GIT : Utilisation simplifiée en dix commandes

Webographie

1. http://blog.xkoder.com/2008/08/13/git-tutorial-starting-with-git-using-just-10-commands/

Pré-requis

1. un ordinateur de préférence sous Linux. (Nous ne traitons pas de Windows ici
2. Git. Sur la distribution Ubuntu, on peut l'installer avec la commande suivante :

```
$ Sudo apt-get install git\\ $ Sudo apt-get install git-core
```

3. un projet en développement, dans n'importe quel langage (C, C + +, D, Python, Perl, HTML, JavaScript, etc.)

Allons-y !

"git init" : initialisation d'une base git vide

Avant de commencer à utiliser la puissance de git, il faut créer un dépôt git vide et dire quels sont les fichiers à suivre.

La première commande présentée initialise un dépôt git vide.

Avant d'exécuter cette commande, se placer dans le dossier de base du projet. Par exemple, commencer par se placer dans /home/xk0der/SourceCode/myProject si tout le code source du projet y est stocké (nous désignerons ce dossier comme “dossier de base” dans la suite). Puis exécuter la commande suivante :

```
$ Git init
```

.
Cela va créer un répertoire vide dans le répertoire courant (le dossier de base).

Pour chaque projet à suivre avec git, commencer par aller dans son dossier de base et lancer cette commande.

Ensuite, lancer les 2 prochaines commandes présentées ci-dessous.

"git add" : Ajout de fichiers au référentiel git

L'étape suivante consiste à ajouter des fichiers au dépôt git, c'est-à-dire signales à git les fichiers dont on veut garder la trace. Exécutez la commande suivante à partir du dossier de base :

```
$ Git add .
```

Cela va ajouter au dépôt git, de manière récursive, tous les fichiers du dossier actuel et de ses sous-dossiers.

"git commit" : La commande commit

C'est une commande qu'on va utiliser souvent.

La commande ci-dessus ajoute les fichiers, mais ne “committe” pas les modifications apportées au dépôt git. Il faut donc exécuter la commande suivante pour effectuer les modifications finales :

```
$ Git commit -a
```

Cela ouvre un éditeur de texte (probablement vi) pour ajouter quelques commentaires sur ce commit. On peut aussi spécifier le message du commit en ligne de commande, avec le format suivant :

```
$ Git commit -a -m "Import initial"
```

L'option -m sert à spécifier le message de commit. Dans notre cas, le message de validation est “Import initial”. On peut mettre n'importe quel message.

On est prêt!

Maintenant, on peut commencer à coder avec un éditeur. Quand on sent qu'une étape est accomplie, ou simplement pour enregistrer l'état courant du projet, refaire une commande git commit comme ci-dessus. La commande commit va créer un nouveau cliché et sauvegarder l'état actuel du code source.

Git, comme tout système de contrôle de version, enregistre toutes les modifications apportées au code source. Cela permet d'aller et venir entre les étapes et d'inspecter les modifications.

Maintenant, nous allons avancer et apprendre quelques commandes de plus.

"git log" : Affichage du journal de validation.

on peut relire tous les commits faits jusqu'à présent avec la commande suivante :

```
$ Git log
```

Cette commande montre l'historique avec :

1. les numéro de hashages unique du commit
2. le nom et l'email de la personne qui a effectué la modif
3. la date et l'heure de la modif
4. Le message de commit

De tous ceux-ci, le premier élément, l'identificateur unique, est le plus important. Ce hachage sera utilisé avec certaines des commandes suivantes, et est utilisé avec une pléthore d'autres commandes git

"git status" : Vérification de l'état actuel

Lors du codage, on peut voir toutes les modifications avant de se résoudre à les stocker dans le dépôt git. C'est ce que permet cette commande :

```
$ Git status
```

La commande ci-dessus liste tous les fichiers qui ont changé depuis la dernière validation.

"git diff" : Trouver les différences entre les commit

Outre la seule visualisation de fichiers qui ont changé, on peut vouloir visualiser les différences dans le code source. C'est ce que fait la commande :

```
$ Git diff
```

Cette commande montre les différences (au format diff) relativement aux dernières modifications enregistrées. La commande diff peut aussi être utilisée.

Pour voir les différences entre un commit précédent et les changements en cours :

```
$ Git diff commit_hash
```

ici, commit_hash est les huit premiers caractères (chiffres hexadécimaux) du hachage indiqué plus haut dans le journal par la commande git-log. On n'a pas besoin de spécifier la valeur de hachage complète, juste les huit premiers chiffres suffisent.

Pour voir les différences entre les deux précédents commit :

```
$ Git diff commit_hash_1 commit_hash_2
```

Cette commande va afficher les différences entre les deux commits identifiés par commit_hash_1 et commit_hash_2.

"git branch" : Créer des branches

Les branches sont faciles à utiliser et très utiles.

Avant d'entrer dans le vif du sujet, quelques mots sur les branches.

- On peut voir les branches comme un détour du tronc principal.
- Toutes les branches sont reliées à la branche master nom donné au tronc principal.

- Chaque branche a son propre journal (log)
- Le code peut différer dans les mêmes fichiers entre les différentes branches. (C'est la caractéristique principale des branches). Git va même plus loin et permet d'avoir non seulement un contenu différent dans les fichiers selon les branches, mais aussi différents jeux de fichiers selon les branches.
- À un moment donné, on ne peut être assis que sur une seule branche. C'est-à-dire que c'est le code source de la branche en cours qui est en vigueur, celui auquel on veut faire des changements.

Pourquoi utiliser les branches ?

Soit un code source quasi-stable à distribuer prochainement. On envisage aussi d'y inclure une caractéristique importante, mais on craint de casser ce qui est fait et de retarder le plan de mise en service.

C'est ici que les branches entrent en scène. Il suffit de créer une branche expérimentale stable à partir du tronc principal (celui sur lequel on a travaillé jusqu'ici, la branche master). Une fois créée la branche expérimentale, on peut continuer la correction des bugs et la mise au point du code sur la branche master. En même temps, on peut continuer à travailler sur la branche expérimentale sans affecter une ligne de code de la branche master.

Finalement, si la branche expérimentale fonctionne bien, on peut la fusionner avec la branche master et publier le produit. Si la branche expérimentale ne fonctionne pas bien,on peut tranquillement publier le produit de la branche master. La branche expérimentale est toujours là pour continuer à travailler dessus jusqu'à ce qu'elle devienne stable.

C'est un scénario (et la principale raison, mais pas la seule), de créer des branches.

Revenons aux commandes. Pour créer une branche, Lancer la commande suivante :

Création d'une branche

```
$ Git branch branch_name
```

branch_name est le nom que l'on veut, par exemple :

```
$ Git branch experimental_feature
```

Branchement dans une branche ou "Je veux plus de branches"

Dans une branche, on peut créer autant de branches que l'on veut, en créant éventuellement un arbre très dense.

Il suffit de s'y placer (checkout de cette branche, voir ci-dessous commande 8) et d'émettre la commande de création de branche.

"git checkout" : Déplacer une branche et lister toutes les branches

Une fois créée une branche, s'y déplacer avec la commande ci-dessous.

<note important>**Toujours** faire un commit avant de lancer cette commande, sinon les changements vont se déplacer à travers les branches. Prendre cette habitude ! (ou jeter un oeil sur la commande git-stash, cf. la Partie II de ce tutoriel).</note>

```
$ Git checkout branch_name
```

branch_name est la branche où on veut se déplacer. Une fois fait le checkout, on peut afficher les états, journal, diff, etc, avec les commandes présentées plus haut.

Pour revenir à la branche principale (le tronc principal), exécuter la commande suivante.

<note important>Encore une fois, s'assurer d'avoir émis la commande commit</note>)

```
$ Git checkout master
```

Liste de toutes les branches

La commande suivante affiche toutes les branches disponibles dans le référentiel actuel.

```
$ Git branch
```

"git merge" : Fusionner deux branches

Se déplacer (checkout) vers la branche à laquelle on souhaite appliquer la fusion et exécuter la commande suivante.

```
$ Git merge branch_name
```

Cela va fusionner la branche branch_name avec la branche courante.

Par exemple, pour fusionner la branche “experimental_feature” avec la branche principale, taper les commandes suivantes

```
$ Git checkout master
$ Git merge experimental_feature
```

Git signale tous les conflits qu'il ne peut pas résoudre automatiquement (s'il en existe). On peut alors les résoudre manuellement.

Suppression d'une branche

La fusion faite, on peut supprimer la branche expérimentale si on le veut, avec la commande :

```
$ Git branch -d experimental_branch
```

Cette commande n'efface la branche que si elle entièrement fusionnée avec la branche HEAD courante. HEAD est la position actuelle dans la branche, le dernier commit.

"git rm" : Supprimer quelquechose dans le référentiel.

Lancer la commande suivante si on ne veut pas que git garde une trace d'un fichier ou un dossier (c'eest le contraire de la commande git-add). Le fichier reste sur le disque, dans le répertoire de travail.

```
$ $ git rm --cached chemin/vers/le/répertoire_ou_fichier
```

La commande git-rm ne supprime les fichiers du référentiel que pour le HEAD, Les révisions ou commits précédents garderont le fichier.

Pour finir

Encore des commandes utiles

Une visionneuse graphique du référentiel

Pour lancer une visionneuse graphique du référentiel, exécuter la commande suivante

```
$ gitk
```

Git - Interface utilisateur graphique

Pour ceux qui préfèrent une interface graphique, on peut installer une interface graphique pour git. Pour ubuntu :

```
$ apt-get install git-gui
```

On la lance par :

```
$ git-gui
```

En fait, il semble que la commande :

```
$ git gui
```

fonctionne de base, sans avoir besoin de l'installer.

Partie II du tutoriel - 10 autres commandes

"git checkout" : Voyage dans le temps, déplacement aller-retour entre révisions

Le but d'un système de contrôle de version est de conserver l'historique des révisions du votre code source… et de pouvoir visualiser à quoi ressemblait le code à tout moment de l'historique enregistré.

Git enregistre les révisions sous la forme de commits. Chaque commit est identifié pau un code de hachage unique. (Git utilise le hachage SHA1). On peut retrouver le numéro de hachage d'un commit en lançant la commande git-log (cf. la première partie). Le numéro de hachage est affiché dans le journal sous forme d'une chaîne hexadécimale sur la ligne qui commence par le mot “commit” dans le log.

Déplacement vers un commit dans le passé

Pour revenir à un commit particulier dans le temps, on utilise la commande git-checkout (introduite dans la partie I pour se déplacer entre les branches). La syntaxe est similaire, mais au lieu du nom branche, on utilise le numéro de hachage du commit.

<note important>Toujours s'assurer de faire des commits ou des stashs (voir la prochaine commande) des changements avant de lancer la commande checkout, Sinon, les changements seront perdus</note>

Voici la syntaxe :

```
$ git checkout commit_hash
```

Voyons un exemple.

On commence par lancer la commande git-log pour trouver le numéro de hachage d'un commit vers lequel se déplacer.

```
$ git log
commit 1ef801f70a99b07bb578bac4a3c2edb52b367e1d
Author: xk0der <amit .. AT .. xkoder .. com>
Date:   Fri Jun 5 12:52:16 2009 +0530

Added few comments

commit ccbaeec43300f19dd04308b6c62a3f03f6233725
Author: xk0der <amit .. AT .. xkoder .. com>
Date:   Fri Jun 5 12:51:30 2009 +0530

Initial import
```

(l'adresse mail est cachée)

On voit qu'il n'y a ici que deux commits. On veut aller au commit avec le message “Initial import”,

c'est à dire le premier commit. On lance donc la commande suivante :

```
$ git checkout ccbaeec4
```

on voit que seuls les huit premiers caractères du hash ont été saisis. C'est le minimum dont git a besoin pour identifier de manière unique le commit. On peut évidemment fournir plus de huit caractères ou même le hachage complet.

L'extraction terminée, on peut afficher les fichiers pour voir leur état dans le passé. Pour modifier les fichiers et propager les changements dans le HEAD du master, cf. **Commande 6** plus loin.

Retour au présent

Pour revenir au HEAD de master, qui est l'état actuel du code, taper la commande :

```
$ git checkout master
```

Comme déjà dit, ne pas oublier de valider les modifications par des commits ou des stashs.

"git stash" : déplacement sans commit

C'est une commande très pratique. Elle permet de stocker les modifications en cours dans un cache temporaire. On peut ensuite les rappeler quand nécessaire. Cela réduit la nécessité de faire un commit à chaque fois que l'on veut se déplacer dans les branches ou les commits.

La syntaxe est :

```
$ git stash
```

Toutes les modifications sont rangées dans un cache temporaire, et le HEAD courant reste propre. On peut le vérifier en lançant une commande git-status.

Le travail terminé, on peut revenir à l'état rangé en exécutant la commande suivante :

```
$ git stash pop
```

Quand utiliser git-stash

Voici un scénario typique.

Supposons qu'on travaille sur une caractéristique de la branche et qu'on apprend par un e-mail que quelque chose ne va pas dans le code sur le tronc (la « branche master »). Une façon de corriger quelque chose sur la branche master est de :

- D'abord faire un commit des modifications sur la branche

- puis faire un checkout de la branche master et corriger ce qui est nécessaire.
- Faire un cimmit des modifications du master
- Faire un checkout de la branche “feature”

Et si on ne veut pas faire de commit des modifications sur la branche “feature” ? Probablement parce qu'on est à mi-chemin de quelque chose, ou qu'on veut tester le code avant de faire un commit. On range tous les changements, comme indiqué ci-dessous.

```
$ git stash
$ git checkout master
...corrections...
$ git commit -a -m "Bug:12 fixed - thread no longer dies prematurely"
$ git checkout feature
$ git stash pop
...Continuer à travailler normalement...
```

Caches multiples

On peut ranger plus d'une série de changements, si nécessaire, en exécutant la commande git-stash plusieurs fois.

Quand on utilise la commande pop de git-stash, le dernier cache arrive en premier, un peu comme une pile.

Voir les caches stockés

Pour afficher une liste des changements mis en cache, exécuter la commande suivante

```
$ git stash list
```

On peut faire d'autres choses plus amusantes avec cette commande (cf. les pages de manuel pour git-stash). Mais pour une utilisation quotidienne normale, cela suffira.

"git diff" : quelques commutateurs pour mieux différencier.

Cette commande a été présentée dans la première partie. Nous allons ici discuter de certaines options en ligne de commande, pour aider à trouver la bonne information.

Trouver quels fichiers diffèrent (au lieu de ce qui diffère dans les fichiers)

Voici la commande :

```
$ git diff --name-status
```

Cela liste les fichiers qui ont été modifiés (M), ajoutés (A) ou supprimés (D) par rapport à l'état actuel

du HEAD et les modifications non validées, le cas échéant. On peut fournir des numéros de hachage de commits pour comparer ces deux commits.

```
$ git diff --name-status hash_1 hash_2
```

Utilisation du symbole HEAD

On peut utiliser le symbole HEAD pour voir les diffs, entre le HEAD courant et les révisions antérieures, comme suit:

```
$ git diff HEAD~1
```

Cette commande montre les diffs entre le HEAD actuel et le commit immédiatement précédent. Cette commande est un raccourci pour :

```
$ git diff commit_hash_for_HEAD
commit_hash_just_before_HEAD
```

Qu'est-ce que le HEAD ?

Comme discuté dans la première partie de ce tutoriel, HEAD se réfère au dernier commit sur la branche courante.

On peut modifier le nombre après le tilde (~) pour trouver les différences entre le HEAD et ce nombre de commits avant le HEAD courant. Expérimenter et voir le résultat. Il n'y a aucun risque à essayer

"git add" revisité.

Dans la première partie de ce tutoriel, nous avons appris à faire que git suive un ou plusieurs fichiers en utilisant la commande « git add ». Mais nous allons voir d'autres choses que peut faire git-add.

Avant cela, voyons comment nous avons committé les changements jusqu'ici.

```
$ git commit -a -m "Commit Message"
```

Nous savons que l'option **-m** précise le message de commit, à quoi sert le commutateur **-a** ?

La commande ci-dessus est (presque) équivalente aux deux commandes suivantes, dans cet ordre :

```
$ git add .
$ git commit -m "Commit Message"
```

On voit que le commutateur -a est en réalité un raccourci pour ajouter tous les fichiers. Mais n'avons-nous pas déjà ajouté les fichiers à suivre par git ?

Oui, mais git a besoin de savoir *quels fichiers il faut committer*, et c'est ce que fait la première commande '**git add .**'. Elle dit à git d'ajouter tous les fichiers, nouveaux, modifiés ou supprimés à l'étape de commit. Ainsi, une fois les fichiers préparés pour le commit, on réalise le commit (c'est-à-dire qu'on enregistre les changements) par la seconde commande (git-commit).

<note important>La commande est presque équivalente à la dernière séquence, alors quelle est la différence ?

Avec un git-commit, le commutateur -a n'ajoute pas les fichiers qui ne sont pas suivis par git. Tandis que '**git add .**' ajoute aussi les fichiers qui ne sont pas suivis par git pour le commit.

On va voir pourquoi utiliser git-add de cette façon.</note>

Ne committer que certains fichiers sélectionnés

Supposons que l'on travaille sur trois fichiers source. Deux fichiers sont corrects mais il reste du travail sur le troisième fichier. On peut committer seulement ces deux fichiers en lançant les commandes suivantes :

```
$ git add path/to/file1
$ git add path/to/file2
$ git commit -m "Some feature done"
```

Quand on a fini avec le troisième fichier, on valide les modifications apportées, soit par le biais des commandes suivantes :

```
$ git add path/to/file3
$ git commit -m "Third feature done"
```

ou en exécutant la commande suivante (puisque nous voulons committer tous les fichiers modifiés)

```
$ git commit -a -m "Third feature done"
```

On peut utiliser les jokers '*' et '?' pour utiliser plusieurs fichiers pour la commande git-add.

Par exemple:

```
$ git add header/*
```

Cette commande va ajouter tous les fichiers dans le dossier “header” à ‘l'étape de commit’.

<note tip>Retenir : git-add prépare simplement les changements pour le commit (c.a.d qu'il dit à git quels fichier il faut committer), le vrai commit (écriture dans le dépôt) intervient quand on lance la commande git-commit</note>

Tester aussi la commande git-reset (plus loin).

git-rm : un peu plus à savoir

Le commutateur –cached

Lorsque git-rm est utilisé avec ce commutateur, les fichiers sont retirés du dépôt git, mais ne sont pas supprimés du disque.

Exemple :

```
$ git rm --cached path/to/file/or/folder
```

La commande ci-dessus va supprimer le fichier ou le dossier, mais ils resteront sur le disque. Si on fait la commande git-status, Le fichier sera affiché comme supprimé, ainsi que un-tracked (non suivi).

git-rm sans le commutateur –cached

Utilisé sans le commutateur –cache, git-rm supprime le fichier du référentiel ainsi que sur le disque.

Il faut committer les changements après un git-rm, avec ou sans le commutateur –cached.

Tout n'est pas perdu !

Se rappeler que la suppression avec git-rm, validée, du ou des fichiers ne s'appliquent qu'à ce commit. Toutes les révisions précédentes ont encore le fichier ou le dossier supprimé. Si donc on fait une extraction des révisions précédentes, on peut retrouver les fichiers ou dossiers.

On ne peut pas modifier l'historique des commits (voir git-reset plus loin) car cela va à l'encontre de la fonctionnalité principale du système de contrôle de version qui est de conserver l'historique des changements dans le référentiel de code source.

Gestion des branches

Nous savons déjà comment créer, supprimer et nous déplacer entre les branches. Voici quelques commandes de plus pour vous gérer les dépôts.

Création d'une branche de certains anciens commits

Supposons que l'on veuille revenir à un commit particulier et créer une branche à partir de lui. On peut émettre les commandes suivantes :

```
$ git checkout commit_hash
$ git branch branch_name
```

```
$ git checkout branch_name
```

Un raccourci pratique pour la commande ci-dessus est :

```
$ git checkout -b branch_name commit_hash
```

La commande ci-dessus est exactement équivalente aux trois commandes précédentes.

Un exemple avec le raccourci:

```
$ git checkout -b bugfix_branch HEAD~2
```

La commande ci-dessus crée une branche nommée bugfix_branch, positionnée à deux commits au-dessous du HEAD courant.

On peut utiliser l'interface graphique gitk pour faire visuellement un checkout des branches.

git reset

Effacer les changements

Pour annuler des changements du code source dont on n'a pas besoin, la commande suivante efface toutes les modifications et revient au dernier état propre, à savoir le dernier commit (la dernière validation).

```
$ git reset —hard
```

<note warning>Etre très prudent en utilisant de cette commande car elle est irréversible.</note>

Cette commande efface toutes les modifications non committées et fait revenir tous les fichiers à l'état lors de la dernière validation (commit).

Cette commande peut être utile pour fusionner des branches des dépôts d'autres personnes en faisant ressortir les conflits, et si on veut restaurer le référentiel à un état propre. Avec cette commande, on retourne à l'état non fusionné.

Effacer les fichiers marqués pour un commit

Pour ne plus faire de commit sur des fichiers que l'on a marqués, on peut lancer la commande suivante pour retirer leur marquage

```
$ git reset
```

Cette commande ne supprime le marquage que pour les commits, les changements restent.

Effacer les commits

Cette commande sert dans les rares cas où il faut vraiment modifier l'historique. Ou pour les rares cas où on a fait de mauvais commits.

La commande suivante supprime un commit définitivement.

```
$ git reset –hard HEAD~2
```

Cette commande supprime les deux derniers commits c'est une sorte de rtour au troisième dernier commit.

Dans un travail collaboratif, cette commande est fortement déconseillée. Mais si on est seul, cela peut servir.

Garbage collection : Compresser les dépôts et libérer de l'espace

La commande suivante compresse et nettoye le dépôt git :

```
$ git gc
```

Cette opération peut prendre du temps0 Une fois terminé, on a gagné de l'espace disque. (gc = garbage collection).

Extra 1 : .gitignore, Ignorer certains fichiers

Tous les fichiers du dossier de projet ne doivent pas être suivis par git. Par exemple : les fichiers objets, les fichiers d'échange, etc

Créer un fichier nommé ‘.gitignore‘ dans le dossier de base du dépôt git et y inscrire le nom des fichiers comme expliqué ci-dessous, pour que git les ignore.

Les jokers simples sont reconnus

Les jokers ‘*‘ and ‘?‘ et les expressions régulières entre crochets `[ ]‘ sont reconnues.

Par exemple, pour ignorer tous les fichiers dont l’extension se termine par 'bak', on peut spécifier ceci dans le fichier .gitignore

- .bak

Si cette ligne est présente le fichier .gitignore, Git ignorera tous les fichiers se terminant par «.bak» n'importe où dans les dossiers du projet.

- Pour ignorer un fichier ou un dossier particulier

On utilise la notation de chemin absolu.

Supposons la structure de répertoires suivante pour le projet, où «Project» étant est le répertoire de base du référentiel du projet.

```
Project
    |-- .gitignore
    |
    |-- .git
    |
    +-- Folder_1
    |
    +-- Folder_2
    |       |
    |       --- Folder_3
    |
    -- Folder_4
            |
            +--- File_1
            +--- File_2
            +--- File_3
```

Supposons que l'on veuille ignorer le dossier «Folder_3» : mettre la ligne suivante dans le fichier .gitignore :

```
/folder_2/folder_3/
```

Notez le slash au début et à la fin.

- Si on supprime le slash précédent, git va ignorer tous les dossiers nommés «folder_3» à l'intérieur de n'importe quel dossier nommé folder_2.
- Si on supprime le slash à la fin, git traitera le nom «folder_3» comme un nom de fichier au lieu d'un nom de dossier. Dans ce cas, si nous avions un fichier nommé «folder_3» à l'intérieur de folder_2, il serait ignoré.

Maintenant, pour ignorer les fichiers File_2 du diagramme ci-dessus, placer ce qui suit dans le fichier .gitignore :

```
/folder_4/file_2
```

Noter qu'il n'y a pas de slash à la fin, si nous l'avions fait, git traiterait «file_2» comme un nom de dossier.

Voici donc les règles :

- Les chemins absolus commencent par un slash ‘/’.
- Les noms de dossiers doivent toujours être suffixé par un slash ‘/’.
- Les noms de fichiers ne doivent JAMAIS être suffixés par un slash ‘/’.

L'exemple ci-dessous montre l'utilisation de crochets. Ils fonctionnent très bien, comme notation des

expressions régulières. An example .gitignore file.

exemple de fichier .gitignore

```
# git ignore file
# comments start with hash

# ignore all object files, all
# files ending either with '.o' or '.a'
*.[oa]

# Ignore all files with
# the extension *.swp
*.swp

# Ignore a single file
/folderA/folderB/build.info

# Ignore a folder named
# temporary in the base folder.
/temporary/

# Ignore folders named _object
#  anywhere inside the project.
_object/
```

A quoi sert .gitignore ?

Regardons un scénario typique de codage.

On a un projet, on a ajouté tous les fichiers sur le dépôt .git et fait un commit initial. Maintenant, on édite quelques fichiers, l'éditeur crée des fichiers de sauvegarde se terminant par l'extension «.bak».

Si maintenant on fait un git-status pour vérifier les fichiers modifiés, git signale quels fichiers ont été modifiés, mais signale aussi les fichiers '.bak' comme non suivis.

Après quelques jours, la liste des fichiers non suivis s'allonge; ce qui fait que le git-status sort de l'écran.

De même, on ne peut donc pas faire de 'git add .' pour signaler les fichiers à commiter, car cette commande va ajouter des fichiers '.bak' à committer, encombrant inutilement le dépôt.

La solution : créer un fichier .gitignore dans le répertoire de base du projet avec une ligne contenant «*. bak».

Extra 2 : Ajouter quelques couleurs et se présenter à git

Votre nom et votre adresse email dans les commits

Editer le fichier .git/config dans le dossier de base (le dossier du projet) et entrer les lignes suivantes.

```
[user]
    name = Your Name
    email = your email Id
```

par exemple :

```
[user]
    name = xk0der
    email = amit@xkoder.com
```

Une fois placés ces détails dans le dossier, au prochain commit, le nom et l'email sera enregistré comme ceux de l'auteur de ce commit.

Coloriser la sortie de git

Un petit post de Gary explique comment faire : http://scie.nti.st/2007/5/2/colors-in-git