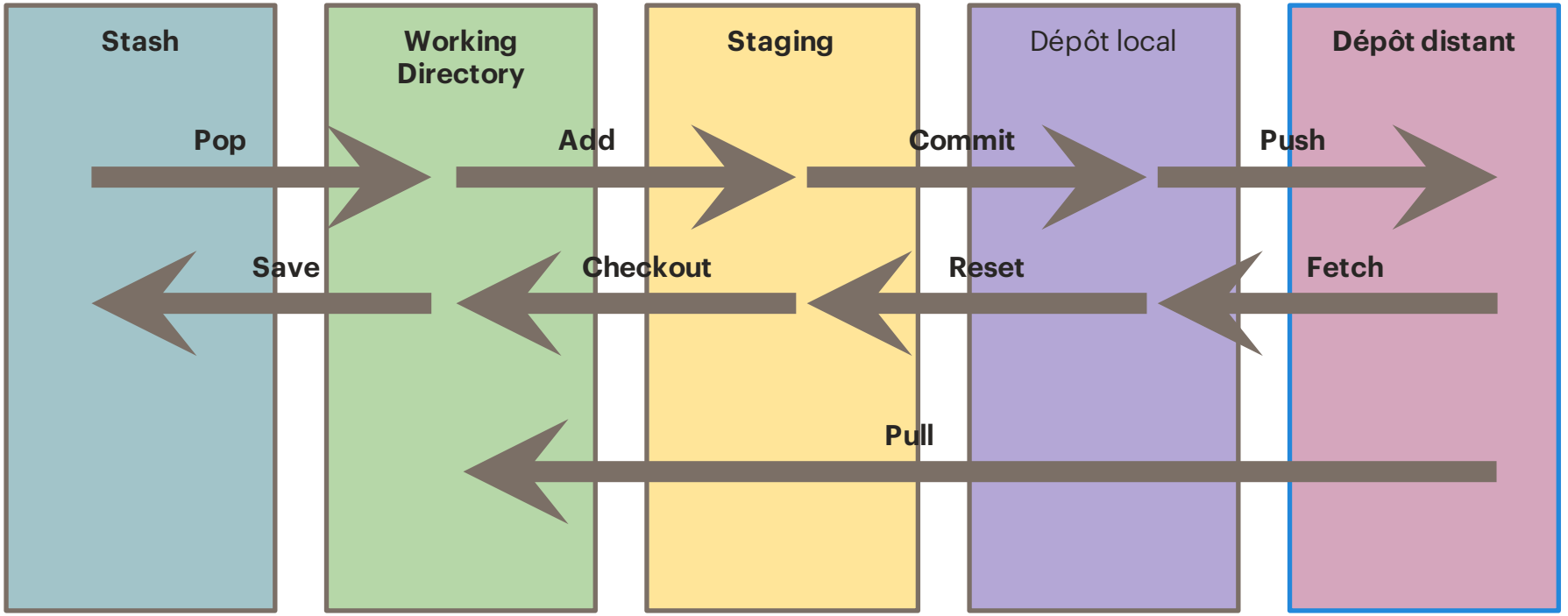
Zones



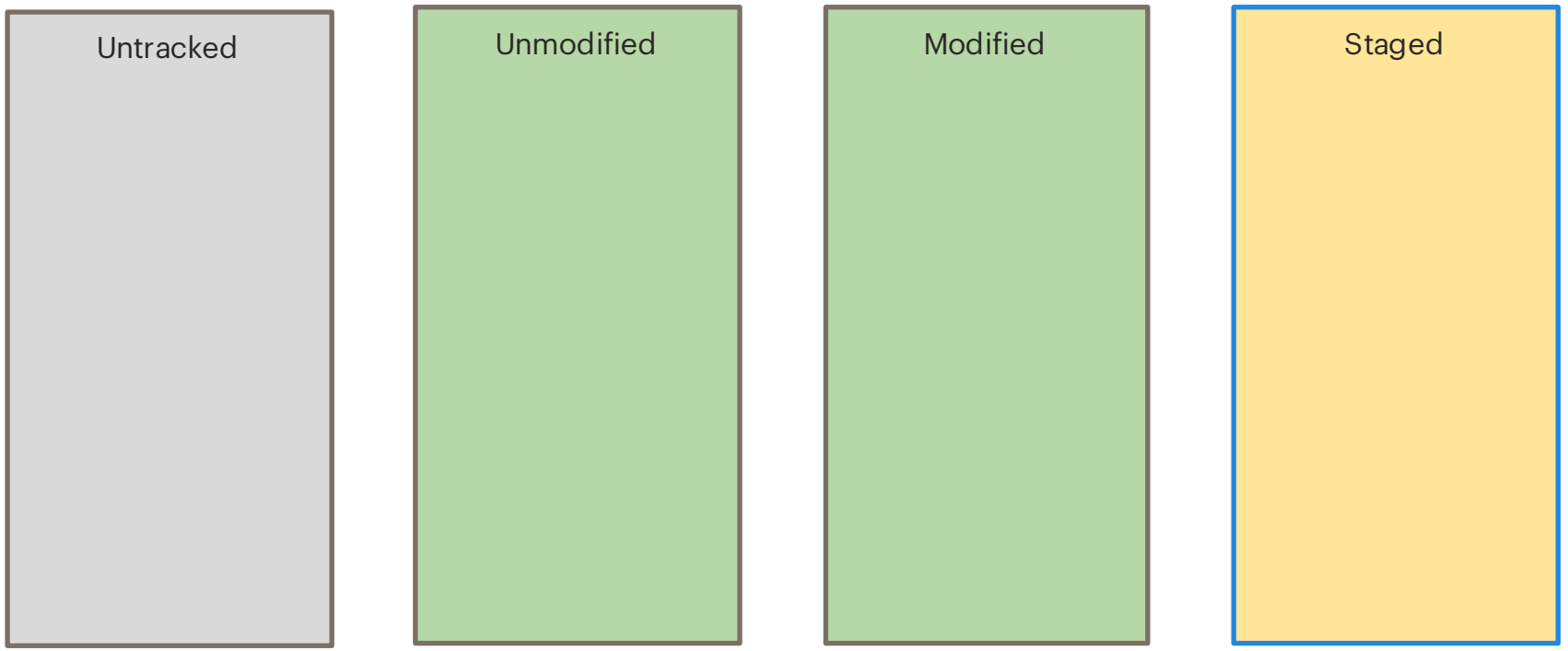
Zones


Stash
Working Directory
Staging
Dépôt local
Dépôt distant

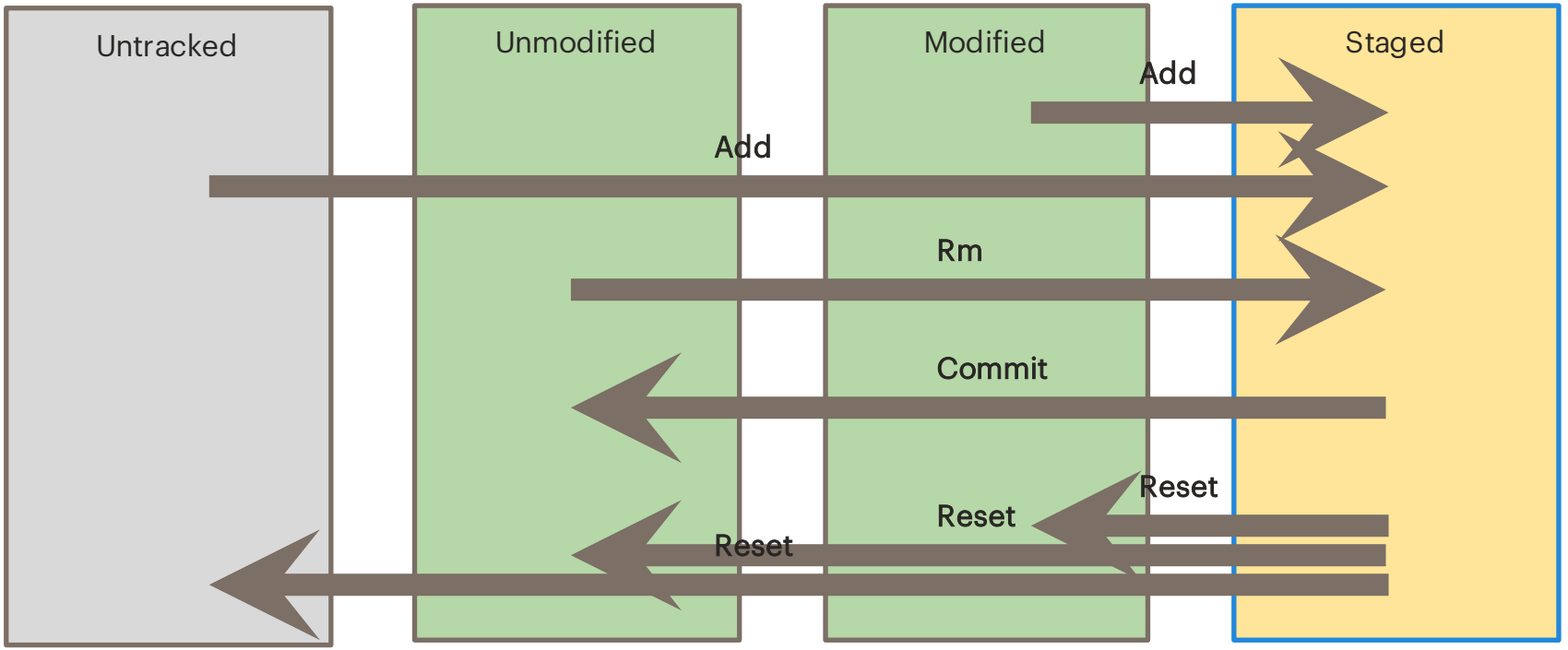
Zones


Stash
Working Directory
Staging
Dépôt local
Dépôt distant
Pop
Add
Commit
Push
Save
Checkout
Reset
Fetch
Pull

Etats



Etats


Untracked
Unmodified
Modified
Staged
Add
Add
Rm
Commit
Reset
Reset
Reset

Je créé un dépot

Créer un dépot et faire son premier commit :

- **cd /mon_repertoire_a_versionner**
- **git init**
- **git add .**
- **git commit -m “Import initial”**

Je clone un dépot

La plupart du temps on travaille sur un dépôt existant :

- git clone git@github.com/kimak/ionic-places.git
- git clone https://github.com/kimak/ionic-places

On peut spécifier le nom du répertoire cloné :

- git clone git@github.com/kimak/ionic-places my_folder

Les commandes git

git add

- git add . — modified + new files
- git add -u : pareil que git rm (deleted)
- git add -A : . + -u = modified/new/deleted
- git add -f : même si le fichier est ignoré

Les commandes git

git commit

- git commit -m : le message à la volée
- git commit -a : Ajouter tout ce qui est déjà « tracked » Attention, pas les untracked !
- git commit -am : la combo
- git commit --amend : éditer le commit précédent

Les commandes git

git rm

- git rm : supprime dans l'index et dans le working directory
- git rm -r : supprime récursivement
- git rm --cached : supprime uniquement dans l'index

Les commandes git

git reset

git reset --soft : ne touche pas au working tree ni à l'index, et laisse les modifications dans le staging

git reset --mixed : réinitialise l'index mais pas le working tree, et laisse les modifications hors staging

git reset --hard : réinitialise l'index et le working tree, en supprimant toutes les modifications

Les commandes git

git status

Permet de voir immédiatement

- Les fichiers non versionnés (*untracked*)
- Les fichiers modifiés non validés (*modified*)
- Les modifications validées (*staged*)
- Certains cas spéciaux (*both modified* : conflit de fusion…)

Fournit des indications utiles aux débutants

- Pour valider une modification
- Pour « dévalider » une modification validée
- etc.

Les commandes git

git stash

“Tu me fais un petit fix vite fait ?”

- Au-delà d’un simple stash / stash apply
  - $ stash save -u : message explicite, conserver le stage...
  - $ stash pop --index

Cas d'utilisation

Fichier oublié

Mon fichier aurait dû être présent dans mon commit précédent.

Que faire?

1. **Ajouter le fichier dans l'index**

   ```
   git add the_given_file
   ```

2. **Modifier le dernier commit pour intégrer les modifications placées dans l'index**

   ```
   git commit --amend -C HEAD
   ```

Cas d'utilisation

Annulé le dernier commit

J'ai commit trop de fichiers dans le même commit. Que faire?

1. **Repasser les modifications commitées dans l'index**

   ```
   git reset --soft HEAD^
   ```

2. **Effectuer les modifications**

Les commandes git

git stash

“Tu me fais un petit fix vite fait ?”

- Au-delà d’un simple stash / stash apply
  - $ stash save -u : message explicite, conserver le stage...
  - $ stash pop --index

Les commandes git

git branch

Liste les branches

- locales par défaut (-l)
- distantes (-r)
- toutes (-a)

Crée une branche locale

- git branch ma-branche

Supprime une branche locale

- git branch -d ma-branche

Les commandes git

git checkout

Positionner l'étiquette HEAD sur un commit

- En général sur une autre étiquette (plus pratique)

Peut s'enchaîner pour creer une branche et se positionner dessus

- git checkout -b ma_feature

Peut servir pour reset un fichier modifié

- git checkout my-file.js

Les commandes git

git tag

Egalement une “étiquette”

➔ Représente généralement une version, un point précis dans les devs.

➔ Ne bouge pas ! (bien que techniquement possible)

- git tag v1.0.4

Crée le tag sur le commit courant

- git tag -l “v1.*”

Liste les tags selon un pattern

- git push origin v1.0.4

Envoie le tag v1.0.4 sur le remote “origin”

➔ Ne pas utiliser “git push --tags” (repush des tags supprimés sur le remote)

Merge ou rebase ?

Un git merge ne devrait être utilisé que pour la récupération fonctionnelle, intégrale et finale d'une branche dans une autre, afin de préserver un graphe d'historique sémantiquement cohérent et utile, lequel représente une véritable valeur ajoutée.

Tous les autres cas de figure relèvent du rebase sous toutes ses formes : classique, interactif ou cherry picking.

http://www.git-attitude.fr/2014/05/04/bien-utiliser-git-merge-et-rebase/

Conflits

Indiqué par git

- A faire manuellement
- Penser à git add les fichiers fusionner
- Pour un merge : git commit
- Pour un rebase : git rebase --continue (parfois git rebase --skip)

Je suis perdu ? Selon le cas

- git merge --abort
- git rebase --abort

Practice

https://pcottle.github.io/learnGitBranching/

Dépots distant

- git remote add *name url*

git remote add kimakhttps://github.com/kimak/formation-angular

- git remote remove *name*
- git remote rename *old new*
- git remote set-url *name url*

ex. git remote set-url origin **https://github.com/kimak/formation-angular**

pull / push

git pull
Mettre à jour le local, par rapport au remote

git push
Publier le travail local de la branche en cours

git push -u origin mafeature (ou --set-upstream)

Publier une nouvelle branche (et la référencer sur origin/mafeature)

git push origin :mafeature
Supprimer une branche distante

N'empêchera pas sa republication automatique si un collaborateur la repush.