Un livre de Wikilivres.

Git

Avant de commencer

Cette première étape est incontournable, nous allons voir comment installer et configurer Git sur votre machine. Suivez les instructions selon votre environnement de travail.

Il est à noter que l'architecture étant décentralisée, ces installations peuvent jouer le rôle du client ou du serveur, qui utilise le port 9418 mais passe en réseau par les services qui écoutent au port 22 (SSH), 80 (HTTP) ou 433 (HTTPS).

Installation

Linux

Sur la plupart des distributions, vous pouvez utiliser votre gestionnaire de paquet.

Vous pouvez aussi installer git depuis les sources (http://git-scm.com/book/en/Getting-Started-Installing-Git), si vous avez du temps à perdre.

Mac OS

Une installateur graphique est disponible sur Google (http://code.google.com/p/git-osx-installer). Sinon avec MacPorts (http://macports.org) lancer :

```
sudo port install git-core
```

Windows

Git pour Windows est téléchargeable sous forme de binaires précompilés sur msysGit (http://msysgit.github.io/) ou encore Git Bash (http://git-scm.com /downloads). Cela inclut l'utilitaire en lignes de commande, une interface graphique, et un client SSH.


Téléchargement de Git sur msysgit

De plus, le setup Cygwin le propose également.

Configuration minimaliste de l'environnement

Avant d'aller plus loin, il est indispensable de configurer Git a minima.

Dans Git, les contributeurs à un projet sont identifiés par leur nom et leur adresse courriel, il faut donc fournir à Git ces deux informations.

```
git config --global user.email "michel.boudran@fr.wikibooks.org"
git config --global user.name "Michel Boudran"
```

Nous verrons plus tard comment améliorer cette configuration minimale.

Obtenir de l'aide

Git gère très bien l'auto-complétion. Au fur et à mesure que vous saisissez vos commandes, utilisez la touche tabulation pour que git vous propose des options.

À tout moment vous pouvez consulter le manuel de git avec

```
git --help
```

Pour obtenir de l'aide sur une commande en particulier, utilisez (exemple pour la commande `branch`)

```
git branch --help
```

Lorsque vous lisez des documentations, vérifiez qu'elles s'appliquent bien à la version que vous utilisez :

```
git --version
```

Création de votre dépôt local

Maintenant que Git est installé, nous allons voir comment créer un dépôt sur notre machine. Un dépôt Git correspond à un projet de développement logiciel : chaque logiciel peut avoir un dépôt Git qui lui est réservé.

Nous allons voir trois cas d'utilisation différents :

1. La création d'un dépôt git pour démarrer un projet vierge
2. La création d'un dépôt git pour un projet existant, les fichiers se trouvant sur votre machine
3. La création d'un dépôt afin de travailler sur un projet existant qui est déjà dans un dépôt git distant, c'est le cas le plus courant.

Création d'un dépôt git pour démarrer un projet vierge

Rendez-vous dans le répertoire dans lequel vous souhaiter créer votre dépôt (dans notre exemple, nous avons utilisé le répertoire temporaire `cd /tmp`), puis

```
git init mon-projet
```

```
Initialized empty Git repository in /tmp/mon-projet/.git/
```

Vous pouvez ensuite vous placer dans le dossier "mon-projet" et travailler avec git.

Création d'un dépôt git avec une base de code existante

C'est tout aussi simple. Il faut d'abord se rendre dans le répertoire où se trouvent les sources et faire un `git init`

```
cd mon-projet
git init
```

```
Initialized empty Git repository in /tmp/mon-projet/.git/
```

Dès lors, vous êtes prêt à travailler avec git dans ce répertoire.

Création d'une copie locale d'un dépôt distant

Cette fois-ci, les sources ne sont pas sur notre machine mais sur un dépôt distant qui existe déjà. C'est le cas d'utilisation le plus typique, vous souhaitez rejoindre un projet pour développer des fonctionnalités, corriger des anomalies et publier vos modifications. Pour cela, vous aurez besoin de l'adresse du dépôt distant.

Contrairement à ce qu'on a vu plus haut, nous n'allons pas utiliser `init` mais `clone` en se plaçant dans le répertoire dans lequel on souhaite placer son dépôt.

Lorsque vous faites un `clone`, vous copiez l'intégralité du dépôt, il est donc normal que cet opération prenne longtemps pour les projets qui ont un long historique de contribution. Par exemple, pour le dépôt officiel du logiciel MediaWiki (`git clone https://git.wikimedia.org/git/mediawiki/core.git`), il faudra télécharger pas moins de 200 Mo.

Dans notre exemple (toujours en travaillant dans le répertoire temporaire `/tmp`), nous allons nous créer une copie locale d'un dépôt officiel qui représente un exemple d'extension MediaWiki :

```
git clone https://gerrit.wikimedia.org/r/p/test/mediawiki/extensions/examples.git
```

```
Cloning into 'examples'...
remote: Total 398 (delta 0), reused 398 (delta 0)
Receiving objects: 100% (398/398), 52.19 KiB | 0 bytes/s, done.
Resolving deltas: 100% (236/236), done.
Checking connectivity... done
```

```
cd examples
ls
```

```
chris_file
chris_pushed_this_file_without_review
ContentAction
ErrorPage
Example
FourFileTemplate
HelloWorld
Parser_function.i18n.magic.php
Parser_function.php
Parser_hook.php
Someone_was_here
SpecialIncludable.php
test1.php
Variable_hook.i18n.magic.php
Variable_hook.php
```

> **Attention !**
>
> On ne peut pas soumettre directement une version à un dépôt cloné. Il faut en passer par un *pull-request*.

Problèmes connus

fatal: unable to access 'https://MonServeur/MonDepotEnLigne.git/': SSL certificate problem: self signed certificate

Lors du clonage, remplacer HTTPS par HTTP, ou bien désactiver la vérification du certificat SSL :

```
git -c http.sslVerify=false clone https://MonServeur/MonDepotEnLigne.git
```

fatal: repository 'http://MonServeur/MonDepotEnLigne.git' not found

Si le dossier existe et est accessible en HTTP, mais que le `clone`, `pull` ou `push` ne le trouve pas :

- Vérifier que l'utilisateur a les droits d'écriture (pour Windows avec IIS, c'est *II_IUSRS*).
- Si le dépôt est sur le LAN, éviter HTTP : `git clone file:////MonServeur/c$/inetpub/wwwroot/MonDepotEnLigne.git`.

Visualiser le contenu de votre dépôt local

Avant de commencer à travailler vraiment sur le projet, nous allons déjà voir comment connaître l'état de notre dépôt.

Un dépôt est complexe, entre les commits, les branches, les tags, l'espace, les dépôts distants, le HEAD, le stash... Il est impératif de savoir comment, à tout moment, savoir où Git en est.

Dans ce chapitre, nous allons voir les différents outils qui sont à votre disposition pour comprendre dans quel état est votre dépôt local. Pour expérimenter les exemples que nous donnons, nous vous recommandons de travailler sur une copie d'un dépôt distant qui a déjà un historique fourni. En effet, ça sera bien plus représentatif qu'avec un dépôt vide où à peine créé.

Nous allons travailler suite à un

```
git clone https://gerrit.wikimedia.org/r/p/test/mediawiki/extensions/examples.git
```

git log

Git log est la principale commande à connaître. Commençons par son utilisation la plus simple

```
git log
```

```
commit a59a042e1a7f1474a11c0bd2585ab2eb71b85c47
Merge: 5c511f2 12c8449
Author: Darkdadaah <example@yahoo.fr>
Date:   Sat May 25 08:49:21 2013 +0000

    Merge "TEST 1"

commit 12c84495c13b7bbf10f7d85fd9d9d2cb129a7952
Author: jackpotte <example@gmail.com>
Date:   Sat May 25 10:31:05 2013 +0200

    TEST 1

    Hackathon test.

    Change-Id: I079fe9c5ef6eee1e2e12a653bbb5cec474d28ec0

commit 5c511f28b89ee2a9cc46707802668b01bf54dec0
Author: Darkdadaah <example@yahoo.fr>
Date:   Sat May 25 10:19:44 2013 +0200

    New file to "leave a trace"

    Change-Id: I5de6da00ad9712599e21ff630eee7d72d7177772

commit 27f4317fa4c325c98404e3cc8d443a164279b041
Author: Udit Saxena <example2@gmail.com>
Date:   Mon Apr 1 23:10:47 2013 +0530

    Second try. Now third.

    Change-Id: Ic7cddbf04394c08bfe5cd240bf48e3641e7feaa5
```

Par défaut, git utilise `less` pour vous permettre d'afficher les logs pages par pages. Ici, nous voyons 4 commits faits par 4 contributeurs différents. L'historique s'affiche dans l'ordre chronologique inversé : les modifications récentes sont au dessus des modifications anciennes.

Essayez d'ajouter de la couleur avec

```
git log --color
```

Vous pouvez voir l'historique des branches et des fusions avec

```
git log --graph
```

Vous pouvez voir l'historique en version réduite (une ligne par commit)

```
git log --oneline
```

Nous verrons plus tard comment utiliser log pour obtenir les informations qu'on souhaite. N'hésitez pas à combiner les options.

```
git log --oneline --graph --color
```

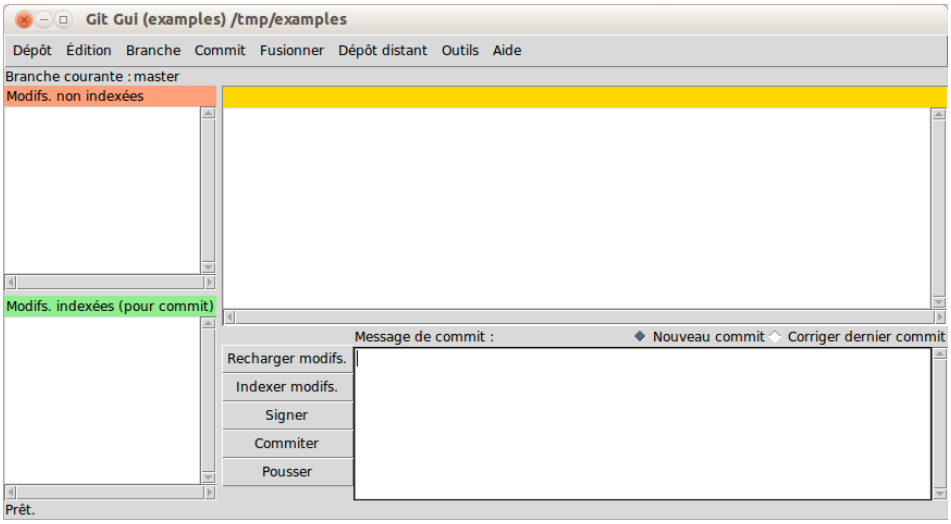
```
* 0eccb68 Increased version number
* fc5fdf8 Working on http://www.mediawiki.org/wiki/Git/Tutorial Doing an unimportant commit And an unimportant amendment Change-Id: I5a8a912234ea1dae6adae5a1371
*   62505b3 Merge "Word-wrapped and punctuated comments."
|\
| * 563788d Word-wrapped and punctuated comments.
* | 506432a So simple if you have a tutorial and someone who shows you how it works and what you shouldn't forget
|/
*   6609666 Merge "Implemented the getVersion function"
|\
| * b1f8deb Implemented the getVersion function
* |   70e59e5 Merge "Revert "Added a get version method""
|\ \
| * | 36af5f2 Revert "Added a get version method"
* | |   14b21cf Merge "Change README to split the lines in more logical places"
|\ \ \
| * | | 4090572 Change README to split the lines in more logical places
| | |/
| |/|
* | |   93ecbf0 Merge "removed a white space trailing the comment"
|\ \ \
| * | | 4d3d534 removed a white space trailing the comment
| |/ /
```

git gui

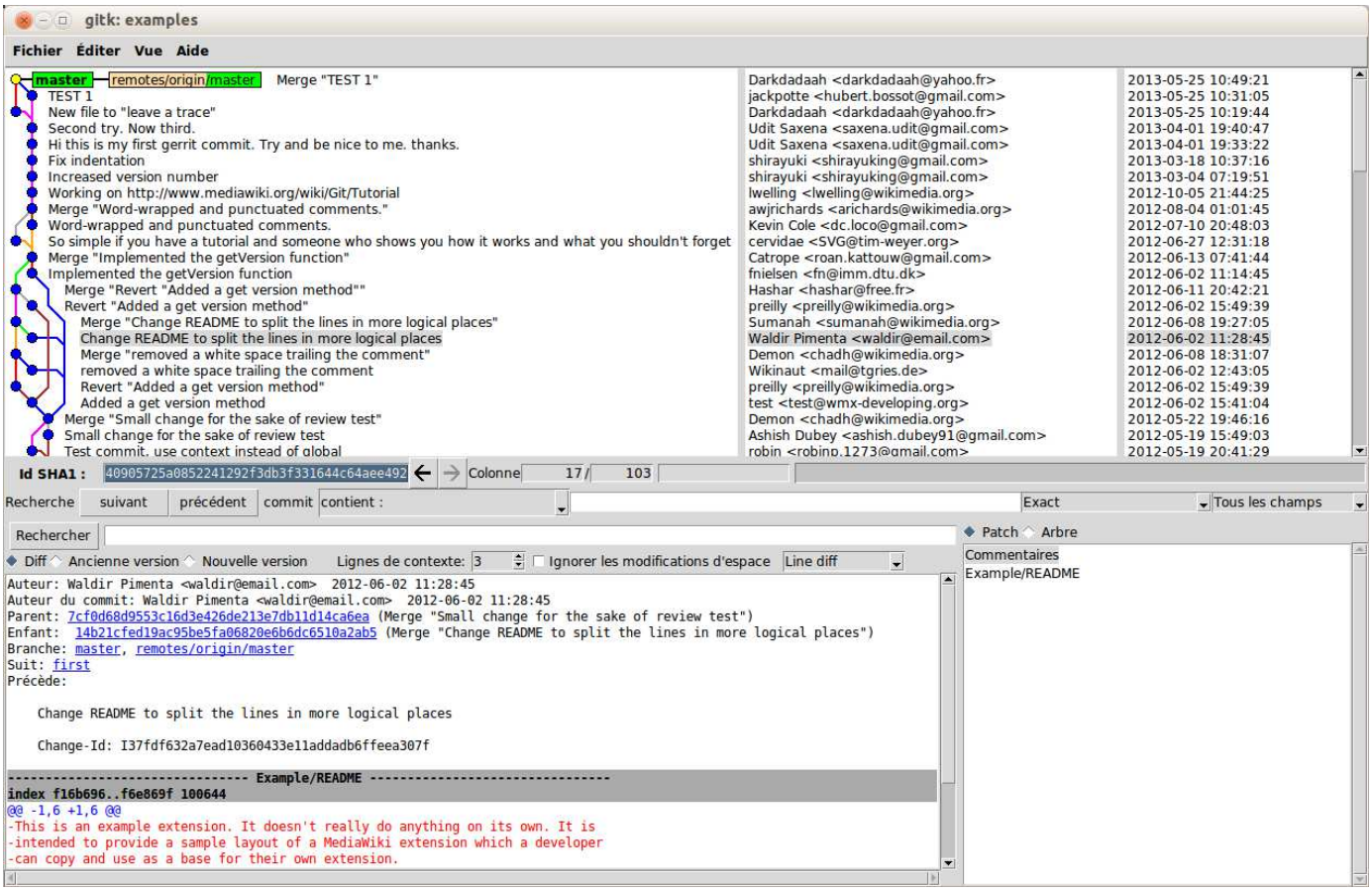
Il existe une extension git qui vous permet de visualiser votre dépôt local à l'aide d'un outil graphique. La plupart du temps, cette extension doit être installée en plus de git.

```
git gui
```

Une fenêtre s'ouvre, elle présente peu d'informations car nous n'avons pas encore de modifications à publier.



Ouvrez le menu « Dépôt » puis sélectionnez « Voir l'historique de toutes les branches ».



La fenêtre qui s'ouvre nous donne l'arborescence graphique qui nous permet de voir, dans l'ordre chronologique, toutes les modifications qui ont été faites.

gitweb

Gitweb (http://git-scm.com/book/fr/Git-sur-le-serveur-GitWeb) est l'interface web officielle intégrée dans git. Elle permet de visualiser le contenu d'un dépôt git depuis tout navigateur web.

```
git instaweb
```

Votre navigateur devrait s'ouvrir automatiquement à l'adresse http://127.0.0.1:1234

Premiers pas

Nous allons maintenant entrer dans le vif du sujet et faire une première modification dans le code source d'une application.

Cette première expérience va nous permettre de découvrir plusieurs notions importantes : l'espace de travail et l'index.

Préalable

On reprend l'exemple précédent.

```
git clone https://gerrit.wikimedia.org/r/p/test/mediawiki/extensions/examples.git
cd examples
```

Faisons d'abord un

```
git branch
```

qui va nous répondre

```
* master
```

Git nous indique qu'il existe une seule branche appelée `master` et que c'est sur cette branche que nous travaillons comme l'indique l'astérisque en face de `master`.

Cela nous est confirmé par

```
git status
```

qui nous répond

```
# On branch master
nothing to commit, working directory clean
```

Vous pouvez faire un

```
git log
```

Pour voir quel est l'auteur et la date de la dernière modification : cela nous servira de repère pour la suite.

Ajouter un fichier

Commençons par une modification simple : l'ajout d'un fichier. Cela peut être une première étape si vous avez créer un dépôt vide.

Par exemple, créons un fichier `mon_nouveau_fichier.txt` avec un petit texte dedans.

```
echo "Ceci est un test de git" > mon_nouveau_fichier.txt
```

Voyons la façon dont git perçoit ce nouveau fichier

```
git status
```

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

Il nous indique qu'on est toujours sur la branche `master`, qu'il y a un fichier `mon_nouveau_fichier.txt` mais qu'il n'est pas suivi (« untracked ») par git.

Comme nous voulons intégrer ce fichier au projet, on ne peut pas encore faire le `commit` cat `commit` n'envoit que les fichiers qui sont *tracked*, c'est à dire dans l'index (*staging*). Ajoutons le fichier, comme git nous le suggère, avec `add`

```
git add mon_nouveau_fichier.txt
```

On refait un

```
git status
```

Et, cette fois, git nous répond

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   mon_nouveau_fichier.txt
#
```

Le fichier `mon_nouveau_fichier.txt` sera bien intégré dans notre prochain `commit`. Allons-y :

```
git commit -m "mon premier commit"
```

Remarquons ici qu'avec `-m`, nous avons choisi de préciser le message de commit directement sur la ligne de commande. En lançant `git commit` tout court, l'éditeur de texte (`$EDITOR`) s'ouvre automatiquement pour inviter à saisir un commentaire de soumission.

```
[master 17eaa3e] mon premier commit
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 mon_nouveau_fichier.txt
```

Constatons immédiatement l'effet de ce commit :

```
git log
```

Notre dernier commit apparaît, en premier de la liste (c'est le plus récent).

```
commit 17eaa3e060b29d708a87867dcb725b7ec64ffaeb
Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
Date:   Tue Jul 22 22:00:39 2014 +0200

    mon premier commit
```

Avec

```
git log --graph
```

On voit clairement que notre commit est lié au commit précédent.

Modifier un fichier

Faisons une autre modification. Par exemple, modifions le fichier `mon_nouveau_fichier.txt` en ajoutant une ligne.

```
echo "Une seconde ligne pour un second test de git" >> mon_nouveau_fichier.txt
```

Voyons ce que git nous dit :

```
git status
```

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# 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:   mon_nouveau_fichier.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git nous indique bien que le fichier a été modifié, voyons le résumé de ces modifications telles qu'elles sont perçues par git :

```
git diff
```

```
diff --git a/mon_nouveau_fichier.txt b/mon_nouveau_fichier.txt
index a031263..762359c 100644
--- a/mon_nouveau_fichier.txt
+++ b/mon_nouveau_fichier.txt
@@ -1 +1,2 @@
 Ceci est un test de git
+Une seconde ligne pour un second test de git
```

Git nous montre la ligne qui a été ajoutée (le « + » en début de ligne).

On va maintenant faire le commit. Comme précédemment, il faut au ajouter le fichier au *staging* :

```
git add mon_nouveau_fichier.txt
```

> **Remarque :** si vous voulez ajouter au staging tous les changements qui ont été effectués (fichiers ajoutés, modifiés, supprimés), il vous suffit de faire[1]
>
> ```
> git add --all
> ```

```
git status
```

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   mon_nouveau_fichier.txt
#
```

```
git commit -m "ma première modification"
```

```
[master 5556307] ma première modification
 1 file changed, 1 insertion(+)
```

Remarquez le code « 5556307 » : il s'agit d'une abréviation de l'identifiant unique de l'objet Git (en l'occurrence une soumission). Chaque objet est haché en SHA-1. L'identifiant complet est en fait `5556307824d8d0425b38c9da696b84430e30f09f`, mais généralement les huit premiers caractères suffisent à l'identifier à coup sûr.

```
git log --graph
```

```
* commit 5556307824d8d0425b38c9da696b84430e30f09f
| Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| Date:   Tue Jul 22 22:18:08 2014 +0200
|
|     ma première modification
|
* commit 17eaa3e060b29d708a87867dcb725b7ec64ffaeb
| Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| Date:   Tue Jul 22 22:00:39 2014 +0200
|
|     mon premier commit
|
```

On voit bien que nos deux *commits* se succèdent.

Supprimer un fichier

```
git rm mon_nouveau_fichier.txt
git status
```

```
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    mon_nouveau_fichier.txt
#
```

Il n'est pas nécessaire de faire un add.

```
git commit -m "ma première suppression de fichier"
```

```
[master 77ea581] ma première suppression de fichier
 1 file changed, 2 deletions(-)
 delete mode 100644 mon_nouveau_fichier.txt
```

Regrouper des modifications

Il est possible de fusionner une soumission avec la dernière révision, via l'argument *amend*, sans avoir à réécrire leur résumé avec *no-edit* :

```
git commit --amend --no-edit
```

Par ailleurs, certains logiciels clients Git permettent de rassembler plusieurs révisions sélectionnées depuis une liste, comme la fonction *cherry-pick* de SmartGit, ou *git blame* pour afficher les auteurs de chaque passage.

Continuer

Vous maîtrisez désormais le strict minimum pour travailler avec git. Vous pouvez ajouter, modifier et supprimer des fichier et enregistrer les changements dans votre dépôt local ainsi que consulter l'historique des modifications. Cela reste toutefois une vision simpliste de la gestion de projet et nous verrons dans la suite comment exploiter les branches locales et comment partager votre travail avec d'autres contributeurs en publiant vos modifications sur un dépôt distant et en récupérant les modifications des autres contributeurs.

Problèmes connus

fatal: This operation must be run in a work tree

Lors d'un `git add` sur un dépôt initialisé avec `-- bare`, il faut soit uniquement lui soumettre ses modifications (sans possibilité de le cloner, avec `git remote add origin`[2]), soit définir un répertoire de branche avec `--work-tree`.

error: src refspec master does not match any

Il faut faire un `git add *` avec au moins un changement.

Références

1. http://git-scm.com/docs/git-add
2. https://git-scm.com/book/fr/v1/Git-sur-le-serveur-Mise-en-place-du-serveur

Branches

Précédemment, nous avons vu comment apporter des modifications à une branche telles que ajouter, modifier ou supprimer un fichier et `commit` nos modifications. Cela fonctionne parfaitement et cela peut suffire pour travailler seul sur un petit projet. Toutefois, ce n'est pas la meilleure façon de procéder sous git qui propose des mécanismes plus élaborés pour développer sur un projet.

L'approche de git est de favoriser l'utilisation de branche pour toute modification du code de l'application.

Ainsi, il ne faut jamais travailler directement sur la branche *master* : cette branche doit rester stable et ne doit être utilisée que pour baser son travail dans d'autres branches.

Pour mieux comprendre, nous allons refaire, pas à pas, exactement les même modifications que celles que nous avons faites précédemment mais, cette fois, nous allons utiliser une branche afin de nous familiariser avec ce concept.

Reprenons notre dépôt d'exemple :

```
git clone https://gerrit.wikimedia.org/r/p/test/mediawiki/extensions/examples.git
cd examples
```

Créer une première branche

D'abord, demandons à git de nous indiquer où nous en sommes au niveau des branches :

```
git branch
```

```
* master
```

Git nous indique qu'il existe une seule branche appelée `master` et que c'est sur cette branche que nous travaillons comme l'indique l'astérisque en face de `master`.

Créons une nouvelle branche que nous allons appelé `ma-branche`

```
git branch ma-branche
```

Constatons les effets

```
git branch ma-branche
```

```
  ma-branche
* master
```

Il y a maintenant deux branches : *master* et *ma-branche*. Actuellement, nous travaillons toujours sur *master* comme l'indique toujours l'astérisque.

```
git log --decorate --graph
```

On peut voir, sur la première ligne que *master* et *ma-branche* sont au même niveau, sur le même commit. Nous allons maintenant demander à git de nous basculer sur *ma-branche* afin de pouvoir travailler sur celle-ci et non sur *master*.

```
git checkout ma-branche
```

```
Switched to branch 'ma-branche'
```

On a basculé, et `git branch` nous le confirme.

```
git branch
```

```
* ma-branche
  master
```

Faire les modifications

On peut désormais faire les modifications dans **ma-branche** on peut développer, sans prendre le risque de modifier **master**.

Faisons les même modifications que précédemment :

```
echo "Ceci est un test de git" > mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'un fichier"
echo "Une seconde ligne" >> mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'une seconde ligne dans le fichier"
echo "Une troisième ligne" >> mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'une troisième ligne dans le fichier"
```

Et ainsi de suite. Vous pouvez commiter et faire autant de commit que vous voulez dans **ma-branche**.

L'idée est que pour chaque évolution du logiciel développé, il faut créer une branche. Ainsi, on peut garder la branche aussi longtemps que nécessaire et continuer de travailler dessus tant qu'on a pas fini la fonctionnalité.

Regardons le `log` que cela produit :

```
git log --decorate --graph
```

```
* commit 635ace69f901dfb1aaff187e6abc54b0c95fe51e (HEAD, ma-branche)
| Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| Date:   Tue Jul 22 23:33:15 2014 +0200
|
|     ajout d'une troisième ligne dans le fichier
|
```

```
* commit dbc6c57019afe80dbb2f3d889eb63cb024656faa
| Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| Date:   Tue Jul 22 23:33:14 2014 +0200
|
|     ajout d'une seconde ligne dans le fichier
|
* commit e2cbadc10289e74a131a728e06ac2421e79b5b9f
| Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| Date:   Tue Jul 22 23:33:14 2014 +0200
|
|     ajout d'un fichier
|
*   commit a59a042e1a7f1474a11c0bd2585ab2eb71b85c47 (origin/master, origin/HEAD, master)
|\  Merge: 5c511f2 12c8449
| | Author: Darkdadaah <darkdadaah@yahoo.fr>
| | Date:   Sat May 25 08:49:21 2013 +0000
| |
| |     Merge "TEST 1"
| |
```

Examinons ce graphique : **master** est en retard tandis que **ma-branche** est en avance de trois *commits*.

Fusionner la branche dans master

Supposons que nous sommes satisfaits du travail réalisé dans notre branche. Nous avons fait plusieurs *commits*, nous avons vérifié que nous n'avons pas créé de bogue, etc. Supposons que notre branche est prête et qu'on peut intégrer les modifications dans master.

D'abord, se placer sur master.

```
git checkout master
```

```
Switched to branch 'master'
```

Puis demander à git de fusionner la branche **ma-branche**, sans fast forward[1] pour éviter de perdre la topologie de la branche :

```
git merge ma-branche --no-ff -m "intégration de ma nouvelle fonctionnalité dans master"
```

Git va faire un commit pour intégrer les changements. Comme précédemment, nous avons choisi d'utiliser `-m` pour préciser le message de commit mais on aurait pu ne rien mettre et git nous aurait ouvert l'éditeur de texte.

```
Merge made by the 'recursive' strategy.
 mon_nouveau_fichier.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 mon_nouveau_fichier.txt
```

Examinons

```
git log --decorate --graph
```

```
*   commit abd3ef0a5978b90db042bf076e82d64c3576194b (HEAD, master)
|\  Merge: a59a042 635ace6
| | Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| | Date:   Tue Jul 22 23:43:51 2014 +0200
| |
| |     intégration de ma nouvelle fonctionnalité dans master
| |
| * commit 635ace69f901dfb1aaff187e6abc54b0c95fe51e (ma-branche)
| | Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| | Date:   Tue Jul 22 23:33:15 2014 +0200
| |
| |     ajout d'une troisième ligne dans le fichier
| |
| * commit dbc6c57019afe80dbb2f3d889eb63cb024656faa
| | Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
| | Date:   Tue Jul 22 23:33:14 2014 +0200
| |
| |     ajout d'une seconde ligne dans le fichier
| |
| * commit e2cbadc10289e74a131a728e06ac2421e79b5b9f
|/  Author: Michel Boudran <michel.boudran@fr.wikibooks.org>
|   Date:   Tue Jul 22 23:33:14 2014 +0200
|
|       ajout d'un fichier
|
*   commit a59a042e1a7f1474a11c0bd2585ab2eb71b85c47 (origin/master, origin/HEAD)
```

On retrouve bien quatre *commits* nous appartenant (symbolisés par une * dans le graphe). On retrouve les trois premières modifications et un quatrième *commit* pour le `merge`. On voit que les branches ont convergé sur le graphique et que **master** est de nouveau sur la première ligne tout en haut du graphe. Nos changements ont bien été intégré à **master**.

Effacer une branche

Nos changements sont intégrés à master, la branche est désormais inutile. Supprimons-la :

```
git branch -d ma-branche
```

```
Deleted branch ma-branche (was 635ace6).
```

Remarque : la suppression de la branche peut échouer si la branche à supprimer n'a pas été fusionnée dans *master* :

```
git branch -d ma-branche
```

```
error: The branch 'ma-branche' is not a strict subset of your current HEAD.
If you are sure you want to delete it, run 'git branch -D dev'.
```

Git se prémunit donc d'effacer des changements potentiellement non vérifiés. Comme git l'indique, on peut forcer la suppression malgré tout :

```
git branch -D ma-branche
```

Continuer

Vous pouvez à tout moment créer des nouvelles branches depuis **master** et ce, à chaque nouvelle fonctionnalité ou nouvelle modification qu'il faudrait apporter au projet. Git vous permet de gérer plusieurs branches en parallèle et ainsi de cloisonner vos travaux et d'éviter de mélanger des modifications du code source qui n'ont rien à voir entre elles.

En gardant une branche **master** saine, vous vous laissez la possibilité de créer de nouvelles branches simplement et vous conservez ainsi une version du logiciel prête à être livrée à tout instant (puisqu'on ne `merge` dedans que lorsque le développement est bien terminé.

`git log` vous permet de retrouver dans l'historique les branches qui ont été crées, et les différents *commits* réalisés pour une même fonctionnalité sont bien regroupés entre eux.

Problèmes connus

Une branche ne peut pas être supprimée si on a pas fait le dernier commit.

Références

1. http://stackoverflow.com/questions/9069061/what-is-the-difference-between-git-merge-and-git-merge-no-ff

Synchroniser le dépôt local avec le dépôt distant

Dans le chapitre précédent, nous avons vu comment vous pouviez travailler seul dans votre dépôt local. Nous allons maintenant nous pencher sur l'aspect **distribué** de git et voir comment travailler de façon collaborative en communiquant avec d'autres dépôts. Nous allons voir comment publier vos modifications et recevoir les modifications des autres développeurs.

Simulons un environnement de travail distribué

Nous allons faire travailler ensemble deux personnages, dont la réputation n'est plus à faire, Alice et Bob. Nous allons supposer que Alice et Bob vont chacun créer leur dépôt local sur leur machine avec git clone. Bien évidemment, ils vont utiliser *clone* en indiquant l'adresse du dépôt principal du projet : `http://`, `https://`, `git://` ou `file:////`.

Pour créer un dépôt HTTP(s), il faut qu'il soit lisible par un serveur web (ex : Apache).

Créons un faux dépôt distant pour nos tests

Il serait prématuré d'expliquer ici comment créer un dépôt sur le réseau. Aussi, pour travailler, nous allons créer un faux dépôt distant en local.

Placez-vous dans un dossier qui ne risque rien (par exemple `/tmp`), nous allons créer le faux dépôt distant.

```
mkdir tests-avec-git
cd tests-avec-git
git init faux-depot-distant --bare
```

```
Initialized empty Git repository in /tmp/tests-avec-git/faux-depot-distant/
```

> **Attention !**
>
> Les fichiers d'un dépôt `--bare` sont cryptés dans le sous-répertoire `objects`, ils ne sont donc pas accessibles par d'autres programmes que Git.

Simulons deux utilisateurs utilisant le dépôt distant

Le faux dépôt distant est créé. Maintenant, Alice et Bob vont créer leur copie locale avec clone.

```
git clone faux-depot-distant depot-local-alice
cd depot-local-alice
git config user.email "alice@fr.wikibooks.org"
git config user.name "Alice"

# Idem pour Bob
cd ..
git clone faux-depot-distant depot-local-bob
cd depot-local-bob
git config user.email "bob@fr.wikibooks.org"
git config user.name "Bob"
```

Alice commence à travailler

En tant qu'Alice, créons quelques modifications.

```
cd depot-local-alice
echo "Ceci est un test de git" > mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'un fichier"
echo "Une seconde ligne" >> mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'une seconde ligne dans le fichier"
echo "Une troisième ligne" >> mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'une troisième ligne dans le fichier"
```

Alice à maintenant quelques modifications dans son dépôt local, nous allons voir comment elle peut échanger avec Bob.

Commencer à travailler avec un dépôt distant

Dès que vous voulez faire une opération qui concerne le dépôt distant (publication ou récupération d'informations), commencez toujours par

```
git fetch
```

Cela met les informations sur les dépôts distants auxquels sont rattachés votre dépôt local, si vous oubliez de le faire, vous risquez d'être faussé par le fait que l'historique des modifications que vous voyez (par exemple) n'est pas complet.

Ainsi, dès que vous voulez faire une opération qui implique le dépôt distant, souvenez-vous de toujours faire un git fetch.

Publier les modifications locales sur le dépôt distant

Alice a fait plusieurs modifications sur *master*, elle voudrait les partager avec Bob, elle doit donc publier ses derniers *commits* sur le dépôt distant.

```
git push origin master
```

```
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 787 bytes | 0 bytes/s, done.
Total 9 (delta 1), reused 0 (delta 0)
To /tmp/tests-avec-git/faux-depot-distant
 * [new branch]      master -> master
```

Git nous indique que la branche master a été créée sur le dépôt distant.

Récupérer les modifications d'un dépôt distant

Voyons comment Bob peut récupérer le travail d'Alice.

```
cd depot-local-bob
git fetch
```

```
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 9 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
Depuis /tmp/tests-avec-git/faux-depot-distant
 * [nouvelle branche] master     -> origin/master
```

Git nous indique qu'une nouvelle branche a été créée sur le dépôt distant. On va essayer de la récupérer.

```
git checkout master
git log
```

```
git log
commit 99d23406a342a94dd8c7be9c21a47d6d11b8d7f0
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:04 2014 +0100

    ajout d'une troisième ligne dans le fichier

commit 659937374dd1612ea8f33c07173f45aa42cabce1
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'une seconde ligne dans le fichier

commit a3b17da118bf2cfda9e6bcb6f70d305566827373
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'un fichier
```

On a bien récupérer les modifications faites par Alice. Si vous ouvrez le fichier mon_nouveau_fichier.txt, vous retrouverez toutes les modifications d'Alice.

```
cat mon_nouveau_fichier.txt
```

```
Ceci est un test de git
Une seconde ligne
Une troisième ligne
```

À notre tour, faisons une modification :

```
echo "Une quatrième ligne" >> mon_nouveau_fichier.txt
git add mon_nouveau_fichier.txt
git commit -m "ajout d'une quatrième ligne dans le fichier"
git log
```

```
commit 6b99e801c2b37535a84fa6f73510b720f8aeeb31
Author: Bob <bob@fr.wikibooks.org>
Date:   Sat Nov 15 11:37:13 2014 +0100

    ajout d'une quatrième ligne dans le fichier

commit 99d23406a342a94dd8c7be9c21a47d6d11b8d7f0
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:04 2014 +0100

    ajout d'une troisième ligne dans le fichier

commit 659937374dd1612ea8f33c07173f45aa42cabce1
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'une seconde ligne dans le fichier

commit a3b17da118bf2cfda9e6bcb6f70d305566827373
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'un fichier
```

Le résultat de ce git log ne devrait pas vous surprendre. Néanmoins, cette sortie ne montre pas l'état du dépôt distant. Pour cela, nous allons utiliser l'option --decorate.

```
git log --decorate
```

```
commit 6b99e801c2b37535a84fa6f73510b720f8aeeb31 (HEAD, master)
Author: Bob <bob@fr.wikibooks.org>
Date:   Sat Nov 15 11:37:13 2014 +0100

    ajout d'une quatrième ligne dans le fichier

commit 99d23406a342a94dd8c7be9c21a47d6d11b8d7f0 (origin/master)
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:04 2014 +0100

    ajout d'une troisième ligne dans le fichier

commit 659937374dd1612ea8f33c07173f45aa42cabce1
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'une seconde ligne dans le fichier

commit a3b17da118bf2cfda9e6bcb6f70d305566827373
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'un fichier
```

```

```

Là, on voit bien que notre branche master locale (master) est en avance de un commit sur la branche master distante (origin/master). HEAD indique simplement le commit sur lequel nous nous trouvons. Cela est confirmé par git status :

```
git status
```

```
Sur la branche master
Votre branche est en avance sur 'origin/master' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
```

Publions nos modifications comme git nous le propose :

```
git push
```

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 341 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /tmp/tests-avec-git/faux-depot-distant
   99d2340..6b99e80  master -> master
```

On retourne chez Alice :

```
cd depot-local-alice
git status
```

```
Sur la branche master
Votre branche est à jour avec 'origin/master'.

rien à valider, la copie de travail est propre
```

La copie d'Alice semble à jour. Mais où est passé la modification de Bob ? Nous avons oublié le git fetch !

```
git fetch
git status
```

```
Sur la branche master
Votre branche est en retard sur 'origin/master' de 1 commit, et peut être mise à jour en avance rapide.
  (utilisez "git pull" pour mettre à jour votre branche locale)

rien à valider, la copie de travail est propre

```

Cette fois-ci, git nous indique que nous sommes en retard de 1 commit, en effet nous n'avons pas récupéré les modifications de Bob. Voyons ce que donne git log. Cette fois-ci, nous allons utiliser l'option --all pour indiquer à git que nous voulons voir toutes les branches, c'est à dire que nous voulons voir origin/master et pas seulement master.

```
git log --decorate --all
```

```
commit 6b99e801c2b37535a84fa6f73510b720f8aeeb31 (origin/master)
Author: Bob <bob@fr.wikibooks.org>
Date:   Sat Nov 15 11:37:13 2014 +0100

    ajout d'une quatrième ligne dans le fichier

commit 99d23406a342a94dd8c7be9c21a47d6d11b8d7f0 (HEAD, master)
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:04 2014 +0100

    ajout d'une troisième ligne dans le fichier

commit 659937374dd1612ea8f33c07173f45aa42cabce1
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'une seconde ligne dans le fichier

commit a3b17da118bf2cfda9e6bcb6f70d305566827373
Author: Alice <alice@fr.wikibooks.org>
Date:   Sat Nov 15 11:27:02 2014 +0100

    ajout d'un fichier
```

On voit bien notre retard sur le commit de Bob puisque origin/master est plus haut que master. Demandons à git de récupérer les modifications distantes et de les fusionner avec notre master local.

```
git pull
```

```
Mise à jour 99d2340..6b99e80
Fast-forward
 mon_nouveau_fichier.txt | 1 +
 1 file changed, 1 insertion(+)
```

Et ainsi de suite. Chacun peut localement faire plusieurs commits et faire régulièrement `git push` (« pousser » en anglais) pour publier ses propres modifications et des pull (« tirer » en anglais) pour récupérer les modifications des autres.

Continuer

Vous maîtrisez maintenant l'essentiel de git pour pouvoir travailler collaborativement sur un projet.

Évidemment, dans notre exemple, nous avons utilisé un faux dépôt distant. Dans la réalité, il faudra créer un dépôt central (régulièrement sauvegardé pour ne pas perdre le projet !) sur une machine réseau pour que tout le monde puisse faire le clone.

Problèmes connus

fatal: remote origin already exists

Pour redéfinir `origin` il faut le supprimer d'abord :

```
git remote rm origin
```

Étiquetage (tags)

Git permet d'ajouter des étiquettes (*tags* en anglais) spécifier des choses dans les historiques[1] :

```
$ git tag -a tag1
```

ou :

```
$ git tag -f tag1 HEAD
```

Pour lister les tags :

```
$ git tag
```

Envoyer les étiquettes sur le serveur distant :

```
$ git push --tags
```

Étiquettes vs branches

Les étiquettes comme les branches pointent vers une soumission, la différence est que la branche pointe toujours en haut de la ligne de développement et sont remplacées par les soumissions postérieures, alors que l'étiquette demeure inchangée.

Remise (stash)

La remise vous permet de mettre de côté temporairement des modifications que vous ne souhaitez pas encore publier dans un commit alors que vous voulez changez de branche.

C'est typiquement le cas quand vous êtes en plein développement sur une branche, vous avez fait des modifications et quelqu'un vient vous interrompre pour aller corriger un bug sur une autre branche. Vous allez remiser vos modifications, changer de branche, corriger le bug, revenir sur la branche où vous étiez et reprendre les modifications de la remise pour reprendre votre travail où vous en étiez.

Remiser les modifications

```
git stash
```

Reprendre les modifications remisées

```
git stash pop
```

Cumuler les modifications remisées

À chaque fois que vous appellez git stash, les modifications sont mises de côté dans une pile, au dessus des autres modifications remisées. À chaque fois que vous appelez pop, on dépile.

Afficher le contenu de la remise

```
git stash show
```

Pour avoir le détail (afficher le diff)

```
git stash show -p
```

Recombinaison

Recombinaison

Pour changer les messages des soumissions, leur ordre ou leur nombre :

```
$ git rebase -i HEAD~3
```

Le paramètre complété par *HEAD* peut aussi l'être par des noms de branches, et le *3* par n'importe quel autre nombre de soumission. On peut effacer et fusionner des soumissions avec "squash" ou changer leur ordre.

Pour déboguer :

```
$ git rebase -i --abort
```

Remarque : cette opération modifie les *commit-id* des soumissions affectées.

Sous-modules et Super-projets

Le **super-projet** est un concept apparu avec Git depuis v1.5.3, ayant pour but de mieux gérer de nombreux dépôts, en distinguant ceux qui sont hors du super-projet, de ceux à l'intérieur que l'on appelle les sous-modules.

Super-projets

Un super-projet est un dépôt Git, que l'on crée via `git init` dans le répertoire, puis `git submodule add` suivi des archives à inclure :

```
$ git submodule add ./examples
 Adding existing repo at 'examples' to the index
 warning: LF will be replaced by CRLF in .gitmodules.
 The file will have its original line endings in your working directory.
$
```

La structure résultante est de la forme suivante :

```
|- super-projet
  |- sous-module (archive Git) [a]
  |- sous-module [b]
  |- sous-module [c]
  |- sous-module [d]
```

Si quelqu'un récupère le super-projet, il trouvera une série de répertoire vide pour chaque sous-module. Pour les utiliser, il faut lancer `git submodule init` pour chacun.

Sous-modules

Une archive Git est considérée comme *sous-module* après avoir exécuté `git submodule add` dans un autre dépôt.

Workflow

Le flux de travail des super-projets et des sous-modules dit généralement adhérer à l'ordre :

1. Changement du sous-module.
2. `git commit` du sous-module
3. `git commit` du super-projet
4. `git submodule update` pour envoyer les changements aux différents dépôts antérieurs au super-projet.

Structure interne

Structure de Git brute

Le schéma suivant représente un dépôt Git v1.5.2.5[2].

```
.
└── .git/
    ├── HEAD
    ├── branches/
    ├── config
    ├── description
    ├── hooks/
    │   ├── applypatch-ms
    │   ├── commit-msg
    │   ├── post-commit
    │   ├── post-receive
    │   ├── post-update
    │   ├── pre-applypatc
    │   ├── pre-commit
    │   ├── pre-rebase
    │   └── update
    ├── info/
    │   └── exclude
    ├── objects/
    │   ├── info/
    │   └── pack/
    └── refs/
        ├── heads/
        └── tags/
```

Fichiers

HEAD

HEAD indique le code actuellement vérifié. Généralement le point de la branche sur lequel on travaille.

Il est possible d'ajouter un état "HEAD détaché", en dehors de la branche locale. Dans ce cas la tête pointe sur une soumission et non sur une branche.

config

Le fichier de configuration pour ce dépôt Git. Il peut contenir les paramètres permettant de gérer et stocker les données dans le dépôt local, les distants connus, et les informations sur les utilisateurs (local et autres).

description

Utilisé par les outils du navigateur de dépôt, contient une description du projet, généralement inchangée dans les dépôts non partagés.

Dossiers

Branches

hooks

Contient les scripts à lancer quand des évènements particuliers surviennent dans le dépôt Git.

Ces points d'entrée sont utilisé par exemple pour lancer des tests avant chaque soumission, filtrer le contenu uploadé, et implémenté ce genre de personnalisations.

info

objects

Stocke les listes de répertoires, fichiers et soumission.

Il y a les objets non compressés des nombreux répertoires, et les "packs" d'objets compressés. Les premiers sont régulièrement collectés via `git gc`.

refs

Contient les informations où les branches pointent. Inclut normalement des répertoires "heads" pour les branches locales, et "remotes" pour les copies des branches distantes. Toutes les branches ne figurent pas dans ces répertoires. Celles qui n'ont pas changé récemment sont listées dans le fichier .git/packed-refs.

Références

1. http://git-scm.com/book/fr/Les-bases-de-Git-%C3%89tiquetage
2. Généré avec la commande *tree* v1.5.1.1 : `tree -AnaF`.

pull-request

Principe

Une fois un dépôt distant cloné en local, il est facile de mettre régulièrement à jour sa version, à l'aide de la commande `git pull` depuis le répertoire du dépôt (via *crontab* par exemple).

Par contre pour envoyer ses versions développées localement sur le dépôt distant, cela nécessite un *pull request* (une demande de tirage), par email à l'administrateur du dépôt distant[1] :

```
git request-pull
```

Si la branche a été mise à jour depuis un autre client, git gère la fusion automatiquement si les fichiers modifiés sont différents. Par contre s'il y en a en commun, il faut procéder manuellement avec un :

```
git rebase -i origin/MaBranche1
```

Pour éviter cela, il faut bien vérifier que la branche sur laquelle on commence à travailler est bien la dernière version, avec :

```
git fetch origin/MaBranche1
```

Références

1. https://git-scm.com/book/fr/v1/Git-distribu%C3%A9-Contribution-%C3%A0-un-projet

- https://git-scm.com/docs/git-request-pull

Recettes

Supprimer un fichier du dépôt tout garder le fichier

`git rm fichier.txt` supprime le fichier du dépôt mais supprime aussi le fichier local.

Pour ne l'enlever que du dépôt, utiliser `git rm --cached`.

Annuler une soumission

Utiliser `git revert` avec *HEAD* pour désigner la dernière soumission effectuée :

```
$ git revert HEAD
Finished one revert.
[master 47e3b6c] Revert "Soumission 2"
 1 files changed, 0 insertions(+), 1 deletions(-)
$ ls -a
.  ..  fichier.txt  .git
```

Pour signifier d'autres soumissions que la dernière :

- `git revert HEAD^` : l'avant dernière.
- `git revert HEAD~5` : la cinquième moins récente.
- `git revert e6337879` : la soumission n°e6337879.

Nettoyer les changements non soumis

Pour effacer les changements en cours en rétablissement les états de la dernière soumission :

```
$ git reset --hard HEAD
```

ou

```
$ git reset --hard e6337879
```

Pour ne toucher qu'un seul fichier :

```
$ git checkout fichier.txt
```

Récupérer une version de fichier

Il faut d'abord récupérer l'identifiant de la version avec `git log` :

```
$ git log
commit 47e3b6cb6427f8ce0818f5d3a4b2e762b72dbd89
Author: NomUtilisateur <NomEmail@exemple.com>
Date:   Sat Mar 6 22:24:00 2010 -0400

    Revert "Soumission 2"

    This reverts commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518.

commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518
Author: NomUtilisateur <NomEmail@exemple.com>
Date:   Sat Mar 6 22:17:20 2010 -0400

    My second commit

commit be8bf6da4db2ea32c10c74c7d6f366be114d18f0
Author: NomUtilisateur <NomEmail@exemple.com>
Date:   Sat Mar 6 22:11:57 2010 -0400

    My first commit
```

Ensuite pour lire la version, utiliser `git show` :

```
$ git show e6337879cbb42a2ddfc1a1602ee785b4bfbde518:fichier.txt
Test Git Wikilivres.
test de suppression Git pour Wikilivres
```

Créer et appliquer un patch

Créer un patch génère un texte de toute la série des changements entre les branches origines et master.

```
$ git format-patch origin/master
```

Pour appliquer un patch :

```
$ git apply --stat  P1.txt  # affiche les stats des changements
$ git apply --check P1.txt  # vérifie les problèmes
$ git am < P1.txt           # applique le patch dans l'ordre
```

Exclure des fichiers du dépôt

Souvent il y a des fichiers dans l'espace de travail qui ne sont pas souhaitables dans le dépôt. Par exemple, `emacs` crée automatiquement une copie des fichiers éditer avec, avec un suffixe tilde, comme `fichier1~`. Il faut donc éviter manuellement de les soumettre.

Pour dire à Git d'ignorer certain fichiers, il est possible de créer un fichier `.gitignore`. Chaque ligne y représente une spécification (avec wildcards) des fichiers à ignorer. Des commentaires peuvent être ajoutés dedans sur les lignes débutant par un blanc ou un dièse :

```
# Ignorer les backups emacs :
*~

# Ignorer le répertoire ''cache'' :
app/cache
```

Écrire des messages de commit

Les bonnes pratiques sont[1] :

1. 50 caractères maximum pour le titre, résumant les changements.
2. Selon le contexte, la première ligne est traitée comme le sujet d'un email et le reste séparé par une ligne blanche, comme le corps du message.
3. Utiliser le présent des verbes.
4. Les listes à puces sont autorisées, typiquement avec un moins ou une astérisque.
5. Le corps du message doit comprendre des lignes de 72 caractères maximum pour plusieurs raisons :
   - `git format-patch --stdout` convertit une série de soumission en une série d'emails.
   - Le log Git ne revient pas automatiquement à la ligne, sans retour chariot il est donc étalé sur une seule ligne donc difficile à lire. Le nombre 72 est le résultat du calcul des 80 du terminal (selon la nétiquette des mails), moins les 4 de l'indentation et les 4 de sa symétrie à droite.

Les utilisateurs de Vim peuvent rencontrer ce prérequis lors de l'installation de vim-git, ou bien en le définissant dans la configuration des messages de soumission Git :

```
:set textwidth=72
```

Ceux de TextMate peuvent ajuster l'option colonne "Wrap" du menu "view", puis utiliser ^Q pour revenir à la ligne des paragraphes (s'assurer qu'il y a une ligne blanche après pour éviter le mélange avec les commentaires). Voici une commande shell pour ajouter 72 au menu afin de ne pas avoir à le faire glisser à chaque fois :

```
$ defaults write com.macromates.textmate OakWrapColumns '( 40, 72, 78 )'
```

Ces résumés peuvent ensuite être lus via :

- `git log --pretty=oneline` affiche une courte liste des historiques avec identifiants et résumés.
- `git rebase --interactive` fournit les résumés de ses propres soumissions.
- Si l'option de configuration `merge.summary` est définie, les sommaires de toutes les soumissions seront fusionnés.
- `git shortlog` n'affiche que les résumés des précédentes soumissions.
- `git format-patch`, `git send-email`.
- `git reflog`, un historique local est accessible pour aider à retrouver d'éventuels erreurs.
- `gitk`, une inerface graphique qui a une colonne résumé.
- Gitweb et d'autres interfaces web comme GitHub (http://github.com) ou Bitbucket (https://bitbucket.org) affichent également les résumés.

La distinction sujet/corps de texte de Git permet donc un confort de recherche d'historique par rapport à d'autres logiciels similaires comme Subversion.

Références

1. http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

Améliorer sa productivité en configurant Git

Dans ce chapitre, nous allons voir comment vous pouvez adapter git à vos propres besoins.

Apprendre à configurer git

Vous pouvez intervenir à trois niveaux :

Au niveau system
: dans ce cas, la configuration s'appliquera à tous les utilisateurs de votre système.

Au niveau global
: dans ce cas, la configuration sera appliquée à vous seul utilisateur et pour tous les dépôts. C'est l'option la plus courante.

Au niveau local
: dans ce cas, la configuration sera appliquée uniquement à ce dépôt.

Vous avez deux possibilités :

Travailler avec la commande `git config`
: Vous devrez donc utiliser respectivement `--system`, `--global` ou ne pas mettre d'argument (git appliquera la commande au dépôt).

Modifier le fichier de configuration de git (avec un éditeur de texte)
: Vous devrez donc modifier respectivement les fichiers `/etc/gitconfig`, `~/.gitconfig` ou le fichier `.git/config` qui se trouve dans le dépôt.

Dans les exemples qui suivent, nous travaillerons sur le niveau **global** car c'est ce que le développeur souhaite la plupart du temps.

Pour voir, à tout moment, votre configuration :

```
git config --list
```

Activer la coloration de la sortie par défaut

Par défaut, git doit colorer la sortie de vos commandes sur le terminal. Si ce n'est pas le cas, vous utilisez une version trop ancienne de git.

Vous pouvez toutefois configurer git pour forcer ce comportement par défaut.

```
git config --global color.ui true
```

Créer des alias pour vos commandes les plus courantes

```
[alias]
        lg = log --graph --all --decorate
```

Éviter de retaper son mot de passe

Git propose un mécanisme pour stocker temporairement votre mot de passe en mémoire et ainsi vous éviter d'avoir à le retaper à chaque push, pull ou toute opération impliquant un repo distant.

```
git config --global credential.helper cache
```

Linux

Nous allons maintenant voir comment, lorsque vous travaillez sous Linux, vous pouvez travailler au mieux avec Git et augmenter votre productivité.

Installer un prompt git

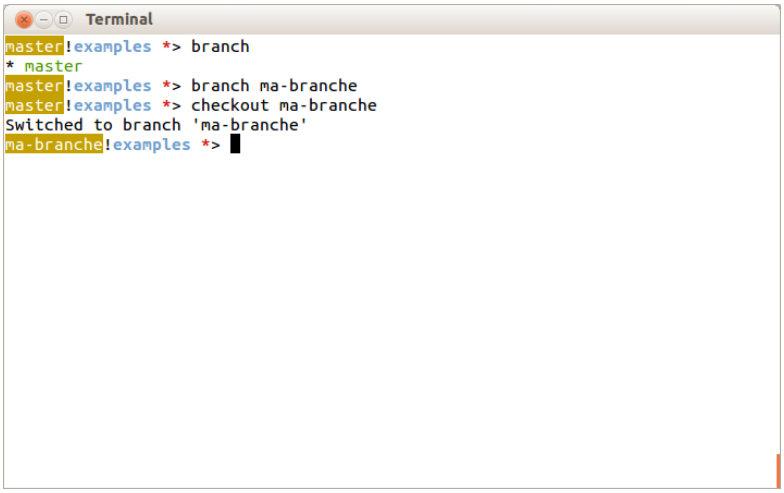
Si vous travaillez sur un projet plusieurs heures d'affilé, vous avez probablement une console qui reste ouverte en permanence pour *commiter* régulièrement. Autant éviter de taper « git » à chaque fois, et de passer son temps à faire des « git branch » pour savoir sur quelle branche on travaille.

Pour gagner du temps, vous pouvez installer un prompt git, et pour commencer à travailler, taper

```
git sh
```

Cela nous ouvre un prompt git. Désormais, l'invite de commande vous indique le répertoire dans lequel vous vous trouvez mais aussi la branche courante. Toutes les commandes sont automatiquement préfixées par "git", vous pouvez taper directement la commande git à appliquer.

Ctrl+D vous permet de quitter le prompt à tout moment.



Ce shell intègre également beaucoup de raccourcis, consulter `git sh --help` pour avoir la liste. Quelques exemples :

```
# L'espace de travail
a       # git add
aa      # git add --update (mnémonique « add all »)
stage   # git add
ap      # git add --patch
p       # git diff --cached (mnémonique « patch »)
ps      # git diff --cached --stat (mnémonique « patch stat »)
unstage # git reset HEAD

# Commits et historique
ci    # git commit --verbose
ca    # git commit --verbose --all
amend # git commit --verbose --amend
n     # git commit --verbose --amend
k     # git cherry-pick
re    # git rebase --interactive
pop   # git reset --soft HEAD^
peek  # git log -p --max-count=1

# Dépôt distant
f # git fetch
pm # git pull (mnemonic: « pull merge »)
pr # git pull --rebase (mnémonique « pull rebase »)

# Divers
d     # git diff
ds    # git diff --stat (mnémonique « diff stat »)
hard  # git reset --hard
soft  # git reset --soft
scrap # git checkout HEAD
```

> **Attention !**
>
> Si vous tapez dans commande système telles que `rm` (pour supprimer un fichier) ou `reset` (pour purger l'affichage dans le terminal), ce sera `git rm` et `git reset` qui seront appelées ! Ce n'est pas ce que vous voulez.

Windows

Configuration

Après installation, trois raccourcis sont accessibles dans le menu démarrer :

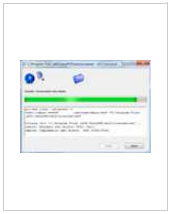
- Git Bash : langage Unix.
- Git CMD : langage DOS.
- Git GUI : interface graphique.

Pour que Git se positionne par défaut dans le répertoire de votre choix à chaque ouverture, sous Windows faire un clic droit sur son raccourci, puis modifier le chemin du champ "démarrer dans".

TortoiseGit

TortoiseGit est un client pour Git qui permet de gérer ses dépôts depuis l'explorateur Windows.


Options
(clic droit d'un dossier)


Clone


Commit


Push

> **Remarque :** les fenêtres de *commit* et *push* peuvent mettre plusieurs minutes à s'actualiser (F5 pour rafraichir), selon la taille du dépôt. Par exemple pour 7 000 fichiers pesant au total 3 Go, il faut s'armer de patience après chaque opération (la solution semble peu adaptée).

La vue des synchronisations permet de consulter la liste des fichiers modifiés (*Out ChangeList*) avec le différentiel dans chacun si on double-clique dessus (dans une fenêtre TortoiseGitMerge par défaut).

Elle permet aussi d'ouvrir l'option *Settings* en cliquant sur *Manage*. Pour enregistrer une connexion dans ces paramètres, cliquer sur le sous-menu de *Git* appelé *Remote* (s'il n'apparait pas, sortir et sélectionner un répertoire avant d'y retourner). L'URL du dépôt peut être de la forme :

- ssh://depot@depot.example.com/home/depot/. Mot de passe à entrer chaque connexion, ou utilisation d'une clé SSH.
- http://depot.example.com/home/depot/. Mot de passe à entrer chaque connexion, ou configuration de *netrc*[1] *over HTTP* avec cURL[2].

En cas d'erreurs SSH, se reporter au wikilivre *Le système d'exploitation GNU-Linux/Le serveur de shell distant SSH#Problèmes connus*.

> **Attention !**
>
> Une fois installé, le processus TortoiseGit se lance à chaque démarrage et est susceptible de bloquer la suppression de fichiers par l'explorateur.

Références

1. http://www.mavetju.org/unix/netrc.php
2. https://curl.haxx.se/docs/manual.html

Passer de Subversion à Git

Si vous maîtrisez subversion, vous aller sûrement être perturbé dans votre passage à git. En effet, l'écart entre gestion de version centralisée et gestion de version décentralisée est important et ces deux outils ne s'utilisent pas du tout de la même façon même s'il y a des similitudes.

Ce chapitre s'adresse aux personnes qui utilisent subversion et il vise à lever les ambiguité et les confusions qui surgissent quand on découvre git.

Quelques confusions habituelles

Les tags ne sont pas des branches
: dans subversion, on crée un tag par copie du *trunk* dans un nouveau dossier qui porte le nom du tag. On recréée ainsi toute l'arborescence du trunk dans un dossier (en fait, une branche puisque c'est une dérivation du *tronc*) du dépôt. Dans git, les tags ne sont pas de branches. Un tag désigne simplement un *commit* précis du dépôt.

L'opération *commit* n'envoie aucune information vers le dépôt distant
: Dans subversion, *commit* envoie toutes les modifications réalisées sur votre copie locale vers le dépôt distant. Dans git, *commit* enregistre les modifications dans votre dépôt local.

Équivalences entre les commandes git et les commandes subversion


Cette section est vide, pas assez détaillée ou incomplète.

Ressources externes

- svn crash course (https://git.wiki.kernel.org/index.php/GitSvnCrashCourse)

Travailler avec Git local et un dépôt Subversion distant

Git permet de participer à de nombreux autres systèmes de contrôle de version, comme `git-svn` ou `git-cvsimport`.

SVN

La compatibilité entre Git et Subversion est assurée par `git-svn` qui autorise un utilisateur à accéder et participer à un dépôt SVN. Les utilisateurs peuvent générer des patchs locaux pour les envoyer par liste de diffusion, ou soumettre leurs changements aux dépôts d'origine.

Premiers pas

Pour commencer à utiliser Git avec des projets sur des serveurs Subversion, il faut créer un dépôt local, et configurer `git-svn` :

```
mkdir projet1
cd projet1
git svn init <URL du dépôt root> -T/chemin/du/tronc
git svn fetch -r <première révision>:HEAD
```

Le paramètre "première révision" peut être "1", mais pour gagner du temps il est possible de ne prendre que les 10 dernières révisions. `svn info` indique alors ces révisions.

Généralement quand on travaille avec des dépôts Subversion, on communique l'URL du projet complète. Pour déterminer l'URL du dépôt racine :

```
git svn info <URL du dépôt root>
```

Une ligne du résultat indique le dépôt racine. Le chemin du tronc est simplement le reste de l'URL qui suit.

Il est possible de simplement donner à `git-svn` l'URL complète du projet, mais cela peut stopper la possibilité de travailler sur des branches SVN.

Exemples

Obtenir Pywikipedia :

```
$ git svn init http://svn.wikimedia.org/svnroot/pywikipedia/trunk/pywikipedia/
Initialized empty Git repository in .../.git/
$ git svn fetch -r 1:HEAD
...
r370 = 318fb412e5d1f1136a92d079f3607ac23bde2c34 (refs/remotes/git-svn)
        D       treelang_all.py
        D       treelang.py
W: -empty_dir: treelang.py
W: -empty_dir: treelang_all.py
r371 = e8477f292b077f023e4cebad843e0d36d3765db8 (refs/remotes/git-svn)
        D       parsepopular.py
W: -empty_dir: parsepopular.py
r372 = 8803111b0411243af419868388fc8c7398e8ab9d (refs/remotes/git-svn)
        D       getlang.py
W: -empty_dir: getlang.py
r373 = ad935dd0472db28379809f150fcf53678630076c (refs/remotes/git-svn)
        A       splitwarning.py
...
```

Récupérer AWB (AutoWikiBrowser) :

```
$ git svn init svn://svn.code.sf.net/p/autowikibrowser/code/
Initialized empty Git repository in .../.git/
$ git svn fetch -r 1:HEAD
...
r15 = 086d4ff454a9ddfac92edb4013ec845f65e14ace (refs/remotes/git-svn)
        M       AWB/AWB/Main.cs
        M       AWB/WikiFunctions/WebControl.cs
r16 = 14f49de6b3c984bb8a87900e8be42a6576902a06 (refs/remotes/git-svn)
        M       AWB/AWB/ExitQuestion.Designer.cs
        M       AWB/WikiFunctions/GetLists.cs
        M       AWB/WikiFunctions/Tools.cs
r17 = 8b58f6e5b21c91f0819bea9bc9a8110c2cab540d (refs/remotes/git-svn)
        M       AWB/AWB/Main.Designer.cs
        M       AWB/AWB/Main.cs
        M       AWB/WikiFunctions/GetLists.cs
r18 = 51683925cedb8effb274fadd2417cc9b1f860e3c (refs/remotes/git-svn)
        M       AWB/AWB/specialFilter.Designer.cs
        M       AWB/AWB/specialFilter.cs
r19 = 712edb32a20d6d2ab4066acf056f14daa67a9d4b (refs/remotes/git-svn)
        M       AWB/WikiFunctions/WPEditor.cs
r20 = 3116588b52a8e27e1dc72d25b1981d181d6ba203 (refs/remotes/git-svn)
...
```

> **Attention !**
>
> Cette opération de téléchargement peut prendre une heure.

Interagir avec le dépôt

L'avantage de travailler avec Git sur des dépôts SVN est l'utilisation en local. Dans ce cas :

1. Ne pas lancer `git pull`
2. Dans une branche mieux vaut éviter de lancer `git-svn dcommit` car les soumissions fusionnées ont tendance à embrouiller `git-svn`. Par contre, combiner les changements avec ceux de Subversion en amont est équivalent à `svn update` :

```
git stash     # cache les changements pour obtenir un arbre propre
git svn fetch # amène les derniers changements
git rebase trunk
git stash apply
```

La première et la dernière ligne ne sont pas nécessaires si l'arbre est propre.

Le `git rebase trunk` laisse les soumissions locales au dessus du HEAD SVN[1].

Changements locaux

Pour éviter de propager des modifications locales indésirables (débogages, tests...), avec `git svn dcommit`, sans les perdre peut passer par deux approches.

Premièrement, maintenir une branche locale pour chacune qui devra contenir des changements locaux. Par exemple faire un `rebase` sur "branche1" au-dessus de "branche1-locale". Exemple :

```
git rebase trunk branche1-locale
git rebase branche1-locale branche1
```

Deux choix sont ensuite possibles, effectuer directement les changements sur la branche locale, ce qui est plus rapide que de les soumettre à la branche distante avant de les récupérer dans la locale. Ensuite il est possible d'utiliser `git reset`[2] pour les retirer de la branche distante.

Comme une alternative à l'approche centrée recombinaison, il existe une méthode basée sur la fusion. Tout en conservant les changements sur une branche locale, mais sans avoir à conserver la branche au dessus de la branche locale par recombinaison.

C'est un avantage car :

1. Sinon il y a plus à écrire [précision nécessaire].
2. Historiquement, la recombinaison a souvent demandé de résoudre le même conflit deux fois, s'il survient pendant la première recombinaison.

Donc à la place des recombinaisons, on crée une nouvelle branche servant à la construction. Il faut la démarrer avec la soumission à tester. Ensuite `git merge` fusionne la branche locale, apportant tous les changements dans un seul arbre. La raison de cette fusion dans une branche reconstruction est pour dissuader l'utilisation de `git-svn dcommit` (qui soumettrait les tests indésirables sur le serveur).

Cette approche peut même rendre facultative la recombinaison quotidienne la branche avec le tronc. En cas de branches multiples, les recombinaisons permanentes peuvent s'avérer chronophages :

```
git checkout build
git reset --hard trunk              # s'assurer de l'absence de changement important
git merge branche1 branche1-locale
```

La construction contient ensuite les changements du tronc, branche1 et branche1-locale !

Il est possible de conserver plusieurs branches locales dont une avec les tests. Cette approche peut être développée avec une branche par sujet dans un arbre :

```
git merge sujet1 sujet2 config debug...
```

Malheureusement, la fusion de cette pieuvre ne résout pas les conflits. Dans ce cas il faut appliquer les fusions une par une :

```
git merge sujet1
git merge sujet1
git merge local
...
```

Envoyer des changements en amont

Éventuellement, pour soumettre au serveur des branches sujet avec un accès `commit`, on peut lancer `git-svn dcommit`. Cela prendra chaque soumission locale dans la branche courante et le soumettra à subversion. Par exemple avec trois soumissions locales, après `dcommit` il y aura trois nouvelles soumissions dans subversion.

Sans accès `commit`, le patch devra probablement être soumis via une liste de diffusion ou un logiciel de suivi de problèmes (bug tracker). Pour cela on peut utiliser git-format-patch (http://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html). Par exemple pour trois soumissions locales :

```
git format-patch HEAD~3..
```

Le résultat sera trois fichiers en $PWD, 0001-commit-name.patch, 0002-commit-name.patch, et 0003-commit-name.patch, qui pourront être attachés à des emails ou joint à Bugzilla. Remarque : il existe git-send-email (http://www.kernel.org/pub/software/scm/git/docs/git-send-email.html) pour envoyer les emails directement :

```
git send-email *.patch
```

Si les séries de patchs ne sont pas dans l'ordre, voir `git rebase -i`.

Références

1. git-rebase (http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html)
2. http://www.kernel.org/pub/software/scm/git/docs/git-reset.html

Participer au développement de Wikimédia

Prérequis

Les soumissions effectuées par Git doivent être authentifiées par Gerrit. Pour ce faire il faut ajouter une clé publique dans son compte https://gerrit.wikimedia.org/r/#/settings/ssh-keys (nommé *login* dans les exemples ci-dessous).

```
apt-get remove git-review
pip install git-review
```

Site Mediawiki

Le dépôt *examples.git* existe pour s'entraîner.

```
git review -s
git branch
git remote -v
ssh login@gerrit.wikimedia.org:29418/test/mediawiki/extensions/examples.git
git review -s
git config -l
git config --global user.name "login"
git clone https://gerrit.wikimedia.org/r/p/test/mediawiki/extensions/examples.git
git review -sgit pull origin master
git pull origin master
git checkout -b branche-1 master
git diff
git status
git add test1.php
git status
git diff --cached
git commit
git pull origin master
git rebase master
git review -R
cd .git
git fetch https://gerrit.wikimedia.org/r/mediawiki/core refs/changes/69/17069/1 && git checkout FETCH_HEAD
```

Le fichier *test1.php* est maintenant présent sur le dépôt de la fondation.

Module quiz

```
cd Git
git clone ssh://login@gerrit.wikimedia.org:29418/mediawiki/extensions/Quiz
cd Quiz
vim Quiz.class.php
git add Quiz.class.php
git commit
git fetch
git push ssh://login@gerrit.wikimedia.org:29418/mediawiki/extensions/Quiz HEAD:refs/for/master
# Error with a change ID
git commit --amend
# Insertion of the change ID at the last line
git push ssh://login@gerrit.wikimedia.org:29418/mediawiki/extensions/Quiz HEAD:refs/for/master
```

Débogage

failed to create a new commit

Reconfigurer :

```
git config --global user.name "Your Name"
```

failed to sync this branch because due to unmerged files

Le dépôt distant possède certains fichiers qui sont plus à jour que le local, et vice-versa.

Si le message provient d'une interface graphique, essayer de la fermer et de lancer la synchro en shell, avec `git push`. Cela peut donner un message plus précis, ex :

```
remote: error: By default, updating the current branch in a non-bare repository is denied, because it will make the index and work tree inconsistent
```

Dans ce cas, on peut modifier le fichier "config" du serveur distant en ajoutant "bare=true".

Sinon, il faut créer une nouvelle branche pour faire un "pull request".

Sinon, copier les fichiers locaux dans un autre dossier (parent), et recloner avant de les replacer.

fatal: index-pack failed

`git pull` n'a pas été lancé depuis le répertoire du dépôt.

Please, commit your changes or stash them before you can merge

Précéder le `pull` d'un `stash` :

```
git stash
git pull
```

Your branch is behind 'xxx' by y commits, and can be fast-forwarded

Dans ce cas il peut être préférable d'effacer la veille branche du serveur pour ne garder que la locale, qui sera synchroniser sur le serveur ensuite.

Ressources externes

Guides pour démarrer

- « Gérez vos codes source avec Git » sur Open Classroom (http://fr.openclassrooms.com/informatique/cours/gerez-vos-codes-source-avec-git)
- git immersion (http://gitimmersion.com/)
- Le git tutorial officiel (http://git-scm.com/docs/gittutorial)

Documentations

- La documentation officielle (http://git-scm.com/documentation)
- Le livre Pro Git (http://git-scm.com/book)
- Git Magic (http://www-cs-students.stanford.edu/~blynn/gitmagic/) montre énormément de choses inattendues que l'on peut faire avec git
- Une documentation proposée sur le site d'Atlassian (https://www.atlassian.com/fr/git) qui est en français et qui propose notamment une partie consacrée au différents workflows Git (https://www.atlassian.com/fr/git/workflows)
- Beaucoup d'astuces dans la page consacrée à Git du wiki de kernel.org (https://git.wiki.kernel.org/index.php/GitDocumentation)

Aide-mémoires

Il n'est pas évident de se souvenir des toutes les commandes git. Aussi, il peut vous être utile d'avoir, sous la main, un aide-mémoire dédié :

- une fiche réalisée par l'équipe GitHub (https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf) (format 2 × A4)
- une référence rapide interactive (http://ndpsoftware.com/git-cheatsheet.html), elle est organisée selon le niveau où on souhaite travailler (stash, index, remote...). Elle a l'avantage d'être disponible en français.

Vidéos

- Les vidéos officielles (http://git-scm.com/videos)

Extensions git pour le développeur

- git-extras (https://github.com/visionmedia/git-extras) est une extension qui se propose d'ajouter des commandes à git pour faire quelques manipulation récurrentes.

Outils de visualisation des dépôts git

Une liste d'outils qui peuvent représenter des alternatives intéressantes à gitweb qui est livré par défaut :

- gitolite (https://github.com/sitaramc/gitolite)
- gitblit (http://gitblit.com/)
- gitbucket (https://github.com/takezoe/gitbucket)
- gogs (http://gogs.io/)
- kallithea (https://kallithea-scm.org/)

Certains projets ont pour objectif d'être des alternatives open-source à GitHub :

- gitlab (https://about.gitlab.com/gitlab-ce/)
- gitorious (https://gitorious.org/)

Enfin, gerrit (https://code.google.com/p/gerrit/) est un outil spécialisé pour la revue de code.