GIT

Concept et cas pratique

Durée : 14 heures

GIT

Introduction

- **Introduction**
- **Avant de commencer**
- **Commencer avec GIT**
- **Gestion des branches**
- **GIT en travaille collaboratif**
- **Gestion des commits**
- **Gestion des tags**
- **L’historique**
- **Gestion des erreurs et déboggage**
- **Déployer avec GIT**

GIT

Introduction

- **Les gestionnaires de versions**
- **Décentralisation du code**
- **Les avantages de GIT**

Les gestionnaires de version

Les gestionnaires de versions

Qu’est qu’un gestionnaire de version ?

- **V.C.S**. pour Version Control System
- Stock un ensemble de fichiers
- Conserve la chronologie des modifications
- Permet de retrouver les différentes versions d’un fichier
- Fonctionne avec tout types de fichiers
- Permet de retrouver des fichiers supprimés

Les gestionnaires de versions

- **D.V.C.S**. pour Distributed Version Control System
- La gestion des fichiers est décentralisée

Les gestionnaires de versions

NE PAS CONFONDRE

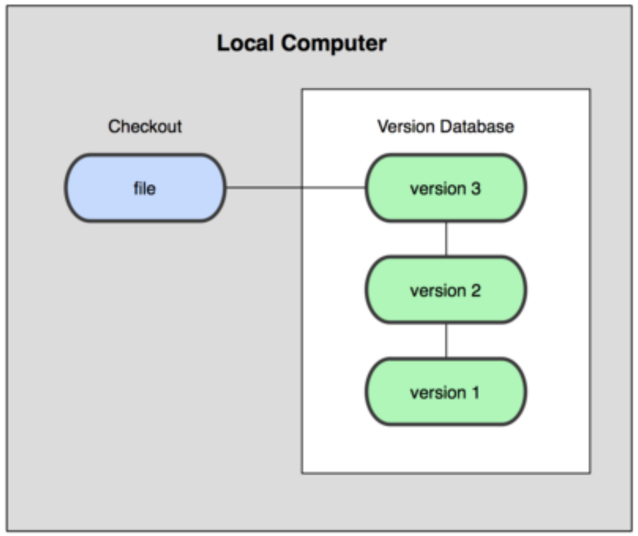
Les VCS ne sont pas des systèmes de Backup

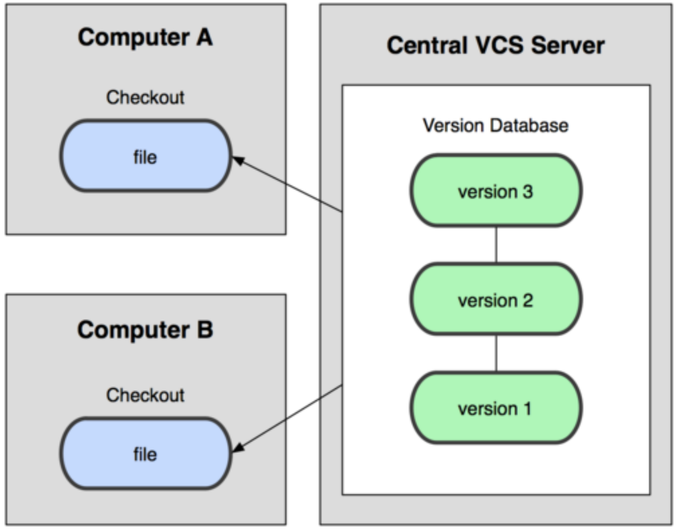
GIT n’est pas GitHub

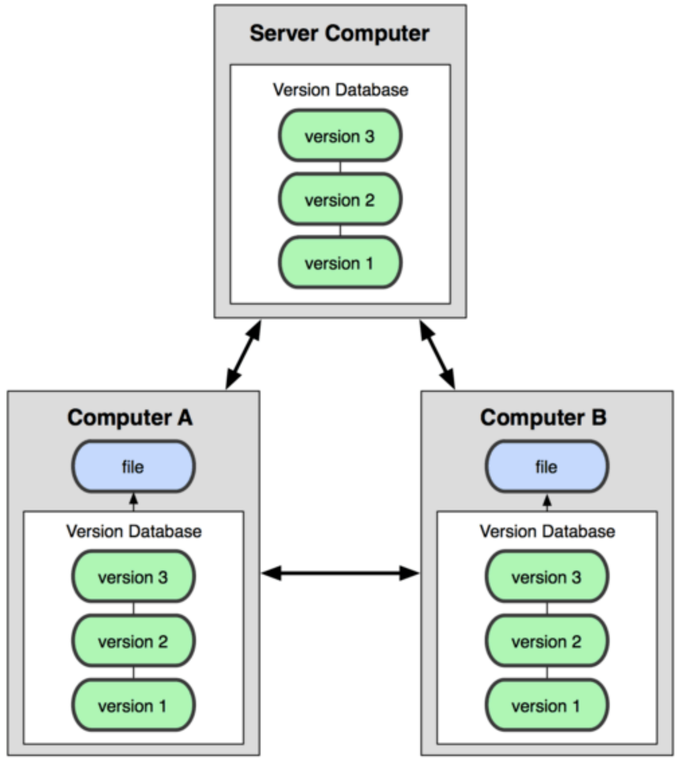
Décentralisation du code

Décentralisation du code

Locale Vs. Centralisée Vs. Distribuée

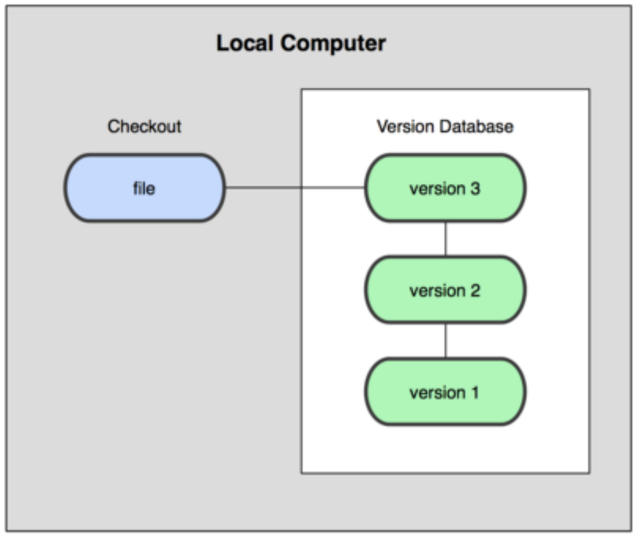






Décentralisation du code

Pourquoi décentralisé son code ?

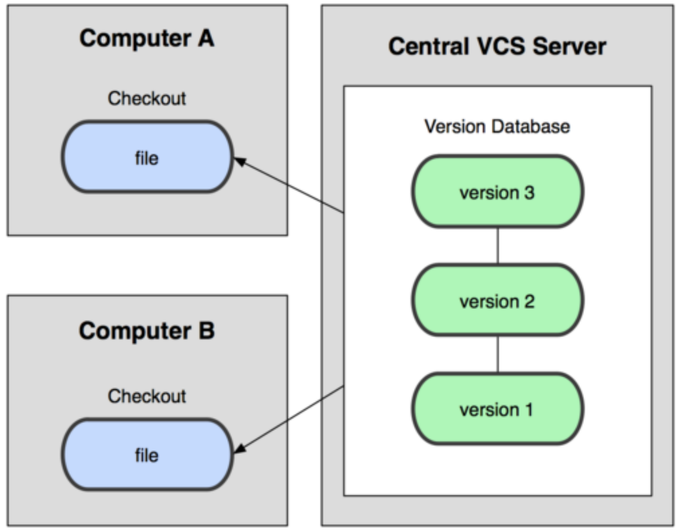


Avec la gestion locale :

- Pas de collaboration possibles
- Perte total en cas de crash disc

Décentralisation du code

Pourquoi décentralisé son code ?

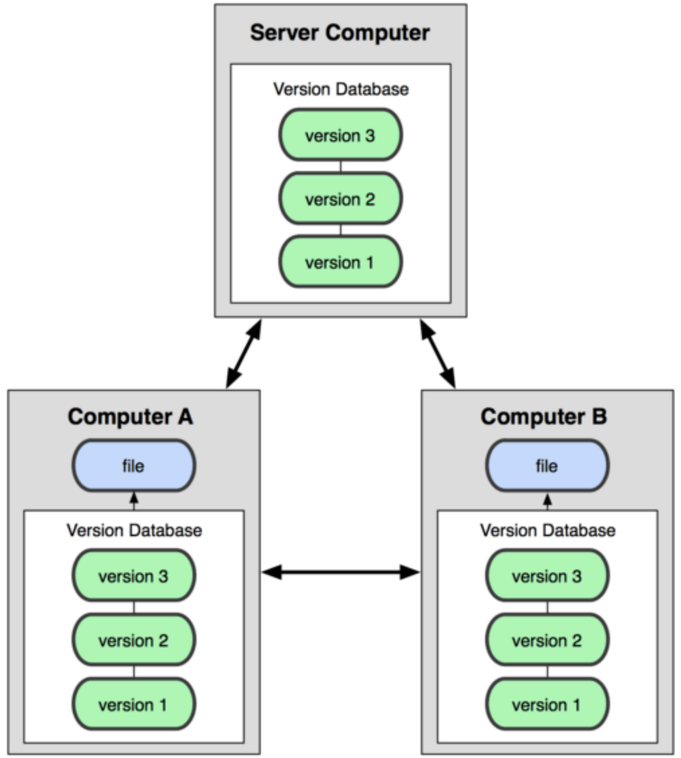


Avec la gestion Centralisée :

- Le serveur conserve l'historique
- Les dév. possèdent la dernière version de fichers

Décentralisation du code

Pourquoi décentralisé son code ?



Avec la gestion Distribuée :

- Chaque partie possède toutes les versions de fichiers
- s’échange directement les modifications

Décentralisation du code

A savoir…

- La plupart des opérations de Git sont local
- Les recherches dans l’historique du projet sont local
- L’intégralité de la base de données Git est en local

Les avantages de GIT

Les avantages de GIT

- **GIT** est un système libre
- Suivie de l’évolution du code source (rollback et identification)
- Travail collaboratif
- Travail par branche
- Publier en production à partir d’une version donnée

Les inconvénients de GIT

- Mais GIT est complexe
- Pas très bien porté sous Windows

GIT

Avant de commencer

- **Installer GIT**
- **Structure d’un dépôt GIT**
- **Configurer GIT**
- **Les objets GIT**
- **Les états de GIT**
- **La gestion des fichiers**

Installer GIT

Installer GIT

Installer GIT sur Linux

- Sous **Debian** et **Ubuntu**

```
apt install git-all
```

- Sous **Fedora**, **CentOS** ou une distribution basée sur **RPM**

```
dnf install git-all
```

- Autres distributions : https://git-scm.com/download/linux

Installer GIT

Installer GIT sur Mac

- En mode graphique :
  - Télécharger le .dmg https://git-scm.com/download/mac

- En ligne de commande (avec **port**)

```
sudo port install git-core
```

- En ligne de commande (avec **homebrew**)

```
brew update
brew install git
```

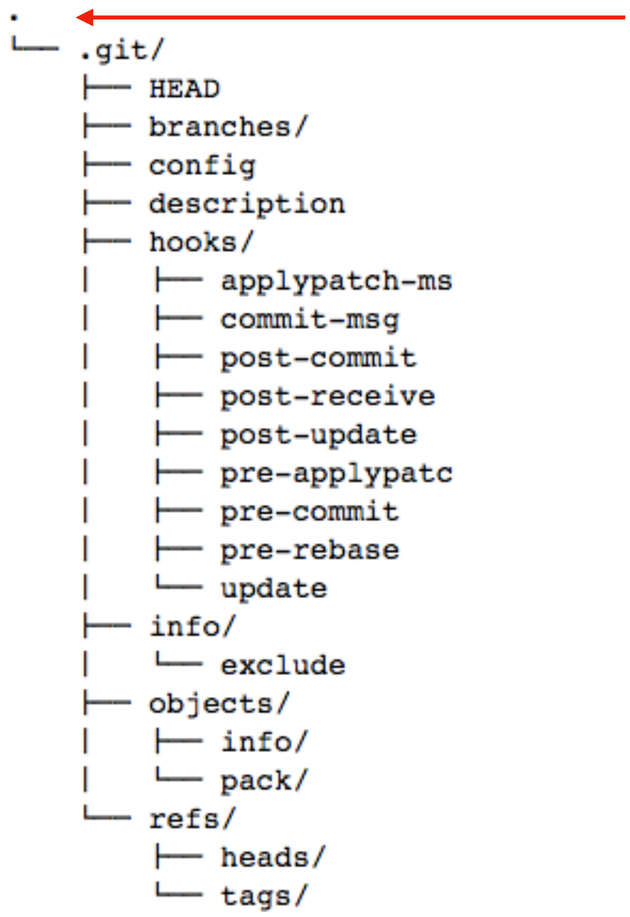
Installer GIT

Installer GIT sur Windows

- En mode graphique :
  - Télécharger l’exécutable https://git-scm.com/download/win

Structure d'un dépôt GIT

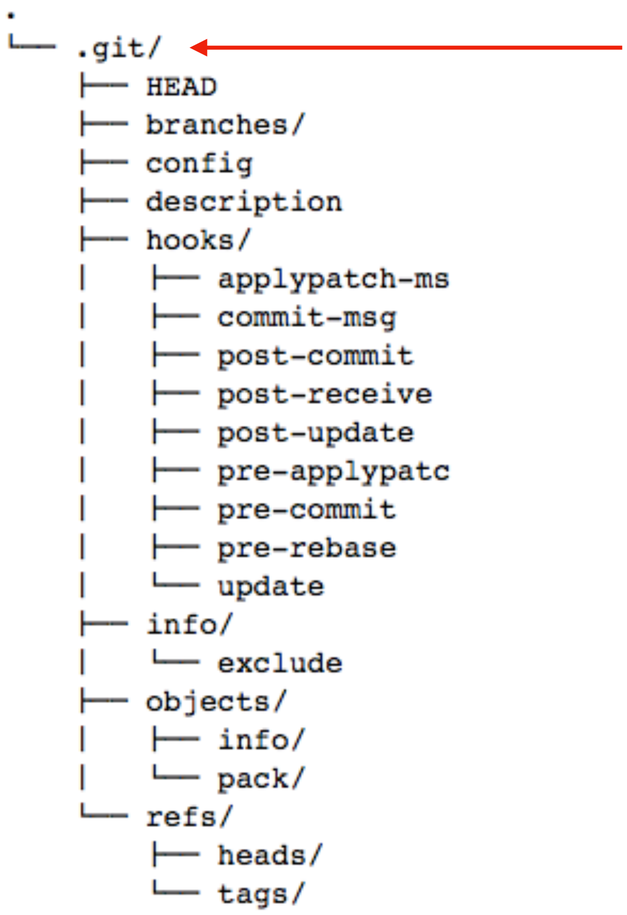
Structure d’un dépôt GIT



Racine du projet

C’est dans ce répertoire que se trouve tous les fichiers du projet.

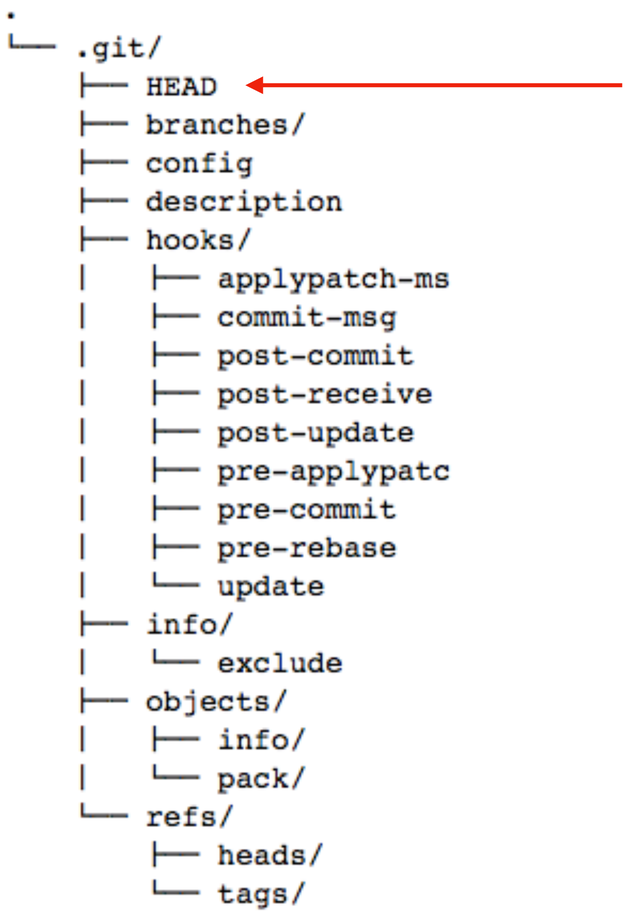
Structure d’un dépôt GIT



.git/

Ce répertoire (masqué dans le système de fichiers)
contient toutes les informations sur l’état de GIT,
la configuration de GIT et les différentes versions des fichiers du projet.

Structure d’un dépôt GIT

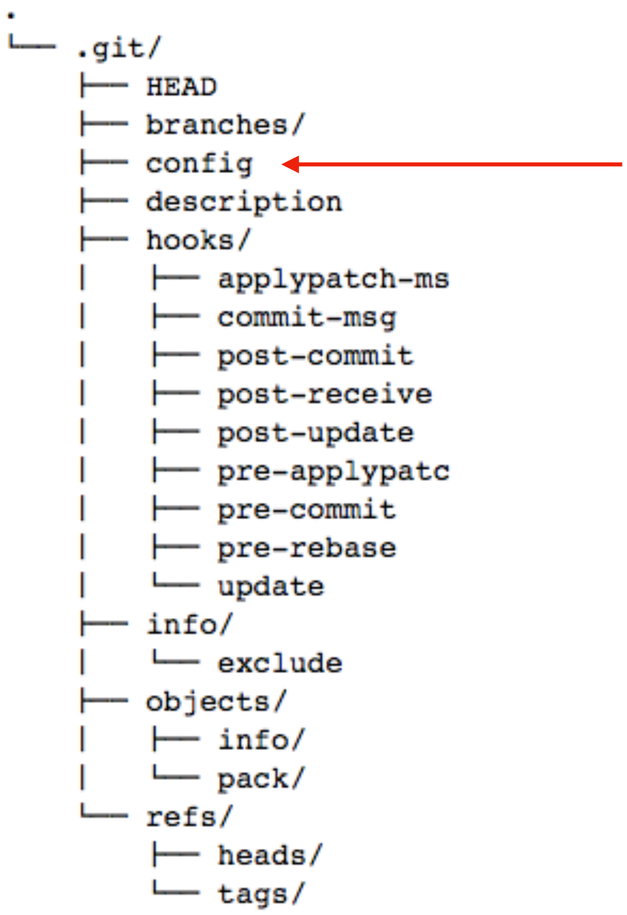


HEAD

Ce fichier indique pour ce dépôt Git, le code actuellement vérifié.

Généralement le point de la branche sur lequel on travaille.

Structure d’un dépôt GIT

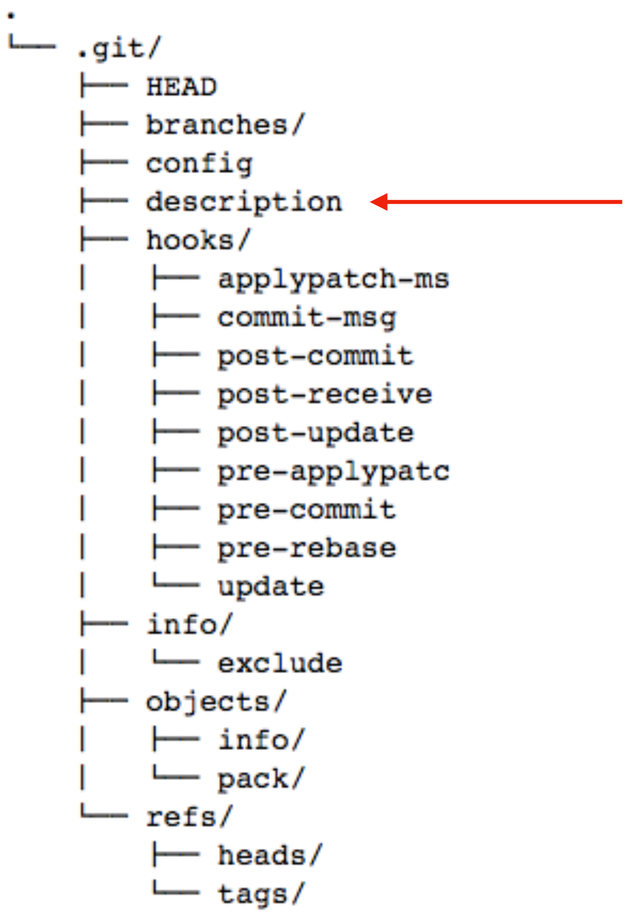


Config

Le fichier de configuration pour ce dépôt Git.

Il peut contenir les paramètres permettant de gérer et stocker les données dans le dépôt local, les dépôts distants connus, et les informations sur les utilisateurs.

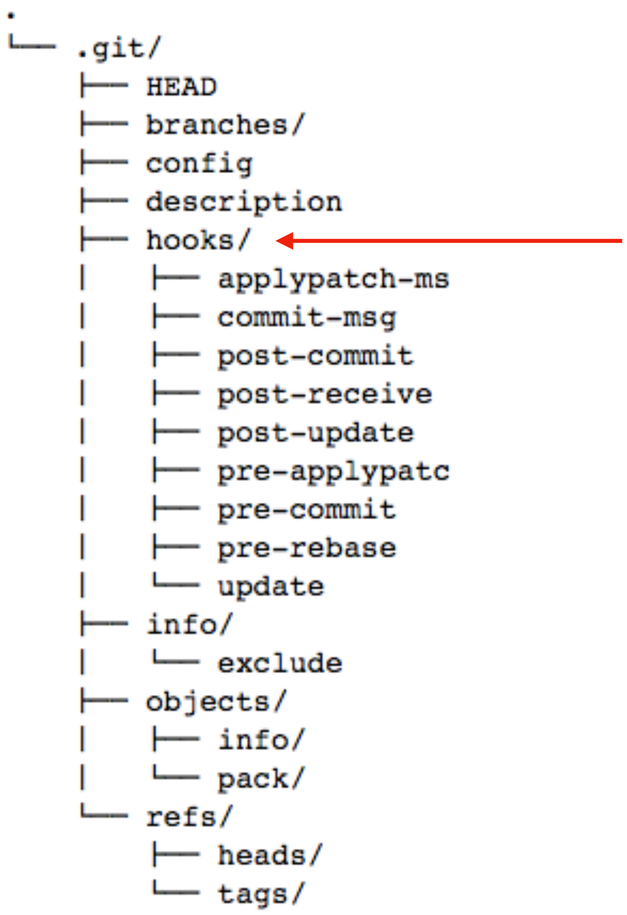
Structure d’un dépôt GIT



Description

Utilisé par les outils du navigateur de dépôt, contient une description du projet, généralement inchangée dans les dépôts non partagés.

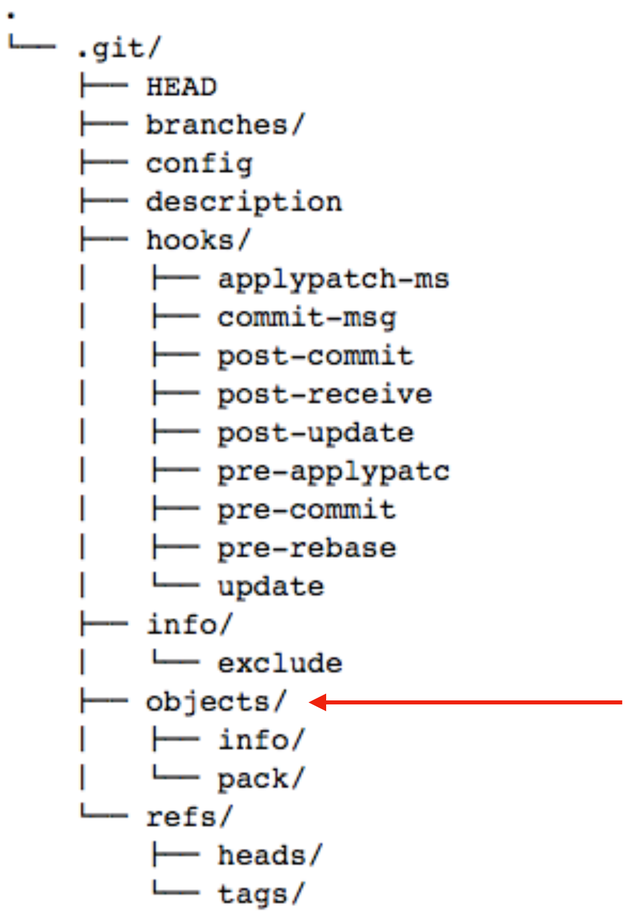
Structure d’un dépôt GIT



Hooks

Contient les scripts à lancer quand des évènements particuliers surviennent dans le dépôt Git.
Ces points d'entrée sont utilisé par exemple pour lancer des tests avant chaque soumission, filtrer le contenu uploadé, et implémenté ce genre de personnalisations.

Structure d’un dépôt GIT

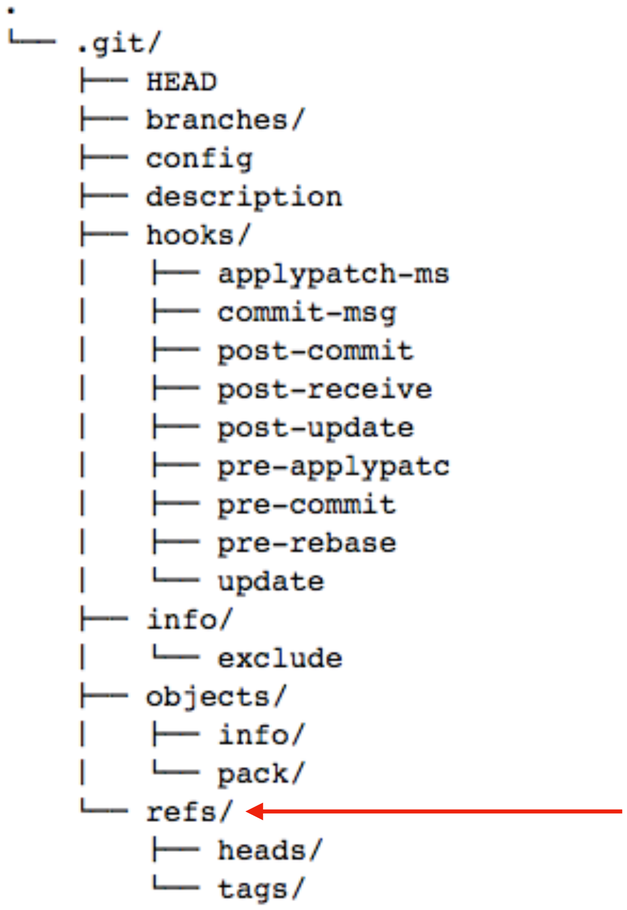


Objects

Stocke les listes de répertoires, fichiers et soumission.

- Les objets non compressés.
- Les objets compressés (fichiers volumineux).

Structure d’un dépôt GIT



Refs

Contient les informations où les branches pointent.

Inclut normalement des répertoires "heads" pour les branches locales, et "remotes" pour les copies des branches distantes.

Toutes les branches ne figurent pas dans ces répertoires.

Celles qui n'ont pas changé récemment sont listées dans le fichier `.git/packed-refs`.

Configurer GIT

Configurer GIT

3 niveaux de configuration

- **Global au système** : **`/etc/gitconfig`**, modifiable avec la commande

```
git config --system
```

- **Global à l'utilisateur** : **`~/gitconfig`**, modifiable avec la commande

```
git config --global
```

- **Global au projet** : **`.git/config`**, modifiable avec la commande

```
git config
```

- Chaque niveau surcharge le précédent

Configurer GIT

Configuration minimal

- Les contributeurs sont identifiés par leur nom et adresse email

```
git config --global user.name "John DOE"
git config --global user.email "john.doe@osw3.net"
```

Configurer GIT

Eviter de retaper son mot de passe

- Git propose de stocker le mot de passe temporairement afin d'éviter de le saisir à chaque push, pull ou toute autre opération impliquant un dépôt distant.

```
git config --global credential.helper cache
```

Configurer GIT

Eviter de retaper son mot de passe

- Sur Linux, si la commande précédente ne fonctionne pas après avoir saisi au moins une fois le mot de passe, éditer le fichier .netrc

```
vim ~/.netrc
```

```
machine github.com
    login <user>
    password <password>
```

Configurer GIT

Eviter de retaper son mot de passe

- Si cela ne fonctionne toujours pas…
- Vérifier que vous êtes dans le répertoire racine du projet
- et que votre fichier `.git/config` soit sous la forme :

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = https://github.com/project.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

Configurer GIT

Vérifier les paramètres

- avec la commande :

```
git config --list
```

Les objets GIT

Les objets GIT

GIT est un système de fichiers adressable par contenu

Une mémoire adressable par le contenu (Content-Addressable Memory - CAM)
est un type de mémoire utilisé par certaines applications pour la recherche à très haute vitesse.
Elle est aussi connu sous le nom de Mémoire Associative.
Contrairement aux mémoire standards (Random Access Memory - RAM), la CAM est parcourue
par l'application utilisatrice comme une tableau associatif.

GIT est une simple base de paires **Clé / Valeur**

Chaque paire est un objet

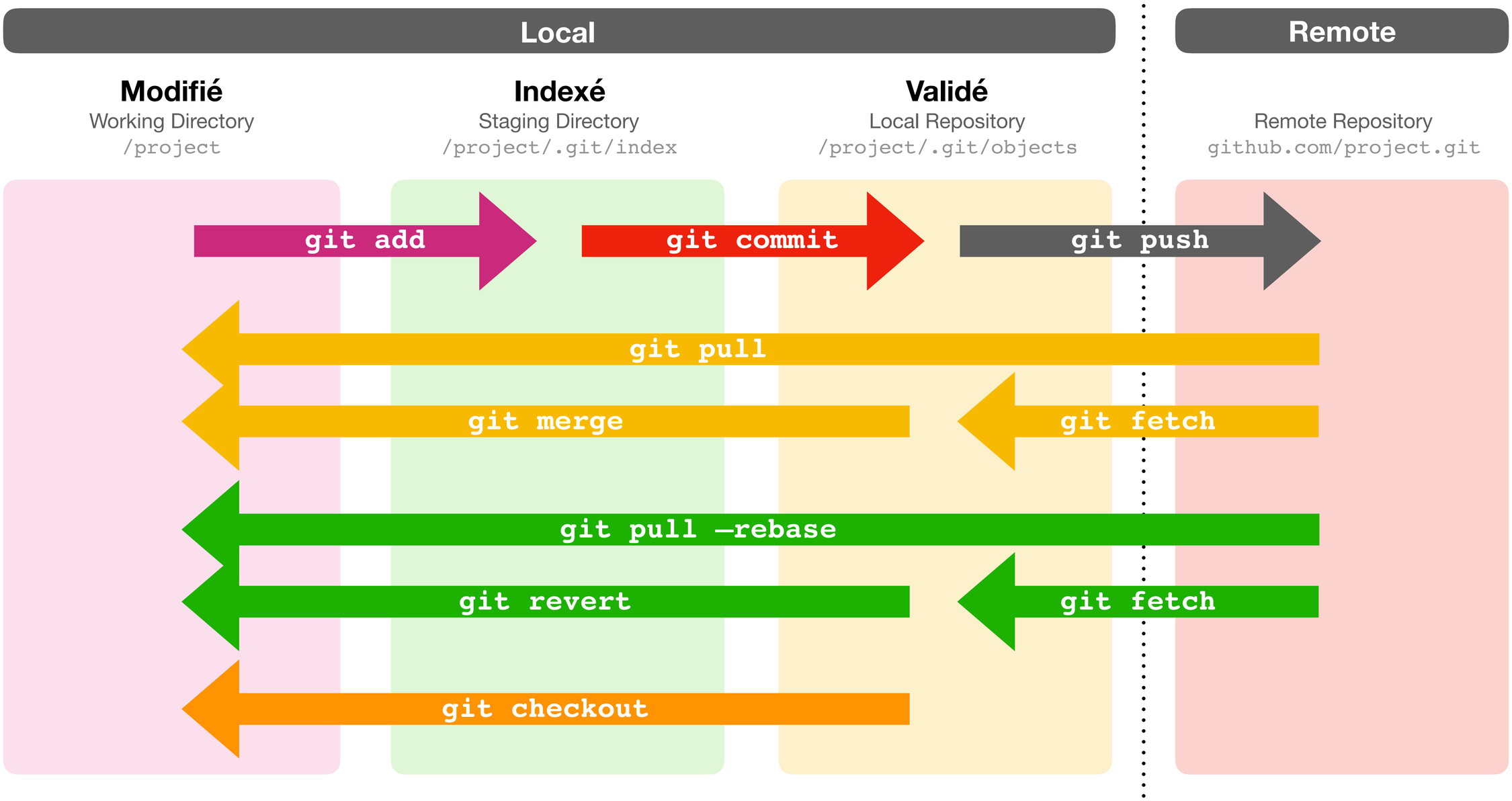
Les objets GIT

Comment GIT créer t’il les objets ?

- Tout est vérifié par une somme de contrôle avant d’être stocké.
- Cette somme de contrôle, une signature unique, sert de référence.
- Impossible de perdre un fichier sans que Git ne s’en aperçoive.
- Les sommes de contrôle sont en SHA-1.

Les états de GIT

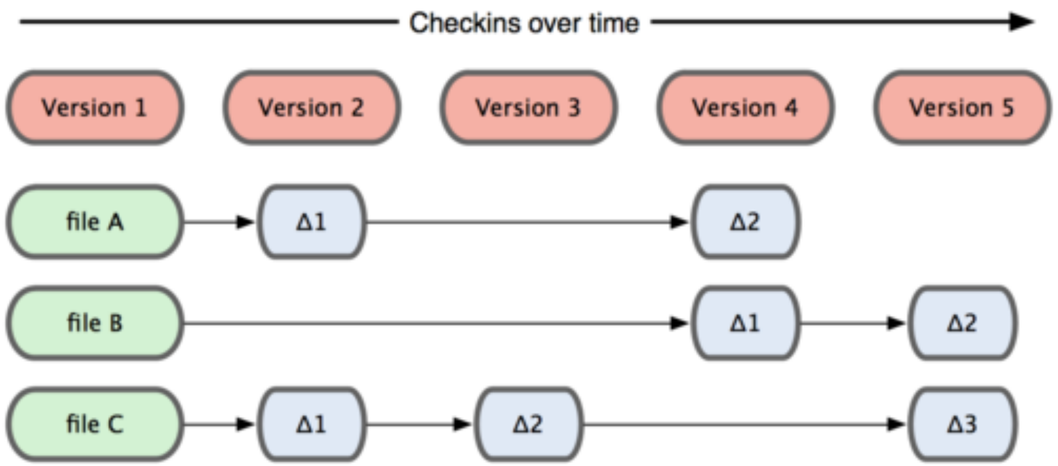
Les états de GIT



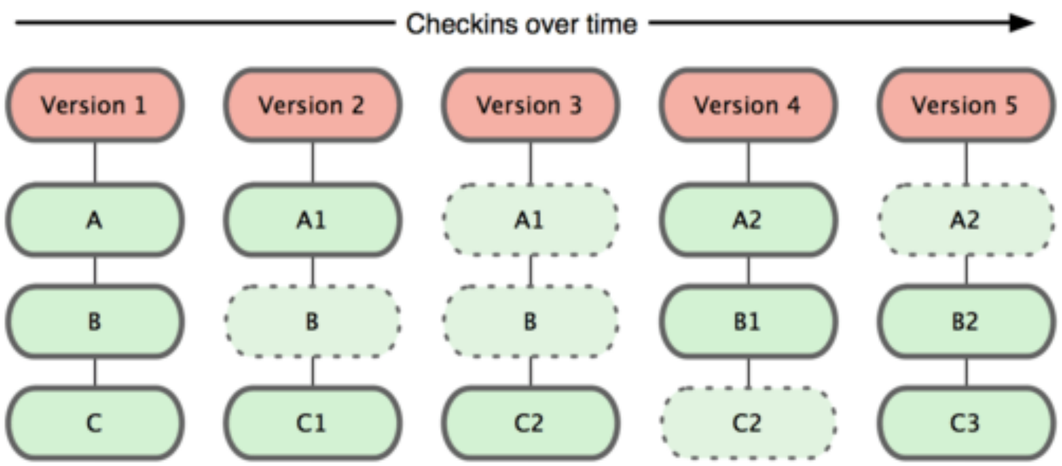
La gestion des fichiers

La gestion des fichiers

avec un V.C.S. classique



avec GIT



GIT prend un instantané (snapshop)
pour chaque fichier, à chaque version

GIT

Commencer avec GIT

- **Créer un dépôt**
- **Vérifier l’état**
- **Indexer les modifications**
- **Placer un fichier sous suivi de version**
- **Exclure des fichiers**
- **Modifier des fichiers**
- **Inspecter les modifications**
- **Valider les modifications**
- **Supprimer et déplacer des fichiers**

Créer un dépôt

Créer un dépôt

Initialiser un dépôt avec du code existant

- Créer le répertoire racine du projet et faire pointer le terminal dans celui-ci

```
mkdir project
cd project
```

- Utiliser la commande

```
git init
```

- le répertoire `.git` est créer à la racine du projet

Créer un dépôt

Initialiser d'un dépôt vierge

- Utiliser la commande

```
git init project
```

- GIT créer le répertoire `project`
- puis le répertoire `.git` est créer à la racine du projet

Vérifier l'état

Vérifier l’état

- Pour connaitre l’état des fichiers :

```
git status
```

- Pour chaque fichier ajouter ou modifier, **`git status`** donnera un nouvel état.

- Pour un affichage plus concis :

```
git status -s
```

Indexer les modifications

Indexer les modifications

- Créer le fichier :

```
vim Hello.txt
> Hello, i am the first line !
```

- Voir le statut

```
git status
```

- Le fichier à le statut `untracked file`, il n'est pas indexé.

- Pour indexer le fichier

```
git add Hello.txt
```

Placer un fichier sous suivi de version

Placer un fichier sous suivi de version

- Pour commencer à suivre un fichier :

```
git add <filename>
```

- En faisant **`git status`** nous voyons le changement d'état du fichier indexé.

- La version du fichier au moment du **`git add`**, sera celle de l'historique.

Exclure des fichiers

Exclure des fichiers

Pourquoi ?

- Des fichiers **non souhaités** dans l'espace de travail.
  - Fichiers .DS_Store
  - Fichiers de restauration créés par les éditeurs (.swp pour VIM)

- Des fichiers **sensibles**
  - config.php

- Les librairies et dépendances externe ou les fichiers de cache
  - /vendor/
  - /node_modules/
  - /app/cache/

Exclure des fichiers

Comment ?

- Avec le fichier **.gitignore**

```
# ignorer les backups emacs
*~

# Ignorer le répertoire de cache
app/cache
```

- Chaque ligne est une spécification des fichiers à ignorer.
- Les lignes vides ou commençant par # sont ignorées.
- Ajouter aux commits et commun à tous les développeurs.

Exclure des fichiers

Comment ?

- Avec le fichier **exclude** .git/info/exclude
- Même syntaxe que **.gitignore**
- Ignore les fichier de son propre poste

Modifier des fichiers

Modifier des fichiers

- Modifier le fichier :

```
vim Hello.txt
> Hello, i am the first line !
> This is my second line !
```

- Voir le statut

```
git status
```

- Le fichier à le statut `modified`, il n'est pas indexé.

- Pour indexer le fichier

```
git add Hello.txt
```

Inspecter les modifications

Inspecter les modifications

- Inspecter les modifications permet d'avoir une vision plus précise des changements apportés aux fichiers.

- On utilise la commande.

```
git diff
```

- Cette commande répond aux questions :
  - Quels fichiers sont modifiés mais pas encore indexé ?
  - Quelles modifications indexées sont prête à être validées ?

- Pour voir les modifications qui feront partie de la prochaine validation :

```
git diff --cached
```

Valider les modifications

Valider les modifications

- Valider les modifications :

```
git commit
```

- Uniquement les modifications indexées avec `git add`.

- Sans l'option **`-m`**, GIT ouvre automatiquement l'éditeur texte pour ajouter un commentaire de soumission.
- Avec l'option **`-m`**, précise le message de commit directement.

```
git commit -m 'message'
```

- L'option **`-v`**, ajoute le `diff` dans le commentaire de retour
- GIT retourne l'ID du commit

Supprimer et déplacer des fichiers

Supprimer et déplacer des fichiers

Supprimer un fichier

- Pour supprimer un fichier de GIT, il faut l'effacer de l'index.
- Un fichier effacer du système de fichier apparaitra dans la liste des modifications non validées

```
rm <filename>
```

- Pour supprimer un fichier de GIT

```
git rm <filename>
```

- l'effacement du fichier est automatiquement indexé

Supprimer et déplacer des fichiers

Déplacer / Renommer un fichier

- GIT est capable de voir les fichiers déplacés.
- Mais ne suit pas le mouvement des fichiers.
- Déplacer les fichiers avec la commande

```
git mv <source> <destination>
```

- Cette commande revient à :

```
mv <source> <destination>
git rm <source>
git add <destination>
```

GIT

Gestion des branches

- **Introduction**
- **Bonnes pratiques**
- **Lister les branches**
- **Créer une branche**
- **Basculer entre les branches**
- **Fusionner des branches**

Introduction

Introduction

- Presque tous les VCS proposent une forme de gestion de branches.
- Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.
- La branche par défaut est **master**.
- Vous pouvez leur donner le noms que vous souhaitez.

Bonnes pratiques

Bonnes pratiques

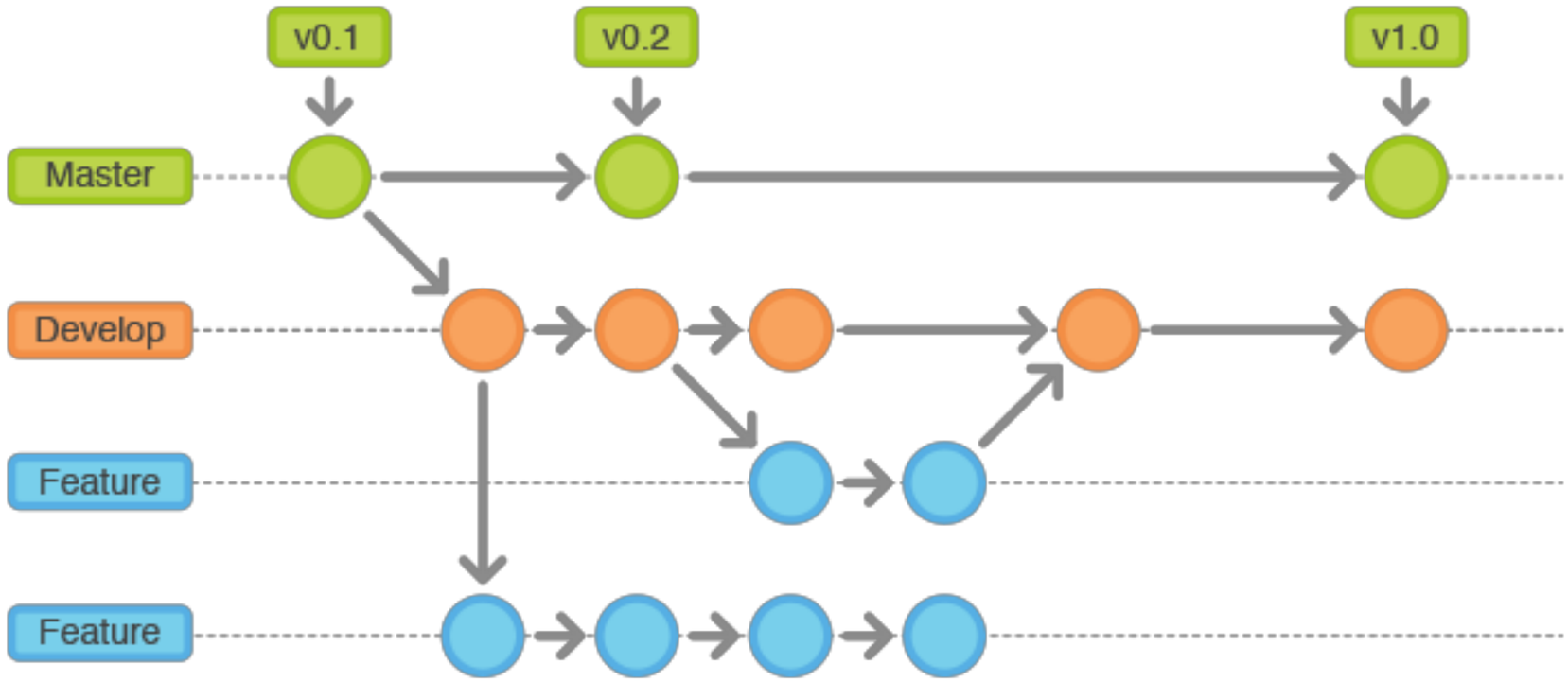
- La branche « **master** » ne servira qu'à la mise en production du projet et ne devra **jamais** être **« commitée »**
- Il faut créer une branche spécifique pour le développement du projet. Elle se nomme généralement « **develop** » ou porte le numéro de version. Cette branche ne devrai **jamais** être « **commitée** »
- Il faut créer une branche par « **feature** », « **release** » ou « **hotfix** »
- Aucune limite pour les commits des branches de « **features** », « **releases** » et « **hotfixes** »
- Lorsque le code est validé, il faut **fusionner** les « **features** », « **releases** » et « **hotfixes** » à la branche « **develop** ».
- Lorsque qu'une version de l'application est prête, **fusionner** « **develop** » avec « **master** »

Bonnes pratiques

Pourquoi une branche par feature ?

- Evite les conflits en travail collaboratif.
- Evite les commits affectant plusieurs modules.
- En un coup d'oeil on peut voir sur quoi les différents intervenant travail

Bonnes pratiques



Lister les branches

Lister les branches

- Pour afficher la liste des branches :

```
git branch
```

  - La branche courante est marquée par un astérisque.

- Pour afficher la liste des branches avec le numéro de la dernière validation :

```
git branch -v
```

  - Affiche sur chaque branche le numéro et commentaire de leur dernier commit.

- Pour afficher la liste des branches déjà fusionnées :

```
git branch --merged
```

- Pour afficher la liste des branches pas encore fusionnées :

```
git branch --no-merge
```

Créer une branche

Créer une branche

- Pour créer une branche :

```
git branch <branch>
```

- Pour créer une branche et basculer immédiatement sur celle-ci :

```
git checkout -b <branch>
```

  - Cela revient à

```
git branch <branch>
git checkout <branch>
```

Basculer entre les branches

Basculer entre les branches

- Pour changer de branche :

```
git checkout <branch>
```

- Cela à pour effet de déplacer HEAD sur la nouvelle branche.
- Et remplacer les fichiers du répertoire par ceux de l'état du snapshot (intantanés) de <branch>.
- Toutes les validations de modifications (commits) appartiennent à la branche sur laquelle ils sont créés.

Fusionner des branches

Fusionner des branches

- Imaginons que nous ayons apporté des modifications à la branche **`feature_a`** et que nous voulons les fusionner à la branche **`develop`**
- nous devons basculer de la branche **`feature_a`** à **`develop`**

```
git checkout develop
```

- puis nous devons dire à GIT d'importer les modification de la branche **`feature_a`** dans la branche develop qui est maintenant notre branche courante.

```
git merge feature_a
```

Supprimer une branche

Supprimer une branche

- Pour supprimer une branche :

```
git branch -d <branch>
```

- Supprimez une branche dont tous les travaux n'ont pas été fusionnés, GIT affiche une erreur.
- Pour forcer la suppression de cette branche, utiliser l'option **`-D`**

```
git branch -D <branch>
```

- GIT ne sais pas supprimer la branche courante.

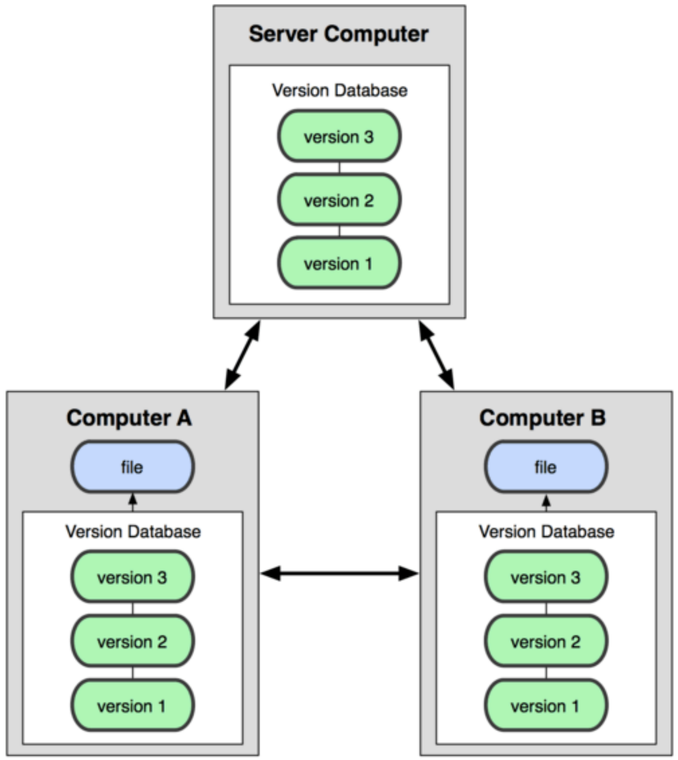
GIT

GIT en collaboratif

- Introduction
- Cloner un dépôt distant
- Lister les dépôts distant
- Ajouter des dépôts à la liste
- Contribuer aux dépôts distant
- Inspecter un dépôt distant
- Supprimer et renommer les dépôts distants
- Garder à jour son dépôt local
- Les pull-request

Introduction

Introduction



- Les dépôts distant sont des versions du projet hébergées sur le réseau d'entreprise ou internet.
- Pour collaborer sur un projet GIT, il est nécessaire de savoir gérer les dépôts distant.
- Il peut y avoir plusieurs dépôts distant.
- On peut donner les droits en lecture seule ou lecture/écriture

Cloner un dépôt distant

Cloner un dépôt distant

- Le clone est une copie identique du dépôt distant
  - fichiers
  - historique

- GIT gère les protocole http://, https:// et git:// (basé sur ssh)

- Commande :

```
git clone https://server.tld/project.git
```

Lister les dépôts distant

Lister les dépôts distant

- Permet de connaitre les alias de dépôts
- Afficher la liste des dépôts distant

```
git remote
```

- l'option **`-v`** permet d'afficher l'url associé aux alias :

```
git remote -v
```

- L'alias par défaut d'un dépôt cloné est **`origin`**

Ajouter un dépôt à la liste

Ajouter un dépôt à la liste

- Pour ajouter un dépôt distant à la liste :

```
git remote add <alias> <url>
```

Contribuer au dépôt

Contribuer au dépôt

- Pousser les modifications validées sur le dépôt distant :

```
git push <repository> <branch>
```

```
git push
```

Inspecter un dépôt distant

Inspecter un dépôt distant

- Pour inspecter et obtenir les information d’un dépôt distant

```
git remote show <alias>
```

Supprimer et renommer les dépôts distants

Supprimer et renommer les dépôts…

- Pour supprimer un dépôt de la liste :

```
git remote rm <alias>
```

- Pour renommer un alias :

```
git remote rename <old_alias> <new_alias>
```

Garder son dépôt local à jour

Garder son dépôt local à jour

- Deux commandes pour garder un dépôt local à jour :

```
git fetch
```

  - Récupère les données des commits de la branche courante qui n'existe pas encore dans le dépôt local.
  - Pour garder la branche à jour, il faut fusionner avec **`git merge`**.

```
git pull
```

  - Regroupe le **`git fetch`** suivis du **`git merge`**.

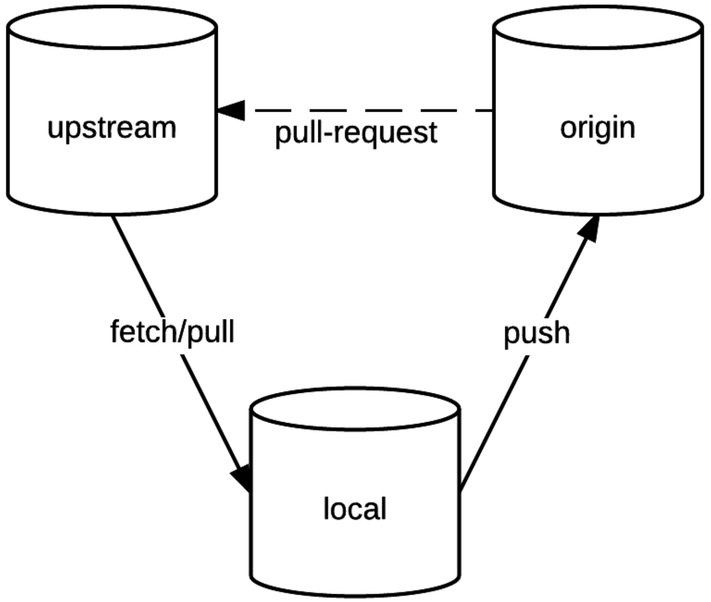
Les pull-request

Les pull-request

- Pour contribuer à un dépôt distant que nous n’administrons pas
- Populariser par GitHub
- Concept simple :
    1. Forker le projet
    2. Créer une branche et travailler dessus
    3. Publier la branche de son fork
    4. Créer la pull-request

Les pull-request

Situation de workflow triangulaire

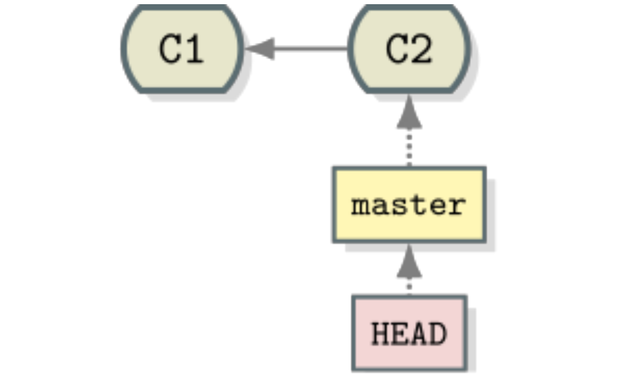


- **UpStream** est le dépôt de référence.
- **Origin** est le dépôt de fork.
- **Local** est le dépôt de travail.

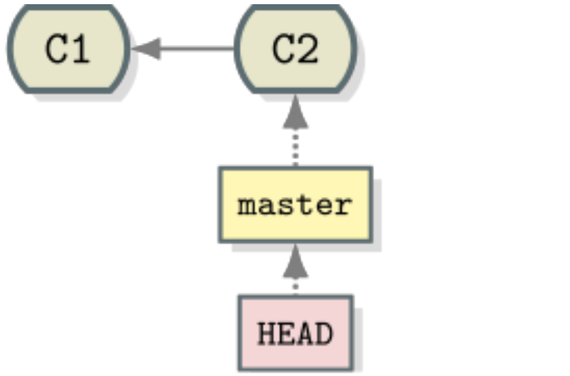
Les pull-request - Coté contributeur

Créer un fork

- Aller sur l'interface web du projet *(http://git.example.com/org/example)* et cliquer sur le bouton **Fork**.
- Cela va cloner le dépôt sur le serveur (GitHub, GitLab ou BitBucket)



http://git.example.com/org/example



http://git.example.com/contributor/example

Les pull-request - Coté contributeur

Cloner le dépôt

```
git clone http://git.example.com/contributor/example
```

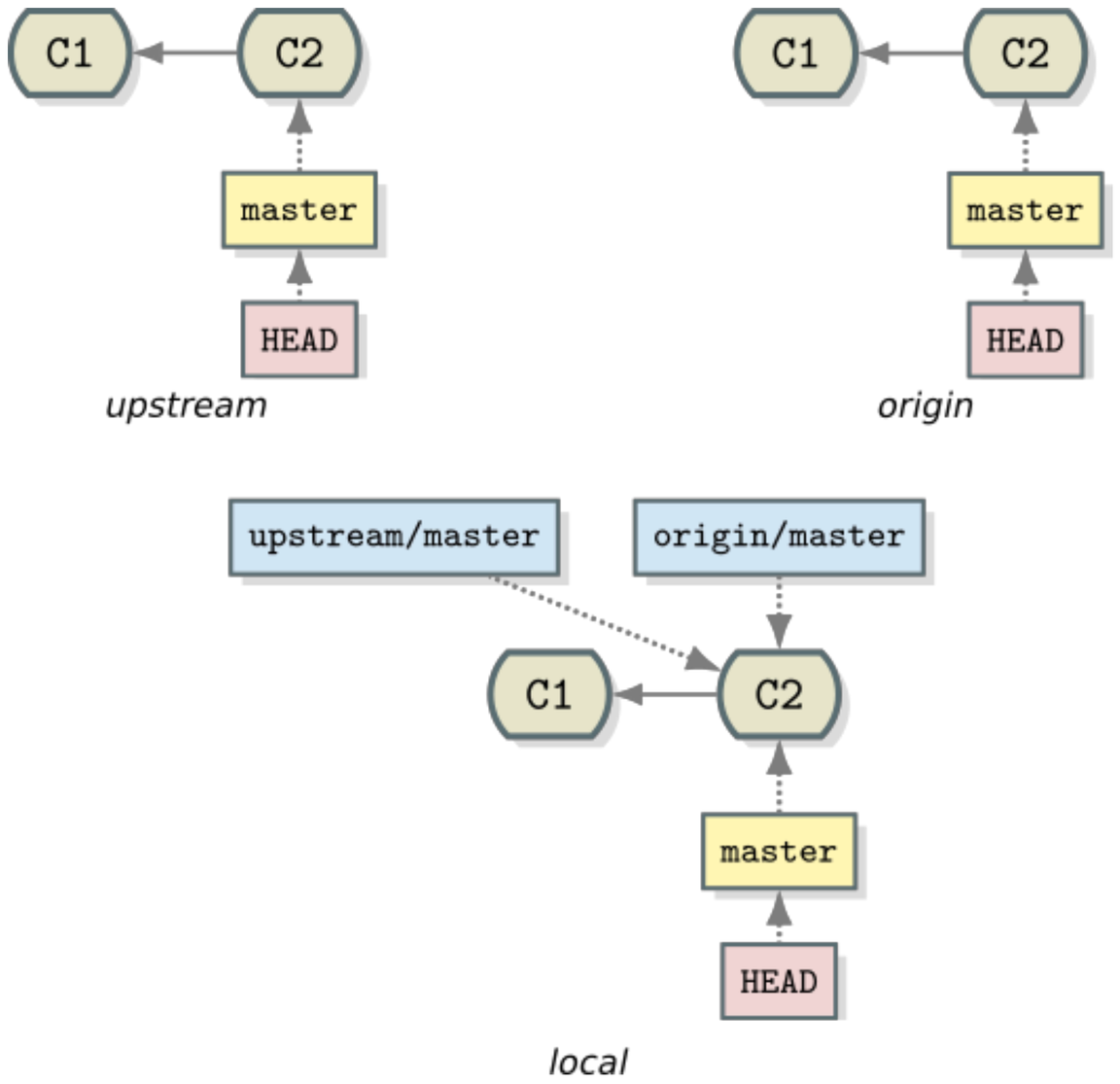
- Configure GIT pour publier sur notre fork **Origin**

```
git config remote.pushdefault origin
git config push.default current
```

- Ajouter le dépôt du projet d'origine à notre liste de dépôt avec l'alias **upstream**

```
git remote add upstream http://git.example.com/org/example
git fetch upstream
```

Les pull-request - Coté contributeur



Les pull-request - Coté contributeur

Créer une branche spécifique

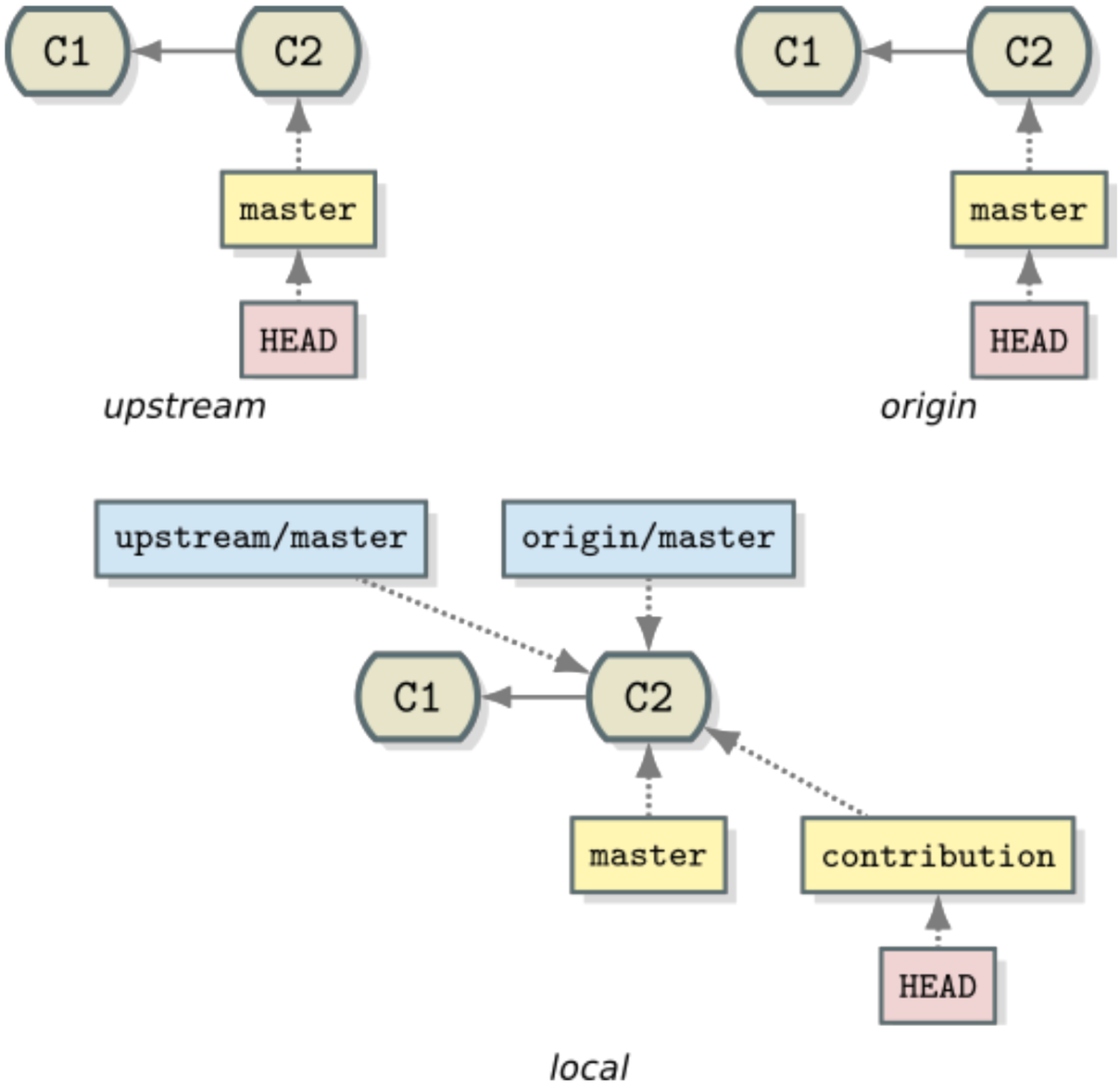
- Une bonne pratique est de créer une branche spécifique à notre contribution
- Créons une branche **`contribution`** qui va automatiquement traquer la branche **`upstream/master`**

```
git checkout -b contribution upstream/master
```

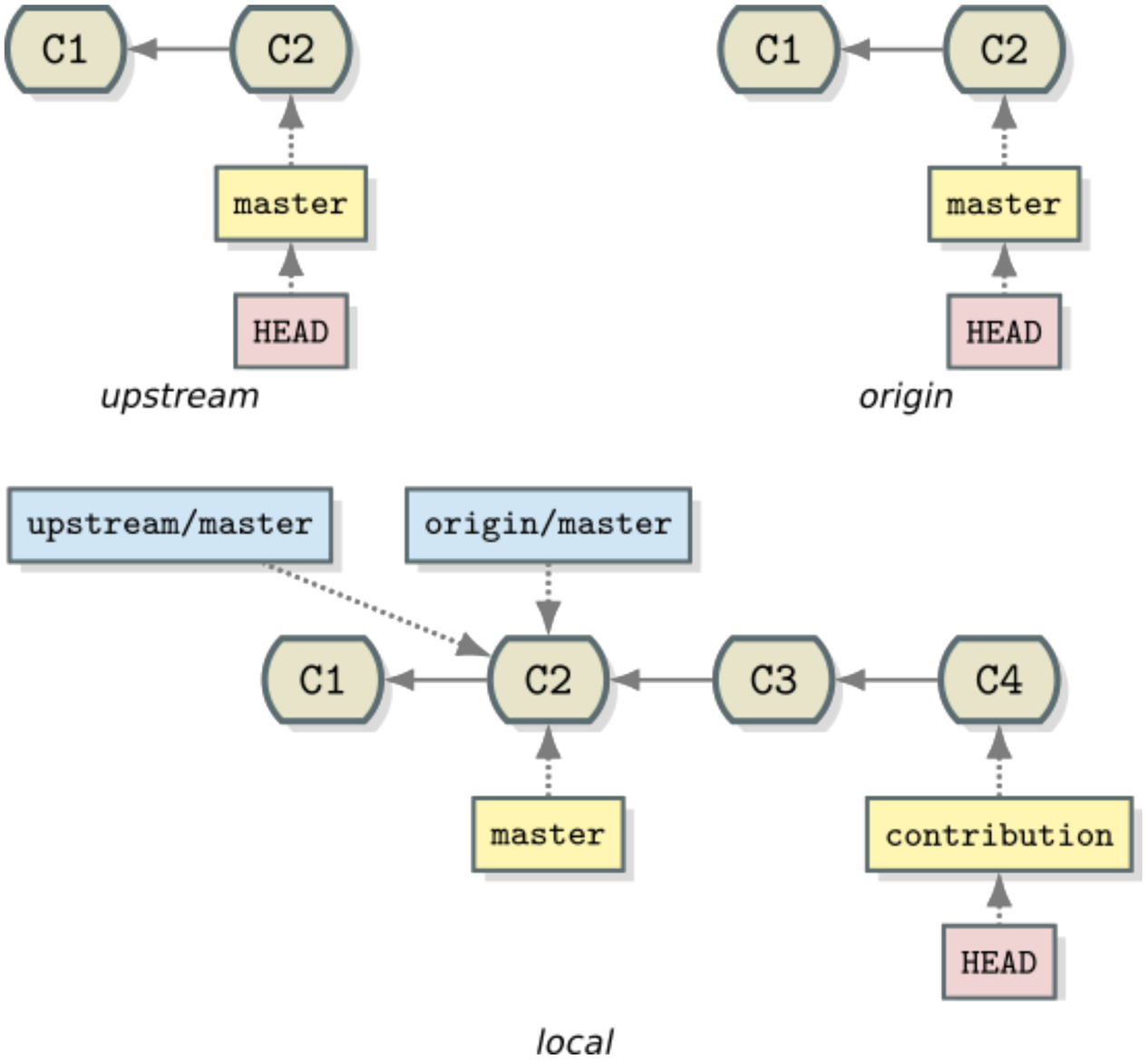
- Nous pouvons commencer a travailler

```
git add …
git commit …
… … …
```

Les pull-request - Coté contributeur



Les pull-request - Coté contributeur



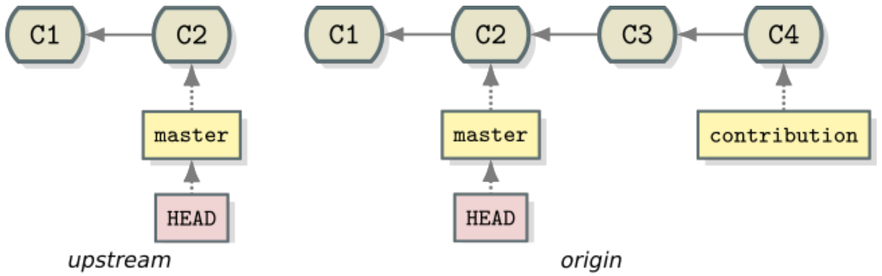
Les pull-request - Coté contributeur

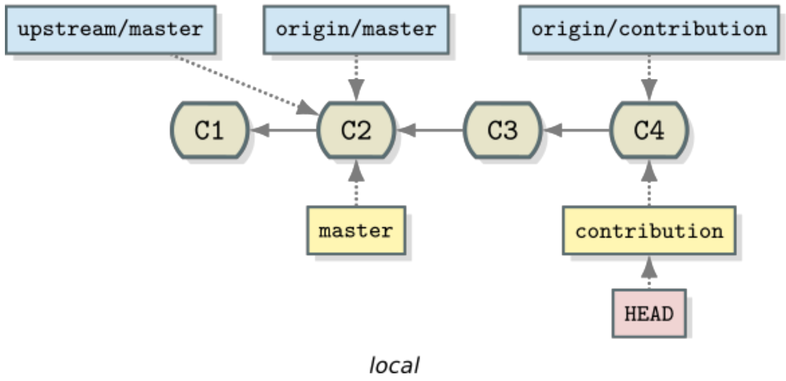
Publier notre contribution

```
git push
```

- GIT va publier sur la branche configurée en publication **`origin/contribution`**

  souvenez vous de la config **remote.pushdefault=origin** et **push.default=current**





Les pull-request - Coté contributeur

Soumettre la contribution

- Via l’interface web
  - Aller sur l’interface web de votre dépôt.
  - Suivre les instructions de la pull-request.
- En ligne de commande

```
git request-pull [-p] <start> <url> [<end>]
```

Les pull-request - Coté contributeur

Corriger une contribution

- Réitérer les étapes précédente :
  - Modifier le code
  - Publier la contribution
  - Créer la pull-request

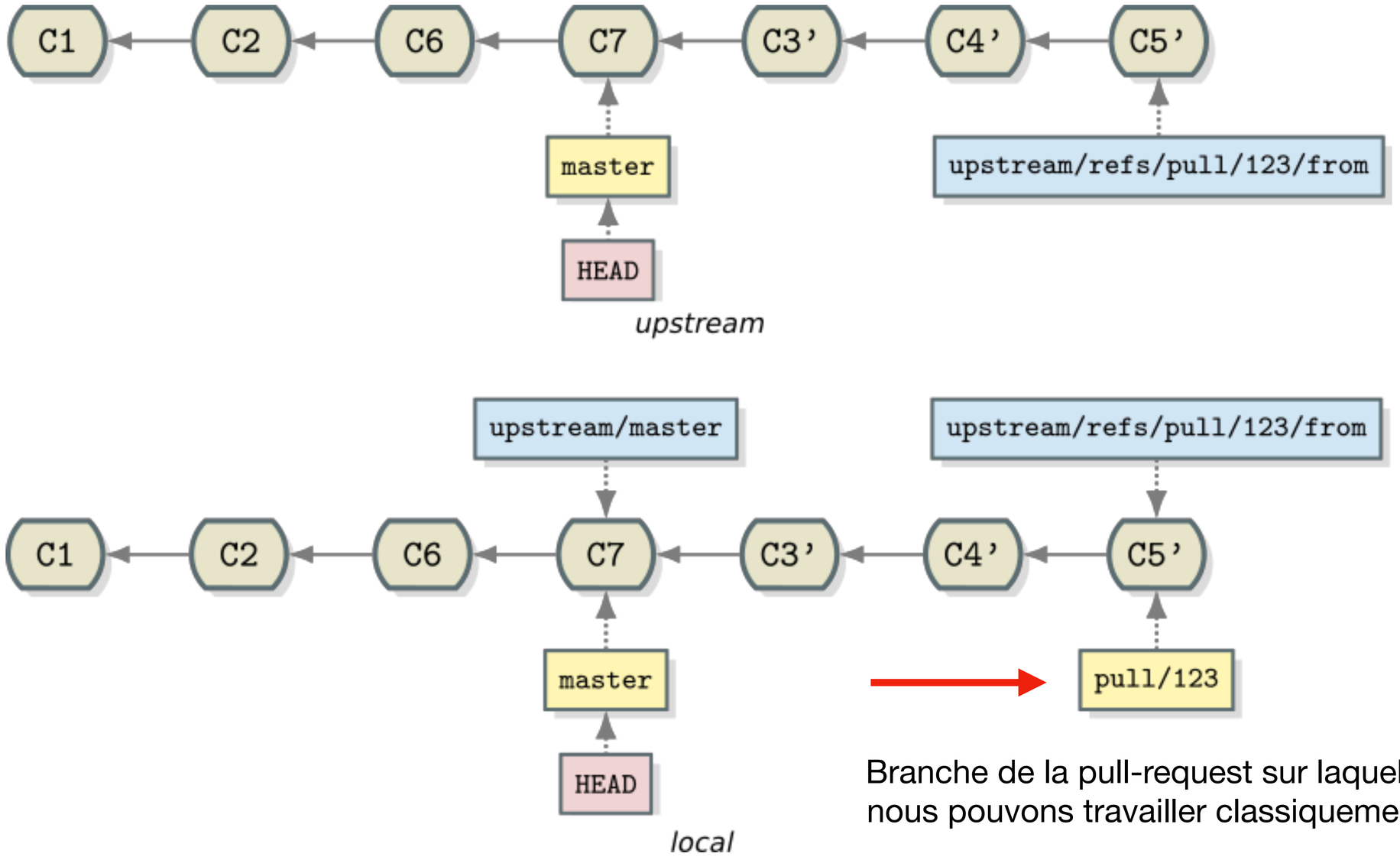
Les pull-request - Coté mainteneur

Vérifier une pull-request

- Récupérer les pull-requests :

```
# GitHub
git fetch upstream refs/pull/{PR_NUMBER}/from:pull/{PR_NUMBER}
# GitLab
git fetch upstream refs/merge-requests/{PR_NUMBER}/head:pull/
{PR_NUMBER}
# BitBucket Server
git fetch upstream refs/pull-requests/{PR_NUMBER}/from:pull/{PR_NUMBER}
# Bitbucket Cloud (obligé ici de taper directement sur le fork du
contributeur)
git fetch http://git.example.com/{PR_CONTRIBUTOR}/example
{PR_SOURCE_BRANCH}:pull/{PR_NUMBER}
```

Les pull-request - Coté mainteneur



Branche de la pull-request sur laquelle nous pouvons travailler classiquement

Les pull-request - Coté mainteneur

Intégrer une pull-request

- L’interface web peux proposer un bouton fusionner

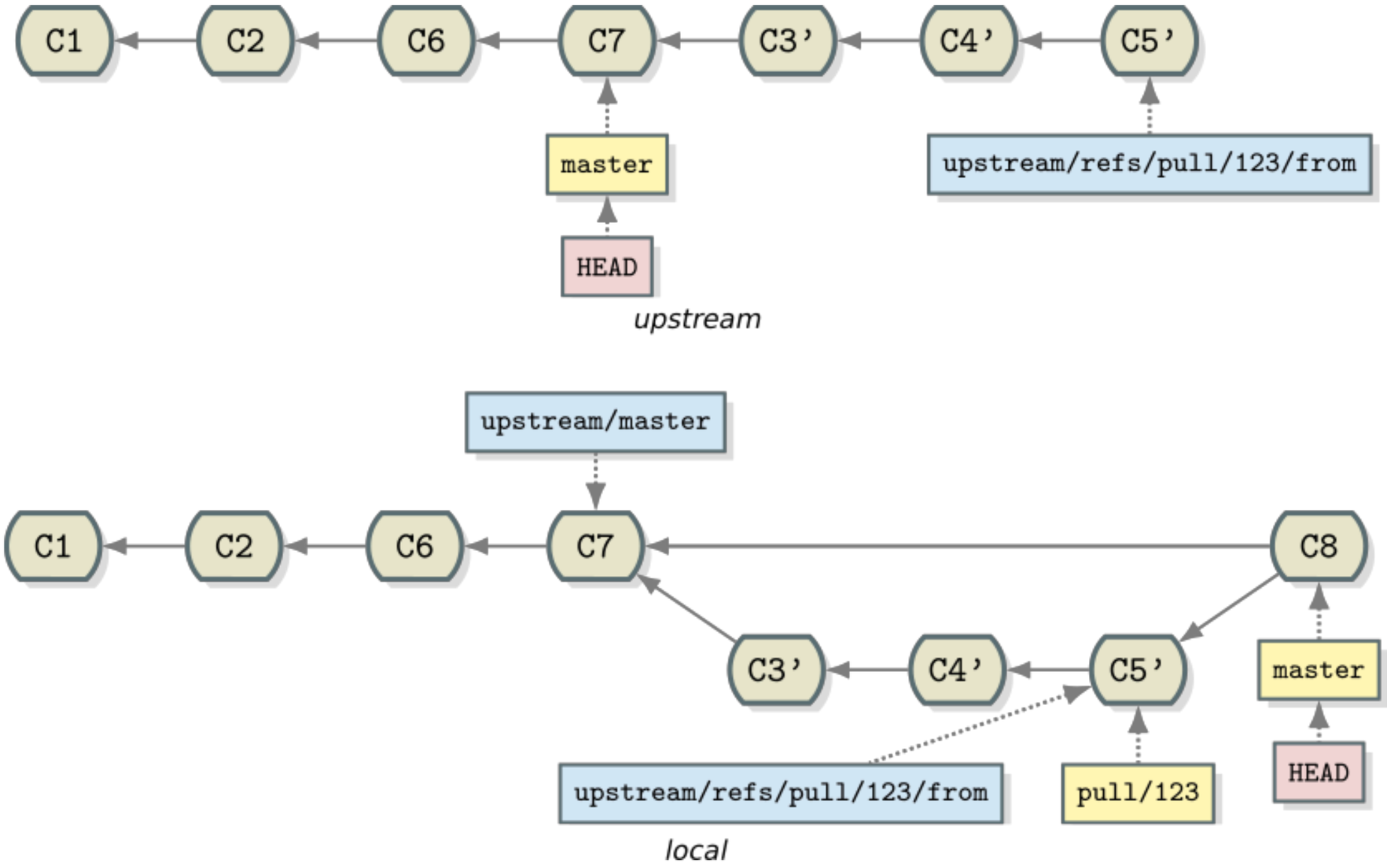
  !! Faites le uniquement après avoir vérifier le code de la pull-request !!

- Si la branche cible à évolué de manière conflictuelle,
- Ou vous préférez gérer la fusion vous même.

  - Récupérer la branche (point précédent)
  - Fusionner avec ou sans **`fast-forward`**, avec ou sans **`squashing`**

```
git merge --no-ff pull/{PR_NUMBER}
```

Les pull-request - Coté mainteneur

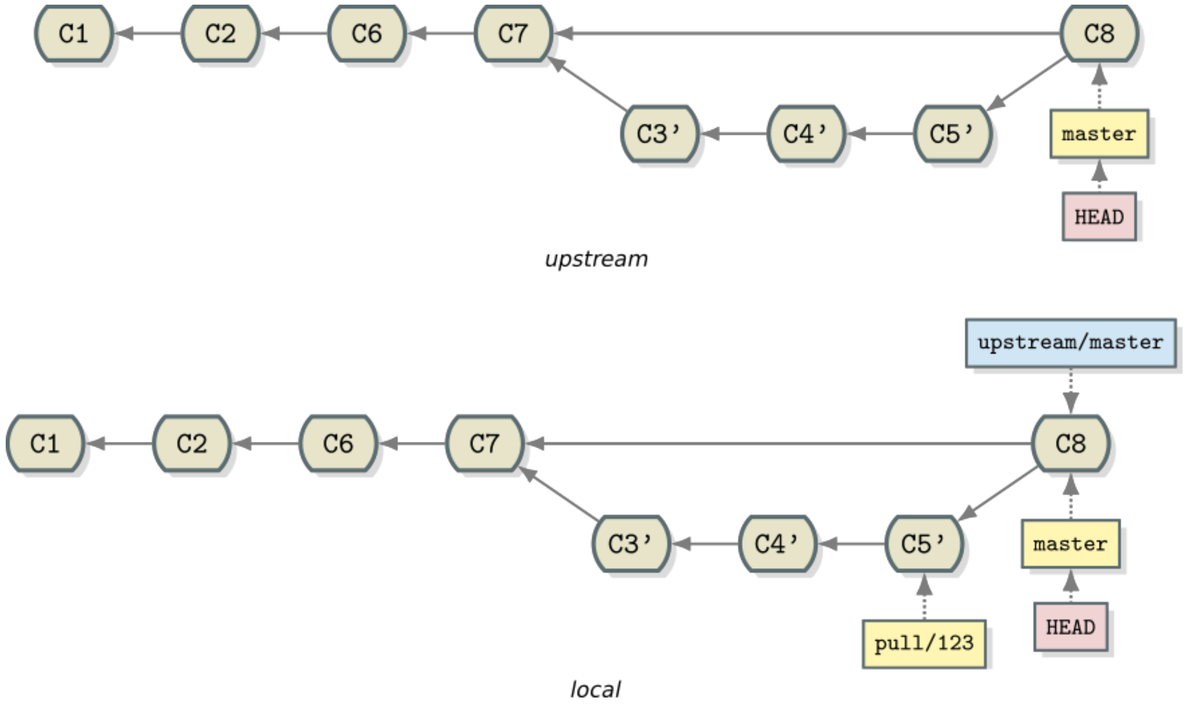


Les pull-request - Coté mainteneur

Publier le code

```
git push upstream
```

- La pull-request sera automatiquement fermée.



GIT

Gestion des commits

- **Les bonnes pratiques**

Les bonnes pratiques

Les bonnes pratiques

- 50 caractères maximum pour le titre, résumant les changements.
- La première ligne est traitée comme le sujet et le reste séparé par une ligne blanche, comme le corps du message.
- Utiliser le présent des verbes.
- Les listes à puces sont autorisées, typiquement avec un moins ou une astérisque.
- Le corps du message doit comprendre des lignes de 72 caractères maximum
  - `git format-patch --stdout` convertit une série de soumission en une série d'emails.
  - Le log Git ne revient pas automatiquement à la ligne, sans retour chariot il est donc étalé sur une seule ligne donc difficile à lire. Le nombre 72 est le résultat du calcul des 80 du terminal, moins les 4 de l'indentation et les 4 de sa symétrie à droite.

GIT

Gestion des Tags

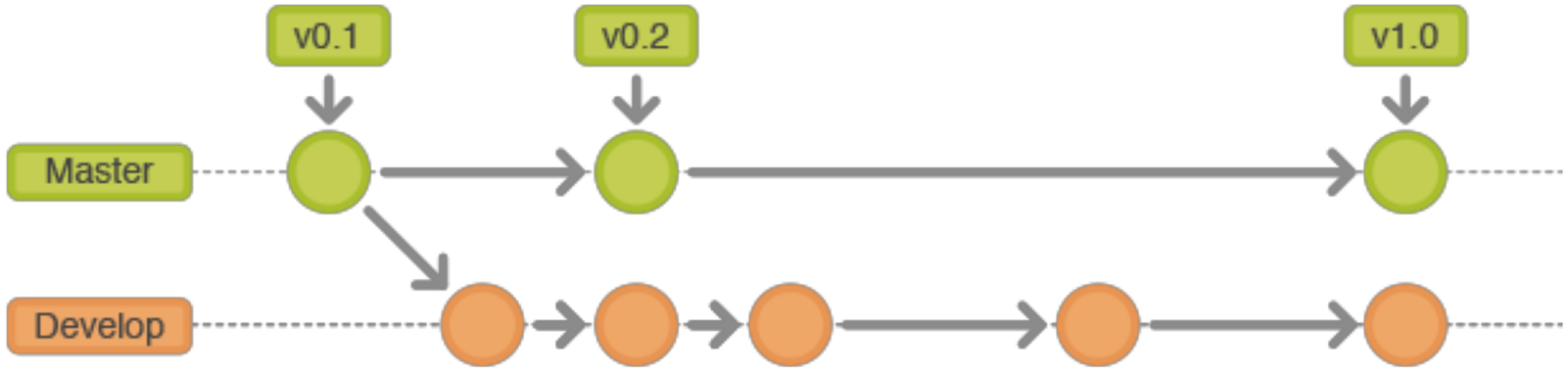
- **Introduction**
- **Lister et filtrer les tags**
- **Créer un tag**
- **Envoyer des tags**
- **Taguer d’anciens commits**

Introduction

Introduction

Le concept

- Les **branches** pointent toujours vers le dernier commit.
- Les **étiquettes** (ou tags) pointent vers des commits spécifiques.



- Une étiquette permet de donner un nom à un commit pour le retrouver plus facilement ou l'utiliser avec des outils comme **composer** ou **npm**…

Lister et filtrer les tags

Lister et filtrer les tags

Lister les étiquettes

- Affiche la liste des étiquettes :

```
git tag
```

- Les étiquettes s’affichent en ordre alphabétique.

Filtrer les étiquettes

- Avec l’option -l et une chaine motif :

```
git tag -l 'v1.*'
```

- La commande affiche tous les tags commençant par la chaine « **v1.** »

Créer un tag

Créer un tag

Les étiquettes légères

- « **Légère** » par ce que aucune branche n’est créée durant le processus.
- Une étiquette légère référence un commit spécifique.

```
git tag <tagname> <commit>|<object>
```

Créer un tag

Les étiquettes Objet (ou annotées)

- « **Objet** » par ce que l'étiquette est stockée comme un objet (commit, branche, …).
- On peut y ajouter un commentaire avec l'option **`-a`**,
- On peut le signer avec une méthode cryptographique avec l'option **`-u`**.

```
git tag -a <tagname> <commit>|<object>
```

- A moins que l'option **`-m`** ou **`-F`** soit fournie, GIT ouvre l'éditeur pour ajouter le message de l'étiquette.
- Ce type d'étiquette pointe vers un Objet Tag plutôt que vers un commit.

Envoyer les tags

Envoyer les tags

- Envoyer les tags sur le serveur distant :

```
git push --tag <remote> <branch>
```

Taguer d’anciens commits

Taguer d’anciens commits

- Ajouter une étiquette sur un ancien commit :

```
git tag -a <tagname> <commit> -m 'message'
```

- Propager l’étiquette sur les dépôts distant :

```
git push --tags <remote> <branch>
```

GIT

L’historique

- **Affichage des logs**
- **Affichages des logs en mode graphique**
- **Connaitre les auteurs des modifications**
- **Afficher le contenu d’un objet**
- **Dompter les logs**

Affichages des logs

Affichages des logs

- Afficher les logs avec la commande :

```
git log
```

- **`git log`** possède plus d'une centaine d'options.
- La majorité des options servent au filtrage.
- Consulter la liste des options et catégories d'option :

```
git help log
```

```
git log --pretty=oneline
```

```
git log -3
```

```
git log --author=<name>
```

```
git log --stat
```

```
git log --before|after <date>
```

```
git log --oneline
```

```
git log --pretty=format:'%h : %s' --topo-order --graph
```

Affichages des logs en mode graphique

Affichages des logs en mode graphique

Avec GITK

```
gitk --all
```

- On peut spécifier un fichier, un commit, une branche…

Avec GitHub Desktop

- Télécharger l'utilitaire : https://desktop.github.com/

Avec SourceTree

- Télécharger l'utilitaire : https://desktop.github.com/

Connaitre les auteurs des modifications

Connaitre les auteurs des modifications


```
git blame <filename>
```

- La commande git blame permet de connaitre les auteurs des modifications
- Avec l'option -L, il est possible de cibler les lignes précise du fichier

```
git blame -L 160,+10 <filename>
```

  - Git affichera les auteurs des modifications des lignes 160 à 170 du fichier

Afficher le contenu d'un objet

Afficher le contenu d’un objet


```
git show <object>
```

- Affiche le contenu d’un objet (branche, étiquette ou commit).
- Affiche par défaut : l’objet HEAD.

Dompter les logs

Dompter les logs

Créer sa propre commande d'affichage des logs

- Ajouter un peu (beaucoup) de config

```
 git config --global alias.prettylog "log --color --graph --
pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"
```

- On vient de créer **`prettylog`**, un alias de **`log`**

- Tester la nouvelle commande GIT

```
 git pretty log
```

GIT

Gestion des erreurs et déboggage

- **Les versions antérieurs**
- **Annuler un GIT Add**
- **Annuler un GIT Commit**
- **Annuler un GIT Push**
- **Résolution des conflits**

Les versions antérieures

Les versions antérieures

Commits non publiés

- Si les commits ne sont pas publiés, il suffit de revenir en arrière :

```
git reset --hard <commit>
```

- Pour conserver les modifications en cours qui ne sont pas indexées :

```
git stash # Sauvegarde des modifications
git reset --hard <commit> # R.A.Z. du répertoire de travail
git stash pop # Applique les modifications sauvegardées
```

Si le fichier est concerné par les modifications en cours et le retour à la version antérieur, il y aura un conflit à résoudre.

Les versions antérieures

Commits publiés

- Si les commits ont été publiés, il est possible de créer un patch :

```
git revert <commit>
```

- On peut préciser le nombre d'enregistrement à annuler :

```
git revert HEAD~4..HEAD
```

On annule 4 index dans l'objet HEAD

- Puis on commit pour finaliser le patch :

```
git commit
```

Annuler un GIT Add

Annuler un GIT Add

- Annuler un **`git add`**, c’est annuler l’indexation d’une modification

- Utiliser la commande :

```
git reset HEAD <file>
```

Modifier un commit

Modifier un commit

- **Attention**, certaines modifications sont permanentes.
- Utiliser la commande :

```
git commit --amend
```

- Cette commande reprend l’index du commit précédent.
- Cette commande ouvre l’éditeur avec les modifications et le message.

Annuler un GIT Push

Annuler un GIT Push

- Obtenir les numéros des commits :

```
git log -2
```

- Créer un patch entre la dernière version et la version -1:

```
git diff <commit-v-1> <commit-v0> > 01.patch.
```

- Appliquer le patch inversé :

```
patch -1 -R < 01.patch
```

- Valider les modifications et publier :

```
git commit -m 'message patch'
git push
```

Résolution des conflits

Résolution des conflits

Mise en situation :

- Modifier le même fichier dans deux branches qui seront fusionnées.
  - Vous modifiez et indexez et validez le fichier <**file**> de <**branch_b**>.
  - Vous basculez vers <**branch_a**>, et modifiez et indexez <**file**>, mais ne le validez pas.
  - Vous fusionnez <**branch_a**> avec <**branch_b**>
  - GIT retourne une erreur.

Solution :

- Faire le commit de <**branch_a**> avant la fusion.

GIT

Déployer avec GIT

- **Préparer le serveur**
- **Envoyer les sources**
- **Préparer Apache**

Préparer le serveur

Préparer le serveur

- Se connecter en SSH

```
ssh user@server
```

- Installer GIT

```
sudo apt-get install git
```

- Créer le dépôt « **hub** »

```
mkdir /home/user/project.git
```

Envoyer les sources

Envoyer les sources

- Revenir sur le dépôt local

- Ajouter le dépôt distant que l'on nomme « **origin** »

```
git remote add origin ssh://user@server/home/user/project.git
```

- Pousser les modifications de la branche « **master** »

```
git push origin master
```

Préparer Apache

Préparer Apache

- Retourner sur le serveur

- Cloner le dépôt vers le répertoire **www** de Apache

```
git clone /home/user/project.git /var/www/Project
```

Préparer Apache

- Automatiser la mise à jour

```
vim home/user/project.git/hooks/post-update
```

- Ajouter le code

```
#!/bin/bash
echo "********** mise en production *********"
cd /var/www/Project
unset GIT_DIR
git pull origin master
```

- Rendre le fichier exécutable

```
chmod +x home/user/project.git/hooks/post-update
```

GIT

Merci