

Formation Git



Pierre Sablé



Pourquoi utiliser git ?

Objectifs :


- ◆ **travailler à plusieurs sans se marcher dessus** : indispensable pour les projets en équipe
- ◆ **garder un historique propre de toutes les modifications** : on organise son travail sous forme de “commits” documentés





Rappel VCS

Rappel VCS :

- ◇ VCS : Version Control System
 - ◇ Enregistre les modifications d'un ensemble de fichiers
 - ◇ Permet de revenir en arrière sur une version spécifique
 - ◇ Permet de revenir en arrière sur un fichier spécifique
 - ◇ Gère des branches
 - ◇ Gère des conflits / merges
- 



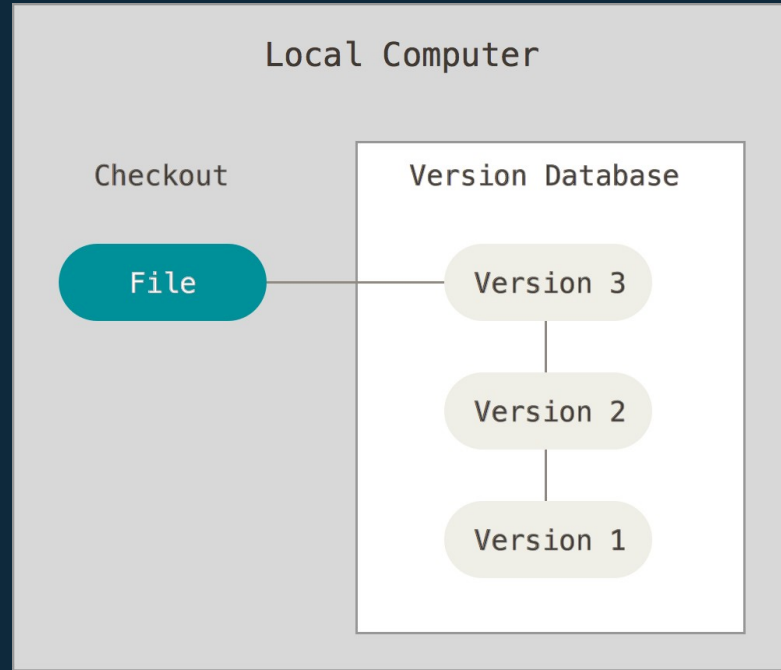
Contrôle de version local

Peut être fait manuellement :

- ◇ Copie des fichiers dans un répertoire spécifique
 - ◇ Renommage des fichiers
 - ◇ Archivage des fichiers
-
- Erreurs de manipulations faciles et non-révertibles
 - Lourdeurs de manipulations



Contôle de version local



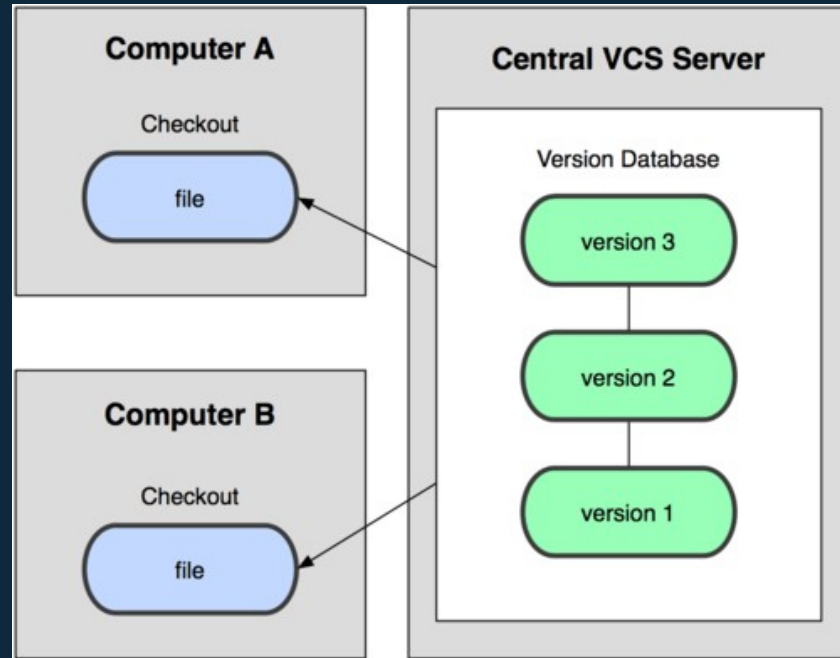


Contrôle de version centralisé

- ◇ Permet le travail collaboratif
- ◇ Serveur unique contenant toutes les versions
- ◇ Clients multiples empruntant des fichiers
- ◇ Mode de fonctionnement standard durant des années
 - CVS, Subversion



Contôle de version centralisé





Contrôle de version centralisé

- ◇ Chacun sait ce que tous les autres sont en train de faire sur le projet
- ◇ Un administrateur peut avoir un contrôle fin des permissions
- ◇ Point unique de panne
- ◇ Sauvegarde fiable à implémenter, corruption de données possible
- ◇ Pertes définitives possibles



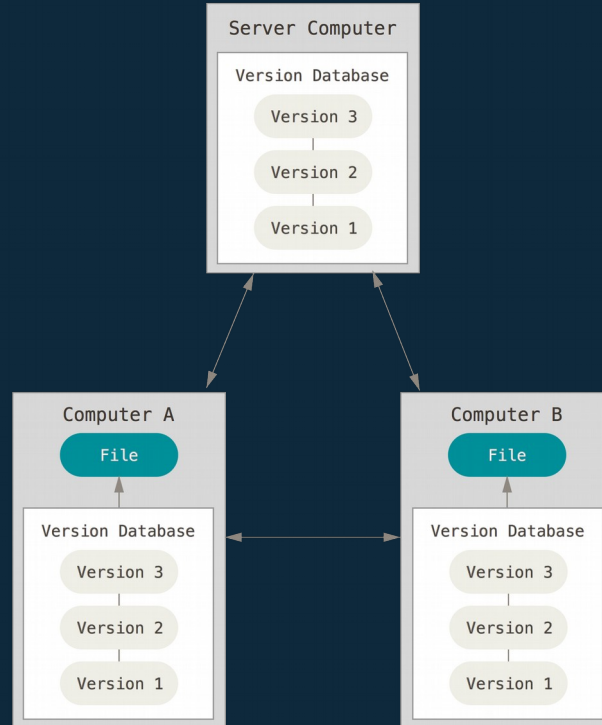


Contrôle de version distribué

- ◇ Plus une simple extraction mais une duplication du dépôt
- ◇ Sécurité : redondance des dépôts
 - N'importe quel dépôt d'un client peut être copié sur le serveur pour le restaurer
- ◇ Permet l'organisation de « groupes de travail »



Contôle de version distribué





Git





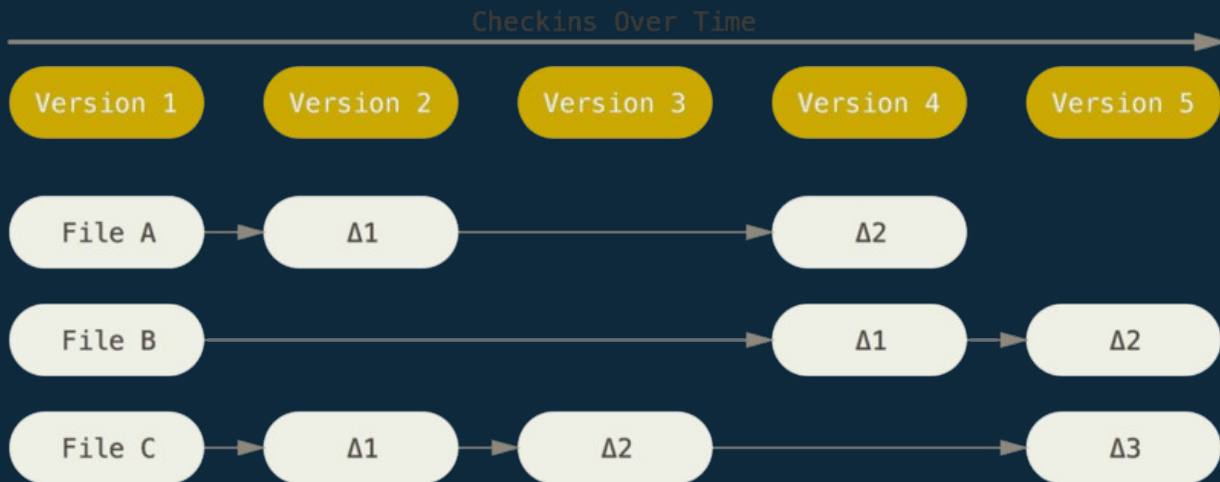
L'histoire...

- ◇ 1991 – 2002 : modifications du noyau Linux transmises sous forme de patches, d'archives
- ◇ 2002 – 2005 : le projet du noyau Linux utilise un DVCS (BitKeeper)
- ◇ 2005 : BitKeeper devient payant... Linus Torvald crée son successeur...
 - Objectifs : vitesse, simplicité, développements non linéaires (branches)
- ◇ Depuis, Git évolue et est adopté en majorité...
 - Incroyablement rapide !
 - Efficace pour toute taille de projet !



Mode de stockage

- ◇ La plupart des VCS gèrent l'information comme une liste de modifications, de différences sur chaque fichier dans le temps





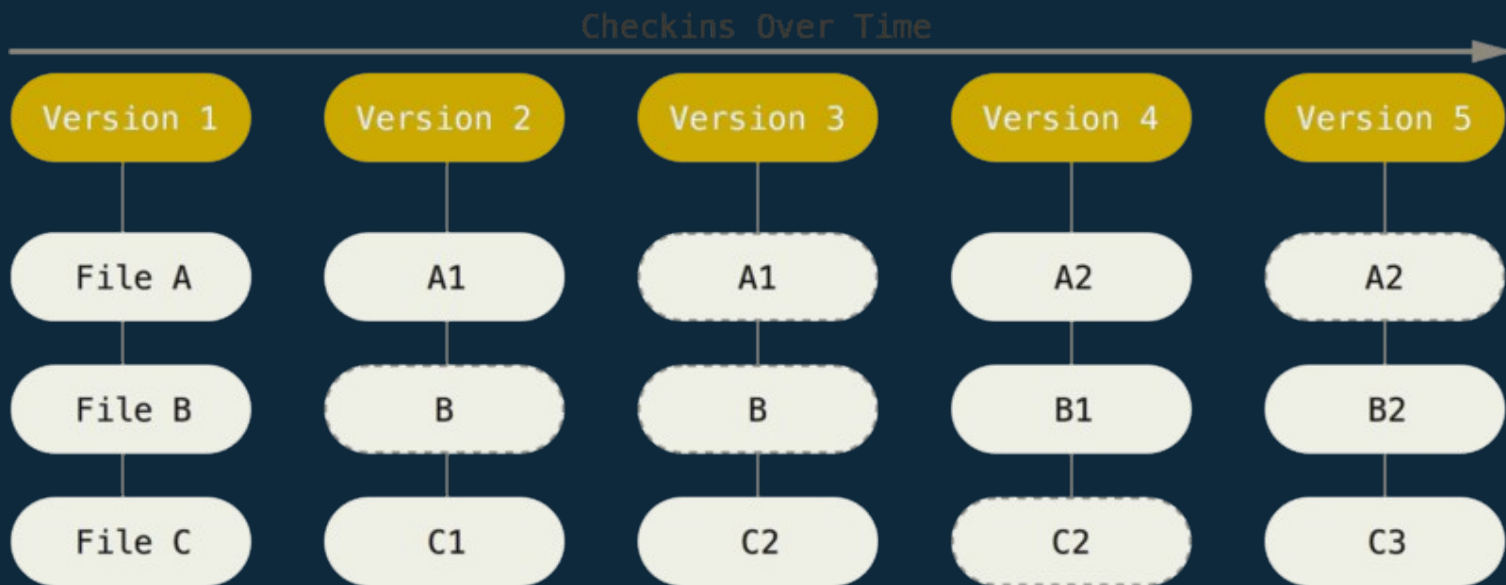
Mode de stockage

- ◇ Git gère les versions sous forme d'instantanés (snapshots)
 - Une validation ou un enregistrement de l'état prend un instantané du contenu et enregistre une référence à cet instantané.
 - Si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original.
- ◇ Git ressemble plus à un mini système de fichiers





Mode de stockage - Git





Travailler en local avec Git

- ◇ La plupart des opérations se déroulent en local
 - Pas de ralentissement dû à la latence du réseau
 - Recherche et calcul des états, différences
- ◇ Travail en mode « hors connexion » !!





Gestion de l'intégrité

- ◇ Tout est vérifié par une somme de contrôle avant d'être stocké (SHA1)
 - Pas de risque de modification de contenu sans que Git s'en aperçoive.
 - Pas de perte de données lors d'un transfert
- ◇ Git stocke tout dans sa base de donnée
- ◇ Principe général d'ajout de données
 - Difficile de faire réaliser des actions qui ne soient pas réversibles
- ◇ Perte ou corruption de modifications qui n'ont pas encore été entrées en base...





La théorie de git

3 zones, 3 ambiances

Les modifications sont sauvegardées 3 fois

Working directory

C'est la zone de travail : les fichiers tout juste modifiés sont ici

Index

Zone qui permet de stocker les modifications sélectionnées en vue d'être commitées

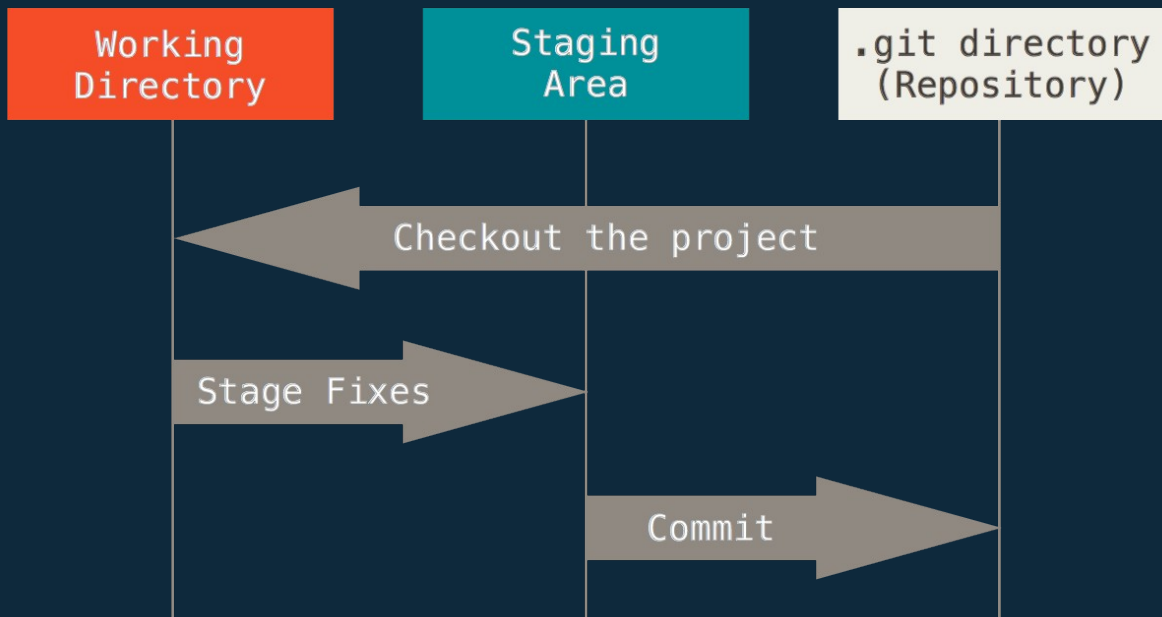
Local repository

Code commité, prêt à être envoyé sur un serveur distant



La théorie de git

3 zones, 3 ambiances





Installation de Git





Paramétrage, personnalisation

◇ Outil : `$ git config`

- Définir les paramètres basiques
 - Identité
 - Editeur de texte

◇ 3 niveaux de configuration

- `/etc/gitconfig`
- `~/.gitconfig`
- `config`

◇ `--global / --system`

◇ Vérification : `$ git config --list`

◇ Obtenir de l'aide : `$ git help <commande>`



Configuration minimale

```
git config --global user.name "Prénom Nom"  
git config --global user.email "prenom.nom@student.ecp.fr"
```

En option mais c'est mieux :

```
git config --global color.ui true  
git config --global color.diff.meta yellow
```





Création du premier dépôt





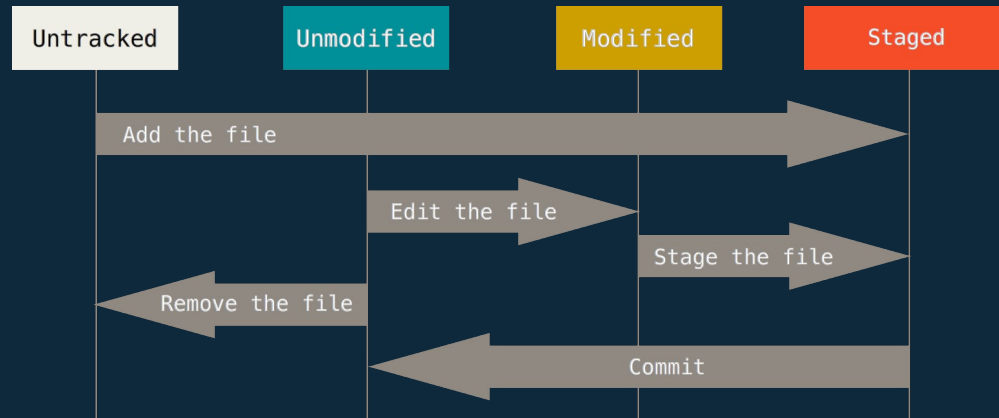
Premier dépôt, nouveau projet

◇ Un nouveau projet se génère à partir :

- À partir d'un répertoire existant
- `$ git init`
- À partir d'un dépôt distant
- `$ git clone`

Enregistrement

- ◇ Enregistrer les modifications dans le dépôt. État souhaité et valide



- ◇ Cycle d'édition, indexation et enregistrement...



Vérification du dépôt

- ◇ Il faut déterminer quels fichiers sont dans quel état !
 - `$ git status`



Ajouter un fichier « non-suivi »

- ◇ Créer un nouveau fichier README.md
- ◇ Vérifier le dépôt
 - `$ git status`
- ◇ Suivi de ce nouveau fichier
 - `$ git add README.md`
- ◇ Ajout de fichier, suivi... modification de fichier, suivi...



Vérification du dépôt – version courte

◇ Affichage compact

- `$ git status -s`

◇ ?? : Fichiers non suivis

◇ A : Nouveau fichiers indexés

◇ M : Fichier déjà indexés et modifiés

◇ M : Fichier modifié mais non ré-indexé



Ignorer des fichiers

◇ Il est possible de définir des règles pour ignorer certains fichiers

- Logs, secrets, caches...

◇ `$.gitignore`

- Une ligne est une règle d'exclusion
- Patron standard de fichiers
 - « ? » caractère quelconque
 - « * » suite quelconque de caractères
 - « [abc] », « [a-f] » caractères alternatifs
- Terminer par « / » pour un répertoire
- Commencer par « ! » pour une exception
- Commenter avec « # »





Inspecter les données indexées et non indexées

- ◇ Plus précis que `$ git status` connaître les fichiers modifiés et le contenu modifié
- ◇ Visualiser ce qui a été modifié et non indexé
 - `$ git diff`
- ◇ Visualiser les modifications indexées et qui feront partie de la prochaine validation
 - `$ git diff --staged`





Validation ! Les commits

Commit : ensemble de modifications cohérentes du code

Un bon commit est un commit :

- ◇ qui ne concerne qu'une seule fonctionnalité du programme
- ◇ le plus petit possible tout en restant cohérent
- ◇ Idéalement qu'il compile seul

C'est quoi concrètement un commit ?

- ◇ une différence (ajout / suppression de lignes)
- ◇ des méta-données (titre, hash, auteur)



Les commits

- ◇ Une fois l'index dans l'état désiré, il peut être validé
 - `$ git commit`
- ◇ Un message est clairement utile pour expliquer la modification !
 - `$ git commit -m « Story 12 : [FIX] add html code »`
- ◇ /!\ L'option « -a » permet de valider tous les fichiers du « working dir » sans passer nécessairement par l'index
 - `$ git commit -am « Story 13 : [ADD] new features »`



Aller jusqu'au commit

3 zones, 3 ambiances

Les modifications sont sauvegardés 3 fois

Working directory

C'est la zone de travail : les fichiers tout juste modifiés sont ici

Index

Zone qui permet de stocker les modifications sélectionnées en vue d'être committées

Local repository

Code commité, prêt à être envoyé sur un serveur distant

git add

git commit



Aller jusqu'au commit

Où j'en suis dans mes 3 zones ?

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: views/add_commentaire.php

no changes added to commit (use "git add" and/or "git commit -a")



Aller jusqu'au commit

Visualiser les différences entre le working directory et l'index

```
$ git diff
```

```
diff --git a/views/add_commentaire.php b/views/add_commentaire.php
index e69de29..8cff573 100644
--- a/views/add_commentaire.php
+++ b/views/add_commentaire.php
@@ -0,0 +1,6 @@
<?php include('header.php'); ?>

+<form action="/commentaire/ajouter" method="post">
+    <p>Pseudo : <input type="text" name="pseudo"/></p>
+    <p>Commentaire : <textarea name="commentaire"></textarea></p>
+</form>
```



Aller jusqu'au commit

Ajouter mes modifications à la zone de staging (index)

```
$ git add views/add_commentaire.php  
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: views/add_commentaire.php



Aller jusqu'au commit

Récapitulatif des commandes

affichage
différences

`git diff`

`git diff --staged`

working directory

index

git repository

ajouter des
modifications

```
git add mon_fichier  
git add -p (interactif)  
git add -A (tout  
ajouter)
```

```
git commit -m  
"message"
```



Supprimer un fichier

- ◇ La suppression d'un fichier impose son élimination des fichiers en suivi de version
 - Le fichier est également effacé de la copie de travail
 - `$ git rm`
 - Le fichier est conservé dans la copie de travail
 - `$ git rm --cached`



Déplacer un fichier

- ◇ Concrètement il n'y a pas de suivi du renommage dans Git. Il faut retirer l'ancien fichier de l'index et ajouter le nouveau

- `$ mv project_gold project_diamond`
- `$ git rm project.gold`
- `$ git add project diamond`

- ◇ Git le fait de manière implicite pour nous

- `$ git mv project_gold project_diamond`



Et les logs ??





Visualiser l'historique

- ◇ `$ git log` :
Liste en ordre chronologique inversé les validations réalisées.
Beaucoup d'options disponibles !!
- ◇ Filtrer par auteur ou contenu du message
 - `$ git log --author=psable --grep=formation`
- ◇ Personnalisation du format avec somme de contrôle abrégée, sujet et représentation du graphe
 - `$ git log --pretty=format: "%h %s" --graph`
- ◇ Spécifier une date limite de début|fin
 - `$ git log --since=2.weeks --until=2019-12-01`



Revenir au dernier commit

```
$ git diff
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working  
directory)
```

```
    modified:   model/commentaires_model.php
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```



Revenir au dernier commit

Enlever des modifications dans le working directory

```
$ git checkout -- monfichier  
$ git status
```

```
On branch master  
nothing to commit, working directory clean
```



Revenir au dernier commit

Revenir à un commit précis pour un fichier

```
$ git checkout IDCommit monfichier  
$ git status
```

Visualiser le contenu d'un commit (mode spectateur)

```
$ git checkout IDCommit
```

```
$ git checkout master
```



Désindexer des fichiers

```
$ git diff
```

```
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)
```

```
    modified:   model/commentaires_model.php
```



Désindexer des fichiers

```
$ git reset HEAD monFichier  
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: model/commentaires_model.php

no changes added to commit (use "git add" and/or "git commit -a")



Corriger un commit

◇ En cas de commit précoce ! Ou incomplet...

- Validation initiale
 - `$ git commit -m`
- Ajout d'un fichier oublié
 - `$ git add fichierOublie`
- Nouvelle validation sur le même commit
 - `$ git commit --amend`



Aller jusqu'au commit

Récapitulatif des commandes

working directory

index

git repository

`git diff`

`git diff --staged`

`git add monfichier`

`git commit -m "titre"`

`git commit -am "titre"` (si le fichier a déjà été indexé)

`git checkout --
monfichier`

`git reset HEAD
monfichier`

`git reset --hard`

A decorative graphic on the left side of the slide. It features a large, central cyan hexagon. Surrounding it are several smaller hexagons of varying shades of blue and cyan. Some of these hexagons contain white icons: a lightbulb, a thumbs-up, a smartphone, a magnifying glass, a gear, and a speech bubble. There is also a small network diagram icon with a central node and several connected nodes.

Les dépôts distants

Centraliser les données sur un dépôt git !

Les dépôts distants

- ◇ git directory sur un serveur distant pour le travail collaboratif

- ◇ Dépôts distants :

- github
- gitlab
- gitea
- ...



GitLab



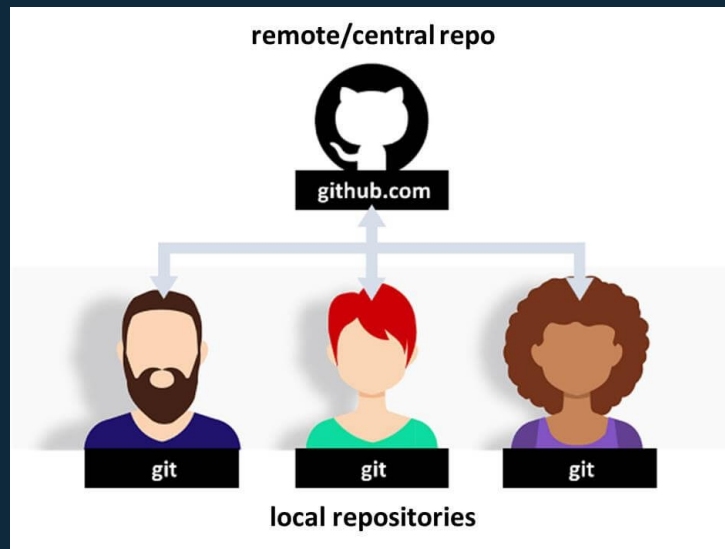
GitHub

Pourquoi github / gitlab ?

- ◇ code review, merge request, interface web, ...

Les dépôts distants : remote

- ◇ Afficher un dépôt distant
 - `$ git remote -v`
- ◇ Ajouter un dépôt distant
 - `$ git remote add`
- ◇ Supprimer / renommer un dépôt distant
 - `$ git remote remove/rename`





Voir et ajouter des dépôts distants

Cloner un dépôt distant : crée un dossier et récupère les fichiers

```
git clone <url>
```



Envoyer sur le dépôt distant

```
git push
```

Envoie notre local repository sur le dépôt distant

On ne touche plus aux commits pushés !



Récupérer depuis le dépôt distant

◇ Obtenir les données du dépôt distant

- `$ git fetch`
- Ne fusionne pas automatiquement avec les dépôt local

◇ Récupérer les données et essayer de les fusionner directement

- `$ git pull`



Le schéma de base de git

Récapitulatif des commandes

**working
directory**

index

git repository

**remote git
repository**

`git diff`

`git diff --staged`

`git add`

`git commit`

`git checkout --`

`git reset`

`git reset --hard`


`git push`

`git pull`

`git fetch`



Authentification par clé

 Pour accéder aux dépôts sur gitlab, il faut y ajouter sa clé

On génère une paire de clés :
`ssh-keygen -t rsa`

On affiche le contenu de la clé publique :
`cat ~/.ssh/id_rsa.pub`

On copie **tout** le contenu de la clé publique sur gitlab > mon profil > settings > clés SSH





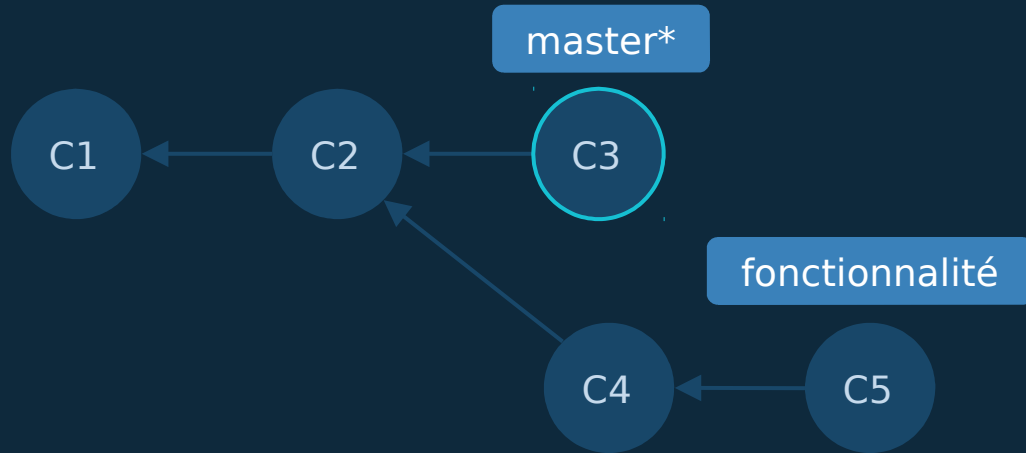
Les branches

- ◇ branche : **pointeur** vers un commit
- ◇ une branche principale : **master**
- ◇ Branche courante : **HEAD**
- ◇ en général, une branche par fonctionnalité en cours de développement (notion de *workflow*)





Les branches



Les branches et commits

Arbre de commits dans le git repository





Gestion des branches

Création et modification de branches

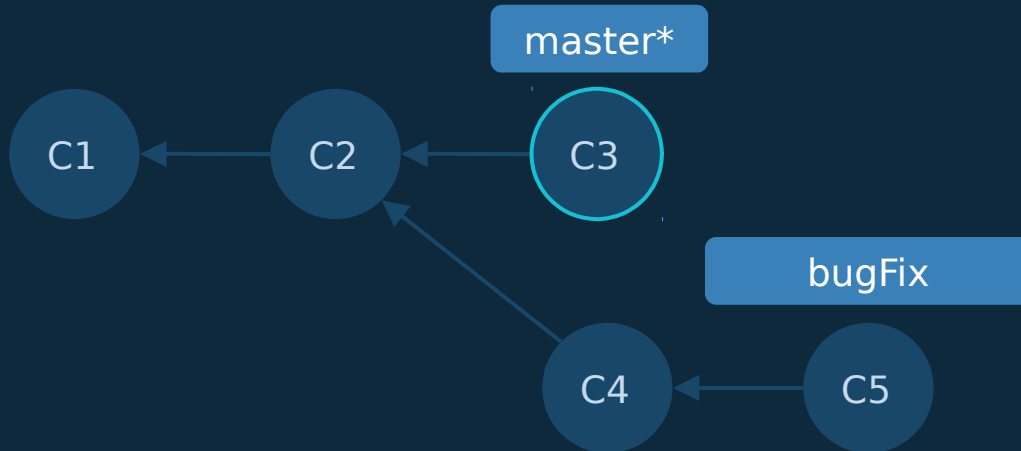
- ◇ Affichage des branches
 - `$ git branch`
- ◇ Créer une branche
 - `$ git branch ma_branche`
- ◇ Déplacer le HEAD vers ma_branche
 - `$ git checkout ma_branche`
- ◇ Créer et déplacer le HEAD
 - `$ git checkout -b ma_branche`



Gestion des branches

Merge : intégration des modifications d'une branche dans la branche courante

git merge *ma_branche*: *merge ma branche dans la branche courante*

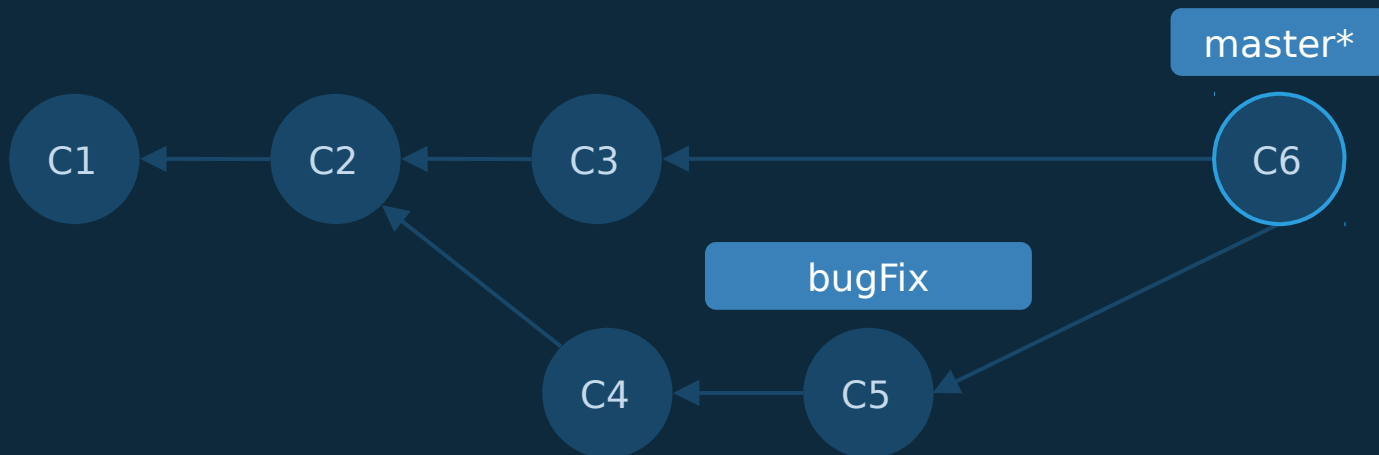




Gestion des branches

Merge : intégration des modifications d'une branche dans la branche courante

git merge *ma_branche: merge ma branche dans la branche courante*





Le remisage : Stash




Le stash ça sert à :

- ◇ Sauvegarder les modifications du working directory dans une zone tampon pour rendre le working directory propre.
- ◇ Possibilité de rejouer les modifications stashées n'importe où
- ◇ Peut être vu comme une zone de brouillons





Stash : les commandes



On stash un ensemble de modifications
`git stash`

On récupères les modifications stashées
`git stash apply`

Pour plusieurs stashes :
`git stash list`
`git stash apply stash@{id}`

Effacer le contenu du stash
`git stash clear`





Pour aller plus loin avec git...

- ◇ **git rebase** ou **git merge** ?
- ◇ balader ses commits avec **git cherry-pick**
- ◇ afficher un commit : **git show** <commit>
- ◇ Faire une réclamation : **git blame** <fichier>
- ◇ visualiser l'historique des commits : **git log**
- ◇ J'ai tout cassé ! **git reflog**





GIT



\(^__-)/

Dawan / Pierre Sablé / 2020
Version 1.0



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 4.0 International.

