Règles d'utilisation de Git

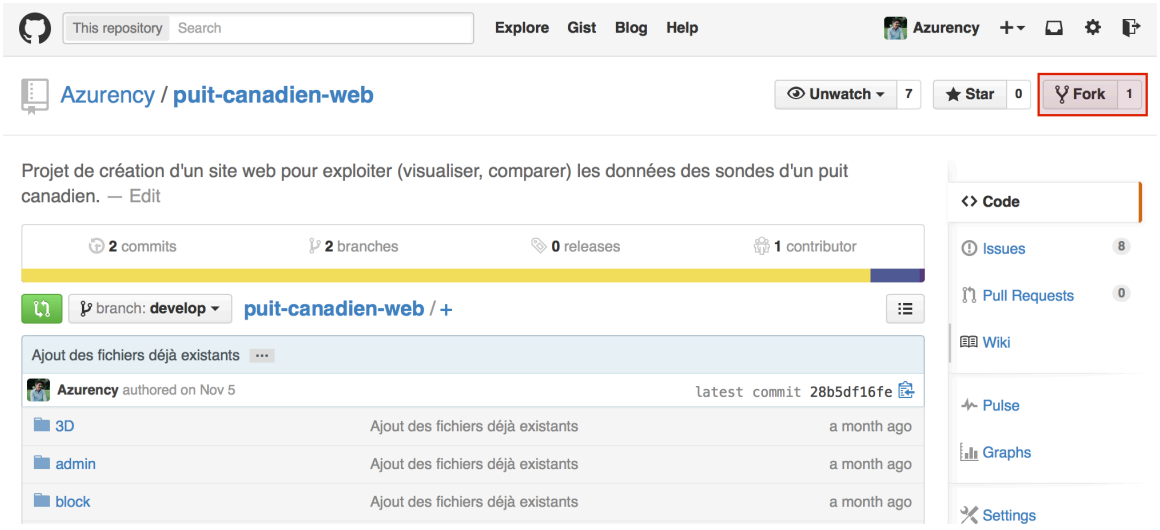
Projet géothermie - Groupe exploitation des données

Pour faciliter la collaboration au sein de notre groupe, il est indispensable que chacun respecte des règles vis-à-vis de l'utilisation de git. Ce guide rassemble toutes les bonnes pratiques à mettre en oeuvre lors du projet.

Mise en place

Fork le projet principal

Première étape d'une bonne collaboration : **fork** le projet principal. Ainsi personne ne pollue le dépôt principal qui est maintenu propre par l'intégrateur git du groupe.
Pour fork le projet principal, il suffit de cliquer sur le bouton `Fork` :



Cloner le projet sur sa machine

Pour pouvoir travailler sur le projet, il faut en récupérer une copie sur sa machine en utilisant la commande `git clone` :

```
git clone https://github.com/<ton identifiant github>/puit-canadien-web.git
```

Méthodes de travail

Maintenant que le projet est configuré en local, nous allons parler de l'organisation qu'il faut avoir lors du développement.

La rédaction des messages de commit

Les messages de commit sont au coeur de la collaboration dans git, ils doivent donc être de qualité et bien rédigé. Il est impératif de tout le monde respecte les mêmes règles :

- Ne pas commit un message avec la commande `git commit -m "un message sur une seule ligne"`. Si vous ne savez pas quoi mettre dans votre commit ou que vous pensez que son explication peut tenir sur une seule ligne, la plupart du temps, c'est que vous n'avez pas à commit !
- Quand vous pensez qu'il est vraiment temps de commit, utilisez seulement la commande `git commit -m` et remplissez le message de commit dans nano, vim ou votre éditeur préféré. Le message doit respecter ces conventions : une description courte du message sur la première ligne (50 caractères), un saut de ligne et une description complète du commit, des modifications, des fix, des ajouts, etc.

Exemple

```
Changed paragraph separation from indentation to vertical space.

Fix: Extra image removed.
Fix: CSS patched to give better results when embedded in javadoc.
Add: A javadoc {@link} tag in the lyx, just to show it's possible.
- Moved third party projects to ext folder.
- Added lib folder for binary library files.
Fix: Fixed bug #1938.
Add: Implemented change request #39381.
```

Les branches master et develop

Regardons ce qui est présent de base sur notre dépôt local avec la commande `git branch -a`, vous devriez avoir quelque chose de cette forme là :

```
* develop
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

La branche **master** représente une version fonctionnelle/stable du projet, vous ne devez pas la modifier, ni créer de branches à partir de master.

La branche **develop** représente l'état actuel du projet, c'est à partir de cette branche que vous créerez vos autres branches, sur la version du dépot principal de cette branche que vous effectuerez vos `pull request`, c'est elle aussi, que vous devrez garder à jour tout au long du développement.

Garder sa branche develop à jour

Pour garder sa branche develop à jour, il faut avant tout ajouter un `remote` vers le dépôt principal :

```
git remote add main https://github.com/Azurency/puit-canadien-web.git
```

Ensuite, il faut fetch et merge la branche develop du dépôt main vers la branche develop du dépôt local :

```
git pull main/develop develop
```

Les branches feature

Il est temps de commencer à développer, mais **attention** il ne faut jamais développer sur la branche develop *(et surtout pas master)* directement.
À la place, vous allez créer une nouvelle branche en **respectant scrupuleusement la convention de nomage** : `feature/le-nom-de-la-feature` par exemple `feature/page-aide`. Pour cela rien de plus simple, il vous suffit d'entrer la commande :

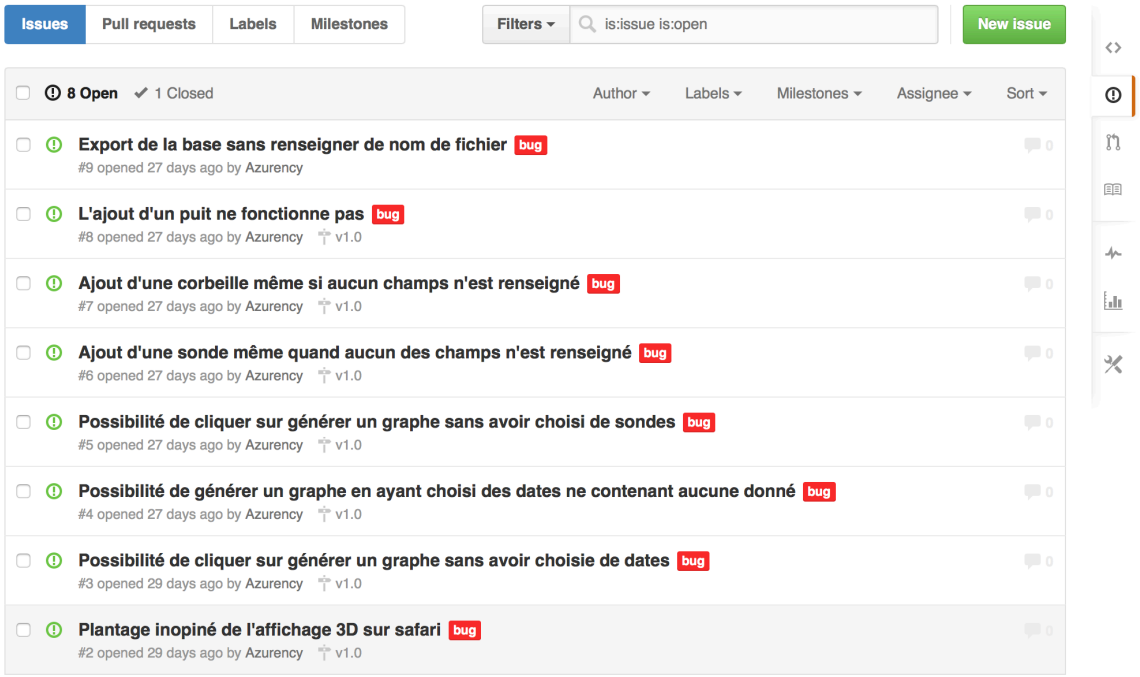
```
# si vous êtes sur la branche develop :
git checkout -b feature/le-nom-de-la-feature

# si vous n'êtes pas sur la branche develop :
git checkout -b feature/le-nom-de-la-feature develop
```

Vous pouvez alors travailler sur votre feature, et même créer d'autres branches feature en parallèle, collaborer avec d'autres membres en ajoutant des remotes…

Issues

Lors du développement, si vous rencontrez des bug ou si vous avez des idées d'amélioration, il existe le concept d'issues, les issues peuvent être directement référencées dans les messages de commit, c'est donc un très bon outil pour s'organiser.



Les issues sur github

Les issues peuvent être assignées à une personne, documentées et référencées grâce à leur numéro. Dans un message de commit l'utilisation de *"fixes", "fixed", "fix", "closes", "closed", ou "close", "resolve", "resolves", "resolved"* devant le numéro de l'issue "#123" permet de la fermer automatiquement dès que la branche est merge sur master.

Les issues sont ouvertes **depuis le dépôt principal** sur le site de github. Prenez l'habitude d'utiliser les issues, pour plus d'informations, un peu de lecture sur le site de github.

Pull request et contrôle de qualité

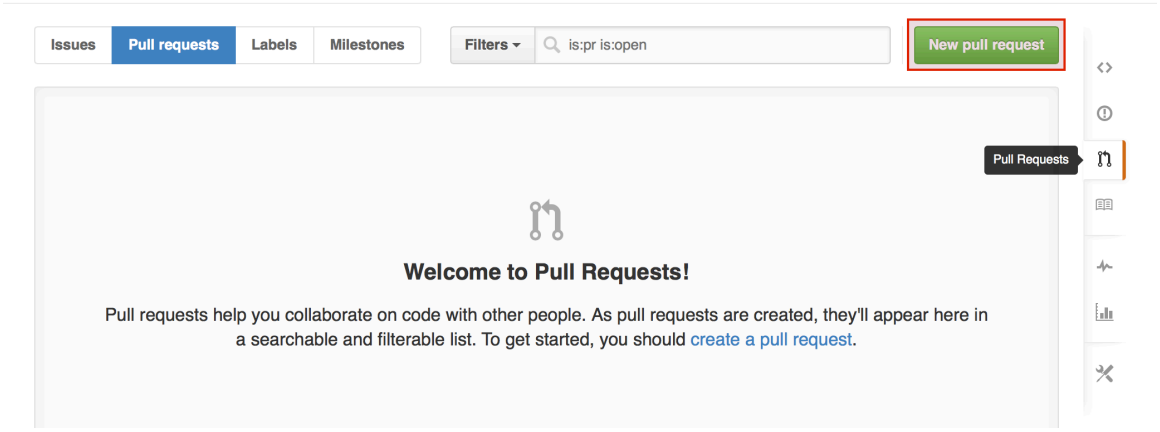
Une fois le développement de votre feature terminé, il va falloir merge cette branche sur la branche develop du dépôt principal.
C'est le rôle de l'intégrateur git d'effectuer un contrôle de qualité sur votre branche feature et d'accepter ou non le `merge` de celle-ci via un `pull request`.

Créer un pull request

Avant d'entamer la création d'un pull request, vous devez push votre branche feature vers votre dépôt distant :

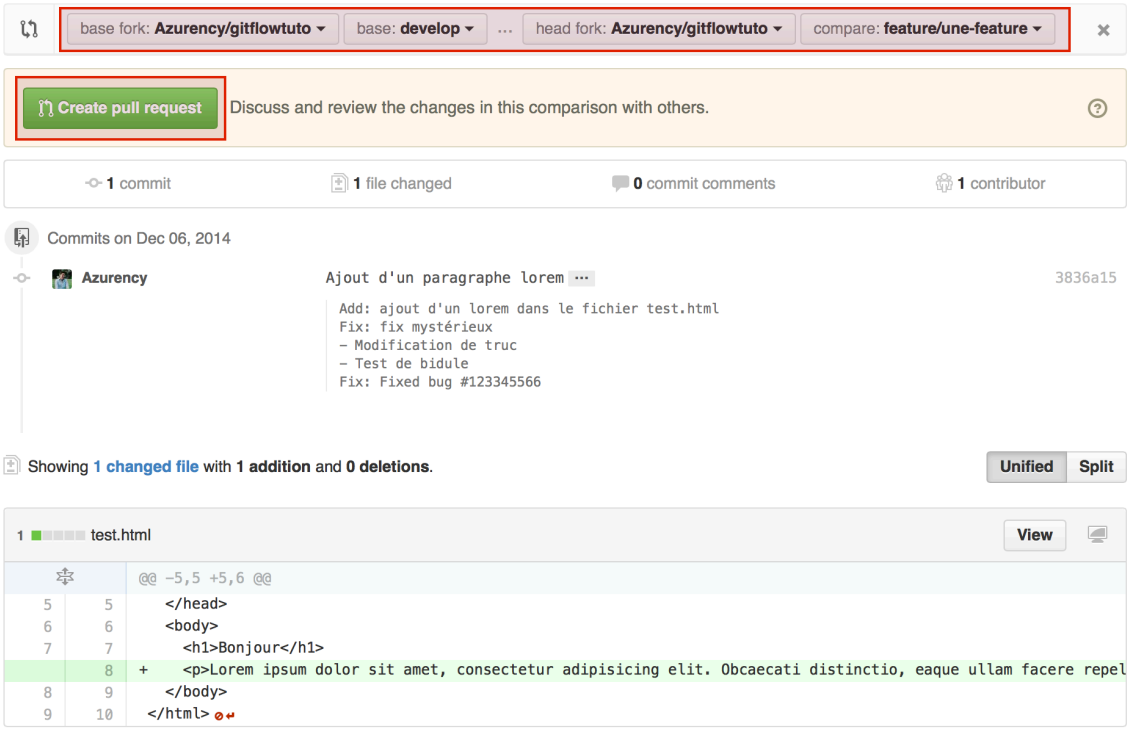
```
git push -u origin feature/le-nom-de-la-feature
```

La création d'un pull request se passe en ligne, directement sur le site de github, cliquer sur l'onglet *pull request* puis *New pull request* :



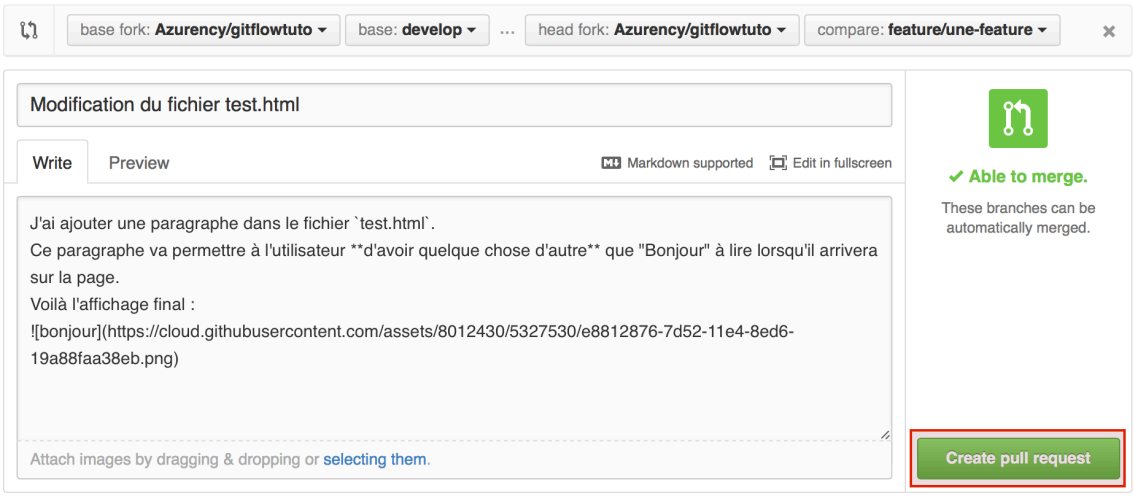
Créer un pull request

Ensuite, sélectionner le develop du dépôt principal et votre branche feature. Vous devriez apercevoir les changements que vous avez apportés, l'historique des commits, etc.



Selectionner les branches du pull request

Il faut ensuite que vous remplissiez le message du pull request, en expliquant clairement le but de votre ajout, ce que vous avez modifié, vous pouvez même illustrer votre message avec des captures d'écran.



Message du pull request

Une fois le pull request posté, l'intégrateur git va passer et contrôler votre code, s'il est satisfait par la qualité de votre travail, il peut décider d'accepter directement votre pull request. Il peut aussi vous demander de modifier certaines choses.

Rebase

Votre intégrateur git peut parfois vous demander de **rebase sur develop**, de **modifier** vos messages de commit, de **fusionner** des commits…

Sur develop

Imaginez le scénario suivant : vous avez commencé à travailler sur votre feature depuis la branche develop qui était à jour

```
      C---D---E feature/le-nom
     /
A---B develop
```

mais le développement continu sur le dépôt principal et plusieurs pulls request sont acceptées sur la branche develop, il arrive rapidement que vous vous retrouviez dans cette situation :

```
      C---D---E---F feature/le-nom
     /
A---B---G---H---I---J---K develop
```

L'intégrateur git n'acceptera que les branches feature qui sont à jour par rapport à la plus récente version de la branche develop. Pour lui faire agréablement plaisir et ne plus voir en commentaire de votre pull request un gentil *"tu peux rebase sur develop stp ?"*, mettez à jour

votre branche develop et utilisez la commande `rebase` :

```
# si vous êtes sur votre branche feature
git rebase develop

# si vous ne savez pas trop où vous êtes
git rebase develop feature/le-nom
```

Il se peut que vous rencontriez des conflits lors du rebase, résolvez les et utilisez :

```
git rebase --continue
```

Et si jamais vous vous rendez compte que vous avez fait une mauvaise manipulation, vous pouvez annuler le rebase en cours grâce à la commande :

```
git rebase --abort
```

Voilà, maintenant vous deviez avoir une branche bien en phase avec l'avancée de develop :

```
                          C---D---E---F feature/le-nom
                         /
A---B---G---H---I---J---K develop
```

Pour modifier son historique

Il est possible que l'intégrateur ne soit pas content de votre historique de commit, certains messages ne sont peut-être pas appropriés/complets, certains commit n'ont peut-être pas de sens d'être séparé et il faut les fusionner, etc.
Il faut alors utiliser la commande `rebase -i`

ATTENTION : avec `rebase -i` vous réécrivez l'histoire de votre git, c'est une commande avec laquelle il faut faire très attention !

Placez vous sur votre branche `feature/le-nom` et utilisez `git log --online` (ou `git lg` si vous avez lu le paragraphe bonus sur les alias) pour obtenir le hash des commits, par exemple :

```
078be40 (2014-12-07) - Ajout d'un autre lorem <Lassier Antoine>
1fb0501 (2014-12-07) - Ajout d'un paragraphe lorem <Lassier Antoine>
1bcad45 (2014-12-06) - Premier commit blabla <Lassier Antoine>
7d7a674 (2014-12-06) - Initial commit <Antoine Lassier>
```

L'intégrateur m'a dit que le message du commit `1bcad45` n'était pas approprié et que les commits `1fb0501` et `078be40` devraient être rassemblés en un seul commit. J'utilise donc la commande `git rebase -i` *à partir du commit* `7d7a674` :

```
git rebase -i 7d7a674
```

J'arrive alors sur :

```
pick 1bcad45 Premier commit blabla
pick 1fb0501 Ajout d'un paragraphe lorem
pick 078be40 Ajout d'un autre lorem

# Rebase 7d7a674..078be40 onto 7d7a674
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Je modifie :

```
# je modifie le message du commit 1bcad45
r 1bcad45 Premier commit blabla

# je squash le commit 078be40 avec le précédent
pick 1fb0501 Ajout d'un paragraphe lorem
s 078be40 Ajout d'un autre lorem
```

je sauvegarde, modifie les messages de commit et obtiens au final :

```
f386128 (2014-12-07) - Ajout d'un paragraphe lorem <Lassier Antoine>
52e576a (2014-12-06) - Ajout du fichier test.html <Lassier Antoine>
7d7a674 (2014-12-06) - Initial commit <Antoine Lassier>
```

J'aimerais maintenant push ma branche modifiée, malheureusement j'ai réécris mon historique git, je dois donc forcer le remplacement de la branche distante par ma branche locale :

```
git push --force
```

Maintenant la branche est push et le pull request est automatiquement mis à jour.

Finir sa feature

Une fois que votre branche `feature` est acceptée, vous pouvez la supprimer à la fois de votre dépôt distant et de votre dépôt local.
Première étape, revenir sur la branche develop :

```
git checkout develop
```

Ensuite il faut supprimer la branche :

```
# supprime la branche en local
git branch -d feature/le-nom

# supprime la branche sur le dépôt distant
git push origin --delete feature/le-nom
```

Autres commandes utiles

Voici un petit tableau des autres commandes utiles avec git :

Commande	Description
`git commit --amend`	Offre la possibilité de réécrire le dernier message de commit
`git stash`	Enregistre les modifications non commit pour y revenir plus tard
`git stash list`	Liste les stash
`git stash pop`	Applique le dernier stash pour retrouver ses modifications
`git reset #`	Restore l'état dans lequel était votre répertoire au niveau du commit numéro #
`git log --oneline`	Affiche l'historique des commits
`git checkout -t uneremote/feature`	Permet de créer et d'aller sur la branche feature de la remote uneremote

Bonus : les alias git

Les alias permettent de créer des raccourcis vers des commandes git pour aller plus vite. Pour ajouter des alias il faut utiliser la commande :

```
git config --global alias.nomDeAlias 'la commande git'
```

Voici quelques alias basiques :

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.st status
```

Désormais si vous tapez `git st` vous obtiendez le même résultat que `git status`

Je vous propose aussi un raccourcis pour `git commit --amend` et une commande `git lg` pour avoir un type `log --oneline` plus propre et comprenant plus d'informations

```
git config --global alias.ammend 'commit --amend'
git config --global alias.lg "log --pretty=format:'%Cred%h%Creset
%Cgreen(%ad)%Creset - %s %C(bold blue)<%an>%Creset' --abbrev-commit --
date=short"
```