Programmation et projet encadré - L7TI005

Git : un peu plus loin

Yoann Dupont yoann.dupont@sorbonne-nouvelle.fr
Pierre Magistry pierre.magistry@inalco.fr

2024-2025

Université Sorbonne-Nouvelle
INALCO
Université Paris-Nanterre

Buts de ces slides :

- Apprendre à corriger des erreurs en git
- Apprendre à gérer certains conflits

GitHub : Corriger des erreurs

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Nous pouvons utiliser les commandes pour cela :

- `git reset`
- `git revert` → pas cette fois !
- `git stash`
- `git checkout` → on le (re?)verra au S2

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Nous pouvons utiliser les commandes pour cela :

- `git reset`
- `git revert` → pas cette fois !
- `git stash`
- `git checkout` → on le (re?)verra au S2

De manière générale, il est recommandé d'utiliser régulièrement `git fetch` pour récupérer des métadonnées du dépôt. Cette commande a l'avantage de ne pas pouvoir échouer (au niveau de git).

Un peu de syntaxe avant

Quelques éléments à savoir avant de continuer :

- `HEAD` : représente le commit sur lequel vous êtes en train de travailler
- `tag` : représente le commit sur lequel on a placé l'étiquette
- `~N` : représente l'ascendance directe de votre commit (linéaire, par défault N=1 représente le commit parent)
- `^N` : représente le n-ième parent du commit (non linéaire, par défault N=1 représente le commit parent)

source : https://git-scm.com/docs/git-rev-parse

Un peu de syntaxe avant

Quelques éléments à savoir avant de continuer :

- `HEAD` : représente le commit sur lequel vous êtes en train de travailler
- `tag` : représente le commit sur lequel on a placé l'étiquette
- `~N` : représente l'ascendance directe de votre commit (linéaire, par défault N=1 représente le commit parent)
- `^N` : représente le n-ième parent du commit (non linéaire, par défault N=1 représente le commit parent)

On peut faire des choses très précises, on se contentera de travailler ici dans l'ascendance directe.

source : https://git-scm.com/docs/git-rev-parse

Faire machine arrière

`git reset` permet de faire machine arrière dans les commits. La commande permet de revenir dans le passé commit par commit.

Faire machine arrière

`git reset` permet de faire machine arrière dans les commits. La commande permet de revenir dans le passé commit par commit.

Ce semestre, on utilisera surtout cette commande pour annuler un/plusieurs commits non poussés, mais la commande a d'autres cas d'usage, qu'on pourra voir au S2.

Git reset au plus simple

`git reset`

Revient à la version "HEAD" du dépôt.

Annule tous les *add*, mais n'annule aucun *commit* !

Si vous avez déjà commité, il faudra utiliser une référence à un commit où on souhaite revenir.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Défaire jusqu'à un commit spécifique

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Défaire jusqu'à un commit spécifique

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

```
git reset --soft HEAD~
```

Revient à dernière la version du dépôt mais n'annule la mise-en-place (*staging*).

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Défaire jusqu'à un commit spécifique

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

```
git reset --soft HEAD~
```

Revient à dernière la version du dépôt mais n'annule la mise-en-place (*staging*).

Pour défaire plus d'un commit, il faudra utiliser les références vers un commit spécifique comme indiqué slide 3.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Défaire jusqu'à un commit spécifique

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

```
git reset --soft HEAD~
```

Revient à dernière la version du dépôt mais n'annule la mise-en-place (*staging*).

Pour défaire plus d'un commit, il faudra utiliser les références vers un commit spécifique comme indiqué slide 3.

Attention :

`git reset` fonctionne sur des commits entiers, pas sur des fichiers spécifiques.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Revenir à un commits spécifique

```
git reset <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettre et nombres).
- un tag
- etc.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Revenir à un commits spécifique

```
git reset <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettre et nombres).
- un tag
- etc.

Les options `soft` et `hard` s'appliquent comme précédemment.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Se "téléporter" à un état

```
git checkout <commit>
git checkout <fichier>
git checkout <commit> - [fichier ...]
```

Où `commit` peut être l'identifiant du commit, un tag, une branche (ex: main), etc.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Se "téléporter" à un état

```
git checkout <commit>
git checkout <fichier>
git checkout <commit> - [fichier ...]
```

Où `commit` peut être l'identifiant du commit, un tag, une branche (ex: main), etc.

Si on ne donne rien (ni `fichier` ni `commit`), la commande ne fait rien et montre juste les fichiers modifiés.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Se "téléporter" à un état

```
git checkout <commit>
git checkout <fichier>
git checkout <commit> - [fichier ...]
```

Où `commit` peut être l'identifiant du commit, un tag, une branche (ex: main), etc.

Si on ne donne rien (ni `fichier` ni `commit`), la commande ne fait rien et montre juste les fichiers modifiés.

Pour que le commande serve à quelque chose d'utile, on donne au moins soit fichier, soit commit. On peut donner les deux.

Si on ne donne qu'un commit, on va à l'état du commit en question pour l'ensemble du dépôt. Si on ne précise aucun commit, on va à `HEAD`.

Si on donne un fichier, seul ce fichier sera modifié.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Il est possible de récupérer les changements avec un `git pull`, mais cette commande peut échouer si nous avons des modifications en conflit (au moins un fichier modifié en local et en ligne). La seule solution est alors de nettoyer votre dépôt local pour retirer tous les conflits.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Il est possible de récupérer les changements avec un `git pull`, mais cette commande peut échouer si nous avons des modifications en conflit (au moins un fichier modifié en local et en ligne). La seule solution est alors de nettoyer votre dépôt local pour retirer tous les conflits.

Pour éviter de perdre ses modifications, il est possible de les mettre de côté (*stash*) pour les ressortir plus tard.

Mettre de côté des modifications 1

```
git stash push [-m <message>]
```

Où :

- `-m <message>` permet d'associer un message à la "mise de côté", utile pour avoir se rappeler ce qu'on a mis dedans sans regarder dans le détail.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Mettre de côté des modifications 1

```
git stash push [-m <message>]
```

Où :

- `-m <message>` permet d'associer un message à la "mise de côté", utile pour avoir se rappeler ce qu'on a mis dedans sans regarder dans le détail.

git stash va mettre de côté vos modifications dans un index. Chaque appel à *git stash push* crée une nouvelle entrée.

source : https://til.bhupesh.me/git/how-to-undo-anything-in-git

Mettre de côté des modifications 2

```
git stash list
```

permet de voir la liste des paquets de modifications mis de côtés.

Mettre de côté des modifications 2

```
git stash list
```

permet de voir la liste des paquets de modifications mis de côtés.

```
git stash show [-p] <stash>
```

Où :

- `-p` permet d'afficher un diff avec le stash
- `<stash>` est un identifiant de stash (typiquement son indice)

permet de voir le contenu d'un *stash*.

Mettre de côté des modifications 3

```
git stash apply <stash>
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

Mettre de côté des modifications 3

```
git stash apply <stash>
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

Le différence en `apply` et `pop` est que `pop` supprime le stash une fois appliqué.

Mettre de côté des modifications 3

```
git stash apply <stash>
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

Le différence en `apply` et `pop` est que `pop` supprime le stash une fois appliqué.

```
git stash drop <stash>
```

Supprime manuellement un *stash*.

git stash pop $\iff$ *git stash apply* + *git stash drop*