Git

1. Git, la théorie

1.1. Le système **distribué**

Git est un logiciel de version **distribué**, c'est-à-dire que le logiciel permet aux utilisateurs de cloner une copie **locale**, donc sur son propre ordinateur, des fichiers stockés sur le **serveur**, en opposition aux logiciels de version locaux, qui ne conservent que les fichiers en local, et centralisés, qui ne conservent que les fichiers sur le serveur.

En tant que système distribué, Git permet aux utilisateurs de faire les modifications en local d'abord, donc de profiter de la rapidité d'accès aux fichiers sur son propre ordinateur, et leur permet ensuite d'envoyer ces modifications sur le serveur, pour pouvoir par exemple les partager avec d'autres personnes.

L'autre avantage est que les données sont ainsi stockées sur deux supports, l'ordinateur en local et le serveur, pour pouvoir pallier à un éventuel problème technique et donc une éventuelle perte des données.

Pour en savoir plus sur les différents systèmes de gestion de versions, voir la documentation Git sur : https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

1.2. Le **working directory**, l'**index** et le **repository**

Pour enregistrer les modifications, Git sépare le processus en trois étapes :

1. Le **working directory**, littéralement *espace de travail* qui est votre dossier Git ou vous éditez vos fichiers sources.

2. L'**index**, aussi appelé *zone de staging*, qui est une zone intermédiaire entre votre espace de travail et le repository, celle-ci permet de stocker temporairement les changements qui seront committés.

3. Le **repository**, ou *dépôt*, est la zone ou les modifications sont finalement committées, c'est-à-dire enregistrées, c'est à partir de cette zone que les modifications sont finalement envoyées sur le serveur.

1.3. Le système de branches et la branche **master**

Pour représenter les différents enregistrements dans le repository, appelés **commits**, Git utilise un système de branches, avec une **branche principale** appelée **master**, ou branche maître.

Les commits sont **identifiés** de **manière unique** avec une **clé SHA-1**, c'est un identifiant unique qui permet de se positionner sur un commit particulier. C'est ici la clé *6de7b0a* pour le dernier commit.

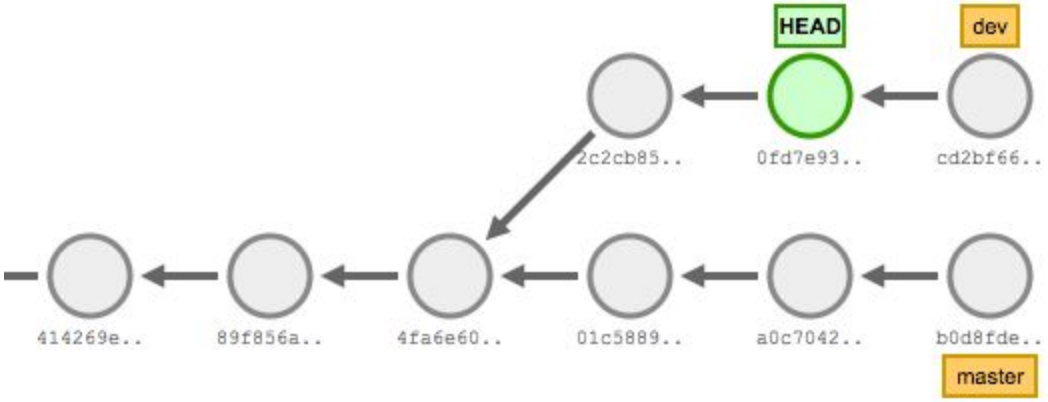
Par **positionner**, j'entend déplacer la **tête de lecture**, appelée **HEAD**, cette tête de lecture est en quelque sorte votre *oeil* sur vos différents commits, lorsque vous vous déplacez sur une branche, c'est en réalité **HEAD** qui se déplace.

Par ailleurs, **HEAD** représente le commit ou vous vous situez actuellement. Ici représenté par la clé *ee7aab9*.

Le **nom de la branche**, ici *master*, **représente** toujours le **dernier commit** effectué.

Et très important, on ne peut **commit** une nouvelle modification qu'en étant **positionné** sur le **dernier commit** effectué, c'est-à-dire celui représenté par *master*.

Lorsqu'on crée une **nouvelle branche**, on peut positionner **HEAD** sur un commit d'une autre branche, liée à la branche principale *master*. Le nom de la branche, ici *dev*, représente le *dernier commit* effectué également.



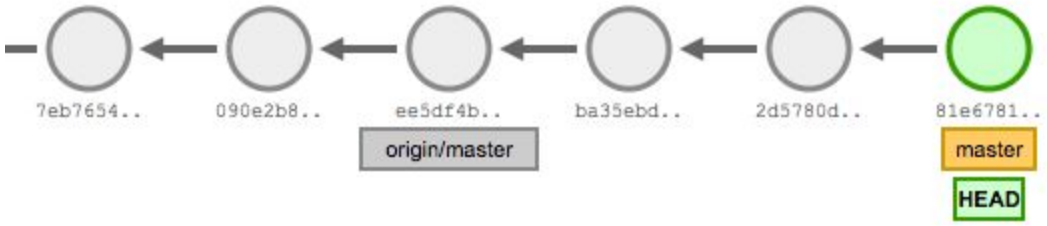
1.4. Le système de **remote**

Git permet de lier votre projet à un serveur distant appelé **remote**. Il est possible de lier plusieurs remote. On crée souvent un *remote* principal **origin**.

On donne souvent le nom **origin** à ce *remote*, mais c'est n'est pas une obligation.

Les branches **branch** sur le serveur distant **remote** peuvent être récupérées sur le *working directory*, elles prendront le nom **remote/branch** dans celui-ci.

Lorsqu'on récupère la branche *master* depuis le serveur distant *origin*, celle-ci prendra donc le nom *origin/master* sur le répertoire local.



Sur l'image ci-dessus, la branche locale *master* est en avance de 3 commits sur la branche remote *origin/master* qui a été récupérée depuis le serveur.

A noter que lorsqu'on envoie les modifications sur le serveur depuis la **branche actuelle**, il faudra alors **spécifier** sur **quel remote** et sur **quelle branche** on souhaite envoyer celles-ci.

Pour pallier à ce problème, il est possible de **lier et tracker** les modifications de la branche locale **master** (ou alors **branch**) à la branche distante **origin/master** (ou alors **remote/branch**).

2. Effectuer des commits

2.1. Vos premiers commits

Tout d'abord, placez-vous dans un répertoire et exécutez la commande **git init** :

```
git init
```

Cette commande va initialiser votre dossier comme un dossier Git en créant un dossier .git à l'intérieur.

Si vous avez déjà un **repository** Git de disponible, ou un **repository** sur un **serveur** distant **remote**, **cloner** le **avec** la commande **git clone** :

```
git clone <path-or-url> <folder-name>
```

Le dernier paramètre permet de **choisir** le **nom du dossier** ou le **repository** sera **cloné**.

Lorsque vous ajoutez un nouveau fichier dans votre *working directory*, ce fichier est dit **untracked**, c'est-à-dire *non-suivi*.

Pour *suivre* ce fichier, il faut d'abord le **tracker**, puis le **stage**, c'est-à-dire l'**indexer**, ou l'ajouter à la *zone de staging*.

Pour **tracker et stage** ou **stage**, on utilise la commande **git add** :

```
git add <file>
```

Pour **tracker et stage** ou **stage** tous les fichiers d'un coup :

```
git add .
```

Une fois le fichier *tracked* et *staged*, vous pouvez donc le **commit** :

```
git commit -m "votre message de commit"
```

Modifier votre fichier staged vous obligera à **re-stage** celui-ci : en effet, les modifications ne seront *pas indexées* donc elle ne seront *pas prises en compte* pour le prochain commit.

En revanche, vos fichiers seront toujours trackés. Vous pouvez d'ailleurs **re-stage** tous les fichiers **déjà trackés** mais **unstaged** avec l'option **-a** au moment du commit :

```
git commit -a -m "votre message de commit"
```

Si jamais vous avez oublié un fichier lors de votre dernier commit, vous pouvez committer celui-ci avec l'option **--amend**, cela mettra à jour le dernier commit :

```
git commit --amend
```

Pour **untracker** un fichier, utilisez la commande **git rm --cached**, option **-r** pour récursif et **-f** pour force :

```
git rm --cached <file>
```

Pour **untracker** et définitivement **supprimer** un fichier utilisez **git rm** :

```
git rm <file>
```

2.2. Afficher le statut des fichiers et les différents commits

Vous pouvez à tout moment afficher le **statut** de vos fichiers (*untracked*, *tracked*, *unstaged*, *staged*) dans votre *working directory* avec la commande **git status** :

```
git status
```

Afficher un commit particulier avec **git show** :

```
git show <commit-id> <file>
```

L'identifiant du commit peut-être sa clé SHA-1, **HEAD** ou le nom de la branche pour représenter le dernier commit.

HEAD représente le commit ou vous vous situez actuellement, **HEAD^** représente le commit juste avant celui où vous situez, et **HEAD^^** représente le commit encore d'avant. **HEAD~3** représente le 3ème commit avant celui où vous vous situez et ainsi de suite...

Si vous ne spécifiez pas de fichier, **tous les fichiers** affectés par le commit seront **affichés**.

Afficher la liste des commits avec **git log**, utilisez l'option **--oneline** pour un affichage sur une seule ligne :

```
git log --oneline <file>
```

Utilisez l'option **-p** pour afficher les modifications apportées à chaque fichier.

Pour afficher les différences entre commits, utilisez **git diff**.

Entre le working directory et l'index :

```
git diff <file>
```

Entre le working directory et le HEAD :

```
git diff HEAD <file>
```

Entre l'index et le HEAD (*cache-HEAD*) :

```
git diff --cached <file>
```

Entre deux commits particuliers :

```
git diff <commit-id> <commit-id>
```

Utilisez le nom de la branche pour se référer au dernier commit.

La encore, si vous ne spécifiez pas de fichier, cela concerne tous les fichiers.

Vous pouvez combiner **git diff --cached** et **git status** avec l'option **-v** de git status.

Vous pouvez afficher les changements ligne par ligne d'un fichier et pour chaque commit avec **git blame** :

```
git blame <file>
```

3. Éditer l'historique

3.1. Éditer l'index ou les fichiers du working directory

L'**index** est une *zone de staging* qui stocke les modifications destinées à être committées par la suite. Les fichiers trackés sont donc stagés et sauvegardés dans cet *index*.

Il arrive que l'on veuille **unstager** des fichiers de l'**index**. Les fichiers seront toujours trackés par Git mais la copie de l'index sera modifiée. **Modifier** la **copie de l'index** par la copie du **dernier commit** provoque donc un **unstage**, donc un écrasement des modifications de l'**index**.

Pour cela on utilise la commande **git reset** :

```
git reset HEAD <file>
```

Le **HEAD** signifie ici que l'on remplace la copie de l'**index** avec la **version stockée** dans le **repository** sur le commit représenté par **HEAD**. On peut aussi renseigner la **clé SHA-1** d'un commit particulier également.

A noter que si l'on veut remplacer la copie de l'index avec la **version** du **dernier commit**, on peut faire :

```
git reset master <file>
```

La ou **master** représente le dernier commit sur la *branche principale*, on notera aussi que **HEAD** est **sous-entendu** si on ne renseigne rien :

```
git reset -- <file>
```

Mais la commande **git reset** ne modifie que l'index et pas les fichiers du **working directory**, pour modifier les fichiers du **working directory**, on utilise **git checkout** :

```
git checkout <commit-id> <file>
```

Pour modifier un fichier du *working directory* avec la version de l'avant avant-dernier commit (à noter que l'on peut aussi spécifier ici aussi la clé SHA-1) :

```
git checkout HEAD^^ <file>
```

Pour modifier un fichier avec la **version** du **HEAD** ou l'on **est placé** :

```
git checkout -- <file>
```

3.2. Se positionner, annuler ou supprimer des commits

Pour supprimer un commit, on utilise **git reset**, la commande suivante **supprime** les commits depuis le commit spécifié :

```
git reset <commit-id>
```

A noter que les **fichiers** dans votre *working directory* ne **sont pas modifiés** et restent tels qu'ils étaient au dernier commit supprimé, **pour modifier** les fichiers de votre working directory, **utilisez** l'option **--hard**.

Pour se placer sur un commit particulier, et visualiser les fichiers tels qu'ils étaient à ce commit là, il faut utiliser **git checkout** :

```
git checkout <commit-id>
```

Vous serez alors en mode **detached HEAD** littéralement "*tête détachée*", vous ne pourrez pas committer les modifications en mode **detached HEAD**, rappelez-vous qu'on ne peut **committer** une **nouvelle modification** qu'en étant **positionné** sur le **dernier commit** effectué, c'est-à-dire celui représenté par **master**.

Pour annuler un commit, utilisez **git revert** :

```
git revert <commit-id>
```

La commande **git revert** va créer un **nouveau commit**, mais avec les **modifications inverses** du commit spécifié.

Rappelons aussi que l'identifiant de commit peut-être la **clé SHA-1** ou le mot-clé **HEAD**.

Les commandes **reset** et **checkout** n'ont donc pas le même but :

Commande	Appliquée sur un **fichier**	Appliquée sur un **commit**

git reset	Modifie la **copie locale** de l'**index** avec celle d'un commit précis	**Supprime** les commits depuis le commit spécifié
git checkout	Modifie le fichier de votre **working directory** avec celui d'un commit précis	Positionne le **HEAD** sur un commit précis

4. Gérer les branches

4.1. Créer des branches et changer de branche

Pour **lister** les **branches**, tapez la commande **git branch** :

```
git branch
```

Pour **créer** une **branche**, on utilise la commande **git branch** :

```
git branch <branch-name>
```

Par exemple, je crée la branche *hotfix* :

Pour se **déplacer** sur une **branche**, on utilise **git checkout** :

```
git checkout <branch-name>
```

Exemple en se déplaçant sur *hotfix* :

Pour faire les deux en même temps, créer une branche et se déplacer sur celle-ci, on peut utiliser l'option **-b** de **git checkout**.

4.2. Supprimer, merger et rebaser des branches

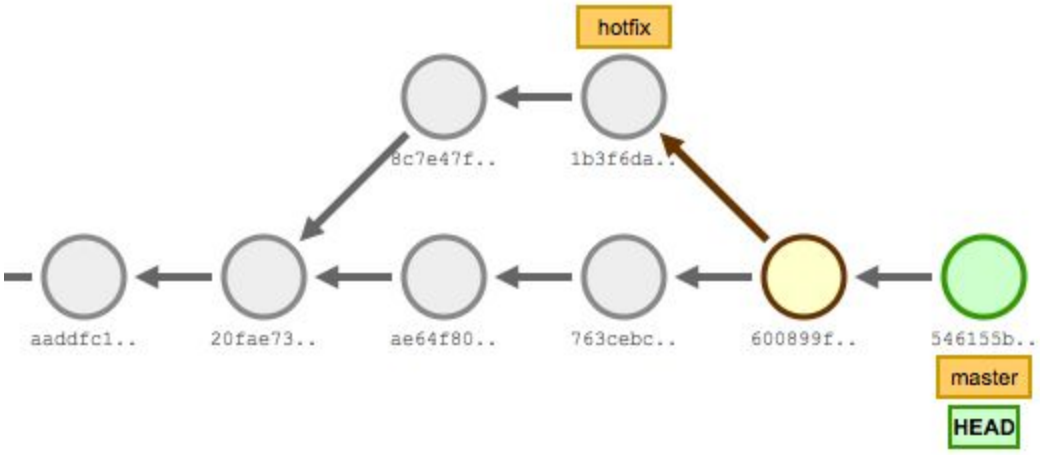
Pour **supprimer** une branche, on utilise **git branch** avec l'option **-D**, ou **-d** si la branche a été mergée (fusionnée, voir ci-dessous) :

```
git branch -D <branch-name>
```

Pour **merger** (**fusionner**) une **branche** dans la **branche actuelle**, utilisez **git merge** :

```
git merge <branch-name>
```

Exemple avec un **merge** de la branche **hotfix dans** la branche **master**, les **modifications** de la branche **hotfix** sont **fusionnées** avec celles de la branche **master**, si des conflits apparaissent, il faudra les résoudre manuellement.



Dans le cas précédent, les modifications de **hotfix** se retrouvent implémentées dans **master**.

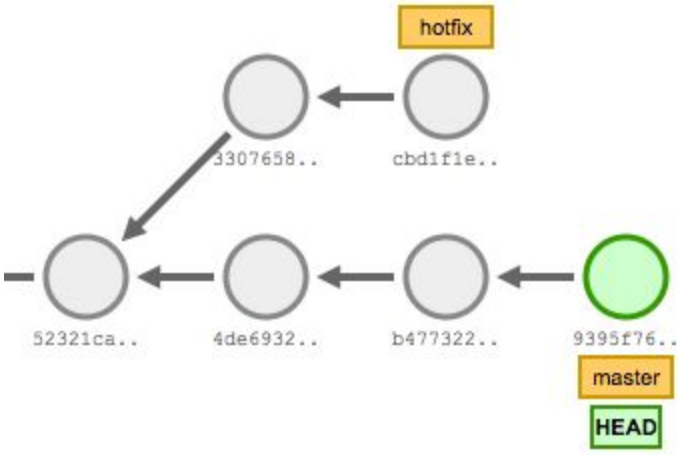
On peut aussi **rebaser** une **branche** à la base de la **branche actuelle**. Les **commits** de la **branche spécifiée** vont se retrouver à la **base** de la **branche actuelle** et vont **commencer à partir** du **commit** ou la **branche** a **été créé**.

Pour **rebaser** une **branche** sur la **branche actuelle**, on utilise **git rebase** :

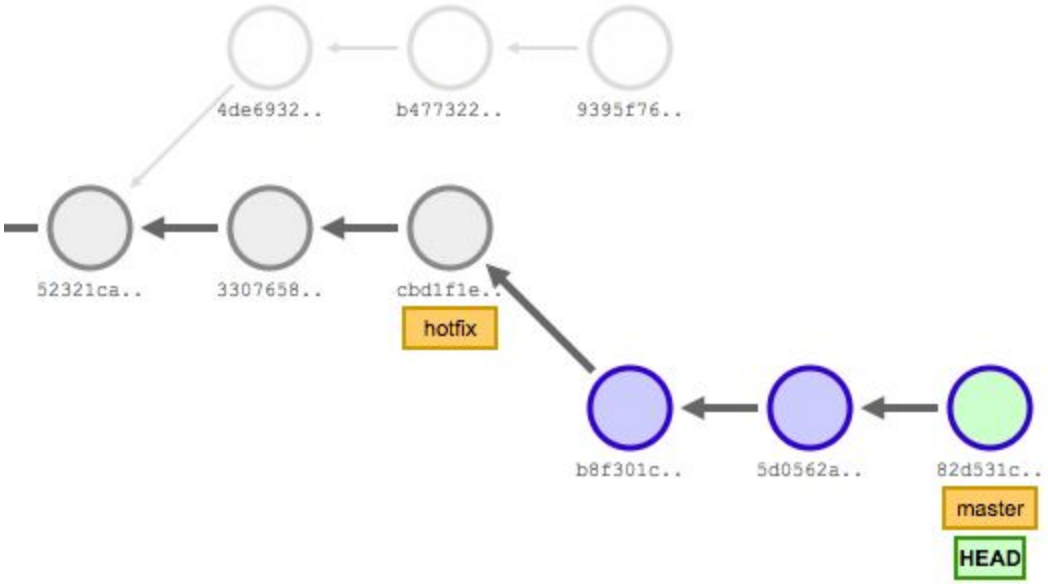
```
git rebase <branch-name>
```

Exemple avec un **rebase** de la branche **hotfix sur** la branche **master**. Les **modifications** de la branche **hotfix** sont **placées** à la **base** de la branche **master à partir du commit** ou la branche **hotfix** a **été créé**, avec la clé SHA-1 *52321ca*.

Avant le **rebase** :



On remarquera que le **rebase** commence au **commit** qui porte la clé SHA-1 *52321ca* de l'exemple précédent :



Les deux branches **ne sont pas fusionnées**, la branche **hotfix est maintenant sur** la branche **master**, à la **base** du **commit** de **création** de la branche **hotfix**.

5. Les remotes

5.1. Ajouter un remote et gérer les remotes

Pour **ajouter** un **remote** sur notre repository, on utilisera la commande **git remote add** :

```
git remote add <remote-name> <remote-url>
```

À noter que l'on donne souvent le nom **origin** au serveur remote par défaut.

Pour **afficher** les **infos** sur un **remote**, on utilise **git remote show** :

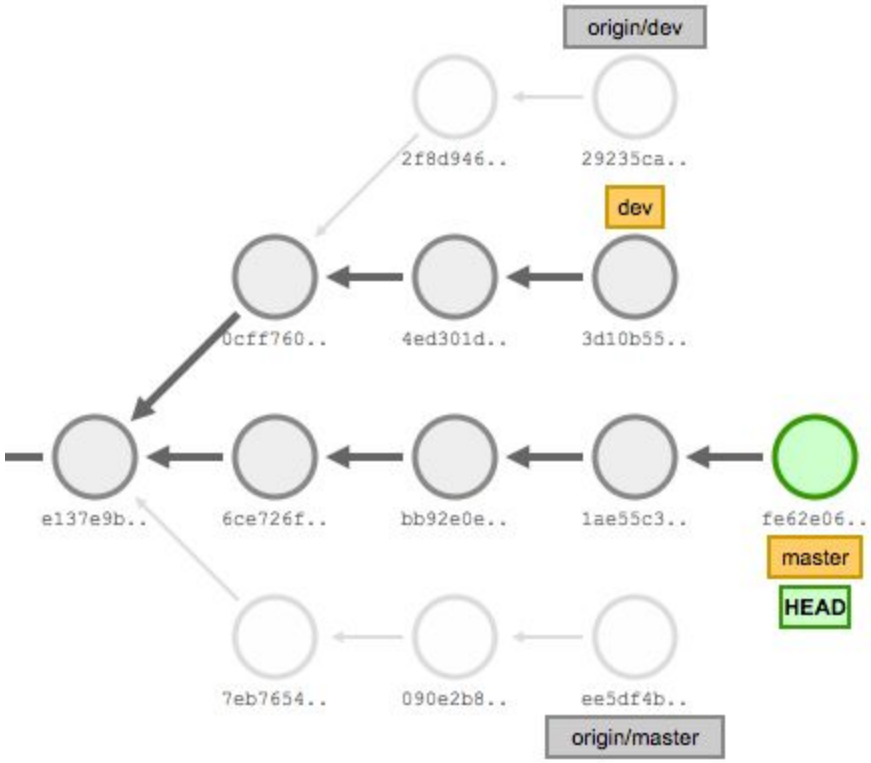
```
git remote show <remote-name>
```

Si l'on ne **renseigne pas** de remote, cela **affichera** la **liste** des remotes.

5.2. Fetch, pull et push

Pour **récupérer** une **branche** sur le serveur remote, on utilise **git fetch** (si l'on ne **spécifie pas** de **remote** ni de **branche**, **toutes** les **branches** de **tous** les **remotes** seront **fetch**) :

```
git fetch <remote-name> <branch>
```



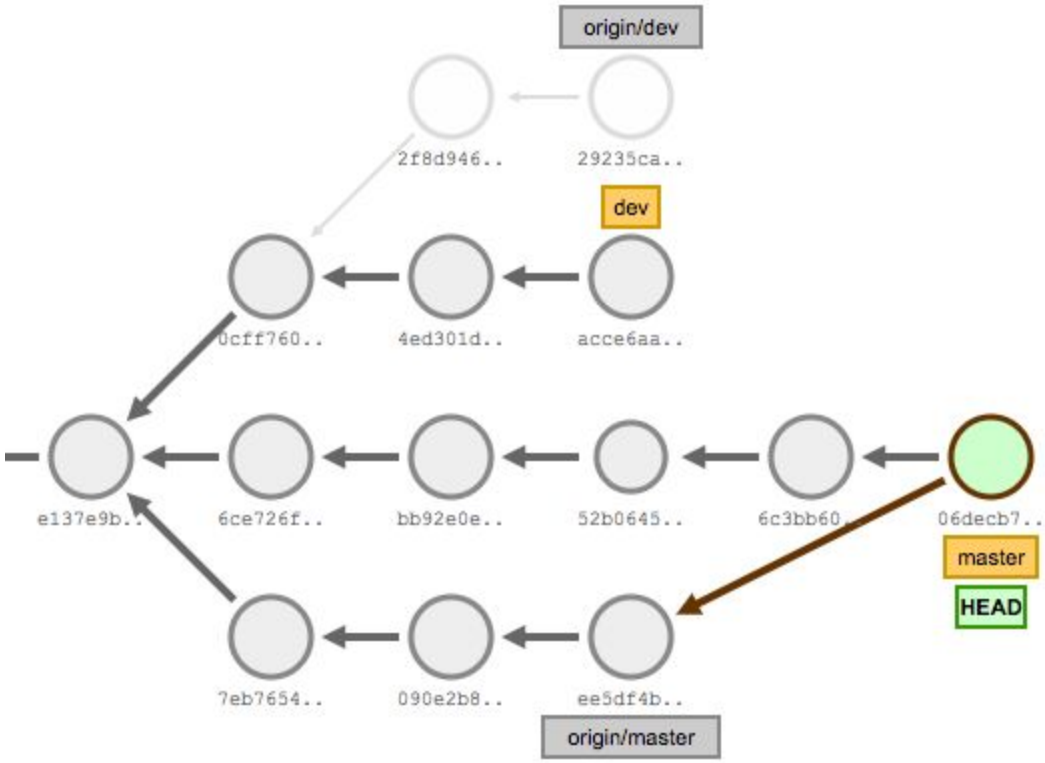
La **branche distante** récupérée sera ajoutée avec le nom **remote/branch** en **partant** du **commit** ou vous êtes **positionné** sur la **branche locale courante**.

Sur l'exemple ci-dessus, on voit que les **branches récupérées** ont le nom **origin/dev** et **origin/master**.

Pour **récupérer** une **branche et** automatiquement **merger** (**fusionner**) celle-ci sur le **repository local**, on utilise **git pull** :

```
git pull <remote-name> <branch>
```

Les branches récupérées seront d'abord **fetch**, **puis la branche distante correspondante** sera **merge** avec la **branche locale courante**. Par exemple **origin/master** sera **merge** avec la **branche locale master** si on se **situe dans celle-ci** lorsqu'on **exécute** la commande.



On **envoie** les **modifications** au serveur **remote** avec la commande **git push** :

```
git push <remote-name> <branch>
```

Par défaut, on ne peux **push** qu'en **fast-forward**, c'est-à-dire que les **push** ne doivent **jamais** faire **perdre** des **commits** sur le **remote**, pour **forcer** le **push**, on peut utiliser l'option **--force**. Il faut aussi bien noter que lors du **premier pull** ou **push**, l'**upstream** n'est **pas défini**, c'est-à-dire que par défaut, on **ne pourra pas pull** la **branche distante équivalente** à celle ou l'on **se situe**, **ou push** sur la **branche distante équivalente** non plus.

Cela veut dire qu'il **faudra renseigner** le **remote** et la **branche** sur laquelle **push** ou **pull**.

Pour pallier à ça, pour pouvoir **automatiquement pull** ou **push** sur la **branche distante master** lorsque que l'on se situe sur la **branche locale master**, on écrira lors du premier push (ou pull) :

```
git push -u origin master
```

On admet ci-dessus que le **remote** porte le nom **origin**.

Dorénavant, si l'on **est situé sur** la **branche locale master**, écrire :

```
git push
```

Enverra **automatiquement** les **modifications** sur la **branche distante master** du remote **origin**.

5.3. Supprimer un remote, afficher des branches distantes

Pour **supprimer** un **remote**, on utilise **git remote rm** :

```
git remote rm <remote-name>
```

À noter que pour **afficher** les **branches distantes** qui ont été **récupérées**, on utilise **git branch -r** :

```
git branch -r
```

On peut se placer avec **git checkout** sur une **branche distante récupérée** mais on ne **pourra pas committer**, en effet, on sera en mode **detached HEAD** et il faudra **merge** ou **rebase** avec l'une de nos **branches** avant.

On ne peut que **committer** sur des **branches** appartenant au **repository local**.

6. Pour aller plus loin

6.1. Les **tags**

Les **tags** sont assez intéressants et permettent d'**associer un numéro de version** à un **commit** particulier par exemple.

Pour **lister** les **tags** :

```
git tag
```

Pour **ajouter** un **tag** sur un **commit** particulier :

```
git tag <tag-name> <commit-id>
```

Si vous **omettez** l'**identifiant de commit**, le tag sera **placé** sur le commit représenté par **HEAD**.

Pour **supprimer** un **tag** :

```
git tag -d <tag-name>
```

Pour pouvoir **push** les **tags** sur le remote, il faudra utiliser l'option **--tags** :

```
git push --tags
```

6.2. Les **stashs**

Les **stashs** sont très utiles et permettent de **temporairement sauvegarder** une **modification**. Admettons que vous vous **situez** sur la branche **hotfix**, que vous procédez à des modifications et que vous devez **revenir** sur la branche **master**.

Sans les **stashs**, vous serez forcés de commit des modifications non terminées, les **stashs** vont donc vous permettre de **stocker temporairement** les **modifications** afin de les **reprendre plus tard**.

Créer un **stash** :

```
git stash
```

Revenir sur la **branche** et **reprendre** ses **modifications** :

```
git stash apply
```

6.3. **Supprimer** les **fichiers inutiles**

Utilisez la commande **git clean -f** pour **supprimer** tous les fichiers **untracked** :

```
git clean -f
```

6.3. Le fichier **.gitignore**

Le fichier **.gitignore** vous permet de spécifier a Git une **liste de fichiers** qu'il doit **ignorer** et ne **pas tracker**. Vous pouvez créer votre .**gitignore** suivant ces **règles** :

- Une **ligne vide** ne matche aucun fichier.
- Les **commentaires** commencent par " `#` ".
- Les **espaces** sont **ignorés sauf** s'ils sont **échappés** avec un **backslash** (" `\` ").
- Il est possible d'**annuler** une **règle** avec le **point d'exclamation** (" `!` ") place devant. Toute règle qui exclut ces fichiers se retrouve donc annulée et les fichiers sont de nouveau inclus.
- Pour matcher un **dossier**, il faut **ajouter** un **slash** (" `/` ") à la **fin** du **nom** de dossier.
- Il est possible d'utiliser les **wildcards** (" `*` ") pour signifier n'importe quel caractère, une ou plusieurs fois.
- Deux astérisques (" `**` ") matchent n'importe quel chemin d'accès, " `**/foo` " matchera le fichier " `foo` " dans n'importe quel sous-dossier, ou sous-sous-dossier.
- Il faut **échapper** les **caractères spéciaux** avec l'antislash (" `\` ").

Le fichier **.gitignore** doit être **tracké** et **committé**.

Plus d'infos ici : https://git-scm.com/docs/gitignore

7. Fin du tutoriel

Merci pour votre lecture attentive, ce tutoriel m'a demandé pas mal de temps, notamment pour la rédaction et pour essayer de condenser un maximum d'infos en moins de 20 pages.

Ce document ne remplace en aucun cas la documentation officielle, et les commandes sont susceptibles d'évoluer en fonction des versions de Git.

Si ce tutoriel vous a plu, vous pouvez me payer un p'tit café pour me donner envie d'en refaire d'autres comme celui-ci :)

À plus,

Edwy Mandret, alias *emandret*.