Tuto GIT/github

Objectif: Revue complète des commandes **Git** de base de partage des sources d'un projet par deux ou plusieurs développeurs d'application.

GIT Première Partie : Manipulation en locale

Etape 1 :Configuration de git en local :

Dans la console, Tapez ces trois commandes:

permettant la `Configuration de la couleur des affichages des logs`

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

Lors de la première configuration, il faut configurer son profil utilisateur, a minima :

```
git config --global user.name "votre_pseudo"
```

```
git config --global user.email user@email.com
```

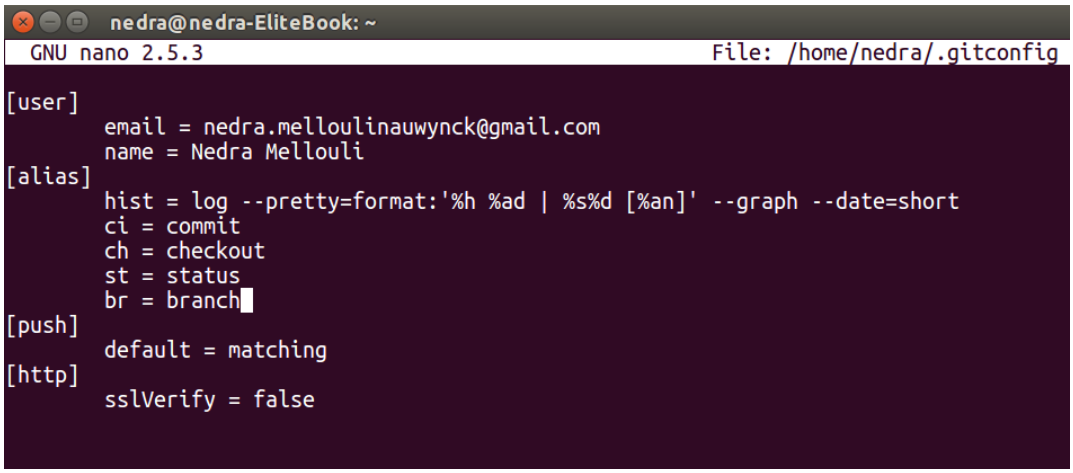
```
git config --global core.editor nano
```

Vous pouvez aussi éditer votre fichier de configuration `.gitconfig` situé dans votre répertoire personnel pour y ajouter une section alias à la fin. Les alias cervent à utiliser des raccourcis de c commandes Git. Par exemple , au lieu d'écrire `commit`, il suffit de faire `git ci`, ce qui est plus court.

Lancer la commande suivante :

```
>nano ~/.gitconfig
```

Résultat :



Etape 2 : Création d'un nouveau dépôt ou « cloner » un dépôt existant

Liste des commandes : git init

Création d'un nouveau dépôt Git :

Commencez par créer un dossier du nom de votre projet sur votre /home/GestionProj/GestionStages

`mkdir /home/GestionProj/GestionStages`

Ensuite, initialisez un dépôt Git tout dans ce dossier avec la commande

`cd /home/GestProj/GestionStages/`

>**git init**

Le résultat est :

>**Initialized empty Git repository in /home/nedra/GestionProj/GestionStages/.git/**

Le système vous a créé un fichier caché .git. Pour le vérifier tapez la commande `ls -al`

Etape 3 : Modification du projet en local

Liste des Commandes : git add, git status, git commit git diff

But : Garder des versions d'un même fichier pour pouvoir récupérer des versions très récentes en cas de problème (bug, fichier corrompu, effacement par erreur, etc)

Exercez-vous à travailler en local. Basculez entre les perspectives Git et JAVA pour vérifier les résultats.

3.1 Premier commit : modification du code

- Créez la classe **Entreprise** depuis Eclipse
- Lancer la commande `git status` (ou bien **git st**), elle vous indique les fichiers que vous avez modifiés récemment . Une version plus courte de la commande :
  - **git status -s ou git status –short**

Réponse : (rien n'a été ajouté et poutant une classe a été ajoutée?)

On branch master

Initial commit

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

Ce message nous informe que rien n'a été modifié (*nothing to commit*).

- Commit : pour enregistrer définitivement les fichiers modifiés dans le repository git de votre projet (master), il faut préciser explicitement les fichiers que vous voulez « commiter ». Pour cela, trois possibilités :
  - **Possibilité 1 : préciser les fichiers à ajouter puis commiter**

```
>git add nomfichier1 nomfichier2 … nomfichiern
>git commit -m « votre message »
>git status
```

  - **Possibilité 2 :commiter tous les fichiers**

```
>git commit -a -m « votre message »
>git status
```

  - **Possibilité 3 : commiter que quelques fichiers à préciser**

```
git commit nomfichier1 nomfichier2 … nomfichiern
git status
```

Réponse : (git ne connais pas encore le ou les fichiers créés récemment)

```
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
     documentation.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Attention : il faut dans ce cas utiliser git add

Annuler un commit effectué par erreur

git reset HEAD^
git reset HEAD
git reset HEAD~2

Etape 4 : Inspecter les modifications et des log

Liste des commandes : git diff git status git log

But : de pouvoir suivre à tout moment les modifications apportées sur vos fichiers (status) et de savoir les différences (diff) entre la version actuelle et la dernière version stable. Accès à l'historique des opérations par git log

git diff montre les lignes exactes qui ont été ajoutées/modifiées/supprimer sur les fichiers qui indexés et non commités.

Pour le tester, vous modifiez le contenu d'un fichier déjà indexé (déjà commité). Puis vous lancez la commande

```
>git diff
```

Le résultat sur mes fichiers est le suivant :

```
diff --git a/Client.java b/Client.java
index 0db7bc0..d013552 100644
--- a/Client.java
+++ b/Client.java
@@ -9,10 +9,11 @@ public class Client {
	private Integer tel;
	private String prenom;
-	public Client (String nom, String adresse, Integer tel) {
+	public Client (String nom, String adresse, Integer tel, String prenom)
{
		this.nom = nom;
		this.adresse = adresse;
		this.tel=tel;
+		this.prenom=prenom;
```

La ligne --- a/Client.java veut dire le contenu avant modification et qui a disparu

la ligne +++ b/Client.java veut dire ce qui a été ajouté

Donc la différence entre la version commité et la version actuelle se résume à la modification du constructeur de la classe Client auquel un attribut prenom a été ajouté.

En lancant la commande git status, le résultat est comme suit

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   Client.java
        modified:   Compte.java
```

Changes not staged for commit: veut dire que les fichiers modifiés n'ont pas été indexés pour les commiter (il faut donc lancer la commande add)

Etape 5 : Ajout d'une nouvelle Branche en local et deplacement d'une branche à une autre

Liste des commandes : git branch et git checkout

A quoi sert les branches ? De pouvoir garder une version toujours stable sur la branche master et travailler sur une version en développement sur une autre branche. Par convention, on la nomme la branche dev

Pour créer une nouvelle branche dans votre git on appelle la commande

```
> git branch  nomNouvelleBranche
```

Exemple : nous souhaitons créer une nouvelle branche que l'on nomme dev

> git branch dev

Pour lister toutes les branches on appelle la commande

```
> git branch - -all

 Résultat :

dev

* master
```

On remarque que master est précédé d'une *. Git nous indique le nom de la branche active (master)

On bascule sur la branche dev avec :

```
> git checkout dev

Résultat :
```

```
M       Client.java
M       Compte.java
Switched to branch 'dev'
```

Maintenant, nous allons créer une nouvelle classe Banque.java. On ajoutera tout d'abord l'id de la banque, puis nous commitons le fichier. Ensuite, nous ajoutons le nom de la banque, puis commiter, ajouter l'adresse et client. Puis commiter (3 commits au total)

Note : si les fichiers ne sont pas commités avant de changer de branche, Git refuse de basculer, voir exemple de réponse ci-dessous

```
>git checkout master
Résultat :
error: Your local changes to the following files would be overwritten by checkout:
        Client.java
Please, commit your changes or stash them before you can switch branches.
Aborting
```

Après modification sur la branche dev, on souhaite merger le travail fait sur cette branche avec celui sur la branche master. Après avoir commité sur dev, vous retournez sur la branche master en utilisant la commande :

```
>git checkout master
Résultat :
Switched to branch 'master'
```

Nous allons merger les deux branches dev et master depuis master pour rendre master stable après l'ajout de Banque.

```
>git merge dev
```

Résultat :

Auto-merging Client.java

Merge made by the 'recursive' strategy.

Banque.java | 51 +++

Client.java | 4 ++++

2 files changed, 55 insertions(+)

create mode 100644 Banque.java

Ici le merge s'est déroulé sans problème ou sans conflit. Or ce n'est pas toujours le cas.

Pour provoquer un conflit, il suffit de modifier sur master un de vos fichier, par exemple Banque.java en ajoutant un constructeur vide et vous commiter. Puis nous retournons sur la branche dev et vous effectuez un autre changement sur le même fichier puis vous commitez. Retour sur la branche master et vous lancez `git merge dev.` Voici la synthèse des commandes :

depuis la branche master

>git commit -a -m « modif Client.java→constructeur par défaut »

>git checkout dev

modification du fichier Client.java

>git commit -a -m « modif de client depuis la branche dev »

>git checkout master

>git merge dev

Résultat :

Auto-merging Client.java

CONFLICT (content): Merge conflict in Client.java

Automatic merge failed; fix conflicts and then commit the result.

git diff nous affiche le détail comme ceci :

```
diff --cc Client.java
index be68a0c,cb96503..0000000
--- a/Client.java
+++ b/Client.java
@@@ -10,11 -13,8 +10,16 @@@ public class Client
       private Integer tel;
       private String prenom;

++<<<<<<< HEAD
 +            public Client (String nom, String adresse, Integer tel, String prenom) {
 +                  this.nom = nom;
 +                  this.adresse = adresse;
 +                  this.tel=tel;
 +                  this.prenom=prenom;
++=======
+            public Client () {
+
++>>>>>>> dev

            }
```

Le message signale que le conflit vient du fichier Client.java. Il signale que dans la version sur le **master** :

```
 ++<<<<<<< HEAD
 +            public Client (String nom, String adresse, Integer tel, String prenom) {
 +                  this.nom = nom;
 +                  this.adresse = adresse;
 +                  this.tel=tel;
 +                  this.prenom=prenom;
++=======
```

est différente de celle sur la branche **dev**

```
++=======
+            public Client () {
+
++>>>>>>> dev
```

Alors dans ce cas il faut prendre une décision entre vous quelle version faut il garder !

Comment gérer les conflits ?

Il suffit d'aller sur votre eclipse, et éditer chacun des fichiers source de conflit (ici Banque.java). Eclipse vous affiche exactement le même message affiché avec la commande git diff, c'est-à-dire celui-ci pour mon cas :

```
++<<<<<<< HEAD
 +            public Client (String nom, String adresse, Integer tel, String prenom) {
 +                  this.nom = nom;
 +                  this.adresse = adresse;
 +                  this.tel=tel;
 +                  this.prenom=prenom;
++=======
+             public Client () {
+
++>>>>>>> dev

                }
```

Il faut donc décider ce qu'il faut laisser, enlever, etc. Après résolution de ces conflits, votre fichier devra être prêt à être commité sur la branche master. En réalité, git commitera la même version sur les deux branches.

GIThub Deuxième Partie : Manipulation en Remote

Liste des commandes :

git -c « http.proxy=adress:port »

en local et en remote (modif des deux côtés)

git fetch

git diff master origin/master

git merge

git pull (fetch + merge)

gestion des conflits

On veut pousser le contenu de notre projet local au niveau de la branche master distante sur github

Il faut donc tout d'abord s'assurer que vous avez déjà un compte github. Si ce n'est pas le cas, vous devrez vous créer un compte github. Puis créer un projet portant le même nom que votre projet en local.

```
> git remote add origin https://github.com/login/nomProjet.git
```

L'emplacement du repo est sur le serveur Git `https://github.com/votreLogin/NomProjet.git`. On peut consulter la liste des remote associés à notre repo avec l'option -v.

Note : Sous linux, on peut visualiser les remote avec la commande tree

```
> tree .git/refs
```

Pusher une première modification sur le repo distant

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

-u est un raccourci pour –set-upstream, donc les prochaines fois, on pourra juste faire git push

Modifier un fichier sur l'interface distante (donc sur github). Récupérer les modifications distantes non encore présentes en local.

```
> git fetch
> git diff master origin/master
> git merge
```

La commande **git diff** permet de comparer la branche locale master avec la branche master du remote origin
fetch permet de récupérer les modifications distantes. On peut ensuite les intégrer avec la commande merge.

Pour récupérer la dernière version depuis le serveur github, il faut lancer la commande

```
>git pull https://github.com/votreLogin/NomProjet.git master
```

Note : la commande fetch suivie par la commande merge revient à lancer directement la commande pull

Fin !

Synthèse des commandes

git init nomDuprojet

git add -a

git commit -a -m « message »

git merge

git checkout NomBranche

git lop -p

git push -u origin master

git clone urlLocationDuprojet

git remote add origin

git log

git -c « http.proxy=adress:port »

en local et en remote (modif des deux côtés)

git fetch

git diff master origin/master

git merge

git pull (fetch + merge)

gestion des conflits

Références

git cheat sheet. https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf.

git cheat sheet interactif. http://ndpsoftware.com/git-cheatsheet.html.

git livre. https://git-scm.com/book/en/v2.

git page d'accueil. https://git-scm.com/.

git tutoriel. https://git-scm.com/docs/gittutorial.

http://git-scm.com/book/fr/v2