Collaborer pour développer des logiciels

Frédéric Dadeau

Département Informatique des Systèmes Complexes – FEMTO-ST

Bureau 405C

frederic.dadeau@univ-fcomte.fr

Licence 2 – Année 2023-2024

Plan du cours

Système de gestion de version - Git

Intégration continue, livraison continue, déploiement continu

Organiser une équipe

Règles de codage

Plan du cours

Les différents types de VCS

VCS = Version Control System

Un gestionnaire de version est un système qui enregistre les évolutions d'un ou plusieurs fichiers au cours du temps de manière à pouvoir rappeler une version antérieure d'un fichier à tout moment.

A quoi cela sert-il ?

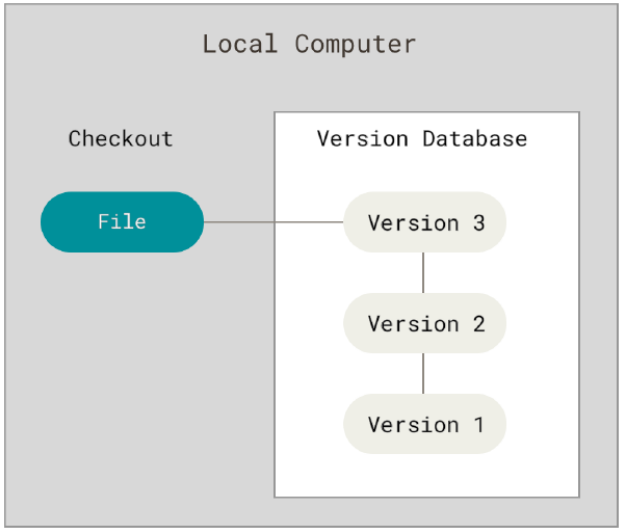
- sauvegarde des versions successives d'un logiciel
- possibilité de changer de version
- partage éventuel entre différents collaborateurs

Source et illustrations de cette partie : ouvrage *Pro Git* par Scott Chacon et Ben Straub - `https://git-scm.com/book/en/v2`

Les différents types de VCS

Les systèmes de gestion de version locaux

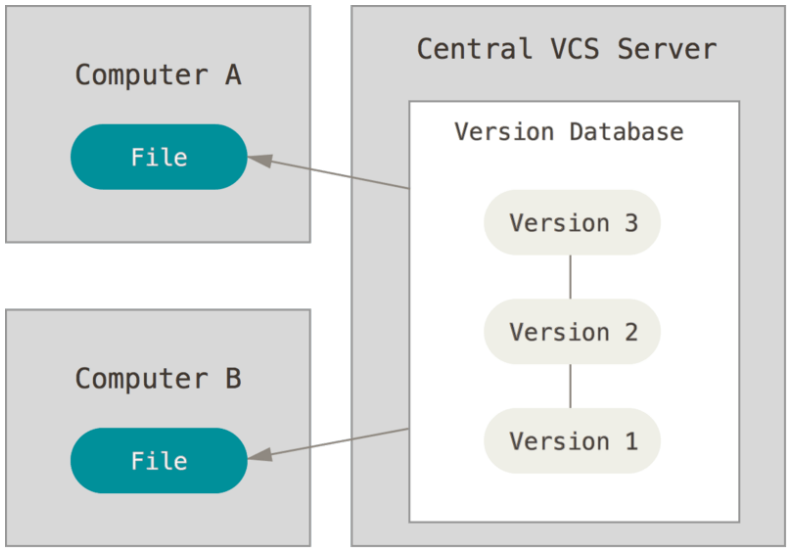
- Utilisation d'une base de données locale qui garde une trace des changements dans les fichiers suivis par le VCS.
- Amélioration du simple copier-coller de fichiers/dossiers
- Le versionning est basé sur les différences entre les fichiers (texte)



Les différents types de VCS

Les systèmes de gestion de version centralisés (CVCS)

- Un serveur central contient tous les fichiers versionnés.
- Les clients peuvent extraire ces versions et les mettre à jour.
- CVS, Subversion (SVN), Perforce, etc. fonctionnent sur ce principe.



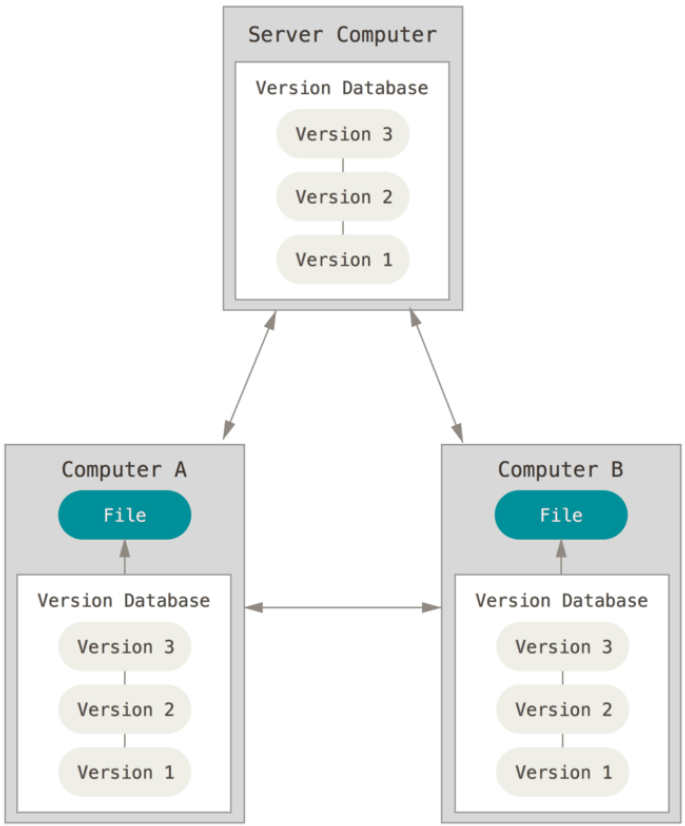
Avantages : travail collaboratif, facilité d'administration (permissions)

Inconvénients : peu tolérant aux pannes (accès au serveur, corruption des données)

Les différents types de VCS

Les systèmes de gestion de version distribués (DVCS)

- Un serveur central contient tous les fichiers versionnés.
- Chaque client récupère une copie complète de la base des versions.
- Git, Mercurial, Bazaar, Darcs fonctionnent sur ce principe.



Avantages : résistance aux pannes (chaque client peut restaurer le serveur), possibilité de travailler avec plusieurs dépôts distants simultanés.

Principes de base de Git

Origine de Git

Git est né en 2005 d'une scission entre l'équipe de développement du Kernel Linux et BitKeeper, le DVCS utilisé depuis 2002 avec un accord d'utilisation "free-of-charge" qui a été révoqué.

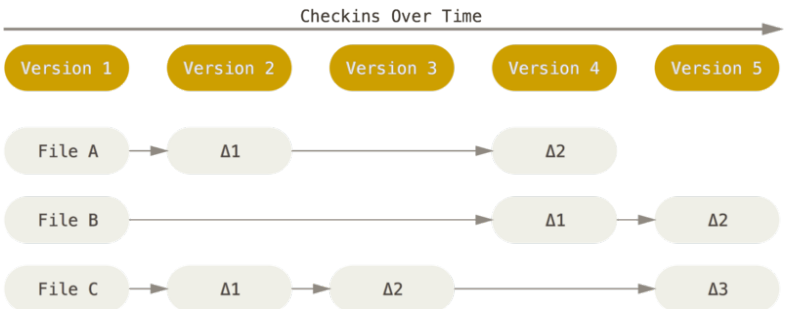
Les objectifs de Git

- rapidité
- conception simple
- bon support de développement non-linéaires (création multiples de branches)
- entièrement distribué
- capacité à gérer des projets larges

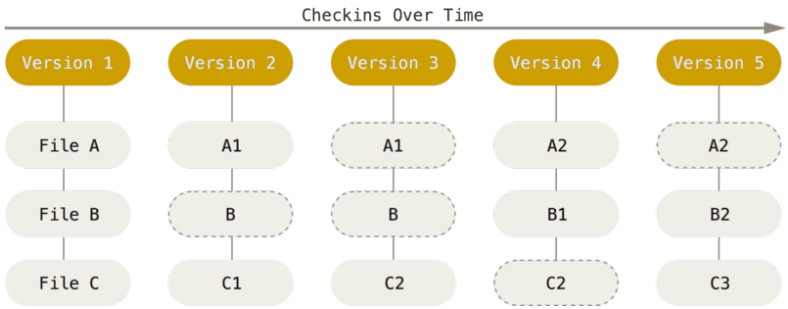
Principes de Git

Des instantanés, au lieu des différences

Contrairement à beaucoup d'autres (D)VCS, Git n'enregistre pas les changements dans les fichiers (delta-based version control). Au lieu de cela, il réalise des instantanés (snapshots), et voit les versions successives comme des séries d'instantanés d'un système de fichier miniature.



delta-based (ex. SVN)



snapshot-based (Git)

Principes de Git

(Presque) Toutes les opérations sont locales

La plupart des opérations avec Git ne nécessitent que les fichiers locaux pour être réalisées. De ce fait, il n'y a pas d'attente ou de contraintes sur la disponibilité du réseau.

Git vérifie l'intégrité

Git calcule des checksums (hash SHA-1) pour tous les fichiers et les référence via ceux-ci. En conséquence, il est impossible de changer le contenu d'un fichier sans que Git ne le détecte.
Exemple : `jube.jpg` → `24b9da6552252987aa493b52f8696cd6d3b00373`

Git ne fait (généralement) que d'ajouter des données

Il est assez rare avec Git de réaliser des actions qui suppriment du contenu qui ne peut pas être récupéré par la suite.

Principes de Git

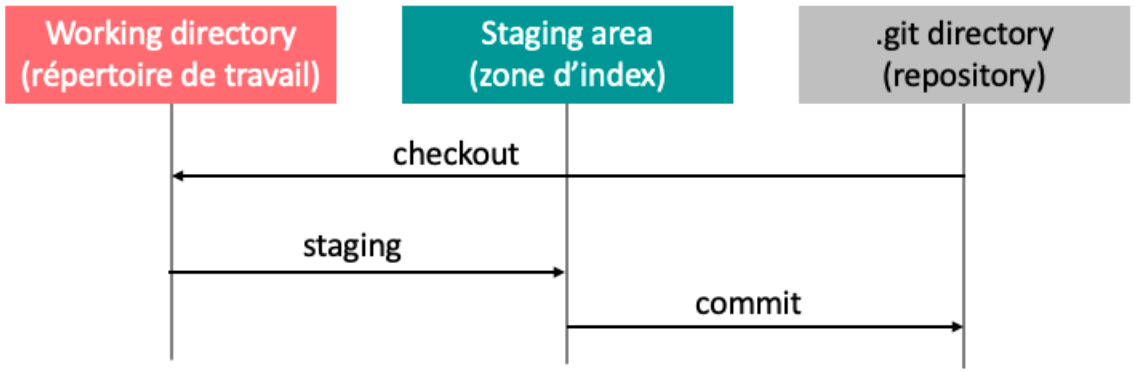
Trois états pour les fichiers...

Chaque fichier versionné est dans l'un des trois états suivants :

- *modified* : quand le fichier a été modifié mais qu'il n'a pas été envoyé (commit) à la base de données
- *staged* : quand le fichier a été marqué comme modifié et qu'il va faire partie du prochain instantané (snapshot)
- *commited* : quand le fichier a été enregistré en toute sécurité dans la base de données

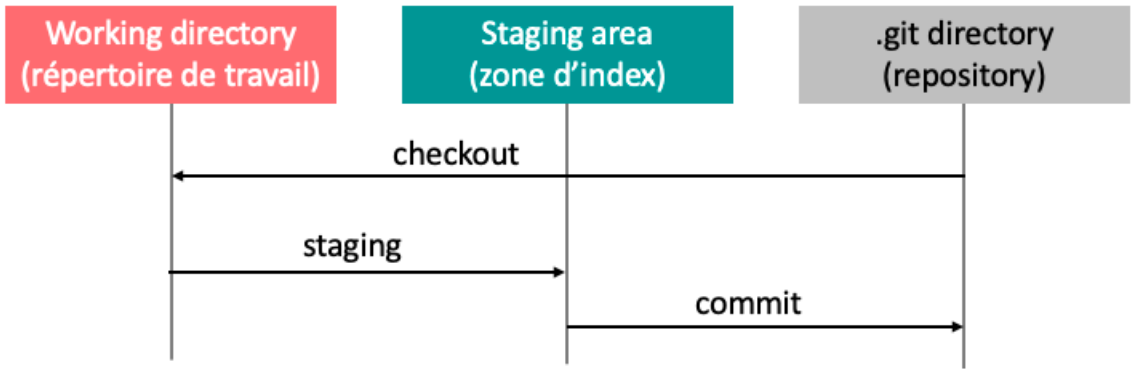
(spoiler : consultables via la commande `git status`*)*

Les trois sections d'un projet versionné



- *working directory* (répertoire de travail) : dossier dans lequel on travaille (modifie) les fichiers représentant une version du projet qui sont récupérés de la base de données compressée du dépôt.
- *staging area* (zone d'index) : un fichier, placé dans le répertoire `.git` qui enregistre les informations des fichiers qui feront partie du prochain `commit`.
- *.git directory* (repository – dépôt) : c'est dans ce dossier que Git enregistre les méta-données ainsi que la base de données du projet. C'est ce qui est copié quand on `clone` un dépôt.

Les trois sections d'un projet versionné



Workflow classique

1. on modifie les fichiers du répertoire de travail
2. on planifie (*stage*) l'ajout de ces modifications pour le prochain commit *(spoiler : commande* `git add`*)*
3. on valide (*commit*) ces modifications pour créer un nouvel instantané (snapshot) dans le dépôt *(spoiler : commande* `git commit`*)*

La ligne de commande

Les installations de Git sont trouvables très facilement sur le net, soit sous forme de paquets intégrés à votre OS (distributions linux, MacOS dans Xcode) ou téléchargeables sur le site `https://git-scm.com` qui présente aussi toute la documentation.
Dans ce qui suit, nous utiliserons la ligne de commande. Toutefois, des plug-ins pour Git existent pour différents IDE et éditeurs de code, qui offrent des raccourcis pratiques pour certaines actions usuelles.

Les commandes Git sont toujours sous la forme `git` *`commande`*. Dans ce qui suit, nous verrons les commandes et les options les plus classiques, mais nous n'aurons pas le temps d'être exhaustifs. En cas de doute sur une commande, n'hésitez pas à consulter l'aide sur celle-ci avec :

- `git help` *`commande`*
- `git` *`commande`* `--help`
- `man git-`*`commande`*

Obtenir un dépôt

Il existe deux manières d'obtenir un dépôt pour travailler avec.

1. Initialiser un dépôt à partir d'un dossier existant

On se place dans le dossier que l'on veut versionner, et on exécute la commande

- `git init`

qui a pour effet de créer le répertoire `.git` et d'initialiser la base de données.

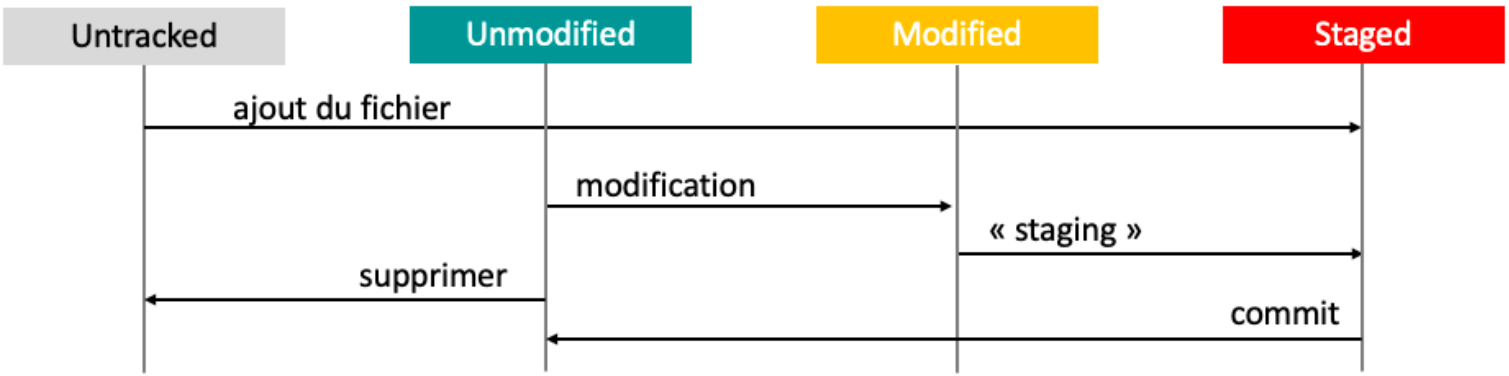
2. Cloner un dépôt distant

On se place dans le dossier où on souhaite enregistrer le projet cloné, et on exécute la commande

- `git clone` *`url_du_depot_distant.git`*

qui a pour effet de créer un répertoire de travail (working directory) contenant le projet cloné, incluant son dépôt (dossier `.git`).

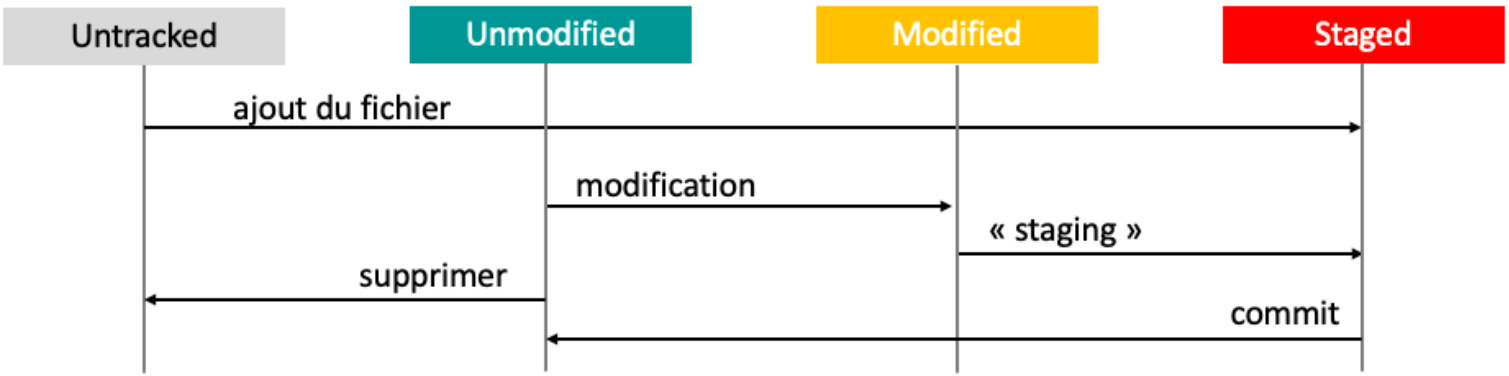
Etats d'un fichier



Untracked

Un fichier *Untracked* est un fichier du répertoire de travail qui n'est pas versionné.

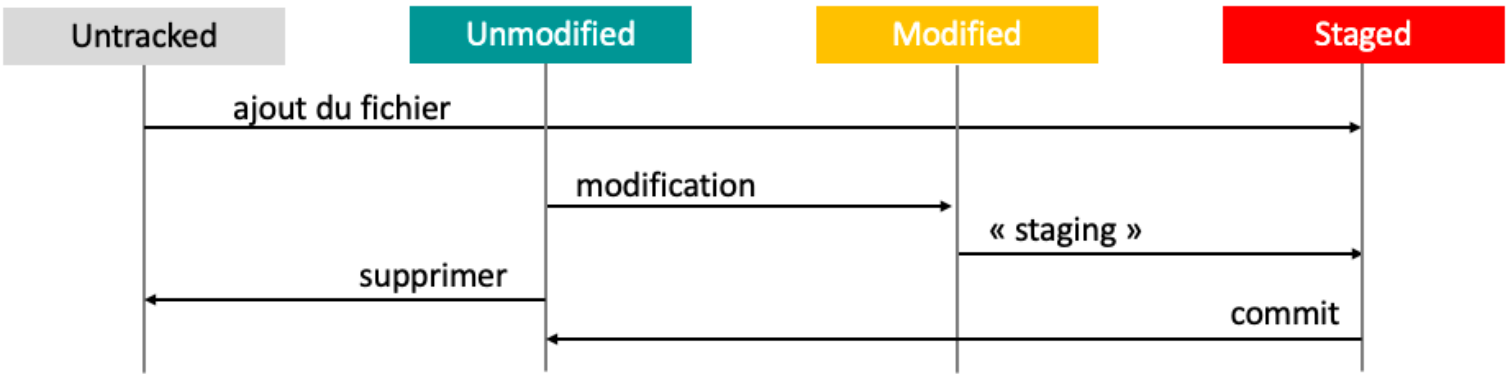
Etats d'un fichier



Unmodified

Un fichier *Unmodified* est un fichier du répertoire de travail qui est versionné et qui n'a pas de modifications par rapport à la version courante du dépôt.

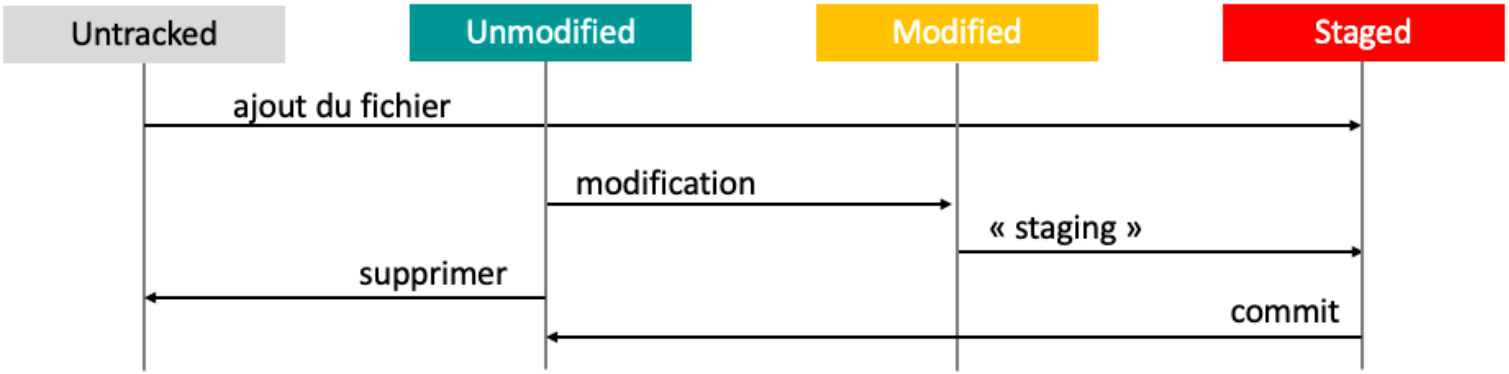
Etats d'un fichier



Modified

Un fichier *Modified* est un fichier du répertoire de travail qui est versionné et qui présente des modifications par rapport à la version courante du dépôt.

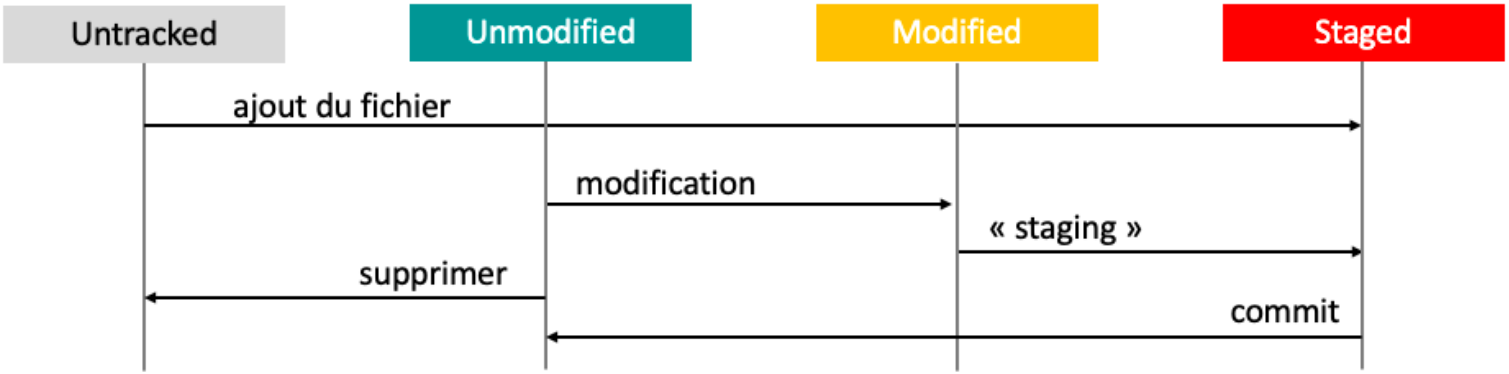
Etats d'un fichier



Staged

Un fichier *Staged* est un fichier du répertoire de travail qui est versionné, qui présente des modifications par rapport à la version courante du dépôt et qui sera pris en compte au prochain commit.

Etats d'un fichier



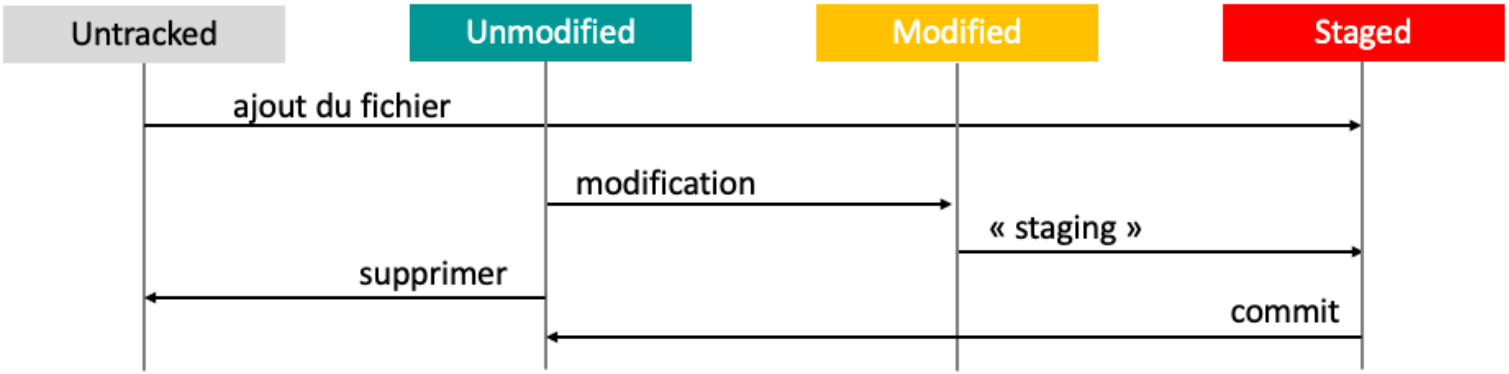
Connaître l'état courant d'un fichier

La commande

- `git status`

permet d'afficher le status des fichiers du projet.
NB. Git s'appuie sur le calcul du hash SHA-1 pour identifier si un fichier a été modifié depuis son dernier stage/snapshot.

Etats d'un fichier



Ajouter des fichiers au projet

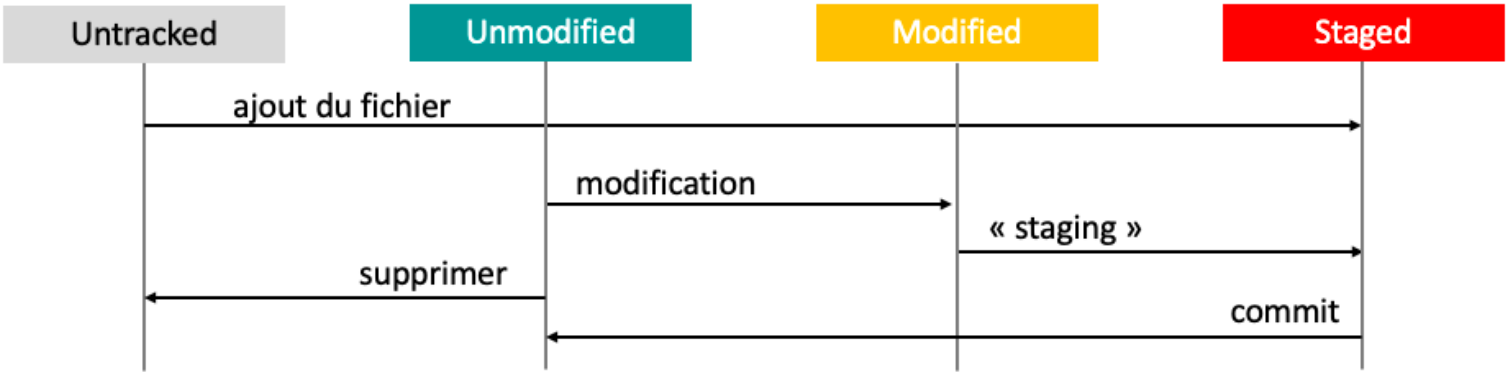
La commande

- `git add file`

permettra d'ajouter le fichier *file* au suivi. Ce fichier sera dans l'état *staged* ; il n'est pour l'instant pas encore concrètement versionné dans le dépôt.

Plusieurs fichiers peuvent être spécifiés à la suite sur la ligne de commande. L'utilisation de wildcards (ex. `*Test.java`) est également possible.

Etats d'un fichier



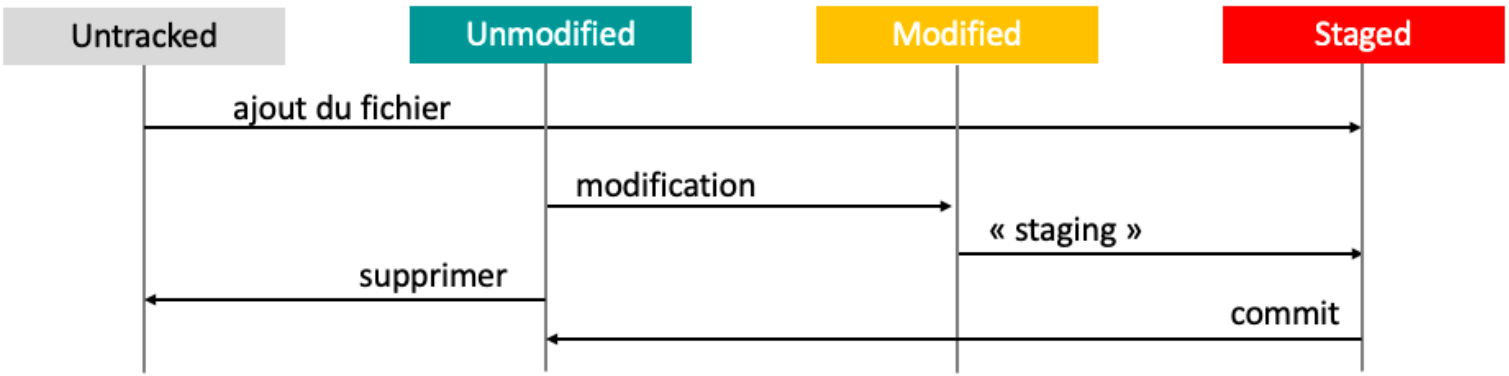
Réaliser un commit

La commande

- `git commit -m "Message explicite de commit"`

permettra de créer un nouvel instantané du projet, enregistrant ainsi les versions actuelles des fichiers qui étaient dans l'état *staged* et les passant dans l'état *Unmodified*. Si on ne précise pas le `-m "..."`, Git ouvrira un éditeur de texte pour permettre la saisie d'un message (le plus **explicite** possible) accompagnant le commit. Il est également possible de spécifier quels fichiers sont enregistrés par ce commit en les ajoutant en fin de commande.

Etats d'un fichier



Et pour les fichiers modifiés ? (1/3)

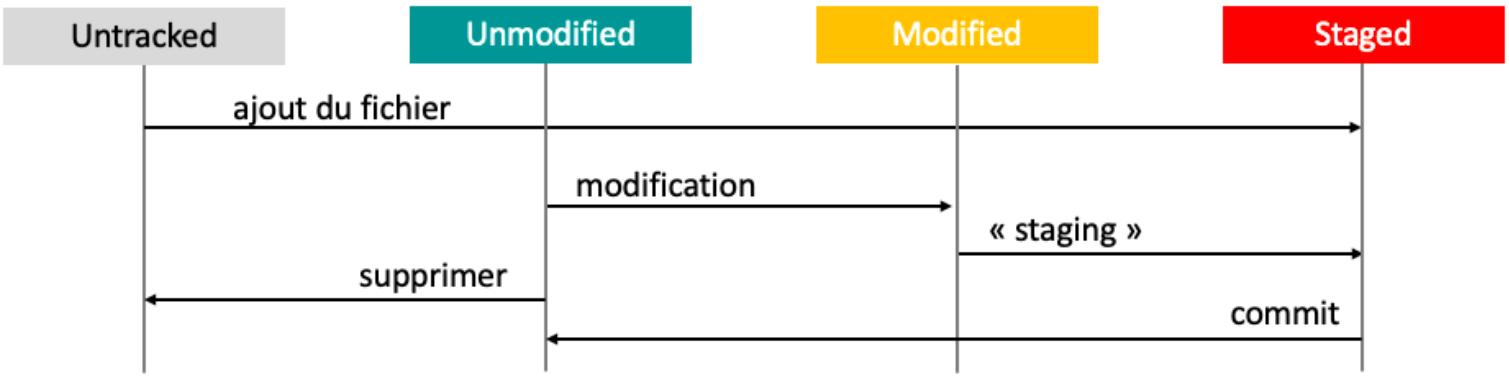
Le principe est le même :

- `git add` *`fichier_modifié`*

passe le *`fichier_modifié`* dans l'état *staged* pour qu'il fasse partie de prochain commit, qui sera réalisé avec la commande

- `git commit -m "Correction de ..."`

Etats d'un fichier



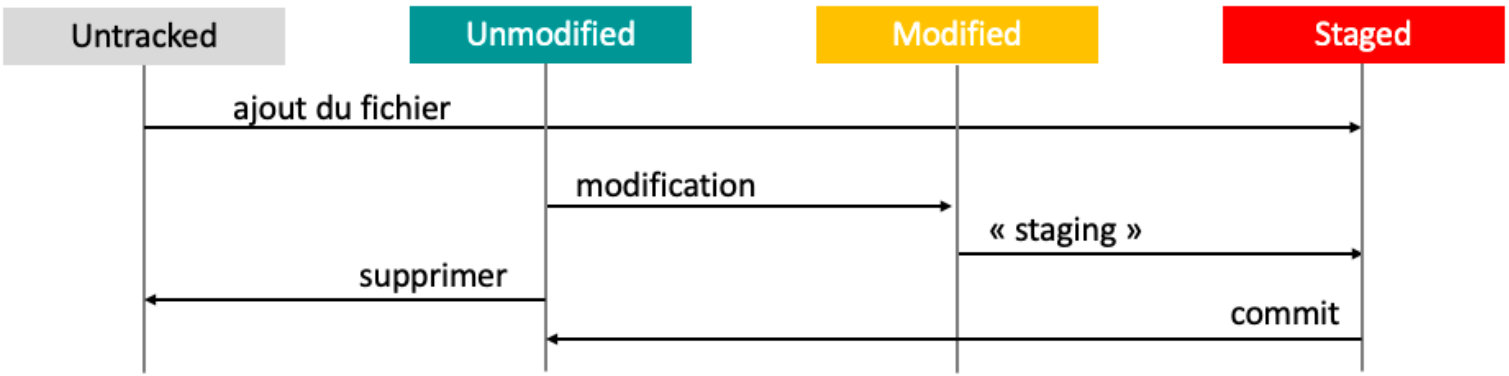
Et pour les fichiers modifiés ? (2/3)

La commande

- `git commit -am "Correction de ..."`

peut être utilisée pour remplacer les deux commandes successives précédentes. Elle a pour effet d'ajouter les fichiers modifiés au prochain commit puis de réaliser celui-ci.

Etats d'un fichier

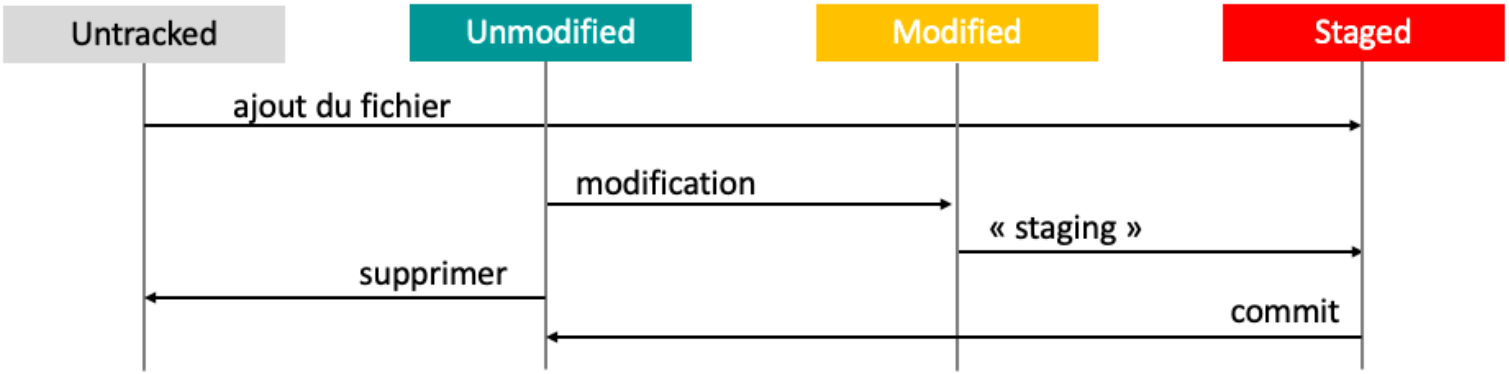


Et pour les fichiers modifiés ? (3/3)

Petite subtilité : si un fichier *staged* est modifié avant d'exécuter un `commit`, alors c'est la version *staged* de ce fichier qui sera enregistrée dans le base.

Après `commit`, ce fichier aura directement le statut *modified* puisqu'il diffère de la version enregistrée.

Etats d'un fichier



Voir les modifications

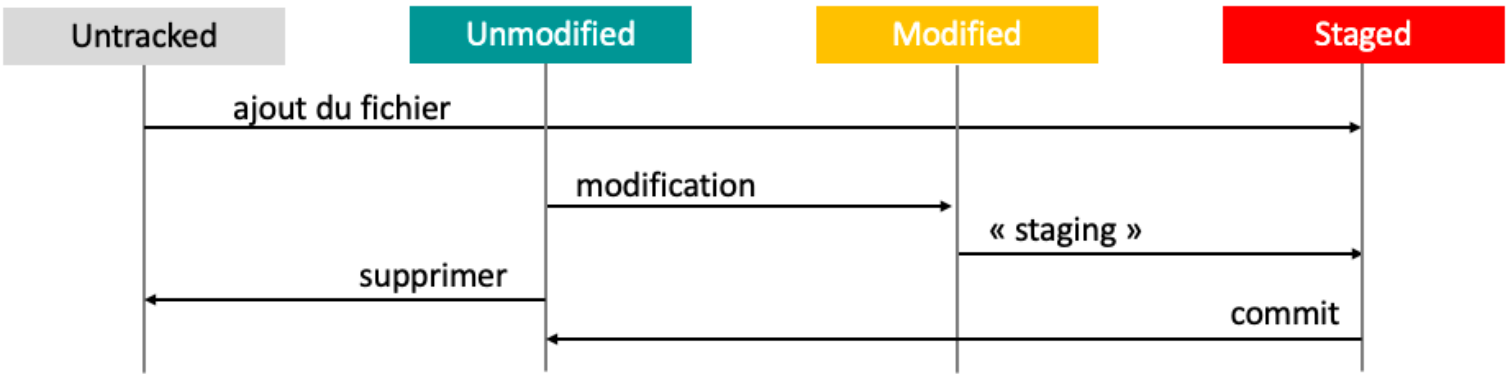
La commande

- `git diff`

permet de visualiser les modifications entre le répertoire de travail et la zone d'index (staging area).

L'option `--staged` permet de comparer la zone d'index avec le dernier commit.

Etats d'un fichier



Supprimer un fichier

La commande

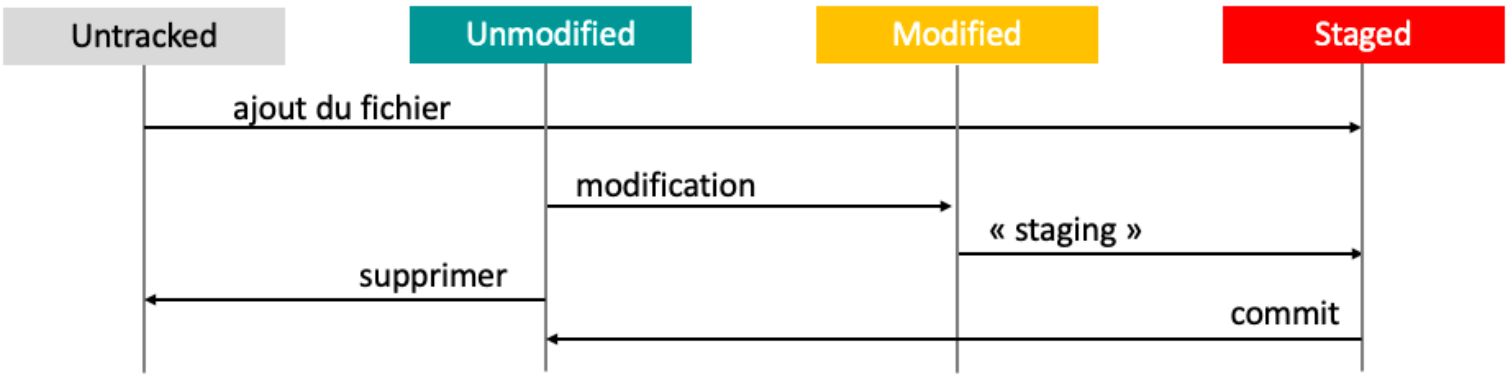
- `git rm file`

permet de supprimer le fichier `file` du répertoire de travail (!) et de la zone d'index (à valider ensuite par un `commit`).

L'option `--cached` permet de supprimer le suivi du fichier sans le supprimer du répertoire de travail.

Si le fichier est dans l'état *staged* il faut utiliser l'option `-f` (force la suppression).

Etats d'un fichier



Déplacer un fichier

La commande

- `git mv file_from file_to`

permet d'explicitement renommer un fichier. Cette commande est équivalente à :
`mv file_from file_to ; git rm file_from ; git add file_to`

Ignorer des fichiers

Il est possible, dans le dossier du projet, mais aussi dans ses dossiers de préciser dans un fichier nommé `.gitignore` les fichiers qui doivent être ignorées par les commandes Git.

Un `.gitignore` correctement paramétré évitera que fichiers non-désirés soient considérés lors de commandes un peu larges (comme par exemple `git add *`).

Que faut-il ignorer ?

- les fichiers résultant de la compilation du projet (`*.class`, `*.o`, exécutables, etc.)
- les fichiers de configuration qui sont propres à votre installation (`*.iml`, etc.)
- les dépendances à vos projets (ex. `node_modules` pour un projet Node.js)
- pour les utilisateurs de MacOS, les `.DS_Store`

Ignorer des fichiers

Il est possible, dans le dossier du projet, mais aussi dans ses dossiers de préciser dans un fichier nommé `.gitignore` les fichiers qui doivent être ignorées par les commandes Git.

Un `.gitignore` correctement paramétré évitera que fichiers non-désirés soient considérés lors de commandes un peu larges (comme par exemple `git add *`).

Que faut-il versionner ?

- tous les fichiers sources de votre projet (`*.java`, `*.c`, `*.cpp`, etc.)
- les fichiers de configuration qui permettent la construction du logiciel (fichier `pom.xml` de Maven, `package.json`, etc.)

Pour résumer, voir https://medium.com/faun/5-file-types-you-should-and-10-you-should-not-store-in-your-vcs-d03c99f37930.

A quoi ressemble mon dépôt ?

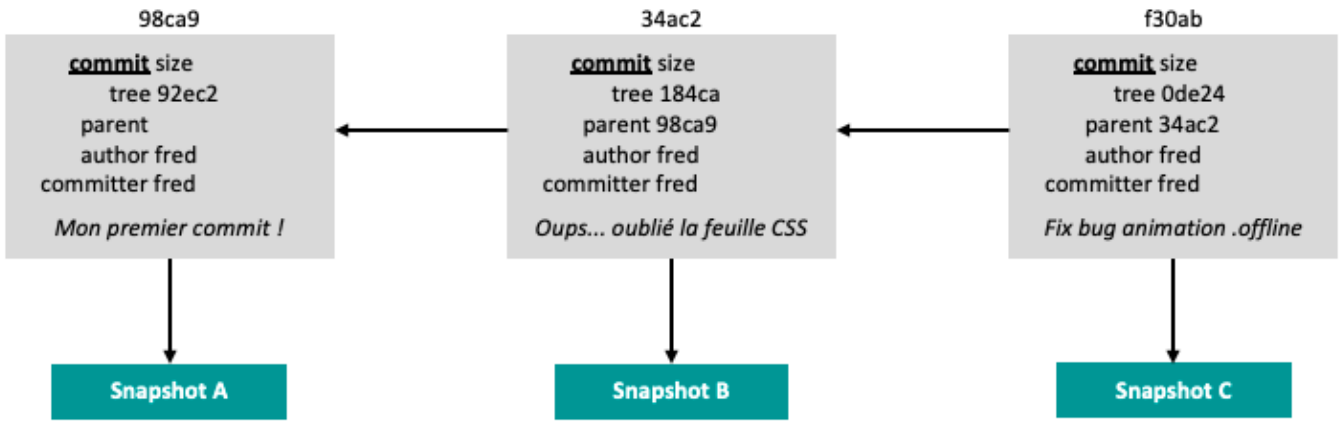
Détail d'un snapshot/instantané issu d'un `commit`

On trouve, pour chaque instantané :

- un objet *commit* qui contient les méta-données du commit (auteur, message, etc.) et un pointeur vers l'arbre
- un objet *tree* qui pointe vers chacun des fichiers versionnés
- un objet *blob* par version de fichier considéré

Tous ces objets sont désignés par des hash SHA-1 (sur 40 caractères, raccourcis ici pour simplifier)

A quoi ressemble mon dépôt ?



Plusieurs commits successifs

On trouve également, pour chaque commit, un pointeur vers le commit précédent... jusqu'au commit initial.

Visualiser les commits

Afficher l'historique des commits

La commande

- `git log`

permet d'afficher les commits successifs par ordre chronologique inversé (plus récents en premier). Elle affiche pour chaque commit son identifiant (hash), son auteur, la date, et le message associé.

Quelques options intéressantes (cumulables)

- `git log -p` permet de voir les différences d'un commit à l'autre (patch).
- `git log -N` (où N > 0) permet de limiter le nombre d'entrées du log ainsi affiché.
- `git log --stat` permet de voir les statistiques en nombre d'ajouts/suppression de lignes dans chaque fichier.
- ... et bien d'autres (voir `git log --help`)

Annuler des actions

Modifier un commit déjà effectué

Il est possible de remplacer le dernier commit effectué avec la commande

- `git commit --amend`

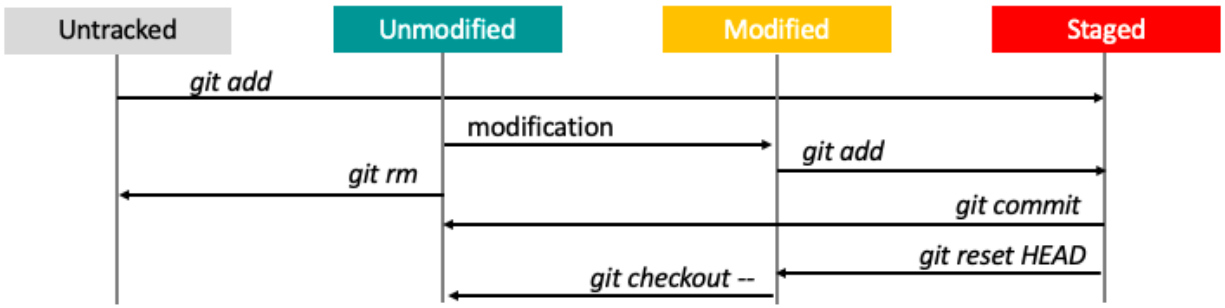
Le dernier commit sera ainsi complété pour intégrer les modifications ayant été “indexées” (avec `git add`) entretemps.

Le grand classique : le fichier oublié

En cas d’oubli d’ajout d’un fichier au dépôt, plutôt que de le rajouter et de créer un commit supplémentaire juste pour cet ajout, on peut amender le commit précédent.

- `git commit -m "Ajout de la fonctionnalité XYZ"`
- `git add styles/style.css`
- `git commit --amend`

Annuler des actions



Retirer un fichier de la zone d'index

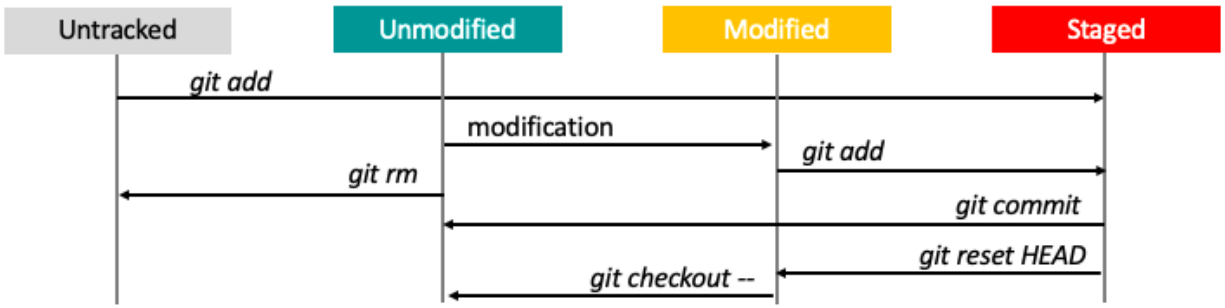
Pour retirer de la zone d'index un fichier, qui y aurait été placé avec un `git add`, on utilise la commande

- `git reset HEAD` *`fichier`*

Le *fichier* ainsi désigné repassera à l'état *modified*.

Les commandes présentées ici peuvent aussi être réalisées avec `git restore` introduit dans la version 2.25 de Git.

Annuler des actions



Annuler les modifications dans un fichier modifié

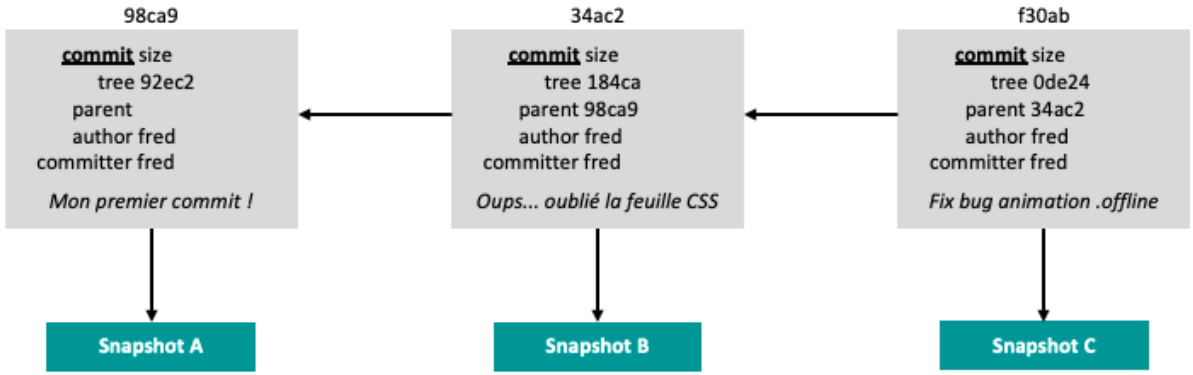
Pour supprimer les modifications courantes d'un fichier versionné, et revenir à la dernière version enregistrée, on utilise la commande

- `git checkout -- fichier`

Le *fichier* ainsi désigné sera remplacé par sa dernière version qui était enregistrée dans le dépôt.
Attention : les modifications du fichier seront définitivement perdues !

Les commandes présentées ici peuvent aussi être réalisées avec `git restore` introduit dans la version 2.25 de Git.

Restaurer un fichier



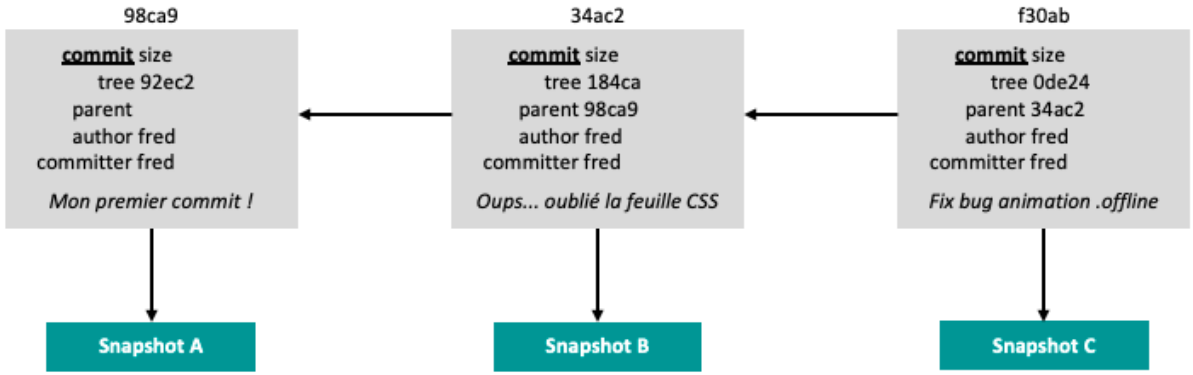
Revenir à une version antérieure d'un fichier (1/2)

Pour faire revenir un fichier à la version enregistrée à un commit spécifique (par ex. `98ca9`), on utilise la commande

- `git checkout 98ca9 -- fichier`

Le *fichier* ainsi désigné sera remplacé par la version qui était enregistrée dans le commit `98ca9` dans le dépôt.
Attention : les modifications du fichier seront définitivement perdues !

Restaurer un fichier



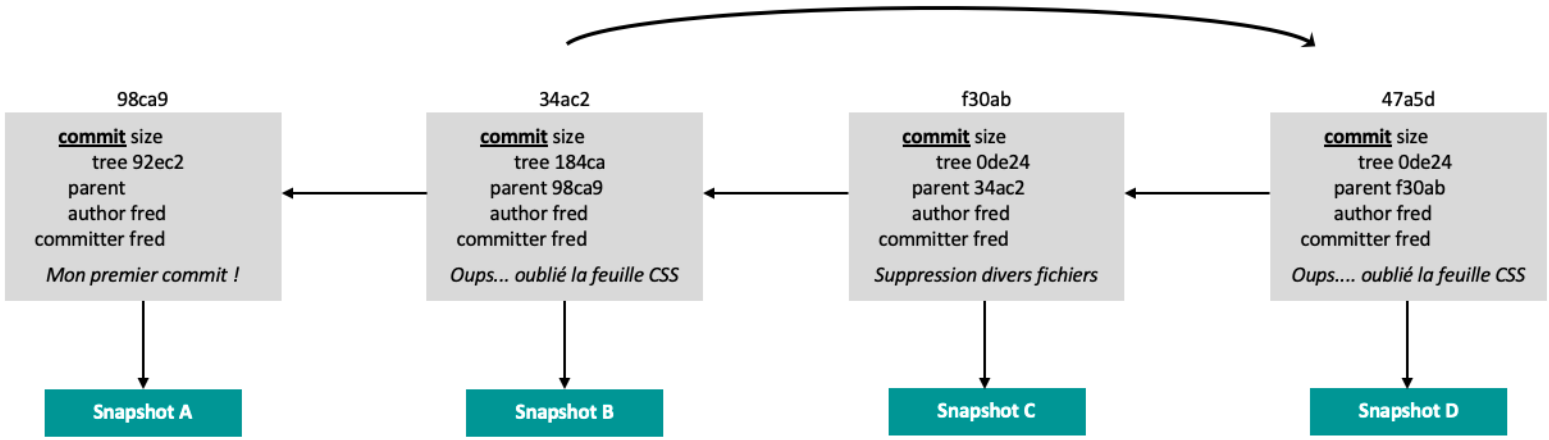
Revenir à une version antérieure d'un fichier (2/2)

Pour faire revenir un fichier à une version enregistrée dans un commit dont on connaît l'antériorité (par ex. 2 commits avant le commit courant), on utilise la commande

- `git checkout HEAD~2 -- fichier`

Le *fichier* ainsi désigné sera remplacé par la version qui était enregistrée dans le 2e commit précédent le commit le plus récent (HEAD) dans le dépôt.

Revenir à une ancienne version du dépôt



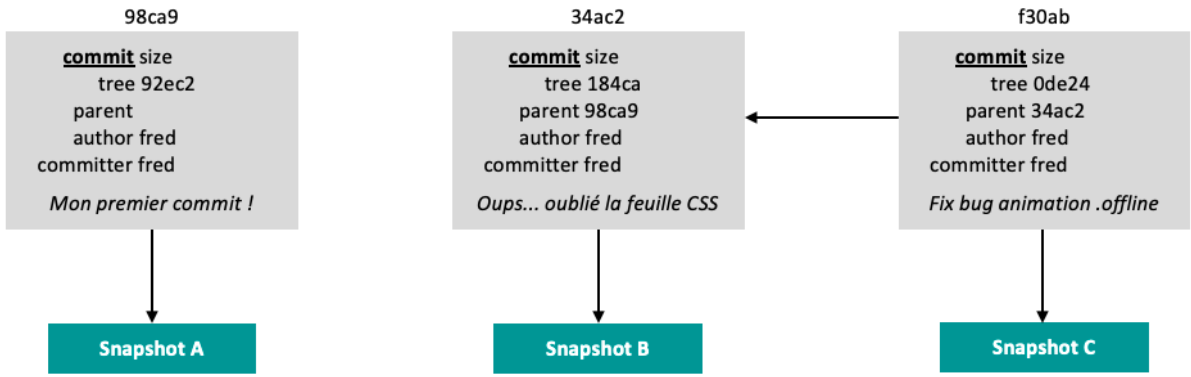
Appliquer les modifications d'un commit

La commande

- `git revert 98ca9`

aura pour effet de revenir aux modifications introduites par les patches du commit spécifié, et de créer un nouveau commit qui les enregistre.
Cette commande est utile pour annuler l'effet de commits antérieurs, sans pour autant les supprimer de l'historique, contrairement à `git reset` qui revient à une version antérieure mais "détache" les commits suivants !

Revenir à une ancienne version du dépôt



Revenir à une version antérieure

La commande

- `git reset 98ca9`

aura pour effet de faire revenir l'état courant du dépôt au commit spécifié. Ceci a pour effet de détacher les commits ultérieurs.

Ceux-ci seront potentiellement éliminés par Git automatiquement après 30 jours, à moins d'être récupérés entretemps à l'aide de la commande `git reflog` (non détaillée dans ce cours).

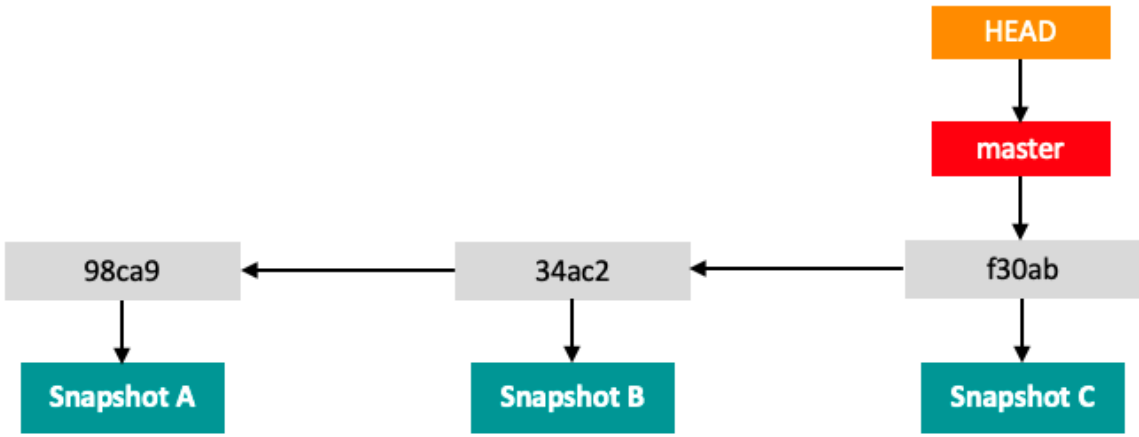
Les branches de Git

La notion de branche est, dans Git, une de ses principales fonctionnalités. Les branchements sont des dérivations dans la ligne de développement qui permettent de continuer à travailler sur des versions successives du projet sans affecter la ligne principale.

Les VCS permettent généralement de créer des branches (dérivations), de naviguer entre les versions sur celles-ci et de fusionner des branches (c'est le moment où les conflits apparaissent...)

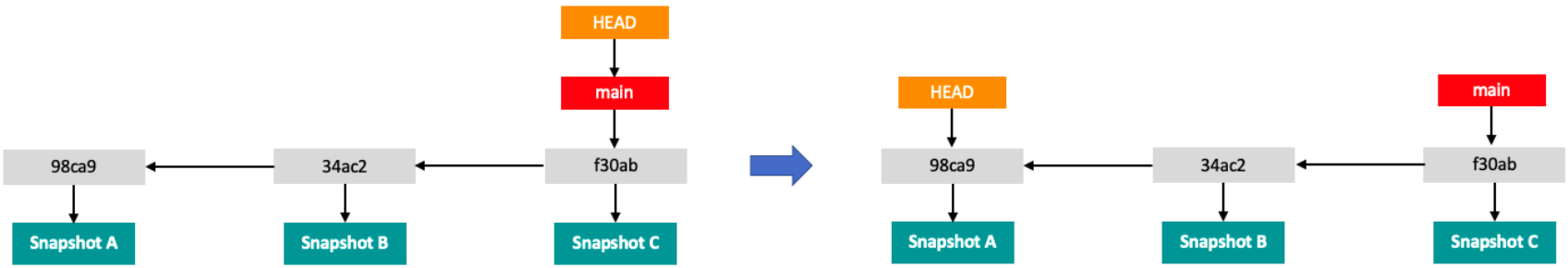
Contrairement aux autres VCS, dans lesquels la création de branches est lourde et potentiellement très coûteuse en temps et en mémoire car il revient à dupliquer entièrement son arborescence, Git propose un système de branche très léger, et très simple à mettre en oeuvre pour créer une branche, changer de branche de travail, etc.

Représentation des branches de Git



- Chaque branche porte un nom. Dans Git, une branche est un pointeur sur un commit particulier.
- La branche principale par défaut se nommait précédemment `master`, elle ne nomme désormais `main` depuis peu.
- A chaque commit, le pointeur `main`, placé sur le dernier commit réalisé, avance automatiquement.
- HEAD est un pointeur spécial qui désigne la branche courante.

Se promener dans les versions



Détacher le pointeur HEAD

La commande

- `git checkout 98ca9`

sans spécifier de fichier aura pour conséquence de détacher le pointeur HEAD. Cela permettra de visualiser l'ancienne version du dépôt correspondant au commit `98ca9`. De là, il sera par exemple, possible de créer une nouvelle branche.
Le retour au dernier commit s'effectuera grâce à la commande :

- `git checkout main`

NB. Git vous signalera l'entrée dans ce mode "détaché".

Créer des branches

Créer une branche

La commande

- `git branch <nom_nouvelle_branche>`

permet de créer une branche portant le nom spécifié en paramètre.
En pratique, cette commande créé un nouveau pointeur sur le commit courant.

Après l'exécution de `git branch testing` :

Identifier les branches


```
$ git log --decorate --oneline

f30ab (HEAD -> main, testing) Fix bug animation .offline
34ac2 Oups... oublié la feuille CSS
92ec2 Mon premier commit !
```

Identifier les branches

Lister les branches existantes

La commande

- `git branch [-v]`

permet de connaître les branche existantes (l'option `-v`) ajoute le détail des commits sur lesquels pointent chacune des branches.

```
* main    f30ab Fix bug animation .offline
* testing f30ab Fix bug animation .offline
```

Naviguer entre les branches

Changer de branche

La commande

- `git checkout <branche_cible>`

permet de changer de branche courante. En pratique, cette commande déplace le pointeur HEAD sur le pointeur de la branche cible et restaure dans le répertoire de travail le commit correspondant au pointeur HEAD.

Quelques remarques

- la commande `git checkout -b nouvelle` créé une nouvelle branche et bascule sur celle-ci.
- depuis Git 2.23, la commande `git switch` peut être utilisée pour changer de branche L'option `-c` permet de créer la nouvelle branche, l'option `-` permet de revenir à la branche précédente.

Naviguer entre les branches

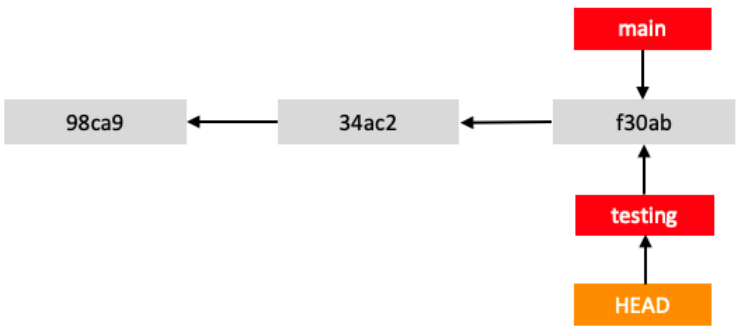
Changer de branche

La commande

- `git checkout <branche_cible>`

permet de changer de branche courante. En pratique, cette commande déplace le pointeur HEAD sur le pointeur de la branche cible et restaure dans le répertoire de travail le commit correspondant au pointeur HEAD.

Après l'exécution de `git checkout testing` :



Utiliser la remise

Les changements de branches impliquent que le dépôt local soit à jour sous peine de perdre ses modifications actuelles.

Quand un développeur a besoin de changer de branche rapidement, il peut mettre en remise ses modications pas encore enregistrées dans le dépôt local (avant commit). Cette fonctionnalité lui permet de ne pas se presser à effectuer un commit bâclé pour pouvoir changer de branche.

Remiser

La commande

- `git stash push "message"`

permet de remiser le contenu de la zone d'index et de réinitialiser celle-ci.

Sortir de la remise

La commande

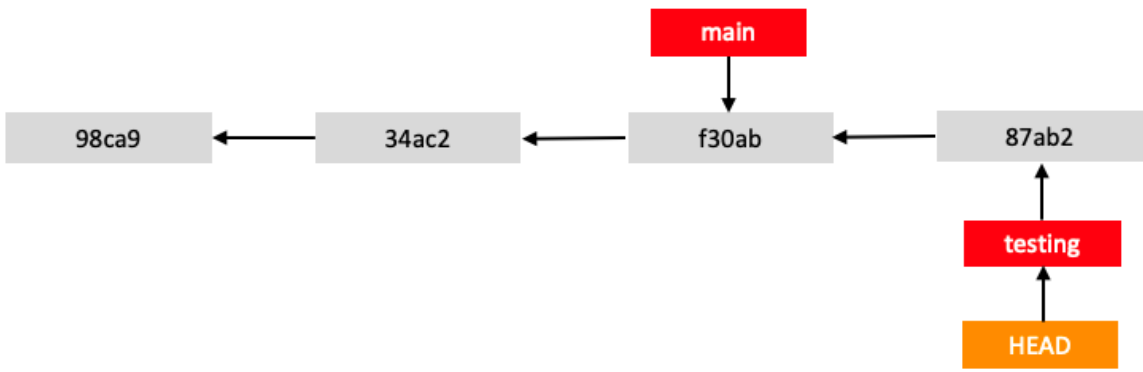
- `git stash pop`

permet de ressortir les modifications remisées et les supprime de la remise.

Créer une dérivation

Après changement de branche, les prochains commits créent des dérivations.

- `git checkout testing`
- `...` création et travail sur le fichier `test42.js`
- `git add test42.js`
- `git commit -m "Ajout tests pour fonctionnalité 42"`

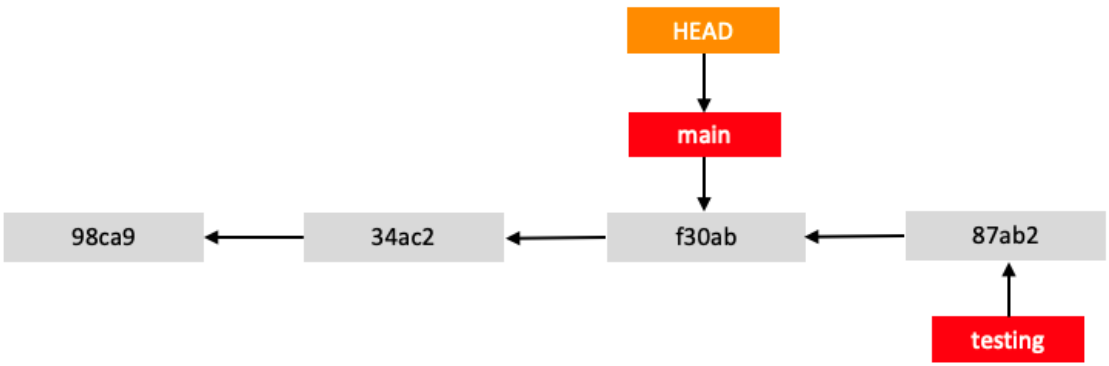


Changement de branche

Après changement de branche, le contenu du répertoire de travail évolue.

```
git checkout main
```

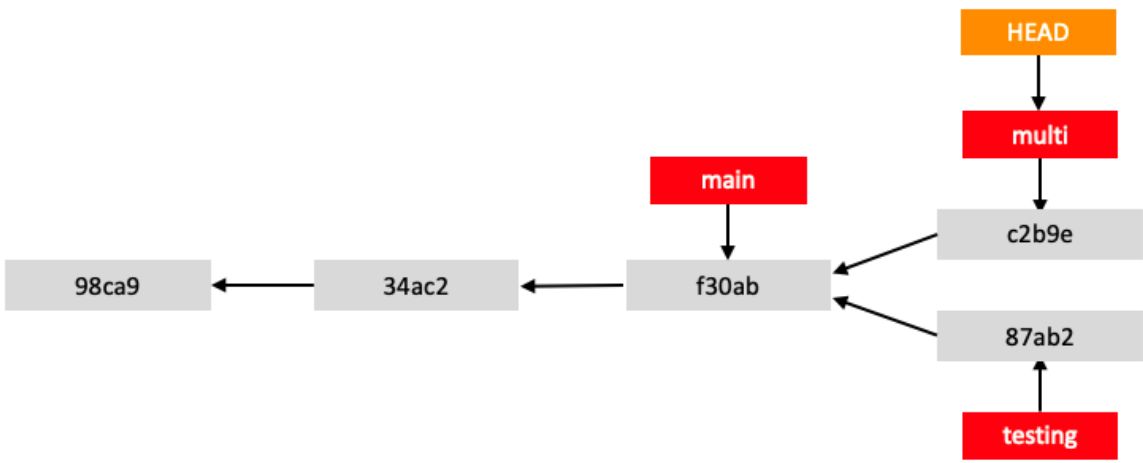
⇒ le fichier `test42.js` créé pour la branche `testing` n'existe pas dans cette branche du projet.



Créer une dérivation

Après changement de branche, les prochains commits créent des dérivations.

- `git checkout -b multi`
- `git add multiplayer.js`
- `git commit -m "Ajout multijoueur"`



Fusionner des branches

Fusionner des branches

La commande

- `git merge <nom_branche>`

permet de fusionner la branche courante avec la branche passée en paramètre.

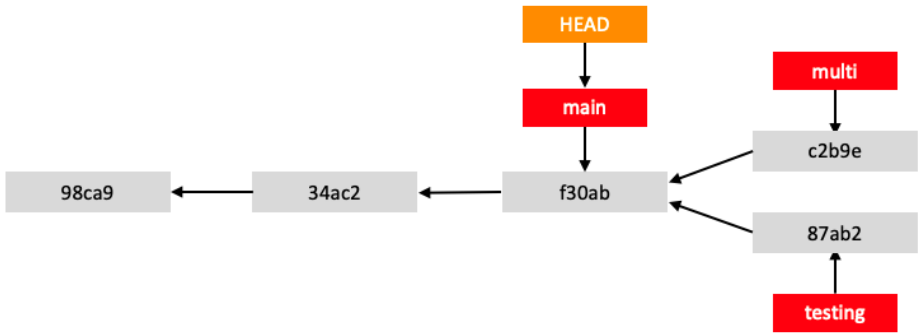
Réaliser une fusion

L'objectif d'une fusion de branche est d'obtenir un commit dans lequel les données des deux commits en tête de chacune des branches sont fusionnés.
Deux possibilités :

- La branche courante est un prédécesseur de la branche fusionnée (on fusionne avec un pointeur en aval de la position courante) : la fusion consistera alors à ramener le pointeur de la branche courante sur celui de la branche fusionnée.
- La branche courante et la branche à fusionner ont un ancêtre commun : un nouveau commit est généré qui possède la particularité d'avoir deux parents (chacune des deux branches fusionnées). La branche courante est ramenée sur ce commit.

Fusionner des branches

Cas “simple” : la branche courante est dans l’historique de la branche avec laquelle on veut fusionner. L’exécution de la commande indiquera `Fast-forward` qui symbolise l’avancée du pointeur courant vers celui de la branche fusionnée.




```
git merge multi
```

Supprimer une branche

La suppression d'une branche consiste à supprimer le pointeur qui existe pour la branche correspondante.

Supprimer une branche

La commande

- `git branch -d <branche_cible>`

permet de supprimer la branche passée en paramètre.

Il n'est pas possible de supprimer la branche sur laquelle on se trouve actuellement. Avec l'option `-d` la branche cible devra avoir été fusionnée avec la branche courante pour permettre sa suppression.

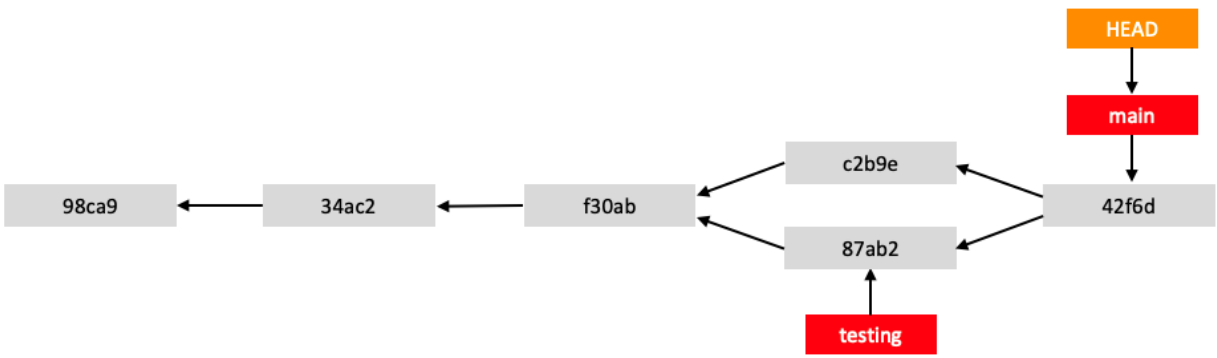
Supprimer des branches


```
git branch -d multi
```

Fusionner des branches

Cas “pénible” : les deux branches sont sur des dérivations différentes ; elles ont un ancêtre commun. Git créé un nouveau commit qui fusionne les deux précédents.


```
git merge testing
```



Conflits lors des fusions

Tant que l'on fusionne des fichiers différents/distincts ou qui n'ont pas été modifiés, alors la fusion se passe généralement sans problème.

S'il existe des différences dans le même fichier, Git peut tenter de les fusionner "proprement" (auto-merge), en particulier si les différences entre les fichiers ne touchent pas les mêmes lignes.

Ces conflits de fusion (merge conflicts) surviennent lorsque deux développeurs ont modifié le même fichier aux mêmes lignes, ou qu'un développeur a supprimé un fichier qu'un autre utilisait.

Conflits lors des fusions

Création d'un conflit

- `git checkout testing`
- `...` modifie la 3e ligne de `index.html`
- `git commit -am "Ajout charset dans l'entête"`
- `git checkout main`
- `...` modifie (aussi) la 3e ligne de `index.html`
- `git commit -am "Ajout author dans l'entête"`
- `git merge testing`

Conflits lors des fusions

Résultat de l'exécution

```
$ git merge testing
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Visualisation avec `git status`

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified: index.html
```

Conflits lors des fusions

Contenu du fichier `index.html` du répertoire de travail

```
<<<<<<< HEAD
<meta name="author" content="Fred Dadeau">
=======
<meta charset="UTF-8">
>>>>>>> testing
```

Pour résoudre le conflit

- Editer le(s) fichier(s) contenant les erreurs de fusion (mentionnés par `git status`)
- `git add` sur chaque fichier en conflit pour marquer le conflit comme résolu.
- `git commit` pour valider ces modifications.

Autre solution : annuler la fusion avec `git merge --abort`

Connaitre les branches fusionnées

Les options `--merged` et `--no-merged`

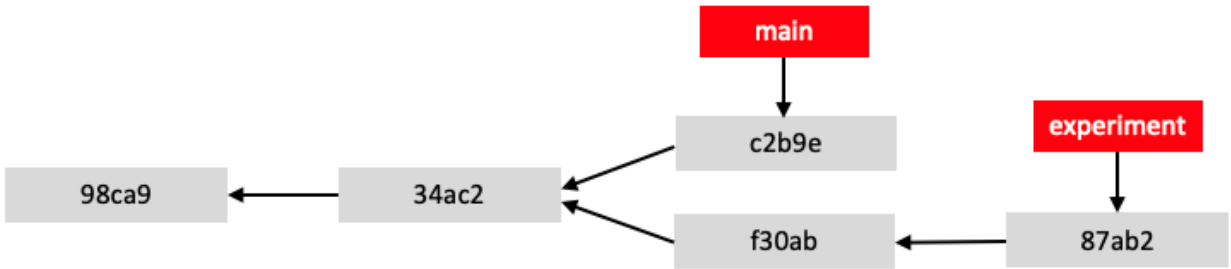
La commande

- `git branch --merged` (resp. `git branch --no-merged`)

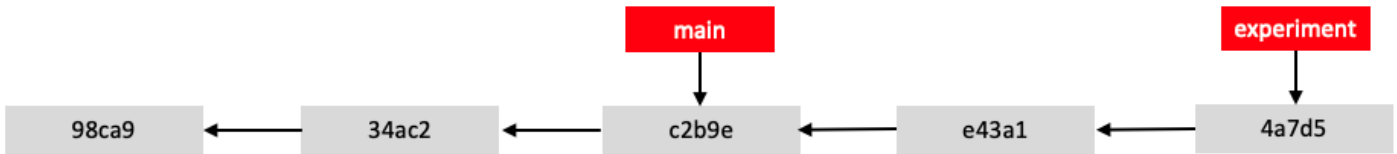
permet de ne lister que les branches qui ont été fusionnées (resp. n'ont pas été fusionnées) avec la branche courante.

Alternative à la fusion : `rebase`

“Rebaser” est une alternative à la fusion de branches qui consiste à appliquer les patches successifs de la branche courante sur une autre branche. L’idée est ainsi de réappliquer les changements qui ont été faits depuis le point de divergence entre les deux branches.




```
git checkout experiments
git rebase main
```



Pour finir, on fusionne et on peut supprimer la branche `experiments`.

Alternative à la fusion : `rebase`

Les conflits sont toujours possibles...

Si les patches provoquent des conflits parce qu'ils ne peuvent pas être appliqués automatiquement, l'opération de rebasage s'interrompt et Git vous redonne la main :

- on peut résoudre le conflit et continuer `git rebase --continue`
- on ignorer le conflit et passer au patch suivant `git rebase --skip`
- on peut annuler le rebase et revenir à la branche d'origine `git rebase --abort`

`rebase` VS. `merge`

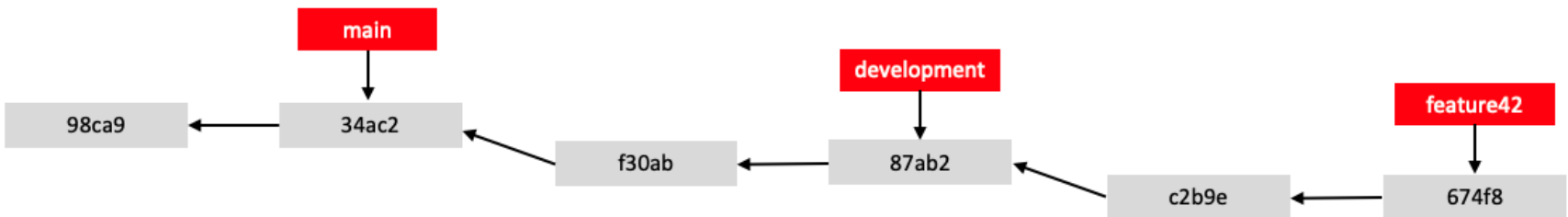
- `rebase` va avoir pour effet d'aplatir l'historique présent dans le log : même s'il y a eu des dérivations, celles-ci seront invisibles une fois le rebase effectué.
- A l'inverse, `merge` conserve l'historique des dérivations ayant existé.

Le résultat étant finalement le même, le choix est potentiellement plus philosophique que technique : souhaite-t-on laisser visible dans l'historique les tergiversations de l'équipe de développement ou veut-on montrer une progression propre et linéaire ?

Schémas classiques de branchements

Développement en silos

Cette pratique consiste utiliser des branches longues et à conserver la branche principale en amont des développements plus avancés. Les fusions successives consisteront alors à des `fast-forwards` qui ne devraient pas créer de conflits.

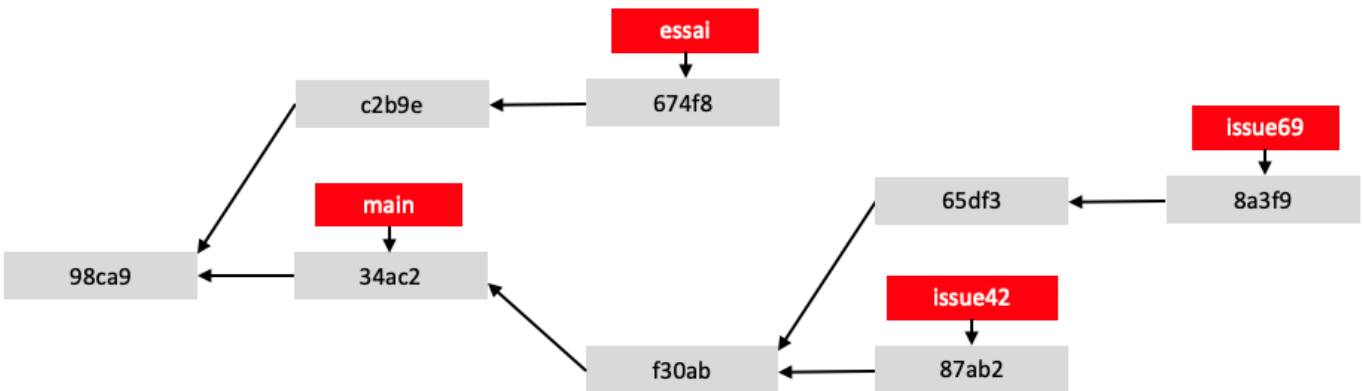


On imagine que la branche principale conserve la dernière version stable du projet (résultant par exemple de l'itération de développement précédente) et qu'une branche de développement représente l'itération courante, à laquelle on ajoute des fonctionnalités intégrées successivement. Lorsque la branche de développement est jugée suffisamment stable, elle fusionne avec la branche principale.

Schémas classiques de branchements

Développements par topics

Un *topic* est une branche de courte qui est créée pour une fonctionnalité spécifique, qui peut être destinés à être intégrée plus tard ou qui sont juste des essais. Dans ce schéma, les branches de topics sont fusionnées régulièrement avec les autres branches.



Bien évidemment, les fusions et rebases sont potentiellement plus complexes à réaliser, et souvent source de conflits... mais on peut s'en sortir.

Dépôts distants

Les dépôts distants sont des versions du projet qui sont hébergées sur des serveurs sur Internet ou sur un réseau d'entreprise. Ils permettent le travail collaboratif en donnant la possibilité de "pousser" (push) ou de "retirer" (fetch, pull) des données pour partager son travail.

Lister les dépôts distants (1/2)

La commande

- `git remote`

permet de lister les dépôts distants associés au projet.

Dépôts distants

Les dépôts distants sont des versions du projet qui sont hébergées sur des serveurs sur Internet ou sur un réseau d'entreprise. Ils permettent le travail collaboratif en donnant la possibilité de "pousser" (push) ou de "retirer" (fetch, pull) des données pour partager son travail.

Lister les dépôts distants (2/2)

La commande

- `git remote -v`

donne plus de détails en spécifiant pour chaque dépôt distant les URL utilisées pour rapatrier (fetch) et écrire (push).

Par exemple :

```
origin http://github.com/fdadeau/truc.git (fetch)
origin http://github.com/fdadeau/truc.git (push)
```

Ajouter/supprimer des dépôts distants

Chaque dépôt distant est connu par un nom donné au moment de l'ajout du dépôt.

Si le projet a été initialisé en clonant (`git clone`) un dépôt distant existant, alors celui-ci apparaîtra sous la dénomination `origin`.

Ajouter un dépôt distant

La commande

- `git remote add <nom_depot> <url>`

permet de raccrocher un dépôt distant au dépôt courant.

Une fois le dépôt distant ajouté, il est visible via la commande `git remote [-v]`.

Ajouter/supprimer des dépôts distants

Chaque dépôt distant est connu par un nom donné au moment de l'ajout du dépôt.

Si le projet a été initialisé en clonant (`git clone`) un dépôt distant existant, alors celui-ci apparaîtra sous la dénomination `origin`.

Renommer un dépôt distant

La commande

- `git remote rename <ancien_nom> <nouveau_nom>`

permet de changer le nom du dépôt distant.

Ajouter/supprimer des dépôts distants

Chaque dépôt distant est connu par un nom donné au moment de l'ajout du dépôt.

Si le projet a été initialisé en clonant (`git clone`) un dépôt distant existant, alors celui-ci apparaîtra sous la dénomination `origin`.

Supprimer un dépôt distant

La commande

- `git remote remove <nom_depot>`

permet de supprimer le dépôt distant (`remove` peut-être raccourci par `rm`).

Le dépôt distant ne sera donc plus accessible pour rapatrier des données ou en envoyer.

Rapatrier depuis un dépôt distant

Rapatrier les données d'un dépôt distant

La commande

- `git fetch <nom_remote>`

permet de rapatrier dans le dépôt local les informations du dépôt distant.

Remarque sur `git fetch`

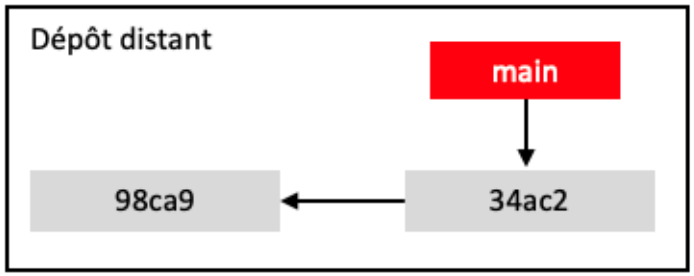
Cette seule commande ne modifie pas le répertoire de travail, elle ne fait que télécharger les données dans le dépôt local, et recréé localement les branches nommées `<remote>/<branche>` qui pointent sur les commits du dépôt distant (pointeur non modifiable).

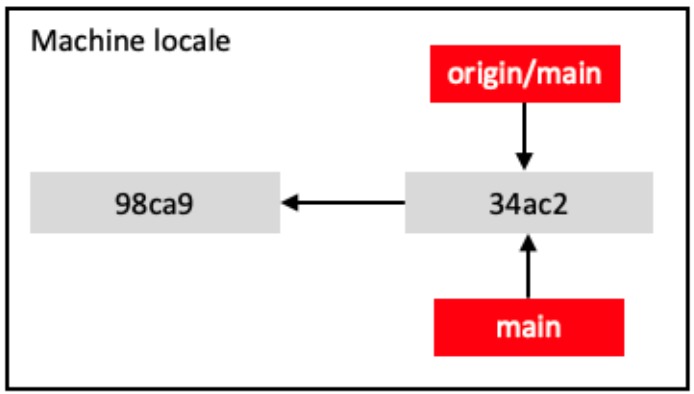
Pour modifier le répertoire de travail, il faudra par la suite fusionner celles-ci avec la branche courante.

Ces branches sont pour l'instant détachées des branches locales.

Rapatrier depuis un dépôt distant

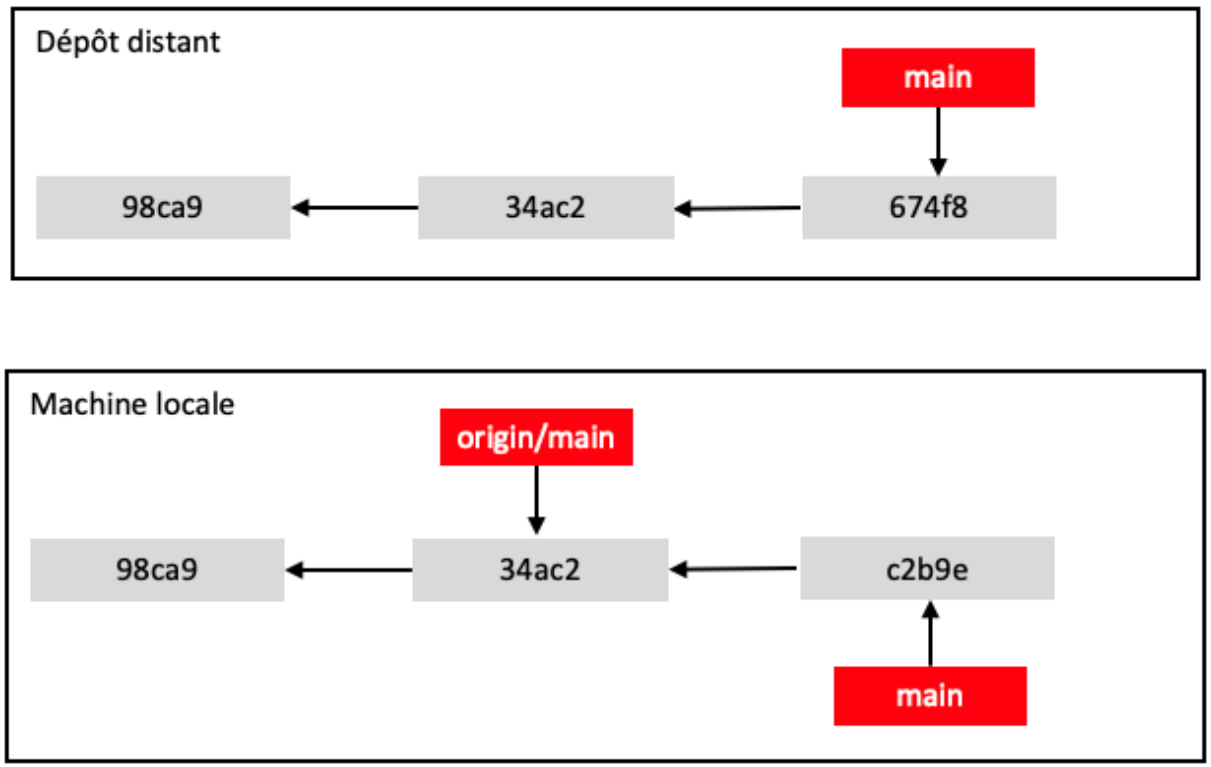
```
git clone URLduProjetSurServeurDistant.git
```





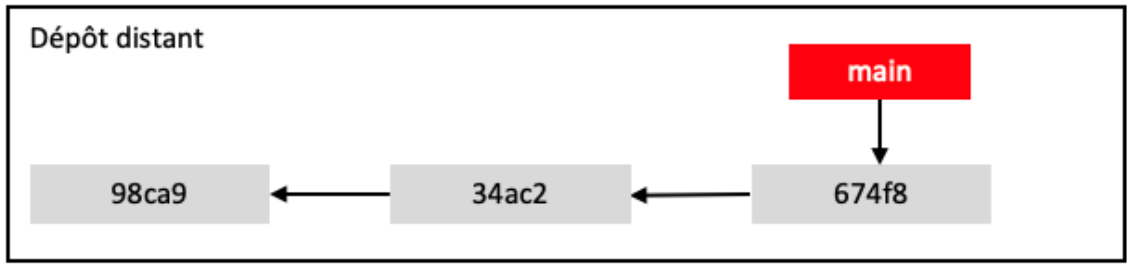
Rapatrier depuis un dépôt distant

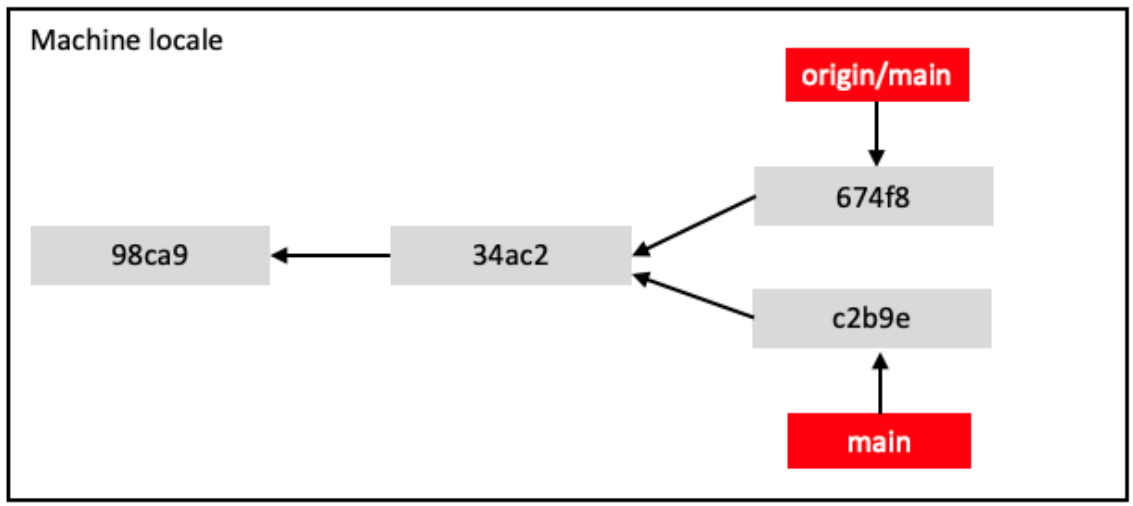
Les dépôts évoluent indépendamment l'un de l'autre.



Rapatrier depuis un dépôt distant

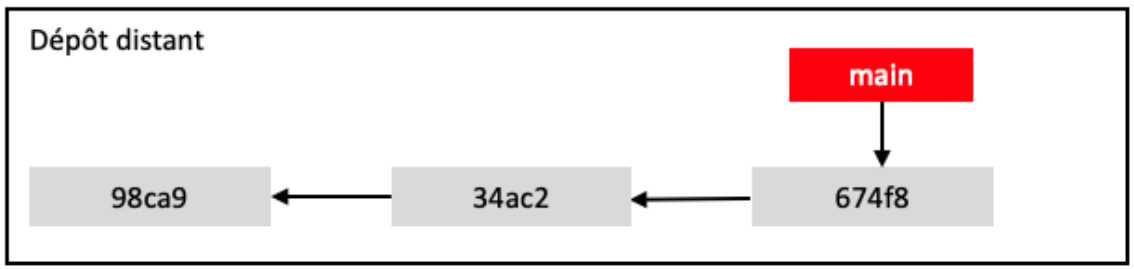
Mise à jour en deux temps : d'abord `git fetch origin`

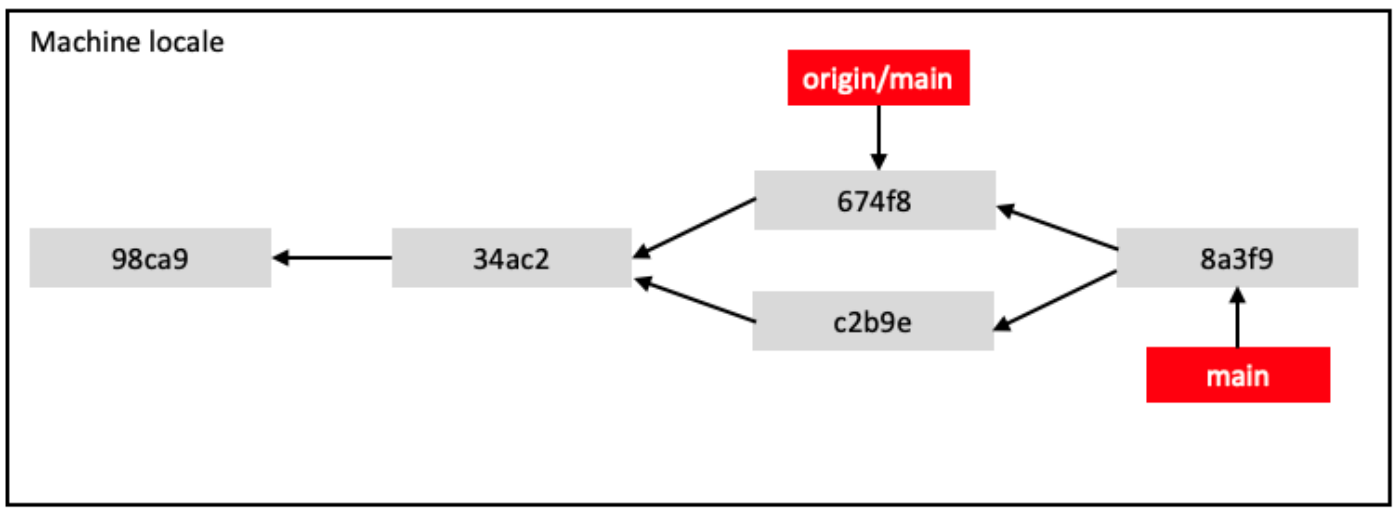




Rapatrier depuis un dépôt distant

Mise à jour en deux temps : ensuite `git merge origin/main`





Rapatrier depuis un dépôt distant

Suivre une branche distante

La commande

- `git branch -u <remote>/<branche>`

permet de lier la branche courante à la branche distante du dépôt passé en paramètre. Ceci évite de devoir préciser la branche distante à mettre à jour.

Rapatrier et fusionner en même temps

La commande

- `git pull <remote>`

combine l'effet des commandes `fetch` et `merge`. Un nouveau commit de fusion est créé sur lequel pointe la branche courante.

Variante : `git pull --rebase <remote>` applique une stratégie de rebasage.

Dépôt distant et branches

Clonage et branches distantes

Les branches d'un dépôt cloné ne sont pas toutes rapatriées localement lors de l'appel à la commande

- `git clone <URL>`

Par défaut, seule la branche principale est récupérée et liée au dépôt local.

Visualiser et rapatrier les branches distantes

La commande

- `git branch -a`

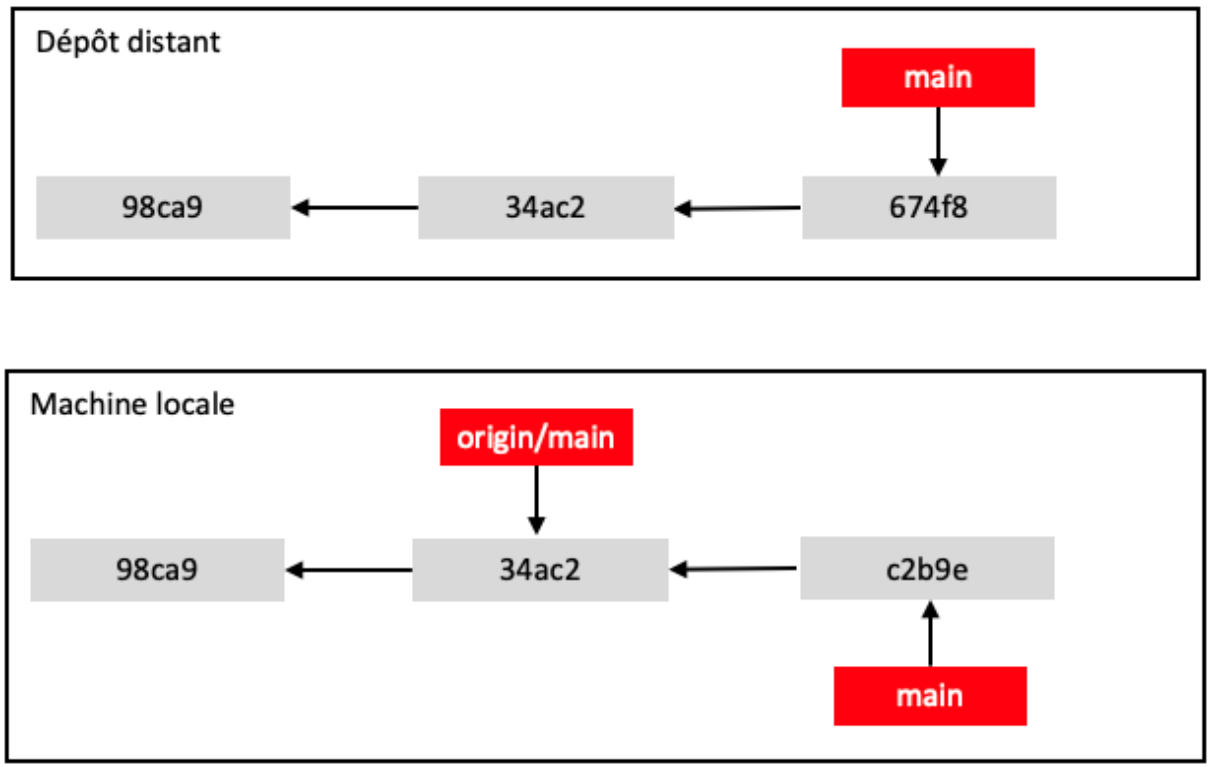
permet de visualiser les branches distantes existantes. Pour rapatrier une de ces branches et basculer dessus, il faut utiliser la commande

- `git checkout <nom_branche_distante>`

La branche distante sera alors rapatriée en local, automatiquement liée à celle-ci, et la bascule sur cette branche sera opérée.

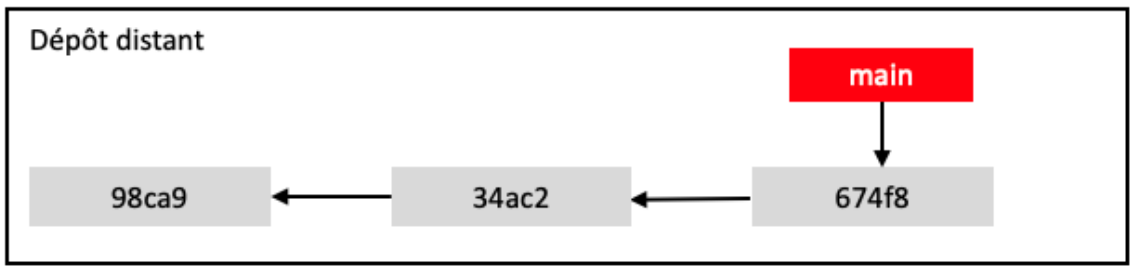
Rapatrier depuis un dépôt distant

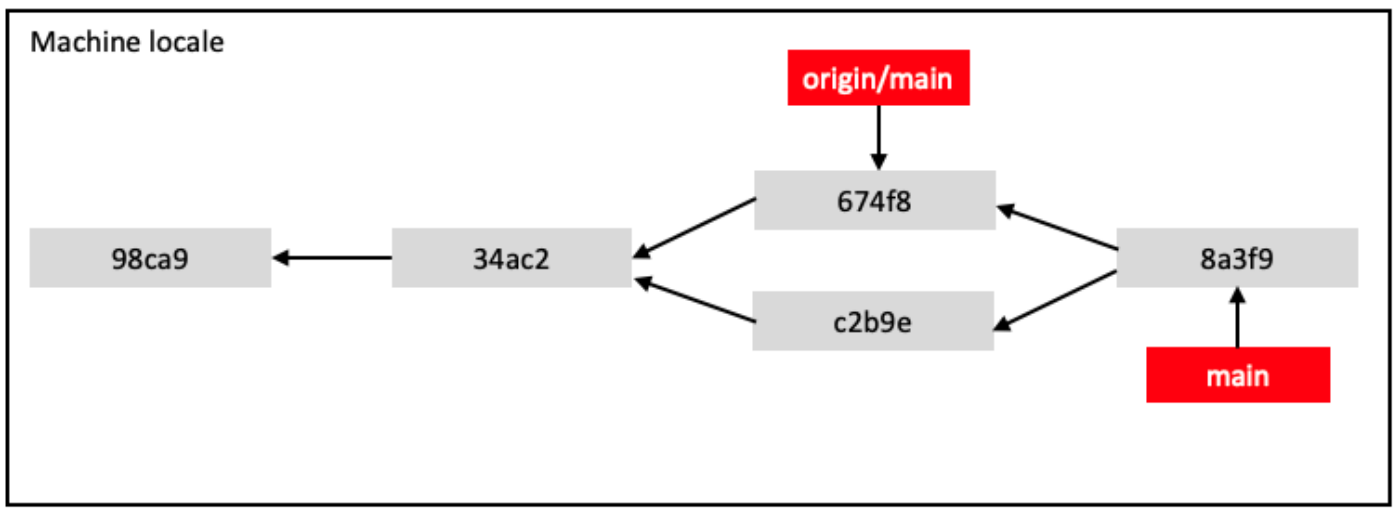
Les dépôts évoluent indépendamment l'un de l'autre.



Rapatrier depuis un dépôt distant

Mise à jour en une seule fois : `git pull origin`





Partager son dépôt

Lorsque le dépôt arrive dans un état satisfaisant (par exemple lorsqu'il a été suffisamment testé...) il est possible de le partager pour le mettre à disposition de ses collaborateurs.

Pousser son dépôt local vers un dépôt distant

La commande

- `git push <nom_remote> <branche>`

permet d'envoyer la branche spécifiée vers le dépôt distant, avec tous les commits et objets internes nécessaires. Pour éviter toute suppression de commits, Git ne vous autorise pas à faire un push si cela engendre un merge sans fast-forward dans le dépôt cible.

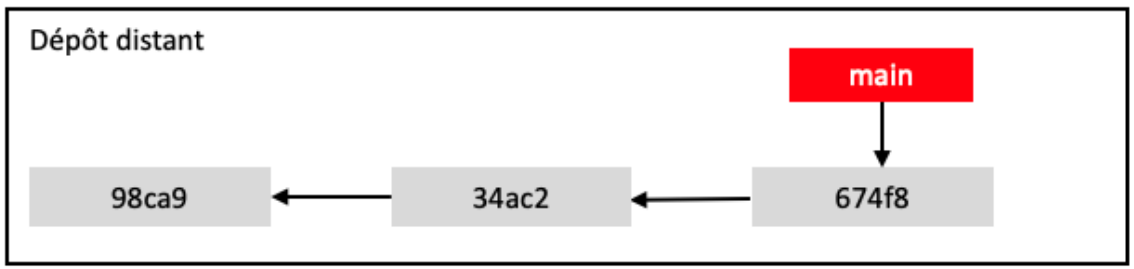
Push forcé

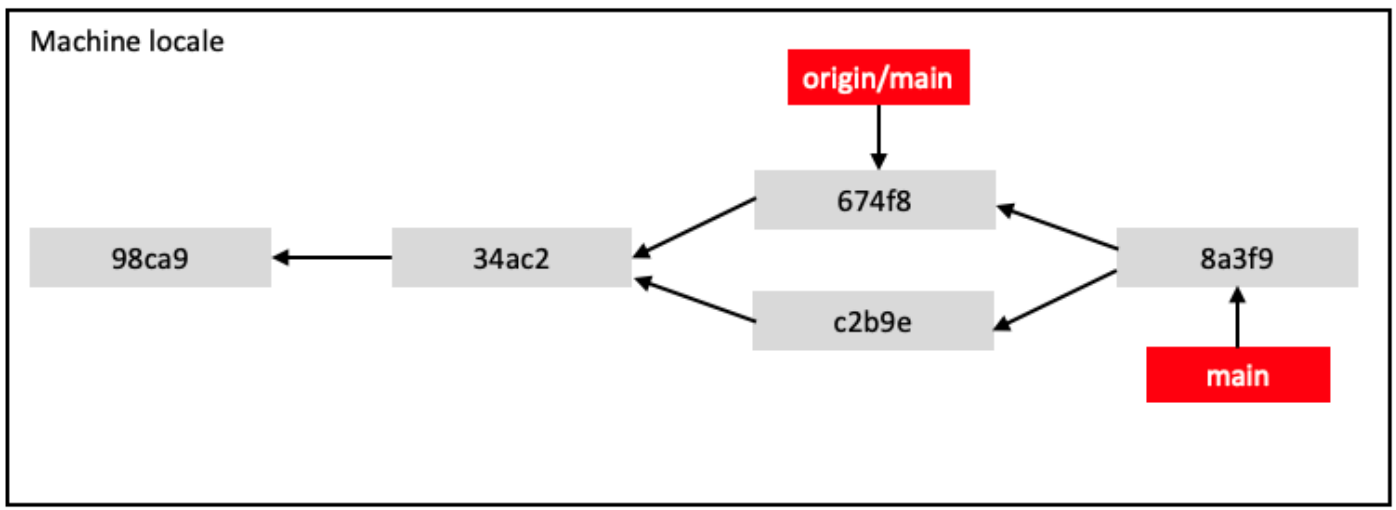
Même si l'option `--force` existe pour pousser son dépôt vers le dépôt distant en contournant la restriction évoquée ci-dessous, il est recommandé de ne pas l'utiliser.

On préférera mettre à jour son dépôt local par rapport au dépôt distant (`git pull`), régler les éventuels conflits de fusion, et partager ensuite une version "propre".

Partager son dépôt

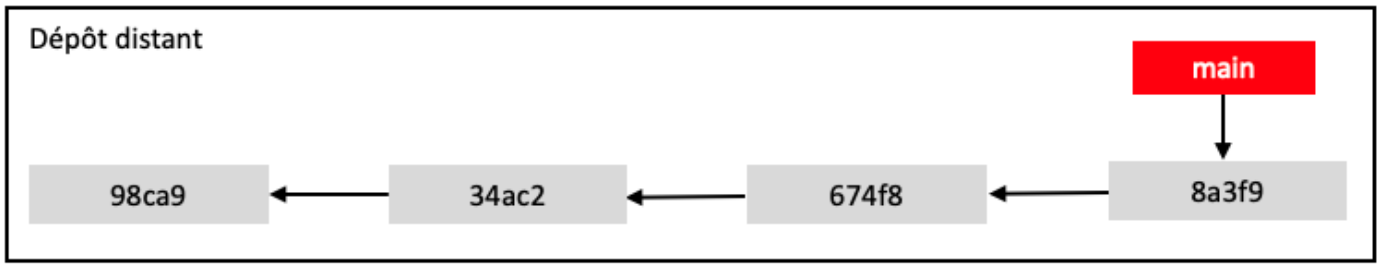
Les dépôt local est à jour mais en avance par rapport au dépôt distant.

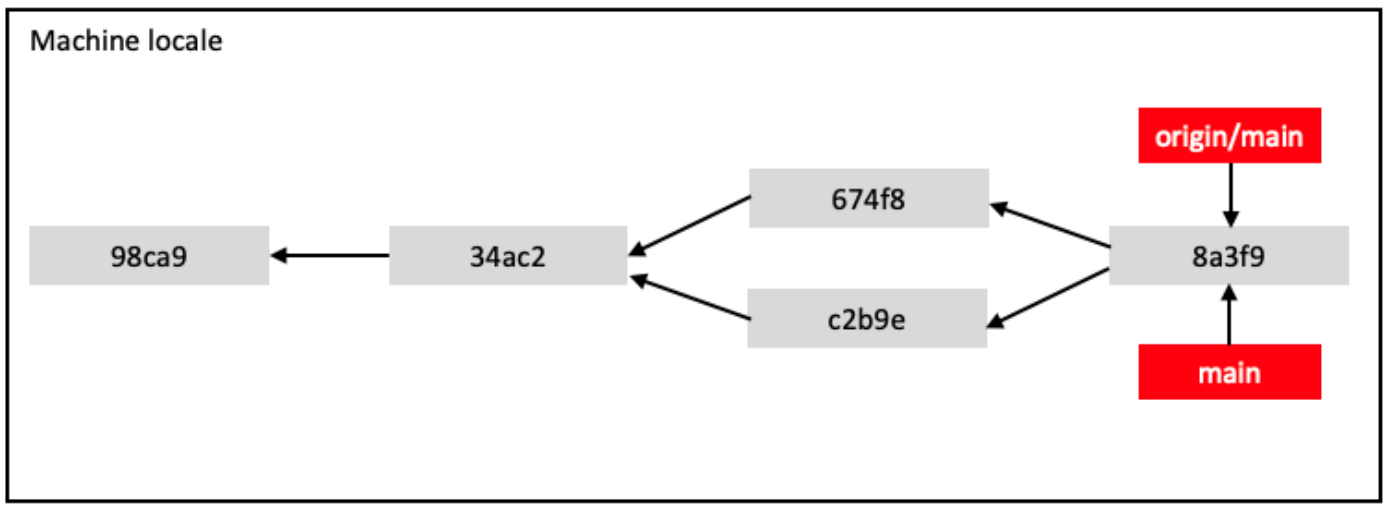




Partager son dépôt

Partage aux autres collaborateurs : `git push origin main`





Ressources sur Git

Hébergeurs

- `https://github.com`
- `https://gitlab.com`
- ... plein d'autres

Références sur l'outil

- Documentation officielle : `https://git-scm.com/docs`
- Ouvrage Pro Git disponible sur : `https://git-scm.com/book/fr/v2`
- Git Cheat Sheet : `http://ndpsoftware.com/git-cheatsheet.html`

+ quantité de plug-ins pour les IDE (IntelliJ) ou les éditeurs de code (VSCode, etc.)

Plan du cours

CI/CD

Ces trois termes désignent des “continuous methodologies” :

Intégration continue (Continuous Integration – CI)

Pour chaque mise à jour du dépôt distant, un script automatise la construction et le test du logiciel, garantissant que le code sur le dépôt est toujours correct [a].

a. dans la limite de la capacité de détection d’erreurs de la base de tests

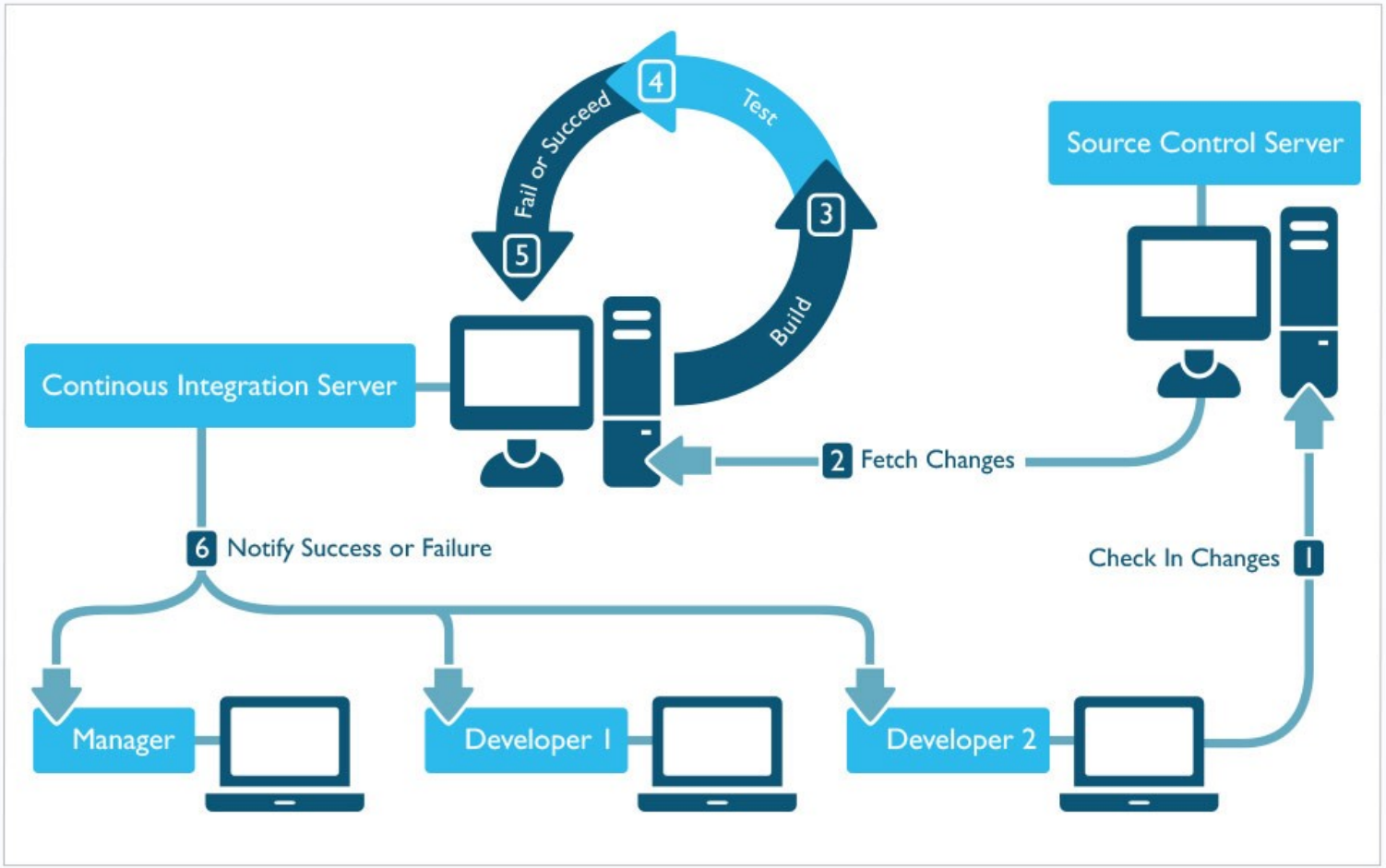
Livraison continue (Continuous Delivery – CD)

A tout moment du cycle de développement d’un logiciel, il est possible de réaliser la livraison celui-ci (étape manuelle). Ceci suppose d’avoir toujours une version “stable” et un minimum fonctionnel du logiciel ($\Rightarrow$ utilisation de branches).

Déploiement continu (Continuous Deployment – CD)

Cette pratique consiste en l’automatisation de la phase de livraison décrite ci-dessus.

Schéma intégration continue



Logiciels d'automatisation de production

Vous connaissez Maven que l'on manipule en TP, mais il en existe beaucoup d'autres : ANT, Gradle, CMake, etc.

Ces logiciels de gestion et d'automatisation de production de logiciels sont particulièrement utiles pour ces tâches CI/CD :

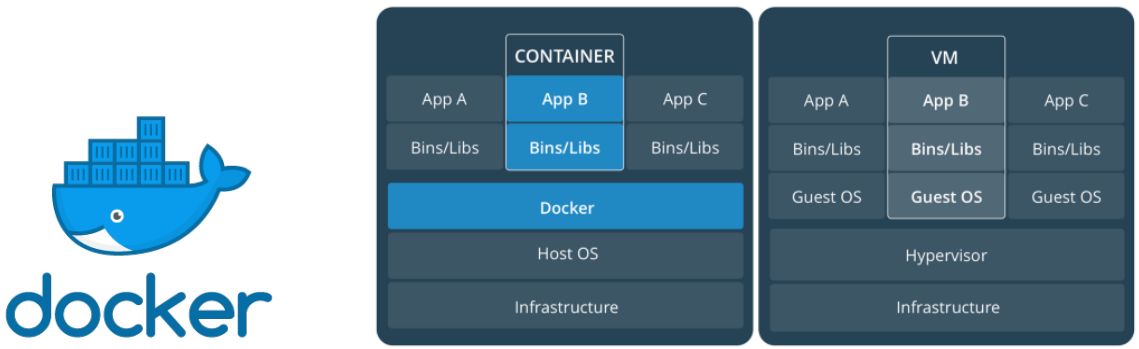
- gestion automatique des dépendances (déclarées dans un fichier de configuration, puis importées à la volée)
- construction du logiciel en une seule ligne de commande
- passage automatique des tests et production de rapports d'exécution
- déploiement de l'application (création d'une distribution, mise en ligne d'un site, etc.)

Des environnements CI/CD dans le cloud

Pour permettre de gérer les dépôts logiciels (notamment Git vu dans ce cours), différentes solutions “dans le nuage” existent : GitHub, GitLab, Heroku (web) etc. Ceux-ci proposent généralement des services de type CI/CD.

Beaucoup de ces outils s’appuient sur des environnements virtualisés qui permettent de construire le logiciel dans un bac à sable contenant le minimum nécessaire.

C’est souvent Docker (`https://www.docker.com/`) qui est utilisé pour créer des *conteneurs* (sortes de machines virtuelles) destinées à construire l’application (contenant toutes les dépendances nécessaires), à la tester, et à la livrer/déployer vers un serveur de diffusion.



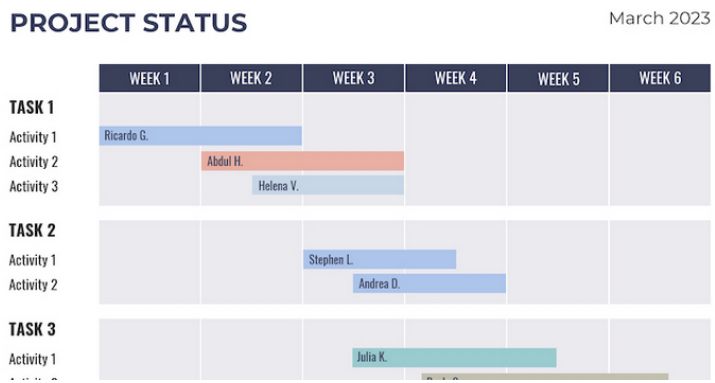
Container vs. VM

Plan du cours

Organiser une équipe

Pour organiser les développements, notamment dans une équipe agile, et suivre les tâches de chacun des membres de l'équipe, des outils de planification existent :

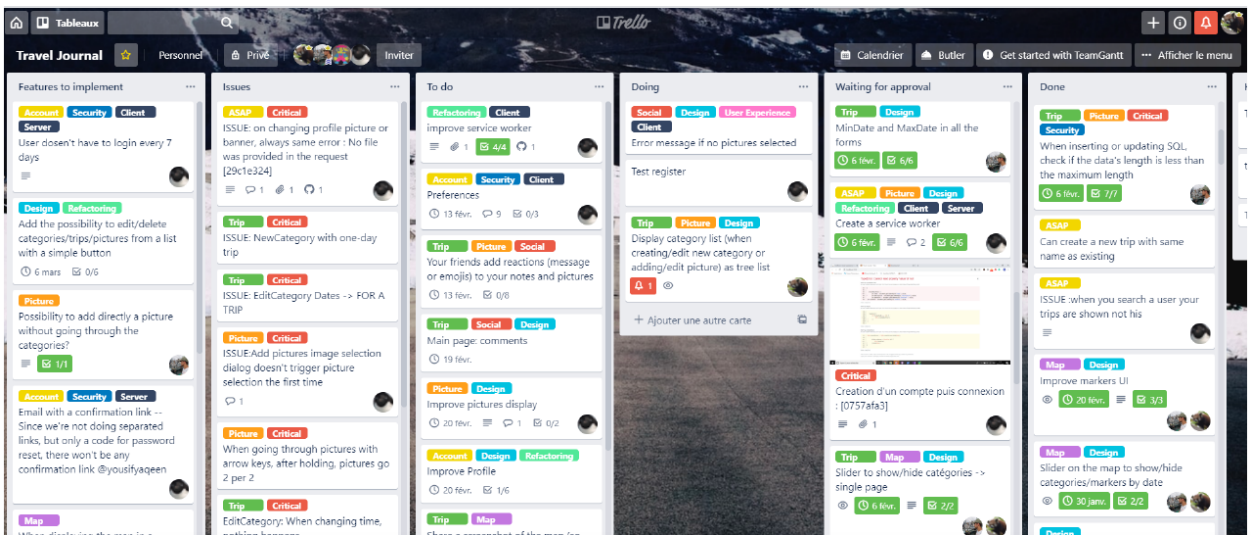
- qui permettent de planifier une réalisation "à l'ancienne" avec des tâches clairement définies et bien identifiées dans le temps (dépendances avec les autres tâches, durée, acteurs, etc.) : diagrammes de GANTT, diagrammes d'activités UML



Organiser une équipe

Pour organiser les développements, notamment dans une équipe agile, et suivre les tâches de chacun des membres de l'équipe, des outils de planification existent :

- qui permettent de planifier une réalisation en mode "développement agile" avec des cycles d'itérations, des tâches précises, affectées aux participants, etc. : KanbanFlow, Trello, etc. (teaser pour le cours de F. Peureux)



A noter que ces outils peuvent également, a posteriori, générer des diagrammes représentant la chronologie des tâches effectuées.

Plan du cours

Les règles de codage

Qu'est-ce que c'est ?

Les règles de codage (ou *coding standards*) définissent un ensemble de règles qui ont pour but d'uniformiser les pratiques de développement logiciel, de diffuser les bonnes pratiques de développement et d'éviter les erreurs classiques au sein d'un groupe de développeurs.

Thèmes couverts

Les règles de codage couvrent en général les thèmes suivants :

- Le nommage et l'organisation des fichiers du code source
- Le style d'indentation
- Les conventions de nommage, ou règles de nommage
- Les commentaires et documentation du code source
- Recommandations sur la déclaration des variables
- Recommandations sur l'écriture des instructions, des structures de contrôle et l'usage des parenthèses dans les expressions.

Exemples de règles de codage

Conventions de nommage, déclaration de variables, etc.

- Ecrire les constantes en majuscules
- Ecrire les identificateurs de classe en camel case en commençant par une lettre majuscule.
- Ecrire les identificateurs de méthode en camel case en commençant par une lettre majuscule.
- Notation hongroise (dont l'utilité est remise en cause) : préfixer un nom de variable par son type ou son usage (ex. `strBidule`, `idxClient`)

Ecriture des instructions

- mettre systématiquement des accolades autour des blocs `if` et `else`
- placer les constantes en premier dans les `if`.
  Par exemple : `if (1 == n)` au lieu de `if (n == 1)`
- en Java, interdiction d'utiliser une affectation dans un `if`
- en C, interdiction de lire une variable non initialisée
- ...

Les règles de codage

Outils existants

Différents outils permettent de s'assurer du respect des règles de codage (entre autres) :

- les *IDE*, modulo d'éventuels fichiers de configuration décrivant les règles
- les *linters*, outils d'analyse statique de code source (ex. JSLint, Pylint)
- des outils spécifiques dédiés à un thème particulier : indent (indentation code C), checkstyle (règle de nommages, commentaires en Java), etc.

Liens

Quelques liens sur les standards :

- Google Java Style Guide : `https://google.github.io/styleguide/javaguide.html`
- GNU Coding Standard (C) : `http://www.gnu.org/prep/standards/standards.html`
- ...

Chaque entreprise ou chaque domaine d'activité peut définir son standard qui lui est propre et qui s'appuiera potentiellement sur des standards existants.