Les bases de Git

Ibrahim ALAME

14/02/2023

1 Le suivi des fichiers

1.1 Enregistrer des modifications dans un dépôt

Nous avons notre dépôt Git qui est initialisé. Nous allons faire un exemple légèrement différent de la vidéo, de cette façon vous aurez deux exemples que nous vous encourageons à refaire.

Nous allons maintenant étudier ensemble comment suivre les changements des fichiers dans notre dossier test-git en utilisant Git. Nous allons créer un premier fichier de test contenant un mot :

```
echo 'Coucou' > test.txt
```

Comme nous l'avons vu :

— la commande echo permet d'afficher sur la sortie standard une ligne de texte.

— l'opérateur > permet de prendre la sortie de la commande précédente et de la rediriger vers un fichier.

Cette commande permet donc d'afficher Coucou mais de rediriger le texte vers le fichier test.txt. Par défaut, le fichier sera créé si il n'existe pas. Nous obtenons donc simplement un fichier test.txt contenant simplement Coucou. Si vous faites la commande suivante :

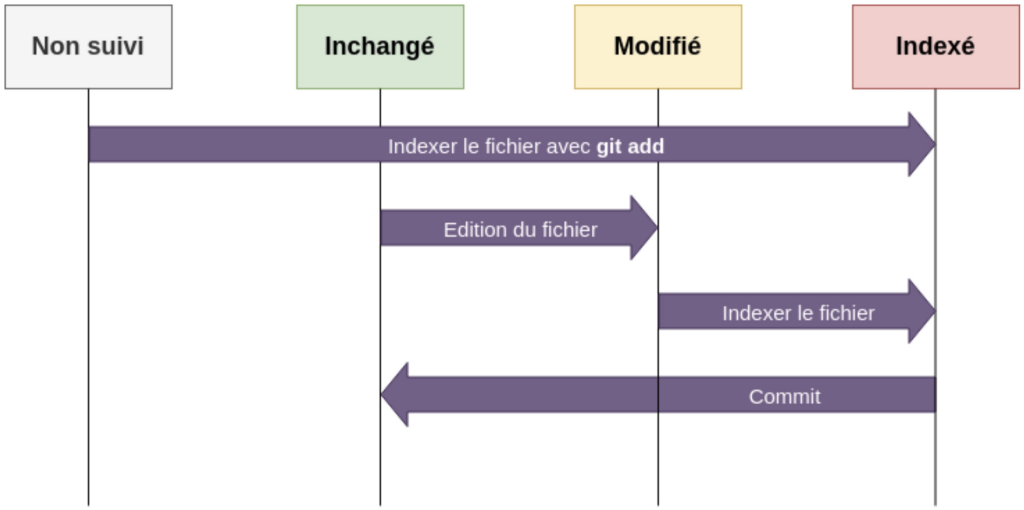
```
git status
```

Git vous indiquera que vous êtes sur la branche master (nous verrons plus tard dans la formation le concept de branche). Il vous indiquera également qu'aucun commit n'a été fait et que le fichier test.txt est non suivi. Enfin, il vous indiquera qu'aucune modification n'a été ajoutée à la validation.

1.2 Le suivi des fichiers : index et zone de transit (staging)

Tous les fichiers peuvent être suivis ou non suivis par Git. Un fichier suivi par Git est un fichier qui appartenait déjà à la version précédente ou qui a été ajouté avec une commande que nous allons voir. Ces fichiers suivis peuvent être inchangés, modifiés ou indexés entre deux versions.

Les fichiers qui n'appartenaient pas à une version précédente et qui n'ont pas encore été ajoutés sont non suivis. Voici un récapitulatif de l'état des fichiers, que nous allons voir en détails au fur et à mesure :



Les fichiers ne sont pas automatiquement suivis par Git, même s'il les détecte car vous pouvez vouloir ne pas suivre certains fichiers avec Git comme nous le verrons. Git vous laisse donc très libre et vous permet de choisir quels fichiers et dossiers suivre.

2 État d'un fichier et fonctionnement de l'index

2.1 Les trois états possibles des fichiers

Lorsqu'un fichier est suivi par Git, il peut être dans trois états :

1. modifié : le fichier est modifié mais la version modifiée n'est pas encore indexée ou sauvegardée.
2. indexé : le fichier est suivi par Git et ajouté en zone de transit (staging area). Git utilise, comme nous le verrons en détails, un fichier d'index pour déterminer quels fichiers feront parties de la prochaine sauvegarde.
3. validé : le fichier est stocké dans la base de données locale Git, qui constitue avec les fichiers de configuration le dépôt local.

Nous allons bien sûr voir ces états en détails et comment passer les fichiers d'un état à un autre.

2.2 Le répertoire de travail (working directory)

Le répertoire de travail est la version des fichiers sur laquelle vous travaillez actuellement : autrement dit c'est l'ensemble des fichiers et des dossiers ouverts actuellement dans votre éditeur de code.

Lorsque vous changez de version en utilisant Git, vous modifiez le répertoire de travail et Git extrait la bonne version des fichiers de sa base de données.

2.3 Indexer un fichier dans la zone de transit (staging area)

Un fichier indexé est un fichier qui est en zone de transit et qui sera ajouté à la prochaine sauvegarde effectuée avec Git. Pour indexer un fichier ou un dossier avec Git, il faut utiliser la commande :

```
git add fichier|dossier
```

Le fichier ou le dossier (et tous les fichiers et dossiers qu'il contient éventuellement) seront alors indexés et suivis par Git. Nous allons donc indexer le fichier de test que nous avons créé pour voir ce qu'il se passe :

```
git add test.txt
```

Ensuite nous regardons où en est Git en faisant ;

```
git status
```

Git nous indique que nous sommes toujours sur la branche master et que nous n'avons aucun commit. Mais à la différence d'avant, nous avons :

```
Modifications qui seront validées : nouveau fichier : test.txt.
```

Git nous dit ici que le fichier est bien indexé. Il apparaît d'ailleurs en vert (alors qu'avant il était en rouge). Le fichier est maintenant en zone de transit (staging area), c'est-à-dire qu'il fera partie de la prochaine sauvegarde que nous effectuerons avec Git. Approfondissons sur ce que Git a fait. Tapez la commande :

```
git ls-files --stage
```

Cette commande permet d'obtenir des informations sur les fichiers contenu dans le répertoire local ou dépôt local suivi par Git. En passant l'option stage, nous obtenons les informations sur les fichiers qui sont dans la zone de transit (staging area). Ces informations se trouvent en fait dans le fichier .git/index, mais en binaire. La commande permet donc de les rendre lisible :

```
100644 ac28f91b8c7314ce04f3f037948520dcc7a88ff7 0 test.txt
```

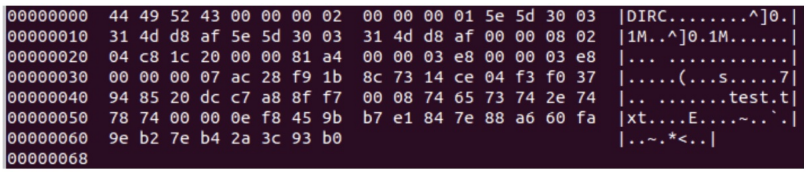
- Le premier nombre contient les bits de permissions et de type de fichier. Nous n'allons pas aller trop loin car c'est bas niveau. Sachez qu'ici 100 signifie fichier standard et 644 fichier que l'utilisateur propriétaire (owner) peut lire et écrire mais pas exécuter (et que le groupe et les autres ne peuvent que lire).
- La deuxième partie est la somme de contrôle SHA-1 du fichier, appelé hash comme nous l'avons vu.
- Le troisième nombre est utilisé pour la fusion (nous la verrons plus tard dans le cours). Il est très important pour la gestion des conflits.
- La dernière partie est le nom du fichier.

Mais que contient exactement le fichier .git/index ? Comme dit précédemment il contient du binaire. Mais nous pouvons l'afficher en hexadécimal :

```
hexdump .git/index
```

hexdump permet d'afficher dans la sortie standard, ici le terminal, un hexadecimal dump, c'est-à-dire d'afficher les données binaires du fichier en hexadécimal.

Ce qui donne quelque chose comme ça :



Dans ces informations il y a : la version de Git utilisée, la dernière fois que le fichier a été modifié (timestamp), le disque dur sur lequel se trouve le fichier (en l'occurence test.txt), l'inode du fichier (identifiant unique du fichier et informations systèmes bas niveau sur le fichier), les permissions du fichier (qui peut le lire, y écrire ou l'exécuter), le chemin vers le fichier, les identifiants de l'utilisateur et du groupe courant, la taille du fichier et bien sûr la somme de contrôle SHA-1, ou hash.

Vous n'aurez bien évidemment jamais à accéder à ce fichier, qui est uniquement un fichier d'administration utilisé par Git. Mais cela vous permet d'avoir déjà une bonne idée des informations qu'il contient.

2.4 Modification d'un fichier suivi

Modifions maintenant notre fichier text.txt pour observer le comportement de Git :

```
nano text.txt
```

L'éditeur nano est un un éditeur libre du système GNU. Il suffit de l'installer si le terminal vous indique que la commande n'est pas connue :

```
apt-get install nano
```

Mettez par exemple Coucou2, sauvegardez (ctrl + o puis entrée) et quittez (ctrl + x). Refaites un :

```
git status
```

Qu'obtenons-nous ? Les informations sur la branche et le commit n'ont pas changé. Nous avons des modifications qui seront validées (en vert) :

```
nouveau fichier : test.txt
```

Et nouveauté, des modifications qui ne seront pas validées (en rouge) :

```
modifié :         test.txt
```

Intéressant ! Cela signifie qu'une fois qu'un nouveau fichier a été indexé, si nous le modifions ensuite, la nouvelle modification n'est pas ajoutée à l'index !

Il faut refaire git add test.txt pour indexer la modification pour la prochaine sauvegarde :

```
git add test.txt
```

Et maintenant la nouvelle modification est bien indexée par Git :

```
nouveau fichier : test.txt
```

— Il faut donc bien retenir que Git détecte les fichiers qui ne sont pas suivis dans le dépôt et les modifications effectuées sur les fichiers.
— Cependant, par défaut, Git ne va pas indexer et mettre ces nouveaux fichiers, ou ces fichiers modifiés, en zone de transit (staging area).
— La raison est simple : Git vous laisse totalement libre pour indiquer quels fichiers et quelles modifications des fichiers doivent être ajoutées à la prochaine sauvegarde.

Maintenant que vous savez ça, reprenez le schéma vu dans la leçon précédente et il devrait être un peu plus clair ! Il le sera encore plus une fois que nous aurons étudié les commits.

Retenez que git add permet en fait d'ajouter la version actuelle d'un fichier à la prochaine sauvegarde.

Cela signifie que si vous modifiez une nouvelle fois test.txt après l'avoir git add, la nouvelle version du fichier ne sera pas indexée et ne sera pas mise en zone de transit (staging area). Il faudra de nouveau le git add si vous voulez inclure la dernière version dans la sauvegarde.

A notez que comme nous l'avons déjà dit, si vous ajoutez un dossier, alors tous les fichiers et les dossiers contenus dans ce dossier seront indexés et mis en zone de transit.

3 Ignorer des fichiers et des dossiers

3.1 Ignorer des fichiers

Dans tous les projets que vous ferez, il y a des fichiers et des dossiers que vous ne voudrez pas du tout suivre avec Git. Vous ne voudrez pas que Git sauvegarde différentes versions de ces fichiers et dossiers.

Pour ignorer des fichiers, il faut créer un fichier spécial à la racine de votre projet, pour nous, dans le dossier test-git :

```
nano .gitignore
```

Ce fichier est un fichier caché, et c'est pour cette raison qu'il commence par un point. Dans ce fichier vous mettrez souvent le dossier des dépendances node_modules, les fichiers de configuration de votre éditeur de code, des fichiers de log en local etc. Par exemple nous pouvons simplement mettre :

```
node\_modules
```

Tous les dossiers (et leur contenu) et les fichiers s'appelant node_modules seront ignorés par Git . Si vous terminez un nom avec /, seuls les dossiers (et tout leur contenu) avec le nom spécifié seront ignorés, par exemple :

```
node\_modules/
```

Vous pouvez également utiliser des expressions régulières simplifiées dans ce fichier. * signifie un ou plusieurs caractères :

```
local-doc/*.txt
```

Ici, tous les fichiers qui sont dans un dossier local-doc et qui finissent par .txt seront ignorés. Ainsi par exemple le fichier /projet/local/local-doc/test.txt serait ignoré. En revanche, /projet/local/local-doc/commands/test.txt ne sera pas ignoré car il est dans un sous-dossier. ** signifie une série d'un ou plusieurs dossiers : Ainsi, pour ignorer tous les fichiers dans le dossier et les sous-dossiers qui finissent par .txt, il faut faire :

```
local-doc/**/*.txt
```

Ici tous les fichiers finissant par .txt et qui sont dans le dossier local-doc ou dans un dossier situé dans local-doc (même sur plusieurs niveaux de dossiers) seront ignorés.

4 Afficher les différences entre répertoire, index et sauvegarde avec git diff

4.1 Vérifier les fichiers indexés, modifiés et non indexés

Nous avons vu que la commande git status nous donnait certaines informations sur les fichiers indexés.

Nous allons voir qu'il existe une autre commande permettant d'obtenir plus d'informations sur les fichiers indexés en zone de transit (staging area) : git diff.

4.2 Afficher les modifications des fichiers non indexés

La commande git diff sans aucune option permet d'afficher les modifications des fichiers dans le répertoire de travail qui ont été modifiés mais non indexés. En fait, elle permet de comparer les fichiers du répertoire de travail avec la zone de transit, c'est-à-dire avec les fichiers indexés, et de n'afficher que les fichiers qui ont donc des modifications non indexées.

Si vous tapez git diff maintenant, vous n'aurez rien car nous n'avons avons indexé le fichier test.txt que nous avons modifié. Modifions de nouveau le fichier :

```
nano text.txt
```

Mettez par exemple Coucou3, sauvegardez (ctrl + o puis entrée) et quittez (ctrl + x). Refaites un :

```
git diff
```

Vous aurez le résultat suivant, que nous allons voir ensemble :

```
diff --git a/test.txt b/test.txt
index ae362e4..1344bd0 100644
--- a/test.txt
+++ b/test.txt
```

```
@@ -1 +1 @@
-Coucou2
+Coucou3
```

La première ligne diff –git permet de spécifier que nous sommes au format git diff et a/test.txt b/test.txt permet de spécifier qu'il y a des différences entre le fichier indexé et le fichier dans le répertoire de travail.

La seconde ligne nous donnes le début du hash de la version du fichier indexée, puis ... et le début du hash de la version du fichier dans le répertoire de travail. Nous avons ensuite 100644 qui comme nous l'avions vu donne le type de fichier et les permissions (644).

Ensuite nous avons deux lignes qui nous donne quels fichiers ont été modifiés, il y a toujours — et +++ quel que soit le nombre de modifications entre les deux versions du fichier.

Ensuite nous avons @@ -1 +1 @@. Il permet de spécifier les intervalles de lignes dans le fichier où il y a des différences entre les deux versions.

1. Le premier nombre précédé par un - permet de spécifier la ligne de départ des différences (ici la ligne 1) dans la version de départ, puis il est éventuellement suivi d'une virgule et d'un autre nombre qui signifie jusqu'à la ligne n.
2. Le second nombre, précédé par un + est le même intervalle de lignes dans le fichiers pour les différences mais pour la version d'arrivée.

Ici, cela signifie que nous avons des différences uniquement sur la première ligne dans le fichier de départ et dans le fichier d'arrivée. Remodifions le fichier test.txt pour avoir un exemple plus complet :

```
nano text.txt
```

Mettez par exemple Test à la seconde ligne et Test2 à la troisième ligne, sauvegardez (ctrl + o puis entrée) et quittez (ctrl + x). Refaites un :

```
git diff
```

Nous avons maintenant :

```
diff --git a/test.txt b/test.txt
index ae362e4..01e8afb 100644
--- a/test.txt
+++ b/test.txt
@@ -1 +1,3 @@
-Coucou2
+Coucou3
+Test
+Test2
```

Nous voyons que l'intervalle de départ est toujours de la première ligne à la première ligne (1,1 est abrégé en 1) ce qui signifie que dans la version de départ la première ligne diffère par rapport à la version d'arrivée.

En revanche, pour l'intervalle d'arrivée, nous avons 1,3 ce qui signifie que nous avons des différences de la première à la troisième ligne dans le fichier d'arrivée.

Enfin, nous avons la liste des différences entre les deux fichiers précédés par des - pour les suppressions et par des + pour les ajouts.

4.3 Afficher les modifications des fichiers indexés

La commande git diff –staged permet d'afficher les modifications des fichiers indexés par rapport à la dernière sauvegarde. Cette commande compare les différences entre la dernière sauvegarde (ou l'origine si il n'y a eu aucune sauvegarde) et l'index. Elle permet de visualiser les différences qui seront donc sauvegardées par Git :

```
git diff --staged
```

Ici, dans notre exemple, nous avons :

```
diff --git a/test.txt b/test.txt
new file mode 100644
index 0000000..ae362e4
--- /dev/null
+++ b/test.txt
@@ -0,0 +1 @@
+Coucou2
```

En effet, nous avons les différences entre l'origine et la version du fichier que nous avions indexée.

- Nous avons la même chose pour la première ligne.
- La seconde ligne indique que nous avons créé un fichier standard avec les permissions 644.
- La troisième ligne indique 0000000 car il n'y a pas de version précédente du fichier vu que nous l'avons créé et il n'y a donc pas de hash de départ. ae362e4 est le début du hash de la version du fichier indexée.
- Ensuite, nous avons −−− /dev/null qui signifie que la source du fichier est null car nous le créons. (Si nous supprimions un fichier nous aurions +++ /dev/null car c'est la cible qui deviendrait null).
- @@ −0,0 +1 @@ signifie que nous n'avions pas de fichier donc l'intervalle de départ est de la ligne 0 à 0 et l'intervalle d'arrivée est 1 qui signifie que nous avons une différence à la ligne 1 avec le fichier d'origine (qui est inexistant ici).
- Enfin, nous avons la différence qui est affichée : +Coucou2. Nous avons ajouté Coucou2.

Rappelez-vous bien que la comparaison des différences est ici entre la dernière sauvegarde ou l'origine et les fichiers dans la zone de transit, c'est-à-dire indexé.

C'est pour cette raison que nous n'avons pas Coucou1 ou autre chose, car nous n'avons pas de sauvegarde donc l'origine est null.

5 Créer des sauvegardes avec git commit

5.1 Créer une sauvegarde

En Git, nous parlons de validation des modifications pour une sauvegarde. Une sauvegarde s'appelle un commit, cela signifie littéralement que nous engageons les modifications indexées.

Un commit n'est donc pas à prendre à la légère ! C'est une sauvegarde qui sera à tout jamais dans le dépôt Git.

La commande git commit

Pour faire un commit, il suffit de taper :

```
git commit
```

Git va alors ouvrir l'éditeur par défaut, si vous rencontrez cette erreur :

```
hint: Waiting for your editor to close the file... error: unable to start editor 'editor'
```

Cela veut dire que Git ne sait pas quel éditeur utiliser. Dans ce cas, nous allons le configurer :

```
git config --system core.editor nano
```

Ici nous mettons en configuration système, la plus haut niveau, que nous utiliserons l'éditeur nano. Refaites la commande : git commit ensuite. Vous aurez alors :

```
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Sur la branche master
#
# Validation initiale
#
# Modifications qui seront validées :
#       nouveau fichier : test.txt
#
# Modifications qui ne seront pas validées :
#       modifié :         test.txt
#
```

Nous avons un résumé de ce qui va être sauvegardé. Comme prévu, la version indexée du fichier test.txt va être sauvegardée, mais pas la version modifiée de ce fichier que nous n'avons pas indexé. Fermez l'éditeur en faisant ctrl + x. Vous aurez le message suivant :

```
Abandon de la validation dû à un message de validation vide.
```

C'était pour vous montrer que les sauvegardes sur Git doivent obligatoirement avoir un message de validation, c'est-à-dire un court texte décrivant les changements apportés dans cette version.

Nous allons ajouter une option afin de vérifier exactement ce que nous avons modifié :

```
git commit -v
```

Cela permet en fait de mettre le résultat de git diff –staged après le commentaire dans l'éditeur :

```
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Sur la branche master
#
```

```
# Validation initiale
#
# Modifications qui seront validées :
#       nouveau fichier : test.txt
#
# Modifications qui ne seront pas validées :
#       modifié :         test.txt
#
# ------------------------ >8 ------------------------
# Ne touchez pas à la ligne ci-dessus.
# Tout ce qui suit sera éliminé.
diff --git a/test.txt b/test.txt
new file mode 100644
index 0000000..ae362e4
--- /dev/null
+++ b/test.txt
@@ -0,0 +1 @@
+Coucou2
```

Nous vous encourageons à le faire au début pour bien vérifier ce que vous allez sauvegarder. Quittez l'éditeur avec ctrl + x.

Le message de commit

Pour l'instant nous n'avons rien sauvegardé, car nous avons à chaque fois abandonné le commit en ne précisant pas de message de validation. Nous allons le refaire une fois, mais en ajoutant à la première ligne le message suivant : Premier commit. Puis nous allons sauvegarder ctrl + o et quittez ctrl + x. Nous avons alors le message suivant :

```
[master (commit racine) bfc88f2] Premier commit
1 file changed, 1 insertion(+)
create mode 100644 test.txt
```

Nous avons effectué notre première sauvegarde dans Git ! Notez que seul le message spécifié est enregistré dans le commit, si les commentaires, ni le résultat de git diff –staged en utilisant git commit -v n'ont été sauvegardés.

La première ligne nous indique d'abord la branche sur laquelle nous avons effectué le commit, ici master (nous reviendrons très en détails sur les branches plus tard). Ensuite un message indique qu'il s'agit du premier commit, appelé root commit ou commit racine. Nous retrouvons le début du hash du commit, ainsi que le début du message de validation, appelé également message de commit.

Enfin, nous retrouvons un résumé de ce que nous avons fait : nous avons modifié un seul fichier, que nous avons créé avec les permissions par défaut (create mode 100644 test.txt) et effectué une insertion d'une ligne

Recommandations sur les messages de commit

Il est recommandé par Git d'effectuer des messages de validation décomposé de la manière suivante :

— 1ère ligne : titre du commit de moins de 50 caractères qui résume la modification.
— Puis une seconde ligne vide (saut de ligne).
— Puis un paragraphe décrivant de façon plus précise les modifications apportées.

Voici par exemple un message de commit du répertoire de Git sur Github (oui Git est développé en utilisant comme système de version... lui-même, c'est une belle mise en abîme).

```
Azure Pipeline: switch to the latest agent pools

It would seem that at least the `vs2015-win2012r2` pool (which we use
via its old name, `Hosted`) is about to be phased out. Let's switch
before that.
```

Le raccourci git commit -m

Lorsque vous serez un peu plus familier de git, nous allons utiliser git commit -m qui permet de ne pas ouvrir l'éditeur et directement de passer le message de validation.

Afin de respecter la recommandation il faut utiliser un premier -m pour le titre et un second pour le corps. Ajoutons les modifications que nous n'avions pas indexé :

```
git add test.txt
```

Puis vérifions ce qui va être sauvegardé :

```
git diff --staged
```

Et nous obtenons :

```
diff --git a/test.txt b/test.txt
index ae362e4..de0e405 100644
--- a/test.txt
+++ b/test.txt
@@ -1 +1,3 @@
  Coucou2
+Test
+Test2
```

Nous effectuons maintenant le commit en suivant les recommandations sur les messages de validation :

```
git commit -m "Second commit de test"
  -m "Nous avons ajouté deux lignes dans notre fichier test.txt afin de tester le comportement
  de Git lorsque nous modifions plusieurs lignes"
```

Nous avons notre titre en premier, puis le descriptif avancé en second.

5.2 Sauter l'étape de la zone transitoire (ou indexation)

Parfois vous souhaiterez faire une petite modification et vous n'aurez pas besoin de contrôler ce que vous allez sauvegarder, et vous ne voudrez pas non plus indexer manuellement ces modifications avec git add. Dans ce cas vous pouvez utiliser l'option git commit -a.

Reprenons notre exemple en modifiant le fichier test.txt une nouvelle fois :

```
nano test.txt
```

Supprimez par exemple une ligne, sauvegardez (ctrl + o) et quittez (ctrl + x). Nous utilisons maintenant le raccourci :

```
git commit -am "Suppression de la dernière ligne"
-m "Nous supprimons la dernières ligne pour tester l'indexation automatique des fichiers suivis."
```

Nous n'avons pas eu à indexer le fichier avec git add après la modification. Git l'a fait automatiquement pour nous.

A noter que Git n'indexera automatiquement que les fichiers qui sont suivis.

Si vous avez des fichiers créés mais non encore suivi, ils ne seront donc pas sauvegardés dans le commit !

5.3 Les informations contenues dans un commit

Les informations suivantes, appelées métadonnées (c'est-à-dire des données informatives sur d'autres données), sont enregistrées dans un objet commit :

- Nom et email l'auteur du code : c'est l'auteur des modifications (celui qui a écrit le code).
- Nom et email du committer : le committer est celui qui effectue le commit de fusion (nous verrons la fusion plus tard). La plupart du temps l'author et le committer sont les mêmes personnes.
  Mais sur certains projets importants, il y a une étape de validation pour chaque commit, les auteurs du code envoient souvent leurs patch (leurs commits de modifications) par email (pour Linux ou Git par exemple, ou comme nous le verrons sur Github/Gitlab), aux reviewers (qui relisent le code) et ce sont ces contributeurs de l'équipe core qui vont effectuer le commit, c'est-à-dire valider les modifications à sauvegarder. Dans ce cas, il y aura un commit de fusion et ce cera le reviewer qui sera le committer.
- Il y a ensuite la date des modifications (GIT_AUTHOR_DATE) et la date du commit (GIT_COMMITTER_DATE).
- Il y a également le hash du tree racine de l'instantané, nous allons détailler juste après.
- Et enfin, le hash du commit parent, nous détaillerons également.
- Ensuite, Git génère le hash de toutes ces informations avec l'algorithme SHA-1 ce qui nous donne le hash du commit.
  Le hash permet de s'assurer de l'intégrité de toutes ces métadonnées. Aucune donnée ne pourra être modifiée sans que Git ne le sache car le hash serait modifié.

5.4 Les objets Git

Il existe trois types d'objet Git : les blob, les commits et les trees (ou arbres). Vous pouvez afficher tous les objets de votre répertoire Git avec cette commande :

```
find .git/objects -type f
```

- find est une commande permettant de rechercher des fichiers et des dossiers. Nous lui passons l'option -type f qui permet de préciser que nous recherchons que des fichiers réguliers (f).

— .git/objects permet de préciser que nous recherchons uniquement les fichiers dans ce dossier. Nous obtenons ainsi tous les fichiers de ce dossier, ce qui donne quelque chose comme ça :

```
.git/objects/35/f8bc3582cb5f2897ccbc51d19f3bf41e748ccb
.git/objects/ac/28f91b8c7314ce04f3f037948520dcc7a88ff7
.git/objects/a5/6ccd8f89364bf9176fea2ab599b81a208ca857
.git/objects/c7/737ce0b53ff193d25019f011d265ae8201ceb3
.git/objects/1a/062bd8f48541dbfb8b3c7f5147b16fba3883e3
.git/objects/7e/8fa1f55e55b84e1339e40cdd8df8060117e6de
.git/objects/10/864573c8ec172a6377f04f4228022ecb7a1a92
.git/objects/ff/8e9239574e9bed3fd1fa4d3bcca1314f7e7107
.git/objects/f8/2c7c8e522e4a2ea086b72b5ffdfc6dd3373ae2
```

Le stockage des objets

Tous ces objets sont stockés dans le dossier .git/objects de la même manière. Chaque objet a un hash SHA-1 qui sera la clé si on considère Git comme une base de données clés / valeurs.

Dans le dossier .git/objects, chaque objet est inclus dans un dossier qui a pour nom les deux premières lettres de son hash (par exemple ici 35) et le nom du fichier contient les 38 caractères restants de son hash (par exemple ici f8bc3582cb5f2897ccbc51d19f3bf41e748ccb).

La valeur de ces objets sont compressés avec la librairie zlib qui utilise l'algorithme de compression defalte. Il est possible de voir ce que contient n'importe quel objet Git avec la commande suivante :

```
git cat-file -p hash
```

Il faut préciser les 40 caractères hexadécimaux du hash, par exemple ici :

```
git cat-file -p 35f8bc3582cb5f2897ccbc51d19f3bf41e748ccb
```

Ce qui donne par exemple :

```
Coucou
Test
Test2
```

Détails sur les types d'objets

Nous avons vu qu'il existe trois types d'objets principaux avec Git. Pour déterminer le type d'un objet il faut faire :

```
git cat-file -t hash
```

Avec t pour type, qui donne par exemple :

```
git cat-file -t 35f8bc3582cb5f2897ccbc51d19f3bf41e748ccb
blob
```

Tout contenu en Git est soit un arbre soit un blob. Les blobs (Binary Large Objects) sont des objets binaires qui peuvent être n'importe quoi : des fichiers de code, des images, des vidéos etc.

Les arbres (ou trees) correspondent à un répertoire et permettent à Git de stocker plusieurs fichiers ensemble. Un arbre contient une ou plusieurs entrées, chacune étant le hash d'un blob ou d'un autre arbre avec ses droits d'accès, son type et son nom de fichier associé.

Nous allons prendre un exemple complet pour bien comprendre. Nous allons repartir d'un répertoire vide et créer un dossier et trois fichiers :

```
echo 1 > fichier1.txt
echo 2 > fichier2.txt
mkdir dossier
echo 3 > dossier/fichier3.txt
```

Nous avons donc deux fichiers au premier niveau et un dossier dans lequel nous avons un fichier. Nous indexons tous ces éléments puis affichons les objets créés :

```
git add .
find .git/objects -type f
```

Nous obtenons :

```
.git/objects/0c/fbf08886fca9a91cb753ec8734c84fcbe52c9f
.git/objects/d0/0491fd7e5bb6fa28c517a0bb32b8b506539d4d
.git/objects/00/750edc07d6415dcc07ae0351e9397b0222b7ba
```

Avec la commande git cat-file -t hash pour chaque objets, nous déterminons que nous avons trois blobs. En fait, lorsque nous indexons des fichiers et des dossiers, seuls les objets pour les contenus sont créés. Nous avons ici trois objets pour les trois fichiers, avec leur hash en clé et le blob en valeur (c'est-à-dire le contenu du fichier en binaire).

Que se passe t-il si nous créons notre premier commit ?

```
git commit -m "Premier commit"
find .git/objects -type f
```

Nous avons maintenant :

```
.git/objects/0c/fbf08886fca9a91cb753ec8734c84fcbe52c9f
.git/objects/9e/633d56381d9f0335320f22ccf3f1562d1f80d8
.git/objects/3c/dcfedb18b063a6bb4f1b43ddd73e2f88d5738a
.git/objects/d0/0491fd7e5bb6fa28c517a0bb32b8b506539d4d
.git/objects/00/750edc07d6415dcc07ae0351e9397b0222b7ba
.git/objects/1e/784b993a4dc7956c84dda1d7060c56fde32ec8
```

En utilisant git cat-file -t hash pour chaque, nous déterminons que nous avons toujours trois blobs (pour les contenus des trois fichiers), mais également deux arbres (ou trees) et un commit. Mais que contient un objet arbre ? Nous allons faire la commande pour obtenir ce que contient l'objet arbre racine (pour ne pas avoir à le chercher) :

```
git cat-file -p master\^{tree}
```

Ne retenez pas cette commande, nous avons besoin de voir de nombreuses choses avant pour l'expliquer. Nous obtenons :

```
040000 tree 3cdcfedb18b063a6bb4f1b43ddd73e2f88d5738a dossier
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d fichier1.txt
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f fichier2.txt
```

Comme nous l'avions vu l'arbre racine contient bien le hash vers les blobs qui sont contenus dans le dossier correspondant à l'arbre (ici le dossier racine) et le hash de l'arbre correspondant au dossier.

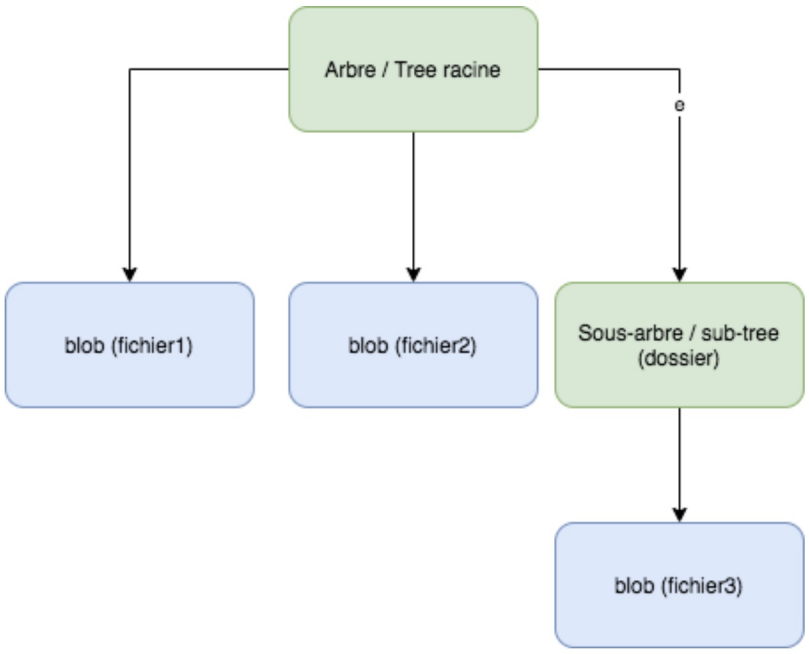
Nous retrouvons également le type (blob ou tree), les permissions et le nom du dossier ou des fichiers. Regardons ce que nous avons dans l'arbre inclus dans l'arbre racine :

```
git cat-file -p 3cdcfedb18b063a6bb4f1b43ddd73e2f88d5738a
```

Sans surprise nous retrouvons :

```
100644 blob 00750edc07d6415dcc07ae0351e9397b0222b7ba fichier3.txt
```

Nous avons donc :



Nous avons vu qu'un commit créait des objets arbres pour sauvegarder un instantané d'un projet : Git enregistre toute la structure du projet grâce à ces arbres. Mais que contient l'objet commit ? C'est ce que nous allons revoir maintenant. Nous faisons la commande suivante pour ne pas avoir à rechercher le hash de l'objet commit :

```
git cat-file -p master
```

Nous obtenons :

```
tree 1e784b993a4dc7956c84dda1d7060c56fde32ec8
author dymafr <erwanp.vallee@gmail.com> 1583754643 +0100
committer dymafr <erwanp.vallee@gmail.com> 1583754643 +0100
```

Premier commit

Nous voyons que le commit contient le hash de l'arbre racine et les métadonnées que nous avions déjà vus (auteur, committer, horodatage et message de validation). Nous comprenons maintenant que grâce au commit nous avons toutes les informations d'une sauvegarde et pouvons retrouver les bonnes versions des fichiers et la bonne structure grâce à l'arbre racine inclus dans le commit.

6 Supprimer un fichier ou un dossier

6.1 Effacer des fichiers avec Git

Effaçons notre fichier de test en utilisant la commande rm (pour remove) :

```
rm test.txt
```

Vérifions le statut de Git :

```
git status
```

```
Nous obtenons Modifications qui ne seront pas validées : supprimé : test.txt.
```

Essayons de faire un commit :

```
git commit -m "test"
```

Git nous dit qu'aucune modification n'a été sauvegardée et qu'il reste des modifications non validées (à savoir la suppression du fichier). En fait, le fichier est effacé de la version de travail appelée également copie de travail ou répertoire de travail, mais la suppression n'est pas indexée. C'est exactement la même chose que pour la modification d'un fichier, si nous n'indexons pas la modification elle ne sera pas sauvegardée. Procédons donc à l'indexation de la suppression :

```
git add test.txt
```

Oui, nous indexons bien un fichier qui n'existe plus. Git sait ainsi que nous souhaitons indexer la suppression de ce fichier. Refaisons maintenant un git status : la suppression apparaît maintenant en vert et Git indique qu'elle sera bien sauvegardée dans le prochain commit.

Il existe un raccourci pour ce que nous venons de faire, au lieu d'utiliser rm test.txt puis git add test.txt. Nous pouvons directement faire :

```
git rm test.txt
```

En effet, en faisant cette commande nous arrivons exactement au même résultat en faisant un git status :

```
git status
```

Nous donne, nous vous l'affichons cette fois en anglais pour que vous soyez habitué au deux :

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  deleted:    test.txt
```

Changes to be commited se traduisent par modifications qui seront validées, c'est-à-dire sauvegardées.

6.2 Effacer des fichiers uniquement de l'index

Vous pouvez utiliser l'option –cached avec git rm :

```
git rm --cached fichier
```

Dans ce cas le fichier (ou le dossier) sera supprimé de l'index (unstaged), mais restera dans le répertoire de travail.