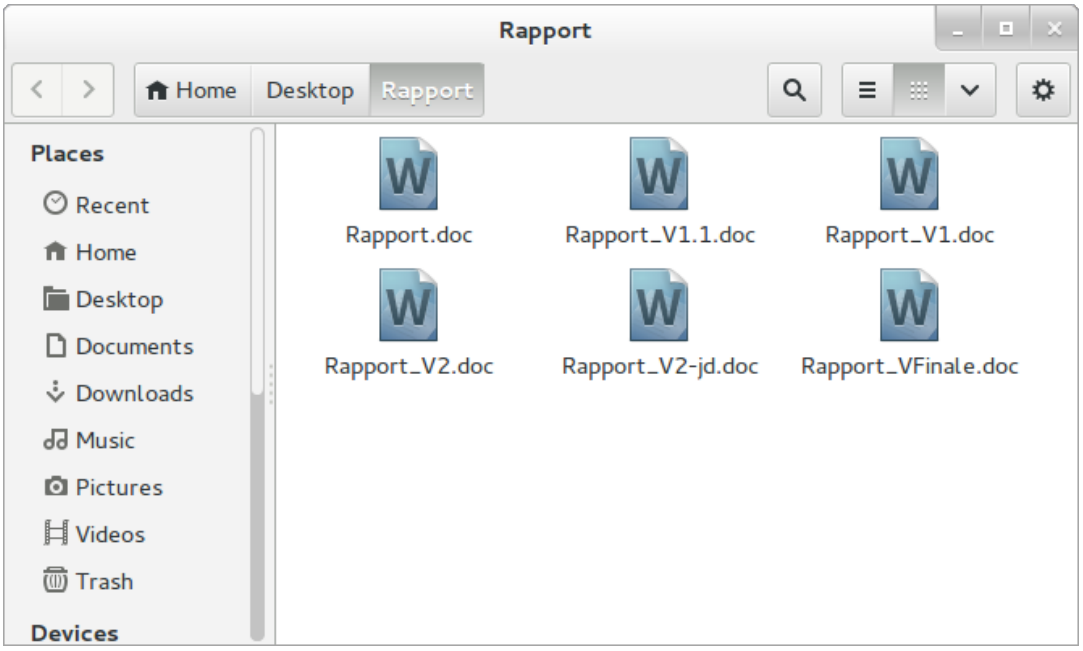
Introduction à GIT ¶ §

Département Informatique (IUT Lyon 1)

Motivations

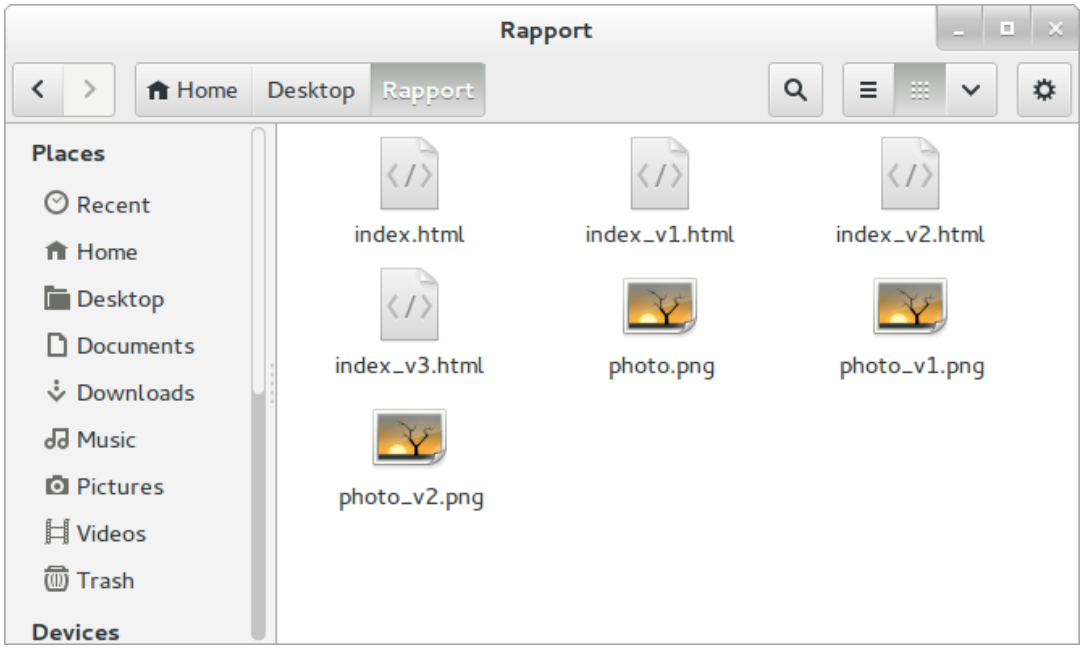
Exemple 1



Note:

- la version la plus à jour est-elle `Rapport.doc` ou `Rapport_VFinale.doc` ?
- et si on avait aussi `Rapport_VFinale1.doc` et `Rapport_VFinale2.doc` (expérience vécue) ?
- les versions n'apparaissent pas dans l'ordre (1.1, 1, 2)
- la version 2-jd vient elle avant ou après la version 2 ?

Exemple 2



✎ **Note:**

- les versions de l'image sont elles numérotées indépendamment, ou par rapport aux versions de la page ?
- nécessité de renommer les fichiers pour visualiser une ancienne version (pour que les liens fonctionnent)

Histoires vraies

- Après que nous avons échangé avec un collègue des versions d'un fichier nommées `X_1.1.doc`, `X_1.2.doc`, `X_1.3.doc` (et ainsi de suite),

  il a nommé la version finale `X_1.0.doc` ...

- Un autre collègue m'a envoyé, le 15 mars 2013, un fichier nommé `2013-03-17-xxx` .

  Je l'ai modifié le 16 mars ; quel nom lui donner ?...

Conclusion

- La gestion des versions est un travail fastidieux et méthodique.
- Les humains ne sont pas doués pour les travaux fastidieux et méthodiques.
- Laissons cela à l'ordinateur,
  - et concentrons-nous sur la partie du travail où nous sommes meilleurs que l'ordinateur.

→ VCS (*Version Control System*)

Avertissement

- GIT (se prononce « guite ») est un outil extrêmement riche ;
  - nous n'en verrons qu'une partie dans ce module.
- Ne vous laissez pas effrayer par l'interface « touffue » ;
  - faites confiances aux réglages par défaut (dans un premier temps).

Historique des VCS

Origines

- Initialement dédiés à la gestion de code source pour les projets logiciels
- mais également :
  - documentation
  - site web
- travail collaboratif :
  - facilité d'échange
  - traçabilité
  - gestion des conflits

Évolutions

Systèmes centralisés

- CVS (Concurrent Versioning System, vieillissant)
- SVN (Subversion, très populaire, mais c'est en train de changer)

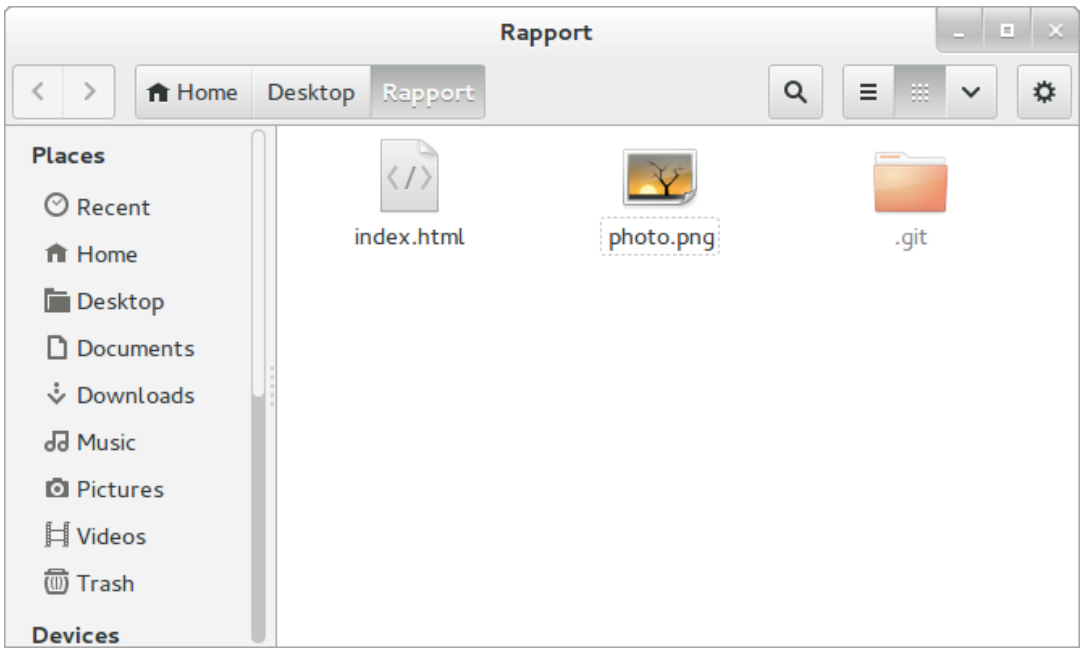
Systèmes décentralisés

- GIT
- Mercurial (Hg)
- Bazaar (bzr)

→ facilitent une utilisation individuelle

Notions de base

Bon exemple



> **Note:** Le répertoire `.git` est un répertoire caché, qui contient tout l'historique des fichiers.

Les avantages de la gestion de versions

- Sauvegarde (modulo la synchronisation avec un serveur distant)
- Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?)
- Possibilité de retour en arrière
- Fusion des modifications lors du travail collaboratif
- Visualiser les changements au cours du temps

Notions

- Dépôt (*repository*)
- Commit
- Copie de travail
- Index

Dépôt (*repository*)

Le répertoire caché `.git` est nommé **dépôt** (en anglais *repository*).

Il contient toutes les données dont GIT a besoin pour gérer l'historique. Sauf rarissimes exceptions, vous ne mo-

difierez jamais son contenu directement, mais uniquement en passant par les commandes GIT.

Commit

L'historique d'un projet est une séquence de « photos », contenant l'état de tous les fichiers du projet.

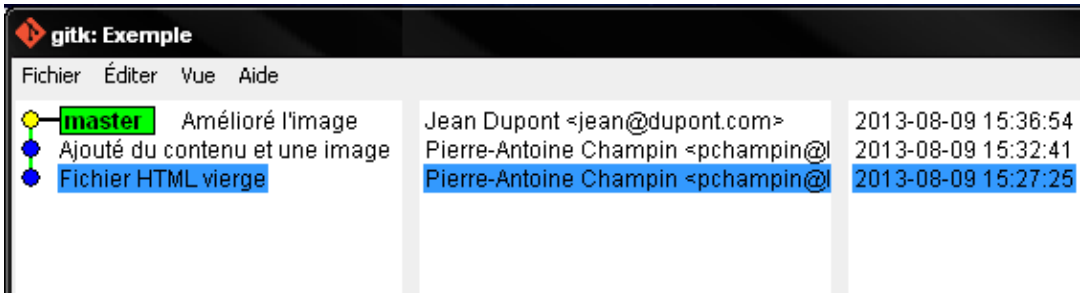
Ces « photos » s'appellent des **commits**, et possèdent :

- une date
- un auteur
- une description textuelle
- un lien vers le(s) commit(s) précédent(s)

On parle également de **révision**.

> ✎ **Note:** GIT fait une distinction subtile entre *revision*, *commit* et *commit object* mais elle n'est pas utile ici.

Illustration



Visualisation d'un historique simple dans un outil graphique.

NB : on ignore pour l'instant le rectangle `master` ; on l'expliquera par la suite.

Remarque

En pratique, GIT ne stocke pas la totalité des fichiers pour chaque commit, mais seulement les différences par rapport à l'état suivant.

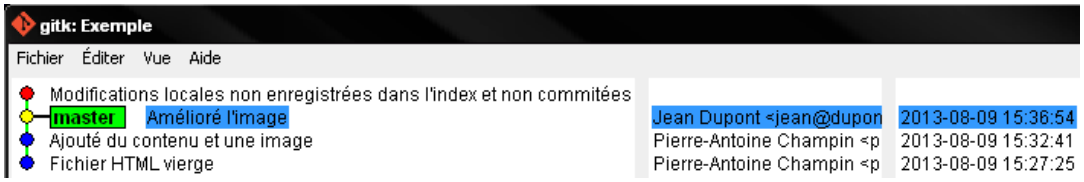
Avantage par rapport à l'approche manuelle : moins coûteux en place.

NB : bien que GIT (et les autres VCS) soient plus particulièrement conçus pour des fichiers texte, ils fonctionnent aussi avec des fichiers binaires (images, bureautique, etc.).

Copie de travail

On appelle **copie de travail** (en anglais *working copy*) les fichiers effectivement présents dans le répertoire géré par GIT.

Leur état peut être différent du dernier commit de l'historique.

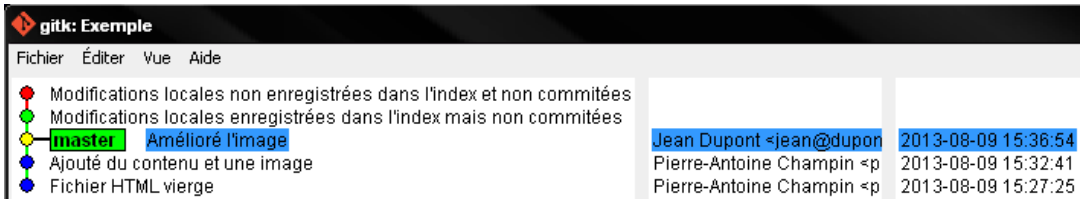


Index

L'index est un espace temporaire contenant les modifications prêtes à être « commitées ».

Ces modifications peuvent être :

- création de fichier
- modification de fichier
- suppression de fichier



✎ **Note:** On remarque le code couleur de l'application `gitk` :

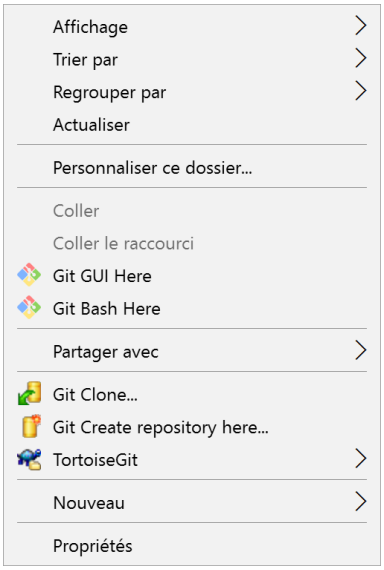
- rouge : changements non indexés, non commités
- vert : changements indexés, non commités
- jaune : commit courant (HEAD)
- bleu : autres commits

NB: la sémantique du rouge et du vert est la même pour la ligne de commande. Cependant, certains outils GIT ne respectent pas ce code couleur; c'est notamment le cas de TortoiseGit que nous allons utiliser par la suite.

Mise en œuvre

Dans ce cours, nous considérerons deux méthodes possibles :

- Ligne de commande (Git Bash)
- Interface graphique (TortoiseGit)



✎ **Note:** Il existe d'autres interfaces graphiques pour GIT, notamment `Git Gui` qui est installé sur les machines de l'IUT, mais que nous ne présenterons pas dans ce cours.

De plus, la plupart des IDE fournissent un accès aux commendes GIT, mais nous ne les traiterons pas non plus.

Création du dépôt

Initialise la gestion de version dans un répertoire en créant le sous-répertoire `.git`.

Mise en œuvre

- Dans le menu contextuel du répertoire concerné, *Git Create repository here…*
- En ligne de commande, depuis le répertoire concerné:

```
$ git init
```

Commiter des modifications

Une fois les fichiers modifiés et dans un état satisfaisant, vous pouvez les commiter.

Remarque : lorsque vous effectuez un commit, il est essentiel d'écrire un message accompagnant le commit. Ce message doit être informatif quant à la nature des modifications que vous êtes en train de commiter.

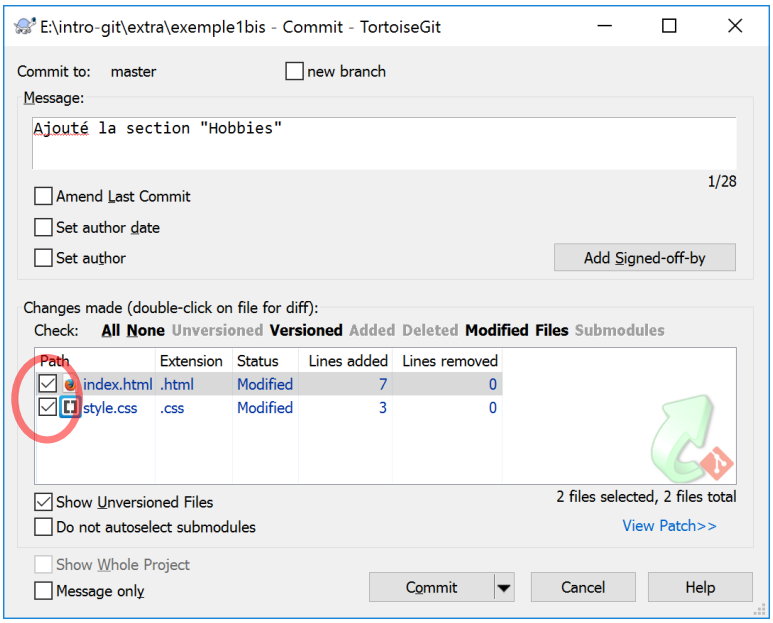
Par exemple, *blip* est un **mauvais** message de commit, mais *Correction des fautes d'orthographe dans la doc technique* est un **bon** message de commit.

✎ **Note:** En cas de problème, il est possible de corriger un commit (tant qu'il n'a pas été partagé avec

d'autres collaborateurs), mais nous étudierons cela plus tard.

Depuis l'interface graphique

Permet d'ajouter ou retirer des fichiers dans l'index avant le commit



En ligne de commande

Ajouter un fichier dans l'index

```
$ git add <filename>
```

Retirer un fichier de l'index

```
$ git reset <filename>
```

Pour voir l'état des modifications en cours

```
$ git status    # résumé
$ git diff      # détail des changements
```

Pour commiter les modifications indexées

```
$ git commit     #ou
$ git commit -m "message de commit"
```

Consulter l'historique

- Menu contextuel > *TortoiseGit* > *Show log*

  (cf. figure suivante)

- En ligne de commande :

  - afficher la liste des commits

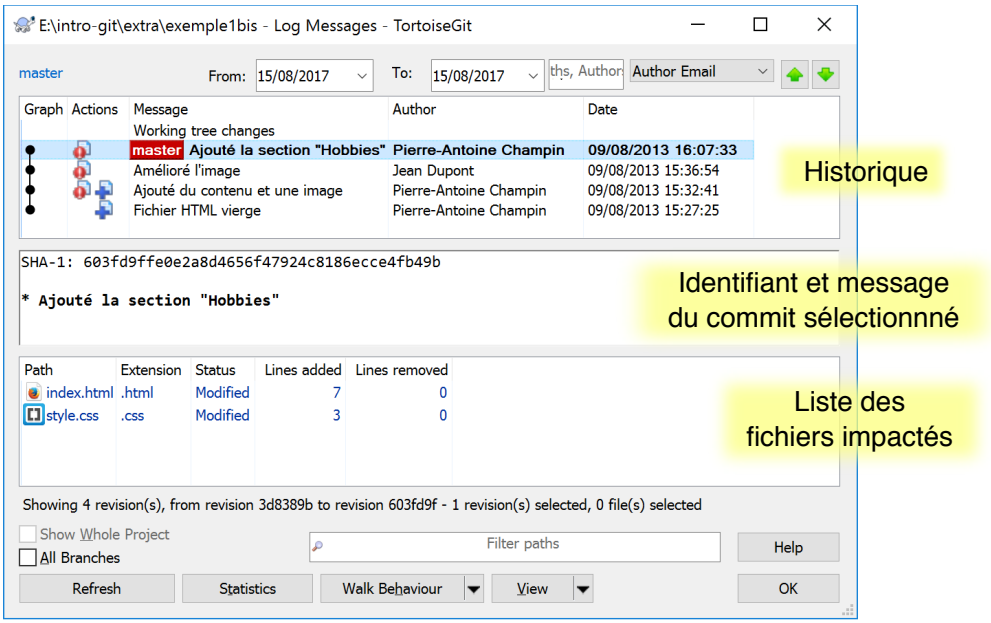
    ```
    $ git log
    ```

    (avec l'identifiant de chaque commit)

  - afficher le détail d'un commit particulier

    ```
    $ git show <id-commit>
    ```

Depuis l'interface graphique



Résumé des états possibles d'un fichier avec GIT

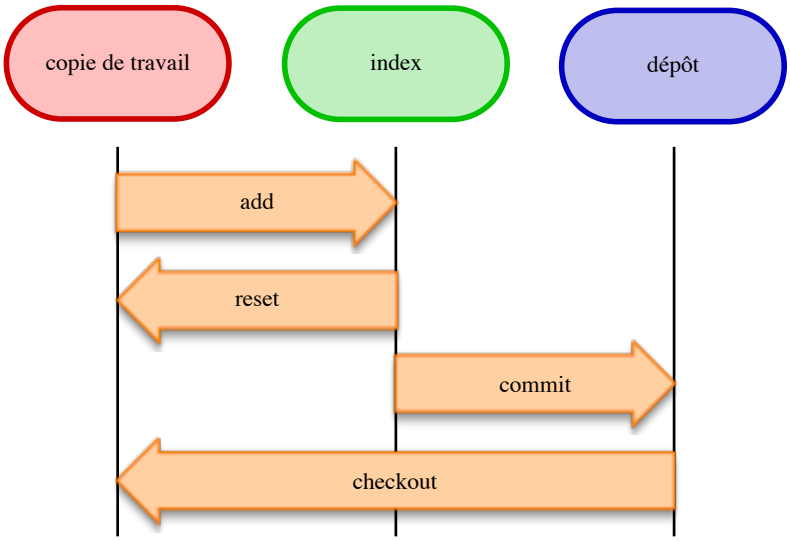


Figure inspirée de git-scm.org.

Exercice

1. Créez un nouveau répertoire, et faites-en un dépôt GIT.
2. Ajoutez un fichier HTML dans ce répertoire, contenant une brève description de vous (ou de n'importe quel autre sujet qui vous intéresse). Commitez ces changements.
3. Ajoutez maintenant une feuille de style *externe* à votre fichier HTML, (c'est à dire sous la forme d'un fichier CSS séparé, référencé par une balise `<link>`). Commitez ces changements.
4. Affichez l'historique de vos changements (selon votre méthode préférée).
5. Entraînez-vous à faire des commits : modifiez le contenu du fichier HTML et/ou de la feuille de style, ajoutez éventuellement d'autres fichiers (images par exemples). À chaque changement, commitez votre travail en rédigeant un message de commit suffisament explicite.

✎ **Note:** Il faut observer le .git. S'il n'apparaît pas, veiller à configurer l'explorateur de fichiers pour qu'il affiche les fichiers et dossiers cachés.

Naviguer dans l'historique

Motivation

L'intérêt d'utiliser un VCS est de pouvoir consulter n'importe quelle version antérieure du projet.

⚠ Attention cependant :

- à ne pas avoir de modification non commitée lorsque vous commencez à naviguer dans l'historique ;
- à ne pas faire de modification sur une version ancienne.

Dans les deux cas, vous risqueriez de perdre ces modifications (GIT affiche d'ailleurs des messages d'avertissement).

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Switch/Checkout*...
    - cocher *Commit*,
    - saisir une *Expression de révision* (cf. ci-après),
    - le cas échéant, décocher *Create New Branch*,
    - valider.
- En ligne de commande :

```
$ git checkout <revision>
```

Expressions de révision

Il existe plusieurs méthodes pour spécifier une révision à GIT :

(liste non exhaustive)

Identifiant

Chaque commit a un identifiant, affiché

- par l'interface graphique (cf. figure),
- par la commande `git log`.

On peut spécifier une révision en utilisant

- l'identifiant complet du commit, ou
- les premiers caractères de l'identifiant, tant qu'il n'y a pas d'ambiguïté

Exemple :

```
$ git checkout 9a063f5fd514e966837163ceffaec332ce66fdff    # ou
```

```
$ git checkout 9a063
```

Relatif

`HEAD~` ou `HEAD~1` désignent le parent du commit courant. Ainsi :

```
$ git checkout HEAD~
```

permet de remonter d'un commit dans l'historique.

NB : `HEAD~2` remonte de deux commits, `HEAD~3` de trois commits, et ainsi de suite.

Retour au présent

- Menu contextuel > *TortoiseGit* > *Switch/Checkout*...
    - s'assurer que `Branch` est bien coché,
    - s'assurer que `master` est bien sélectionné dans la liste correspondante,
    - valider.
- En ligne de commande :

    ```
    $ git checkout master
    ```

NB : ceci est en fait un cas particulier de l'action Changer de branche que nous étudierons un peu plus tard.

Exercice

1. Naviguez dans l'historique du dépôt créé à l'exercice précédent, pour afficher l'avant-dernière version de votre fichier HTML.
2. Affichez ensuite l'antépénultième (avant-avant-dernière) version.
3. Affichez ensuite la deuxième version de votre fichier HTML.
4. Affichez ensuite la première version de votre fichier HTML.
5. Revenez au « présent » (*i.e.* la dernière version).

Entractes

Nous venons de voir les fonctionnalités les plus basiques de GIT, qui permettent de gérer ~~efficacement~~ correctement l'historique d'un ensemble de fichiers → à utiliser *sans modération*.

Dans la suite, nous allons étudier des fonctionnalités un peu plus avancées, qui seraient impraticables avec une gestion « manuelle » de l'historique.

- elle peuvent donc vous sembler superflues,
- mais s'avèrent vite indispensables quand on y a pris goût.

Branches

Motivation

Dans certaines situations, on peut souhaiter faire cohabiter et évoluer *plusieurs* versions divergentes du même projet.

Ces versions peuvent parfois converger à nouveau (mais pas forcément).

✎ **Note:** Les copies d'écran de ces exemples sont faites avec un autre logiciel que les précédentes, ce qui explique le code couleur différent.

Exemple 1 : CV

Pour un CV, on souhaite avoir :

- une version « maître » que l'on maintient à jour,
- des variantes pour chaque demande d'emploi, adaptées en fonction de l'employeur visé.

Illustration

✎ **Note:** L'historique n'a plus une structure linéaire, mais *arborescente* (ce qui justifiera la métaphore de la « branche »).

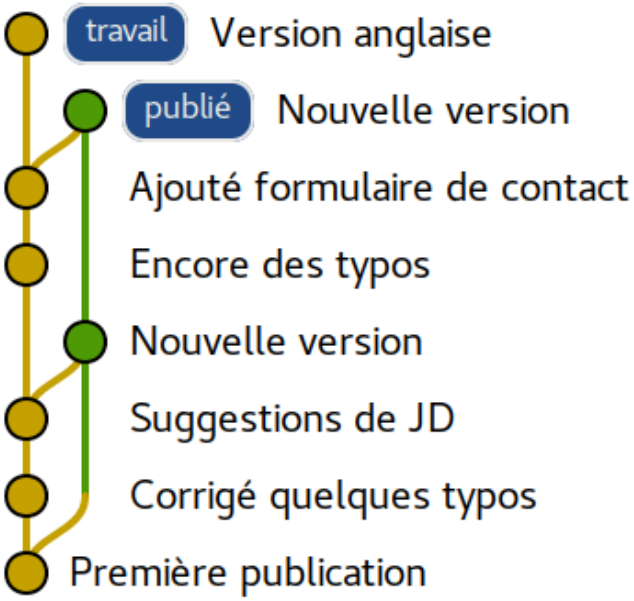
Exemple 2 : site web

Pour un site web, on souhaite avoir :

- la version publiée,
- une version de travail, dans laquelle on apporte des modifications incrémentales.

Les deux versions mènent leur existence en parallèle, la version publiée étant régulièrement mise à jour par rapport à la version de travail.

Illustration



✎ **Note:** L'historique n'est même plus un arbre, mais un graphe orienté sans cycle.

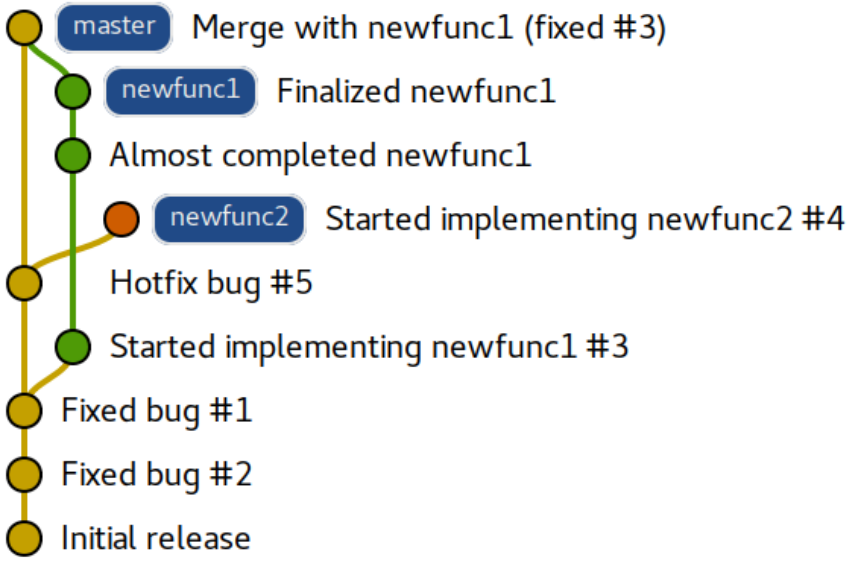
Exemple 3 : logiciel

Dans un projet logiciel, on souhaite avoir :

- la version stable, dans laquelle on se contente de corriger des bugs, et
- une ou plusieurs versions expérimentales, dans lesquelles on implémente de nouvelles fonctionnalités ;

Une fois au point, chaque nouvelle fonctionnalités est intégrée à la version stable.

Illustration



Notions

- Une **branche** est la *lignée* (généalogique) de commits, à laquelle on a donné un nom.
- Le commit le plus récent de la branche est appelé *sommet* (en anglais *tip*) de cette branche.
- La copie de travail est (en général) liée au sommet d'une branche (`master` par défaut).
- À chaque nouveau commit, le sommet de la branche courante est avancé vers ce nouveau commit (la branche « pousse »).

> ✎ **Note:** Par « lignée », on entends : l'ensemble des commits ancêtres du sommet de la branche.
>
> Dans le cas simple, cette lignée a une structure linéaire, mais ce n'est pas toujours le cas (comme en témoignent, dans les illustrations ci-avant, la branche `publié` dans l'exemple du site web et la branche `master` dans l'exemple du logiciel).

Accessibilité

Un commit est **accessible** s'il appartient à au moins une branche. Les commits non accessibles sont automatiquement supprimés par GIT.

> ✎ **Note:** Cette suppression n'est cependant pas immédiate. Il est donc parfois possible de « sauver » un commit devenu récemment inaccessible, en créant une nouvelle branche avant sa suppression effective.

Lorsqu'on navigue dans l'historique, on lie la copie de travail à un commit particulier plutôt qu'à une branche (mode *detached HEAD*).

→ Les commits que l'on ferait dans cet état n'appartiendraient à aucune branche et seraient donc perdus.

Mise en œuvre

Afficher la liste des branches

- Dans les interfaces graphiques :

  elle apparaît chaque fois qu'elle est nécessaire

  (par exemple, dans la boite de dialogue *Switch/Checkout*... vue précédemment).

- En ligne de commande :

  ```
  $ git branch
  ```

  Le nom de la branche courante apparaît précédé d'une étoile.

Créer une nouvelle branche

Cette opération consiste à placer, sur un commit existant, le sommet d'une *nouvelle* branche (qui pourra croître indépendamment des autres).

Depuis l'interface graphique

Menu contextuel > *TortoiseGit* > *Create Branch*...

- on doit choisir un nom pour la nouvelle branche ;
- dans la section « Base on », on peut choisir sur quel commit la nouvelle sera créée (par défaut: commit courrant `HEAD`) ;
- si on coche la case *Switch to new branch* (en bas à droite) la nouvelle branche deviendra la branche courante.

En ligne de commande

Pour créer une nouvelle branche sur le commit courant :

```
$ git branch <nom_nouvelle_branche>
```

Pour créer une nouvelle branche à un autre emplacement :

```
$ git branch <nom_nouvelle_branche> <revision>
```

Ces commandes ne changent pas la branche courante. Pour créer une nouvelle branche *et* en faire la branche courante, utilisez plutôt :

```
$ git checkout -b <nom_nouvelle_branche>     # ou
$ git checkout -b <nom_nouvelle_branche> <revision>
```

Changer de branche

Cette opération consiste à modifier la copie de travail pour la mettre dans le même état que le sommet d'une branche.

 **Indice:** Pour pouvoir l'effectuer, il est nécessaire que la copie de travail ne contienne aucune modification non commitée.

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Switch/Checkout*...
    - s'assurer que `Branch` est bien coché,
    - sélectionner dans la liste correspondante le nom de la branche,
    - valider.
- En ligne de commande :

```
$ git checkout <branche>
```

À propos de `git checkout`

La commande `git checkout` est utilisée dans divers contextes, qui rendent difficile à percevoir sa cohérence interne.

La fonction première de cette commande est de *modifier l'état de la copie de travail*. Selon ses arguments, elle a des effets supplémentaires :

- un branche : changer la branche courante
- une révision : passer en mode *detached HEAD*

Exercice

1. Dans le dépôt que vous avez créé au premier exercice, créez une branche nommée `style`, et placez-vous dans cette branche.
2. Modifiez la feuille de style (par exemple pour changer la couleur de fond) et commitez vos changements.
3. Revenez sur la branche `master`. Constatez que vos changements de style ont disparu (pour l'instant).
4. Dans la branche `master`, modifiez ou ajoutez du contenu au fichier HTML, et commitez vos modifications.
5. Revenez sur la branche `style`. Constatez que vos changements de style ont réapparu, mais que vos dernières modifications dans le fichier HTML ont, elles, disparu.
6. Modifiez à nouveau la feuille de style (par exemple pour changer la police) et commitez vos changements.

Fusionner deux branches

L'opération de **fusion** (en anglais *merge*) permet d'intégrer les modifications d'une branche dans une autre.

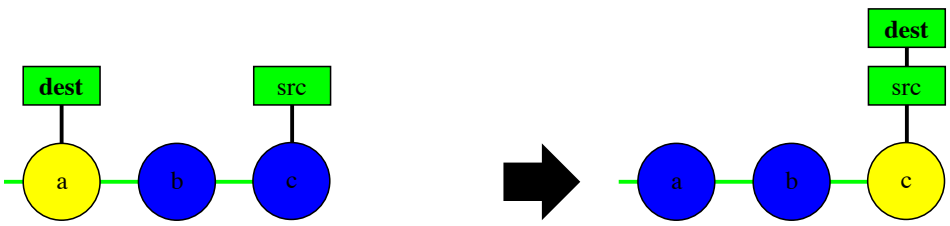
> ✎ **Note:** GIT permet également de fusionner plus de deux branches dans une même opération, mais nous n'irons pas jusque là dans ce cours.

Il y a deux situations possibles, selon les positions relatives de la branche à fusionner (source) et de la branche destination.

Fusion sans commit

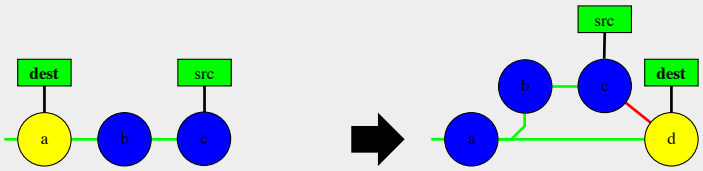
Si la branche destination est contenue dans la branche source,

la fusion a simplement pour effet de déplacer le sommet de la branche cible.



Ce type de fusion est appelée *fast forward*.

> ✎ **Note:** Ce comportement préserve autant que possible un historique linéaire, donc plus simple.
>
> Cependant, dans certains cas, on souhaite forcer la création d'un commit même lorsqu'on est dans cette situation (c'est notamment le choix qui a été fait dans l'exemple du site web).
>
> Pour cela, en ligne de commande, on ajoutera l'option `--no-ff`.



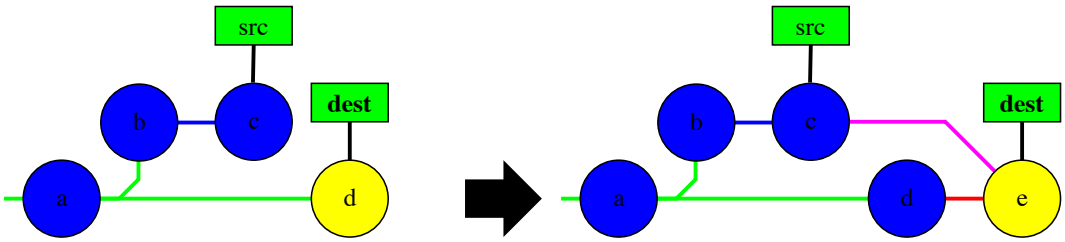
Avantage : les deux branches gardent leur identité dans le graphe.

Fusion avec commit

Si la branche destination et la source ont divergé,

la fusion crée un nouveau commit intégrant les modifications des deux branches ;

ce commit devient le sommet de la branche destination.



✎ **Note:** Bien sûr, cela suppose que les modifications des deux branches soient compatibles. La section suivante traite des conflits, et de comment les résoudre.

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Merge*...
  - s'assurer que `Branch` est bien coché,
  - sélectionner la branche à fusionner dans la branche actuelle,
  - valider.
- En ligne de commande :

```
$ git merge <branche>
```

Exercice

1. Nous allons maintenant fusionner la branche `style` (créée à l'exercice précédent) avec la branche `master`.

2. Placez-vous dans la branche `master`, et appliquez la méthode de votre choix (ligne de commande ou interface graphique) pour y fusionner la branche `style`.
3. Constatez que toutes vos modifications (contenu HTML et style) sont maintenant visibles.
4. Constatez également (dans l'historique) que le commit ainsi créé a *deux* commit parents.

Gérer les conflits

Motivation

La fusion de branches est automatiquement gérée par GIT lorsque les modifications des deux branches portent sur :

- des fichiers différents, ou
- des parties distinctes des mêmes fichiers texte.

Exemple géré par GIT

Branche 1 :

```
- La première ligne
+ La première ligne modifiée
La deuxième ligne
La troisième ligne
```

Branche 2 :

```
La première ligne
La deuxième ligne
- La troisième ligne
+ La troisième ligne modifiée
```

Fusion :

```
La première ligne modifiée
La deuxième ligne
La troisième ligne modifiée
```

Exemple non géré par GIT

Branche 1 :

```
- La première ligne
+ La première ligne modifiée
```

```
La deuxième ligne
- La troisième ligne
+ La troisième ligne modifiée
```

Branche 2 :

```
- La première ligne
+ La première ligne changée
La deuxième ligne
La troisième ligne
```

Conflit

On a donc un **conflit** lorsque les deux branches modifient :

- un même fichier binaire, ou
- la même partie d'un fichier texte.

Dans ce cas, le conflit doit être résolu à la main avant de pouvoir créer le commit de fusion.

Remarque

⚠ **Avertissement:** La stratégie de GIT n'est qu'une heuristique.

Cela signifie que des branches jugées compatibles par GIT peuvent être sémantiquement incohérentes. Il convient donc de vérifier le résultat de la fusion, même lorsqu'aucun conflit n'est signalé.

Mise en œuvre

Lorsque GIT rencontre un conflit au moment d'une fusion, un message indique les fichiers en conflit.

On est dans un état instable qui suppose :

- de résoudre le conflit, ou
- d'abandonner la fusion.

Fichiers comportant un conflit

Les fichiers texte comportant un conflit sont automatiquement modifiés pour :

- inclure les modifications non conflictuelles, et
- faire apparaître les deux versions concurrentes pour les modifications conflictuelles.

```
<<<<<<< HEAD
La 1e ligne modifiée
=======
La 1e ligne changée
>>>>>>> src
La 2e ligne
La 3e ligne modifiée
```

Les fichiers binaires ne sont pas modifiés.

Résolution du conflit

Une fois les fichiers en conflit corrigés, on peut résoudre le conflit :

- Menu contextuel > *TortoiseGit* > *Resolve*...
- En ligne de commande :

```
$ git commit -a
```

Le nouveau commit aura pour parents les sommets des branches fusionnées.

Abandon

On peut également décider d'abandonner la fusion :

- Menu contextuel > *TortoiseGit* > *Abort Merge*
- En ligne de commande :

```
$ git merge --abort
```

Exercice

1. Créez un nouveau dépot, et ajoutez-y un fichier `conflit.txt` contenant le texte suivant :

```
La première ligne
La deuxième ligne
La troisième ligne
```

2. Créez plusieurs branches, dans lesquelles vous modifierez différemment le fichier `conflit.txt`, en suivant les exemples ci-avant. Tentez ensuite de fusionner ces branches.
3. Lorsque GIT vous signale un conflit, constatez comment le fichier `conflit.txt` a été modifié, et résolvez le conflit.

Dépot distant

Notion

Un **dépôt distant** (en anglais *remote repository*) est un dépôt GIT, tout à fait similaire à un dépôt local, mais accessible à distance *via* une URL.

Exemple : https://github.com/pchampin/intro-git.git

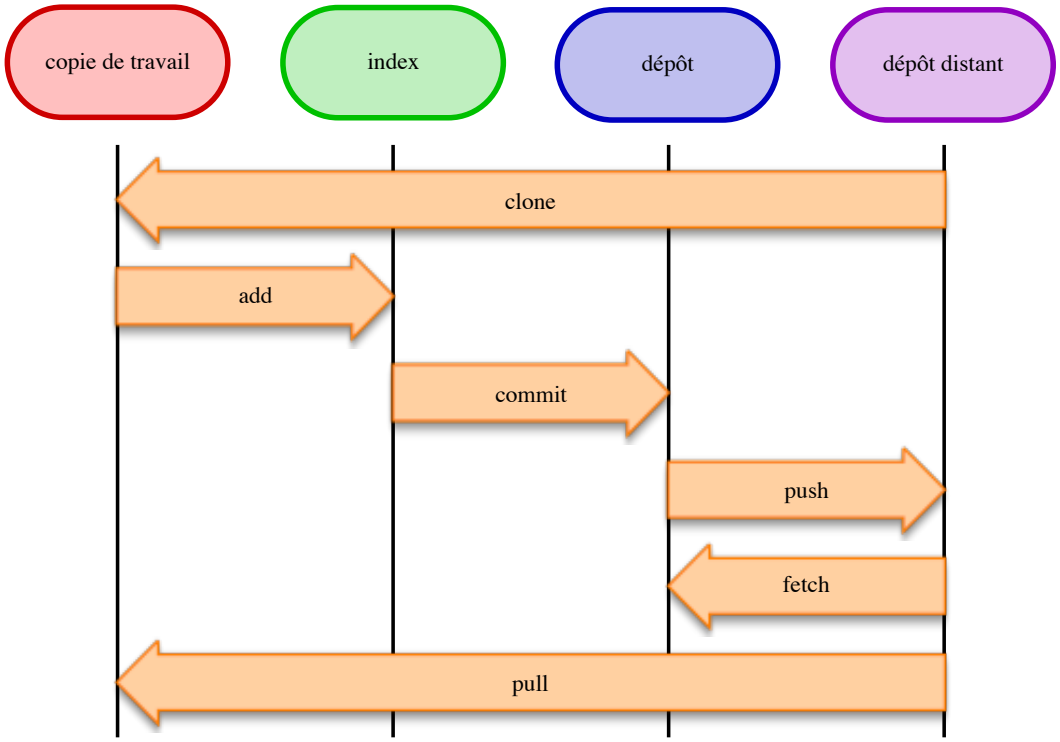
Un dépôt local peut être lié à un dépôt distant ; GIT offre des fonctionnalités pour copier des commits de l'un à l'autre.

> ✎ **Note:** En fait, un dépôt distant n'est pas forcément « à distance » (même si c'est le plus souvent le cas).
>
> On peut aussi utiliser comme dépôt distant un dépôt accessible dans un répertoire partagé, ou sur un support amovible (clé USB, disque dûr externe)...

Motivations

- sauvegarder le projet (fichiers + historique),
- travailler sur plusieurs machines,
- rendre le projet accessible à d'autres personnes,
- travailler sur un projet publié par quelqu'un d'autre,
- collaborer à plusieurs sur un projet (ce sera l'objet du chapitre suivant).

Vue d'ensemble



Mise en œuvre

- Créer un dépôt sur un serveur
- Cloner un dépôt distant
- Publier des commits
- Récupérer les commits

Créer un dépôt sur un serveur

Il existe plusieurs sites permettant d'héberger et de partager vos projets GIT :

github SOCIAL CODING GitLab Atlassian Bitbucket Framagit

dont un hébergé par l'université Lyon 1 :

http://forge.univ-lyon1.fr/

> ✎ **Note:** Concernant la forge de Lyon 1, en cas de problème avec les certificats SSL, consultez cette FAQ

Cloner un dépôt distant

Une fois votre dépôt distant créé, vous pouvez en faire un **clone** (une copie à la fois des fichiers et de l'historique) sous forme d'un dépôt local, qui sera lié à ce dépôt distant.

Mise en œuvre

- Menu contextuel > *Git Clone*...
    - dans le champs URL, sélectionner l'URL du dépôt distant ;
    - le cas échéant, modifier l'emplacement du dépôt local ;
    - valider.
- en ligne de commande :

```
$ git clone <url-dépôt-distant> <emplacement>
```

✎ **Note:** Il est possible de procéder à l'inverse : commencer à travailler sur un dépôt local, puis créer un dépôt distant pour publier le travail.

La procédure est un peu plus complexe, mais est généralement bien documentée par les services d'hébergement GIT, au moment où vous créerez le dépôt distant.

Problème depuis le campus Lyon1

Si vous rencontrez des problèmes pour cloner un dépôt, cela peut venir d'une mauvaise configuration du *proxy*. Dans `Git Bash`, tapez les deux commandes suivantes :

```
$ git config --global http.proxy http://proxy.univ-lyon1.fr:3128
$ git config --global https.proxy https://proxy.univ-lyon1.fr:3128
```

puis tentez à nouveau.

Publier des commits

Cette opération copie sur le dépôt distant les commits locaux (de la branche courante) qui n'y sont pas encore présents.

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Push*...
- En ligne de commande :

```
$ git push
```

> **Indice:** Cela suppose évidemment que vous soyez propriétaire du dépôt distant, ou que le propriétaire ait configuré son dépôt pour vous autoriser à le modifier.

Désynchronisation

Le `push` n'est possible que si la branche locale contient tous les commits présents dans la branche distante (plus, bien sûr, les nouveaux commits que vous voulez pousser).

Si la branche distante contient des commits inconnus de votre dépôt local (poussés depuis une autre machine, par vous ou quelqu'un d'autre), il faudra au préalable les récupérer (cf. ci-après).

Récupérer les commits

Cette opération copie dans le dépôt local les commits distants (de la branche courante) qui n'y sont pas encore présents. *Elle met également à jour la copie de travail*.

Cela est nécessaire

- lorsque vous travaillez sur plusieurs machines, et utilisez le dépôt distant pour les synchroniser, ou
- lorsque le dépôt distant est modifié par quelqu'un d'autre.

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Pull*...
- En ligne de commande :

```
$ git pull
```

Désynchronisation

Dans le cas le plus simple, le dépôt distant est « en avance » par rapport au dépôt local : il contient tous les commits du dépôt local, plus ceux que vous cherchez à récupérer.

Dans un cas plus complexe, des commits ont pû être ajoutés parallèlement dans les deux dépôts. Dans ce cas, `pull` effectue automatiquement une opération de fusion. Cela peut entraîner un conflit, qu'il faudra résoudre comme vu précédemment.

> **Indice:** Il est bien sûr préférable d'éviter ces conflits plutôt que de les résoudre *a posteriori*. Modifier la même information en parallèle n'est, de toute façon, pas une bonne idée...

Exercice

1. Créez un dépôt distant sur le service d'hébergement de votre choix.
2. Clonez le dans un dépôt local
3. Créez un fichier "message.txt" dans le dépôt local, commitez le et poussez le.
4. Constatez que le fichier message.txt apparaît bien sur la page Web de votre dépôt distant.

Collaboration

Motivation

- On a vu que GIT gérait l'évolution des fichiers, qu'elle soit linéaire ou non linéaire (branches) :
    - en facilitant la fusion des modifications parallèles, et
    - en détectant les conflits.
- Déjà utiles dans un contexte individuel, ces fonctionnalités vont s'avérer primordiales dans un contexte *collectif*.

Notions

- Lorsqu'on travaille à plusieurs,
    - chacun possède une copie des fichiers.
- Lorsqu'on travaille à plusieurs **avec GIT**,
    - chacun possède une copie des fichiers **et du dépôt**.
- On ne s'échange plus les fichiers individuellement,
    - mais des **commits** (donc des états *cohérents* de l'ensemble des fichiers).
- On met en commun en fusionnant les branches.

Dépôt distant

On a vu dans la section précédente qu'un dépôt local pouvait être lié à un dépôt distant.

En fait, un dépôt GIT peut être lié à un nombre arbitraire de dépôts distants, chacun identifié par un nom.

Lors d'un `clone`, le dépôt cloné est automatiquement ajouté comme dépôt distant, sous le nom `origin`.

Branche de suivi

Pour chaque branche de chaque dépôt distant, GIT crée dans le dépôt local une branche spéciale appelée **branche de suivi** (en anglais *remote-tracking branch*). Son nom est de la forme :

```
<nom-dépôt-distant>/<branche>
```

Cette branche reflète l'état de la branche distante correspondante ; elle n'est jamais modifiée directement.

Elle peut en revanche être *fusionnée* à une branche locale, afin d'y intégrer les modifications faites par d'autres.

Mise en œuvre

- Lier à un dépôt distant
- Mettre à jour les branches de suivi
- Fusionner une branche de suivi

Lier à un dépôt distant

- Menu contextuel > *TortoiseGit* > *Settings* > *Git* > *Remotes*

  (interface complète de gestion des dépôts distants)

- En ligne de commande :

```
$ git remote add <nom> <emplacement>
```

 **Indice:** Cette opération est à faire une seul fois par dépôt (et par dépôt distant), pour pouvoir ensuite interagir avec le dépôt distant.

Mettre à jour les branches de suivi

On a vu précédemment la commande `git pull` qui récupère les commits distants *et les fusionne* dans la branche courante.

La commande `git fetch` permet de simplement récupérer les commits distants et de mettre à jour les branches de suivi, *sans impacter* les branches locales.

 **Note:** En fait, `git pull` n'est ni plus ni moins un raccourci qui effectue un `git fetch` suivi d'un `git merge`.

Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Fetch...*
  - s'assurer que `Remote` est bien coché,

    - sélectionner le dépôt distant souhaité la liste correspondante,
    - valider.
- En ligne de commande :

```
$ git fetch <nom-dépôt-distant>
```

Indice: Les branches de suivi sont créées par le `fetch`.

Ainsi, si de nouvelles branches sont créées dans le dépôt distant, les branches de suivi correspondantes seront également ajoutées.

Fusionner une branche de suivi

Le principe est le même que pour la fusion entre branches locales.

- Menu contextuel > *TortoiseGit* > *Merge*...
    - s'assurer que `Branch` est bien coché,
    - sélectionner la branche de suivi à fusionner dans la branche actuelle,
    - valider.
- En ligne de commande :

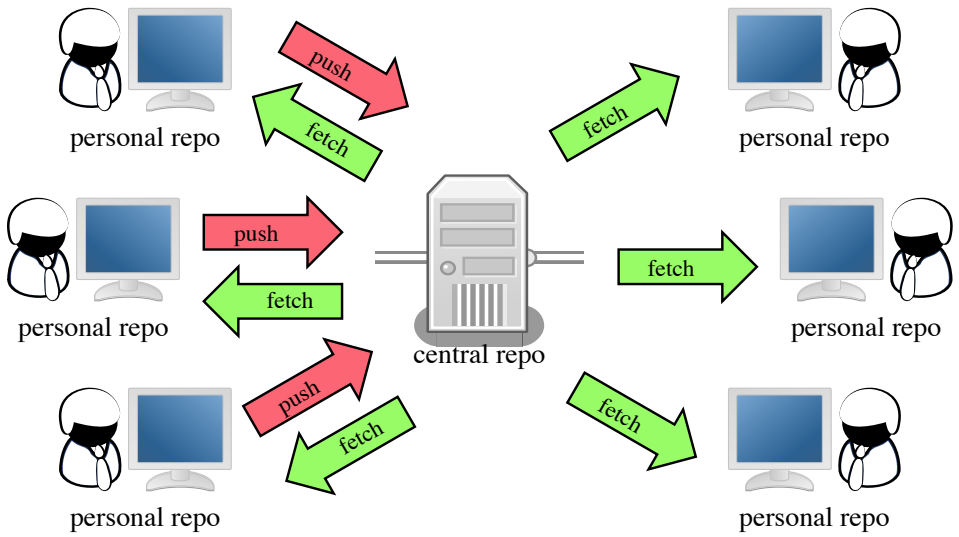
```
$ git merge <branche-de-suivi>
```

Types de collaborations

La flexibilité de GIT permet de multiples formes d'organisation pour le travail collaboratif.

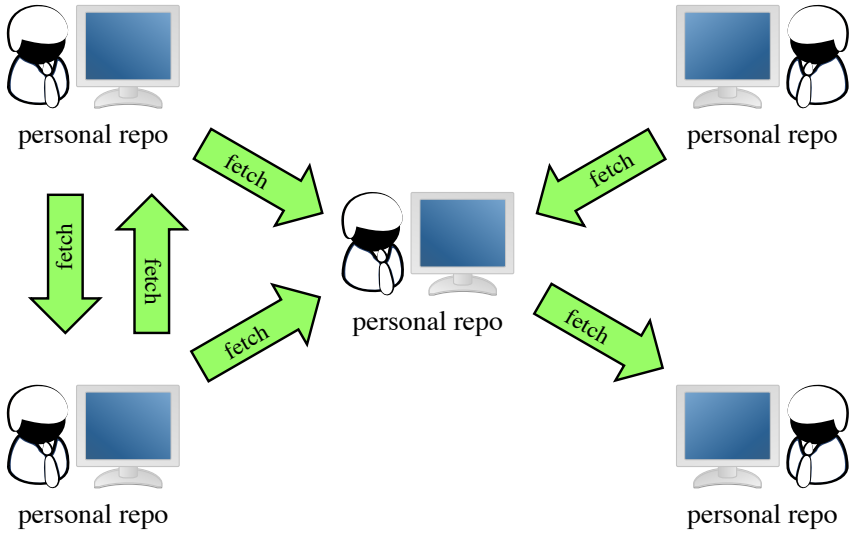
On donne ici quelques exemples (non exhaustifs, et non exclusifs).

Organisation en étoile



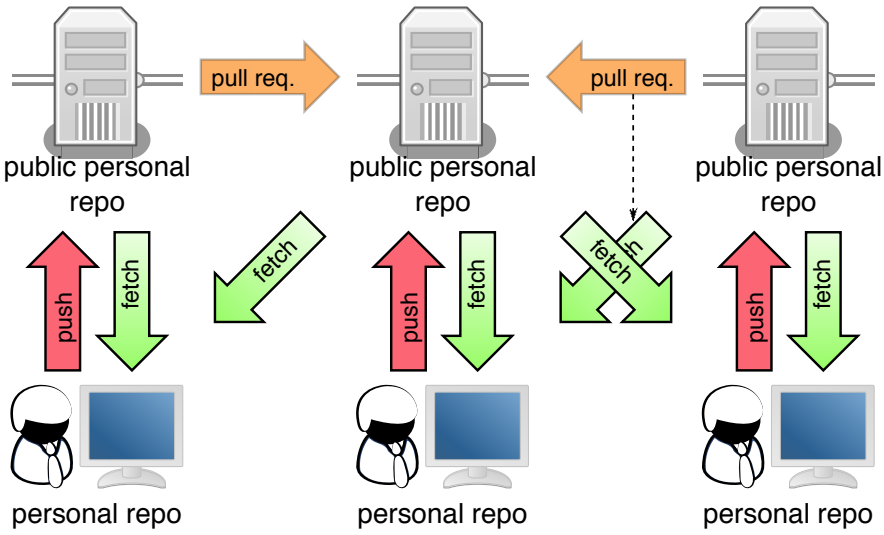
> ✎ **Note:** Ce type de collaboration est inspiré des VCS centralisés, et est simple à mettre en œuvre.
>
> Il suppose de mettre un place un unique dépôt distant sur lequel plusieurs collaborateurs ont les droits en écriture.

Organisation pair-à-pair



> ✎ **Note:** Ce type de collaboration est très flexible. On peut notamment le mettre en œuvre sans disposer d'un serveur, en utilisant par exemple des média amovibles (clé USB, carte DS, disque externe...) comme "dépot distant" pour communiquer entre les différents acteurs.

Organisation "github"



✎ **Note:** Ce type de collaboration est encouragé par les sites d'hébergement comme Github (mais également d'autres).

Chaque collaborateur dispose d'un clone public du projet, et y publie les commits qu'il souhaite partager.

Il sollicite ensuite les autres collaborateur pour tirer ces commits dans leur propre dépôt/ On appelle cette sollicitation un *pull request* (ou plus rarement un *merge request*).

La plupart du temps, l'un des dépôts publics (celui du leader du projet, typiquement) est considéré comme la référence, donc ce type d'organisation se rapproche de l'organisation en étoile, mais permet à n'importe qui de proposer des modifications, sans avoir besoin d'obtenir les droits en écriture sur le dépôt de référence.

Exercice

Le meilleur moyen d'expérimenter la collaboration est de travailler avec des collaborateurs !

Si vous voulez essayer, publiez votre dépôt sur l'espace partagé de votre choix, et demandez à un collègue d'en faire un clone.

C'est à vous de fixer les droits sur votre dépôt distant en fonction de ce que vous souhaitez (accessible en lecture seule, ou bien en lecture / écriture).

Ré-écrire l'histoire

✎ **Note:** L'objectif n'est pas de travailler sur ces notions, mais de signaler leur existence pour plus tard... et pour les curieux :-)

Motivation

Avant de publier un ensemble de commits, on peut souhaiter le « nettoyer » un peu,

notamment pour rendre l'historique du projet plus lisible.

⚠ **Avertissement:** Ceci ne doit **jamais** être fait sur des commits qui ont déjà été partagés avec d'autres personnes (notamment avec `git push`).

Cela créerait une incohérence entre les dépôts.

Amendement

Il arrive que l'on fasse un commit incomplet :

- oubli d'ajouter certains fichiers / certaines modifications,
- coquilles dans les ajouts...

On peut bien sûr corriger cet oubli dans un nouveau commit, mais cela contredit l'idée qu'un commit représente un état *cohérent* de l'ensemble des fichiers.

Dans ces situations, il est possible de modifier (**amender**) le dernier commit créé.

Mise en œuvre

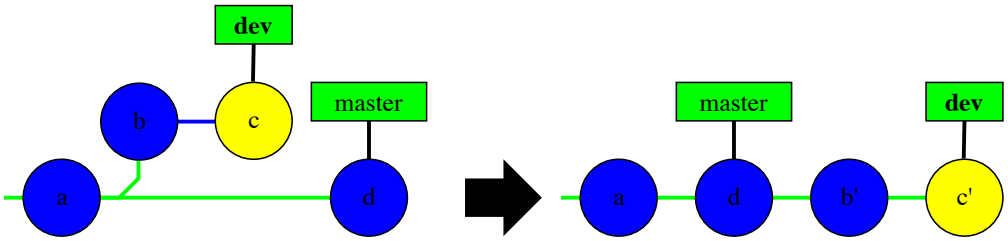
- Dans la boite de dialogue de commit :
    - cocher le bouton radio *Amend last commit*
- En ligne de commande :

```
$ git commit --amend
```

Rebase

On a vu que la fusion de deux branches créait un commit à plusieurs parents si les branches avaient divergé.

Si on préfère garder un historique linéaire, GIT permet de « rejouer » les modifications d'une branche à partir du sommet de l'autre branche, en re-créant les commits correspondants.



Mise en œuvre

- Menu contextuel > *TortoiseGit* > *Rebase*...
- En ligne de commande (depuis la branche à « rebaser ») :

```
$ git rebase <branche-destination>
```

Pour aller plus loin

Se documenter

- Deux tutoriels graphiques et animés ici et là.
- Si vous voulez en savoir plus sur GIT, consultez son excellente documentation sur git-scm.org ainsi que les vidéos très instructives !

Autres outils de gestion de version

- Vous n'êtes pas certains de préférer GIT? Prenez le temps de comparer les différents outils de gestion de version. Il existe de nombreux comparatifs en ligne, comme par exemple sur Wikipedia.

Mercurial

- Mercurial (abbrégé Hg) est un gestionnaire de version, notamment utilisé sur la forge de Lyon1 ou BitBucket.
- Il est similaire à GIT, mais comporte quelques différences (de terminologie notamment).
- Un guide de pour passer de GIT à Mercurial est disponible ici : https://www.mercurial-scm.org/wiki/GitConcepts

Un dernier conseil

Rien de tel que la pratique pour maîtriser GIT (ou tout autre outil de gestion de version), alors n'hésitez pas à utiliser abondamment ces outils, même pour vos petits projets...

Crédits

Ce support a été réalisé par Pierre-Antoine Champin et Amélie Cordier.

Merci à Isabelle Gonçalves et Jocelyn Delalande pour leurs contributions.