Le travail à distance

Jusqu'à présent nous avons vu comment travailler avec `git` en local (sur notre machine) pour ajouter des points de sauvegarde (`commits`) et ainsi garder une trace de chaque changement apporté au fil du temps à notre code.

De cette manière, en cas de besoin et/ou si on a fait une grosse boulette dans notre code et que plus rien ne fonctionne , il est totalement possible de revenir en arrière dans un état précédent. Comme cela nous pourrons retrouver du code plus stable et donc en état de marche.

Maintenant, comme je le disais plus haut, ces modifications et cet historique de commit est toujours localisé sur notre machine.

Maintenant imaginez ce scénario :

Après une longue journée ou quelques semaines de dur labeur, vous avez bien avancé sur votre projet. Consciencieusement, vous avez fait vos commits qui étaient magnifiquement intitulés. Chacun de ces commits possédait une description très descriptive de chaque modification apportée.

Vous quitter donc votre ordinateur avec le sentiment du travail bien accompli. Un réel sentiment de plénitude.

Malheur!

Durant la nuit, votre ordinateur à été anéhanti par une horde de zombie atomique de l'espace. Vous le retrouvez donc fumant le lendemain et ceci sans mot d'excuse... Evidement.

En plus de râler sévèrement de ne pas avoir souscrit à une assurance contre ce type de dommage matériel vous vous rendez compte avec effroi que TOUT (mais alors tout) votre travail à lui aussi été détruit.

Tant d'heures de travail parties dans ces volutes de fumée...

Bon, perdre votre ordinateur c'est déja assez furstrant mais perdre votre travail ça l'est encore plus.

D'autant plus si vous travailliez sur une application qui aurait peut-être pû éradiquer la faim du monde ou juste envoyer des images de perroquets portant des chapeaux mexicains à vos amis. (Chacun son ambition hien 😃).

Bref, grâce à `git` et ses fonctionalités liées au travail à distance, cette situation aurait pu être évitée.

Mais comment ce pourrait-ce ?

Grâce à `git` nous pouvons communiquer directement avec des machines situées à distance.

Ces machines que l'on nomme dans le jargon des `remote` (distant, en anglais), peuvent être de deux types :

1. Interne : un autre ordinateur que vous possédez par exemple.
2. Externe : un service tiers qui hébergera sur internet votre code. `GitHub` , `GitLab` , `BitBucket` , ... sont autant de site internet permettant l'hébergement de code et étant accessible via `git` .

Ici notre scénario catastrophe aurait pû être éviter si nous avions utiliser ces services tiers pour y pousser (`push`) régulièrement notre code.

Ansi, en plus d'avoir une version sur notre ordinateur, nous aurions eu une version identique de notre projet qui co-existerait sur internet.

Déjà que la destruction totale de notre machine par une horde de zombie est peu probable, la destruction totale simultanée de notre ordinateur ET d'un serveur sécurisé l'est encore moins.

Donc toute probalitité gardée, en poussant notre code sur ce type de service, notre travail pourrait toujours été acccessible. Bon peut-être en cas de guerre nucléaire mais ça c'est une autre histoire.

Un autre avantage non négligeable du travail avec ces `remote` est que cela favorise et facilite énormément le travail collaboratif.

En effet, une fois ayant reçu l'autorisation de se connecter via `git` (via HTTPS ou autre) à un `remote` , chaque collaborateur d'un projet pourra récupérer un projet et ainsi aider à sa construction.

Récupérer un projet hébergé sur un `remote`

Pour commencer, nous allons devoir un peu changer nos habitudes : pour cette leçon nous n'allons pas créer notre projet en ajoutant des fichiers à la main et en initialisant un repo `git` (via la commande `git init`) mais nous allons récuperer notre projet depuis un remote.

Pour ce faire nous allons devoir nous placer via le terminal à l'endroit où l'on souhaite ajouter le projet.

Pour des raisons de simplicité, n'hésitez pas à aller dans le dossier où se trouve les autres projets CSS par exemple.

Une fois que vous êtes au bon endroit taper cette commande dans le terminal :

```
git clone http://codephenix.com:2500/root/css-flexbox.git
```

Décortiquons cette commande :

Ici nous avons demander à `git` de cloner le repo distant (depuis le remote) sur notre disque dur en local.

Essayons cette commande.

Soyez patient le clonage peut prendre quelques minutes avant de se terminer. Tout dépend la taille des fichiers présent dans le repo sur le `remote`.

Dès que c'est terminé vous pouvez ouvrir le dossier avec votre éditeur de texte. Ce projet devrait vous être familier.

En effet c'est celui que l'on a construit ensemble durant le cours sur `FlexBox`.

Super! Tout est déja prêt pour nos experimentations.

Master, la branche sacrée

Pour ce projet, nous pourrions uniquement travailler sur la branche `master`.

Cependant en cas de travail en équipe, il est fortement déconseillé de travailler directement sur cette branche.

Mais pourquoi ?

Sur beaucoup de projet et pour beaucoup d'équipe de développement, la branche `master` possède une importance particulière.

Cette branche doit toujours conserver la version du code la plus `stable` c'est-à-dire la version du code dont on est certains qu'elle contient le moins de bug possible. Il faut aussi garder en tête que cette branche à vocation à être mise en `production`.

Point vocabulaire :

Mettre en `production` un site web ou une application signifie la mettre `en ligne` sur internet et donc disponnible pour tout le monde.

Vous comprenez aisément qu'il est important de vouloir présenter la meilleure version de notre travail aux utilisateurs finaux. Nous souhaitons aussi éviter que ces utilisateurs ne se plaigne de dysfonctionements ou encore du fait que notre application web ne fonctionne plus du tout!

Souvenez vous la furstration que vous avez déjà peut-être ressenti face à quelque chose qui ne fonctionne pas comme il le devrait.

Un de nos objectifs principaux en tant que développeur et d'éviter que nos utilisateurs ressente ce type de furstration.

```
VOUS (avec un regard confu) :

— D'accord mais si je ne peux pas travailler sur master, cela veut donc dire q
```

Ben oui effectivement, on peut remballer. C'est chômage technique ...

Mais non malheureux! Ici nous allons privillégier une autre solution. Nous allons créer une autre branche qui hébergera nos experimentations.

Plaçons nous via le terminal sur le dossier fraîchement cloné et créons une branche que l'on va nommer `fix/ajout-readme`.

Normalement, vous devriez avoir toutes les clés en mains pour retouver la commande qui permet de créer une branche.

Vous ne vous rappelez plus ?

Allez la voici (voyez comme je suis sympa)

```
git  checkout-b `fix/ajout-readme`
```

Et nous voilà sur notre branche `fix/ajout-readme`

Modifions notre façon de travailler.

Depuis le début du cours, nous avons appris à commiter notre code en utilisant la commande `git commit` ou sa version raccourcie `git commit -m <mon message de commit>`.

Pour ceci rien ne change. Cependant une fois que nous avons bien avancé sur notre projet, il va être temps de le mettre en sécuritée et de le rendre disponible pour nos cammarades

de travail.

Pour ce faire nous pouvons utiliser la commande `git push`

Décortiquons cette commande :

Ici nous demandons à `git` de pousser notre code sur un `remote`.

Petite astuce, en réalité la commande éxectuée par `git` va être un peu plus complexe. En tapant cette commande `git push`, `git` va l'interpréter comme étant `git push origin/<la branche où nous nous trouvons>`.

Ici `git` devrait essayer de `push` notre code sur un serveur distant que git connaîtra sous le nom de `origin`. Plus spécifiquement `git` essayera de stocker notre code sur la branche `fix/ajout-readme` située sur notre remote.

Il est tout à fait possible de choisir de pousser notre code sur un autre `remote` que `origin` mais ceci est une thématique encore un poil trop avancée pour nos connaissances actuelles. Chaque chose en son temps.

Essayons donc de pousser notre code sur le remote via la commande `git push`

Mmmmh un étrange message sauvage est apparu dans notre terminal :

```
$ git push
fatal: The current branch feature/ajout-readme has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature/ajout-readme
```

Si vous avez le message en français c'est super mais pour ceux qui ont `git` configuré en anglais, nous allons essayer de décortiquer le message ensemble :

Ici `git` nous dit qu'il ne connait pas de branche nommée `feature/ajout-readme` sur le repo distant `origin`.

Effectivement c'est vrai nous n'avons pas encore signifié au remote que nous souhaitons aussi créer une copie distante de notre branche.

Actuellement, uniquement et par défaut `master` possède une copie distante.

Comme `git` est un outil fort sympatique, il nous indique aussi la commande à utiliser pour lui expliquer comment créer une copie distante de notre branche `feature/ajout-readme`

Et cette commande est celle-ci : `git push --set-upstream origin feature/ajout-readme`

Tapons-la dans notre terminal.

Si tout c'est bien passé `git` nous confirme que notre branche distante à bien été créée.

Donc nous pouvons réessayer de `push` notre code sur le remote.

Je vous laisse le faire.

Ici comme nous n'avons fait aucune modification, `git` nous dit que tout est à jour des deux coté.

Cela confirme que les 2 branches (locales et remote) sont bien synchronisées.

Lisez moi, lisez moi , lisez moiiiiiiii !

Si tout va bien pour vous, je vous propose d'ajouter un fichier nommer `README.txt` à la racine du projet (`READ ME` , signifie `lisez moi` en anglais).

Ce fichier à pour extension `.txt` cela signifie juste que ce fichier est un simple fichier `texte` .

Dans ce fichier nous allons rajouter ceci :

```
e README.
très important car il a pour but d'aider les autres développeurs à collaborer su

rmations qui les aiderons à travailler sur notre projet comme un e-mail de conta
```

Une fois ce fichier `README.txt` ajouter, sauvegardons et commitons notre travail. Une fois ceci fait, mettons notre travail à l'abri sur le remote via la commande `git push` .

Aainsi grâce à `push` et au remote, notre manière de travailler (`workflow` , en anglais) peut devenir le suivant :

- je clone un repo via `git clone`
- je crée une branche de travail avec `git checkout -b <nom de ma branche>`
- je modifie du code
- je commit `git commit -m <mon message>`
- je modifie du code
- je commit `git commit -m <mon message>`
- ...
- ...
- je suis satistafait et je crée une copie distante de ma branche avec `git push --set-upstream origin feature/ajout-readme`
- je push mon code via la commande `git push`
- je modifie du code

- je commit `git commit -m <mon message>`
- je modifie du code
- je commit `git commit -m <mon message>`
- ...
- ...
- je suis satistafait et je push mon code via la commande `git push`

et ainsi de suite jusqu'à la fin du projet.

Toujours se maintenir à jour.

Retournons sur `master`. Vous le constatez, le fichier `ReadMe.txt` n'est pas sur cette branche.

Nous allons supprimer notre branche `feature/ajout-readme` en local via la commande

```
git branch -d feature/ajout-readme
```

Ok maitenant j'aimerais que vous fassiez en sorte d'avoir le fichier `ReadMe.txt` disponible sur la branche master.

Ici vous pouvez utiliser un `rebase` ou un `merge`, cela n'a que peu d'importance. Personellement je préfère le `rebase` car il garde l'historique des `commits` et ne rajoute pas de commit supplémentaire.

Bon soyons fou faisons donc un

```
git rebase feature/ajout-readme
```

Et bardaf voici la `fatal Error` ..

Et bien oui, je vous ait piégé.

Etant donné que la branche locale `feature/ajout-readme` n'existe plus `git` ne peu plus l'utiliser pour mettre à jour `master`.

Voilà fin du cours. De nouveau chômage technique.

Ah ben non vous ne vous en tirerez pas aussi facilement.

N'oubliez pas que nous avions aussi créé une branche distante `feature/ajout-readme` qui était la copie conforme de notre branche locale et qui donc, elle, contient encore notre fichier `ReadMe.txt`.

Nous pouvons donc peut-être utiliser cette branche distante pour mettre à jour master.

Ici deux solutions :

1. La plus longue :

Recréer une branche locale `feature/ajout-readme` depuis la branche en remote et mettre à jour master depuis cette branche locale.

2. La plus courte :

Directement mettre à jour `master` depuis la branche distante sans recréer une branche en local.

Pour la solution 1 voici les étapes que nous pourrions faire :

- D'abord nous avons besoin d'aller chercher (`fetch` , en anglais) toutes les branches distantes existantes sur `origin` :

```
git fetch origin
```

- Ensuite nous allons recréer notre branche locale `feature/ajout-readme` .

```
git checkout feature/ajout-readme
```

- nous devons nous replacer sur master

```
git checkout master
```

- Et enfin faire notre `rebase`

```
git rebase `feature/ajout-readme`
```

Un peu long non ?

Pour la solution numéro 2 c'est beaucoup plus court

```
git pull --rebase origin feature/ajout-readme
```

Allez y essayer la solution 2. Elle est auto-magique !

Gérer les conflits

Bon jusqu'ici nous travaillions avec nous même dans un monde idéal, calme et paisible. Mais imaginez cette situation :

Vous êtiez tranquillement en train d'enrichir sur votre branche locale votre fichier `README.txt` avec les instructions les plus précises que vous puissiez founrir à de futurs collaborateurs.

Jack, le méchant de notre histoire (c'est un pseudonyme - pour préserver son identitée ;=)) lui aussi a eu cette idée et modifie le même fichier. Mais Jack à été plus rapide que vous : Il a déjà fait ses modifications et les a déja mises sur master.

Vous, évidemment, n'êtes absolument pas au courrant. Vous continuer donc à travailler tranquillement. Le temps passe et arrive le moment où vous aussi souhaitez mettre votre travail sur master.

A votre avis comment `git` va agir ?

Va-t-il supprimer les modifications de Jack pour les remplacer par les vôtres ? Ou le contraire, va-t-il ignorer vos modifications et garder celles de Jack ?

Voyons par nous même. Essayons de reproduire cette situation :

Etape 1: Reproduire les actions de Jack

- Placçons nous sur `master` et modifions le fichier `README.txt` en remplaçant le contenu par ceci :

  ```
  Voici les premières lignes de MON sublime README. Bien meilleur que
  Le document README est un fichier très important car il a pour but d

  Ici je peux écrire toutes les informations qui les aiderons à travai
  ```

- Commitons ceci et faisons un `git push`

Etape 1: Modifions le README sur la branche locale `feature/ajout-readme`

- Si vous ne l'avez pas encore recrée faite, depuis la branche `master`, un `git checkout feature/ajout-readme`
- Une fois sur la branche remplaçons le contenu `README.txt` par ceci :

  ```
  Ce Readme est en cours de construction.
  Cependant, ce projet est un projet commun créé en collaboration. Nou

  Le document README est un fichier très important car il a pour but d
  ```

```
Ici je peux écrire toutes les informations qui les aiderons à travai
```

- Sauvegardons, commitons et faisons notre `git push`

Une fois ceci fait, retournons sur `master` et essayons de mettre cettre branche à jour avec notre branche `feature/ajout-readme` via un `git rebase feature/ajout-readme`

Normalement vous devriez avoir ce type de message d'erreur :

```
$ git rebase feature/ajout-readme
First, rewinding head to replay your work on top of it...
Applying: update readme
Using index info to reconstruct a base tree...
M       readMe.txt
Falling back to patching base and 3-way merge...
Auto-merging readMe.txt
CONFLICT (content): Merge conflict in readMe.txt
error: Failed to merge in the changes.
hint: Use 'git am --show-current-patch' to see the failed patch
Patch failed at 0001 update readme
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abor
```

Encore une fois, si votre `git` est configuré en français, cela devrait être un peu plus compréhensible.

Dans tout les cas, voyons en détail ce que dit ce message :

Ici `git` nous informe qu'il n'arrive pas à mettre à jour master depuis la branche `feature/ajout-readme` car des deux cotés (sur la branche master et sur la branche feature) des modifications ont été faites.

Actuellement `git` est dans la panade et ne sait pas réellement quelle modification prendre en compte. Il nous demande donc de l'aide pour lui indiquer quelle modification il doit prendre.

Il y a donc un `conflit` et `git` ne sait pas quel parti prendre.

Resolvons les conflits

Mais nous nous savons comment faire pour résoudre ce conflit.

Pour ce faire grâce à notre éditeur de texte, ouvrons le fichier `ReadMe.txt`

Vous devriez le trouver dans un état proche de ceci :

```
<<<<<< HEAD
  Ce Readme est en cours de construction.
  Cependant, ce projet est un projet commun créé en collaboration. Nous avons

  Le document README est un fichier très important car il a pour but d'aider l
=======
Voici les premières lignes de MON sublime README. Bien meilleur que celui écri
Le document README est un fichier très important car il a pour but d'aider les
>>>>>>> update readme

  Ici je peux écrire toutes les informations qui les aiderons à travailler sur
```

Pour résoudre ce conflit nous allons devoir choisir manuellement ce que nous souhaitons faire : garder nos modifications, garder celles de Jack ou garder les deux.

Pour ce faire remarquer que la partie du texte conflictuelle est délimitée par ceci :

```
<<<<<< HEAD
>>>>>>> update readme
```

et que vos modifications sont séparée de celle de Jack par ceci `=======`.

Avec ces indicateurs nous pouvons mieux cerner ce que nous devons faire.

Ici, nous somme beau joueur et nous allons garder aussi bien les modifications de Jack que les nôtres.

Nous devons donc simplement supprimer tous les signes superflus de notre fichier C'est-à-dire ceux-ci:

```
<<<<<< HEAD
>>>>>>> update readme
```

ainsi que les `=======`

Ainsi notre fichier `ReadMe.txt` ressemblera à ceci :

```
Ce Readme est en cours de construction.
Cependant, ce projet est un projet commun créé en collaboration. Nous avons be
Le document README est un fichier très important car il a pour but d'aider les
Voici les premières lignes de MON sublime README. Bien meilleur que celui écri
Le document README est un fichier très important car il a pour but d'aider les
 Ici je peux écrire toutes les informations qui les aiderons à travailler sur
```

Bon cette version du Readme est pas totalement compréhensible mais c'était pour l'exercice. 😃

Une fois tous les signes enlevés nous pouvons sauvegarder notre fichier et signifier à `git` que nous avons avons résolu les conflits.

Pour ce faire nous devons faire un `git add .` (le `.` est un raccourci pour dire à `git` d'ajouter toutes les modifications) pour ajouter les modificatons que nous avons faites et nous pouvons continuer le rebase en faisant un `git rebase --continue`.

Je vous laisse essayer.

Et voilà `git` a fini de remettre à jour master depuis notre branche `feature/ajout-readme`.

Maintenant nous devons mettre le branche distante `origin/master` à jour avec nos modifications.

Pour ceci nous pouvons utiliser la commande

```
git push -f
```

Nous utilisons le flag `-f` pour `f`orcer git à envoyer nos modifications sur le serveur. Nous avons besoin de ce flag car n'oublions pas que le rebase `réecrit` l'historique des commits.

Notre remote lui n'est pas au courrant de ces modifications d'historique et risque donc de rejeter notre `push`.

C'est pourquoi nous utilisons le flag `-f` pour éviter ce rejet.

Gérer un conflits plus grave

Ici le conflit était assez simple à résoudre. Parfois, lors d'un rebase, les conflits peuvent être très complexe à résoudre car la différence d'historique des commits entre les deux branches est très grande.

Si vous sentez que vous n'allez pas pouvoir effectuer correctement ce rebase vous pouvez le quitter en faisait un `git rebase --abort`.

Si vous êtes dans cette situation, peut-être qu'effectuer un `git merge` à la place d'un `git rebase` sera plus judicieux.

En effet, avec le `merge` git ne tentera pas de comparer les deux historiques. Il se contentera de créer un commit avec les modifs de chaque branches. Vous aurez toujours les conflits mais ceux-ci devraient être plus simple à résoudre.

Voici qui termine ce cours sur GIT.

Pour résumer :

Git est un outil essentiel de travail que, en tant que développeur, nous sommes ammener à utiliser au quotidien. Cet outil permet de créer des points de sauvegardes (`commits`) de notre travail et de le mettre en sécurité sur des serveurs distants (`remote`). Ainsi cet outil, permet de travailler de manière collaborative sur des projets de plus grande échelle.

Codez bien !