GIT

bstorm TRAINING

Table des matières

Introduction

Introduction - Historique

GIT est un **logiciel de gestion de versions**, conçu par Linus Torvalds, l'auteur du noyau Linux.
Gratuit et open-source, il est actuellement distribué par Software Freedom Conservancy.

A l'origine, BitKeeper, un autre logiciel de gestion de versions, était utilisé pour la maintenance du noyau Linux. Suite à des désaccords avec les développeurs de ce dernier, Linus ne trouvant aucun logiciel adéquat à ses besoins, préféra créer son propre gestionnaire de version.

21 décembre 2005 :	Première version stable
7 avril 2005 :	Première version publiée
2016 :	GIT devient le logiciel le plus populaire de sa catégorie avec plus de 12 Millions d'utilisateurs.
27 décembre 2020 :	Dernière version - V2.30

Introduction - Qu'est-ce que GIT?

GIT est un logiciel de gestion de versions distribué (qui ne nécessite pas d'utiliser de serveur centralisé).

Ce qui permet à l'utilisateur de gérer l'évolution du contenu de ce qu'il produit, et ainsi mettre en place différentes versions.

Il utilise un système de connexion pair à pair, où chaque entité est à la fois client et serveur.

Introduction - Installer GIT

Pour travailler avec GIT, il faut dans un premier temps l'installer. Il est disponible gratuitement sur le site : https://git-scm.com/

Rendez-vous dans la section Download et choisissez la version selon votre OS.

Une fois téléchargé, exécutez l'installeur.

Ce site procure aussi gratuitement le "Pro Git book" en différents formats.

Introduction - Installer GIT

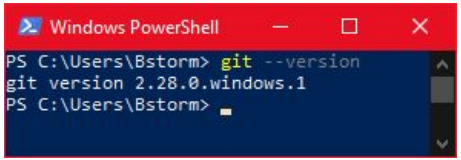
Lors de l'installation (selon la version téléchargée et l'OS), on vous permettra de choisir diverses options tel que :

- Quel éditeur utiliser?
- Permettre la modification du PATH de l'OS ?
  (permet d'utiliser GIT dans l'invite de commande ou juste via le GIT Bash)
- Quelle librairie utiliser pour effectuer des connexions HTTPS?
- Quel est le type de fin de ligne dans nos fichiers texte?
- Quel émulateur de terminal utilisera GIT Bash?
- Quelles sont les fonctionnalités à ajouter ?

Je vous recommande de laisser les options par défaut, elles permettent d'éviter tous conflits avec des logiciels externes.

Introduction - Installer GIT

Pour vérifier que l'installation c'est déroulé sans accroc, entrez la ligne de commande suivante dans votre PowerShell / Invite de commande / Terminal : **git —version**



Si votre numéro de version apparaît, aucun problème,
GIT s'est installé avec succès!

Une fois installé, GIT doit enregistrer votre identité. Si vous oubliez cette étape, il vous le rappellera régulièrement entre chaque action. Constitué de votre nom, prénom et email, ses données seront ajouter aux diverses méta-données du projet, mettant en place un historique complet au développement.

git config —global user.name "*Nom Prénom***"**

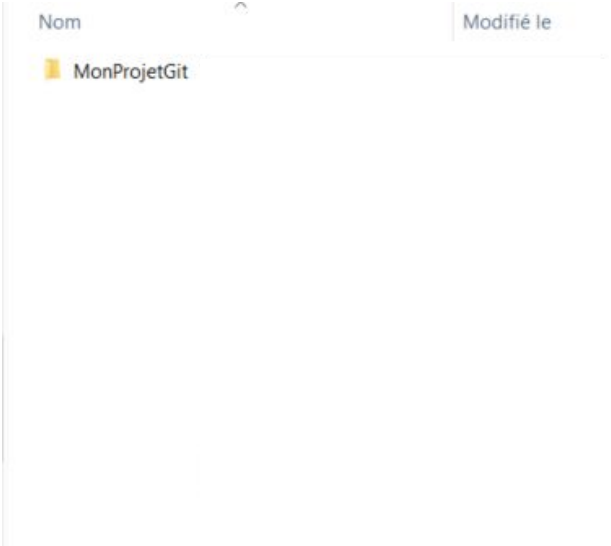
git config —global user.email "*Email***"**

Introduction - Intégrer GIT à son projet

Une fois GIT installé sur votre poste de travail,
il faut l'intégrer à vos projets.

Pour cela, via l'explorateur Windows, faites un clique droit sur le dossier de votre projet, et choisissez l'option "*GIT bash here*".
Une fenêtre ressemblant fort à l'invite de commande s'ouvrira.
Introduisez la ligne de commande : **$ git init**

GIT va placer un dossier caché "**.git**" dans le dossier de votre projet.



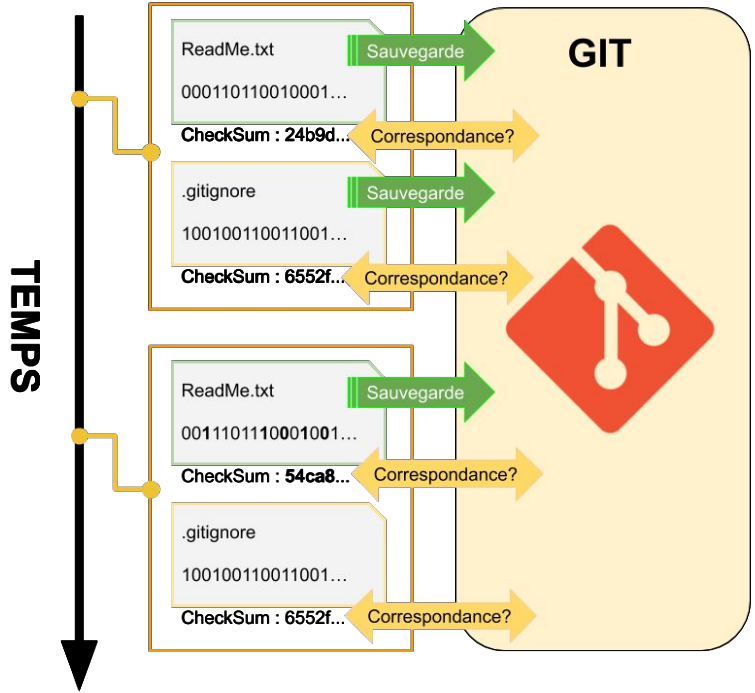
Il est possible de récupérer un projet externe, mais ceci sera expliqué dans un futur chapitre.

Introduction - Comment cela fonctionne?

Une fois GIT intégré à un projet, il sauvegarde toutes modifications effectuées sur celui-ci.

Entre chaque enregistrement de vos fichiers, il les compare avec leur version précédente grâce au CheckSum (un encodage du binaire en SHA-1), et si ceux-ci sont différents, GIT peut les indexer et les lier à un “Commit”.

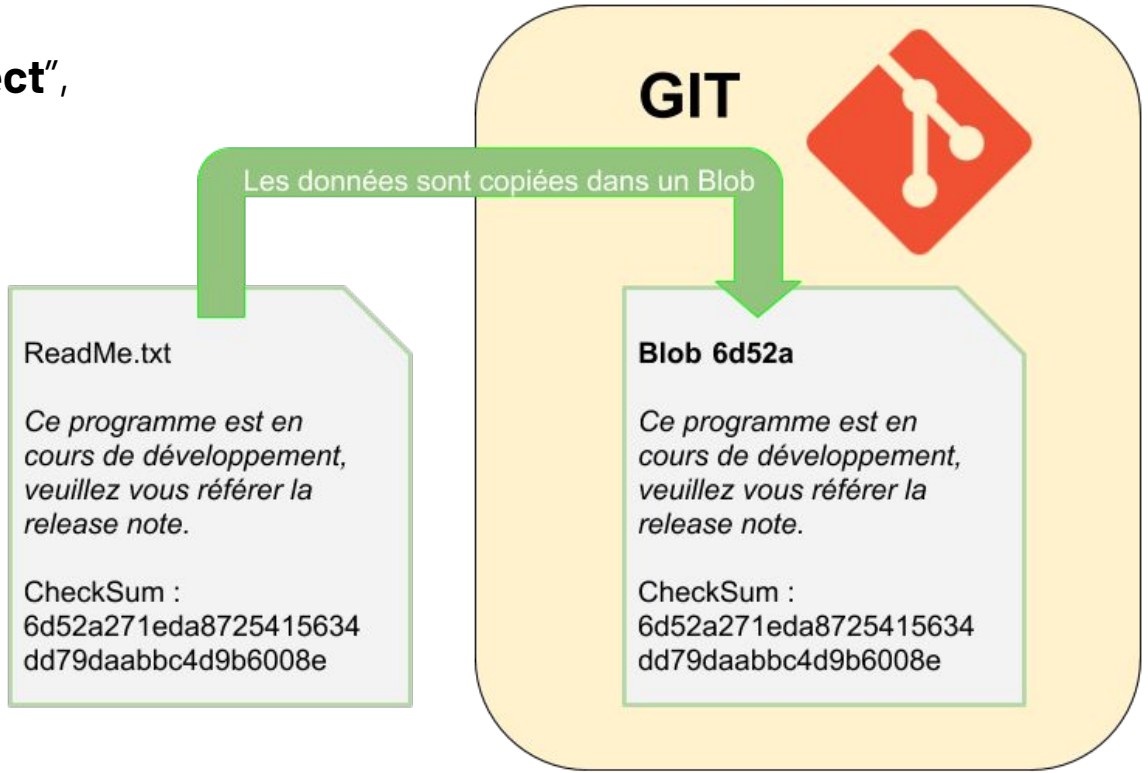
GIT conserve toutes les versions de chaque fichier composant un projet comme des instantanés.



La composition

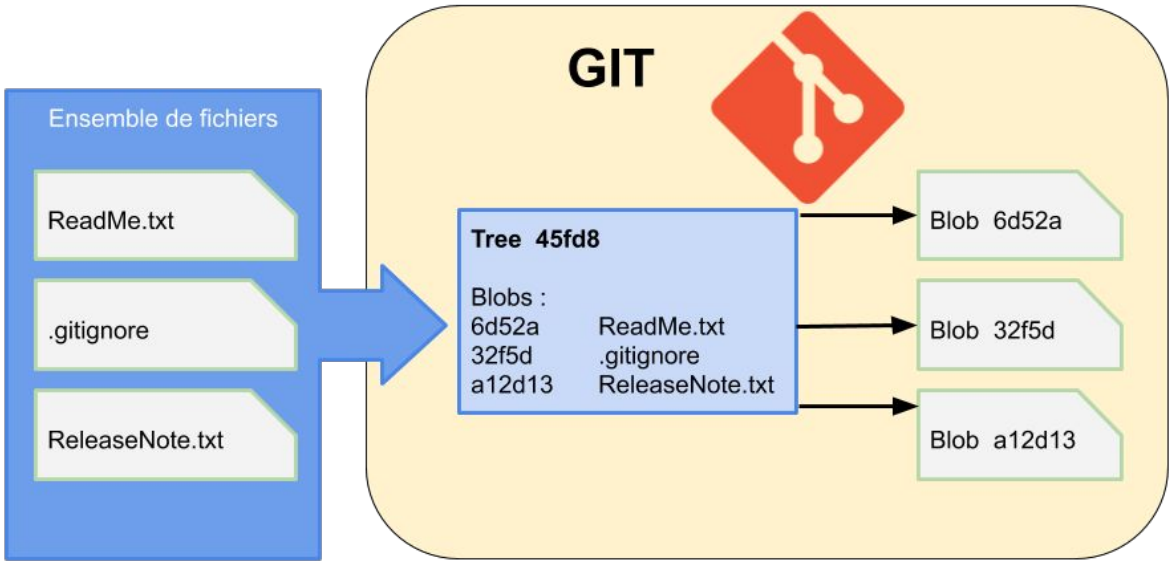
La composition - Le blob

"**Blob**" pour "**Binary Large Object**", sont des données brutes représentant le contenu d'un fichier.



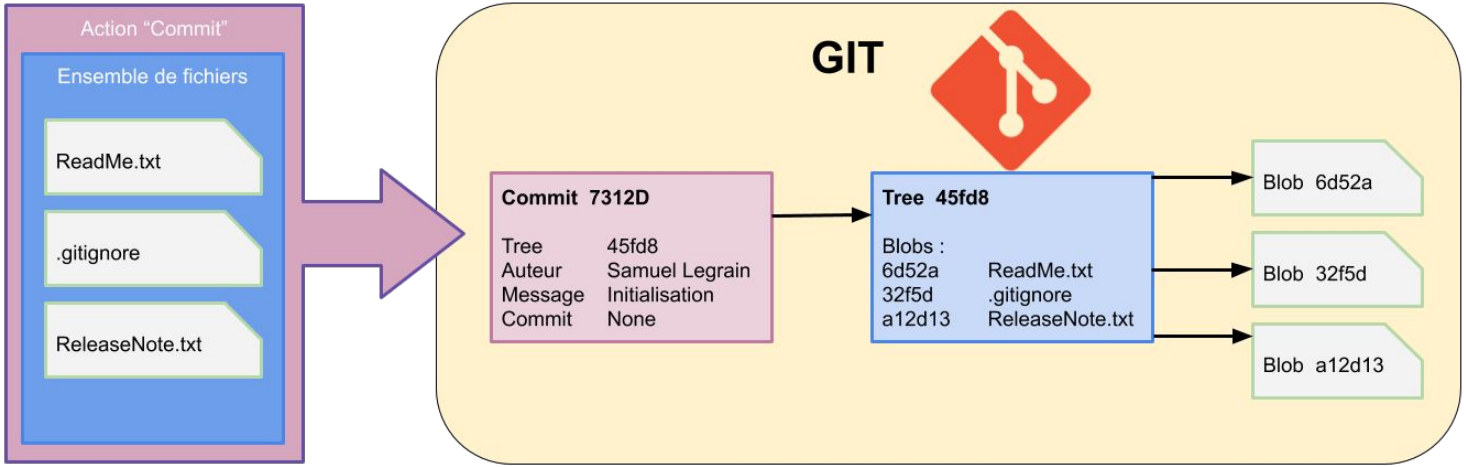
La composition - Le tree

Le "**tree**" est une arborescence de fichiers, c'est une liste de blob avec des informations supplémentaires (nom et permissions). Il peut aussi contenir d'autres trees pour définir des sous-répertoires.



La composition - Le commit

Objet résultant d'une validation, aussi dit "**commit**". Objet pointant un tree, il est composé de métadonnées supplémentaires (description, auteur, message, ...).
A part pour le premier commit, ils se lient à un ou plusieurs commits parents formant l'historique du projet.

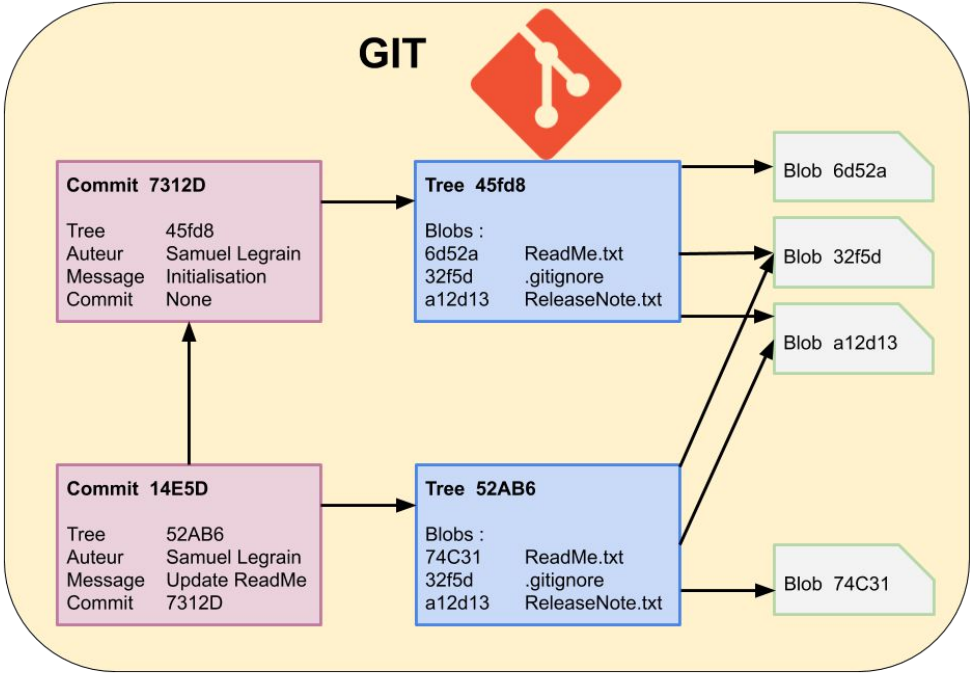


La composition - Le commit

Dans cet exemple, notre projet a obtenu deux commits :

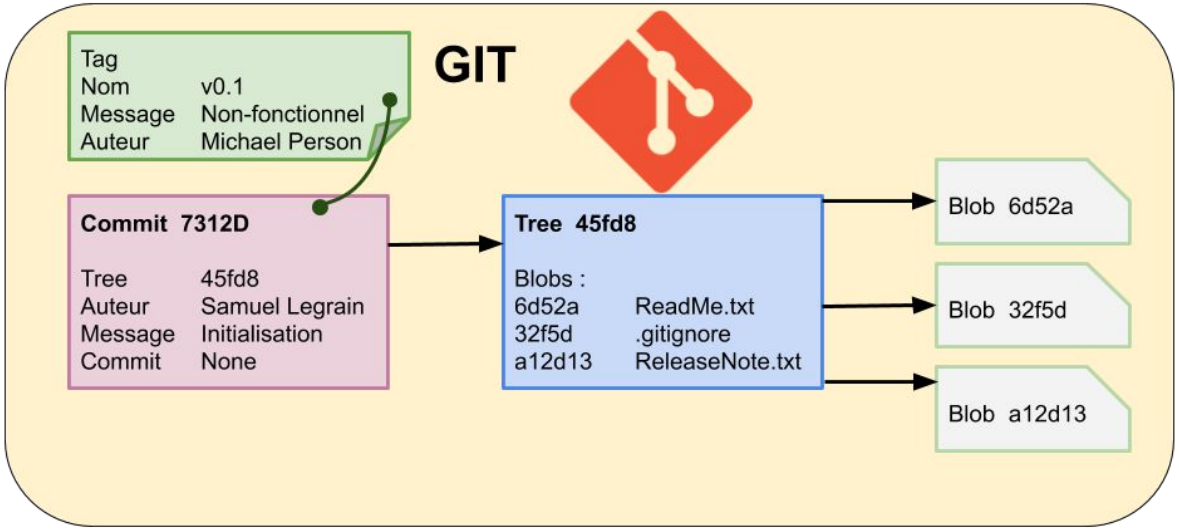
Un premier initialisant notre GIT.
Comme il s'agit ici de notre premier commit, il n'est lié à aucun autre commit et il est lié au tree 45fd8.

Un second où notre fichier ReadMe.txt a été mis à jour.
Le contenu du fichier change, et crée donc un nouveau blob. Ce blob est lié à un nouveau tree, liant les autres blobs inchangés. Et ce tree est lié à notre nouveau commit, pointant vers le commit précédent.



La composition - Le tag

Le "**tag**" est une étiquette, il est souvent lié à un commit important, permet de les nommer ou les numéroter pour les identifier aisément. Il peut apporter des informations complémentaires.



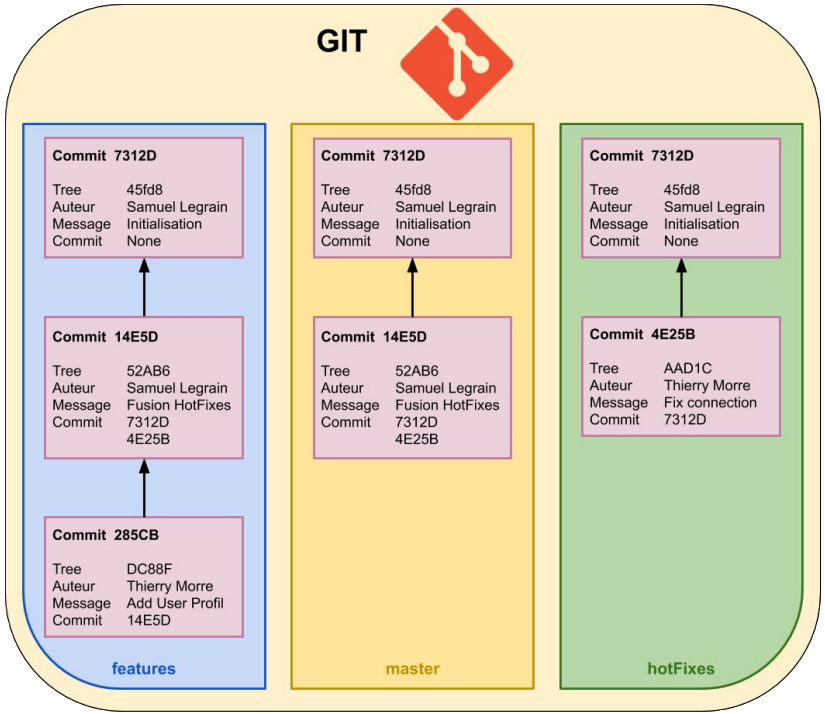
La composition - La branch

Une “**branch**” est un pointeur, il est lié à un commit, et se base sur les liaisons de se dernier pour définir un historique de travail.

La branch est généralement utilisée pour subdiviser un projet en différentes parties et permet d’organiser au mieux les environnements de travails.

La branch “**master**” est la première branch créé par défaut dans GIT, est considérée comme la branch principale.

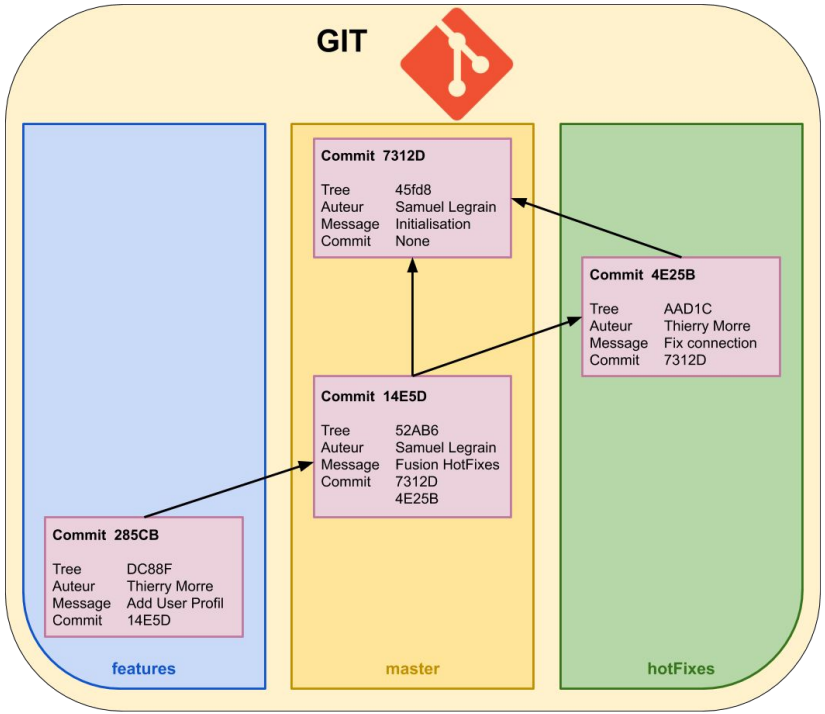
La manipulation de nos branchs peut varier selon les méthodologies de travail, l’une des plus utilisée : GitFlow



La composition - Le merge

Un "**merge**" est un commit résultant de la fusion de deux branch. Celui-ci sera donc lié à deux commits.

Il permettra aussi de gérer la gestion de conflit de contenu entre les modifications apportés aux blobs des commits auxquels il est lié.



Il existe plusieurs façons de fusionner des commits entre-eux. Nous verrons les différences dans un chapitre à venir.

La composition - Les repositories

GIT étant un logiciel de versionning non-centralisé, il nous faudra stocker les fichiers du projet sur l'ordinateur de chaque développeur travaillant sur le projet. C'est ce que l'on appel le dépôt local : **Local repository**.

Mais il est tout aussi possible de connecter un projet GIT à un ou plusieurs serveurs, permettant de centraliser les fichiers du projet et leurs différentes versions, et offrant la possibilité de travailler en équipe. Ce stockage réseau est ce que l'on appel un dépôt distant : **Remote repository**.

L'utilisation d'un dépôt distant ne dispense pas de la présence de dépôts locals.

Nous verrons dans les prochains chapitres qu'un dépôt local ne s'utilise pas de la même manière qu'avec un dépôt distant, et que diverses méthodologie de travail vont pouvoir être appliquée.

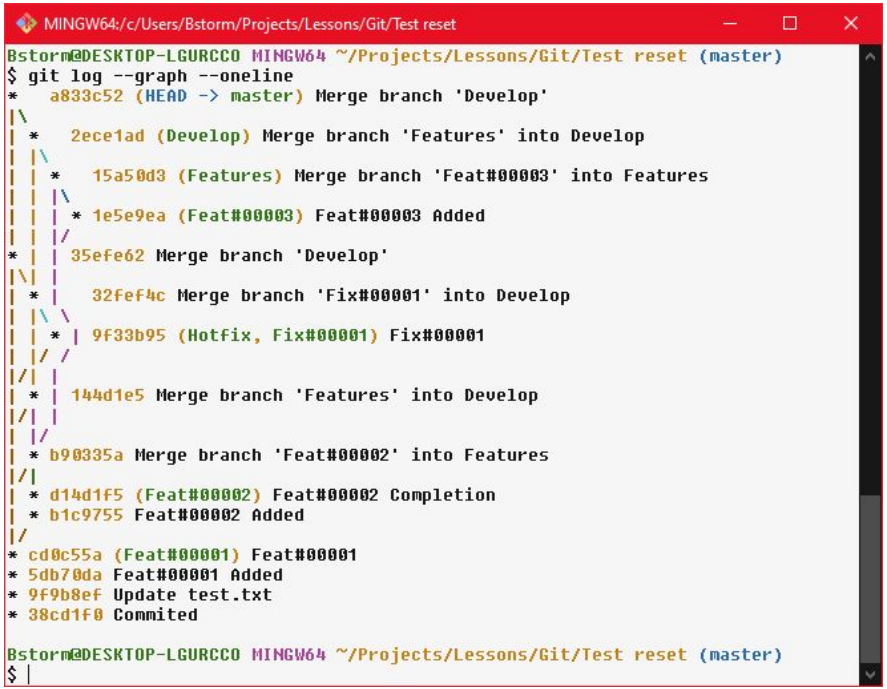
La composition - Le log

Le “log” d’un projet GIT est un listing détaillé des liaisons entre nos différents commits et branches formant ainsi un historique de notre projet.

L’option ***“oneline”*** permet de concentrer les informations essentiel d’un commit en une seule ligne.

L’option ***“graph”***, quant à lui, ajoute un graphique sur la gauche, apportant un aspect visuel simplifiant la compréhension des liaisons de nos éléments.

$ git log [–graph] [–oneline]



La composition - .gitignore

Fichier à la racine du projet, .gitignore permet de déterminer quels fichiers présent dans le dossier local du projet, GIT se doit d'ignorer tout au long du développement.

Pour créer un fichier .gitignore dans un projet GIT : **$ touch .gitignore**

Le fichier contient juste du texte, chaque ligne correspondant à une nouvelle condition de rejet. Ces conditions peuvent juste indiquer le fichier à exclure (chemin et nom de fichier basé sur la racine du projet) ou bien répondre à un code caractère qui est propre au .gitignore:

Pour vérifier les conditions de rejet du gitignore d'un projet GIT : **$ cat .gitignore**

La composition - .gitignore

*****	présence de 0 à n caractères
?	présence d'un caractère
[*abc***]**	présence d'un des caractères proposé (*a* OU *b* OU *c*)
[*1-4***]**	présence d'un des caractères compris entre les deux caractères séparés par le tiret (*1* OU *2* OU *3* OU *4*)
dossier / fichier	indique que ce qui se trouve à gauche du / est un nom de dossier
*dossier /**/ fichier*	présence de 0 à n dossiers séparant le dossier et le fichier
!	négations de la règle citée à droite du symbole

Exercices

Dans votre optique de recherche d'emploi, vous vous décidez à mettre en place un système de versionning de vos différents CV's et lettres de motivation.

- Créez un répertoire "RechercheEmploi", qui contiendra lui même un répertoire supplémentaire "CV".
- Initialisez un projet GIT dans le répertoire "RechercheEmploi".
- Ajoutez un fichier .gitignore qui interdit tous les fichiers ayant pour extension "tmp".
- Tester votre solution en créant un fichier .tmp et en tentant de vérifier votre statut.

Les actions de base

avec un dépôt local

Les actions de base - États d'un fichier

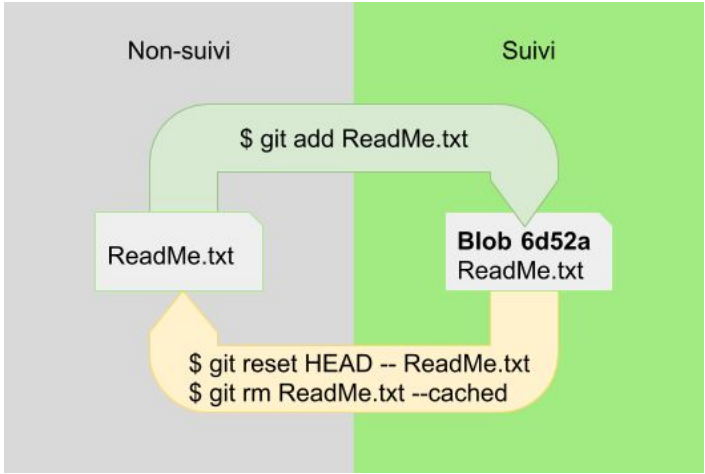
Tous fichiers composant un projet GIT ont un état :

Ils peuvent être "**Tracked**" (suivi) ou "**Untracked**" (non-suivi).

Un fichier est considéré comme non-suivi tant que celui-ci n'est pas encore indexé ou qu'il répond aux critères de sélections du .gitignore sans avoir été indexé au préalable.

Pour indexer un fichier et qu'il soit suivi, il faut utiliser la commande :
$ git add *nom_du_fichier*

et inversement avec la commande de suppression dans l'index :
$ git rm *nom_du_fichier* **—cached**
ou
$ git reset HEAD — *nom_du_fichier*



Nouvelles commandes équivalentes depuis la v2.24 :
$ git restore --staged *nom_de_fichier*

Les actions de base - États d'un fichier

Les fichiers suivis ont un sous-état, il en existe 3 :

- **Staged** (Indexé) : le fichier a été ajouté dans l'index du GIT de notre projet;
- **Commited** (*Validé*) : le fichier a été validé et enregistré tel quel dans la dernière version du projet;
- **Modified** (Modifié) : le fichier contient des modifications depuis sa dernière version validée, mais n'a pas été enregistré par GIT.

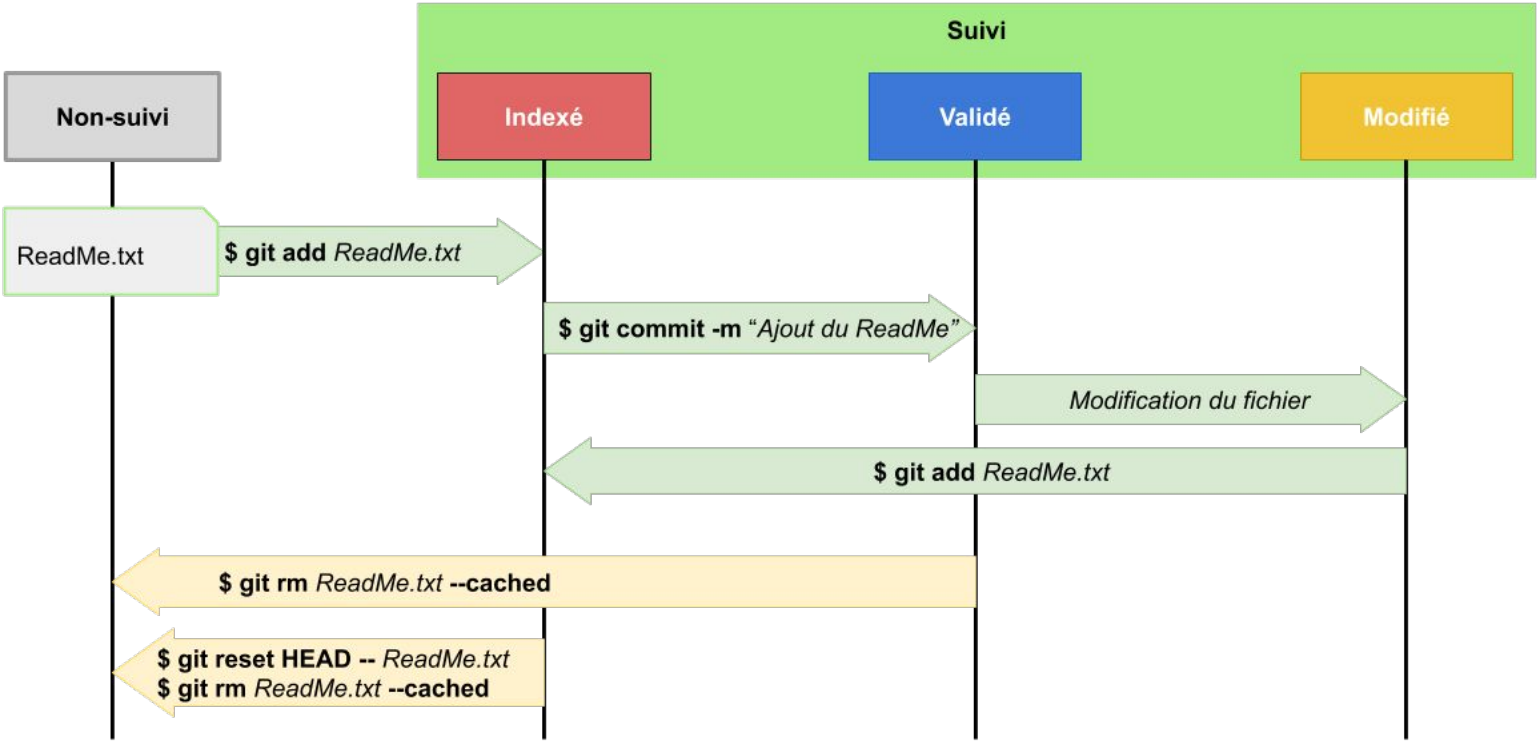
A l'aide de ces différents états, nous pouvons visualiser un cycle.

L'indexation se fait, comme vu précédemment, par la commande : **$ git add** *nom_du_fichier*

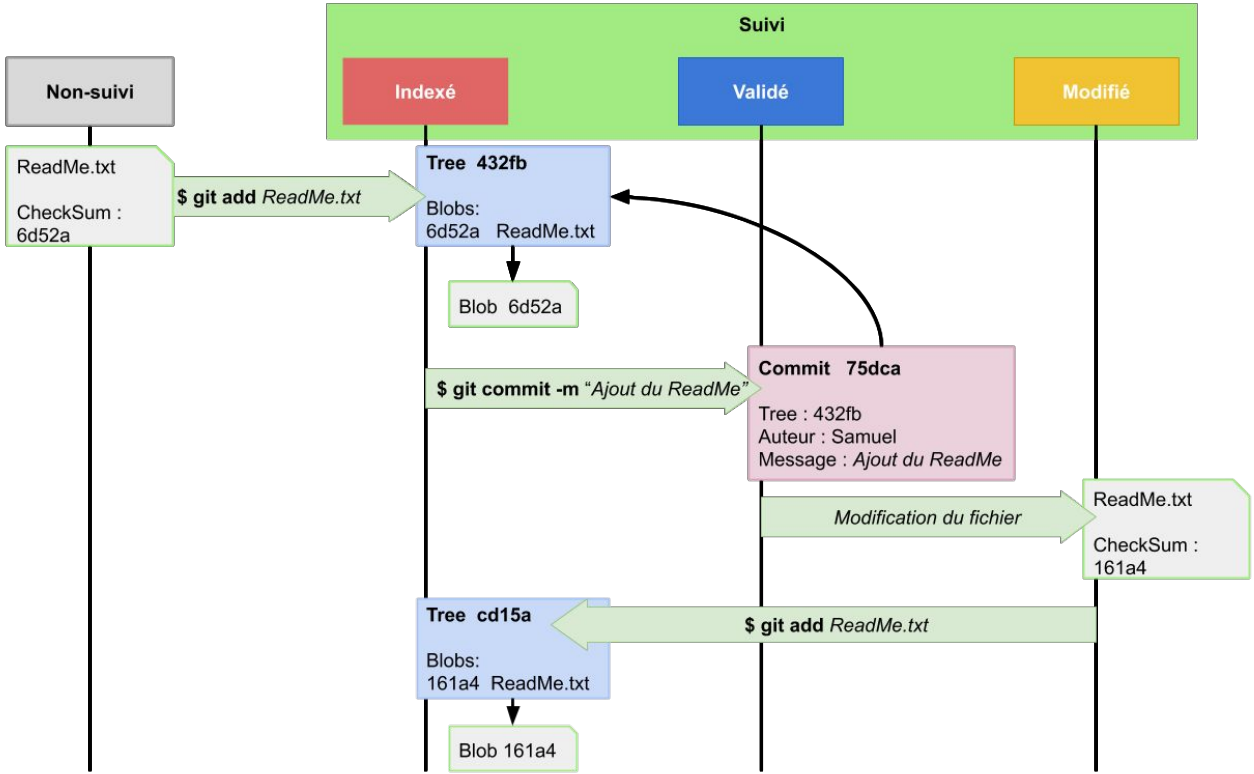
La validation se fait en créant un commit via la commande : **$ git commit -m "***message***"**
De plus, avec l'option **-a**, la commande **$ git add** peut être omise : **$ git commit -a -m "***message***"**

Bien entendu, la modification ne s'effectue pas par une commande, mais juste en éditant le contenu du fichier, GIT alors comparera de lui-même les checksums des blobs indexés aux fichiers enregistrés.

Les actions de base - États d'un fichier



Les actions de base - États d'un fichier



Exercices

Maintenant que vous connaissez les bases des manipulations de GIT, faîtes un premier commit de votre dossier “RechercheEmploi” ayant pour message “Init”.

Ajoutez dans le répertoire “CV” un premier document texte “MyInfo” qui contiendra vos informations personnelles (Nom, prénom, date de naissance, adresse e-mail, adresse postale et numéro de téléphone).

Dans ce même répertoire, ajoutez-y un autre fichier texte nommé “MyExp” qui lui contiendra un listing de vos expériences professionnels (Intitulé du job, nom de l’entreprise, année de début et de fin de contrat).

Finalisez en effectuant un commit avec le message “SetInfo_01 : Personnal and Exp”.

Les actions de base - États d'un fichier

Il existe une commande GIT permettant de connaître l'état de chacun des fichiers qui composent un projet :
$ git status

Cette commande affiche l'état de chaque fichiers et dossiers présents sur la branch courante.
Elle affiche ceci en formant 3 listes :

1. une liste des fichiers indexés mais non-validés
2. une liste des fichiers supprimés ou modifiés, ne correspondant plus aux dernières versions validés.
3. une liste des nouveaux fichiers du dossier de travail, non-présent dans les commits précédents.

Les fichiers ou dossiers ayant déjà fait l'objet d'une validation, et n'ayant pas eu de modification, ainsi que les fichiers ou dossiers, répondant au filtrage du ".gitignore" ne seront simplement pas repris avec cette commande.

Les actions de base - États d'un fichier

```
MINGW64:/c/Users/Bstorm/Projects/Lessons/Git/MonProjetGit

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ touch .gitignore

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ touch README.MD

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        README.MD

nothing added to commit but untracked files present (use "git add" to track)

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ git add .

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   README.MD


Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$
```

Création de nouveaux fichiers : *README.MD* et *.gitignore*

Affichage de l'état : ils sont tous deux "*Untracked*"

Indexation de la totalité des fichiers du projet

Affichage de l'état : ils sont tous deux en attente de validation ("*to be committed*")

```
MINGW64:/c/Users/Bstorm/Projects/Lessons/Git/MonProjetGit

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ git commit -m "Initialisation"
[master (root-commit) 624786a] Initialisation
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
 create mode 100644 README.MD

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$ git status
On branch master
nothing to commit, working tree clean

Bstorm@DESKTOP-LGURCCO MINGW64 ~/Projects/Lessons/Git/MonProjetGit (master)
$
```

Les actions de base - États d'un fichier

Avec la commande **$ git status -s**, il est possible d'obtenir un affichage plus concis de l'état des fichier.

Un caractère nous aidera à spécifier l'état d'un fichier, les plus fréquent sont :

- A : Fichier/dossier indexé mais non validé.
- M : Fichier/dossier modifié depuis sa dernière validation.
- D : Fichier/dossier supprimé.
- ? : Fichier/dossier non indexé et non-validé.

Il y a deux caractères par objets, celui de gauche représente l'état dans le dépôt local, celui de droite représente l'état dans le dossier de travail.

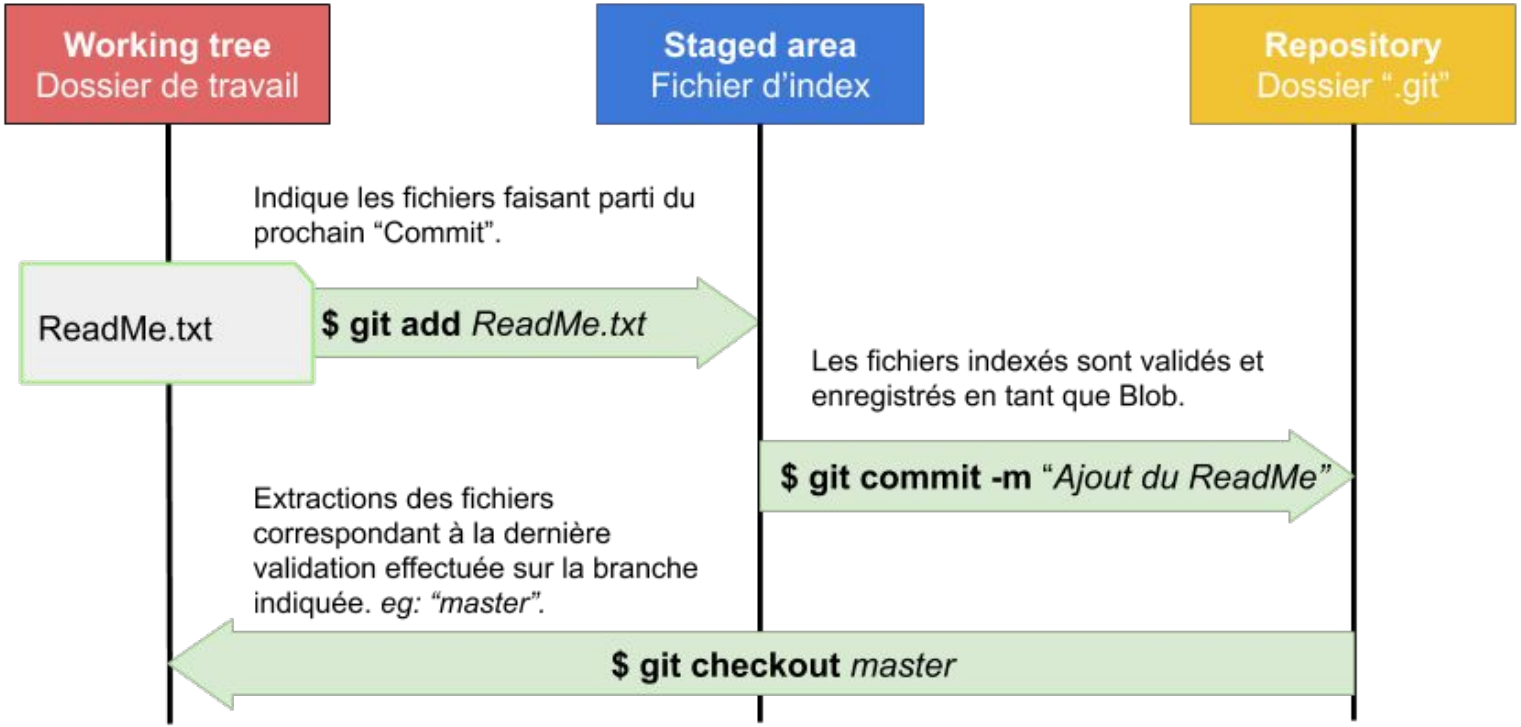
Dans notre exemple : "README.MD" a été modifié, ".gitignore" n'apparaît pas car inchangé.
"index.html" et "Style/main.css" ont été ajouté dans le dossier de travail, et seul "index.html" a été indexé.

Les actions de base - Zones de travail

Comme dit précédemment GIT peut afficher jusqu'à deux sous-états à vos fichiers. Ceci est dû au fait que le fichier est présent dans plusieurs zones de travail. Il en existe 3 dans tout projet GIT :

- **Working tree** (répertoire de travail) : c'est votre espace de travail, là où vous pouvez modifier le contenu de votre projet. Il s'agit à la base d'une extraction des fichiers du dernier commit effectué sur votre branche ("checkout"), ceux-ci proviennent d'une autre Zones de travail.
- **Staging area** (fichier d'indexation) : Simple fichier présent dans le répertoire .git du projet, il répertorie les informations en prévision du prochain "commit".
- **Repository** (répertoire .git) : Répertoire regroupant les données et la base de données des objets GIT (Blob, Tree, Commit, ...) du projet.

Les actions de base - Zones de travail



Les actions de base - HEAD

Le "**HEAD"** dans un projet GIT permet de connaître quel est le point de départ du "WorkingTree" actuel, permettant à l'utilisateur de cibler un commit précis ou bien le dernier commit d'une branche. Par défaut, le HEAD suit les commits effectués par l'utilisateur.

Pour définir la position du HEAD dans un projet, nous utiliserons la commande :

$ git checkout *code_sha1_commit*

ou

$ git checkout *tag*

ou

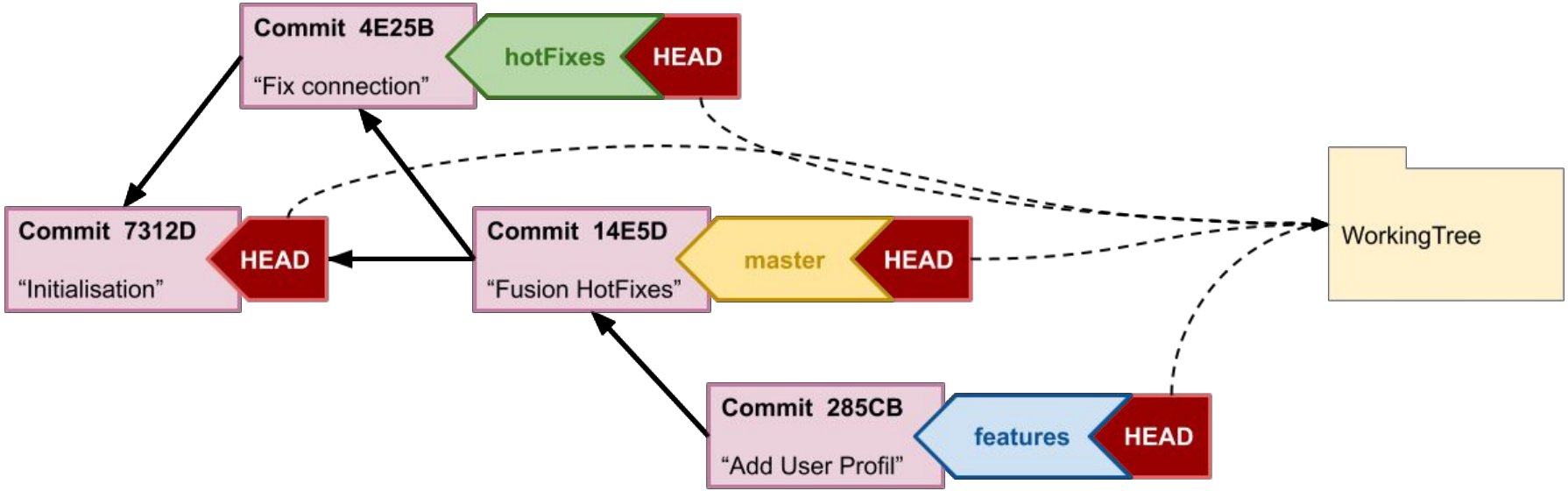
$ git checkout *nom_de_branche*

Nouvelles commandes équivalentes depuis la v2.24 :
$ git switch *code_sha1_commit / tag / nom_de_branche*

Pour connaître le SHA1 d'un commit, il nous faut le récupérer grâce à l'historique de notre projet :

$ git log [–graph] [–oneline]

Les actions de base - HEAD



Les actions de base

avec un dépôt distant

Les actions de base - Récupérer un dépôt distant

Bien que GIT ne nécessite d'aucun serveur distant pour fonctionner correctement, et a été conçu pour ne surtout pas en dépendre, l'utilisation de ceux-ci constitue, malgré lui, l'une de ses plus grande force : l'aspect collaboratif.

Cet aspect est essentiel dans un secteur métier tel que le développement où les équipes sont formées d'un nombre de collaborateurs variable et pouvant vivre dans n'importe quel coin du globe!

Plusieurs serveurs de dépôts distants proposent un service gratuit. Pour ne citer que les plus populaires :

- GitHub
- Bitbucket
- GitLab
- SourceForge
- etc. ...

Les actions de base - Récupérer un dépôt distant

De part l'élargissement de ce secteur, plusieurs projets open-source ont vu le jour, et ceux-ci peuvent être récupérer directement sur une machine locale, soit en vue de collaborer et fournir des fonctionnalités ou des correctifs au projet, soit en vue de démarrer une nouvelle version basée sur ce code-source.

Il est donc possible de récupérer une copie d'un projet à partir d'un dépôt distant : **$ git clone** *url_du_depot*

Les fichiers et dossier seront alors téléchargés et copiés dans votre dossier courant devenant votre "WorkingTree", et une liaison au dépôt distant nommé "***origin***" sera enregistré.

Les actions de base - Gérer les dépôts distants

Un projet GIT peut avoir plusieurs dépôts enregistrer. Ils ont un nom et un url, certains sont protégés et nécessite un identifiant et un mot de passe.

Pour connaîtres les dépôts liés à notre projet : **$ git remote**
et avec l'url de chacun : **$ git remote -v**

Pour en ajouter un : **$ git remote add** *nom_du_depot url_du_depot*

Les actions de base - Échanges avec un dépôt distant

Deux commandes permettent de récupérer les données de notre dépôt :

- **$ git fetch** *nom_source* : permet la récupération des données de notre source, mais ce ne sont que des références, et ne sont pas fusionner avec notre “WorkingTree”. Pour ce faire, ajoutez : **$ git merge**
- **$ git pull** *nom_source* : permet la récupération des données de notre source et les intègre à notre “WorkingTree”.

Nous pouvons aussi, à l’inverse, ajouter des données à l’intérieur de notre dépôt distant, à l’aide de la commande : **$ git push** [*nom_du_depot*][*nom_de_branche*]

Cette dernière manipulation ne peut se faire qu’à une condition :
on ne peut envoyer de commits sur un dépôt distant, tant que tous les commits de ce dépôt n’ont pas été récupéré sur le “WorkingTree”.

Exercices

A l'aide de GitHub, créez un dépôt distant, liez le à votre projet "RechercheEmploi" et partagez l'URL à votre formateur.

Une fois le dépôt accessible, ajoutez un nouveau fichier texte dans le répertoire "CV", nommé "MyStudies", qui doit contenir un listing de vos études (Année de finalité, titre du diplôme obtenu, nom de l'établissement).

Modifiez le fichier "MyInfo" pour y ajouter un listing de vos hobbies.

Faites un nouveau commit avec le message "SetInfo_02 : Studies" et publiez le sur votre dépôt distant.

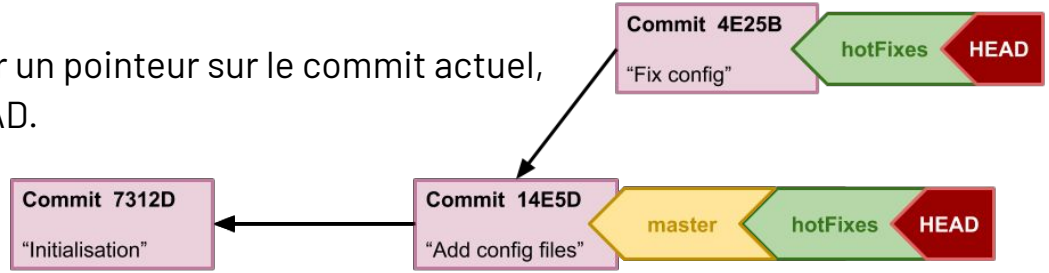
Les actions de base

avec les branchs

Les actions de base - Ajouter une branch

Le but de nos branchs est de subdiviser notre projet en différentes parties, soit dans le soucis de structurer le travail fournis, soit dans le but d'organiser pour chacun des développeurs leur propre environnement de travail.

La création d'une branch, a pour effet d'ajouter un pointeur sur le commit actuel, sur laquelle nous pourrons faire pointer le HEAD.



$ git branch *nom_de_branch*
$ git checkout *nom_de_branch*
ou
$ git checkout -b *nom_de_branch*

A partir de là, tous les prochains commits récupérerons le pointeur de cette branch.

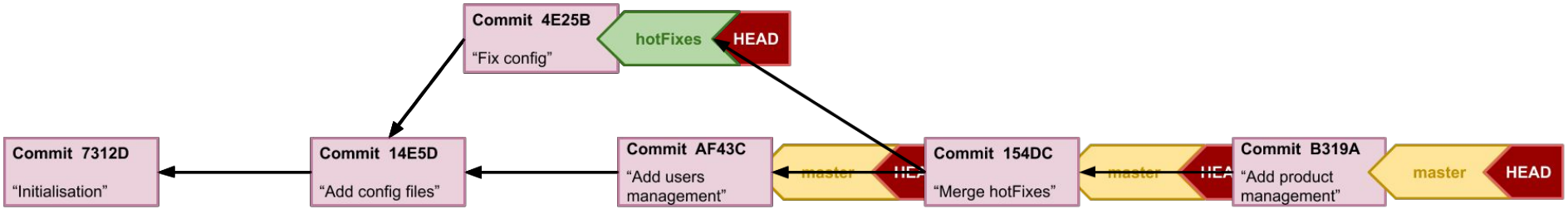
Pour revenir sur une autre branch : **$ git checkout** *nom_de_branch*

Les actions de base - Fusionner des branchs

Que cela soit pour une raison ou une autre, le plus fréquemment, ces branchs se terminent par un "*merge*", un commit spécifique qui a pour but de fusionner plusieurs commits en un seul.

Pour ce faire, il nous faut placer le HEAD sur la branch avec laquelle nous souhaitons poursuivre notre travail, ensuite créer le merge en indiquant la branch dont le dernier commit sera lier :

$ git checkout *nom_de_branch_1*
$ git merge *nom_de_branch_2*

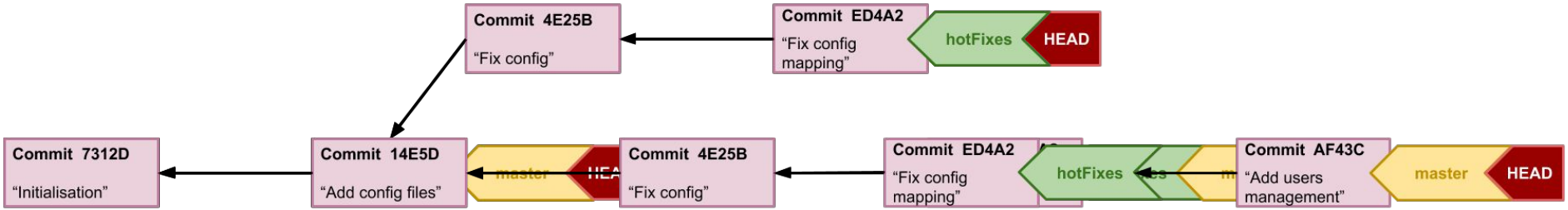


Les actions de base - Fusionner des branchs

Bien qu'il n'y ait qu'une seule commande pour produire un merge, celle-ci aura un comportement différent selon la situation dans laquelle notre projet se trouve.

L'exemple de la slide précédente présente le fait que nos branchs ont eu de nouveaux commits avant la mis en place du merge. Mais, il se peut qu'aucune modification ne soit apporté sur notre branch avant la fusion.

Dans cette situation très précise, notre GIT fera un merge dit "fast forward".
Aucun commit "merge" ne sera ajouté, mais la branch de travail pointera le dernier commit de la branch fusionnée. Pour éviter ce comportement par défaut : **$ git merge –no-ff** *nom_de_branch*



Les actions de base - Gestion des conflits

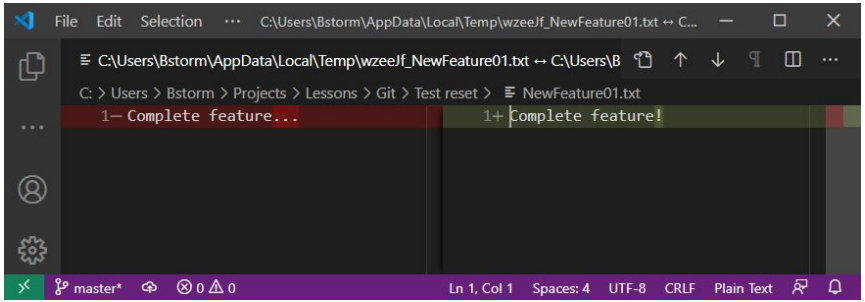
Lors de vos merges, il arrivera fréquemment qu'un fichier d'un des commits soit différent de son homologue d'un autre commit. C'est ce que l'on appelle un conflit.

L'utilisateur devra choisir entre ces deux fichiers, ou encore décider de soit supprimer, soit garder des informations venant de l'un et l'autre, ne formant plus qu'un fichier commun.

$ git diff *sha1_commit* : permet de visualiser les différences entre chaque fichiers ayant eu modification entre le commit du HEAD et le commit indiqué.

La version du commit ciblé à gauche et la version du commit du HEAD à droite.

En rouge les éléments supprimés, en vert ceux ajoutés.



Les actions de base - Gestion des conflits

Mais avant de commencer, il nous faut actualiser notre configuration. En effet, de base les outils proposés par GIT, via les lignes de commande, sont complexes et peu représentatifs des modifications effectu

Nous utiliserons Visual Studio Code pour nous aider dans cette tâche.
Il a l'avantage d'être gratuit et multiplateforme, avec une forte communauté.

Pour se faire, installez d'abord Visual Studio Code : https://code.visualstudio.com/download

Ensuite ajoutez dans le fichier .gitconfig les lignes suivantes :

(NB : .gitconfig se situe sur Windows dans le dossier
"C:\Users\nom_utilisateur\")

```
[merge]
   tool = vscode
[mergetool "vscode"]
   cmd = code --wait $MERGED
[diff]
   tool = vscode
[difftool "vscode"]
   cmd = code --wait --diff $LOCAL $REMOTE
```

Les actions de base - Gestion des conflits

Si un merge rencontre au moins un conflit, celui-ci interrompt la procédure automatiquement, et vous demande de fixer les conflits.

Pour ce faire, un fichier rencontrant un ou plusieurs conflits se verra modifié par l'ajout de balises.

```
Contenu présent dans la version
<<<<<<< HEAD
du HEAD
=======
du commit ciblé lors du merge
>>>>>>> Commit ciblé
```

Une première balise "**<<<<<<< HEAD**" : elle marque le début du contenu exclusif à la version actuelle, pointé par le HEAD.

Une séparation "**=======**" indiquant la fin du contenu exclusif au HEAD, marquant le début du contenu exclusif à la version du commit ciblé par le merge.

Une balise finale "**>>>>>>>** *sha1_du_commit* || *message_du_commit*", marquant la fin du conflit.

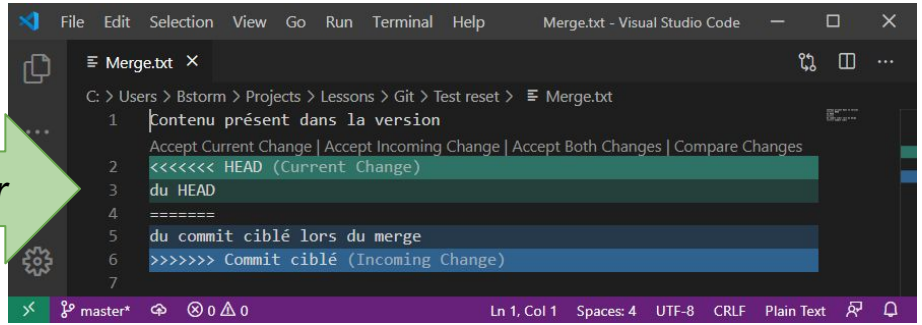
Les actions de base - Gestion des conflits

Nous pouvons donc ouvrir n'importe quel éditeur et modifier le contenu du fichier rencontrant des conflits.

Mais si vous avez suivi nos conseils et avez modifié votre "*.gitconfig*" pour obtenir Visual Studio Code comme éditeur de merge, le conflit sera plus facile à gérer.

```
Contenu présent dans la version
<<<<<<< HEAD
du HEAD
=======
du commit ciblé lors du merge
>>>>>>> Commit ciblé
```

$ git mergetool *nom_du_fichier*



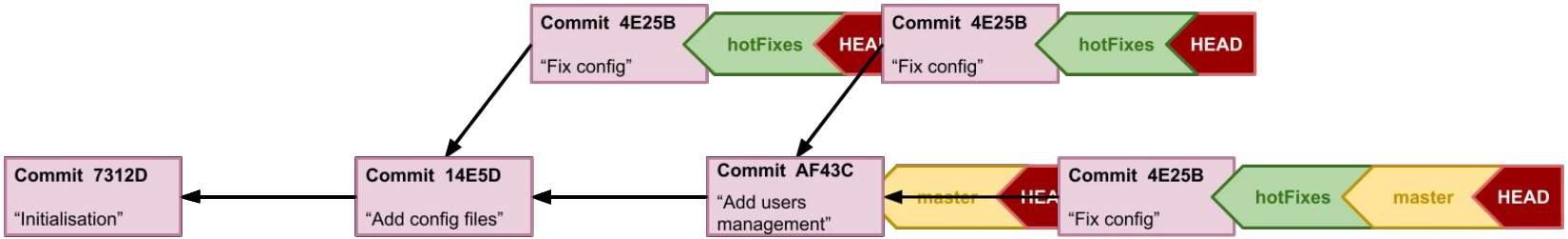
Une fois les conflits corrigés, sauvegardez et introduisez la commande : **$ git merge –continue**

Il y a aussi une commande permettant d'annuler la gestion des conflits ainsi que le merge : **$ git merge –abort**

Les actions de base - Rebaser une branch

Une autre méthode de fusion entre commit est ce que l'on appelle le "*rebase*", permettant de détacher un commit de sa liaison précédente, et le relier à une liaison ultérieure de la branch.

Cela a pour avantage de recréer la configuration nécessaire à un historique linéaire : le merge fast-forward ; tout en masquant que les opérations ont été effectués en parallèle.



Les actions de base - Rebaser une branch

Pour ce faire, nous allons devoir positionner le HEAD sur la branch que l'on veut repositionner dans notre log. Ensuite, nous lui indiquons sur quelle autre branch se rattacher.

$ git checkout *branch_a_detachee*
$ git rebase *branch_nouvelle_base*

Cette procédure va apporter un certain nombre de conflits, dans chacun des commits concerné par le rebase.

Une fois ces derniers réglés, nous terminons par un **$ git rebase –continue**
Une commande pour annuler l'ordre de rebase existe : **$ git rebase –abort**

Nous avons enfin une structure compatible pour un merge fast-forward :

$ git checkout *branch_nouvelle_base*
$ git merge *branch_rattachee*

Les actions de base - Branchs et dépôts distants

Tout ce qui a été vu dans les slides précédentes ne concernent que les branch d'un dépôt local, sur votre machine locale. Mais si vous travaillez avec des collaborateurs, en équipe, il serait préférable de pouvoir déposer et récupérer des branchs sur votre dépôt distant commun.

Pour déposer une branch de votre "local repository" dans un "remote repository", la procédure est simple :

$ git push -u *nom_depot nom_branch*

A l'inverse, pour récupérer une branch d'un "remote repository" dans votre "local repository", il nous faudra d'abord récupérer les données dans l'indexeur, ensuite créer une branch se basant sur celle de votre dépôt :

$ git fetch
$ git checkout -b *nom_branch_local nom_depot/nom_branch*

ou

$ git fetch
$ git checkout —track *nom_depot/nom_branch*

Exercice

Organisez-vous pour former un groupe de trois apprenants.

Votre formateur va vous délivrer un fichier, ce dernier est la rédaction d'un récit de 3 chapitres, chaque chapitre étant incomplet, cela sera à vous de compléter ce récit.

Sur une branch qui vous sera dédié et portant votre nom, vous devrez écrire la partie manquante d'un chapitre.

Une fois fini, fusionnez le tout sur un seul fichier dans la branch master.

Informations complémentaires

Attention, le logiciel git et le site github on un peu de mal avec les multis account… par exemple si on travail sur 2 dépôts distants il est préférable de suivre la manipulation suivante pour travailler avec les deux dépôts..

D'origin, pour se logger avec login mdp sur le push, le pc va prendre en compte les credentials initialement écrit avec un compte github.. il faudra donc lui dire lequel va avec lequel pour le projet untel.

How to setup Git on your PC for Multiple GitHub Accounts | fofx Academy

Multi accounts gitHub for one PC

Pour permettre le multi account, lancer en premier lieu dans un bash git, la commande suivante : git config —list

Trouver la ligne : **credencial.helper=wincred**

Si elle ne s'y trouve pas, taper la commande :

git config —global credential.helper wincred

et maintenant activons le multi credential pour git avec :

git config —global credential.useHttpPath true

```
Evengyl@DESKTOP-BJ6JDQ3 MINGW64 ~/Desktop/Bstorm/2021/GIT/RechercheEmploi (master)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
user.name=evengyl
user.email=dark.evengyl@gmail.com
merge.tool=vscode
mergetool.vscode.cmd=code -n --wait $MERGED
diff.tool=vscode
difftool.vscode.cmd=code -n --wait --diff $LOCAL $REMOTE
credencial.helper=wincred
credential.usehttppath=true
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
remote.origin.url=https://github.com/LoicBstorm/learnGit.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
branch.main.merge=refs/heads/main
```

Multi accounts gitHub for one PC

Pour terminer refaite un git push -u origin master (ou la branch voulue)

Attention ! veillez à être loggé avec le compte github de votre choix sur votre navigateur par défaut, car le prompt va lancé un micro logiciel qui utilisera les cookies de votre nav par défaut ce qu'y aura pour but de vous logger sur github automatiquement et le lié à votre projet, si votre remote origin blablabla n'est pas celui qu'y correspond au compte sur votre nav, ça ne changera rien et vous devrez recommencer...

cette manipulation n'est à faire qu'une fois par projet ou par account.

Merci pour votre attention.

bstorm
TRAINING