Cours Inf3523 – Architecture et Génie des Logiciels

Lab_2 : Travailler localement avec GIT

Ce lab a pour but de nous montrer comment Git fonctionne en interne. Cela nous permettra de travailler efficacement avec git dans un dépôt local. Nous allons explorer les objets Git tels que les Trees, Staging area, HEAD, etc.

Objets Git

Dans ce lab, nous utilisons Git pour créer un fichier contenant une « *liste de courses à faire, avant d'aller au supermarché* » ; alors, créez un nouveau dossier **grocery**, puis initialisez un nouveau dépôt Git.

Un dépôt Git vide a été créé dans **C:/Repos/grocery**. Comme nous l'avons déjà vu précédemment, le résultat de la commande **git init** est la création d'un dossier nommé : **.git**, où Git stocke tous les fichiers dont il a besoin pour gérer notre dépôt :

Ainsi, nous pouvons déplacer ce dossier "**grocery**" où nous le souhaitons sans perdre aucune donnée.

Une autre chose importante à souligner est que nous n'avons pas besoin d'un serveur : nous pouvons créer un référentiel localement et y travailler quand nous le souhaitons, même sans connexion LAN ou Internet. Nous n'en avons besoin que si nous voulons partager notre référentiel avec quelqu'un d'autre, directement ou en utilisant un serveur central.

NB : Au cours de ce lab, nous n'utiliserons aucun serveur distant, car cela n'est pas nécessaire.

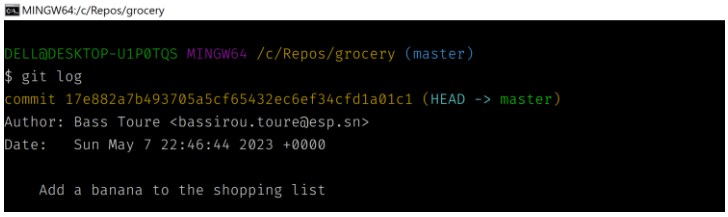
Continuez et créez un nouveau fichier **README.md** qui va vous rappeler le but de ce référentiel. Ensuite, ajoutez une **banane** à la liste de courses.

À ce stade, comme vous le savez déjà, avant de faire un commit, nous devons ajouter des fichiers à l'aire de **staging** ; ajoutez les deux fichiers en utilisant le raccourci **git add .** (le point après la commande **git add**).

Avec cette commande, vous pouvez ajouter tous les fichiers nouveaux ou modifiés en une seule fois.

Commits

Maintenant, il est temps de commencer à examiner les commits. Pour vérifier le commit que nous venons de créer, nous pouvons utiliser la commande **git log** :



Comme vous pouvez le voir, git log montre le commit que nous avons fait dans ce référentiel; git log montre tous les commits, dans l'ordre chronologique inverse; nous n'avons qu'un seul commit pour le moment, mais ensuite, nous verrons cette fonctionnalité en action.

Le Hash

Il est maintenant temps d'analyser les informations fournies. La première ligne contient le **SHA-1** du commit (https://en.wikipedia.org/wiki/SHA-1), une séquence alphanumérique de 40 caractères représentant un nombre hexadécimal. Ce code, ou **hash**, comme on l'appelle généralement, identifie de manière unique le commit au sein du dépôt, et c'est grâce à lui que nous pouvons désormais nous y référer pour effectuer certaines actions.

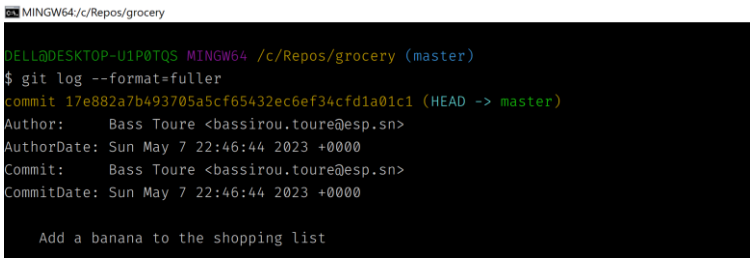
L'auteur et la date de création du commit

Nous avons déjà parlé des auteurs ; l'auteur est la personne qui a réalisé le commit, et la date est la date complète à laquelle le commit a été créé. Depuis cette instance, ce commit fait partie du référentiel (repository).

Le message de commit

Juste en dessous de l'auteur et de la date, après une ligne vide, nous pouvons voir le message que nous avons attaché au commit que nous avons effectué ; même le message fait partie du commit lui-même.

Mais il y a quelque chose de plus ; essayons d'utiliser la commande **git log** avec l'option **--format=fuller**



Le commiteur et la date de commit

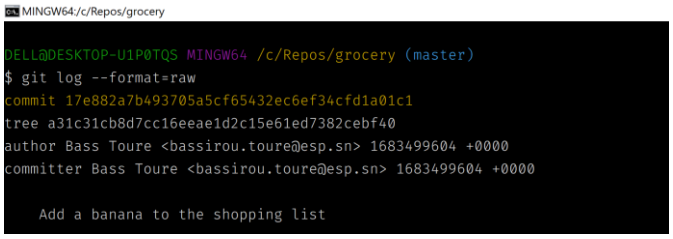
Un commit préserve également le commiteur et la date de commit; quelle est la différence par rapport à l'auteur et à la date de création ? Tout d'abord, ne vous inquiétez pas trop à ce sujet : 99 % des commits dans votre référentiel auront les mêmes valeurs pour l'auteur et le commiteur, ainsi que les mêmes dates.

Dans certaines situations, telles que le ***cherry-pick***[1], vous transportez un commit existant sur une autre branche, en créant un nouveau commit qui applique les mêmes modifications que précédemment. Dans ce cas, l'auteur et la date de création restent les mêmes, tandis que le commiteur et la date de commit sont liés à la personne qui a effectué cette opération et à la date à laquelle elle l'a effectuée.

[1] A voir plus tard

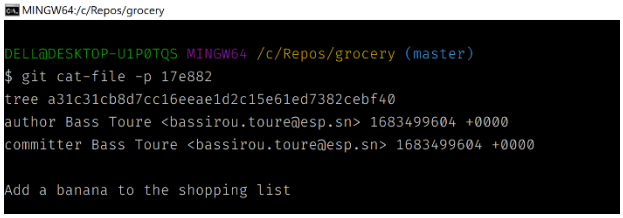
Les arbres (ou trees en anglais)

Utilisons à nouveau la commande **git log**, en utilisant l'option **--format=raw** :



Cette fois-ci, le format de sortie est différent; nous pouvons voir l'auteur et le commiteur, comme nous l'avons vu précédemment, mais de manière plus compacte; ensuite il y a le message de commit, mais quelque chose de nouveau apparaît: **c'est un arbre**.

Nous pouvons voir une nouvelle commande **git cat-file -p** qui nous permet de faire focus sur un élément. Pour que cela fonctionne, nous devons spécifier l'objet que nous voulons étudier ; nous pouvons utiliser le hachage de l'objet, notre premier commit dans ce cas. Vous n'avez pas besoin de spécifier l'intégralité du hachage, mais les cinq à six premiers caractères suffisent pour les petits dépôts. Git est suffisamment intelligent pour comprendre quel est l'objet même avec moins de 40 caractères ; le minimum est de quatre caractères, et le nombre augmente à mesure que le nombre total d'objets Git dans le dépôt augmente.



Les **arbres** sont des **conteneurs** pour les **blobs** et autres arbres. La façon la plus simple de comprendre comment cela fonctionne est de penser aux dossiers dans votre système d'exploitation, qui collectent également des fichiers et d'autres sous-dossiers à l'intérieur d'eux.

Blobs

Comme vous pouvez le voir, à droite de la sortie de la commande précédente, nous avons **README.md** et **shoppinglist.txt**, ce qui nous fait deviner que les blobs Git représentent les fichiers. Comme précédemment, nous pouvons vérifier son contenu est exactement le contenu de notre fichier **shoppingFile.txt**; voyons ce qu'il y a à l'intérieur de **637a0**:

Les blobs sont des fichiers binaires, rien de plus, rien de moins. Ces séquences de bytes, qui ne peuvent pas être interprétées à l'œil nu, contiennent les informations de n'importe quel fichier, qu'il s'agisse de fichiers binaires ou textuels, d'images, de code source, d'archives, etc. Tout est compressé et transformé en blob avant d'être archivé dans un référentiel Git. Comme déjà mentionné précédemment, chaque fichier est marqué avec un hash; cet hash identifie de manière unique le fichier au sein de notre référentiel, et c'est grâce à cet ID que Git peut le récupérer lorsque nécessaire, et détecter les changements lorsque le même fichier est modifié (des fichiers avec un contenu différent auront des hashs différents).

Nous avons dit que les hashs SHA-1 sont uniques; mais qu'est-ce que cela signifie ? Mettons la commande : **git hash-object** :

La commande **git hash-object** est la commande pour calculer le hachage de n'importe quel objet ; dans cet exemple, nous avons utilisé l'option **--stdin** pour transmettre comme argument de commande le résultat de la commande précédente, **echo "banana"** ; en quelques mots, nous avons calculé le hachage de la chaîne "**banana**", et il est sorti **637a09b86af61897fb72f26bfb874f2ae726db82**.

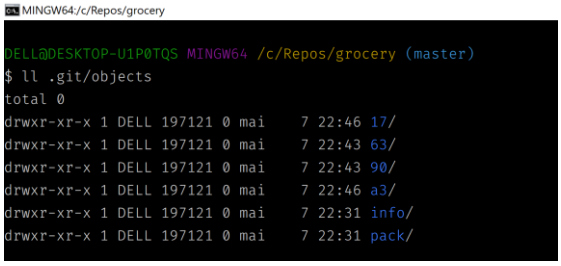
Et sur votre ordinateur ? C'est le même résultat !

Vous pouvez essayer de ré exécuter la commande autant de fois que vous le souhaitez, le hachage résultant sera toujours le même (si ce n'est pas le cas, cela peut être dû à des fins de ligne différentes dans votre système d'exploitation ou votre Shell).

Cela nous fait comprendre quelque chose de très important : un objet, quel qu'il soit, aura toujours le même hachage dans n'importe quel dépôt, sur n'importe quel ordinateur.

Enfin, mais non des moindres, je tiens à souligner comment Git calcule le hachage sur le contenu du fichier, en effet, le hachage **637a09b86af61897fb72f26bfb874f2ae726db82** calculé en utilisant **git hash-object** est le même que le blob que nous avons inspecté précédemment en utilisant **git cat-file -p**. Cela nous donne une information importante : si vous avez deux fichiers différents avec le même contenu, même s'ils ont des noms et des chemins différents, dans Git, vous finirez par avoir un seul blob.

Inspectons le dossier **.git**, plus précisément son sous dossier **objects**.



Mis à part les dossiers info et pack qui ne nous intéressent pas pour le moment, comme vous pouvez le voir, il y a d'autres dossiers avec un nom étrange de deux caractères. Entrez dans le dossier **63**:

Remarque : **63 + 7a09b86af61897fb72f26bfb874f2ae726db82** est en fait le hachage de notre **blob shoppingList.txt** !

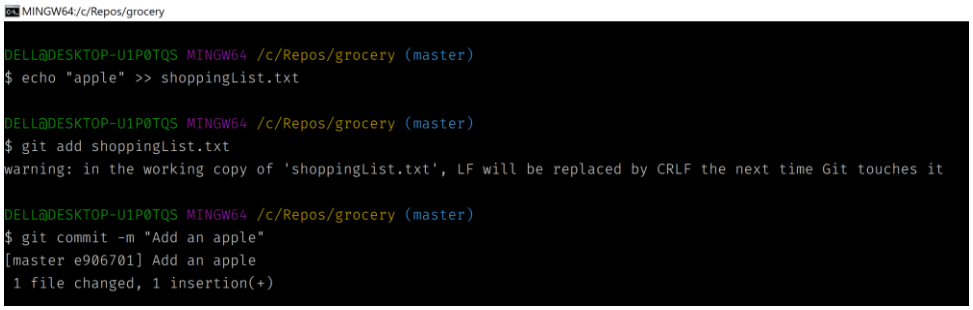
Pour être plus rapide lors de la recherche dans le système de fichiers, Git crée un ensemble de dossiers où le nom est long de deux caractères, et ces deux caractères représentent les deux premiers caractères d'un code de hachage; à l'intérieur de ces dossiers, Git écrit tous les objets en utilisant comme nom les 38 autres caractères du hachage, indépendamment du type d'objet Git. Ainsi, l'arbre **a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40** est stocké dans le dossier **a3**.

Maintenant, si vous essayez d'inspecter ces fichiers avec une commande **cat** commune, vous verrez que ces fichiers sont des fichiers texte simples, mais Git les compresse à l'aide de la bibliothèque **zlib** pour économiser de l'espace sur votre disque. C'est pourquoi nous utilisons la commande **git cat-file -p**, qui les décompresse à la volée pour nous.

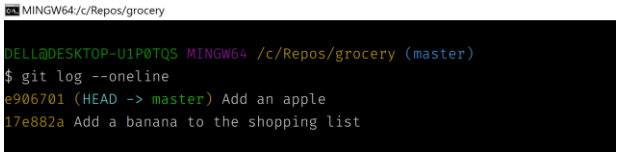
Cela met une fois de plus en évidence la simplicité de Git : pas de métadonnées, pas de bases de données internes ou de complexité inutile, mais des fichiers et des dossiers simples suffisent à permettre la gestion de n'importe quel référentiel.

À ce stade, nous savons comment Git stocke les objets et où ils sont archivés ; nous savons également qu'il n'y a pas de base de données, de référentiel central ou de truc comme ça, alors comment Git peut-il reconstruire l'historique de notre référentiel ? Comment peut-il définir quel commit précède ou suit un autre ?

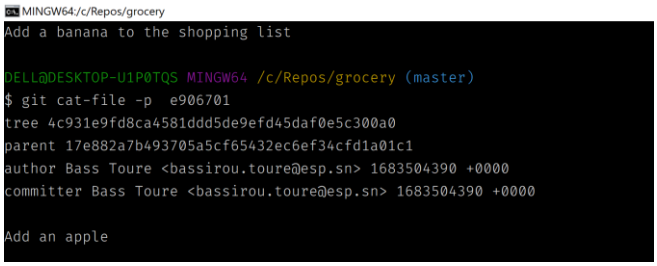
Pour prendre conscience de cela, nous avons besoin d'un nouveau commit. Donc, passons maintenant à la modification du fichier **shoppingList.txt** :



Utilisons la commande "**git log**" pour vérifier le nouveau commit ; l'option "**--oneline**" nous permet de voir le journal de manière plus compacte :



Okay, nous avons un nouveau commit, avec son hash. Il est temps de voir ce qu'il contient :



Il y a quelque chose de nouveau : le parent **17e882a7b493705a5cf65432ec6ef34cfd1a01c1**. Un parent d'un commit est simplement le commit qui le précède. En fait, le hash **17e882** est en réalité le hash du premier commit que nous avons fait. Ainsi, chaque commit a un parent, et en suivant ces relations entre les commits, nous pouvons toujours naviguer depuis n'importe quel commit jusqu'au premier, le commit racine (**root** commit) déjà mentionné.

Si vous vous en souvenez, le premier commit n'avait pas de parent, et c'est la principale (et unique) différence entre tous les commits et le premier. Git, tout en naviguant et en reconstruisant notre référentiel, sait simplement qu'il a terminé lorsqu'il trouve un commit sans parent.

Git n'utilise pas les deltas

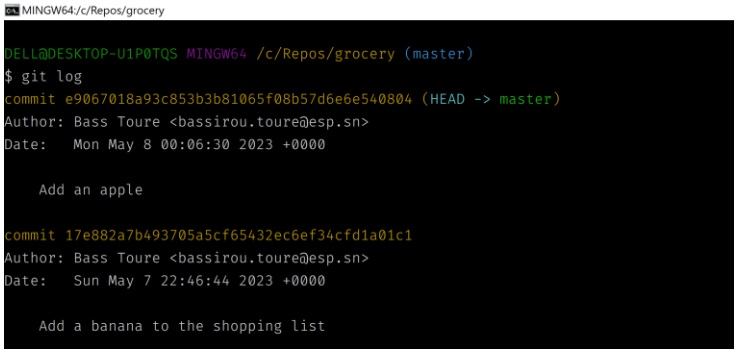
Il est maintenant temps d'examiner une autre différence bien connue entre Git et d'autres systèmes de versioning. Prenons Subversion comme exemple : lorsque vous effectuez un nouveau commit, Subversion crée une nouvelle révision numérotée qui ne contient que les deltas par rapport à la précédente. C'est une façon intelligente d'archiver les changements apportés aux fichiers, en particulier pour les gros fichiers texte, car si seule une ligne de texte change, la taille du nouveau commit sera beaucoup plus petite.

En revanche, dans Git, même si vous ne modifiez qu'un seul caractère dans un gros fichier texte, il stocke toujours une nouvelle version du fichier : Git n'utilise pas les deltas (du moins pas dans ce cas), et chaque commit est en fait un instantané de l'ensemble du dépôt.

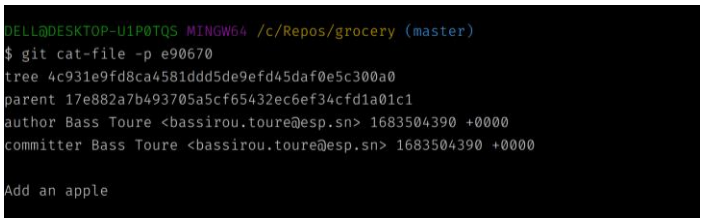
À ce stade, les gens pensent que Git gaspille une grande quantité d'espace disque en vain ! Ce n'est pas le cas.

Dans un dépôt de code source classique, avec un certain nombre de commits, Git n'a généralement pas besoin de plus d'espace que les autres systèmes de versioning. Par exemple, lorsque Mozilla est passé de Subversion à Git, le même dépôt est passé de 12 Go à 420 Mo d'espace disque ; Pour en savoir plus : https://git.wiki.kernel.org/index.php/GitSvnComparison

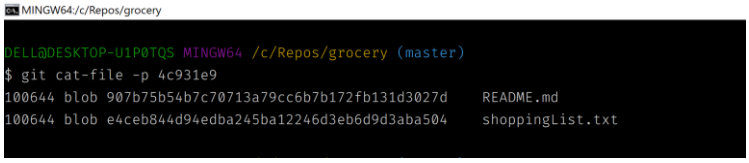
De plus, Git a une façon astucieuse de gérer les fichiers :



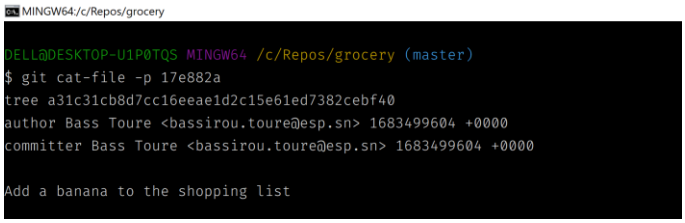
Examinons le dernier commit :



Ensuite son arbre :



Puis le premier commit



Puis son arbre

Le hachage du fichier **README.md** est le même dans les deux arbres des premier et deuxième commits ; cela nous permet de comprendre une autre stratégie simple mais intelligente adoptée par Git pour gérer les fichiers. Lorsqu'un fichier est inchangé, lors de la validation, Git crée un arbre où le blob du fichier pointe vers celui déjà existant, le recyclant ainsi et évitant le gaspillage d'espace disque.

La même chose s'applique aux arbres : si mon répertoire de travail contient des dossiers et des fichiers qui resteront inchangés, lorsqu'un nouveau commit est effectué, Git recycle les mêmes arbres.

Les références

Dans la section précédente, nous avons vu qu'un dépôt Git peut être imaginé comme un arbre qui, à partir d'une racine (le commit racine), se développe vers le haut à travers une ou plusieurs branches.

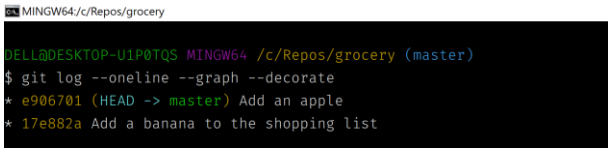
Ces branches sont généralement distinguées par un nom. Sur ce point, Git ne fait pas exception, les commandes précédentes nous ont amenés à valider sur la branche principale (master) de notre dépôt de test. Master est le nom de la branche par défaut d'un dépôt Git.

Branches

Dans Git, une branche n'est rien de plus qu'une étiquette ou label, une étiquette mobile placée sur un commit.

En fait, chaque feuille d'une branche Git doit être étiquetée avec un nom significatif pour nous permettre de la référencer, puis de nous déplacer, de revenir en arrière, de fusionner, de refaire ou de supprimer certains commits lorsque cela est nécessaire.

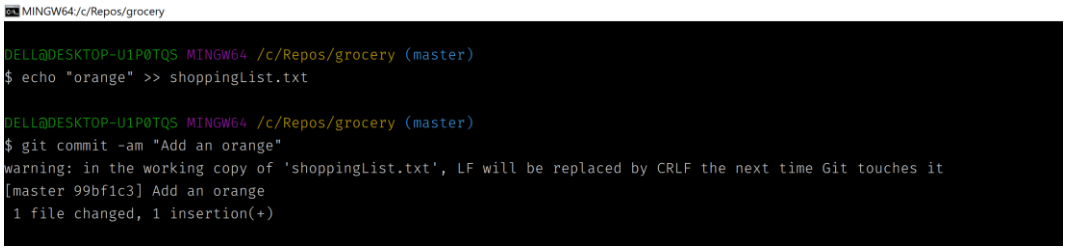
Dans notre dépôt **grocery**; faisons la commande **git log**, en ajoutant de nouvelles options :



Jetons un coup d'œil à ces options en détail :

- **--graph** : Dans ce cas, cela ajoute simplement un astérisque à gauche, avant le hachage du commit, mais lorsque vous avez plusieurs branches, cette option les dessinera pour nous, en fournissant une représentation graphique simple mais efficace du dépôt.
- **--decorate** : Cette option affiche les étiquettes attachées aux commits qui sont affichés ; dans ce cas, elle affiche (**HEAD -> master**) sur le commit **e906701**.
- **--oneline** : C'est facile à comprendre : cela rapporte chaque commit en utilisant une seule ligne, raccourcissant les choses, si nécessaire.

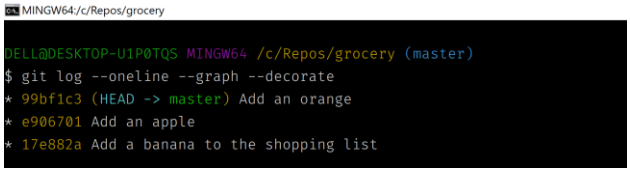
Nous allons maintenant effectuer un nouveau commit et voir ce qui se passe :



Après avoir ajouté une **orange** à la liste de courses (**shopingList.txt**), j'ai effectué un commit sans avoir d'abord fait de **git add** ; l'astuce réside dans l'option **-a (--add)** ajoutée à la commande **git commit**, ce qui signifie ajouter à ce commit tous les fichiers modifiés que j'ai déjà validés au moins une fois auparavant. Dans notre cas, cette option nous a permis d'aller plus vite et de sauter la commande **git add**.

Quoi qu'il en soit, utilisez-la avec prudence, vous risquez facilement de faire un commit avec plus de fichiers que vous ne le souhaitez.

D'accord, maintenant continuez et jetez un coup d'œil à la situation actuelle du dépôt :



Intéressant ! À la fois **HEAD** et **master** ont maintenant avancé sur le dernier commit, le troisième.

Les branches sont des étiquettes mobiles

Nous avons vu dans les sections précédentes comment les commits sont liés les uns aux autres par une relation parent-enfant : chaque commit contient une référence vers le commit précédent.

Cela signifie que, par exemple, pour naviguer dans un dépôt, je ne peux pas partir du premier commit et essayer d'aller au suivant, car un commit n'a pas de référence vers ce qui vient après, mais vers ce qui vient avant. En restant dans notre métaphore arboricole, cela signifie que notre arbre est navigable uniquement à partir des feuilles, de l'extrémité supérieure d'une branche, puis vers le commit racine.

Ainsi, les branches ne sont rien d'autre que des étiquettes qui se trouvent sur le dernier commit, le plus récent. Ce commit, notre feuille, doit toujours être identifié par une étiquette afin que ses ancêtres puissent être atteints lors de la navigation dans un dépôt. Sinon, nous devrions nous souvenir pour chaque branche de notre dépôt du code de hachage du dernier commit, et vous pouvez imaginer à quel point cela serait facile pour les humains.

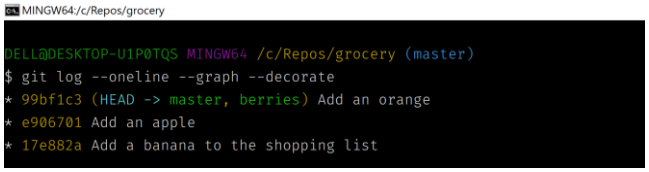
Création d'une nouvelle branche

Voyons ce qui se passe lorsque vous demandez à Git de créer une nouvelle branche. Puisque nous allons servir une délicieuse salade de fruits, il est temps de réserver une branche pour une recette variante aux baies :

C'est tout ! Pour créer une nouvelle branche, il vous suffit d'appeler la commande git branch suivie du nom de la branche que vous souhaitez utiliser. Et c'est super rapide ; travaillant toujours en local, Git effectue ce genre de tâche en un clin d'œil.

Pour être honnête, il y a quelques règles (compliquées) à respecter et des choses à savoir sur les noms possibles d'une branche (tout ce que vous devez savoir se trouve ici : https://git-scm.com/docs/git-check-ref-format), mais pour l'instant, ce n'est pas pertinent.

Donc, git log à nouveau :

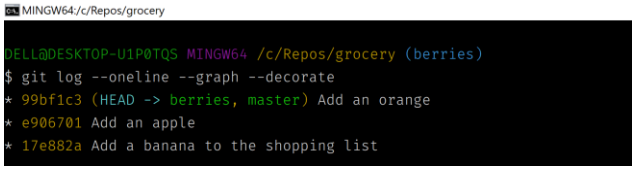


Maintenant, Git nous indique qu'il existe une nouvelle branche, "berries", et qu'elle fait référence au même commit que la branche "master".

Cependant, pour le moment, nous sommes toujours situés dans la branche "master". En effet, comme vous pouvez le voir dans l'invite de commande de la console, il continue d'apparaître (master) entre les parenthèses.

Comment puis-je changer de branche ? En utilisant la commande "git checkout" :

En faisant un git log on a :



Maintenant, il y a un signe (berries) dans l'invite de commande de la console, et en plus, quelque chose s'est passé avec ce truc "HEAD" : maintenant les flèches pointent vers "berries", plus vers "master".

HEAD, ou tu es ici

Au cours des exercices précédents, nous avons continué à voir cette chose appelée "HEAD" lors de l'utilisation de la commande git log, et maintenant il est temps d'enquêter un peu.

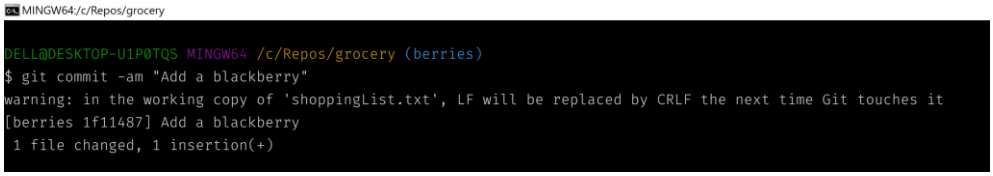
Tout d'abord, qu'est-ce que HEAD ? Comme les branches, HEAD est une référence. Il représente un pointeur vers l'endroit où nous nous trouvons actuellement, rien de plus, rien de moins. En pratique, c'est simplement un autre fichier texte :

La différence entre le fichier HEAD et le fichier texte des branches est que le fichier HEAD se réfère généralement à une branche, et non directement à un commit comme le font les branches. La partie "ref:" est la convention utilisée par Git en interne pour déclarer un pointeur vers une autre branche, tandis que "refs/heads/berries" est bien sûr le chemin relatif vers le fichier texte de la branche "berries".

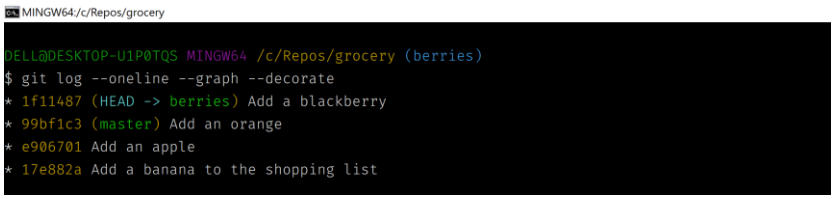
Donc, en ayant effectué le checkout de la branche "berries", nous avons en fait déplacé ce pointeur de la branche "master" vers la branche "berries" ; à partir de maintenant, chaque commit que nous ferons fera partie de la branche "berries". Essayons.

Ajoutez blackberry à la liste de courses :

Ensuite, un commit



Faisons un git log



Commentaire :

- La branche "berries" s'est déplacée vers le dernier commit que nous avons effectué, confirmant ce que nous avons dit précédemment : une branche est simplement une étiquette qui vous suit lors de la création de nouveaux commits, se collant au dernier.
- Le pointeur HEAD s'est également déplacé, suivant la branche à laquelle il est actuellement pointé, c'est-à-dire "berries".
- La branche "master" reste à sa position initiale, attachée à l'avant-dernier commit, le dernier que nous avons fait avant de passer à la branche "berries".

Maintenant notre fichier **shoppingList.txt** semble contenir ces lignes de texte :

Que se passe-t-il si nous revenons à la branche "master" ?

Passons à la branche "master" :

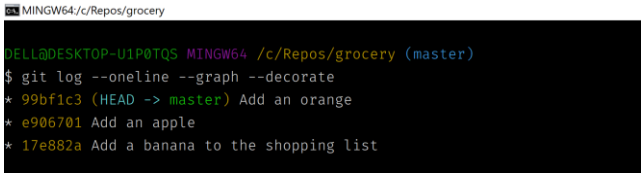
Basculé sur la branche 'master'

Regardons le contenu du fichier **shoppingList.txt** :

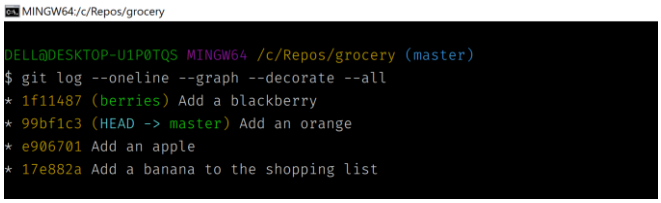
Nous sommes effectivement revenus à notre état précédent avant d'ajouter la mûre ; puisqu'elle a été ajoutée dans la branche "berries", elle n'existe pas ici dans la branche "master" : ça a l'air bien, n'est-ce pas ?

Même le fichier HEAD a été mis à jour en conséquence :

Lorsque vous changez de branche, Git se rend au commit vers lequel la branche pointe, puis en suivant la relation parent et en analysant les arbres et les blobs, il reconstruit le contenu du répertoire de travail en conséquence, récupérant ainsi les fichiers et les dossiers associés.



`git log` affiche généralement uniquement la branche sur laquelle vous vous trouvez et le commit qui lui appartient. Pour voir toutes les branches, il vous suffit d'ajouter l'option `--all` :



Accessibilité et annulation des commits