Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/gil

Gestion d'identité en ligne

TP Git : Premiers pas avec Git

Exercice 0.

Présentation de Git.

1. Git est un *logiciel de gestion de versions*, c'est à dire qu'il permet de stocker un ensemble de fichiers (un projet) en conservant l'historique de leurs modifications. Les versions enregistrées dans l'historique d'un projet s'appellent des *commit*.

   La façon dont cet historique est conservé permet de revenir à une version passée d'un fichier ou de tout un projet, mais aussi d'appliquer à une version passée la suite de modification permettant d'aboutir à une version plus récente.

   Cela facilite la collaboration, en permettant de partager avec chaque membre d'un projet les modifications faites par les autres, ce qui permet à chacun·e d'avoir une version synchronisée et à jour en permanence, même si plusieurs personnes travaillent en même temps sur le projet.
2. Git est *décentralisé*. Cela signifie qu'il n'y a pas nécessairement besoin d'un serveur central unique avec lequel chacun·e doit se synchroniser. À la place, chaque copie du projet contient localement l'ensemble de l'historique des versions. Chaque copie du projet est ce qu'on appelle un *dépôt* Git.

   On travaille systématiquement sur un dépôt *local*, et on se synchronise avec des dépôts *distants*, desquels on récupèrent des commits, et auxquels on envoie nos commits.

   Le plus souvent, pour des questions de simplicité, il y a un dépôt distant particulier qui joue le rôle de serveur central et par lequel tou·tes les membres d'un projet passent, mais ce n'est pas obligatoire.
3. Git est pensé pour le développement logiciel. Il est adapté au code source : il gère les différences entre deux versions successives d'un même fichier au niveau des *lignes*. C'est bien de garder cela en tête pour comprendre son comportement.

Exercice 1.

Initialisation d'un dépôt Git.

1. Avant de commencer à utiliser un répertoire comme dépôt, il doit être initialisé. Pour cela il suffit de taper la commande `git init` dans le répertoire.

   → Initialisez un dépôt git dans un répertoire vide.
2. → Quels sont les résultats de la commande ? Que peut-on voir dans le répertoire ?
3. On peut utiliser la commande `git status` pour voir l'état d'un dépôt Git.

   → Dans quel état est votre dépôt ?

Exercice 2.

Ajout de contenu et premiers commits.

1. Créez un fichier `main.c` dans votre dépôt avec ce contenu :

```c
#include <stdio.h>

int
main (int argc, char *argv[])
{
  printf("Hello, world!\n");
  return 0;
}
```

   → Une fois le fichier `main.c` créé, comment a changé l'état de votre dépôt ?
2. Pour suivre un fichier, on utilise la commande `git add` *`fichier`*. Cela ajoute le fichier à la *zone de validation*.

   → Suivez le fichier `main.c` dans votre dépôt. Quel est maintenant l'état de votre dépôt ?
3. Pour valider des modifications, on utilise la commande `git commit`. On doit lui donner un message qui décrit les modifications avec l'argument `-m` (abréviation de `--message`), par exemple `git commit -m "mon premier commit"`.

   → Validez l'ajout du fichier `main.c` dans votre dépôt. Que constatez-vous ?

4. → Pour configurer Git et lui dire quel est votre nom et votre adresse email, utilisez les commandes suivantes :
   `git config --global user.name "Votre Nom"`
   `git config --global user.email "votre@email"`.
5. → Que est l'état de votre dépôt Git maintenant ?
6. Créez rapidement deux fichiers texte `foo.txt` et `bar.txt` (avec une dizaine de lignes de quelques mots dans chacun, mais n'y passez pas de temps).
   → Quel est l'état de votre dépôt git maintenant ?
7. → Suivez les fichiers texte d'un seul coup avec la commande `git add *.txt`.
8. → Validez le suivi des deux fichiers texte avec `git commit -m "ajouts des fichiers texte"`.
9. Compilez le programme avec `gcc main.c` pour créer l'exécutable `a.out`.
   → Quel est l'état de votre dépôt git ?
10. Dans le dépôt, on veut suivre les versions des fichiers sources mais pas de l'exécutable.
    → Pourquoi ?
11. Git permet de lui donner une liste de fichiers à ignorer, il faut pour cela mettre leur nom (par exemple `a.out`) ou motif (par exemple `*.out`) dans un fichier appelé `.gitignore` à la racine du dépôt.
    → Dites à Git d'ignorer `a.out` en créant un fichier `.gitignore`, puis vérifiez l'état de votre dépôt pour voir si cela fonctionne bien.
12. → Dites maintenant à Git de suivre le fichier `.gitignore`, puis validez cette modification.

Exercice 3.

Modifications.

1. Modifiez un peu deux ou trois lignes vers le début du fichier `foo.txt` et ajoutez lui quelques lignes à la fin. Ensuite remplacez le "Hello, world !" dans le fichier C par "Bonjour tout le monde !".
   → Dans quel état est votre dépôt Git ?
2. Git vous permet de visualiser les modifications entre l'état actuel des fichiers dans le dossier et celui que lui a enregistré (soit validés, soit déjà dans la zone de validation). On utilise pour cela la commande `git diff`.
   → Visualisez les modifications que vous avez faites à la question précédente.
3. Comme vous le voyez, les modifications sont présentées ligne par ligne car c'est souvent le plus pratique avec du code. Il est tout de même possible de lui demander de visualiser les modifications mot par mot avec l'option `--color-words`.
   → Visualisez mot par mot les modifications que vous avez faites à la question 1.
4. Il est important quand on gère du code source de ne pas se servir du gestionnaire de version comme d'un simple système de sauvegarde. On dit au contraire que l'historique du code doit être *sémantique*, c'est à dire que les commits doivent avoir un sens :
   — "Correction du bug #42" est un bon message de commit.
   — "Tout mon travail du week-end en vrac" n'est pas un bon message de commit.
   On va donc valider nos changements en deux fois (en imaginant que la traduction du programme C est une modification indépendante de celles qu'on a faites dans le fichier `foo.txt`).
   Pour ajouter des modifications à la zone de validation, on utilise aussi `git add`.
   → Dites à git de mettre dans la zone de validation le fichier `main.c`. Quel est le nouvel état du dépôt ?
5. → Validez la modification du fichier `main.c` avec le message "traduction en français".
6. → Mettez le fichier `foo.txt` dans la zone de validation avec `git add`, puis vérifiez l'état de votre dépôt.
7. Oups, en fait non, on ne veut pas le mettre d'un seul coup ! Git nous permet de retirer un fichier de la zone de validation en utilisant `git reset fichier`.
   → Retirez le fichier `foo.txt` de la zone de validation, puis vérifiez l'état de votre dépôt.
8. Parfois on a fait plusieurs modifications pour des raisons différentes au même fichier sans les avoir validées au fur et à mesure. On va imaginer que c'est le cas de notre fichier `foo.txt` : les quelques lignes modifiées vers le début du fichier le sont pour corriger le bug #42 alors que les quelques lignes ajoutées à la fin du fichier implémentent la *feature request* #13.
   On va donc devoir faire deux commits. Git nous permet cela grâce à la commande `git add -p` (abréviation de `--patch`), qui vous proposera quelles modifications vous voulez ou non ajouter dans la zone de validation.
   → Ajoutez les modifications qui concernent le bug #42 dans la zone de validation, puis vérifiez l'état du dépôt.
9. Avant de valider les modifications qui concernent le bug #42, vous pouvez vérifier que les modifications restantes correspondent bien seulement à l'implémentation de la feature request #13 avec `git diff`.
   → Si tout va bien, validez les modifications avec un message de commit approprié.

10. Comme on est maintenant sûr qu'il ne reste que les modifications qui correspondent à l'implémentation de la feature request #13, on peut les ajouter d'un seul coup à la zone de validation puis les valider avec la commande `git commit -a -m "message"` (le `-a` est une abréviation de `--all`, qui dit a Git de faire automatiquement un `add` sur tous les fichiers suivis qui ne sont pas encore dans la zone de validation avant d'effectuer le `commit`).

    → Validez ces modifications avec un message de commit approprié.

Exercice 4.

Historique.

1. Pour visualiser l'historique de votre dépôt, il faut utiliser la commande `git log`.

    → Décrivez les informations que vous voyez dans l'historique.

2. On a parfois besoin de voir aussi quels sont les fichiers qui ont été ajoutés ou modifiés par un commit. On peut pour cela utiliser l'option `--name-status`.

    On a parfois au contraire besoin de regarder rapidement l'historique sans trop d'information, juste les messages de commit. On peut le faire avec l'option `--oneline`.

    → Essayez l'une et l'autre de ces options, puis les deux ensembles pour voir les différences.

3. On peut aussi vouloir limiter le nombre de commits affichés, on peut le faire avec l'option `-`n (en remplaçant n par le nombre souhaité de commits).

    → Affichez seulement le dernier commit.

4. Oups !! On a oublié dans le dernier commit de mettre une modification dans le fichier `bar.txt` dans lequel vous devez rajouter une ligne disant "feature #13 ok" à la fin du fichier.

    Heureusement, Git nous permet de modifier le dernier commit quand on est tête en l'air, avec `git commit --amend`.

    Attetion, il se peut que Git vous ouvre un éditeur de texte pour vous proposer de modifier le message de commit, dans ce cas vous pouvez y apporter les modifications souhaitées (peut-être aucune), puis sauvegarder et quitter l'éditeur, ce qui permettra à Git de continuer ses opérations en prenant en compte vos éventuelles modifications.

    → Ajoutez la ligne dans `bar.txt`, puis ajoutez ce fichier dans la zone de validation, pour enfin le rajouter au dernier commit, qui concerne la feature request #13. Que constatez-vous dans l'historique ?

5. On pourrait aussi imaginer que vous avez besoin de revoir la version anglaise du programme `main.c`.

    → Trouvez le nom du commit précédent celui où vous l'avez traduit en français.

6. On peut remettre le fichier dans l'état où il était à ce commit grâce à la commande `git checkout commit fichier`.

    → Remettez le fichier `main.c` en version anglaise. Quel est alors l'état du dépôt Git ?

7. Compilez et exécutez `main.c`. C'est bon, vous avez pu voir ce que vous vouliez voir, on va maintenant revenir dans l'état "normal". Il y a un alias pour le dernier commit de la branche courante, c'est `HEAD`.

    → Revenez à l'état actuel du fichier avec `git checkout HEAD main.c`. Quel est alors l'état du dépôt Git ?

Exercice 5.

Travail collaboratif.

1. Un des principaux points forts de git est de faciliter le travail collaboratif.

    Git permet de récupérer (*pull*) des commits d'un dépôt distant, et de pousser (*push*) des commits vers celui-ci. Bien sûr il faut que les dépôts soient compatibles, c'est à dire qu'ils correspondent au même historique.

    Comme expliqué en 0.2, on a souvent un dépôt distant qui joue le rôle de serveur central avec lequel chacun·e des membres du projet va se synchroniser.

    Dans le cas le plus courant, ce dépôt particulier est géré par une *forge*, c'est à dire une plateforme telle qu'une instance de GitLab (ou GitHub, ou BitBucket, etc.).

    Pour la suite de l'exercice, vous devez vous mettre en groupe d'au moins deux personnes. Choisissez l'un·e des membres du groupe qui jouera le rôle de l'initiateurice du projet.

    → Sur la forge, l'initiateurice du projet va créer un nouveau projet (vous pouvez suivre la démarche ensemble, mais seul·e l'initiateurice la réalise).

2. Tant que le dépôt sur la forge sera vide, celle-ci vous indiquera probablement la marche à suivre pour y pousser votre projet. C'est ce que l'initiateurice va faire (à nouveau, les autres membres du groupe suivent la démarche, mais ne la réalise pas sur leur machine) :

(a) On commence par indiquer à notre dépôt Git local qu'il existe un dépôt distant (*remote*) avec la commande `git remote add origin adresse`.

Le nom *origin* est le nom que l'on donne par défaut au dépôt qui joue le rôle de serveur central, mais vous pourriez en choisir un autre.

L'*adresse* vous est donnée par la forge, généralement vous avez le choix entre une adresse SSH et une adresse HTTPS (si vous ne savez pas de quoi il s'agit, choisissez HTTPS, sinon SSH).

→ Indiquez à votre dépôt local l'existence du nouveau dépôt distant.

(b) Maintenant, on peut pousser l'historique du dépôt local vers le dépôt distant, qui est forcément compatible puisqu'il est encore vide. Pour cela on utilise la commande `git push`. La première fois, on va lui passer quelques arguments supplémentaires : `-u` (abréviation de `--set-upstream`) et `origin master` pour lui signifier que dorénavant, le remote "origin" sera celui par défaut et que la *branche* "master" sera celle qu'on pousse par défaut sur la branche distante "master" qui sera créée au passage sur le dépôt distant "origin" (on verra au prochain exercice ce que sont les branches).

→ Poussez l'historique de votre dépôt local sur la forge.

(c) Si tout s'est bien passé, vous devriez pouvoir voir vos fichiers et votre historique sur la forge.

→ Vérifiez que c'est bien le cas.

3. Les autres membres du groupe doivent maintenant participer au développement du projet. Dans un nouveau répertoire, illes vont *cloner* le projet avec la commande `git clone adresse`. Cela va créer un nouveau dépôt local qui sera une copie exact du dépôt distant, qui sera lui automatiquement mis comme "origin".

   Attention, cela doit bien être fait *en dehors de votre dépôt* car `git clone` en crée un nouveau (vous pouvez complètement mettre de côté ce que vous avez fait jusqu'à présent).

   (a) → Sur les machines locales des autres membres du groupe, clonez le dépôt distant.

   (b) → Vérifiez l'état et l'historique du dépôt nouvellement créée.

4. Vous avez normalement accès en lecture au dépôt de l'initiateurice du projet, mais pas en écriture (c'est à dire que vous ne pourrez pas pousser des nouveaux commits sur son dépôt distant).

   Pour cela vous avez deux solutions :
   — soit vous *forkez* sont dépôt, c'est à dire que vous en créée une copie suivant la même procédure que lui sur votre compte (mais en l'appelant autrement que "origin" pour pouvoir différencier vos deux remotes), et ille fait de même pour que vous puissiez chacun·e pousser des commits vers votre copie distante et récupérer des commits depuis le sien ;
   — soit l'initiateurice vous rajoute comme membre du projet sur la forge et vous donne le droit d'écriture sur le dépôt (le rôle de "Développeur" sur GitLab).

   La première approche, même si elle fonctionne, devient vite compliquée quand le nombre de membre du groupe augmente, du coup, on va privilégier la seconde.

   → L'initiateurice du projet rajoute les autres membres du projet comme contributeurices sur la forge.

5. → Pour vérifier que ça a bien fonctionné, chaque membre du groupe, sur sa copie locale du projet, crée un nouveau fichier à son nom, le fait suivre par le dépôt puis valide cette modification.

   → Pour partager vos modifications avec les autres, vous allez maintenant utiliser `git push`.

6. C'est toujours mieux avant de pousser ses propres modifications, et même avant de commencer à travailler, de mettre à jour le contenu de son dépôt local, cela évitera de potentiels conflits de modifications.

   → Pour récupérer les modifications des autres, utilisez `git pull`. Regardez l'évolution de l'historique de votre dépôt.

7. Mettez-vous d'accord pour créer un conflit, par exemple deux membres du groupe essayent de modifier le même fichier chacun·e de leur côté en même temps, puis l'un·e pousse sa modification, et l'autre la récupère.

   → Comment Git vous signale le conflit ? Ouvrez le fichier concerné, que constatez-vous ?

8. → Éditez le fichier pour le remettre dans le bon état (ce qui est un choix arbitraire qu'il vous incombe de faire) puis validez les modifications pour résoudre le conflit.

9. → Une fois le conflit résolu, vous pouvez poussez la version fusionnée du projet sur le dépôt distant.

10. Vous pouvez visualiser l'historique de votre dépôt et voir visuellement la fusion avec l'option `--graph` de `git log` (ça se combine bien avec `--oneline`).

    → Visualisez et essayez de comprendre ce que Git vous montre.

Exercice 6.

Branches.

Pour cet exercice c'est mieux que tout le groupe travaille ensemble sur la machine de l'un·e des membres.

1. Git permet très facilement de créer des *branches*. On peut les voir comme des univers parallèles où le code évolue de façon différente.

   C'est par exemple une bonne pratique de garder sa branche "master" (celle par défaut) propre, et de faire son développement dans d'autre branche pour ne pas que le code dans la branche "master" se retrouve dans un état instable.

   → Listez les branches existantes dans votre dépôt avec `git branch`.

2. On veut maintenant rajouter une fonctionnalité à notre programme C. Plutôt que de lui faire dire "Bonjour tout le monde !", on veut qu'il salue la personne passée en argument sur la ligne de commande lorsqu'on l'appelle. On va dire que c'est notre feature request #14.

   Créer une nouvelle branche se fait avec la commande `git branch` *nom*.

   → Créez une branche "salut-perso". Quel est l'état de votre dépôt git ?

3. → Basculez sur la branche que vous venez de créer avec `git checkout salut-perso`

4. On va commencer à implémenter la feature.

   (a) Modifiez le fichier `main.c` pour que la ligne 6 soit :

   ```
   1  printf("Bonjour %s !\n", argv[1]);
   ```

   (b) Compilez le avec `gcc main.c`.

   (c) Testez le avec `./a.out Sam`.

   → Validez la modification avec `git commit main.c -m "salut personnalisé"`.

5. Si on s'arrête là, notre programme pourrait avoir un comportement non désiré si on ne lui passe aucun argument. On va donc finir d'implémenter proprement notre fonctionnalité avant de la rendre publique.

   (a) Modifiez le fichier `main.c` pour avoir dans le main :

   ```
   1  if (argc != 2) {
   2    fprintf(stderr, "Usage: %s <nom>\n", argv[0]);
   3    return 1;
   4  }
   5  printf("Bonjour %s !\n", argv[1]);
   6  return 0;
   ```

   (b) Compilez le avec `gcc main.c`.

   (c) Testez le avec `./a.out Sam` et `./a.out`.

   → Validez la modification avec `git commit main.c -m "gestion de l'erreur en cas d'absence de nom"`.

6. Maintenant on estime que le programme est prêt, alors vous poussez votre branche "salut-perso" pour qu'elle soit vu par les autres membres du projet et qu'illes vous donne leur avis avant de la mettre dans "master". Comme pour la branche "master", on va demander à Git de se rappeler que la branche distante nouvellement créée sera celle à synchroniser par défaut avec la branche locale "salut-perso" (op)

   → Poussez votre branche de développement avec `git push -u origin salut-perso`.

7. Pendant ce temps, la branche "master" n'a pas bougée. Vous pouvez y retourner à tout moment avec `git checkout master` et constater que l'état de vos fichiers et de l'historique correspond bien à celui dans lequel vous avez laissé la branche "master".

   Vous pouvez même travailler en parallèle sur différentes branches si vous le souhaitez, mais attention au conflit quand vous chercherez à fusionner vos différents travaux (si cela doit arriver).

   → Vérifiez que le fichier `main.c` est dans l'état original quand vous basculez sur la branche "master", et qu'il revient dans son nouvel état quand vous rebasculez sur la branche "salut-perso".

8. Pendant ce temps, les autres membres du projet voient le nouveau code dans la branche "salut-perso" et sont d'accord qu'il est bon, mais l'un·e d'entre elleux se plaint tout de même du cassage de la rétrocompatibilité avec les versions précédentes du programme (et ille a raison !). Cette personne crée donc un nouveau rapport de bug, le #43.

   Du coup, un·e des autres membres du groupe va corriger le bug (déplacez-vous sur une autre machine) :

   (a) Il va d'abord falloir récupérer la nouvelle branche "salut-perso" localement. Pour cela, il faut utiliser la commande `git checkout -b salut-perso origin/salut-perso`, qui a pour effet de créer une nouvelle branche locale (`-b`) "salut-perso" à partir de la branche distante du même nom sur le remote "origin", et de basculer dessus au passage.

   (b) Modifiez le fichier `main.c` pour avoir dans le main :

```c
char *name;
if (argc < 2) {
  name = "tout le monde";
}
else {
  name = argv[1];
}
printf("Bonjour %s !\n", name);
return 0;
```

(c) Compilez le avec `gcc main.c`.

(d) Testez le avec `./a.out Sam` et `./a.out`.

→ Validez la modification avec le message de commit “correction du bug #43”, puis pousser là sur la branche “salut-perso” de dépôt distant “origin” pour que les autres membres du projet puissent y accéder.

9. La nouvelle fonctionnalité est maintenant prête, elle a été testée et validée par plusieurs personnes.
   L’un·e des membres du groupe va donc la fusionner avec la branche stable du projet, “master”.

   (a) → Sur l’une des machines du groupe, vérifiez que la branche “salut-perso” est à jour en basculant dessus et en consultant l’historique.

   (b) → Si la branche n’est pas encore présente sur cette machine, suivez les explications de la questions précédentes pour la récupérer.

   (c) → Si la branche est présente mais pas à jour, vous pouvez récupérer le contenu de cette branche spécifiquement avec la commande `git pull`.

   (d) → Maintenant que la branche est à jour, vous rebasculez sur la branche “master”.

   (e) → Une fois dans la branche “master”, importez la nouvelle fonctionnalité avec la commande `git merge salut-perso`. Normalement, tout devrait bien se passer, mais si vous aviez des modifications incompatibles dans la branche “master”, il pourrait y avoir des conflits à gérer.

10. Maintenant que l’ensemble de la branche “salut-perso” a été fusionnée dans “master”, on pourrait vouloir supprimer cette branche. On peut le faire avec la commande `git branch -D salut-perso`.

Exercice 7.

Historique amélioré et alias.

1. On a déjà vu comment visualiser l’historique de Git avec `git log`.
   Voici quelques astuces pour avoir un log plus sympa et informatif :
   — `git log --oneline`
   — `git log --oneline --decorate`
   — `git log --oneline --decorate --graph`
   — `git log --oneline --decorate --graph --all`
   → Qu’apporte chacun de ces arguments ?

2. Vous pouvez définir des alias pour les commandes que vous faites souvent.
   Par exemple :
   — `git config --global alias.s "status"`
   — `git config --global alias.ci "commit"`
   — `git config --global alias.lol "log --oneline --decorate --graph --all"`
   Cela vous permettra d’utiliser `git s` à la place de `git status` par exemple.