Agilité : Introduction à l'outil Git

TP1 Agilité

Introduction à l'outil Git

Jérôme Buisine
jerome.buisine@univ-littoral.fr

3 septembre 2021

Durée : 3 heures

L'objectif de ce TP est la prise en main de l'outil de versioning `Git`.

1 Travail

Ce support vous offre dans un premier temps un rappel du besoin ayant amené au développement d'un tel outil et de son principe d'utilisation. Ensuite, il vous sera proposé un travail collaboratif pour vous permettre de manipuler `Git`. En effet, il est important de noter ici que `Git` est un outil à des fins collaboratives afin de mener à bien n'importe quel projet.

Voici un ensemble de ressources qui peuvent vous être utiles :

— 1. Documentation officielle de l'outil `Git` ;

— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git` ;

2 Qu'est-ce que `Git` ?

Avant de détailler comment fonctionne cet outil, revenons à un problème que vous avez déjà du rencontrer une fois dans votre vie.

2.1 Gestion de version

Généralement dans le cadre d'un développement ou simplement d'une rédaction (je pense que vous avez dû tous au moins passer par cette étape), vous souhaitez réaliser des sauvegardes intermédiaires. Généralement, dans le cadre de la rédaction d'un mémoire, voici à quoi pourrait ressembler votre historique de version de fichier :

```
Mémoire_v1.doc
Mémoire_v2.doc
Mémoire_v3.doc
Mémoire_Final.doc
Mémoire_Final_v2.doc
Mémoire_Final_Final.doc
...
Mémoire_Final_pour_de_vrai.doc
```

Il faut avouer que ce n'est pas le plus pratique... La première remarque ici à faire, est que l'on pourrait déjà envisager une nette amélioration de notre système de version de fichier. On peut en effet y introduction des niveaux de versions : mineures, intermédiaires, majeures. Voici un autre exemple, avec cette nouvelle approche où `v1.0.0` représente une version majeure :

```
Mémoire_v0.0.1.doc
Mémoire_v0.0.2.doc
Mémoire_v0.1.0.doc
Mémoire_v0.2.0.doc
Mémoire_v1.0.0.doc
Mémoire_v1.0.1.doc
...
Mémoire_v2.0.0.doc
```

C'est déjà un peu plus compréhensible et le suivi est quand plus facile et mieux organisé. Toutefois, un nouveau problème peut-être soulevé. Comment maintenir les mises à jour d'un tel fichier si l'on travaille en binôme, voire 3, 4, ou encore, 5 personnes ? C'est ici que tout devient d'un coup plus complexe. Imaginez un peu comment vous allez devoir gérer la fusion des documents modifiés par chacun des membres du groupe. Qui a travaillé sur quelle version ? Quelle est réellement la dernière version ?

FIGURE 1: Visage de la personne désignée volontaire pour la fusion du document

Plus sérieusement, on peut maintenant exploiter des outils collaboratifs permettant d'éditer en ligne et en parallèle le même fichier aisément. Bien que spécifique à l'édition de fichier de texte, le mot clé ici, est **collaboratif**. Qu'en est-il donc des outils pour les développeurs visant à faciliter le développement de projet ?

2.2 Vers des outils collaboratifs

Il existe plusieurs outils collaboratifs pour le développement et versioning de projet. Le plus connu étant `Git` mais l'on peut également citer Apache Subversion (SVN) sorti en 2004. De son côté, `Git`, est un logiciel développé par Linus Thorvald (connu également pour avoir créé en 1991 à 21 ans, le noyau Linux) créé en 2008 pour le versioning des systèmes UNIX. Chacun ayant des similarités et des différences, `git` possède la particularité d'être décentralisé, ce qui rend possible le travail sans être connecté au serveur distant. Nous ne rentrerons pas plus dans le détail ici, mais si vous êtes curieux, voici le lien d'un article mettant plus avant les différences.

2.2.1 Particularité de `Git`

Git est un système de contrôle de version décentralisé. Chaque participant possède un clone de l'ensemble du référentiel en local. Il est utilisé pour ajouter, contribuer et suivre les changements dans le code source.

Un autre point très important est que la plupart des opérations ne nécessitent pas de connexion réseau, car elles ne travaillent que sur votre clone du référentiel. C'est-à-dire que vous pouvez travailler hors-ligne sur votre projet local et proposer une mise à jour du projet plus tard avec vos modifications sur le serveur hébergeant le projet.

L'exemple de collaboration classique d'échanges entre développeurs et le projet distant est représenté par la figure 2).

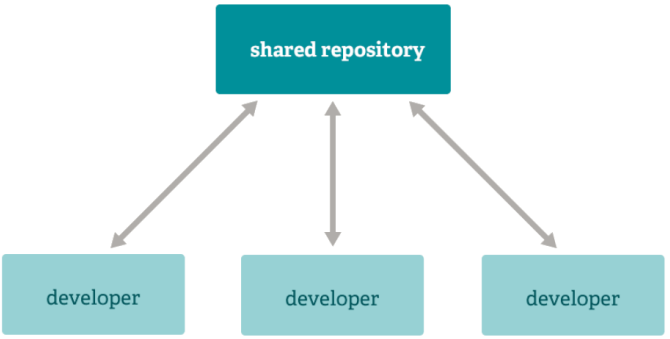


FIGURE 2: Échanges bilatéraux de développeurs (avec chacun un projet local) et le projet distant (projet centralisé), aussi appelé projet commun (source : git-scm).

Généralement, chaque développeur `clone` (crée une copie) localement le projet distant. Ils peuvent avancer localement, ajouter des modifications locales, se mettre à jour en récupérant les modifications des autres développeurs, puis les soumettre.

On peut imaginer toutefois plusieurs manières de collaborer avec cette notion de répertoire projet distant et local. Cela peut être généralement le cas en entreprise de posséder un ou plusieurs projets distant pour une meilleure fluidité et suivi. Le site officiel de `Git` illustre quelques représentations.

2.3 Proposition de modifications

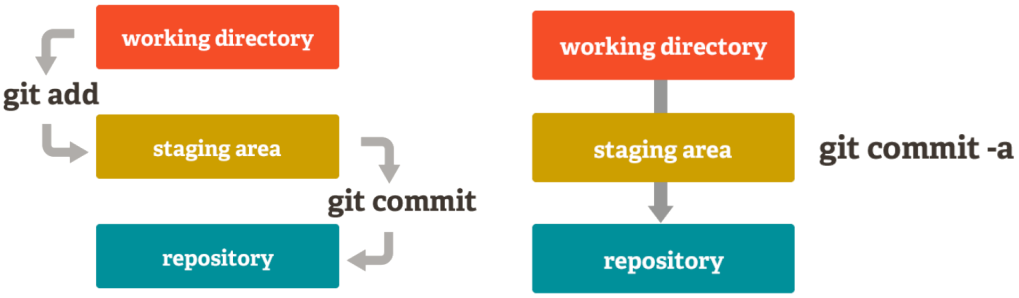
`Git` propose une interface de commande que nous utiliserons par la suite. Lorsque des modifications ont été réalisées sur un ou plusieurs fichiers du projet, il est alors possible de les ajouter pour le prochain `commit`. Un quoi, un `commit` ? Un `commit` est un état où les modifications apportées sur les fichiers du projet sont actées (une validation). Les fichiers modifiés sont donc liés à ce `commit`. À noter qu'un `commit` possède un numéro d'identification unique (*hash* ou encore clé de *hashage*).

La figure 3 illustre l'utilisation des commandes `Git` pour ajout de contenu et passage à un état de validation.

2.4 Gestion des versions

Les modifications apportées au logiciel sont communément rassemblées et désignées par des numéros de version sous la forme 'X.Y.Z'. On formule ainsi des degrés d'importance de ces changements de gauche à droite, du plus significatif au moindre correctif :

- X – Majeur : suppression d'une fonctionnalité obsolète, modification d'interfaces, renommages...



(a) Ajout de modifications à l'index en utilisant le contenu actuel trouvé puis validation des modifications à l'aide d'un `commit`.

(b) Ajout de toutes les modifications courantes à l'index avec validation directe des modifications à l'aide d'un `commit`.

FIGURE 3: Différentes commandes `Git` pour ajout et validation de contenu de fichiers (source : git-scm).

- Y – Mineur : introduction de nouvelles fonctionnalités, fonctionnalité marquée comme obsolète...
- Z – Correctif : modification/correction d'un comportement interne, failles de sécurité...

Nous détaillerons dans la suite les commandes `git` à utiliser pour générer une livraison d'un projet.

2.5 Gestion des branches

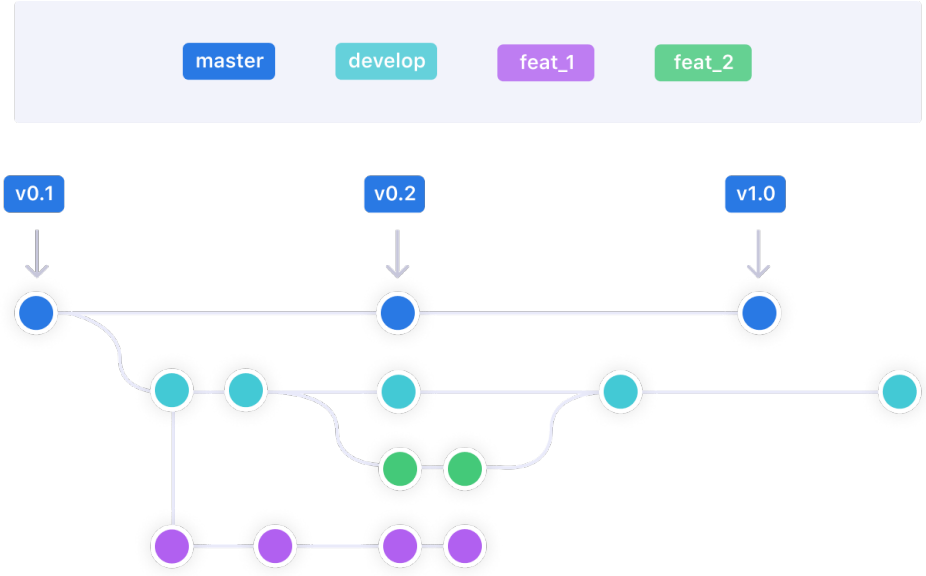


FIGURE 4: Exemple de gestion des branches sous `git` (source : qovery.com).

La notion de branches sous `git` est très importante. Elles permettent notamment une collaboration plus efficace et séparée pour chacun des développeurs. Mais au finale qu'est-ce que branche ? Il faut savoir que l'historique de `git`, c'est-à-dire l'ensemble des `commits` (validations) peut être interprété comme étant un arbre. Un arbre sans branche, ce n'est pas envisageable, nous sommes tous d'accord là-dessus ? La notion de branche est alors utilisée pour préciser une nouvelle vie possible du projet à partir d'un point de validation (`commit` en particulier) d'une autre branche. La branche initiale est généralement la branche principale.

Une branche est ainsi créée par un développeur à partir d'un état (généralement stable) du projet. Il y ajoute ses modifications liées à la fonctionnalité qu'il doit développer. Enfin, une fois la fonctionnalité finie,

il peut procéder à la fusion de ses modifications sur la branche initiale. Sa branche peut être supprimée ou non, mais quoi qu'il soit, l'ancienne est complétée avec de nouvelles modifications.

La figure 4 montre un aperçu d'un projet qui est composé de plusieurs branches. Le projet est actuellement composé de 4 branches, la branche principale *master*, celle de développement *develop* et deux autres liées à des fonctionnalités développées *feat_X*. La branche *develop* basée sur une version stable de *master* est celle permettant de récupérer les modifications des branches *feat_X*. Les branches *feat_X* quant à elles, se basent sur une version de la branche *develop*. Il est donc possible pour un développeur de travailler en parallèle sur une branche, soit sur une version stable du projet sans impacter directement les autres. Ainsi, uniquement la fusion de branche (voir exemple *feat_2*) impliquera une vérification des modifications à ajouter.

2.6 Principales commandes `Git`

Voici une liste non exhaustive des commandes de `Git`. Vous pouvez en prendre connaissance puis les utiliser dans la seconde partie de ce TP. Ses commandes sont à utiliser au d'un terminal et d'un dossier versionné par `git`.

Initialisation :

```
# initialisation d'un dossier comme étant un projet git
# création d'un dossier caché .git pour la gestion de version du dossier
git init

# clone un projet distant localement
git clone <project-url> <nom-projet>

# À ne pas oublier !
git --help
```

Ajout de modifications :

```
# Montre le status des fichiers versionnés ou non versionnés
git status

# Ajoute le fichier ‘file.txt‘ pour un prochain commit
git add file.txt

# Ajoute tous les fichiers pour un prochain commit
git add .

# Ne prend plus en considération un fichier pour le prochain commit
git reset file.txt

# Crée un commit avec les fichiers ‘ajoutés‘
git commit -m "mon premier commit"

# Supprimer le dernier commit local
git reset --soft HEAD^

# Supprimer les 2 derniers commits locaux
git reset --soft HEAD~2

# Visualisation complète des différents commits de la branche courante
git log

# Visualisation condensée des différents commits de la branche courante
git log --oneline
```

Gestion des branches :

```
# Liste toutes les branches disponibles
git branch

# Crée une nouvelle branche
git branch ma-branche

# Change de branche de développement
git checkout ma-branche

# Création et changement de branche courante
git checkout -b ma-branche

# Supprime une branche
git branch -d ma-branche

# Depuis la branche ‘develop‘, récupération des modifications de ‘ma-branche‘
# Plus évident à manipuler que rebase
git merge ma-branche

# Depuis la branche ‘develop‘, récupération des modifications de ‘ma-branche‘
# À la différence de merge l’historique complet est conservée
git rebase ma-branche
```

Interaction avec le serveur d’hébergement :

```
# Énumère les serveurs d’hébergements référencés
git remote -v

# Ajouter un serveur distant
git remote add <remote-name> XXXXXXX.git

# Modifier un serveur distant
git remote set-url <remote-name> XXXXXXX.git

# Récupère et applique les modifications du serveur d’hébergement
git pull <remote-name> ma-branche

# Prendre connaissance des nouvelles modifications existantes sur le serveur
    mais sans les appliquer
git fetch <remote-name>

# Prendre connaissance des modifications de tous les serveurs
git fetch --all

# Publie les modifications apportés par une branche sur un serveur
# !! toujours réaliser un pull avant de push !!
git push <remote-name> ma-branche

# Suppression du dernier commit soumis au serveur
# !! attention ça peut être dangereux si ce n’est pas le dernier commit !!
git revert <commit-hash>
```

Réaliser une livraison avec version :

```
# Ajout d’un tag de version à l’état actuel du projet
git tag -a v1.0.0 -m "Releasing version v1.0.0"

# Liste l’ensemble des versions du projet
git tag -l
```

```
# Affiche les informations détaillées d'un tag
git show v1.0.0
```

2.7 Fichier .gitignore

Le fichier '.gitignore' permet d'éviter de tracker inutilement et involontairement des fichiers inutiles, compilés (voués a changé régulièrement) ou lourd. En voici un exemple :

```
# ignore le dossier target
target

# ignore tous les fichiers .class
*.class

# ignore toutes les images sauf celles dans le dossier 'resources'
*.png
!resources/*.png
```

2.8 Commandes à ne pas confondre

— **git revert** : crée un nouveau commit qui annule les changements d'un commit précédent ;

— **git checkout** : extrait le contenu du référentiel et le place dans votre arbre de travail (elle permet aussi le déplacement vers une autre branche) ;

— **git reset** : il modifie l'index (la « zone de transit »). Ou elle change le commit sur lequel une tête de branche pointe actuellement. Cette commande peut modifier l'historique existant.

Cas d'utilisations possibles :

— **git revert** : si un commit a été fait quelque part dans l'historique du projet, et que vous décidez plus tard que ce commit est mauvais. L'utilisation de *git revert* annulera les changements introduits par le mauvais commit, en enregistrant le « undo » dans l'historique ;

— **git checkout** : restaure une révision historique d'un fichier donné (`git checkout <file-path> <commit-hash>`) ;

— **git reset** : si vous avez fait un commit, mais que vous ne l'avez pas partagé avec quelqu'un d'autre et que vous décidez que vous n'en voulez pas, alors vous pouvez utiliser git reset pour réécrire l'historique de sorte qu'il semble que vous n'ayez jamais fait ce commit.

3 Projet collaboratif `Git`

Après toute cette partie théorique, nous allons passer aux choses sérieuses ! Vous allez par groupe de 3, développer votre premier (peut-être) projet collaboratif sous `Git`.

Pour cela, vous allez dans un premier temps, télécharger (si ce n'est pas déjà le cas), la version de `Git` adaptée à votre système depuis le site officiel.

3.1 Récupération du projet

Nous allons utiliser Gitlab comme serveur d'hébergement. Il vous faudra créer un compte sur la plateforme. Pour éviter tout problème de récupération de projet par la suite, il vous faudra ajouter votre clé SSH.

Une fois la configuration terminée, l'un des membres de votre projet devra être chargé de réaliser un `fork` du dépôt FightGame. Un `fork` permet de réaliser une copie du projet distant. Le `fork` d'un dépôt vous permet d'expérimenter librement des changements sans affecter le projet original. Le membre de l'équipe ayant réalisé cette copie devra ajouter les deux autres membres de l'équipe au projet en tant qu'administrateur.

3.2 Consigne du projet

Afin de simuler un environnement projet réel d'entreprise, il vous sera demandé durant ce TP une gestion des branches de la manière suivante :

- **`master`** ou **`main`** : branche de production du projet (livrable à fournir pour le client).
- **`develop`** : branche de développement du projet avant livraison.
- **`feature/XXXXX`** : branche liée à un développement en particulier où `XXXXX` est un nom donné à cette branche spécifiquement au développement demandé.

Le processus d'utilisation de `Git` durant le TP sera le suivant :

- 1. Le projet sera composé de 3 grandes étapes qui seront détaillées par la suite ;
- 2. Pour chacune des étapes, 2 **développeurs** effectueront une des deux tâches proposée par l'étape de développement, le 3^e membre de l'équipe effectuera la fusion du travail et délivrera une version du projet. Étant donné que vous êtes 3 membres et que 3 étapes sont demandées, vous jouerez chacun votre tour le rôle du **livreur de projet** ;
- 3. Bien entendu, la personne chargée de la livraison peut aider ses camarades lors du développement et inversement lors de la livraison ;
- 4. Pour chaque nouvelle tâche, chaque **développeur** doit créer une branche 'feature/XXXXX' à partir de la branche 'develop' ;
- 5. Une fois un développement terminé, le **livreur de projet** doit fusionner les modifications de la branche 'feature/XXXXX' d'un des développeurs vers la branche 'develop', vérifie le contenu, puis, supprimer la branche 'feature/XXXXX'.
- 6. Le **livreur de projet** doit ensuite réaliser une livraison du projet en soumettant une version de projet. Pour cela, il doit mettre à jour la branche principale ('master' ou 'main') relativement à la branche 'develop', puis soumettre un 'tag' de version.
- 7. Pour chaque nouvelle étape, les **développeurs** doivent mettre à jour leur projet local.

3.2.1 Développement

Les étapes à développer sont disponibles dans le fichier 'README.md' du projet copié ! Lisez attentivement les instructions !