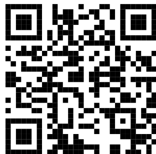
Gérer son historique de fichiers LaTeX avec Git

Maïeul Rouquette

Université de Lausanne — IRSB

12 juin 2019

https://geekographie.maieul.net/231



Licence Creative Commons France 3.0 - Paternité - Partage à l'identique

1. Gestionnaire de version, quesaco ?

2. Versionner son travail individuel

3. Travailler à plusieurs

4. Pour aller plus loin

Section 1

Gestionnaire de version, quesaco ?

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers
  - Marquer les grandes étapes du travail

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers
  - Marquer les grandes étapes du travail
  - Synchroniser facilement les fichiers et l'historique avec d'autres personnes

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers
  - Marquer les grandes étapes du travail
  - Synchroniser facilement les fichiers et l'historique avec d'autres personnes
  - Proposer et accepter facilement des nouvelles modifications

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers
  - Marquer les grandes étapes du travail
  - Synchroniser facilement les fichiers et l'historique avec d'autres personnes
  - Proposer et accepter facilement des nouvelles modifications
  - Gérer plusieurs versions parallèles d'un même projet afin d'avancer de façons indépendantes sur des points indépendants

Un gestionnaire de version, pour quoi faire ?

- Écrire un texte, créer ses propres macros, personnaliser LaTeX, créer ses packages impliquent :
  - Une démarche progressive
  - Des risques d'erreur
  - Parfois un travail collaboratif avec d'autres personnes
  - Parfois la nécessité d'essayer de nouvelles choses sans être certain du résultat
- Il faut donc pouvoir
  - Avoir un historique des fichiers
  - Marquer les grandes étapes du travail
  - Synchroniser facilement les fichiers et l'historique avec d'autres personnes
  - Proposer et accepter facilement des nouvelles modifications
  - Gérer plusieurs versions parallèles d'un même projet afin d'avancer de façons indépendantes sur des points indépendants
- C'est le but d'un logiciel de versionnement (ou versionnage) ou gestionnaire de version

Quelques autres intérêts d'un gestionnaire de version

- En informatique : souvent couplé avec une forge logicielle

Quelques autres intérêts d'un gestionnaire de version

- En informatique : souvent couplé avec une forge logicielle
  - Gestion de tickets

Quelques autres intérêts d'un gestionnaire de version

- En informatique : souvent couplé avec une forge logicielle
  - Gestion de tickets
  - Outils automatisés de test

Quelques autres intérêts d'un gestionnaire de version

- En informatique : souvent couplé avec une forge logicielle
  - Gestion de tickets
  - Outils automatisés de test
- En rédactionnel : pousse à travailler étape par étape, et donc à ne pas se disperser (ex : ma thèse de doctorat)

Quelques autres intérêts d'un gestionnaire de version

- En informatique : souvent couplé avec une forge logicielle
  - Gestion de tickets
  - Outils automatisés de test
- En rédactionnel : pousse à travailler étape par étape, et donc à ne pas se disperser (ex : ma thèse de doctorat)
- En général : outil simple de sauvegarde

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - ★ Simple d'approche
    - ★ Économe en ressources
  - Inconvénients
    - ★ Peu souple, plus complexe pour gérer des versions parallèles
    - ★ Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - ★ Simple d'approche
    - ★ Économe en ressources
  - Inconvénients
    - ★ Peu souple, plus complexe pour gérer des versions parallèles
    - ★ Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages
    - ★ Pas besoin de connexion internet pour travailler

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages
    - Pas besoin de connexion internet pour travailler
    - Souple, puissant

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages
    - Pas besoin de connexion internet pour travailler
    - Souple, puissant
  - Inconvénients

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages
    - Pas besoin de connexion internet pour travailler
    - Souple, puissant
  - Inconvénients
    - Plus complexe à appréhender

Gestionnaire de version centralisé vs décentralisé

- Centralisé (CVS, SVN)
  - Un serveur contient l'historique unique des modifications
  - Des utilisateurs, depuis leurs ordinateurs, envoient leurs modifications, ce qui complète l'historique
  - Avantages
    - Simple d'approche
    - Économe en ressources
  - Inconvénients
    - Peu souple, plus complexe pour gérer des versions parallèles
    - Nécessite une connexion internet pour travailler
- Décentralisé (Git, Mercurial)
  - Un historique (ou plus) conservé sur l'ordinateur de chaque utilisateur
  - Historiques synchronisés, soit directement, soit, le plus souvent, via un serveur de référence
  - Avantages
    - Pas besoin de connexion internet pour travailler
    - Souple, puissant
  - Inconvénients
    - Plus complexe à appréhender
    - Plus gourmand en ressources

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)
  - Dans l'interface de certains éditeurs de texte (*Atom*, *Vim*, *Emacs*)

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)
  - Dans l'interface de certains éditeurs de texte (*Atom*, *Vim*, *Emacs*)
  - En interface web

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)
  - Dans l'interface de certains éditeurs de texte (*Atom*, *Vim*, *Emacs*)
  - En interface web
  - L'interface ligne des commande est la base, les autres interfaces n'ont pas toujours toutes les fonctionnalités

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)
  - Dans l'interface de certains éditeurs de texte (*Atom*, *Vim*, *Emacs*)
  - En interface web
  - L'interface ligne des commande est la base, les autres interfaces n'ont pas toujours toutes les fonctionnalités
- *Github* est une forge logicielle, propriétaire

Un peu de terminologie autour de *Git*

- *Git* est un logiciel de versionnement pouvant s'exécuter :
  - En ligne de commande
  - Par interface graphique spécifique (`https://git-scm.com/downloads/guis`)
  - Dans l'interface de certains éditeurs de texte (*Atom*, *Vim*, *Emacs*)
  - En interface web
  - L'interface ligne des commande est la base, les autres interfaces n'ont pas toujours toutes les fonctionnalités
- *Github* est une forge logicielle, propriétaire
- *Gitlab* est une forge logicielle, libre, dont *Framagit* est l'une des instances

Section 2

Versionner son travail individuel

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment
    - « Windows Explorer Integration »

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment
    - « Windows Explorer Integration »
    - « Use git from the Windows command Prompt »

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment
    - « Windows Explorer Integration »
    - « Use git from the Windows command Prompt »
    - « Checkout Windows-style, commit Unix-style line endings »

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment
    - « Windows Explorer Integration »
    - « Use git from the Windows command Prompt »
    - « Checkout Windows-style, commit Unix-style line endings »
- Dans votre *Terminal* (*Git Bash* sous Windows), exécuter les commandes suivantes :

Installation et configuration

- Pour l'installation, suivre la procédure correspondant à votre système d'exploitation indiquée sur `https://git-scm.com/downloads`
  - GNU/Linux : avec gestionnaire de paquets
  - Mac Os : fichier d'installation à télécharger puis à exécuter
  - Windows : idem, choisir les options par défaut, notamment
    - « Windows Explorer Integration »
    - « Use git from the Windows command Prompt »
    - « Checkout Windows-style, commit Unix-style line endings »
- Dans votre *Terminal* (*Git Bash* sous Windows), exécuter les commandes suivantes :

Code 1 : Configuration de base de *Git*

```
% git config --global user.name "<Prénom> <Nom>"
% git config --global user.email "<email>"
```

La notion de dépôt

- Un dépôt contient l'historique du projet

La notion de dépôt

- Un dépôt contient l'historique du projet
- Il peut être local ou distant

La notion de dépôt

- Un dépôt contient l'historique du projet
- Il peut être local ou distant
- Un dépôt peut récupérer (`pull`) ou envoyer (`push`) de l'historique depuis/vers un autre dépôt (`remote`)

Création du dépôt local

Code 2 : Création d'un dépôt local

```
% cd <dossier où créer le depôt>
% git init <nom du depôt>
% cd <nom du depôt>
```

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant *t*
- Garder un historique avec *git*, c'est garder une succession de *commits*

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant *t*
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit
  - L'auteur du commit (identifié par nom et courriel)

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit
  - L'auteur du commit (identifié par nom et courriel)
  - Un court message résumant les modifications apportées (par ex. « relecture du chap. 23 »)

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit
  - L'auteur du commit (identifié par nom et courriel)
  - Un court message résumant les modifications apportées (par ex. « relecture du chap. 23 »)
  - Le ou les commit(s) parent(s)

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit
  - L'auteur du commit (identifié par nom et courriel)
  - Un court message résumant les modifications apportées (par ex. « relecture du chap. 23 »)
  - Le ou les commit(s) parent(s)
- Un commit possède un identifiant unique, un *hash* de 40 caractères hexadecimaux

Le commit, élément de base

- Un *commit* correspond à l'état des fichiers à un instant t
- Garder un historique avec *git*, c'est garder une succession de *commits*
- Dans un *commit* sont stockés :
  - L'état des fichiers à un instant t
  - Le moment du commit
  - L'auteur du commit (identifié par nom et courriel)
  - Un court message résumant les modifications apportées (par ex. « relecture du chap. 23 »)
  - Le ou les commit(s) parent(s)
- Un commit possède un identifiant unique, un *hash* de 40 caractères hexadecimaux
- Généralement, les huit premiers caractères du *hash* suffisent à identifier un commit

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire
  - On ne mélangera pas relectures orthographiques, relectures de fond et corrections de macro dans un même commit

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire
  - On ne mélangera pas relectures orthographiques, relectures de fond et corrections de macro dans un même commit
  - Le cas échéant, il est possible de commiter partiellement une modification (`git commit -p`)

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire
  - On ne mélangera pas relectures orthographiques, relectures de fond et corrections de macro dans un même commit
  - Le cas échéant, il est possible de commiter partiellement une modification (`git commit -p`)
- En règle générale il ne faut versionner que les fichiers sources (`.tex`, `.bib`) pas les fichiers finaux (`.pdf`) ni les fichiers intermédiaires (`.aux`, `.log`, etc.)

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire
  - On ne mélangera pas relectures orthographiques, relectures de fond et corrections de macro dans un même commit
  - Le cas échéant, il est possible de commiter partiellement une modification (`git commit -p`)
- En règle générale il ne faut versionner que les fichiers sources (`.tex`, `.bib`) pas les fichiers finaux (`.pdf`) ni les fichiers intermédiaires (`.aux`, `.log`, etc.)
- On crée un fichier .gitignore dans le dépôt

Quelques règles pour un bon *commit*

- Un *commit* doit disposer d'un message clair et explicite le résumant
- Il doit correspondre à un modification unitaire
  - On ne mélangera pas relectures orthographiques, relectures de fond et corrections de macro dans un même commit
  - Le cas échéant, il est possible de commiter partiellement une modification (`git commit -p`)
- En règle générale il ne faut versionner que les fichiers sources (`.tex`, `.bib`) pas les fichiers finaux (`.pdf`) ni les fichiers intermédiaires (`.aux`, `.log`, etc.)
- On crée un fichier .gitignore dans le dépôt
- `https://github.com/github/gitignore/blob/master/TeX.gitignore`

Commiter par la pratique

- Dire quels fichiers on veut commiter

Commiter par la pratique

- Dire quels fichiers on veut commiter
- Puis commiter

Commiter par la pratique

- Dire quels fichiers on veut commiter
- Puis commiter

Code 3 : Syntaxe de base d'un premier commit

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Commiter par la pratique

- Dire quels fichiers on veut commiter
- Puis commiter

Code 3 : Syntaxe de base d'un premier commit

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Code 4 : Exemple de premier commit

```
% git add .gitignore
% git commit -m "Nous allons commencer un projet en LaTeX. Gitignore de
↪ base"
```

Commiter par la pratique

Un oubli fréquent

Ne pas oublier le `git add <fichier>`

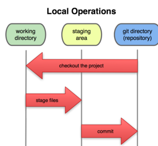


Fig. : 3 états dans Git (Schéma par Scott Chacon, sous licence CC-by-nc-sa 3.0)

Commiter par la pratique

Un oubli fréquent

Ne pas oublier le `git add <fichier>`

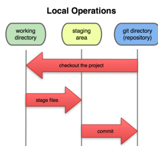


FIG. : 3 états dans Git (Schéma par Scott Chacon, sous licence CC-by-nc-sa 3.0)

Pour aller plus loin

Un raccourci possible `git commit -a -m "message"`
Mais ce raccourci ne fonctionne que pour les fichiers déjà versionnés.

Commiter par la pratique

Un oubli fréquent

Ne pas oublier le `git add <fichier>`

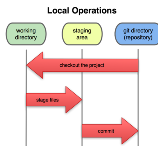


FIG. : 3 états dans Git (Schéma par Scott Chacon, sous licence CC-by-nc-sa 3.0)

Pour aller plus loin

Un raccourci possible `git commit -a -m "message"`
Mais ce raccourci ne fonctionne que pour les fichiers déjà versionnés.

Une petite précision

Un commit peut bien sûr concerner plus d'un fichier.

Quelques commandes utiles

Code 5 : Corriger / compléter le dernier commit

```
% git commit --amend -m  "<message de commit>"
```

Quelques commandes utiles

Code 5 : Corriger / compléter le dernier commit

```
% git commit --amend -m  "<message de commit>"
```

Code 6 : Déplacer ou renommer un fichier

```
% git mv <fichier source> <fichier destination>
% git commit -m "Renommage du fichier <completer le message>"
```

Quelques commandes utiles

Code 5 : Corriger / compléter le dernier commit

```
% git commit --amend -m  "<message de commit>"
```

Code 6 : Déplacer ou renommer un fichier

```
% git mv <fichier source> <fichier destination>
% git commit -m "Renommage du fichier <completer le message>"
```

Code 7 : Effacer un fichier

```
% git rm <fichier à effacer dans les futurs commits>
% git commit -m "Nous n'avons plus besoin du fichier <completer le
↪  message>"
```

Quelques commandes utiles

Code 8 : Voir les différences entre l'état actuel du projet et le dernier *commit*

```
% git diff
```

Quelques commandes utiles

Code 8 : Voir les différences entre l'état actuel du projet et le dernier *commit*

```
% git diff
```

Code 9 : Annuler les modifications d'un fichier

```
% git checkout <nom du fichier>
```

Quelques commandes utiles

Code 8 : Voir les différences entre l'état actuel du projet et le dernier *commit*

```
% git diff
```

Code 9 : Annuler les modifications d'un fichier

```
% git checkout <nom du fichier>
```

Code 10 : Retrouver l'état des fichiers à un commit précis

```
% git checkout <hash du commit>
% git checkout master
```

Tirer parti de l'historique

Code 11 : Afficher l'historique

```
% git log
```

Tirer parti de l'historique

Code 11 : Afficher l'historique

```
% git log
```

Code 12 : Annuler un commit précis

```
% git revert <hash du commit>
```

Tirer parti de l'historique

Code 11 : Afficher l'historique

```
% git log
```

Code 12 : Annuler un commit précis

```
% git revert <hash du commit>
```

Un petit piège

Lorsqu'on fait un `git revert`, Git propose de modifier le message du *commit* d'annulation.
Il ouvre pour cela l'éditeur *Vim* (on peut configurer pour avoir un autre éditeur).
Taper `:wq` dans l'éditeur pour enregistrer le fichier et garder le message par défaut du commit.

Tirer parti de l'historique

Code 13 : Marquer une étape du projet

```
% git tag <nom du tag>
```

Tirer parti de l'historique

Code 13 : Marquer une étape du projet

```
% git tag <nom du tag>
```

Code 14 : Marquer une étape du projet - exemple

```
% git tag depot_final
```

Section 3

Travailler à plusieurs

Deux modalités de travail

- La souplesse de *Git* permet une multitude de modalités de travail collaboratif

Deux modalités de travail

- La souplesse de *Git* permet une multitude de modalités de travail collaboratif
- Pour le présent exposé, nous présenterons deux modes simples et courants :

Deux modalités de travail

- La souplesse de *Git* permet une multitude de modalités de travail collaboratif
- Pour le présent exposé, nous présenterons deux modes simples et courants :
  - Tout le monde sur un pied d'égalité travaille sur la même branche

Deux modalités de travail

- La souplesse de *Git* permet une multitude de modalités de travail collaboratif
- Pour le présent exposé, nous présenterons deux modes simples et courants :
  - Tout le monde sur un pied d'égalité travaille sur la même branche
  - Des mainteneur·euse·s reçoivent des propositions de modification de la part de contributeur·trice·s, en utilisant plusieurs branches

Deux modalités de travail

- La souplesse de *Git* permet une multitude de modalités de travail collaboratif
- Pour le présent exposé, nous présenterons deux modes simples et courants :
  - Tout le monde sur un pied d'égalité travaille sur la même branche
  - Des mainteneur·euse·s reçoivent des propositions de modification de la part de contributeur·trice·s, en utilisant plusieurs branches

Précision technique

La gestion des droits d'accès ne dépend pas directement de *Git* mais bien de la *forge logicielle*.
Ici nous fonctionnerons avec *Gitlab*. Pour *Github*, quelques adaptations seront nécessaires, mais les principes globaux restent les mêmes.

Une difficulté majeure

- La principale difficulté avec le travail collaboratif résulte de la gestion des conflits, si deux personnes travaillent sur le même fichier.

Une difficulté majeure

- La principale difficulté avec le travail collaboratif résulte de la gestion des conflits, si deux personnes travaillent sur le même fichier.
- Par conflit, nous entendons le fait que deux personnes peuvent modifier en parallèle les mêmes lignes d'un fichier.

Une difficulté majeure

- La principale difficulté avec le travail collaboratif résulte de la gestion des conflits, si deux personnes travaillent sur le même fichier.
- Par conflit, nous entendons le fait que deux personnes peuvent modifier en parallèle les mêmes lignes d'un fichier.
- Pour limiter les risques il est important, avant toute session de travail de récupérer (`pull`) l'état des fichiers distants.

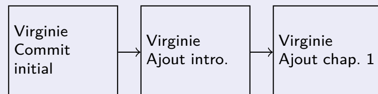
Une difficulté majeure

- La principale difficulté avec le travail collaboratif résulte de la gestion des conflits, si deux personnes travaillent sur le même fichier.
- Par conflit, nous entendons le fait que deux personnes peuvent modifier en parallèle les mêmes lignes d'un fichier.
- Pour limiter les risques il est important, avant toute session de travail de récupérer (`pull`) l'état des fichiers distants.
- Dans certains cas toutefois, des conflits resteront. Il faudra alors les résoudre manuellement en choisissant quelle version on retient.

Fusion *vs* rebasage

La fusion et le rebasage sont deux manières différentes de gérer des divergences dans l'historique.

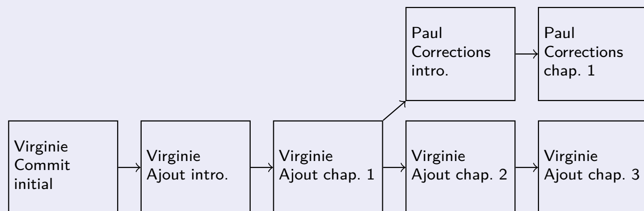
Étape 1 : Virginie travaille



Fusion *vs* rebasage

La fusion et le rebasage sont deux manières différentes de gérer des divergences dans l'historique.

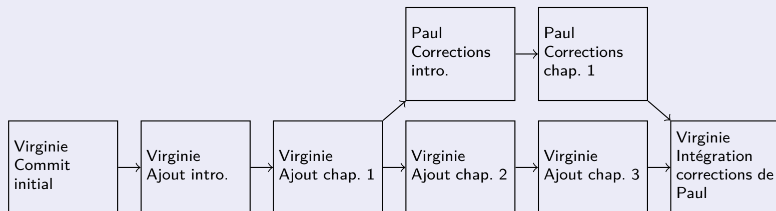
Étape 2 : Paul et Virginie travaillent en parallèle



Fusion *vs* rebasage

La fusion et le rebasage sont deux manières différentes de gérer des divergences dans l'historique.

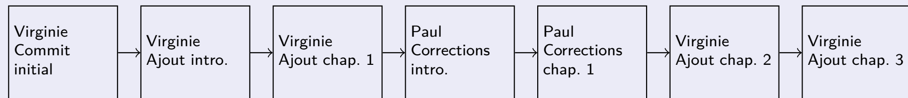
Étape 3 - Solution a. : Virginie fusionnent les branches



Fusion *vs* rebasage

La fusion et le rebasage sont deux manières différentes de gérer des divergences dans l'historique.

Étape 3 - Solution b. : Virginie rebase le travail de Paul



Fusion *vs* rebasage

- On rebase lorsqu'on récupère veut simplement s'actualiser par rapport à un dépôt distant

Fusion *vs* rebasage

- On rebase lorsqu'on récupère veut simplement s'actualiser par rapport à un dépôt distant
- On fusionne lorsqu'on veut garder trace du travail parallèle notamment lorsque

Fusion *vs* rebasage

- On rebase lorsqu'on récupère veut simplement s'actualiser par rapport à un dépôt distant
- On fusionne lorsqu'on veut garder trace du travail parallèle notamment lorsque
  - On suit le modèle de relecture du travail des autres contributeur·trice·s

Fusion *vs* rebasage

- On rebase lorsqu'on récupère veut simplement s'actualiser par rapport à un dépôt distant
- On fusionne lorsqu'on veut garder trace du travail parallèle notamment lorsque
  - On suit le modèle de relecture du travail des autres contributeur·trice·s
  - On crée des branches de développement pour de nouvelles fonctionnalités sur une classe ou un package

Fusion *vs* rebasage

Remarques

Il s'agit de bonnes pratiques pas d'obligation. Certains projets préfèrent le rebasage systématique. Plus rarement la fusion systématique.
`https://delicious-insights.com/fr/articles/bien-utiliser-git-merge-et-rebase/`
Pour faciliter la tâche, nous allons configurer git pour faire le bon choix par défaut.

Fusion *vs* rebasage

Remarques

Il s'agit de bonnes pratiques pas d'obligation. Certains projets préfèrent le rebasage systématique. Plus rarement la fusion systématique.
https://delicious-insights.com/fr/articles/bien-utiliser-git-merge-et-rebase/
Pour faciliter la tâche, nous allons configurer git pour faire le bon choix par défaut.

Code 15 : Configuration de *Git* pour avoir une bonne politique de gestion de l'historique

```
% git config --global pull.ff only
% git config --global pull.rebase preserve
% git config --global merge.ff false
```

Fusion *vs* rebasage

Remarques

Il s'agit de bonnes pratiques pas d'obligation. Certains projets préfèrent le rebasage systématique. Plus rarement la fusion systématique.
`https://delicious-insights.com/fr/articles/bien-utiliser-git-merge-et-rebase/`
Pour faciliter la tâche, nous allons configurer git pour faire le bon choix par défaut.

Code 15 : Configuration de *Git* pour avoir une bonne politique de gestion de l'historique

```
% git config --global pull.ff only
% git config --global pull.rebase preserve
% git config --global merge.ff false
```

Sur l'historique

Rebaser revient à réécrire l'historique. Les commits rebasés changent de hash et date. Mais auteur, contenu et message restent identiques.

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
  - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
  - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »
  - Configurer le nom et la visibilité du projet

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
  - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »
  - Configurer le nom et la visibilité du projet
  - Créer un `README.md` minimal (cela crée un premier commit)

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
    - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »
    - Configurer le nom et la visibilité du projet
    - Créer un `README.md` minimal (cela crée un premier commit)
    - Puis pour le projet, aller dans « Paramètres / Membres » / « Parameters / members » et donner les droits de « Mainteneur » / « Maintener »

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
  - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »
  - Configurer le nom et la visibilité du projet
  - Créer un `README.md` minimal (cela crée un premier commit)
  - Puis pour le projet, aller dans « Paramètres / Membres » / « Parameters / members » et donner les droits de « Mainteneur » / « Maintener »
  - Une autre solution, plus propre mais plus complexe, est de créer un groupe

Fonctionnement monobranche par la pratique : création et configuration du dépôt distant

- Dans *Framagit* / *Gitlab* créer un dépôt commun et donner les droits aux personnes concernées
  - Sur la page d'accueil, une fois connecté, choisir « Nouveau projet » / « New project »
  - Configurer le nom et la visibilité du projet
  - Créer un `README.md` minimal (cela crée un premier commit)
  - Puis pour le projet, aller dans « Paramètres / Membres » / « Parameters / members » et donner les droits de « Mainteneur » / « Maintener »
  - Une autre solution, plus propre mais plus complexe, est de créer un groupe
- Sur la page du projet, le bouton clone permet de connaitre l'url du projet distant à cloner

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 16 : Clonage initial du dépôt distant

```
% git clone <url du dépot distant>
```

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 16 : Clonage initial du dépôt distant

```
% git clone <url du dépot distant>
```

À propos de l'identification

Pour communiquer avec un dépôt distant, il faut la plupart du temps s'identifier. La méthode la plus basique est celle du login/mot de passe. Elle est utilisée lorsque l'url commence par `https ://`.
Pour éviter de taper systématiquement son mot de passe, deux solutions :

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 16 : Clonage initial du dépôt distant

```
% git clone <url du dépot distant>
```

À propos de l'identification

Pour communiquer avec un dépôt distant, il faut la plupart du temps s'identifier. La méthode la plus basique est celle du login/mot de passe. Elle est utilisée lorsque l'url commence par `https ://`.

Pour éviter de taper systématiquement son mot de passe, deux solutions :

- Tirer profit du système `credential` de Git `https://git-scm.com/book/fr/v2/Utilitaires-Git-Stockage-des-identifiants`

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 16 : Clonage initial du dépôt distant

```
% git clone <url du dépot distant>
```

À propos de l'identification

Pour communiquer avec un dépôt distant, il faut la plupart du temps s'identifier. La méthode la plus basique est celle du login/mot de passe. Elle est utilisée lorsque l'url commence par `https ://`.

Pour éviter de taper systématiquement son mot de passe, deux solutions :

- Tirer profit du système `credential` de Git `https://git-scm.com/book/fr/v2/Utilitaires-Git-Stockage-des-identifiants`
- Plus sécurisé, utiliser une clé `ssh` (dans ce cas, l'url du dépôt distant est différente) `https://git-scm.com/book/fr/v2/Git-sur-le-serveur-Génération-des-clés-publiques-SSH`

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 17 : Faire autant de *commits* que nécessaire

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 17 : Faire autant de *commits* que nécessaire

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Code 18 : Envoyer sur le serveur distant

```
% git push
```

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 17 : Faire autant de *commits* que nécessaire

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Code 18 : Envoyer sur le serveur distant

```
% git push
```

Code 19 : Récupérer du serveur distant

```
% git pull
```

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Code 17 : Faire autant de *commits* que nécessaire

```
% git add <nom du fichier>
% git commit -m "<message de commit>"
```

Code 18 : Envoyer sur le serveur distant

```
% git push
```

Code 19 : Récupérer du serveur distant

```
% git pull
```

Code 20 : Envoyer les tags sur le serveur distant

```
% git push --tags
```

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés
    - Repérer les lignes encadrées par >>>> et <<<< : ce sont les lignes qui posent problème

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés
    - Repérer les lignes encadrées par >>>> et <<<< : ce sont les lignes qui posent problème
    - Choisir la version à retenir

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés
    - Repérer les lignes encadrées par >>>> et <<<< : ce sont les lignes qui posent problème
    - Choisir la version à retenir
    - Il existe des outils de gestion des conflits qui permettent de comparer les versions et de choisir la bonne, plutôt que de modifier « à la main » les fichiers en conflits

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés
    - Repérer les lignes encadrées par >>>> et <<<< : ce sont les lignes qui posent problème
    - Choisir la version à retenir
    - Il existe des outils de gestion des conflits qui permettent de comparer les versions et de choisir la bonne, plutôt que de modifier « à la main » les fichiers en conflits
    - Puis commiter

Fonctionnement monobranche par la pratique : travail sur les dépôts locaux et synchronisation

Sur les divergences d'historique

- Pour limiter les risques, toujours faire un `git pull` en début de session de travail
- Si quelqu'un a pushé sur le serveur distant depuis votre dernier `pull`, nous ne pourrons pas pusher :
  - Faire un `git pull`
  - Avec la config que nous avons effectuée plus haut, les modifications distantes seront rebasées avant vos modifications locales
  - Si les mêmes lignes ont été modifiées en local et à distance :
    - Il faudra modifier les fichiers concernés
    - Repérer les lignes encadrées par >>>> et <<<< : ce sont les lignes qui posent problème
    - Choisir la version à retenir
    - Il existe des outils de gestion des conflits qui permettent de comparer les versions et de choisir la bonne, plutôt que de modifier « à la main » les fichiers en conflits
    - Puis commiter
  - Puis `git push`

Fonctionnement multibranche par la pratique : configuration

- Configurer le dépôt distant comme pour le travail monobranche, mais donner seulement les droits de « Développeur » / « Developer » aux personnes qui ne peuvent pas travailler directement sur la branche principale

Fonctionnement multibranche par la pratique : configuration

- Configurer le dépôt distant comme pour le travail monobranche, mais donner seulement les droits de « Développeur » / « Developer » aux personnes qui ne peuvent pas travailler directement sur la branche principale
- Puis cloner le dépôt distant en local

Fonctionnement multibranche par la pratique : création et bascule de branche

Noms et politique de branche

La branche par défaut est la branche master.
Il est conseillé de créer une branche par bloc fonctionnel.
On pourra aussi créer des branches pour préparer la sortie d'une nouvelle version.
Normalement la branche master doit toujours être opérationnelle.

Fonctionnement multibranche par la pratique : création et bascule de branche

Noms et politique de branche

La branche par défaut est la branche master.
Il est conseillé de créer une branche par bloc fonctionnel.
On pourra aussi créer des branches pour préparer la sortie d'une nouvelle version.
Normalement la branche master doit toujours être opérationnelle.

Code 21 : Créer une nouvelle branche et y basculer

```
% git checkout -b <nom de la nouvelle branche>
```

Fonctionnement multibranche par la pratique : création et bascule de branche

Noms et politique de branche

La branche par défaut est la branche master.
Il est conseillé de créer une branche par bloc fonctionnel.
On pourra aussi créer des branches pour préparer la sortie d'une nouvelle version.
Normalement la branche master doit toujours être opérationnelle.

Code 21 : Créer une nouvelle branche et y basculer

```
% git checkout -b <nom de la nouvelle branche>
```

Code 22 : Lister les branches

```
% git branch
```

Fonctionnement multibranche par la pratique : création et bascule de branche

Noms et politique de branche

La branche par défaut est la branche master.
Il est conseillé de créer une branche par bloc fonctionnel.
On pourra aussi créer des branches pour préparer la sortie d'une nouvelle version.
Normalement la branche master doit toujours être opérationnelle.

Code 21 : Créer une nouvelle branche et y basculer

```
% git checkout -b <nom de la nouvelle branche>
```

Code 22 : Lister les branches

```
% git branch
```

Code 23 : Basculer vers une branche

```
% git checkout <nom de la branche>
```

Fonctionnement multibranche par la pratique : gestion des branches distantes

Code 24 : Envoyer toutes les branches locales sur le dépôt distant

```
% git push --all
```

Fonctionnement multibranche par la pratique : gestion des branches distantes

Code 24 : Envoyer toutes les branches locales sur le dépôt distant

```
% git push --all
```

Code 25 : Envoyer une branche locale sur le dépôt distant

```
% git push origin <nom de la branche>
```

Fonctionnement multibranche par la pratique : gestion des branches distantes

Code 24 : Envoyer toutes les branches locales sur le dépôt distant

```
% git push --all
```

Code 25 : Envoyer une branche locale sur le dépôt distant

```
% git push origin <nom de la branche>
```

Sur origin

origin est le nom du dépôt distant automatiquement créé lors d'un `git clone`.
Il est possible d'avoir plusieurs dépôts distants, mais nous n'en parlerons pas ici.

Fonctionnement multibranche par la pratique : gestion des branches distantes

Code 26 : Travailler pour la première fois sur une branche distante

```
% git fetch origin
% git checkout --track origin/<nom de la branche distante>
```

Fonctionnement multibranche par la pratique : gestion des branches distantes

Code 26 : Travailler pour la première fois sur une branche distante

```
% git fetch origin
% git checkout --track origin/<nom de la branche distante>
```

Code 27 : Récupérer le dernier état de la branche distante

```
% git checkout <branche distante>
% git pull
```

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées
  - Le cas échéant, faire des commentaires / poser des questions

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées
  - Le cas échéant, faire des commentaires / poser des questions
  - Et même modifier la branche distante en y apportant des commits

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées
  - Le cas échéant, faire des commentaires / poser des questions
  - Et même modifier la branche distante en y apportant des commits
  - Puis accepter la fusion

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées
  - Le cas échéant, faire des commentaires / poser des questions
  - Et même modifier la branche distante en y apportant des commits
  - Puis accepter la fusion
- On peut aussi fusionner les branches en local

Fonctionnement multibranche par la pratique : fusion de branches

- Via *Gitlab* la personne qui propose une fonctionnalité fait une demande de fusion de branche :
  - Se rendre sur le dépôt
  - Choisir dans le menu déroulant la branche à fusionner
  - Puis « Créer une demander de fusion » / « Create a merge request »
  - Ou bien, suivre le lien qui apparaît lorsqu'on fait un `git pull`
- Les mainteneur·euse·s du projet peuvent, à travers l'interface de *Gitlab* :
  - Voir les modifications proposées
  - Le cas échéant, faire des commentaires / poser des questions
  - Et même modifier la branche distante en y apportant des commits
  - Puis accepter la fusion
- On peut aussi fusionner les branches en local

Code 28 : Fusionner une branche en local

```
% git checkout <nom de la branche qui recevra la fusion>
% git merge <nom de la branche à fusionner>
```

Fonctionnement multibranche par la pratique : suppression de branche

Code 29 : Supprimer une branche locale

```
% git branch -d <nom de la branche>
```

Fonctionnement multibranche par la pratique : suppression de branche

Code 29 : Supprimer une branche locale

```
% git branch -d <nom de la branche>
```

Code 30 : Supprimer une branche distante

```
% git push origin :<nom de la branche>
```

Fonctionnement multibranche par la pratique : suppression de branche

Code 29 : Supprimer une branche locale

```
% git branch -d <nom de la branche>
```

Code 30 : Supprimer une branche distante

```
% git push origin :<nom de la branche>
```

Code 31 : Supprimer une branche distante (syntaxe alternative)

```
% git push -d origin <nom de la branche>
```

Section 4

Pour aller plus loin

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail
- Le rebasage interactif (`git rebase -i`) pour « nettoyer » son historique avant un `git push`

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail
- Le rebasage interactif (`git rebase -i`) pour « nettoyer » son historique avant un `git push`
- Le cherry-pick (`git cherry-pick`) pour reproduire un commit d'une branche à une autre

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail
- Le rebasage interactif (`git rebase -i`) pour « nettoyer » son historique avant un `git push`
- Le cherry-pick (`git cherry-pick`) pour reproduire un commit d'une branche à une autre
- Le test par bisection (`git bisect`) pour trouver quel *commit* a introduit un bug

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail
- Le rebasage interactif (`git rebase -i`) pour « nettoyer » son historique avant un `git push`
- Le cherry-pick (`git cherry-pick`) pour reproduire un commit d'une branche à une autre
- Le test par bisection (`git bisect`) pour trouver quel *commit* a introduit un bug
- Le blame (`git blame`) pour trouver quel commit a modifié une ligne

Autres fonctions utiles

- Les alias pour avoir des raccourcis de commande
- Le prompt coloré et autocomplétant pour y voir plus clair
- La remise (`git stash`) pour mettre temporairement de côté un travail
- Le rebasage interactif (`git rebase -i`) pour « nettoyer » son historique avant un `git push`
- Le cherry-pick (`git cherry-pick`) pour reproduire un commit d'une branche à une autre
- Le test par bisection (`git bisect`) pour trouver quel *commit* a introduit un bug
- Le blame (`git blame`) pour trouver quel commit a modifié une ligne
- Les hooks pour faire des tests / exécuter du code lors de certaines actions

Ressources en lignes

- Chacon, Scott et Ben Straub, *Pro Git*, url : https://git-scm.com/book/fr/v2 (visité le 02/06/2019), Manuel de référence, pour tout comprendre à Git.
- Morin, Cédric, *Git c'est facile*, url : https://www.yterium.net/Git-c-est-facile (visité le 02/06/2019), Quelques réglages utiles à mettre en place.
- Delicious Insights, *Bien utiliser merge et rebase*, url : https://delicious-insights.com/fr/articles/bien-utiliser-git-merge-et-rebase/ (visité le 02/06/2019), Pour avoir un historique propre quand on travaille à plusieurs.
- Rouquette, Maïeul, *Geekographie Maïeulesque. Git*, url : https://geekographie.maieul.net/Git (visité le 02/06/2019), Quelques élèments que j'ai écrits sur Git.