Chapitre 4
Manipulation des fichiers et commit

1. Gestion des fichiers et commit

La fonction principale de Git est de suivre les différentes versions d'un projet. Un projet n'est rien d'autre qu'un ensemble de fichiers. De manière très pragmatique, Git ne s'intéresse qu'aux fichiers et ce sont les développeurs qui vont lui indiquer quels fichiers il doit suivre.

Le commit est l'élément central de Git. C'est la pierre angulaire qui permet de lier tous les éléments et tous les concepts de Git. Un commit représente un ensemble cohérent de modifications.

Par exemple, si un développeur corrige un bug, il va indiquer à Git le fichier dans lequel il a effectué la modification et Git va sauvegarder le fichier corrigé (tout en gardant l'ancienne version).

Avant d'aborder la suite du chapitre, il est utile de savoir comment Git voit un fichier.

2. Une histoire de hash

Un hash (qui peut également être appelé un condensat ou une signature) est une valeur calculée à partir d'une autre valeur. Dans la grande majorité des cas, cette valeur est représentée sous forme d'une chaîne de caractères hexadécimaux. Le calcul du hash fait appel à un algorithme complexe que ce livre ne détaillera pas.

Voici deux exemples de hash calculés avec l'algorithme SHA-1 :

Valeur	Hash
Git	5819778898df55e3a762f0c5728b457970d72cae
git	46f1a0bd5592a2f9244ca321b129902a06b53e03
Je veux une phrase assez longue, au moins plus que le hash en tous cas	7a5a57cde20a9bbda76b70e9223292ce7f8472f9

Dans cet exemple, nous remarquons deux choses :

– Un changement mineur dans le contenu change totalement le hash. Nous remarquons cela en comparant les hash de « Git » et « git ».

– Un hash fait toujours la même taille : 40 caractères (ce qui équivaut à 160 bits).

Il est impossible de retrouver le contenu original à partir du hash. Au mieux on peut essayer de le deviner, mais on ne peut avoir aucune certitude étant donné qu'un même hash peut correspondre à différentes chaînes. En effet, si on calcule le hash de $2^{160}+1$ chaînes différentes, on a forcément au moins une chaîne qui partage le hash d'une autre (voir la section Risque de collision).

Les hashs sont souvent utilisés pour vérifier qu'un fichier n'est pas corrompu ou alors pour authentifier un utilisateur sans devoir stocker son mot de passe en clair.

2.1 Une identification par contenu

En interne, Git travaille sur un certain nombre d'objets (contenus dans le dossier *.git/objects*) : des fichiers, des dossiers, des commits, etc.

Tous les éléments que Git manipule sont en réalité rangés dans des dictionnaires de paires clé/valeur dont la clé est le hash calculé en fonction du contenu. En réalité, Git permet de stocker des informations (comme le contenu d'un fichier) et nous donne un identificateur (le hash) nous permettant de récupérer ces données.

C'est-à-dire que Git ne va pas identifier un fichier *index.html* en fonction de son nom, mais plutôt à partir du hash généré à partir de son contenu. Cette méthode permet à Git de détecter facilement la moindre modification dans un fichier.

2.2 Risque de collision

De nombreux développeurs qui débutent avec Git se disent "Si un hash est noté avec 160 bits quel que soit le contenu en entrée alors il y a un risque de tomber sur un doublon". D'ailleurs avec toutes les versions différentes de fichiers et tous les commits, cela peut faire un grand nombre d'objets Git.

Ce risque est réel mais extrêmement minime. Tellement minime qu'il est réellement négligeable même pour les projets colossaux. En un peu plus de dix ans (entre avril 2005 et octobre 2015) le projet Linux a généré plus de 4 300 000 objets dans son dépôt Git. Ce nombre d'objets sera la base de l'exemple suivant.

Un chef de projet a peur de continuer à utiliser Git dans son entreprise à cause du risque de conflit. En effet, son projet est 500 fois plus prolifique (en termes d'objets Git) que le projet Linux. Pour le rassurer, il est possible de lui dire que si son équipe garde le même rythme pendant 1000 ans sans changer de système de gestion de versions, elle aura généré plus de 200 milliards d'objets Git. Pour prendre conscience de ce nombre il est possible de dire que les hashs (de 40 caractères hexadécimaux) des 200 milliards d'objets occupent à eux seuls 4 To d'espace disque.

Un script a calculé la probabilité de tomber sur une collision dans la vie d'un dépôt composé d'un peu plus de 201 milliards d'objets. Cette probabilité est inférieure à 1.3*10e-26. En comparaison, il existe une probabilité de 1.96*10e-15 qu'une météorite d'un mètre de diamètre tombe sur le chef de projet dans l'année suivante (puisqu'on estime qu'une météorite d'un mètre de diamètre tombe chaque année sur Terre et que la Terre fait 510 000 milliards de mètres carrés de surface). Le chef de projet a donc beaucoup plus de chance de se faire tuer par une météorite que de rencontrer un conflit d'identifiant d'objet Git.

En quelques mots, le risque de conflit est extrêmement négligeable et ne doit même pas être pris en considération. Tout le monde peut utiliser Git sans s'inquiéter de ce faux problème.

La formule suivante calcule la probabilité de tomber sur un conflit au cours d'un nième objet Git ajouté :

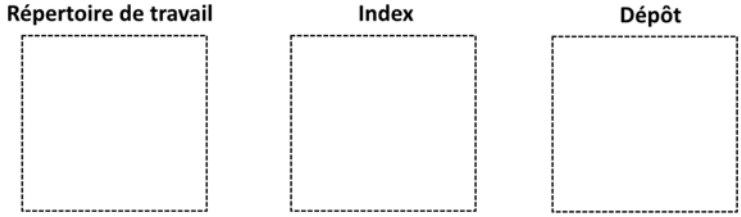
$$P_n = P_{n-1} + \frac{(1 - P_{n-1}) * (n - 1)}{2^{160}}$$

où la valeur de la probabilité de n = 1 est 0 puisqu'au premier objet créé il est impossible d'obtenir un conflit.

3. Les trois zones d'un fichier

Un fichier peut se trouver à trois endroits différents : le répertoire de travail, l'index et le dépôt. Le fichier va se trouver dans les différentes zones selon son avancée dans le projet.

Le schéma suivant montre un projet totalement vierge dans lequel un dépôt Git vient d'être initialisé, c'est-à-dire que nous n'avons enregistré aucun fichier dans le projet. La seule action effectuée sur ce projet est un `git init`.



Pour arriver à ce résultat, vous pouvez utiliser les commandes :

```
mkdir zoneFichier
cd zoneFichier
git init
```

Pour visualiser l'état des fichiers, nous allons utiliser la commande suivante :

```
git status
```

Cette commande affiche la sortie suivante :

```
git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

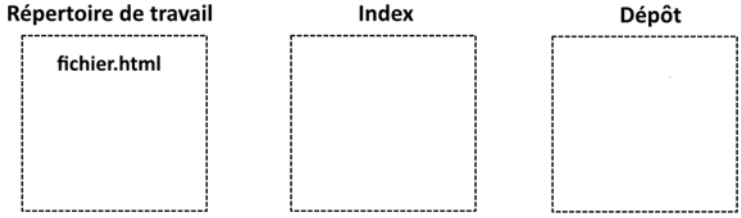
La commande `git status` permet d'afficher l'état des fichiers du dépôt. Cette commande va être expliquée à l'aide des exemples qui vont suivre dans la suite de ce chapitre.

Dans l'exemple précédent, la commande `git status` nous précise qu'aucun fichier n'est à commiter (*nothing to commit*).

3.1 Le répertoire de travail

Cette zone correspond au répertoire du système de fichiers sur lequel travaille le développeur. C'est le dossier du projet tel qu'il est stocké sur le disque dur. Les fichiers qui se trouvent dans cette zone peuvent être connus de Git selon qu'ils ont été ajoutés au moins une fois dans Git ou non. Un fichier qui se trouve uniquement dans cette zone est un fichier totalement inconnu pour Git.

Ce genre de fichier est également appelé fichier non suivi (*untracked file* en anglais).



Pour arriver à ce résultat, vous pouvez utiliser les commandes :

```
echo "<html>Le fichier est dans le répertoire de travail</html>"
> fichier.html
```

La commande `git status` nous indique que le fichier est visualisé en tant que fichier non suivi :

```
git status
On branch master

Initial commit

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

    fichier.html

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

Le fichier *fichier.html* n'est pas suivi par Git. Il n'est pas encore identifié par Git et ne possède donc pas encore d'identifiant sous forme de hash. Git possède une commande de plomberie qui permet de visualiser les fichiers non suivis. Cette commande présente l'avantage d'afficher l'identifiant du fichier lorsque cet identifiant existe. Voici la commande qui permet de visualiser les fichiers non suivis :

```
git ls-files --others
```

Elle affiche la sortie suivante :

```
fichier.html
```