Introduction à Git

TDs

Mathieu Loiseau

Master Didactique des Langues Parcours DILIPEM
Master Sciences du Langage Parcours IdL
Université Grenoble Alpes

septembre 2019

Table des matières

TD n° 1

Fonctionnalités de base de Git

Le présent document vise à montrer en contexte des fonctionnalités de base de *Git*, en les intégrant dans la pratique — de manière un peu artificielle, certes. . .

Pour un premier panorama des commandes de git peut être trouvé ici `https://try.github.io/`.
Beaucoup d'informations proviennent de `https://www.atlassian.com/git/tutorials/`.
Merci à Arnaud Bey pour ses suggestions.

Conventions : Dans le document suivant, les lignes de commande shell seront indiqués de la `manière suivante` (avec les mots-clés notés `ainsi`). Des chevrons seront utilisés pour différencier les paramètres où vous pouvez mettre la valeur de votre choix de ceux qui ont une sémantique propre. Il n'est pas opportun de recopier les chevrons en question.

Exemple : Alain Connu, pour exécuter la commande `git config user.name "<nom>"`, saisira `git config user.name "Alain Connu"`.

1 Débuter avec git

Git est un logiciel open source et peut être obtenu ici : `http://git-scm.com/downloads`

Git permet de gérer les versions et évolutions de code source (et par extension de tout fichier texte). Pour ce faire, pour chaque modification significative du code (ajout de fonctionnalités, correction de bugs, etc.), on pourra faire un *commit*, c'est-à-dire sauvegarder une version du code.

Git stocke les versions successives d'un dossier. Une fois « surveillé » par Git, un dossier est un *repository* ou « *repo* ». En français on parlera de « dépôt ».

Git permet également de travailler à plusieurs. Afin de savoir qui fait quoi, git permet de stocker des informations à apposer à chaque *commit* de l'utilisateur.

- `git config user.name "<nom>"` : tous les *commit* effectués sur ce dépôt sont crédités à l'utilisateur appelé <nom> ;
- `git config --global user.name "<nom>"` : tous les *commit* effectués à partir de ce compte utilisateur (quel que soit le dépôt) sont crédités à l'utilisateur appelé <nom> ;

— `git config --global user.email "<email>"` : l'adresse e-mail de l'utilisateur est <email>.

Exercice d'application : Nous allons voir ici quelques uns des effets de ces commandes.

1. Installer Git
2. Définir son nom pour l'usage de git sur la machine utilisée *`git config --global user.name "John Doe"`*
3. Dans le dossier utilisateur (`~` sous Linux / `C:\Users\NomDUtilisateur` sous Windows), ouvrir le fichier `.gitconfig`. Que remarquez-vous ? *Le nom de l'utilisateur est indiqué.*
4. Modifier l'adresse de courrier électronique associée à l'utilisateur *`git config --global user.email "john.doe@anonymo.us"`*
5. Que s'est-il passé dans le fichier de configuration ? *L'adresse e-mail a été ajoutée.*

Beaucoup d'autres choses peuvent être configurées, du logiciel utilisé pour éditer les fichiers à celui pour traiter les cas de conflits entre les versions. . .

En effet, il peut arriver que deux versions entrent en conflit, dans ce cas l'utilisateur devra décider de la version à conserver. Plutôt que de manipuler ces différentes version à la main (git annote le code entrant en conflit) un logiciel de comparaison de fichier peut être utilisé. Voir par exemple :
— `https://alexjoz.gitbooks.io/code-life/content/chapter6.html` (Linux).
— `http://blog.nerdbank.net/2011/10/how-to-get-meld-working-with-git-on.html` (Windows) ;

> La commande `git config --list` affichera toutes les informations de configuration de votre git. Le fichier `~/.config`, donne moins d'informations mais de manière plus lisible.

Nota Bene 1.1: Afficher la configuration de git

2 Créer un dossier un dépôt *Git*

Pour créer un dépôt git vous pouvez :
— prendre n'importe quel dossier et l'initialiser en tapant `git init` si vous êtes dans le dossier ou `git init <dossier>` ;
— cloner un dépôt git existant avec `git clone <dépôt> <directory>` (si `<directory>` n'est pas spécifié, c'est dans le répertoire courant que le *dépôt* sera cloné.)

Exercice d'application : Pour l'exercice suivant vous allez vous contenter de créer un dossier (en ligne de commande) dans un dossier existant. Vous ferez ensuite de ce dossier votre dépôt git.

6. Ouvrir la console.
7. Se déplacer jusqu'à votre dossier de travail en utilisant `ls` et `cd`[1].
8. Créer un dossier au nom de votre projet.
9. Aller dans le dossier ainsi créé.
10. Faire de ce dossier un dépôt git. *En étant dans le bon dossier, il suffit de saisir : `git init`*

1. `dir` et `cd` sous windows

3 Effectuer un “commit”

Le flux de travail de base de *git* consiste en le fait d'effectuer des modifications puis de les grouper en ensemble cohérent avant de les stocker dans un état du code. On appelle l'action de créer une version du code un *commit*.

- La (sous-)commande `add` permet d'ajouter le ou les fichiers ainsi sélectionnés au prochain *commit* :
  - `git add <fichier>` : ajoute <fichier> au prochain *commit* ;
  - `git add <dossier>` : ajoute tout le contenu de <dossier> au prochain *commit* ;
  - `git add -p` : ajoute tous les changements au prochain *commit* ;
  - `git add --all <fichier>` : ajoute tous les changements à <fichier> (y compris la suppression) pour le prochain *commit* ;
  - `git reset <fichier>` : retirer les modifications à <fichier> de la sélection pour le prochain *commit*.
- `git status` : permet d'avoir la liste des fichiers modifiés et leur situation par rapport au prochain *commit* ;
- `git log` : permet d'avoir la liste des *commits* pour la branche courante ;
- La (sous-)commande `commit` permet de créer une nouvelle version du code [2] :
  - `git commit` : ouvre un éditeur de texte pour spécifier les informations du commit, qui est effectué à la fermeture de l'éditeur de texte ;
  - `git commit -m "<message>"` : effectue le commit avec comme informations <message> ;
  - `git commit -a` : ajoute au commit suivant toutes les modifications effectuées sur des fichier préalablement inclus dans un commit (les créations de fichiers ne sont pas ajoutées au commit) ;
  - `git commit --amend` : met à jour le dernier commit avec les changements sélectionnés. **Attention** : cette commande est à utiliser avec parcimonie, surtout quand vous avez plusieurs dépôts pour le même projet. `git commit --amend` « réécrit l'historique », et ne doit être utilisé que si le `commit` précédent n'a pas été publié ou utilisé par ailleurs.

Exercice d'application :

11. Dans le dépôt, créer un fichier texte appelé `README.md`.
12. Dans ce fichier indiquer le nom de votre projet en titre. Vous pouvez utiliser pour cela la syntaxe *Markdown* [3] *Pour indiquer le titre d'un projet en titre du document README.md :* `# Titre du document`
13. Toujours dans le dépôt, créer un fichier texte appelé `LICENSE`.
14. Ajouter dans ce fichier, le contenu de la licence MIT [4].
15. Exécuter la commande `git status`. Quelle est la situation ? *Aucun des deux fichiers créés n'est « suivi » (ajouté à la liste des modification pour le prochain* commit*).*
16. Préparer un *commit* pour stocker dans une nouvelle version l'ajout de la licence, puis regarder le statut.
    `git add` *LICENSE*
    `git status`
    La création de LICENSE est ajoutée pour le prochain `commit`*.*

2. Pour toujours bien savoir ce qu'on fait, il est conseillé d'utiliser `git status` avant tout `commit`.
3. `https://guides.github.com/features/mastering-markdown/`
4. `http://opensource.org/licenses/MIT`

17. Effectuer le *commit*.
    `git commit -m "ajout de licence"`
18. Ajouter une section contributeurs dans votre `README.md`, y inclure votre nom. *Avec la syntaxe Markdown, une possibilité :*
    `## Contributors`
    `* Prénom Nom`
19. Effectuer un *commit* pour cette modification `git commit -m "ajout du README"`
20. Vous avez oublié d'ajouter vos informations de contact : mettre à jour le README et amendez le *commit* précédent pour qu'il intègre votre modification.[5] *Modification du texte :*
    ** Prénom Nom → * Prénom Nom <prenom.nom@monmail.org>*
    `git commit --amend -m "ajout du README"`

4 Gérer des branches

Git permet également d'expérimenter, de faire des tests sans affecter le code stable. Pour ce faire, on va avoir recours à ce que l'on appelle des « branches », qui permettront de scinder le projet en différente version. Dans la figure 1.1, 4 branches sont représentées simultanément (il est à noté qu'elles sont nommées).

— `git branch` : lister toutes les branches du dépôt ;
— `git branch <nomBranche>` : créer la branche <nomBranche> (pas d'espace dans les noms de branche) ;
— `git checkout <nomBranche>`[6] : passer sur la branche <nomBranche> (elle doit exister[7]) ;
— `git merge <nomDUneBrancheNonManipulée>` : Fusionne la branche courante avec le contenu de la branche <nomDUneBrancheNonManipulée> ;

NB : Il n'est pas possible de changer de branche tant que toutes les modifications n'ont pas été soit annulées soit « *commitées* ». Pour traiter ce genre de problématique la commande `git stash`[8] existe, mais dépasse le cadre de cette formation. Il est bon de savoir qu'elle existe, malgré tout. De plus, fusionner des branches, peut créer des conflits, dans ce cas, il peut être bien pratique d'avoir un outil de fusion (*merge tool*) correctement configuré (cf. § 1 p. 4).

Exercice d'application :[9]

21. Comment s'appelle la branche principale par défaut ? Comment l'avez-vous trouvée ? *La branche principale s'appelle* `master` *par défaut. Dans votre projet « monobranche », vous l'obtenez en tapant :*

5. Pour bien voir ce qu'il se passe, vous pouvez utiliser `git status` et `git log`.
6. On peut également s'en servir pour voir un `commit` passé. Chaque commit a un identifiant unique (accessible avec `git log`), qui peut être passé en paramètre de `git checkout`.
7. `git checkout -b <nomBranche>` permet de basculer sur la branche <nomBranche>, après l'avoir créée si elle n'existait pas
8. `http://git-scm.com/docs/git-stash`
9. Sur une formation aussi courte, c'est difficile de montrer à quel point c'est pratique. On se contentera ici de montrer comment ça marche...

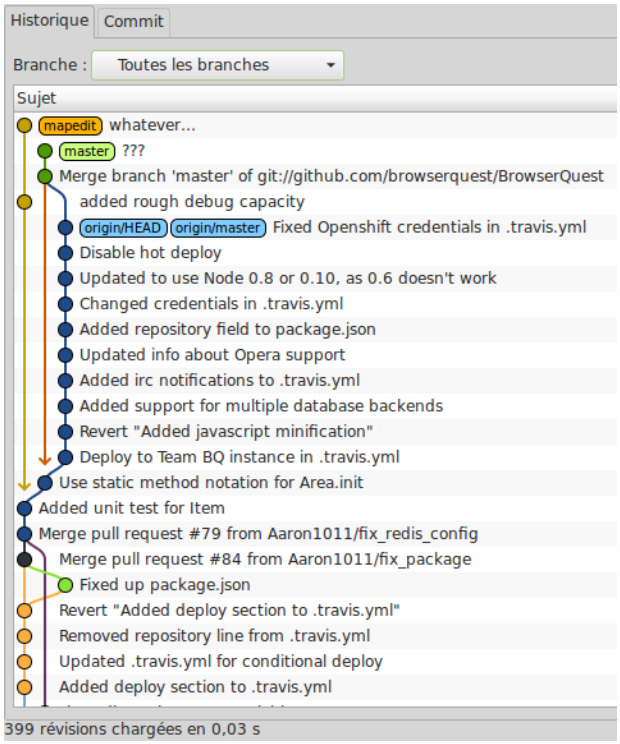

FIGURE 1.1 – Représentation graphique (gitg) de différentes branches d'un projet

`git branch` qui renverra :

```
[illegible]@[illegible]:~/Bureau/testgit$ git branch
* master
```

22. Créer une branche « testInstall » et l'afficher (vérifier que vous êtes bien sur la bonne branche avec `git branch`) *`git branch testInstall`*
`git checkout testInstall`
OU *`git checkout -b testInstall`*
23. Une fois dans la branche « testInstall », créer un fichier `INSTALL.md`, qui contiendra quelques informations (du genre « nécessite un serveur Apache avec php 5 »). Vous en profiterez pour ajouter à `README.md` un paragraphe « install » indiquant de se référer au document que vous venez de créer.
24. « Commiter » ces changements sur la branche « testInstall ».
25. Retourner effectivement sur la branche `master`. Que remarquez-vous ? *Le fichier `INSTALL.md` disparait quand on passe sur la branche master.*
26. Vérifier que vous n'avez rien perdu en retournant sur la branche `testInstall`. *C'est bon, il revient, comme de rien...*
27. Dans la branche master, modifier le fichier « README.md », pour y inclure une section « Install ». Et faire un commit des changements.
28. Passer d'une branche à l'autre pour vérifier que « README.md » est effectivement différent selon la branche.

29. Finalement, cela vous semble préférable d'avoir un fichier séparé pour l'installation, nous allons récupérer les modification faites dans la branche `testInstall` et les insérer dans la branche `master`. Se mettre dans la branche `master` et appeler la commande `merge` avec la branche `testInstall`. Que se passe-t-il ? *La fusion a échoué, du fait d'un conflit...*
30. Pour corriger ce problème, il va falloir résoudre le conflit manuellement en modifiant « README.md », avant de l'ajouter aux modifications à venir, puis farie le `commit`. *L'arborescence réalisée dans ce TD ressemblera à la figure 1.2.*

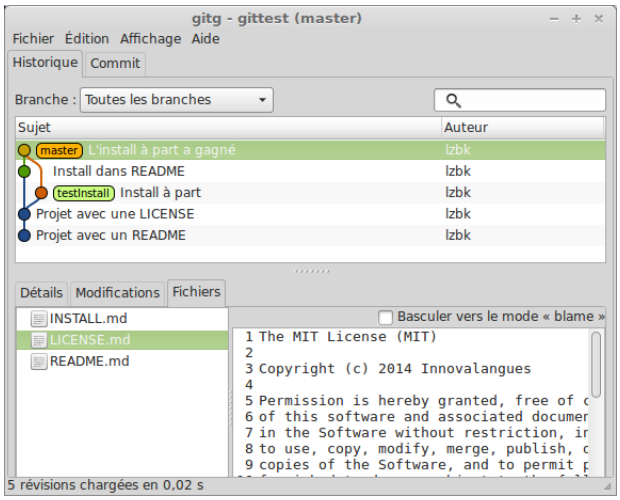

FIGURE 1.2 – Représentation graphique du projet réalisé jusqu'à la fin de la section 4

TD n° 2

Gestion de projet avec Gitlab

1 Travailler à plusieurs avec git : utilisation d'un dépôt distant

Les manipulations effectuées ici peuvent utiliser un autre système que framagit [1].

1.1 Créer un compte git

1. Aller à l'adresse `https://gricad-gitlab.univ-grenoble-alpes.fr/users/sign_in` et se créer un compte.
2. Un membre par groupe crée un dépôt au nom du projet et ajoute les autres membres en tant que collaborateur (cf. figure 2.1)

1.2 Gestion de dépôt distant

Les commandes données ci-dessous peuvent être approfondies ici : `http://git-scm.com/book/fr/v1/Les-bases-de-Git-Travailler-avec-des-d%C3%A9p%C3%B4ts-distants`.

- `git remote -v` : liste les dépôts distants ;
- `git remote add <nomDepotDistant> <urlDepot>` : ajoute le dépôt distant <urlDepot> et lui donne le nom <nomDepotDistant> (court si possible il sera réutilisé [2]) ;
- `git fetch <nomDepotDistant>` : récupérer le contenu du dépôt distant (les branches obtenues peuvent alors être fusionées avec la commande `merge` (cf. § 4 p. 6)) ;
- `git pull <nomDepotDistant> <brancheARecuperer>` : récupére la branche <brancheARecuperer> du dépôt distant <nomDepotDistant> et la fusionne avec la branche courante. Le paramètre <brancheARecuperer> est optionnel, s'il n'est pas spécifié, les fusions sont faites en fonction des paramètres par défaut. [3] ;

1. cf.
2. Git permet de renommer un dépôt distant avec `git remote rename <ancienNom> <nouveauNom>`
3. En général on créera en premier lieu le dépôt en ligne avant de le cloner sur sa machine. Ce faisant des associations par défaut sont faites automatiquement entre les branches ainsi créées. Mais si l'on veut définir manuellement ces associations, voir `http://stackoverflow.com/questions/4847101/git-which-is-the-default-configured-remote-for-branch#answer-4847136`.

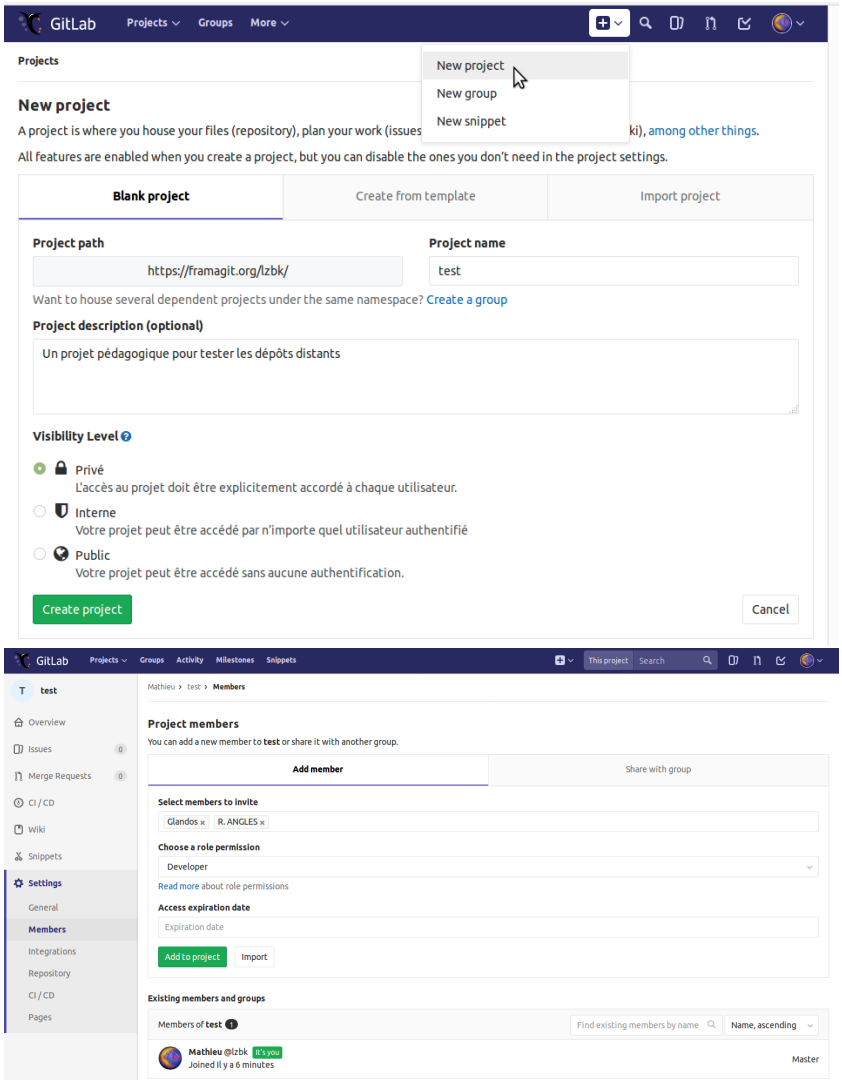

FIGURE 2.1 – Interface de création d'un dépôt sur Gitlab (cf. bouton « New project »)
Ajout d'un collaborateur à un dépôt (settings → members)

— `git push` `<nomDepotDistant>` `<nomBranche>` : ajoute à la branche <nomBranchs> du dépôt distant <nomDepotDistant> les modifications nouvelles de la branche <nomBranche> du dépôt local.

Exercice d'application : Dans cet exercice, deux rôles (A et B) doivent être joués. Tout l'exercice peut être fait indépendamment (néanmoins de manière synchrone). À partir de la question 6, A peut assister B pour ses tâches et *vice-versa*, pour bien comprendre ce qui se passe.

3. S'assurer que tout le monde est collaborateur du dépôt de travail distant.
4. Ajouter le dépôt distant à votre dépôt local. Pour ce faire récupérer l'url du dépôt distant

à partir de sa page github, cf. figure 2.2. *Si on veut nommer le dépôt distant « proj » et*

FIGURE 2.2 – url du dépôt distant, presser le bouton « presse papier »

que son url est `https://github.com/utilisateur/depot.git`, la commande sera :
`git remote add` *`proj https://github.com/utilisateur/depot.git`*

5. Récupérer la branche `master` du dépôt distant et la fusionner avec la branche `master` locale
   `git checkout` *`master`*
   `git pull` *`proj master`*
   OU
   `git checkout` *`master`*
   `git fetch` *`proj`*
   `git merge` *`proj/master`*
6. A envoie les modifications faites dans les questions précédentes au dépôt distant. Vérifier depuis la page github que le projet a bien évolué. ***`git push`*** *`proj master`*
7. B modifie le fichier `INSTALL.md` et y ajoute une section « Pré-requis techniques » et *commit* ses changements
8. A modifie le fichier `INSTALL.md` et y ajoute une section « Procédure d'installation » et *commit* ses changements
9. B envoie ses modifications au serveur distant. Que se passe-t-il ? Comment résoudre le problème ? [4] *Un message d'erreur s'affiche « error : impossible de pousser des références vers '<dépôt>'*
   astuce : Les mises à jour ont été rejetées car la branche distante contient du travail que vous n'avez pas en local. Ceci est généralement causé par un autre dépôt poussé vers la même référence. Vous pourriez intégrer d'abord les changements distants (par exemple 'git pull ...') avant de pousser à nouveau.
   Voir la 'Note à propos des avances rapides' dans 'git push --help' pour plus d'information. »
   Il suffit de suivre la procédure proposée.
   `git pull proj master` *Puis, si nécessaire résoudre les conflits avant de faire un commit. Et enfin :*
   `git push proj master`
10. A envoie ses modifications au serveur distant. Que se passe-t-il ? Comment résoudre le problème ? [4] *cf. correction de la question 9.*
11. Que reste-t-il à faire pour que tous les dépôts (locaux et distants) aient le même contenu *B doit récupérer l'état du dépôt distant.*

4. **Coup de pouce** : bien lire le message d'erreur.