Prise en main de git en ligne de commande

Dans ce TP, pour prendre en main git,
vous allez refaire pas à pas les exemples du cours d'Introduction à Git disponible
https://github.com/iblasquez/enseignement-but1-developpement
N'hésitez pas à naviguer entre le TP et le cours 😉

1. Installer git
2. Configurer git

3. Créer un nouveau dépôt par initialisation
4. Effectuer un premier commit
5. Continuer à committer
6. Créer une branche
7. Se déplacer de branche en branche
8. Fusionner deux branches (fast-forward)
9. Travailler sur deux branches simultanément puis les fusionner
10. Tagger un commit
11. Gérer un conflit
12. Visualiser un workflow à l'aide d'un éditeur graphique
13. Supprimer une branche

14. Cloner un dépôt existant distant
15. Travailler en local sur un dépôt cloné
16. Synchroniser les dépôts

Pour chaque commande présentée, vous pourrez également, si vous le souhaitez, sélectionner en même temps
cette commande dans : **https://gitexplorer.com/**,
outil très pratique pour retrouver rapidement la commande git de vos rêves😉

1 . Installer git

Pour installer git sous votre environnement de travail, rendez-vous sur : https://git-scm.com/download

Pour vérifier que git est bien installé, ouvrir un terminal (console) et par taper : `git --help`

2. Configurer git

✓ **Sous Git, il n'y a pas de commit anonyme**, dans la console vous devez commencer par :

➔ Configurer votre nom :
git config --global user.name "Prénom Nom"

➔ Configurer votre email :
git config --global user.email email@domaine.extension

➔ Pour vérifier votre configuration, vous pouvez taper :
git config --global user.name
git config --global user.email

Pour en savoir plus :
https://help.github.com/articles/setting-your-username-in-git/
https://help.github.com/articles/setting-your-commit-email-address-in-git/
https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration
https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Param%C3%A9trage-%C3%A0-la-premi%C3%A8re-utilisation-de-Git

✓ **Pour aider à la lisibilité des messages**, vous pouvez aussi activer la couleur.

git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto

✓ Le fichier contenant toutes ces configurations est le fichier : **.gitconfig**
Recherchez ce fichier dans votre répertoire personnel et ouvrez-le.
Ce fichier contient pour l'instant les rubriques suivantes : **[user] [color]** et peut-être **[filter "lfs"]**

Ajoutez la rubrique **[alias]** configurée de la manière suivante :

```
[alias]
    ci = commit
    co = checkout
    st = status
    br = branch
```

Ces alias vous permettront de raccourcir certaines commandes de git très fréquemment utilisées.
Par exemple, au lieu d'écrire **git commit**, vous pourrez écrire **git ci**. Vous pouvez bien sûr personnaliser ces alias à votre convenance et en créer de nouveaux pour d'autres commandes git.

3. Créer un nouveau dépôt par initialisation

✓ Dans votre explorateur de fichiers, **commencez par créer un répertoire demo**.

✓ ***Si vous êtes sous Windows***, pour manipuler git,
mieux vaut utiliser la console **git Bash.**
Ouvrez cette application avant de continuer et **travaillez dorénavant dans cette console**.

✓ **Depuis la console**, **placez-vous dans le répertoire demo** que vous venez de créer
et **initialisez un dépôt git** sur ce répertoire grâce à la commande : **git init**

→ Normalement la mention **(master)** s'affiche dans votre prompt à côté du chemin pour vous indiquer que ce répertoire est désormais sous contrôle du gestionnaire de version et que vous êtes actuellement sur la branche principale **master**.

→ Consultez le contenu du répertoire :
- Soit depuis l'explorateur de fichiers
- Soit directement depuis la console en utilisant la commande **ls -a** pour pouvoir visualiser les fichiers cachés.
  *Remarque : Dans le **git Bash,** vous devez utiliser les commandes Unix :*
  - *Lien vers les principales commandes Unix :*
    https://fr.wikipedia.org/wiki/Commandes_Unix

  *Attention, si vous êtes encore dans une simple console sous Windows, vous ne verrez pas le label **(master)**, passez sous **git Bash** !!!*

✓ Vérifiez qu'un répertoire **.git** a bien été créé !

4. Effectuer un premier commit

✓ Dans le répertoire **demo**, ajoutez un fichier **test.txt** dans lequel vous écrirez par exemple :
Bonjour tout le monde !

✓ Pour que **git** puisse versionner ce fichier, vous devez (depuis la console **git Bash**) :

→ 4.1 dans un premier temps, commencer par **ajouter ce fichier dans l'Index (add)** via le commande : **git add test.txt**

Remarque : Pour vérifier que ce fichier a bien été ajouté dans l'Index, vous pouvez effectuer la commande : **git status**
Cette commande permet d'afficher l'état courant du dépôt. Le fichier **test.txt** devrait donc apparaître en vert comme un fichier prêt à être commité (*to be committed*)

→ 4.2 dans un second temps, **enregistrer réellement les modifications** contenues dans l'index via la commande **commit** suivi d'un **message explicite** indiquant l'objet du commit : **git commit -m "premier commit"**

Remarque : Pour vérifier que ce fichier a bien été commité, exécutez à nouveau : **git status** qui indique cette fois-ci qu'il n'y a plus rien à commiter (*nothing to commit*)

✓ Pour obtenir un **historique des commits**, tapez : **git log**
La liste des commits, réduit à un seul pour le moment, devrait s'afficher.

✓ Pour obtenir le **détail du dernier commit**, tapez : **git show**
Le détail du commit précédent devrait s'afficher.

5. Continuer à committer

✓ Ajoutez dans votre fichier **test.txt** la ligne suivante : **J'utilise git !**
N'oubliez pas d'enregistrer ce fichier avant de continuer 😉

✓ Vous pouvez comparer la dernière version committée et la version en cours de votre fichier.
Pour visualiser la **diff**érence entre ces deux versions, tapez : **git diff**

✓ Pour que **git** puisse versionner ce fichier c-a-d ajouter cette modification à l'historique des commits, il faut réaliser les deux étapes présentées précédemment, à savoir l'ajout du fichier dans l'index (**add**) avant de pouvoir réellement **commit**er.
Ces deux étapes peuvent être regroupées en une seule ligne de commande en utilisant l'option **-am,** l'option **-a** permettant d'ajouter dans l'index tous les fichiers *trackés* (c-a-d les fichiers qui ont déjà été ajoutés une fois dans l'index).
Pour enregistrer ce nouveau commit en une seule instruction, tapez la commande suivante :
git commit -am "ajout du message sur git"

✓ Demandez l'**historique des commits** avec **git log** pour visualiser les deux commits effectués jusqu'à présent.

✓ Pour obtenir le **détail du dernier commit**, tapez : **git show**

6. Créer une branche

Jusqu'ici vous avez travaillé sur une seule branche **master** qui est la branche principale, celle qui en général contient le « vrai » code source de votre projet.

✓ Pour voir toutes les branches, tapez la commande **git branch**
Pour l'instant, vous n'avez que la branche **master** 😉

✓ Nous allons créer une branche **anglais** pour internationaliser le texte du fichier **test.txt** !
Pour créer une branche, il faut utiliser la commande **branch**.
Tapez : **git branch anglais**

✓ Visualisez toutes les branches avec la commande **git branch**
Le symbole étoile ***** indique la branche sur laquelle vous vous trouvez.
On est donc toujours sur **master** 😉

✓ Une fois la branche créée, il faut s'y positionner dessus si on souhaite enregistrer les commits sur cette branche.
Pour se déplacer sur une branche, il faut utiliser la commande **checkout.**
Tapez : **git checkout anglais**

✓ Que s'est-il passé dans le prompt de la console ?
La mention **anglais** vous indique sur vous êtes désormais sur la nouvelle branche **anglais**.
Tapez **git branch** et vérifiez que l'étoile * désigne bien la branche **anglais**.

✓ Dans le répertoire **demo**, créez un nouveau fichier **test_EN.txt** que vous remplirez avec le texte suivant :
Hello world !
I'm using git !

✓ Ajoutez ce nouveau fichier dans l'index via la commande : **git add test_EN.txt**
✓ Commitez ces modifications via l'instruction : **git commit -am "ajout du message en anglais"**

✓ Demandez l'**historique des commits** avec **git log**.

7. Se déplacer de branche en branche

✓ Dans votre explorateur de fichiers, consultez le contenu de votre répertoire **demo**.
(ou tapez l'instruction **ls** dans la console)
Vous devez actuellement visualiser dans ce répertoire des fichiers **test.txt** et **test_EN.txt**.

✓ Pour vous repositionner sur la branche **master**, tapez dans la console : **git checkout master**
Si la commande s'est correctement exécutée, la mention **master** apparaît désormais dans le prompt de la console.

✓ Dans votre explorateur de fichiers, consultez le contenu de votre répertoire **demo**.
(ou tapez l'instruction **ls** dans la console)
Vous devez visualiser dans ce répertoire uniquement le fichier **test.txt** puisque vous êtes actuellement sur la branche master et que le fichier **test_EN.txt** n'a été ajouté que dans la branche **anglais**.

8. Fusionner deux branches (fast-forward)

✓ Positionnez-vous sur la branche **master**

✓ Pour fusionner deux branches, il faut utiliser l'instruction : **merge**
Dans la console, tapez : **git merge anglais**

✓ Demandez l'**historique des commits** avec **git log**.
Vous venez de faire une **fusion de type *fast-forward***.
C'est la fusion la plus simple : elle ne créée pas un nouveau commit lors de la fusion, mais déplace seulement le sommet de la branche **master** puisqu'ici seules des modifications ont eu lieu sur une seule branche (la branche **anglais**) et le dernier commit de la branche **master** est resté inchangé.

✓ Dans votre explorateur de fichiers, consultez le contenu de votre répertoire **demo**.
(ou tapez l'instruction **ls** dans la console)
Vous devez actuellement visualiser dans ce répertoire les deux fichiers **test.txt** et **test_EN.txt**, ce qui confirme qu'ils ont bien été tous deux fusionnés dans la branche **master**.

9. Travailler sur deux branches simultanément puis les fusionner

✓ Tapez **git branch**
Vous êtes actuellement sur la branche **master**.
Notez au passage, que la branche **anglais** continue à exister dans votre historique…

Nous allons maintenant faire évoluer l'état de ces branches en faisant apparaître de nouveaux commits sur chacune des branches, puis nous fusionnerons le tout…

✓ Assurez-vous d'être sur la branche **master**
Ouvrez le fichier **test.txt** et ajoutez dans ce fichier la ligne suivante :
git est un logiciel de gestion de versions décentralisé.

✓ Enregistrez ce fichier, puis **committer** avec le message suivant : **"ajout définition git en français"**

✓ Demandez l'**historique des commits** avec **git log**.

✓ Positionnez-vous maintenant sur la branche **anglais**
Ouvrez le fichier **test_EN.txt** et ajoutez dans ce fichier la ligne suivante :
git is a distributed version control system.

✓ Enregistrez ce fichier, puis **committer** avec le message suivant : **"ajout définition git en anglais"**

✓ Demandez l'**historique des commits** avec **git log**.

✓ Positionnez-vous maintenant sur la branche **master**

✓ Fusionner les deux branches en tapant : **git merge anglais**
Attention, comme il y a eu des modifications (nouveaux commits) sur les deux branches à fusionner, un commit de fusion est nécessaire pour centraliser le tout.
La fusion va donc donner naissance à un nouveau commit qui doit avoir son propre message …

L'éditeur s'ouvre donc dans la console pour vous permettre d'ajouter le message de commit.

A la place de **Merge branch 'anglais'**, écrire le nouveau message de commit :
Fusion définition git en anglais
Attention, vous êtes dans un éditeur de type **vim**.
Pour entrer dans le mode insertion appuyer sur **i**
Pour enregistrer vos modifications vous devez faire **Escape** puis **:wq** puis **Entrée**.

Remarque : Un lien vers la liste des commandes **vim** : https://doc.ubuntu-fr.org/vim
D'autres guides de survie sous vi :
https://www.linuxtricks.fr/wiki/guide-de-sur-vi-utilisation-de-vi
https://linux.goffinet.org/administration/traitement-du-texte/editeur-de-texte-vi/
A retenir plus particulièrement : **:w** => enregistrer le fichier
:q => quitter vi
:wq => enregistrer le fichier et quitter vi
:x => quitte en enregistrant le fichier
:**q !** => quitter vi en annulant les changements

✓ Demandez l'**historique des commits** avec **git log**.
La fusion a donc créé un nouveau commit faisant apparaître le message que vous venez de saisir.

10. Tagger un commit

✓ Vérifiez que vous êtes toujours bien positionner sur la branche **master**, si ce n'est pas le cas positionnez-vous dessus.

✓ Pour tagger le commit courant, il faut utiliser l'instruction **tag**.
Tapez l'instruction suivante : **git tag v1.0.0**

✓ Demandez l'**historique des commits** avec **git log**.
Vous devriez visualiser le tag sur le dernier commit.

A quoi sert un tag ?
Un tag permet de repérer plus facilement un commit.
Un commit est habituellement identifié via son hash.
Pour se déplacer sur un commit, il faut utiliser l'instruction **checkout** suivi du **hash**.
Si un tag a été placé sur un commit, il est possible de se repositionner sur ce commit à l'aide de l'instruction **checkout** suivi du **nom du tag**.

11. Gérer un conflit

Un conflit peut apparaître lors d'une fusion entre deux branches.
Un conflit pourra apparaître si les deux branches ont évoluées chacune de leur côté avec des modifications qui ne sont pas compatible. Nous allons illustrer cela à l'aide de l'exemple suivant.

11.1 Créer une nouvelle branche

✓ Nous allons créer une nouvelle branche **espagnol**.
Il est possible de créer une nouvelle branche et s'y positionner directement dessus en faisant appel à l'instruction **checkout** suivie de l'option **-b** (au lieu d'un appel à **branch** puis d'un appel à **checkout**)
Tapez l'instruction suivante : **git checkout -b espagnol**

✓ Visualisez toutes vos branches avec la commande **git branch** et vérifiez que l'étoile ***** indique bien la branche **espagnol**.

✓ Sur la branche **espagnol**, dans le répertoire **demo**, créez un nouveau fichier **test_ES.txt** que vous compléterez avec le texte suivant :
Hola Mundo !
git fue creado por Linus Torvalds en 2005.
Estoy usando git !
git es un software de control de versiones.
Enregistrez ce fichier.

✓ Toujours sur la branche **espagnol**, ouvrez le fichier **test.txt** et corrigez une petite erreur au passage 😉 c-a-d remplacez la première ligne **Bonjour tout le monde !** par **Bonjour le monde !** (c-a-d supprimer le mot **tout**)

✓ Faites un **git status**.

Cette commande vous indique entre autres que le nouveau fichier **test_ES.txt** n'est pas encore tracké.
Ajoutez ce *nouveau* fichier **test_ES.txt** dans l'index via la commande : **git add test_ES.txt**

✓ Committez toutes les modifications précédentes via l'instruction :
git commit -am "ajout git en espagnol"

11.2 Continuer à travailler sur la branche master

✓ Positionnez-vous maintenant sur la branche **master**

→ Ouvrez la fichier **test.txt**
Ajoutez comme 2ème ligne de ce fichier (après Bonjour tout le monde !), la ligne suivante :
git a été créé par Linus Torvalds en 2005.
Enregistrez ce fichier.

→ Ouvrez la fichier **test_EN.txt**
Ajoutez comme 2ème ligne de ce fichier (après Hello world !), la ligne suivante :
git was created by Linus Torvalds in 2005.
Enregistrez ce fichier.

→ **Committez** ces deux modifications en indiquant le message suivant : **"ajout création de git"**
→ Demandez l'**historique des commits** avec **git log**.

11.3 Fusionner les deux branches et gérer le conflit...

✓ En étant sur la branche **master,** demandez la fusion de la branche **espagnol** en tapant l'instruction suivante : **git merge espagnol -m "fusion message en espagnol"**

Un conflit (**CONFLICT**) est détecté.
git détecte un conflit lorsque deux personnes modifient la même zone de code en même temps.
Comme il ne peut pas décider seul quel code est le bon et doit être gardé, il signale un conflit.
git met alors la fusion en pause ce qui est indiqué dans le prompt par : **(master|MERGING)**

Pour voir les fichiers en conflit, vous pouvez taper : **git status**
Le **git status** indique deux solutions pour résoudre ce problème de *fusion* :
→ solution n°1 : résoudre le conflit puis redemander un **git commit**
→ solution n°2 : abandonner la fusion avec un **git merge --abort**

Nous allons mettre en place la solution n°1, c-a-d résoudre le conflit pour mener la fusion à son terme.

11.3.1 Résoudre manuellement le conflit de manière à obtenir le fichier souhaité

✓ Pour résoudre le conflit, git a modifié un fichier pour montrer les différences entre la branche en cours et celle que vous essayez de merger. C'est pour cela que lors du **git status**, le fichier **test.txt** apparaît en rouge : c'est le fichier qui a été modifié par git (**modified**).

✓ Ouvrir ce fichier (**test.txt**).
Recherchez la ligne contenant les symboles « **<<<<<<<<<** ». Ces symboles indiquent le début du conflit et délimitent le contenu de la branche de base (**HEAD**) en conflit.
Les symboles « **=========** » permettent de séparer les deux contenus en conflit.
Les symboles « **>>>>>>>>>** » indiquent la fin du contenu en conflit sur la branche à fusionner (ici la branche **espagnol**)

Il ne reste donc plus qu'à *nettoyer* ce fichier pour qu'il affiche uniquement le contenu à committer c-a-d supprimer les marqueurs de conflits (<, =, >), les noms de branches et procéder à une fusion manuelle de la *bonne version* à commiter.

✓ Nettoyez le fichier **test.txt** pour qu'il ne contienne plus que :
Bonjour le monde !
git a été créé par Linus Torvalds en 2005.
J'utilise git !
git est un logiciel de gestion de versions décentralisé.

✓ Puis enregistrez vos modifications.

11.3.2 Avertir git que le conflit a été résolu

Une fois le conflit résolu, il faut procéder à un nouveau commit pour terminer la fusion.
Il faut donc commencer par avertir git que le conflit a été résolu en ajoutant le fichier modifié dans l'index. Pour cela, tapez l'instruction : **git add test.txt**
Et vérifier à l'aide d'un **git status** que git a bien pris en compte que le conflit a été résolu.

11.3.3. Terminer la fusion par un commit

Il ne reste plus qu'à terminer la fusion en (re)demandant le commit de fusion.
Pour cela, tapez : **git commit -m "fusion message en espagnol"**

Si le commit s'est bien passé, vous devriez avoir un prompt qui affiche à nouveau seulement **(master)**

Remarque : Vous pouvez également vérifier la prise en compte de ce commit avec un petit **git log** ou **git show**.

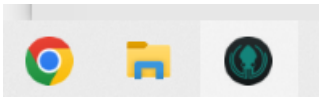
12. Visualiser un workflow à l'aide d'un éditeur graphique

Un outil graphique peut simplifier la visualisation et la gestion du workflow.
Il existe plusieurs outils graphiques. Vous en trouverez des listes :
ici (https://git-scm.com/downloads/guis) ou
là (*https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools#Graphical_Interfaces*)
Dans ce tutoriel, nous allons par exemple installer **gitKraken**, mais vous pouvez très bien choisir un autre outil graphique dans les listes précédentes si ça vous fait plaisir 😉.

Rendez-vous sur le site : https://www.gitkraken.com/.
Cliquez sur **Try Free** et sélectionnez le **Download GitKraken Client Free** pour le télécharger. Une Fois téléchargé, exécutez-le et sélectionnez la manière dont vous voulez vous connecter.
Si vous déjà avez un compte **Github**, utiliser l'option hosting service 😉

Remarque : La période d'essai dure **7 jours**, c'est parfait pour le TP aujourd'hui 😉
Si vous souhaitez continuer à utiliser par la suite Gitkraken, vous pourrez le faire en profitant du pack pouvoir profiter du pack **Github Student Developper Pack** comme indiqué ici :
https://www.gitkraken.com/github-student-developer-pack-bundle (lien à regarder une fois le TP terminé 😉)

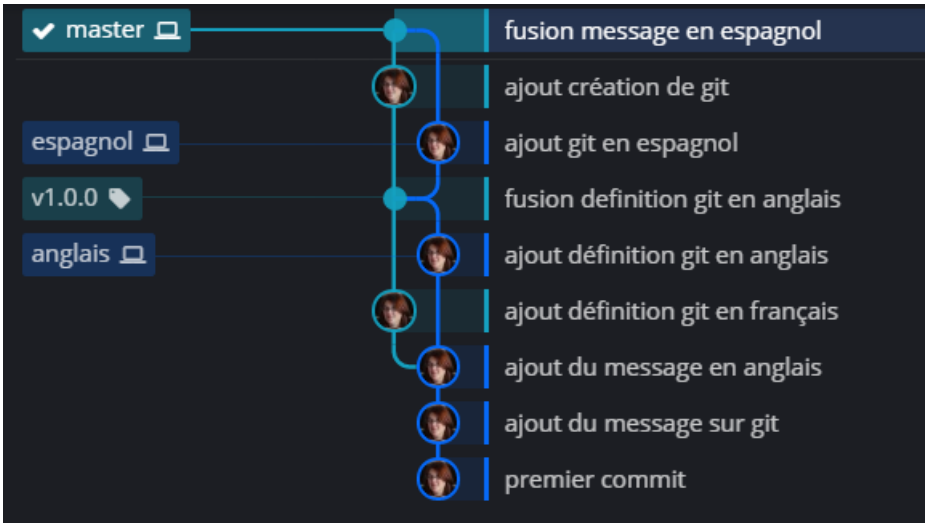
Ouvrir **GitKraken.** Sous Windows, GitKraken est accessible, après cette installation, dans la barre des tâches .



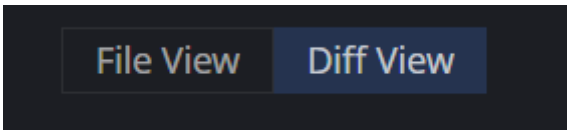
Complétez éventuellement le profil, puis choisir l'option **Open an existing repository on your machine** du **Welcome to GitKrakenClient !**

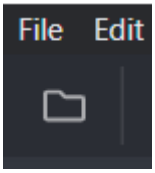
Sélectionnez alors votre répertoire **demo**.

Votre workflow s'affiche alors dans la fenêtre.
Pour *bien* faire apparaître les branches :placez-vous sur un trait vertical (une double flèche s'affiche) et tirez de manière à obtenir :



Cliquez sur un commit. Son détail s'affiche à droite.
Lorsque ce détail est affiché, vous pouvez **cliquer sur un fichier** pour voir son contenu (**File View**) ou son **diff** par rapport au commit précédent (**Diff View**)





Remarque : Par la suite, lorsque vous souhaiterez ouvrir de nouveaux dépôt, il vous faudra cliquer sur l'icône en forme de répertoire en haut à gauche (au-dessous du menu **`File`**) pour pouvoir ouvrir un nouveau répertoire.

Remarque : GitKraken (et tout ce genre d'outil graphique) vous permet de manipuler facilement votre workflow. Nous ne nous attarderons pas plus que cela dans le cadre de ce tutoriel, mais si vous voulez en savoir plus sur les possibilités, le site de GitKraken propose pas mal de documentation, notamment :

- Un aide-mémoire récapitulatif sur GitKraken : https://www.gitkraken.com/pdfs/gitkraken-git-gui-cheat-sheet
- Un aide-mémoire récapitulatif sur GitKraken et Github: https://www.gitkraken.com/pdfs/gitkraken-for-github-cheat-sheet
- Une présentation des fonctionnalités : https://www.gitkraken.com/git-client/features
- De nombreuses vidéos en bas de la page : https://www.gitkraken.com/github-student-developer-pack-bundle
- de nombreuses ressources autour de git (GitLibrary) : https://www.gitkraken.com/learn/git

- GitKraken Tutorial: For Beginners (4 minutes) (https://www.youtube.com/watch?v=ZKkMwTeAij4)
- et comprendre git : GitKraken (12 minutes) (https://www.youtube.com/watch?v=daBPgzan_wI)

13. Supprimer une branche

Lorsque vous fusionnez une branche, cette branche existe toujours dans votre workflow.
Par exemple, si vous tapez dans la console **git branch,** vous visualisez que les branches **anglais**, **espagnol** et **master** sont présentes et donc toujours accessible.

Pour supprimer une branche, il faut utiliser l'option **-d** de la commande **branch**
Par exemple, pour supprimer la branche **anglais**, tapez dans la console : **git branch -d anglais**
Faites maintenant un **git branch** et vous constaterez que seules les branches **master** et **espagnol** sont désormais connues.

Remarques :

- ➔ Si vous visualisez votre workflow à l'aide de GitKraken, vous constaterez bien sûr également que la branche **anglais** a disparu du workflow😉
- ➔ Il peut être utile de supprimer des branches pour nettoyer le **workflow** (l'historique). Des commandes telles que **rebase** permettent par exemple de réorganiser le workflow. Nous ne travaillerons pas sur ces commandes dans le cadre de ce tutoriel, **si vous êtes interessés des références à propos de merge et rebase vous sont données dans le cours** 😉.

Créer un nouveau dépôt git peut se faire de deux manières :

- ➔ *Soit il s'agit d'un nouveau dépôt, l'instruction **git init** est utilisée pour mettre en place le contrôleur de version : c'est ce que vous devez faire sur vos nouveaux projets.*
- ➔ *Soit il existe déjà un dépôt git du projet sur lequel vous travaillez et vous souhaitez récupérer ce dépôt (son historique) pour travailler en local, dans ce cas-là il faudra **cloner le dépôt existant**, c'est que nous allons voir maintenant avec **git clone**.*

14. Cloner un dépôt existant distant

Cloner un dépôt existant consiste à récupérer tout l'historique et tous les codes source d'un projet en utilisant l'instruction **git clone**.

✓ Depuis la console, placez-vous dans votre répertoire de travail à l'endroit où vous souhaitez cloner votre nouveau dépôt (sortez par exemple de **demo** avec l'instruction : **cd ..**).
Le prompt ne vous indique plus **master** puisque vous n'êtes plus (ou pas encore 😉) dans un dépôt git….

✓ Vous allez récupérer en local le dépôt existant distant : https://github.com/iblasquez/geekgit
Pour cela, tapez l'instruction : **git clone https://github.com/iblasquez/geekgit.git**

✓ Rendez-vous ensuite dans le répertoire **geekgit** qui vient d'être créé : **cd geekgit**
Le prompt vous indique que vous vous trouvez sur **master** !

✓ Consultez l'historique des commits en tapant **git log**.
Ceci vous confirme bien que lors d'un **git clone**, git fournit à chaque développeur sa propre copie du dépôt avec son propre historique.
Remarque : si vous le souhaitez, vous pouvez aussi visualiser l'historique de ce dépôt avec GitKraken 😉.

15. Travailler en local sur un dépôt cloné

Faites-vous plaisir, effectuez des modifications en local dans ce dépôt en prenant soin de commiter régulièrement et de manipuler toutes les commandes git vues précédemment…

16. Synchroniser les dépôts

Nous ne traiterons pas cette partie dans le cadre de ce tutoriel : elle sera mise en pratique lors de la séance suivante.
Prenez quand même le temps de consulter les transparents du cours relatifs à la partie **Synchroniser les dépôts** et notez bien que :

- ➔ grâce à l'instruction **git push origin master**, vous pourriez **publier vos modifications** (dépôt local) vers le dépôt distant (remote) c-a-d le dépôt que vous venez de cloner.
- ➔ grâce aux instructions **git pull origin**, vous pourriez **récuperer les commits du dépôt distant** c-a-d mettre à jour votre historique local. Cette instruction revient à réaliser deux commandes :
  - la première qui consiste à télécharger les commits du dépôt distant : **git fetch**
  - et la seconde qui consiste à fusionner ces commits avec votre historique local : **git merge**

Cet énoncé est également disponible en ligne sur :
https://github.com/iblasquez/tuto_git

17. Un peu de lecture …

Après ces petites manipulations,
vous allez apprécié
la lecture de l'article suivant :

GIT, compétence de base pour la classe développeur

extrait du **LOG_MAG de Mars 2022** et écrit par un ancien étudiant du Département Informatique du Limousin devenu aujourd'hui un de vos vacataires 😉

GIT, compétence de base pour la classe développeur

8 minutes

Arnaud STRILL

Arnaud Strill, intervenant en PHP a voulu vous donner des bases solides en git écrites en mode JDR afin de mieux comprendre l'outil.

En parcourant le Log de février, j'ai lu l'article de Paul Rezonnico à propos du Jeu De Rôle (mettre le lien de l'article), le JDR c'est cool, faites du JDR. Mais si vous alliez plus loin ? En ne faisant pas que jouer, mais en faisant un blog qui retrace vos parties, ou encore une app qui va automatiser votre fiche de perso, ou bien un jeu en javascript qui adapte les règles du JDR plutôt que d'avoir à se fader des montagnes de tableaux ? Vous avez des potes rôlistes, vous pouvez faire un projet sur le rôlisme !

Git peut vous aider et je me propose de faire le tour de quelques cas de figure intéressants, et je vous le présente en Command Line Interface, pourquoi ? Parce que la CLI, c'est la vie ! Alors si ce n'est pas déjà fait, installez git sur votre machine et créez-vous un compte GitHub / GitLab / GitAre / GitouEtSonPtitAccordéon,...

Un peu de background pour comprendre la partie Quelques points de repère pour la suite, lorsque l'on parle de repository c'est un projet stocké chez votre hébergeur Git, dans les faits, c'est un dossier contenant des fichiers le tout avec une surcouche propre à Git qui va permettre de garder la trace de chaque modification enregistrée.

Des modifications que l'on regroupe par commit, qui gardent la trace des lignes modifiées par fichier, par qui, à quelle date, quel commentaire a été donné. Ces commits sont identifiés par leurs SHA, un code alphanumérique genre cb22b4fe9b780565e89eb8f471704e711ea909a5.

Ces commits sont placés sur des branches, il y a une branche principale avec des appellations libres, mais par convention, c'est généralement "main", "trunk" ou "master". Des branches annexes (aussi d'appellation libre) peuvent être créées en partant de la branche principale (ou de n'importe quel commit en fait, mais on va rester simple) pour faire un développement spécifique sans perturber le reste de l'application.

Considérez pour l'analogie dans l'article que le commit est une enveloppe avec le SHA dessus (ce qui permet de savoir après quel autre commit et sur quelle branche se place celui-là) contenant les modifications pour chaque fichier modifié.

Une fois le développement terminé, on va faire une demande de fusion (ou Merge Request dans la langue de John Malkovitch) qui résumera les modifications, et qu'on pourra accepter ou rejeter. Si on la rejette, ben RIP in peace la branche ; si on l'accepte, le(s) commit(s) de la MR seront ajoutés à la branche principale et tout le monde peut récupérer une nouvelle version pour la suite.

Chaque membre de l'équipe possède une copie du projet qu'il doit remettre à jour régulièrement pour ne pas tout péter en envoyant sa version périmée, comme un gros bourrin.

Pour voir l'organisation de votre projet, des interfaces existent bien sûr, mais je préfère vous donner les instructions en ligne de commande parce que c'est plus robuste, plus rapide (si !) et plus fiable, parce que la CLI (Command Line Interface) fera ce que vous souhaitez, là où un IDE (Integrated … OK j'arrête) ou un logiciel va caler des options mine de rien, ce qui pourrait créer une situation qui ne serait pas telle que vous l'imaginiez (c'est du vécu). Et personne n'a envie d'être fusillé du regard par une équipe de devs en colère.

Maintenant, la théorie, c'est bien beau, je vais essayer de vous expliquer ça en situation "réelle".

Vous êtes dans une taverne, quand soudain un vieux vous interpelle… Toute aventure a un commencement et pour un projet, c'est pareil. Le Ranger, ayant installé Git, va donc créer un dossier sur son appareil d'informancie pour narrer ses exploits, et comme il est motivé, il l'écrit.

```
$RangerDuKo/> mkdir chroniquesJDR &&
cd chroniquesJDR

$RangerDuKo/chroniquesJDR/> touch
a_l_aventure_compagnons.txt

$RangerDuKo/chroniquesJDR/> notepad
a_l_aventure_compagnons.txt

#a_l_aventure_compagnons.txt

Sa y est! Fini le rafistolage de
chéses en bois, à moi l'aventure sur
lé chemins!
```

Le ranger a peu de points d'INTelligence, d'où ses fautes diverses, on y reviendra.

Le Ranger va alors utiliser Git pour démarrer la sauvegarde de son épopée de fier aventurier et s'empressera de le publier sur le repository qu'il n'aura pas manqué de créer sur GitLab (ou autre).

```
$RangerDuKo/chroniquesJDR/> git init

Initialized empty Git repository in
<...>/chroniquesJDR/.git/

$RangerDuKo/chroniquesJDR/> git
remote add origin
git@gitlab.com:LeRangerDuKO/chronique
sjdr.git

$RangerDuKo/chroniquesJDR/> git add .

$RangerDuKo/chroniquesJDR/> git
commit -m "Le début d'une grande
aventure !"

$RangerDuKo/chroniquesJDR/> git push
-u origin main
```

Ça y est la saga du Ranger débute !

Une femme armée d'un bâton enchâssé d'un cristal pourpre s'assoit à votre table Un JDR seul, c'est pas la folie, donc on invite les potes pour bien rigoler, la Magicienne va donc rejoindre le Ranger, en plus elle connaît les bouquins, elle pourra rendre ça plus lisible, pratique !

```
$Mago/> git clone
https://gitlab.com/LeRangerDuKO/chron
iquesjdr.git

$Mago/> cd chroniquesjdr
```

Tant qu'à faire, elle va aussi stocker son grimoire, pour ça elle va
- créer une branche
- copier le fichier-grimoire contenant ses sorts
- indiquer l'ajout d'un nouveau fichier à Git
- créer un commit avec un message explicatif
- et pousser tout ça sur le serveur

Ce qui donne respectivement ces commandes :

```
$Mago/> git checkout -b ajoutGrimoire

$Mago/> cp
../bouquins/grimoire_des_arcanes_mine
ures.pdf
grimoire_des_arcanes_mineures.pdf

$Mago/> git add .

$Mago/> git commit -m "Ajout du
grimoire des Arcanes Majeures"

$Mago/> git push --set-upstream
origin ajoutGrimoire
```

En arrivant sur le repository du projet (sur GitLab, par exemple) une Merge Request (demande de fusion de branche) peut être faite.

Le groupe doit maintenant se diviser pour couvrir plus de terrain Du côté du Ranger, l'arrivée d'une coéquipière est un nouvel évènement, donc : nouvelle entrée dans le fichier de départ, en faisant lui aussi une branche (parce qu'en dehors du commit initial, on n'ajoute pas de commit directement sur la branche principale, sinon c'est malus de 4 à tous les jets de CHArisme jusqu'en fin de partie) :

```
$RangerDuKo/chroniquesJDR/> echo "Une
majiciene arrive dans le groupe ! Ces
le débus de la gloire !" >>
a_l_aventure_compagnons.txt

$RangerDuKo/chroniquesJDR/> git add .
&& git commit -m "l'équipe gagne un
compagnon" && git push -u origin
arriveeMago
```

Et même topo que précédemment, une MR peut être publiée et validée par les autres membres de l'équipe. Ne jugeant pas nécessaire de vérifier son travail (c'est lui le chef, c'est lui qui décide !), le Ranger va fusionner directement son travail, ajoutant alors le commit de sa branche sur la branche principale du projet. C'est là que les ennuis commencent.

La Magicienne, ne soupçonnant rien parcourt pendant ce temps le fichier qui vient d'être modifié… sauf qu'il ne l'est pas pour elle ! La version du fichier a_l_aventure_compagnons.txt qu'elle peut lire, c'est celle du commit initial, pour pouvoir lire les changements apportés elle devrait :

- se replacer sur la branche principale
- récupérer la liste des modifications qui ont eu lieu sur le projet
- constater alors que sa branche main est en retard sur la dernière version et donc, appliquer les dernières modifications

```
$Mago/> git checkout main

$Mago/> git fetch

$Mago/> git pull
```

Mais admettons qu'elle N'AIT PAS mis à jour sa branche principale, qu'elle ait lu le fichier texte du Ranger et (après avoir essuyé les larmes de sang causé par les fautes) qu'elle se soit empressée de faire une branche, de modifier le contenu du fichier :

```
$Mago/> (main) git checkout -b
correctionFautes

## édition du fichier ##

$Mago/> (correctionFautes) git add .
&& git commit -m "Merci de laisser
les mots tranquilles, ils n'ont rien
fait pour mériter pareil traitement
!" && git push --set-upstream origin
correctionFautes

#a_l_aventure_compagnons.txt

Ça y est ! Finie la confection de
sièges de bois ! Le chemin de
l'aventure s'ouvre devant moi
désormais !!
```

... et de pousser ça sur le serveur...
C'est un conflit de version que l'on doit alors résoudre et la Magicienne va devoir :
- mettre à jour son projet
- actualiser sa branche par rapport à la branche principale du serveur
- passer en revue les changements conflictuels apportés par le Ranger (re-pleurer du sang) et rapatrier ses modifications à la suite (une fois les conflits résolus =>)

```
$Mago/chroniquesjdr>
(correctionFautes) git fetch

$Mago/chroniquesjdr>
(correctionFautes) git rebase -i
origin/main

## gestion des conflits ##

$Mago/chroniquesjdr>
(correctionFautes) git add . && git
rebase --continue
```

Résultat, la Magicienne devrait avoir :

```
#a_l_aventure_compagnons.txt

Ça y est ! Finie la confection de
sièges de bois ! Le chemin de
l'aventure s'ouvre devant moi
désormais!!

<Au-dessus, les lignes corrigées par
la Magicienne, en-dessous, les lignes
ajoutées par le Ranger>

Une majiciene arrive dans le groupe!
Ces le débus de la gloire!
```

```
#a_l_aventure_compagnons.txt

Ça y est ! Finie la confection de
sièges de bois ! Le chemin de
l'aventure s'ouvre devant moi
désormais !!

Une praticienne des arts occultes
daigne battre la campagne dans ma
compagnie ! Quel honneur !
```

Cette modification pourrait s'ajouter à la précédente (avec la création d'un nouveau commit comme vu précédemment) mais est-ce bien utile d'avoir deux étapes partielles de correction ? Il semble plus judicieux de ne conserver qu'une étape de correction qui va enregistrer toutes les modifications de la Magicienne, on va alors préférer l'utilisation d'une option de commit, permettant la modification du dernier commit : "amend".

```
$Mago/chroniquesjdr>
(correctionFautes) git add . && git
commit --amend
```

Cette commande va rouvrir le commit précédent, ce qui laisse la possibilité de changer aussi le commentaire, au besoin.

Et comme l'historique de la branche correctionFautes a changé (rebase de la branche main et réécriture du précédent commit par un nouveau) il faut alors forcer l'action de git push vers le serveur :

```
$Mago/chroniquesjdr>
(correctionFautes) git push --force
```

La harpie est morte à vos pieds et vous vous préparez pour un nouveau combat Pour aller plus loin je vous invite à faire du JDR ; pour les débutants, il y a Naheulbeuk, qui est gratuit et bien loufoque. Vous renseigner sur les workflows Git, ce qui peut vous permettre de trouver une organisation qui vous conviendra pour vos travaux de groupes, et aussi regardez ce qui peut se faire avec les Git hooks et la CI/CD pour automatiser votre process de développement et de déploiement.

6

18. Pour les plus rapides …
Prise en main des commandes de Git au travers d'exercices interactifs en ligne

➔ Commencez par la **manipulation autour des branches** :
Learn Git Branching accessible directement via **http://learngitbranching.js.org/**

➔ Continuez par la **simulation d'un workflow et la manipulation/visualisation de commandes comme revert ou cherry-pick** :
Visualizing git accessible directement via **http://git-school.github.io/visualizing-git/**

➔ Terminez par **Git-It** : une application multi-plateforme (à installer) proposant des défis utilisant *vraiment* git et GitHub sans passer par un émulateur...

Remarques :
- Les précédentes ressources sont accessibles : **https://try.github.io**
- **Git-Gud** est un autre simulateur web très simple d'utilisation pour manipuler/visualiser les commandes de bases de git. Il est accessible sur **https://nic-hartley.github.io/git-gud**

Annexe

✓ **Pro Git Book :** Documentation accessible en ligne
- ➔ Version française : https://git-scm.com/book/fr/v2
- ➔ Version anglaise :https://git-scm.com/book/en/v2/

✓ **Editeur vim :**
- ➔ Attention, par défaut la console **git Bash** va vous ouvrir **l'éditeur vim**.

Un petit lien utile vers la liste des commandes **vim** : https://doc.ubuntu-fr.org/vim
A noter : **:w** => enregistrer le fichier
:q => quitter vi
:wq => enregistrer le fichier et quitter vi
:**q !** => quitter vi en annulant les changements

➔ Sachez que si cet éditeur ne vous convient pas, il est possible d'en configurer un autre via la commande : **git config –global cores.editor nomEditeur**
Pour en savoir plus : https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Param%C3%A9trage-%C3%A0-la-premi%C3%A8re-utilisation-de-Git

✓ **Autres commandes git qui pourraient vous être utile…**
- ➔ **git commit --amend**
pour pouvoir modifier le dernier message de commit

- ➔ **git stash**
  - pour éviter d'avoir à faire un commit au milieu d'un travail en cours : les fichiers sont sauvegardés et mis de côté (le *working directory* apparaîtra alors comme *propre* si vous faites un **git status**)
  - **git stash apply** pour vous permettre de restaurer les fichiers *stashés* : les fichiers seront restaurés et retrouveront l'état dans lequel ils étaient avant **git stash**

✓ **Autres liens :**
- ➔ **Essential git commands every developer should know :**
https://dev.to/dhruv/essential-git-commands-every-developer-should-know-2fl

- ➔ **Théorie des graphes appliqués à git :**
https://mixitconf.org/en/2017/la-theorie-des-graphes-appliquee-a-git
https://speakerdeck.com/ubermuda/la-theorie-des-graphes-appliquee-a-git