JETPULP
ALTAVIA AURA

- Accueil
- On air
- Webdesign
- Technologie
- Webmarketing
- Mash up
- JETPULP
- Contact
-

-

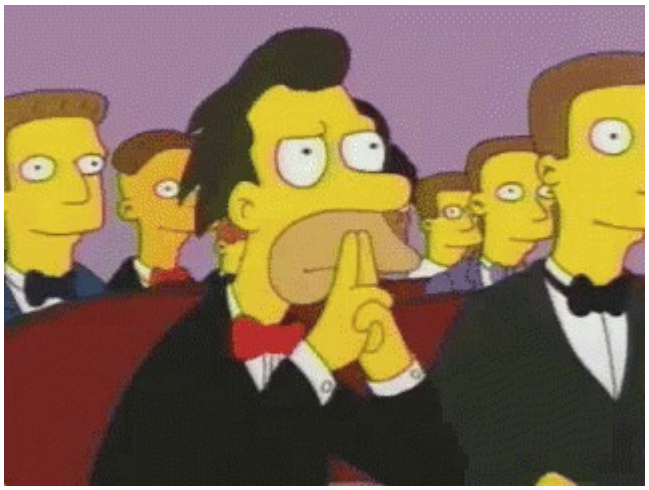
Publié le 22 novembre 2018

- Débutant
- 7 min

- Développement
- Intégration

GIT COMMENT ÇA MARCHE ? TUTO POUR DÉBUTANT

*Pour **comprendre GIT**, il est nécessaire de connaitre les éléments de base. Cette documentation n'est pas exhaustive et recommande des bonnes pratiques surtout adaptées aux débutants. Les informations présentes dans ce document s'étendent parfois au-delà de **l'outil GIT** pour donner des contextes d'utilisation.*
Pré-requis : l'utilisation d'un terminal (aussi appelé « console » ou « shell ») est nécessaire à l'exécution des commandes listées dans ce document.



QUELLES SONT LES DÉFINITIONS À CONNAITRE SUR GIT ?

Il existe des termes de base à connaitre pour **utiliser GIT** correctement. En voici une liste :

- COMMIT

  Contribution dans un **projet GIT**. Regroupement d'une ou plusieurs modifications, identifié par une série unique de caractères alphanumériques (SHA). Ce **commit** représentera une « version » du projet.

- BRANCH

  Par commodité, c'est le nom donné à une **série de commits**. C'est aussi et surtout une référence pour positionner le projet sur le dernier commit d'une série. La **branche GIT** par défaut s'appelle « master » et devrait correspondre à l'état de la production.

- MASTER

  C’est le nom donné à la **branche par défaut par GIT**. Elle fait généralement office de référence pour l’ensemble des contributeurs, et peut donc aussi correspondre à l’état du **projet en production**.

- REPOSITORY (OU « DÉPÔT »)

  C’est le nom d’un **projet GIT**. Il peut être local (sur son poste), ou distant (sur un serveur) et référencé par une **remote**.

- REMOTE

  Référence à un **projet GIT distant**. Ce projet distant doit être hébergé sur un serveur accessible par l’ensemble des contributeurs. La remote par défaut est appelée « origin ».

- INDEX (HEAD)

  C’est le nom donné à l’état courant du projet et qui constituera le prochain commit. On parle « d**’ajouter des fichiers dans l’index** » lorsqu’on souhaite créer un commit avec ceux-ci.

- MERGE/PULL REQUEST

  Ce n’est pas une opération propre à GIT à proprement parler, mais cette opération enrichit fortement l’aspect collaboratif de l’outil. Il s’agit de la procédure par laquelle une nouvelle branche sera soumise à vérification et acceptation de la part de l’équipe, et donc mergée dans le tron commun du projet. Elle cible **la branche master**, ou toute autre branche commune à l’ensemble des contributeurs. Le terme « **Merge** » est utilisé par l’outil interne **Gitlab**, alors que le terme « Pull » vient des collaborations publiques sur **Github** (équivalent public de Gitlab).

LES SCHÉMAS DES OPÉRATIONS GIT

Ces schémas sont des représentations des opérations menées avec GIT. Ils servent d’illustration aux commandes et principes expliqués dans le reste de ce document. Il convient donc de s’y référer chaque fois que nécessaire, pour faciliter la compréhension des textes.
À noter qu’un **historique GIT** se lit de bas en haut.

BRANC

COMMIT

Les commits A, B et C font partie d’une seule et même branche. L’**état HEAD** contient les modifications en cours. Si on réalise un commit sur la base de ces modifications, cela deviendra un commit supplémentaire, le dernier de la branche courante.

BRANCH A

MERGE A

Les 3 commits de la branche A ont été mergés dans la branche B. Ils font désormais partie de la branche B.

QUELLES SONT LES COMMANDES ESSENTIELLES SUR GIT ?

QUELLES SONT LES COMMANDES DE BASES SUR GIT ?

Les commandes ci-dessous permettent de réaliser des opérations simples sur un projet et satisfont donc l'essentiel des besoins de **versioning** et de **collaboration**.

- GIT STATUS

  Donne l'état du projet : quels sont les fichiers ajoutés dans l'index, ceux qui sont modifiés mais pas encore ajoutés, et ceux qui n'ont encore jamais été ajoutés dans le projet (« untracked files »). Permet également de savoir si la branche courante est en avance ou en retard par rapport à l'état connu de la branche distante (cf « **GIT fetch** »). Donne aussi d'autres informations en cas de situation « anormale » (conflits lors d'un merge par exemple). A utiliser autant que nécessaire !

- GIT ADD {/path/to/file(s)}

  C'est l'ajout d'un fichier ou dossier dans l**'index**. En d'autres termes, il s'agit d'ajouter ce qui sera contenu dans le prochain **commit**.

- GIT COMMIT

  Valide l'ensemble des modifications contenues dans l'index pour en faire un commit. Cette **commande** ouvre un éditeur de texte pour renseigner le commentaire du **commit**. Dans cet éditeur, des informations en commentaire (préfixées par un #) permettent de voir ce qui sera contenu dans le commit. Ces commentaires n'apparaîtront pas dans le message du commit.

- GIT CHECKOUT {ref/to/commit}

  Permet de basculer d'un commit à un autre. La référence d'un commit peut être son **SHA** (identifiant), le nom d'une branche ou un tag (« étiquette » marquant un commit en particulier).

- GIT FETCH {origin}

  Prend connaissance des modifications opérées sur le dépôt distant (référencé par le nom de la remote correspondante).

- GIT MERGE {branche à merger} {branche de destination}

  Consiste à récupérer les commits d'une branche pour les ajouter sur une autre. Deux cas d'usage :

  - le merge d'une branch d'évolutions vers prod ou preprod (à faire sur Gitlab par l'intermédiaire d'une **Merge Request**)
  - le merge de la branche distante dans la branche courante (eg : **GIT merge origin/master**) permet de mettre à jour la

branche locale (celle du poste) avec les développements réalisés par d'autres collaborateurs.

- GIT PUSH {origin} {branche}

  Envoie les derniers commits de la **branche** mentionnée sur le dépôt distant.

- GIT BRANCH {nom branche}

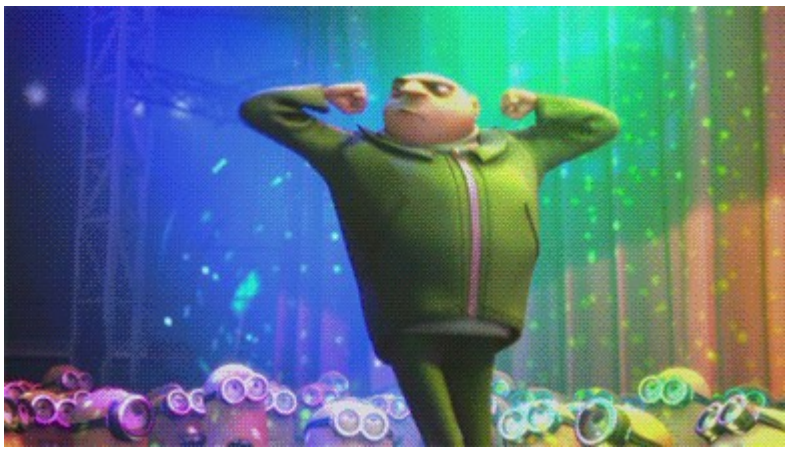
  Crée une nouvelle **branche** à partir du commit courant.

- GIT BRANCH –d {nom branche}

  Permet de supprimer la branche spécifiée, pour peu que celle-ci ait déjà été mergée complètement dans une autre branche. **GIT** empêche la suppression si ça n'est pas le cas.

QUELLES SONT LES COMMANDES COMPLÉMENTAIRES SUR GIT ?

En complément des commandes ci-dessus, quelques raccourcis peuvent être utilisés sans risques :



- GIT CHECKOUT –b {branch}

  Permet de basculer sur une nouvelle branche sans avoir à la créer au préalable. C'est la combinaison des commandes suivantes : git branch [branch] + git checkout [branch]

- GIT PULL {origin} {branch}

  Permet de prendre connaissance des modifications distantes sur une branche et les merger sur la branche courante. C'est la combinaison des commandes suivantes : git fetch [origin] + git merge [origin/branch]. À noter qu'ici, il est implicitement prévu que la branche à merger est la branche distante correspondant à la branche courante.

- GIT FETCH {origin} –prune

  Oublie les branches qui ont été supprimées sur le dépôt distant. Cela permet de « **nettoyer » le projet GIT** et d'éviter la suggestion de branches obsolètes lors de l'autocomplétion des commandes dans le terminal.Valide l'ensemble des modifications contenues dans l'index pour en faire un commit. Cette commande ouvre un éditeur de texte pour renseigner le commentaire du commit. Dans cet éditeur, des informations en commentaire (préfixées par un #) permettent de voir ce qui sera contenu dans le commit. Ces commentaires n'apparaîtront pas dans le message du commit.

Les commandes suivantes sont à connaître, mais peuvent comporter des risques. Attention à les utiliser en toute connaissance de cause :

- GIT BRANCH –D {branch}

  La mise en majuscule de l'option « d » permet de supprimer une branche même si celle-ci n'a pas été mergée dans une autre branche (master par exemple). Cela veut dire que les commits de cette branche qui ne sont pas dans une autre branche ne seront plus référencés simplement. Attention donc, il est ainsi possible de perdre tout son travail.

- GIT ADD .

  Plutôt que de cibler un fichier ou un dossier, cette commande cible le répertoire courant et ajoutera tous les fichiers du projets dans l'index. Attention aux fichiers de tests ou produits par le code (exports, css générés, sprites, js générés…) et qui n'étaient pas encore connu de GIT : ils seront ajoutés également à l'index.

- GIT COMMIT –a

  Ajoute tous les fichiers modifiés et déjà connus par GIT dans l'index, pour le prochain commit. Attention aux fichiers de code qui n'ont pas été au moins une fois explicitement ajoutés par ***git add***.

Il existe encore beaucoup d'autres commandes, mais ces dernières nécessitent une compréhension plus fine de GIT, et ne sont surtout pas nécessaires pour tenir correctement un projet GIT. Pour s'assurer de justement tenir un projet correctement, il faut se pencher sur un **workflow de travail avec GIT**. Et quoi qu'il arrive, il faut toujours faire attention à ce que GIT indique dans ses retours de commande ! GIT donne beaucoup d'informations sur l'état du projet et apporte son aide dans l'apprentissage de ses commandes : lisez-le 😉

QUELLE MÉTHODOLOGIE DE TRAVAIL ADOPTER SUR GIT ? (WORKFLOW)

GIT n'impose aucune méthodologie de travail. Pour assurer une certaine cohérence dans un projet, il est important de se mettre d'accord sur une méthodologie de travail commune. Il existe des workflows répandus, **GIT flow** est l'un d'entre eux. Voici les grands principes de travail adoptés chez **JETPULP**, et notamment basés sur GIT flow :

- UNE BRANCHE PAR FEATURE

  Sur une **branche**, on ne développe qu'un aspect fonctionnel du site, et le plus restreint possible. Cela permet de limiter les dépendances entre fonctionnalités en cas de roll-back (retour à une version précédente du projet) ou non validation. Cela facilite également la relecture de code lors d'une **merge request**. La convention de nommage de la branche sera la suivante : feature/nom-feature (sans majuscules)

- UNE BRANCHE PAR FIX

Pour les mêmes raisons. A défaut, lors d'une recette, il faudra au minimum **un fix par commit**. Autrement, ce sont plus généralement des branches comportant un commit unique. La convention de nommage de la branche sera la suivante : fix/nom-bug (sans majuscules).

- UNE BRANCHE COMMUNE DE DEV/RECETTE

  Chaque branche **feature/fix** est d'abord mergée sur une branche commune qui n'est pas **master** (dev, develop, integration/preprod…). Ce merge permet de faire une recette globale de l'ensemble des **features mergées** au fur et à mesure. Pour des raisons pratiques, on l'appellera « **dev** ».

- MASTER, ISO, PROD

  La branche « dev » est régulièrement mergée dans master après recette/validation. Cette opération donnera lieu à une « release », i.e. une mise en production. Le **merge dans master** ne se fait que dans l'intention de déployer en production. Autrement, la branche master ne représentera plus la production.

- HOTFLIX

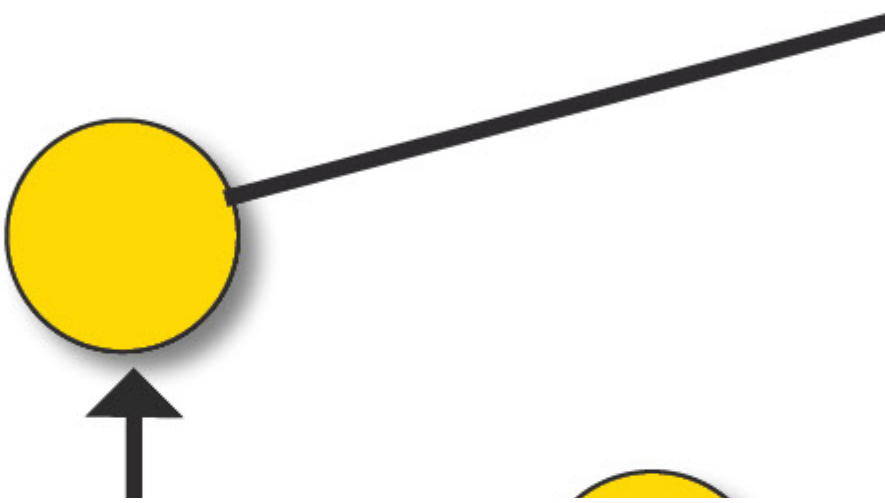
  Occasionnellement, un correctif urgent peut être nécessaire. Dans ce cas, on crée une branche depuis master, et on la mergera directement dans master. La **convention de nommage** de la **branche** sera la suivante : hotfix/nom-bug (sans majuscules).

- MERGE REQUEST

  Chaque **branche mergée** dans dev ou dans master doit faire l'objet d'une relecture par un autre collaborateur du projet.

FEATURE

MERGE FEATUR



Victor A.

En savoir +

Je fais ma BA et je partage cet article !

Mashup, Technologie

E-commerce : gérez votre base de données produits simplement grâce au PIM

0 Commentaires

/

Non classé, Technologie, Webdesign

5 propriétés CSS expérimentales à découvrir

0 Commentaires

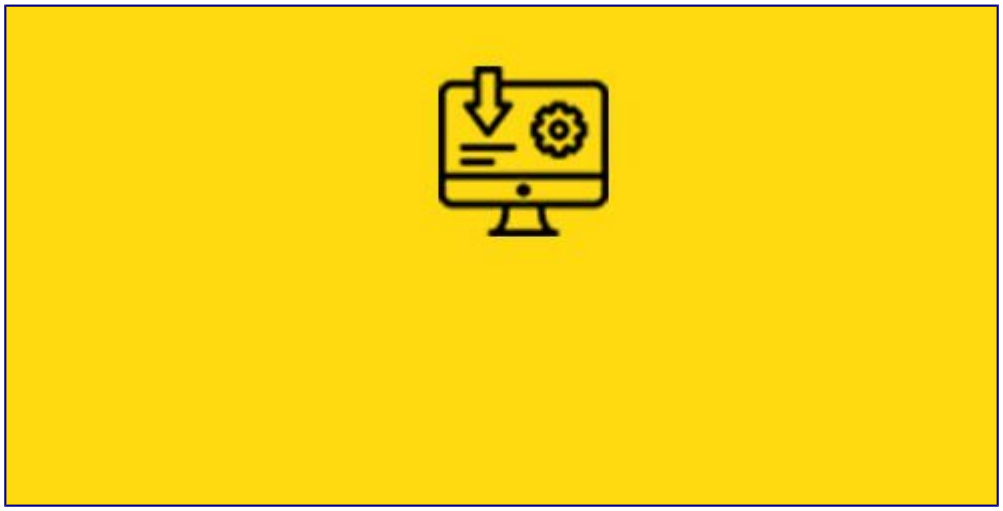
/

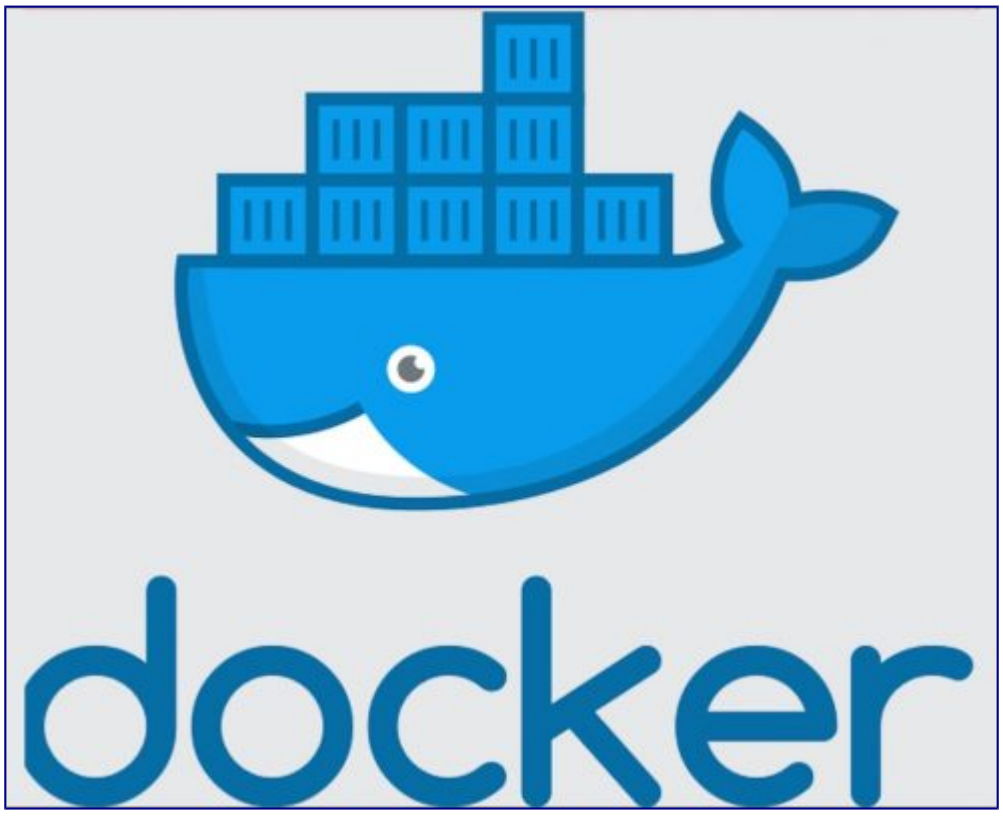
Mashup, Technologie

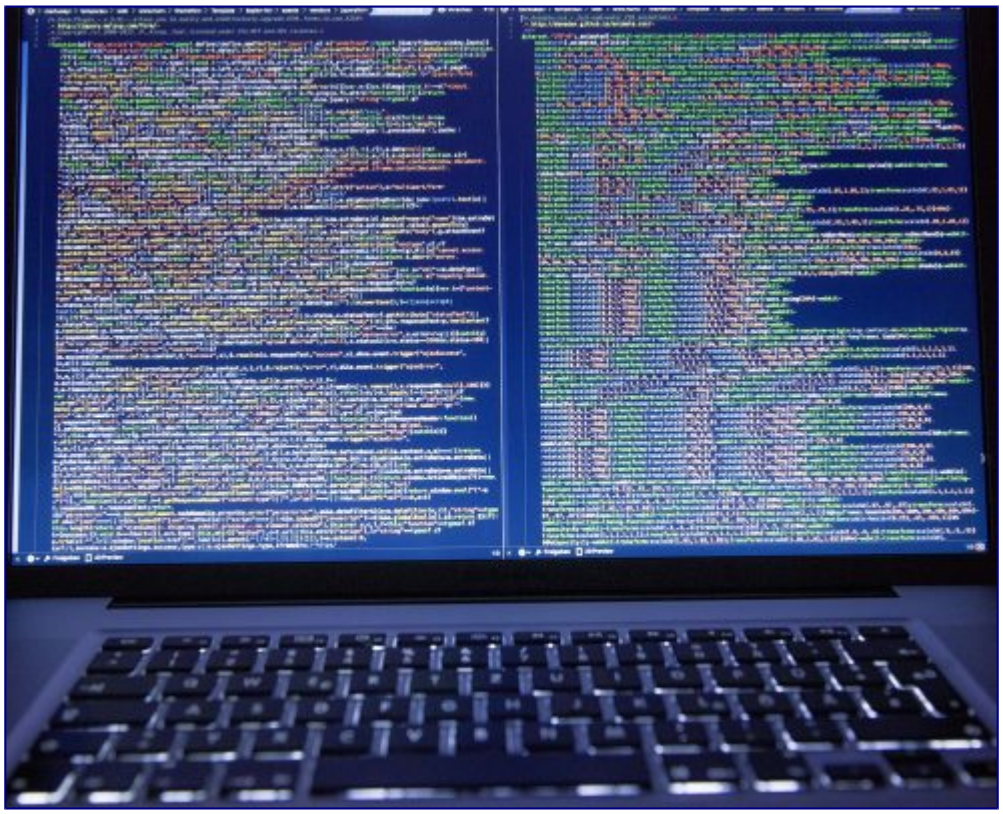
GIT COMMENT ÇA MARCHE ? TUTO POUR DÉBUTANT

0 Commentaires

/








'replace_interests' => false,
'send_welcome' => false,
('error', $result)) {
('response'=>'error', 'messa
('response'=>'success');
($errResult);


GULF OF MEXICO
Havana
Cape Hatteras
Philadelphia
New York
Boston
Montreal
Quebec
Atlanta
GEORGIA
FLORIDA
CAROLINA
VIRGINIA
MAINE
BELIZE
YUCATAN

[Précédent](#)[Suivant](#)

0 Commentaires

Répondre

Se joindre à la discussion ?

Vous êtes libre de contribuer !

*

*

© Copyright - Blog JETPULP

En poursuivant votre navigation sur ce site, vous acceptez l’utilisation de cookies.Accepter En savoir plus

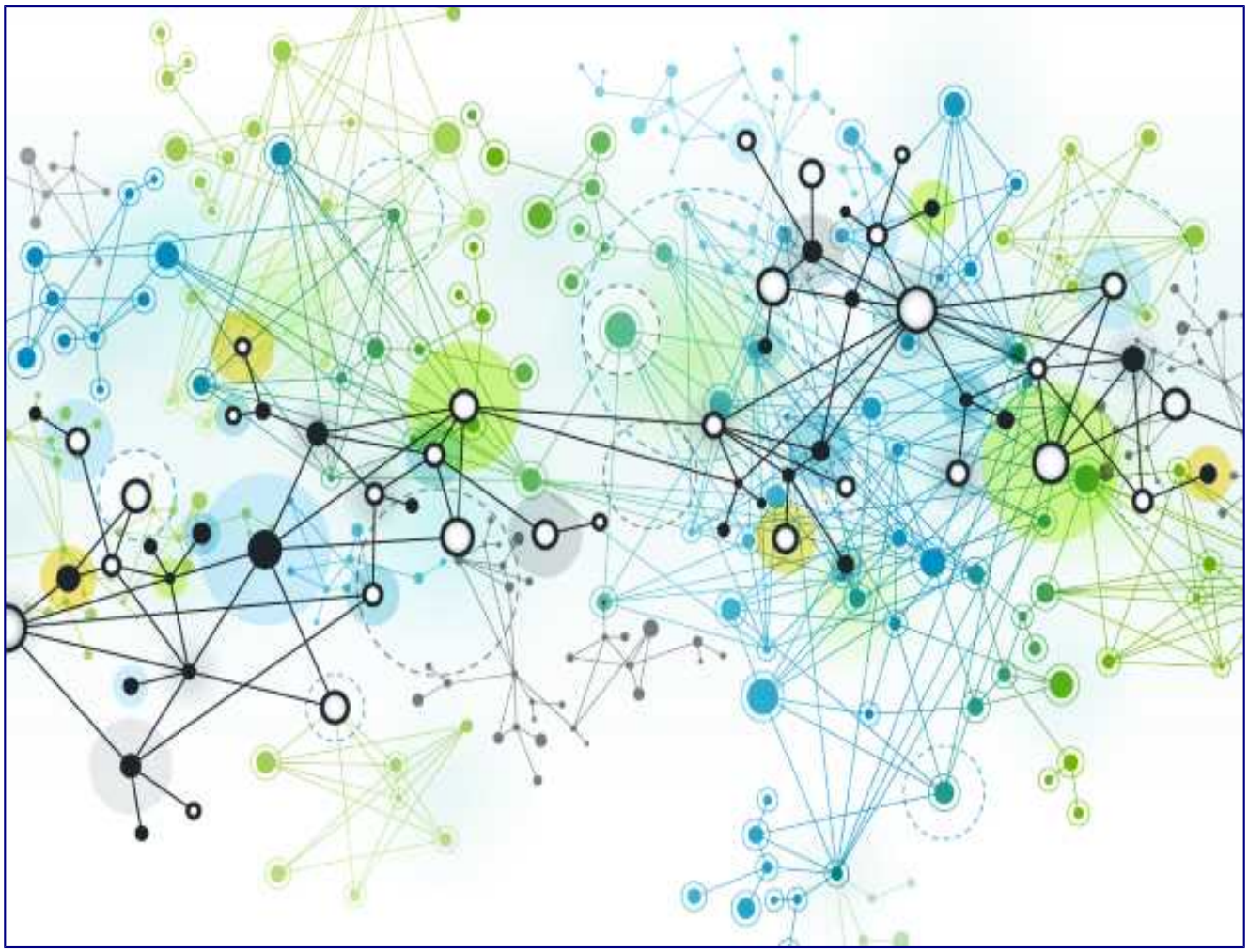
Nécessaire Toujours activé

Et si vous compreniez enfin Git et GitHub ?

5 commentaires

Si Svn (Subversion) est la version « client-server » d'un système de gestion de version, alors Git en est sans conteste la version « peer-to-peer ».

Si vous découvrez le système de gestion de version Git et que vous l'utilisez seul et occasionnellement, il existe peut-être encore beaucoup de zones d'ombre pour vous sur son utilisation ainsi que sur la totalité des possibilités offertes. D'ailleurs, êtes-vous réellement sûr de comprendre la portée de ce système de gestion de version distant et distribué ?



C'était également mon cas. Tout le long de mon amélioration continue de l'utilisation de Git, je vais maintenir cet aide-mémoire. Il abordera :

- la raison d'être de Git par l'exemple,
- la description des actions de Git pour un travail sur une branche,
- la description d'une méthode de travail multi-branche avec Git et
- le description de Git avec de multiples sources via GitHub.

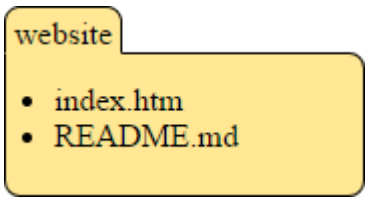
Si vous ne connaissez vraiment pas grand chose à Git n'hésitez pas à lire : Démarrage rapide de GIT et particulièrement :

- À propos de la gestion de version et
- Rudiments de Git

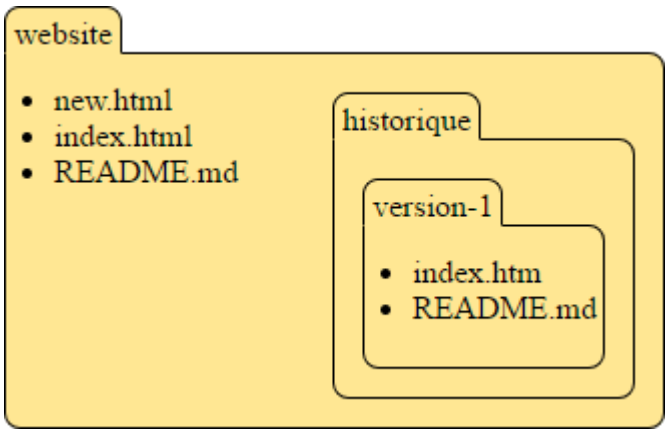
Git, comment ça marche ?

Créons un système de version local

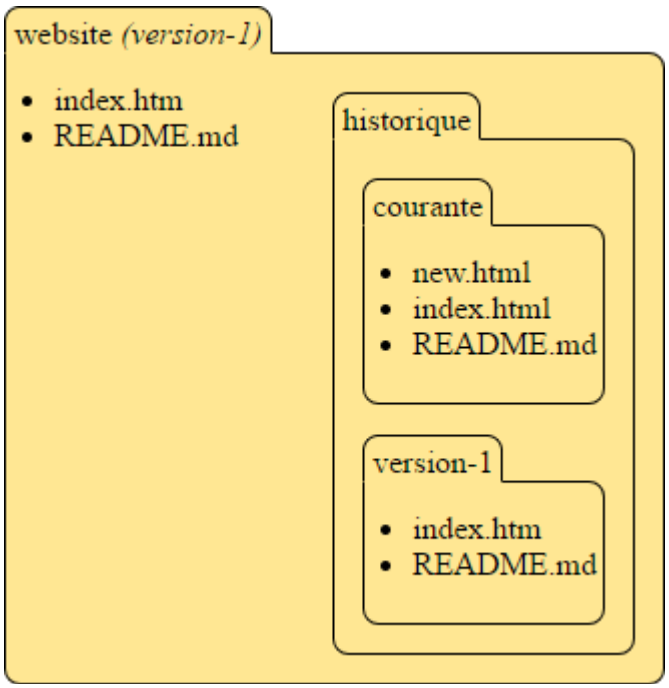
Imaginons que je crée un dossier `website` et que je décide d'y créer divers fichiers destinés à être ouverts dans un navigateur. Actuellement je travail seul sur ce projet dans ma société et avec mon propre ordinateur.



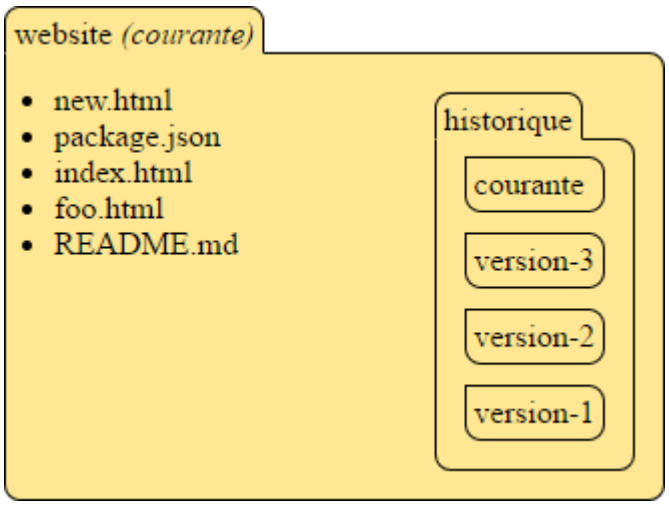
S'il me venait l'envie de modifier le site, mais de toujours être capable de revenir à sa version antérieur, je peux adopter une approche d'historisation dans un dossier annexe `historique`. Dans ce dossier, je placerais une copie du site appelé `version-1`. Ensuite je modifierais l'original.



Pour permettre au site visionné dans le navigateur d'être toujours dans le même dossier, nous pourrions également décider de créer un dossier `courante` qui représenterait toujours la version du site la plus récente. En faisant cela je pourrais mettre mon dossier `website` dans l'état de la `version-1` et je sais que je pourrais revenir à l'état de la version `courante` puisque j'en ai une copie dans l'historisation des versions.



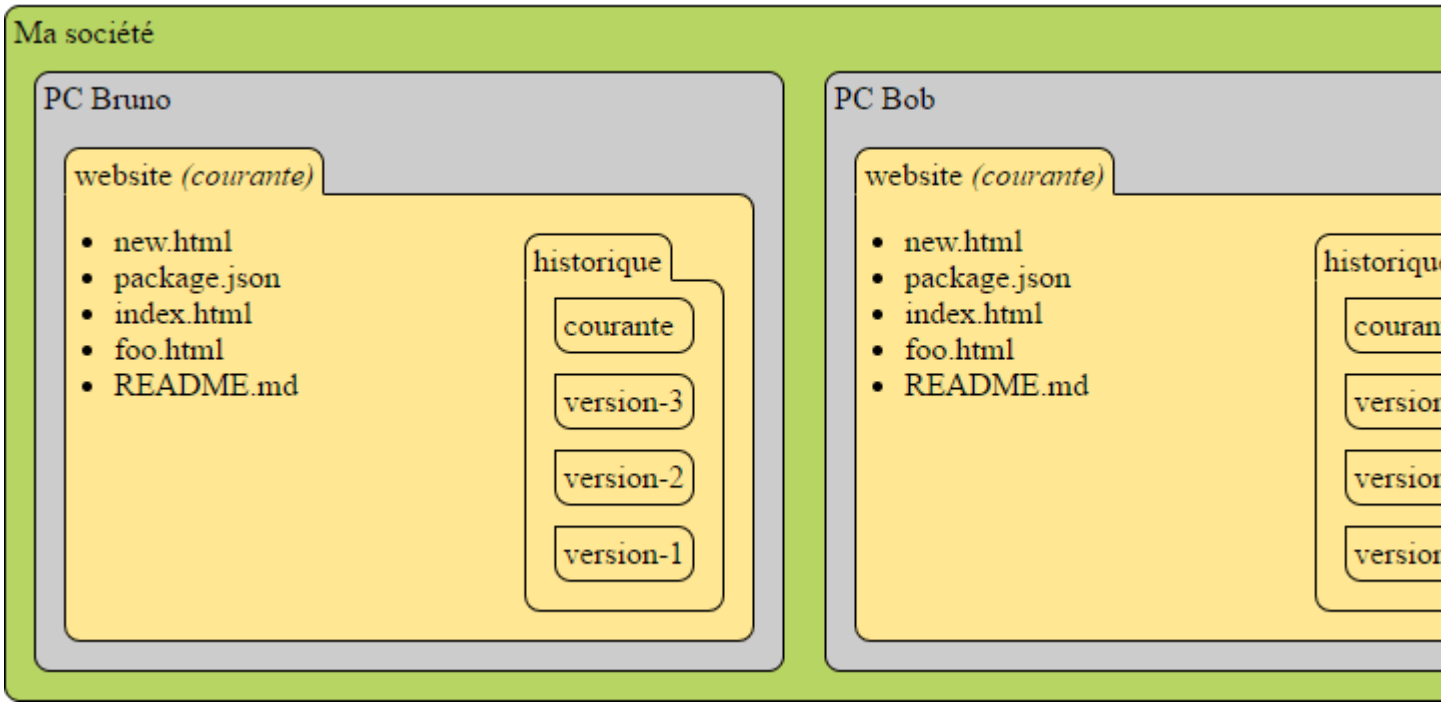
Ainsi après plusieurs versions du site, mon dossier d'historisation pourrait ressembler à celui ci-dessous. Mon dossier de travail serait sur la version courante.



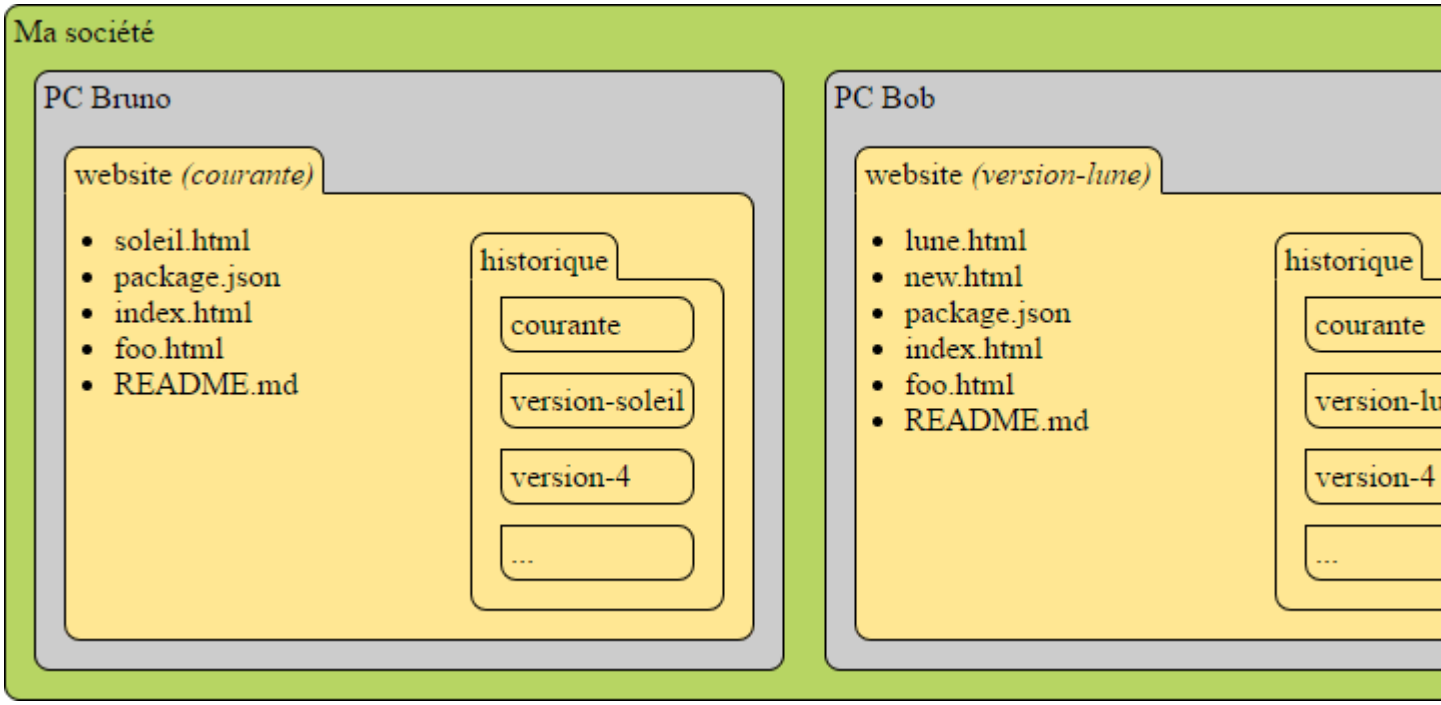
Nous venons de créer un système de version rudimentaire !

Créons un système de version centralisé

Imaginons maintenant qu'une seconde personne dans ma société rejoigne le développement du site. Pour travailler sur le même site que moi (Bruno), cette personne (Bob) copierait intégralement mon dossier `website`.

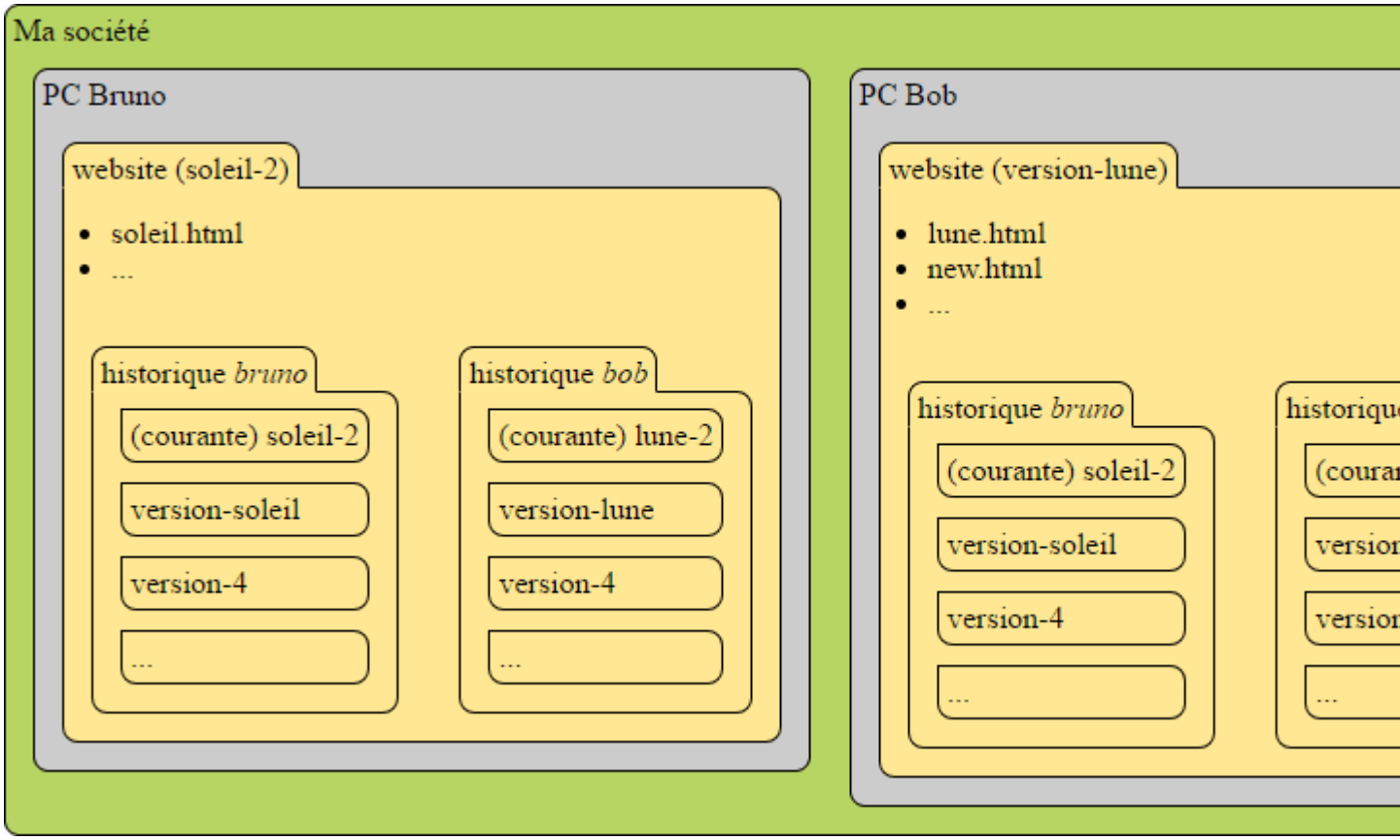


Bruno pourrait très bien créer une version du site soleil pendant que Bob continuerait une version du site lune. Les deux dossiers pourraient donc ressembler respectivement à l'image ci-dessous. Bruno serait dans la version de travail `courante` et Bob serait revenu dans une version de travail `version-lune` pour une raison quelconque.

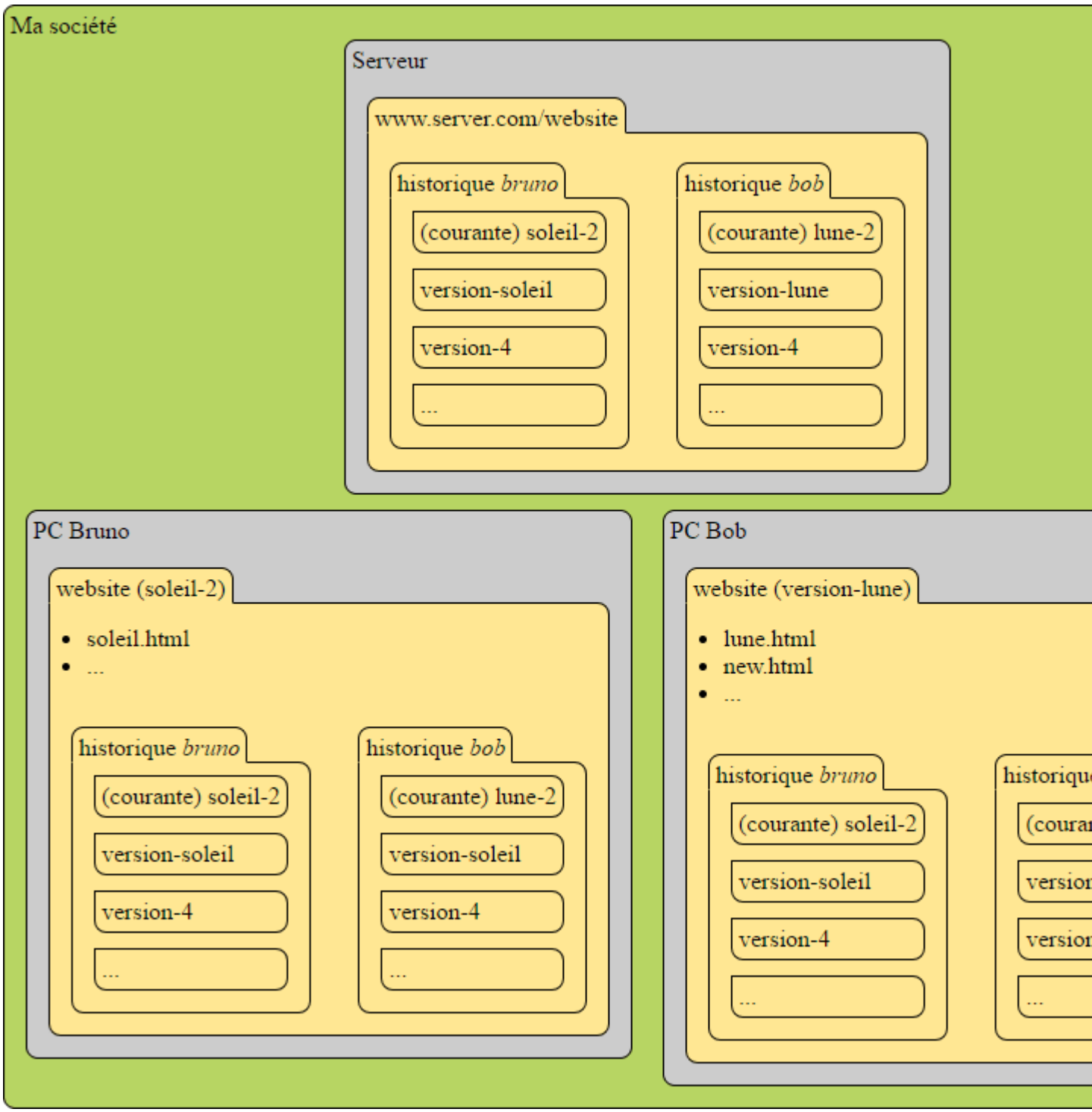


Si à présent nous souhaitions chacun pouvoir afficher l'état du travail de l'autre, nous devrions pouvoir différencier nos historiques de version, et chacun nous partager notre historique. Nous devrions alors adopter une structure comme celle ci-dessous. Nous pouvons d'ailleurs voir que si

nous souhaitions afficher la `version-4` nous pourrions la prendre depuis n'importe quel historique puisque « les versions » `version-4` sont identiques.



Pour ne pas avoir à chaque fois à copier le projet d'un ordinateur à l'autre (en prévisions des autres ordinateurs qui pourraient nous rejoindre), nous décidons de copier nos deux historiques de projet sur un serveur distant qui servirait de référence. À présent nous pouvons supprimer ce que nous voulons sur nos machines, il suffira à chaque fois d'aller récupérer l'original sur le serveur. À chaque fois que nous ferrons des modifications sur les historiques, nous prendrons soin de les monter sur le serveur. Pour avoir nos historiques à jour, il nous faudra également le demander au serveur. Pour finir, le serveur n'a pas d'espace de travail, il ne contient en fait que les historiques. C'est Bruno ou Bob qui choisissent ce qu'ils vont souhaiter afficher sur leur PC.

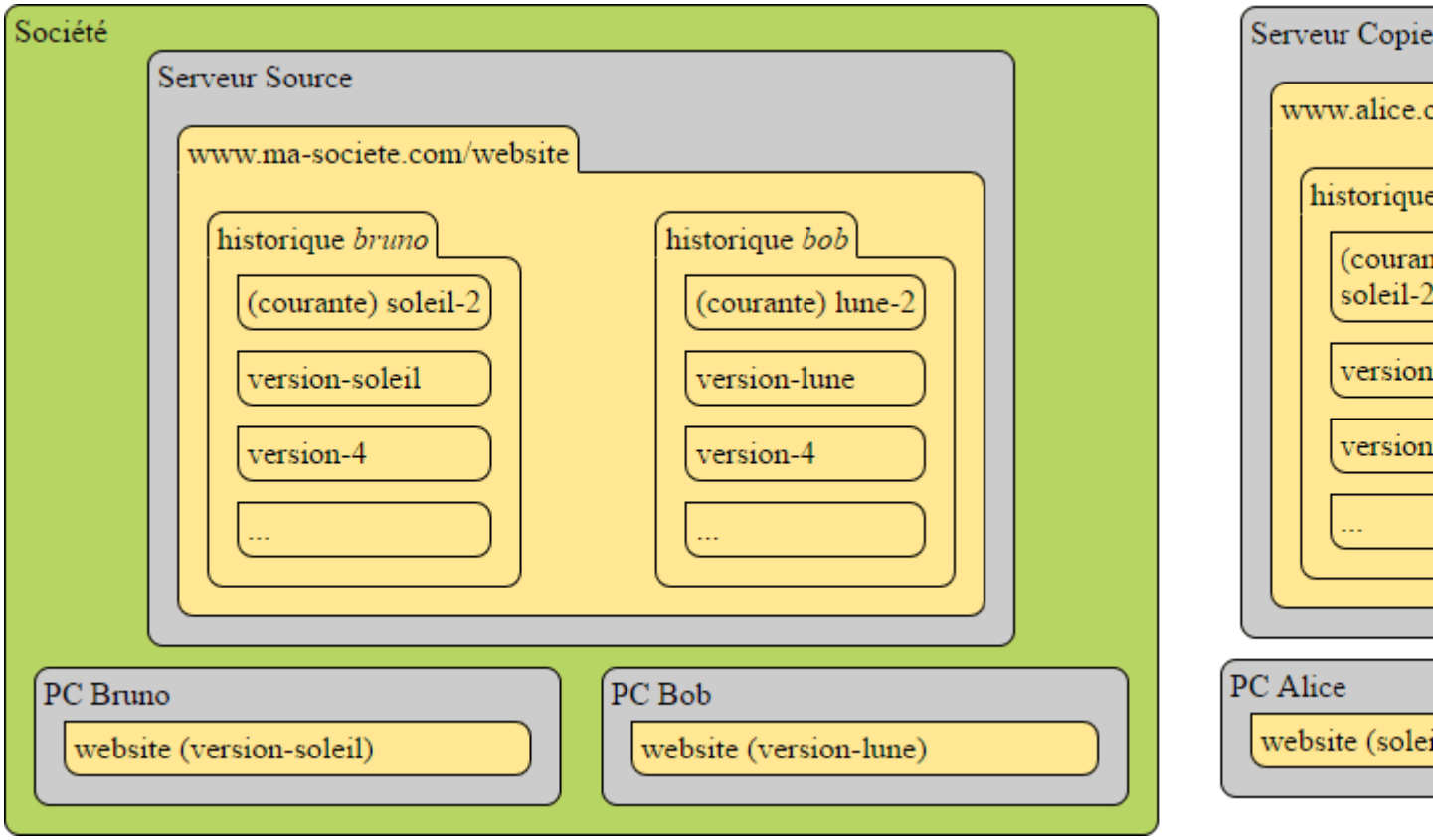


Nous venons de créer Svn (Subversion), un système de version centralisé !

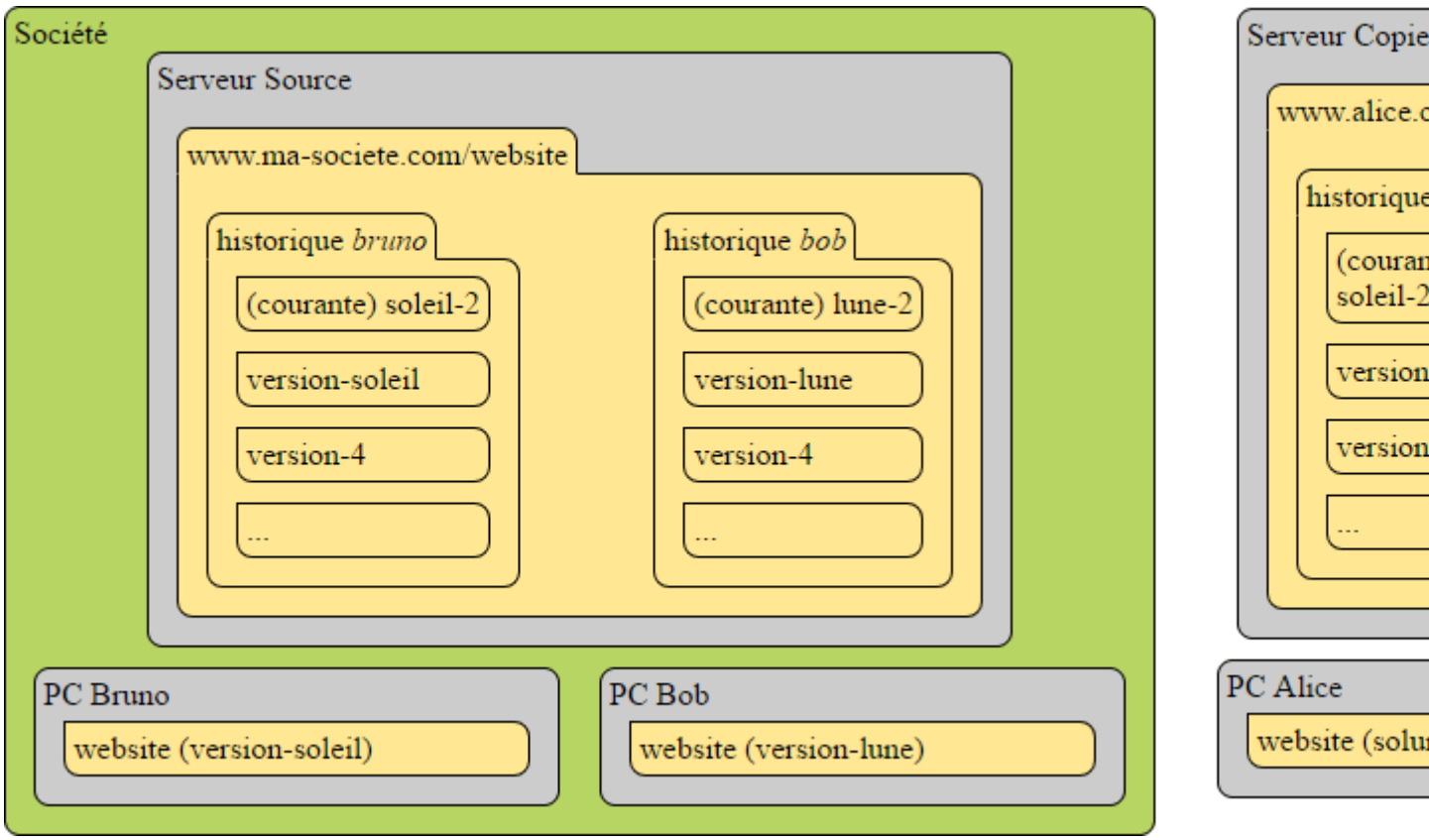
Créons un système de version distribué

À présent un troisième développeur (Alice) veut également accéder au code du site pour le modifier. Alice ne fait pas partie de la société. La société ne souhaite pas qu'Alice puisse ajouter son historique au sien. La société souhaite cependant qu'elle puisse non seulement partir de son travail, mais aussi avoir accès à toutes les versions de celui-ci dans tous les historiques ! La société lui permet donc de copier l'intégralité du serveur... chez elle ! Dans l'exemple ci-dessous, Bruno ne

travail par exemple plus sur la dernière version de son historique mais sur `version-soleil` et Alice travail sur la version `courante` de Bruno.



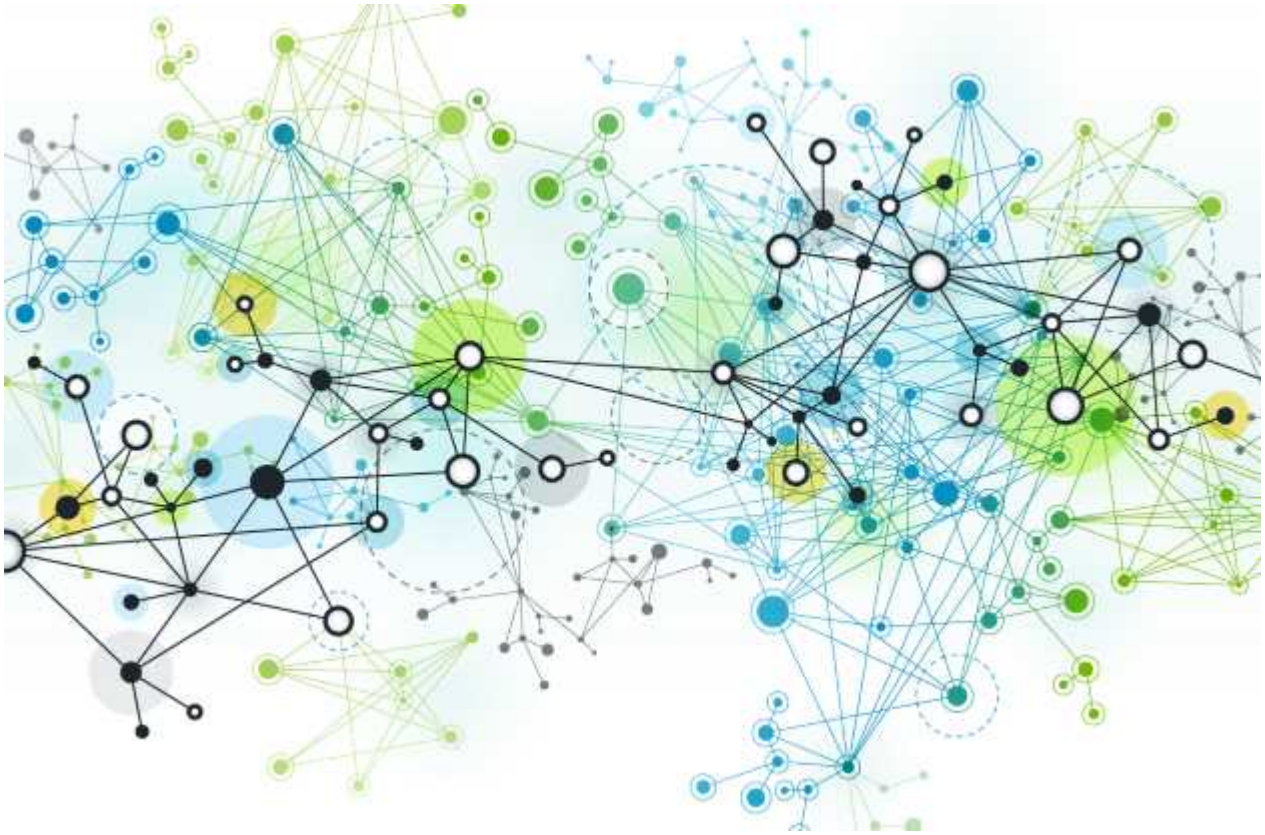
Comme Alice travail maintenant sur son propre serveur, elle peut aménager ses historiques comme elle le souhaite. Tout le travail effectué après la copie sera versionner comme elle l'entend. Par exemple Alice pourra renommer l'historique `bruno` en historique `base`. Elle pourra supprimer l'historique de `bob` de son serveur ou encore copier l'historique `base` en historique `solune` et continuer son travail dans cet historique !



Ajoutons à cela la possibilité

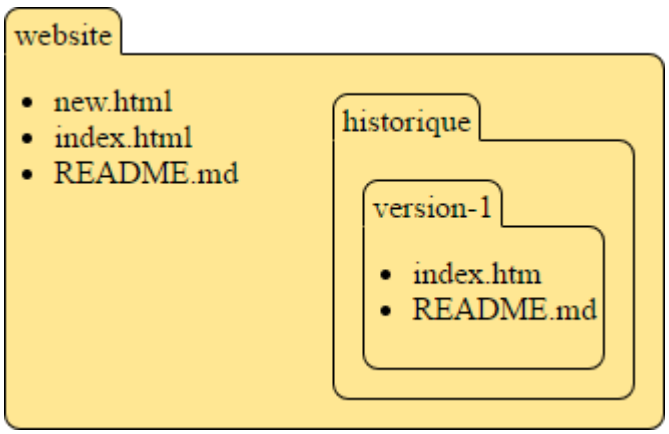
- de reverser le travail effectué par Alice sur le serveur copié dans le serveur source ou de synchroniser le serveur copié avec les nouveautés du serveur source,
- de directement se connecter à d'autres serveurs pour mettre à jour différents historiques ou encore
- de fusionner le travail entre des historiques sur le même serveur ou même de serveur à serveur,

et vous venez de créer Git !

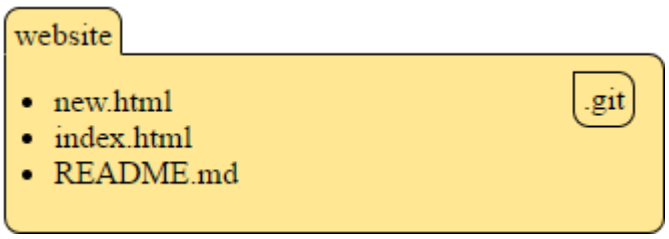


Imaginez qu'il y a pleins de dépôts distants différents pour un même projet avec des tas de répliques et que sur chaque poste de travail on en trouve des clones avec une version de travail précise. Un poste de travail peut très bien avoir plusieurs dossiers provenant d'un même dépôt distant et dans des états de version différents.

Pour finir, en réalité, sur vos postes machines, ce que vous voyez en réalité ce n'est pas cela :



mais cela :



car tout est optimisé et rangé de manière à ne pas avoir la copie de toutes les versions, mais de pouvoir en sortir n'importe laquelle à partir des informations de « différence » d'un instantané (révision) à l'autre. On remarque d'ailleurs qu'une même révision peut exister dans deux branches différentes si celle-ci est une copie de l'autre à un certain point. Cela signifie que les branches « n'existe pas » mais sont une vue de Git pour présenter un groupement de révision. Il existe aussi des étiquettes (« tags ») qui permettent de réellement indiquer à partir de quel instantané on peut estimer qu'on en est à la version `vX.X.X` d'un dépôt ou utiliser des livrables (« release ») par exemple avec GitHub.

Gérer son projet Git avec une branche

Avant de commencer, il me semble important de se rappeler que Git copie l'intégralité d'un dépôt (« repository ») distant (« remote ») en local et que la majorité des actions effectuées par l'outil de commande reviendra en réalité à uniquement changer votre dépôt local. Ainsi recherche, changement de version, changement de branche, tout sera très rapide !

Voici un schéma interactif de toutes les interactions dans le système de gestion de version de Git, vous pourrez aisément constater les actions en local bien plus nombreuse que les autres. Cliquez sur un élément pour accéder à son descriptif ci-après.

commit -a
add
commit
push
pull [-r]
fetch
merge *or* rebase
checkout HEAD
checkout
reset
reset
diff HEAD
diff
Local
Workspace
Index
Internet
Local Repository
Remote Repository

Workspace

Le *Workspace* ou *Working Tree* ou *Working Directory* est l'**espace de travail**. C'est une extraction unique d'une version du projet à partir de laquelle vous pouvez travailler. Ce dossier est stocké sur votre machine. Les fichiers ici sont soit identiques à la dernière révision enregistrée dans le dépôt local soit à l'état « Modifié ».

Index

L'**index** ou *Staging Area* correspond à la liste complète des fichiers présents physiquement dans votre espace de travail qui ont été déclarés comme faisant partie de la prochaine révision que vous allez enregistrer. Il est donc possible qu'un fichier présent dans l'espace de travail ne soit pas présent dans l'index si vous ne l'avez pas ajouté. Les fichiers indexés sont dans un état « Indexé ».

Local Repository

Le *Local Repository* est le **dépôt local**. C'est le dossier `.git` stocké dans votre espace de travail. C'est une réplique totale, sur votre machine, du dépôt distant. C'est dans le dépôt local que vous aller stocker vos révisions du projet. Le couple « .git » et l'espace de travail constitue ce que l'on appel le clone du dépôt distant. Les fichiers ici sont dans un état « Validé ».

Remote Repository

Le *Remote Repository* est le **dépôt distant**. C'est le dossier dans lequel est stocké le projet sur le serveur qui fait office de référant. Vous ne pouvez pas directement modifier ces fichiers. C'est la référence lorsque vous clonez un projet sur votre environnement afin d'en créer une réplique local et d'en initialiser la révision la plus récente en tant qu'état des fichiers de l'espace de travail.

add

Utiliser `add` permet d'ajouter des fichiers présent dans votre espace de travail vers l'index en prévisions d'une action futur vers un dépôt. Chaque fichier passe donc de l'état « Modifié » à « Indexé ».

reset

Utiliser `reset` permet de retirer des fichiers de l'index ou annuler un instantané dans le dépôt local tout en conservant (ou non) les modifications faites. Pour annuler les modifications faites tout en laissant les index et dépôt local en place il faut utiliser `revert`.

commit

Utiliser `commit` permet d'acter l'intégralité des fichiers de votre index dans le dépôt local en tant que nouvel révision. Chaque fichier passe donc d'« Indexé » à « Validé ».

commit -a

Utiliser `commit -a` pour ajouter automatiquement la totalité des nouveaux fichiers à l'index puis les acter dans le dépôt local.

push

Utiliser `push` pour monter et aligner l'intégralité des fichiers du dépôt local sur le dépôt distant.

fetch

Utiliser `fetch` pour descendre et aligner l'intégralité des fichiers du dépôt local depuis votre dépôt distant.

merge ou rebase

Utiliser `merge` pour fusionner les modifications présentent entre les fichiers de votre dernière révision et ceux rapatrier du dépôt distant (pour plus d'information sur la différence entre `merge` et `rebase` voir ici).

pull [-r]

Utiliser `pull` pour descendre et ré-aligner les modifications présentent entre l'espace de travail, le dépôt local et le dépôt distant. En cas de conflit, l'action `fetch` aura été effectuée et vous devrez résoudre les conflits et utiliser `merge` vous même.

checkout

Utiliser `checkout` permet d'extraire un instantané (une révision) et d'aligner l'espace de travail avec cet instantané du dépôt local depuis : la branche courante, une autre branche ou l'index.

status ou diff

Utiliser `status` ou `diff` permet de comparer vos fichiers de l'espace de travail avec ceux de l'index ou du dépôt local. Vous pouvez ainsi voir ce qui a changé dans un comparateur de version.

stash

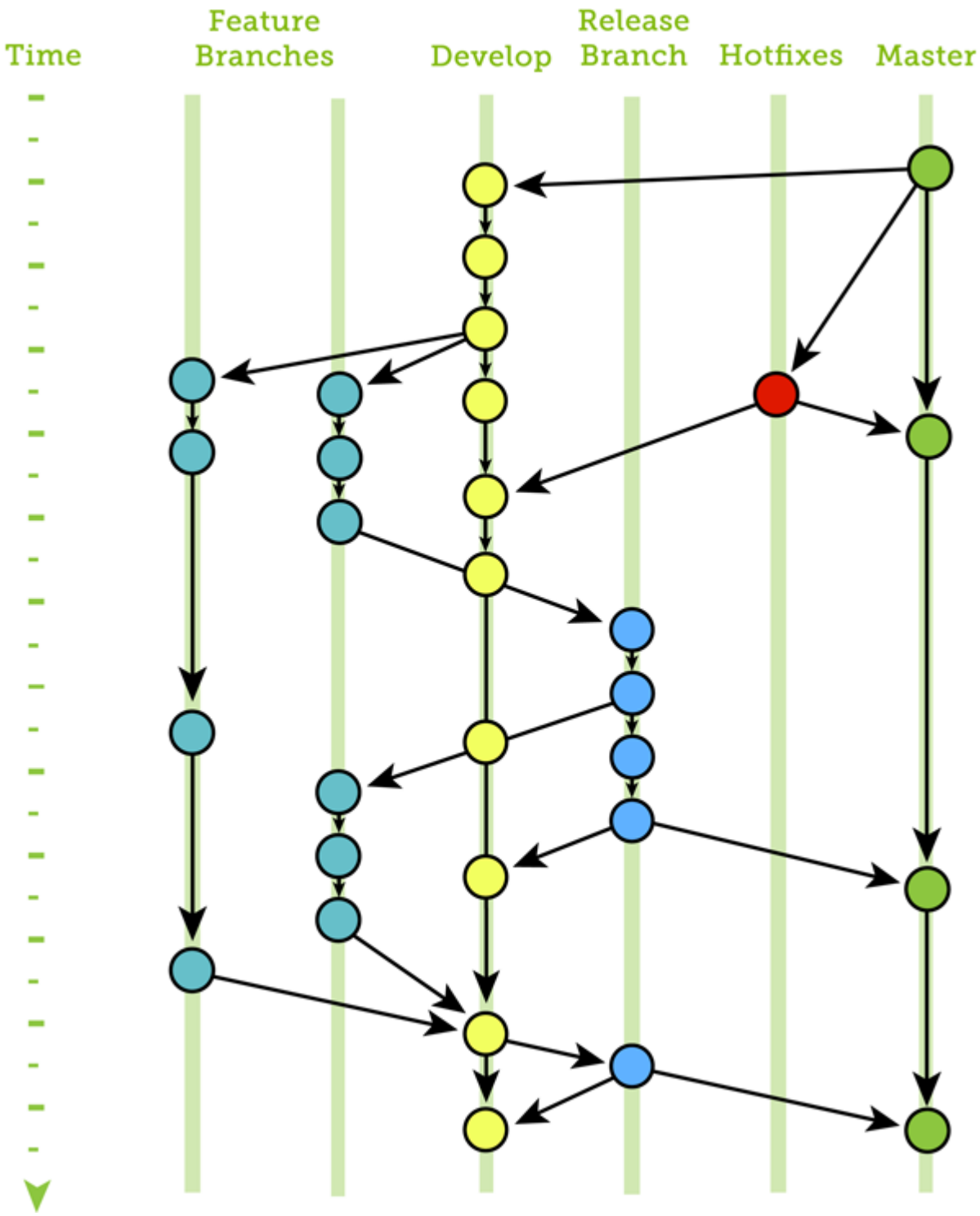
Utiliser `stash` pour remiser l'état de votre espace de travail et de votre index dans la remise afin de ne pas perdre votre travail lors d'une extraction. Cela vous permet de ne pas acter un travail à moitié fini.

> Pour en savoir plus, vous pouvez lire Les Bases de GIT et plus particulièrement :
>
> - Enregistrer des modifications dans le dépôt et
> - Travailler avec des dépôts distants

Gérer son projet Git sur plusieurs branches

Comme créer une branche ne revient qu'à créer des aiguillages d'instantanés, cette opération est très rapide. Il ne faut donc pas hésiter à le faire ! Je vous propose ici une structure de travail avec plusieurs branches pour un projet afin de vous organiser en équipe.

Il me semble que la structure proposée par SmartGit est pertinente aussi vais-je la décrire juste après ce schéma.



Le but d'une branche et sa position dans l'avancement du travail sur le projet depuis le développement jusqu'aux livrables doit être indiqué dans son nom, c'est pourquoi nous utiliserons les noms suivants :

- **release/{nom de branche}** – chaque livrable est associé à une branche correspondant à une version final de l'application.
- **master** – une branche permanente comme référence correspondant à la version définitive des modifications entre chaque version de l'application. Elle correspond donc à chaque instant à la dernière version de l'application.

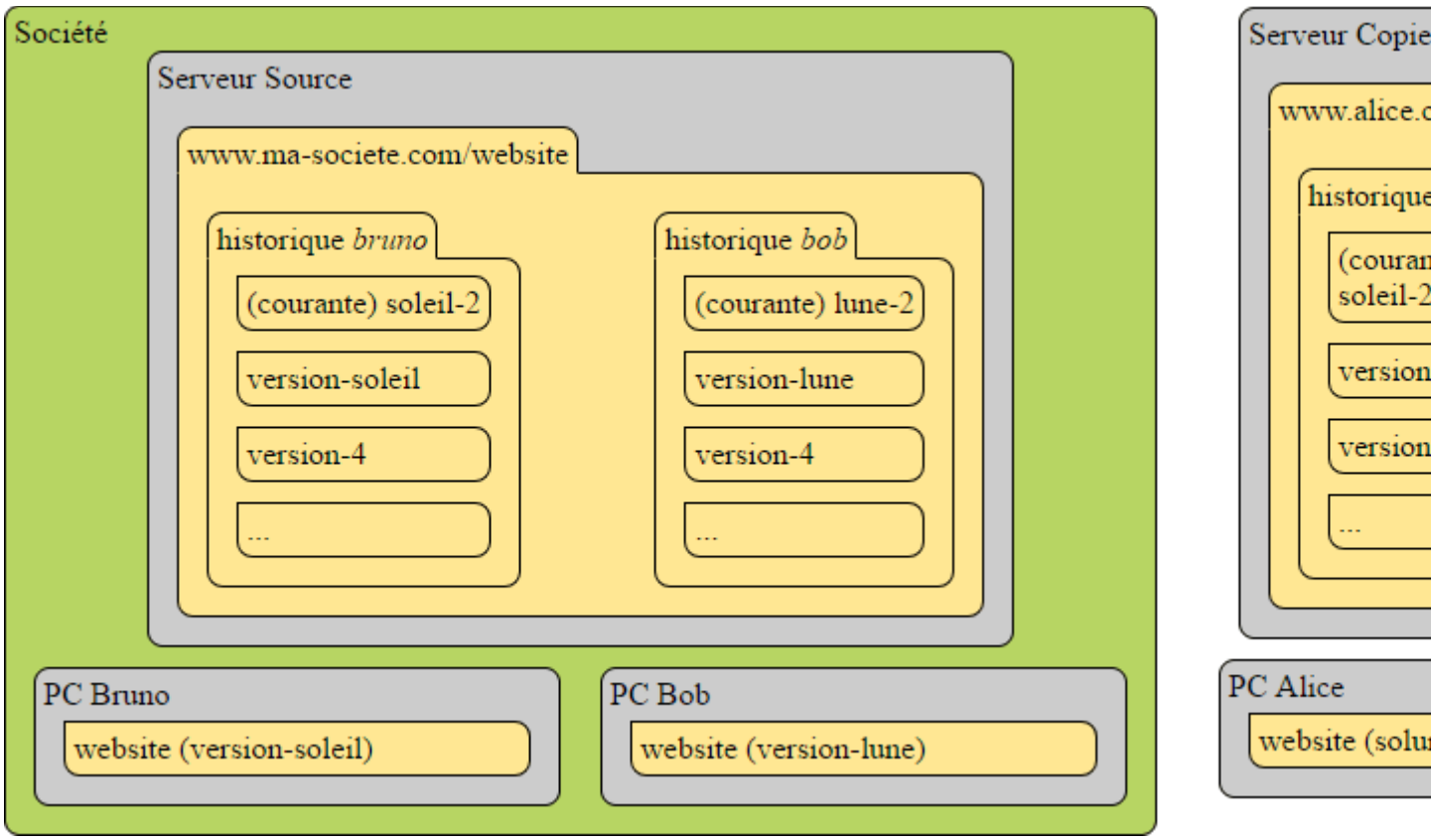
- **develop** – une branche permanente contenant l'avancement de l'application dans son ensemble au file des versions, avec des branches de fonctionnalités créées à partir de cette version et reversées dedans.
- **feature/{nom de branche}** – chaque fonctionnalité correspond au travail sur un sujet particulier (ex: une correction de bug, une nouvelle fonctionnalité...).
- **hotfix/{nom de branche}** – des branches pour s'occuper des fixes urgent à apporter qui seront prioritairement reversés dans le `master` puis reversés dans la `develop` en attendant de finir dans le prochain livrable.

Pour en savoir plus, vous pouvez lire Les Branches avec GIT. et plus particulièrement :

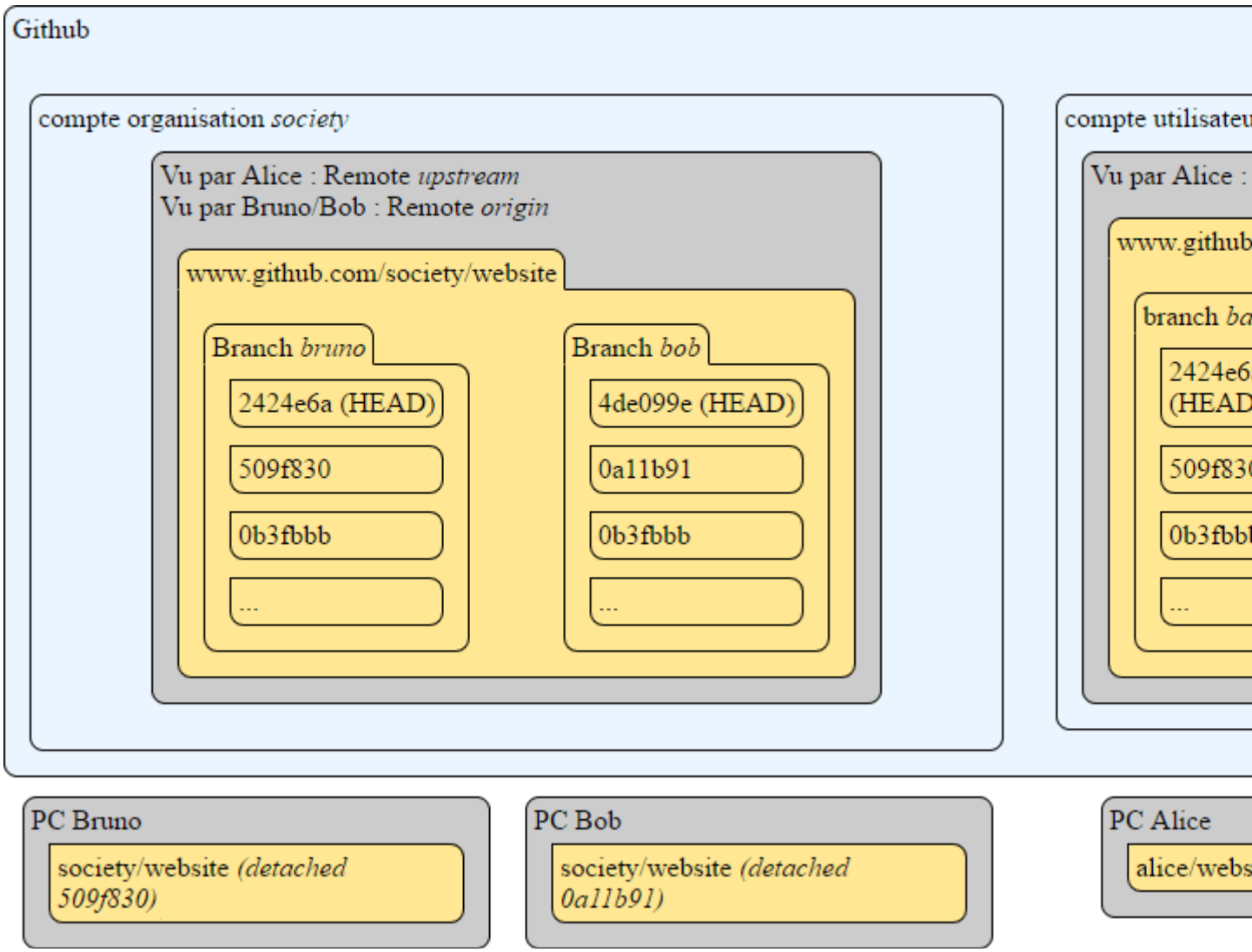
- Branches et fusions et
- Rebaser

Partagez avec le monde : bienvenue sur GitHub

Parce que mettre en place des serveurs Git vous-même est fastidieux, vous pouvez utiliser le plus grand réseau distribué de projet au monde, GitHub. Pour utiliser Git et GitHub au mieux il va être nécessaire de bien associer le vocabulaire de Git avec les exemples que nous avons précédemment décrit.



Faisons comme si tous les serveurs de notre exemple précédent était sur GitHub. Une société quelconque partage un projet sur GitHub et vous découvrez Git et GitHub en tant qu'Alice pour travailler avec le projet de cette société.



Voici ce que nous pouvons dire de l'image ci-dessus avec le vocabulaire Git(Hub) :

- Nous sommes l'utilisateur Alice.
- Sur notre PC, notre dépôt local (« local repository ») dans le dossier `website` à un espace de travail (« workspace ») qui pointe sur la révision (« commit ») `9696190` de la branche (« branch ») `solune`.
- La dernière révision de la branche `solune` est `9696190`, c'est pourquoi on peut également dire que c'est `HEAD`.
- Comme le pointeur du dossier est sur `HEAD`, cela signifie que nous sommes sur la branche `solune` en elle-même (sa version la plus à jour).
- Depuis le poste d'Alice, la source (« remote ») depuis laquelle elle a fait un clone sur son poste est `www.github.com/alice/website`.
- Un alias de `www.github.com/alice/website` pour le dépôt local d'Alice est `origin`.
- Alice a fait une réplique (« fork ») de `www.github.com/society/website` vers `www.github.com/alice/website`

- Le dépôt distant de Alice (aliasé « origin » pour son poste) possède les branches `base` et `solune`. Elle a donc ajouté/supprimé des branches après la réplication (sinon elle aurait les mêmes que celle de `www.github.com/society/website`).
- Depuis le poste d'Alice, le dépôt distant `www.github.com/society/website` a pour alias `upstream`.
- Si Alice monte (« push ») les modifications actés (« commited ») en local, une nouvelle révision va s'ajouter. Sa `HEAD` va être déplacée sur cette nouvelle révision.
- Si Alice précise le serveur `upstream` comme destination de la monté, alors sa branche `solune` sera ajoutée aux branches sur le serveur `www.github.com/society/website` et pas sur `www.github.com/alice/website`.
- Comme Bruno est sur une révision qui n'est pas une `HEAD`, il n'est pas sur une branche. Il est sur une branche détachée (« detached branch »). S'il modifie des choses ici, il devra les acter dans une `HEAD` et donc créer une nouvelle branche pour avoir une nouvelle `HEAD`.
- Si la branche `bruno` reçoit de nouvelles révisions, alors si Alice veut mettre à jour sa branche `base`, elle devra faire une récupération (« fetch ») puis une fusion sur sa `HEAD` pour créer une nouvelle révision. Il faudra ensuite acter cela sous un nouvelle révision (ce qui déplacera la `HEAD` dessus).
- Si Alice veut proposer des changements de sa branche `origin/base` à ré-introduire dans la branche `upstream/bruno` elle peut faire une proposition de fusion (« pull request »). Bruno pourra ensuite décider d'accepter ou refuser sa proposition, ou de lui demander des ajustements.

Etc.

Index de termes

Il existe encore énormément de cas de figure et voici un petit index pour mieux appréhender ceux-ci en fonction de la langue de votre interlocuteur.

- **dépôt** (repository/*repo*) : ensemble de données constituant un projet et la totalité de ses historiques de modification.
- **dépôt local** (local repository) : ensemble des historiques (`.git`) et de l'espace de travail sur une machine cliente.
- **dépôt distant** (remote repository) : ensemble des historiques sur une machine serveur.
- **révision/instantané** (commit) : entrée d'historisation créant un instantané de l'état du projet à un moment donné.
- **acter** (to commit/*commiter*) : action de créer un instantané.
- **branche** (branch) : ensemble de révisions formant un historique.
- **branche principale** (master branch) : nom donné à la branche par défaut lors de la création d'un dépôt.
- **réplique** (fork) : copie d'un dépôt distant à une autre adresse serveur. Cette copie pourra ensuite avoir des divergences avec l'original et recouper des similitudes.
- **répliquer** (to fork/*forker*) : action de faire une réplique.

- **clone** : copie d'un dépôt distant sur une machine en local. Créer le `.git` et l'espace de travail sur la révision courante.
- **cloner** (to clone) : action de faire un clone.
- **récupérer** (to fetch) : action de descendre l'état du dépôt distant dans le dépôt local `.git`.
- **fusionner** (to merge/*merger* — to rebase/*rebaser*) : action de fusionner deux branches (local ou distante) sur une machine cliente.
- **descendre** (to pull/*puller*) : action de récupérer l'état du dépôt distant et de fusionner la branche de travail avec.
- **monter** (to push/*pusher*) : action d'aligner les révisions du dépôt distant avec les derniers ajouts de révision de votre dépôt local.
- **proposition de fusion** (pull request/PR) : proposer la fusion de deux branches différentes (du même projet ou entre réplique).
- **origine** (origin) : alias attribué à la réplique depuis laquelle vous avez fait un clone.
- **référent** (upstream) : alias attribué à la réplique originale depuis laquelle la votre a été faites.
- **HEAD** : alias de la révision utilisé comme état courant d'une branche.

Avant de finir

Cette article étant également un aide mémoire, toute remarque pertinente que vous souhaiteriez y apporter peut-être discuté dans les commentaires ci-dessous ! N'hésitez pas !

Comprendre et utiliser Git pour la gestion de version

By Xavier Gouchet 17 mars 2014

.com

La plateforme de contenu, créée par des développeurs, **qui crée du lien** entre les entreprises et les développeurs à la recherche de l'entreprise où ils seront heureux.

Mentions Légales
Confidentialité
Reporter un bug

© 2016 - 2020 WeLoveDevs.com - Tous droits réservés.

Entreprises Jobs Tests Blog Je suis recruteur Inscriptio

habitués à d'autre systèmes, comme par exemple SVN.

REÇOIS EN 24H LES + BELLES OFFRES TECH MATCHANT AVEC TA RECHERCHE

Je vous propose donc de faire le point sur Git et d'en comprendre l'essentiel afin de pouvoir l'utiliser au jour le jour.

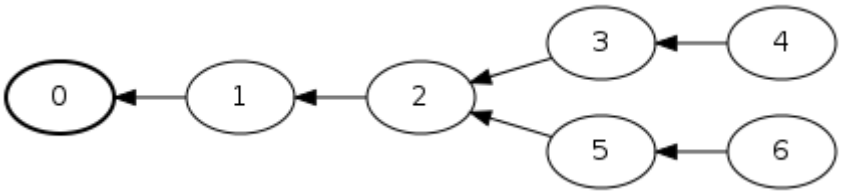
Le maillon de base : le commit

L'élément principal de Git est le commit. Tout dans Git est fait pour gérer ces objets, qui ne sont rien de plus qu'une description de l'état de chacun des fichiers dans le projet.

Un commit est identifié par un numéro unique, pas très *user friendly*, comme par exemple *1882fdda054a2c4ac783787c08c67a995872a5b5*. Un commit contient également un message écrit par le développeur pour décrire les modifications apportées, ainsi que la date et le nom de l'auteur du commit.

Un commit va également retenir l'identifiant de son (ou ses) parent(s). C'est pour cela que je parle de maillon : les commits s'organisent sous une forme chaîne, partant du commit 0 (celui présent à la création d'un projet).

branches



L'intérêt des commits, c'est que plusieurs d'entre eux peuvent avoir le même parent. Le principe des branches est né de cette fonctionnalité, et permet d'avoir plusieurs chaines de commits (les branches) partant d'un ancêtre commun.

Chaque branche permet de faire des développement parallèles, et peuvent, à un moment donné, être fusionnées à nouveau en une seule branche. Et en réalité, une branche n'est simplement qu'un pointeur vers le dernier commit dans une chaine donnée.

Le contenant : dépôt et espace de travail

Le dépôt (repository en anglais) désigne l'historique des commits, c'est à dire l'intégralité des modifications apportées aux fichiers gérés par Git.

L'espace de travail correspond lui aux dossiers et fichiers de votre projets, ceux que vous avez l'habitude de manipuler quotidiennement.

Entre les deux va se trouver l'index, un registre qui va garder en mémoire le

Entreprises Jobs Tests Blog Je suis recruteur Inscriptio

REÇOIS EN 24H LES + BELLES OFFRES TECH MATCHANT AVEC TA RECHERCHE

Un exemple concret

Simplement avec ces notions, il est possible de travailler en local, et de gérer les changements apportés à des fichiers. Je m'en vais donc vous proposer un exemple concret d'utilisation de git.

Pour initialiser un dépôt dans un espace de travail, utilisez la ligne de commande suivante :

```
$ git init
Initialized empty Git repository in /home/xgouchet/Perso/workspac
```

Pour connaitre l'état de l'espace de travail et de l'index, utilisez la commande status :

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add ..." to include in what will be committed)
#
#       Main.java
#       Pojo.java
nothing added to commit but untracked files present (use "git
```

commit initial (sans aucun parent), et que deux fichiers ont été modifiés dans l'espace de travail sans être ajoutés à l'index.

Remedions à cela et ajoutons notre main à l'index, comme ceci :

```
$ git add Main.java

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached ..." to unstage)
#
#       new file:   Main.java
#
# Untracked files:
#   (use "git add ..." to include in what will be committed)
#
#       Pojo.java
```

Le fichier `Main.java` a bien été ajouté à la liste des modifications à prendre en compte pour le prochain commit. Continuons en faisant ce commit :

```
$ git commit -m "Mon premier commit"
[master (root-commit) ac3b917] Mon premier commit
 0 files changed
 create mode 100644 Main.java
```

L'argument `-m` permet de spécifier le message associé au commit. Faisons

```
 $ git status
# On branch master
# Untracked files:
#   (use "git add ..." to include in what will be committed)
#
#       Pojo.java
nothing added to commit but untracked files present (use "git add'
```

Nous pouvons également créer une nouvelle branche à partir du dernier commit. La commande `git branch` permet de créer cette branche, puis `git checkout` change l'espace de travail pour refléter l'état de la branche désiré.

```
$ git branch develop

$ git checkout develop
Switched to branch 'develop'
```

Enfin la commande `git log` permet de visualiser l'historique de commit sur la branche actuelle.

REÇOIS EN 24H LES + BELLES OFFRES TECH MATCHANT AVEC TA RECHERCHE

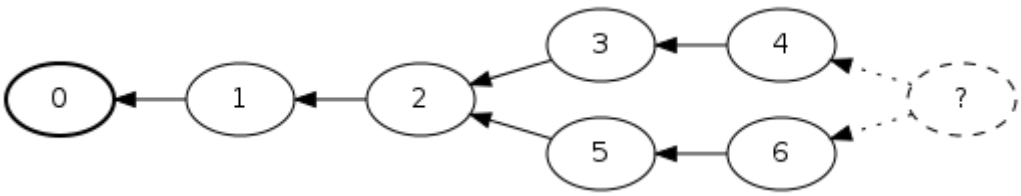
Un repo, deux repo, trois repo, ...

Pour l'instant nous avons vu le cas d'un dépôt autonome, mais l'intérêt de Gi

Il est possible de définir dans un dépôt une liste de dépôts distants. Puis il est possible de synchroniser les commits entre les dépots distants et le dépot local. Le plus important à savoir est que lors de la synchronisation, qu'elle soit montante (push) ou descendante (pull), les deux dépôts doivent avoir un commit parent en commun.

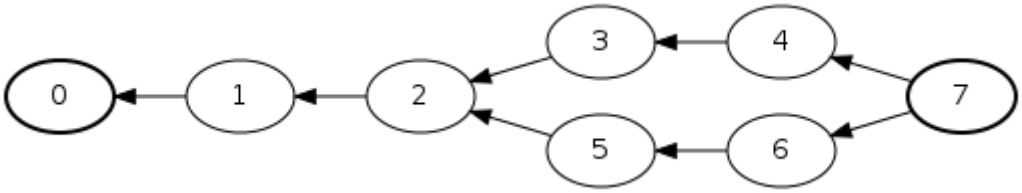
Gestion des conflits

Dans un monde idéal, tout se passe bien, malheureusement il arrive que des conflits arrive. Dans l'exemple ci dessous, deux développeurs sont partis du même commit (le 2). L'un d'entre eux à créé les commits 3 et 4, qu'il a envoyé sur le serveur commun. Le second a créé les commits 5 et 6 et au moment d'envoyer lui aussi ses modifications reçoit un message d'erreur. En effet son historique n'est plus synchrone avec celui du repo distant.



Si les commits du second developeut ne touchent pas aux mêmes fichiers que le premier développeur (ou si ils modifient le même fichier mais à des endroits différents), alors il est possible d'avoir recours à un rebase. Le rebase permet de changer le parent du commit 5 pour qu'il pointe vers le commit 4.

Par contre si les commits touchent aux même parties du code, alors il est nécessaire de fusionner les deux chaines de commits à la main.



Un exemple concret (bis)

Lorsque l'on veut travailler avec un dépôt distant, il existe plusieurs façon de faire. Il est possible de créer un clone d'un dépôt distant existant, ce qui va créer le dépôt local :

```
$ git clone https://website.tld/path/to/remote/repo.git
Cloning into 'repo'...
remote: Reusing existing pack: 318, done.
remote: Total 318 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (318/318), 630.47 KiB | 299 KiB/s, done.
Resolving deltas: 100% (103/103), done.

$ cd repo

$ git status
# On branch master
nothing to commit (working directory clean)
```

un nom l'identifiant (par convention, le dépôt distant principal s'appelle origin, mais il est possible d'avoir plusieurs remote).

```
$ git remote add origin https://website.tld/path/to/remote/repo.g
```

Une fois notre dépôt lié à un autre dépôt, il y a principalement trois commandes à connaitre. La plus importante, git fetch, permet de mettre à jour le dépôt afin qu'il connaissent l'état du dépôt distant.

```
$ git fetch
remote: Reusing existing pack: 318, done.
remote: Total 318 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (318/318), 630.47 KiB | 61 KiB/s, done.
Resolving deltas: 100% (103/103), done.
From website.tld/path/to/remote/repo
 * [new branch]      master     -> origin/master
```

Ici le fetch a détecté l'existance d'une branche master sur le dépôt distant (origin) et a créé la branche correspondante en locale.

La commande git pull permet de mettre à jour la branche courante locale à partir de celle présente sur le dépôt distant.

Ici deux cas de figure sont illustrés : dans le premier cas, je n'ai pas de commit en local, et la commande applique ce que l'on appelle un `fast forward`, c'est-à dire que les nouveaux commits sur le dépôt distant vont venir s'ajouter à mo index, rendant les deux dépôts synchrones.

```
$ git pull --rebase origin master
```

Dans le second cas, j'ai déjà des commits en locals, ce qui pourrait causer un conflit. le fait d'ajouter `--rebase` indique à git qu'il doit tenter de faire un rebase, c'est à dire qu'il va défaire mes commits en local, appliquer les commits distants puis tenter d'appliquer à nouveau les modifications que j'avais en local.

```
$ git pull --rebase origin master
First, rewinding head to replay your work on top of it...
Applying: Initial commit
```

Enfin, git push au contraire ajoute nos nouveaux commits à la branche correspondante sur le dépôt distant. Pour faire un push propre il est toujours nécessaire de faire un pull avant pour être sur qu'il n'y aura pas de conflit sur le dépôt distant. En effet, si un conflit arrive il n'y aura personne pour le résoudre sur la machine distante.

```
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 531 bytes, done.
Total 5 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4)
remote: Processing changes: refs: 1, done
To website.tld/path/to/remote/repo
   1037995..731a007  master -> master
```

Conclusion

fonctionnalités sans rentrer en profondeur. Le plus important est de comprendre les concepts de commit, dépôt et index, le reste viendra au fur et à mesure de l'utilisation de l'outil.

Xavier Gouchet (@xgouchet)

REÇOIS EN 24H LES + BELLES OFFRES TECH MATCHANT AVEC TA RECHERCHE

Xavier

Geek, Android-ophile, curieux de nature et aimant partager mes connaissances, je baigne dans le monde de la programmation depuis un bon moment, et aujourd'hui je suis principalement porté sur les technologies mobiles, et principalement Android.

J'apprécie autant découvrir de nouvelles techniques, développer des bouts de codes pour générer des images, échanger sur divers sujets techniques, que partager mes connaissances par quelque moyen que ce soit.
Aujourd'hui, je passe mon temps entre Deezer, où je fais

Entreprises Jobs Tests Blog Je suis recruteur Inscriptio

entre ma campagne percheronne et Paris.

En savoir +

Vous souhaitez devenir blogueur pour JobProd ?

Previous Post

ncipes d'architecture d'applis Web Java/JEE

À la une

Articles tech

Technique et Ingénierie

Docker, comment ca marche?

Jérémy Mathon
12 juin 2020

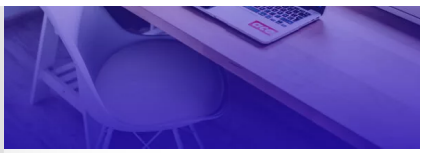
À la une

Articles tech

Technique et Ingénierie

Télécharge moi un PDF [QIMA]

Cyril Lakech
9 juin 2020



À la une

Technique et Ingénierie

Deploy Temporary Acceptance Systems on AWS

Marcy Charollois
8 juin 2020

Join the discussion

2 Comments

Vilain Mamuth
23 mars 2014 à 10 h 37 min

Répondre

Merci pour ces précisions, je comprend mieux pourquoi un commit n'implique pas automatiquement une mise à jour du dépôt distant

Issac

27 juin 2016 à 10 h 03 min

Répondre

Bonjour. Ce tutoriel est très réussi. Moi, j'ai réussi à comprendre les fonctionnalités avancées du Git grâce à des vidéos sur http://www.alphorm.com/tutoriel/formation-en-ligne-git-fonctionnalites-avancees. En tout cas, merci pour ce partage.

Leave a Reply

Name *

Email *

Website

Submit Comment

By Xavier Gouchet 17 mars 2014

.com

La plateforme de contenu, créée par des développeurs, **qui crée du lien** entre les entreprises et les développeurs à la recherche de l'entreprise où ils seront heureux.

© 2016 - 2020 WeLoveDevs.com - Tous droits réservés.

Miximum

Le blog d'un ingénieur Web freelance.

👉 Découvrez mon nouveau projet : Mamie-note.fr, un cours de théorie musicale complet, accessible et pas barbant. 🎵

Enfin comprendre Git : le tutoriel complet

July 9, 2013 Tutoriel

J'utilise Git quotidiennement depuis plus de dix ans. Bien que Git soit un outil extrêmement puissant, il n'est pas très intuitif. Sans bien comprendre les mécanismes internes du logiciel, on se retrouve vite coincé. Par conséquent, voici un tutoriel ulta-détaillé pour bien appréhender les principes et les principales commandes de Git.

Ce billet fait partie d'une série d'articles sur Git. Qu'est-ce que Git et comment s'en servir ? Quelles sont les commandes les plus utiles ? Comment éviter les pièges courants ? Quelles astuces pour gagner du temps ?

- Découvrir Git : introduction et premiers pas
- ➔ Enfin comprendre Git : le tutoriel complet
- Git rebase : qu'est-ce que c'est ? Comment s'en servir ?
- Débusquer une régression avec git bisect
- Activer la coloration avec git
- Ignorer des fichiers avec git
- Créer un patch avec git
- Utiliser git-svn pour interfacer Git avec un depot subversion
- Utiliser Git pour travailler sur un dépot CVS

Je dois pourtant reconnaître que Git n'est pas forcément l'outil le plus abordable qui soit. Toutes ces commandes bizarres ! Toutes ces options apparemment redondantes ! Cette documentation cryptique ! Et ce worflow de travail, qui nécessite 18 étapes pour pousser un patch sur le serveur. Tel un fier et farouche étalon des steppes sauvages, Git ne se laissera approcher qu'avec cisconspection, et demandera beaucoup de patience avant de s'avouer dompté.

Tenez, prenez l'exemple suivant :

Comment j'annule une modification d'un fichier ?

```
git checkout
```

Ok, comment je change de branche ?

```
git checkout
```

Ok, et comment je créé une nouvelle branche ?

```
git checkout
```

Mmm... Ok, et comment je supprime une branche ?

```
git branch -d ma_branche
```

D'accord, et comment je supprime une branche distante ?

```
git push origin :ma_branche
```

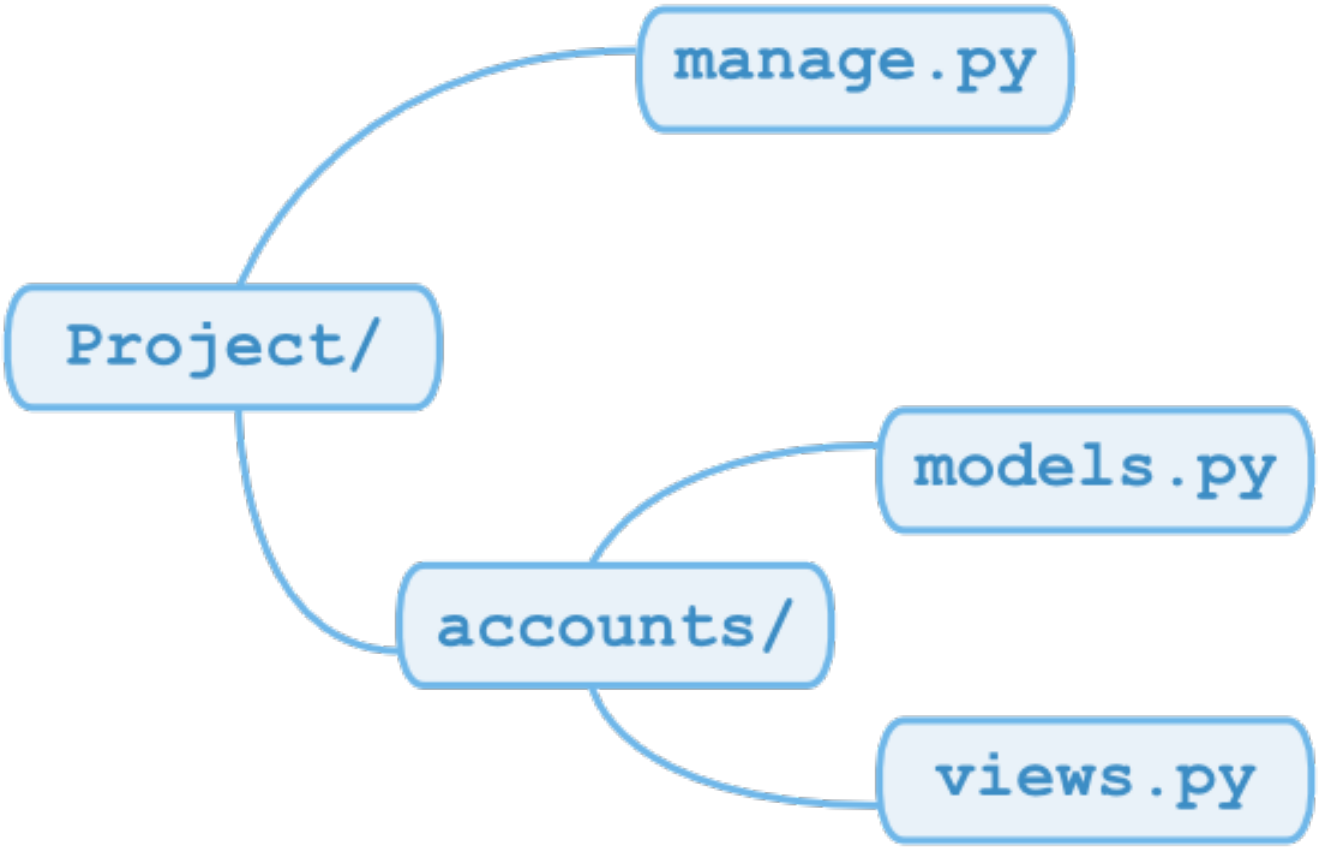
Heu...

À vrai dire, il est trés difficile de percevoir la logique de ce qu'on fait si on n'a pas un minimum de compréhension du fonctionnement interne de Git. De plus, la plupart des commandes sont plutôt bas niveau, ce qui explique leurs effets qui peuvent paraître sans rapports entre eux.

Tâchons de comprendre un peu mieux la bête pour mieux la maîtriser.

Comprendre les zones et le workflow de travail

Imaginons que vous soyez en train de travailler sur un quelconque projet, constitués d'une arborescence de fichiers tout à fait classique.

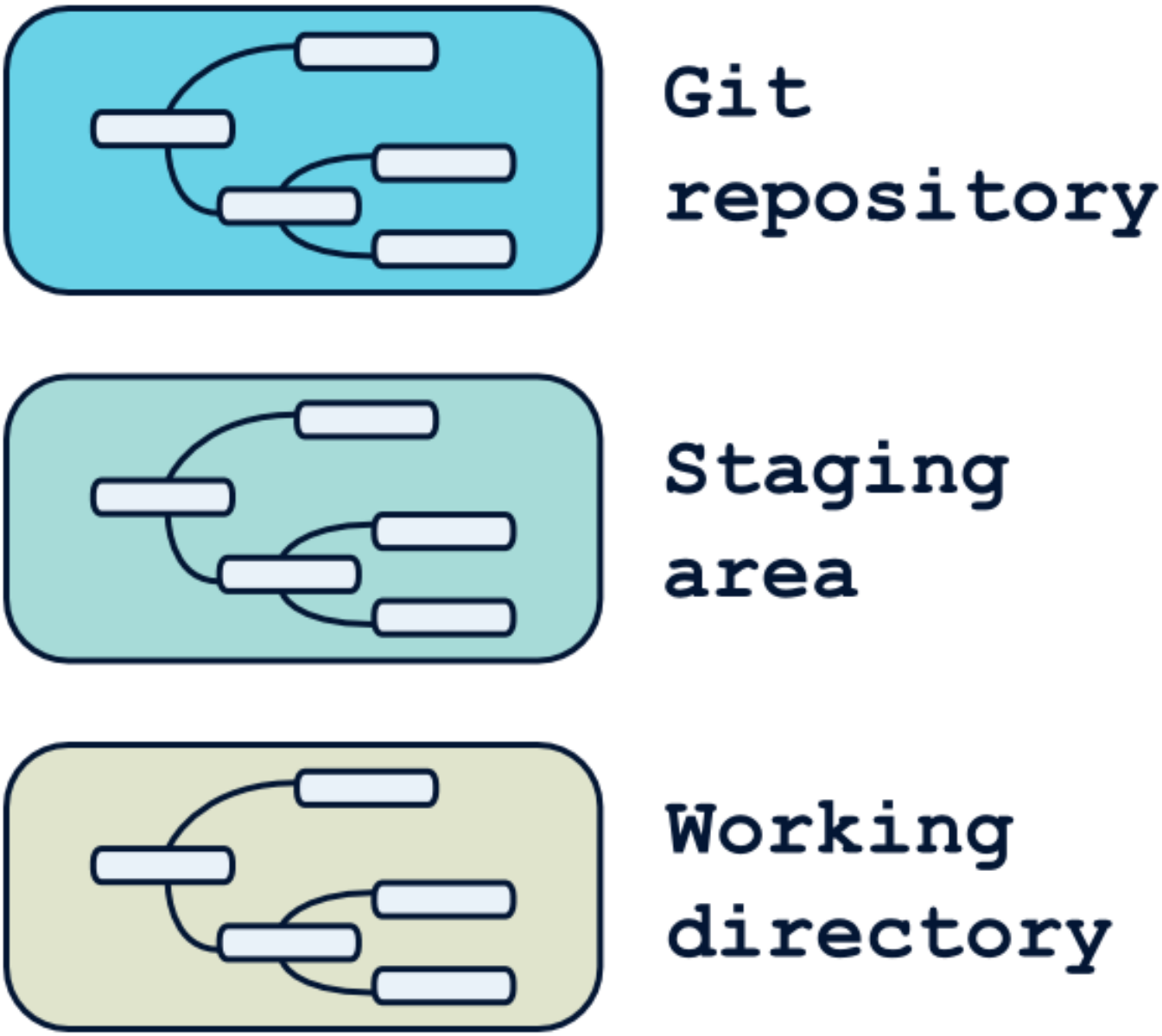


Si votre projet est géré avec Git, on peut grosso-modo considérer que votre arborescence n'est pas stockée une, mais trois fois.

D'abord, les fichiers eux-mêmes sur votre disque dur, que vous éditez grâce à votre éditeur préféré. C'est votre répertoire de travail, ou *Working Directory* en angliche.

Ensuite, dans une mystérieuse zone spéciale que l'on appelle l'index, ou la zone de staging.

Enfin, dans la base de données de Git est stockée l'arborescence de votre projet telle qu'elle était lors de votre dernier commit.



Pourquoi trois zones, et pas seulement deux ? Quelle est donc cette mystèrieuse « staging area » ?

Qu'est-ce qu'un bon commit ?

Laissez-moi digresser quelque peu pour rappeler qu'un bon commit est un commit *atomique* :

- il ne concerne qu'une chose et une seule ;
- il est le plus petit possible (tout en restant cohérent).

Pourtant, lorsqu'on travaille sur une fonctionnalité, il n'est pas rare d'en profiter pour corriger une petite faute d'orthographe par ci, un petit bug qui trénouillait par là, et on se retrouve avec un répertoire de travail contenant des modifications totalement indépendantes. Ces modifications doivent alors faire l'objet de commits séparés, et c'est à ça que sert la zone de staging : préparer le prochain commit, en y ajoutant ou retirant des fichiers (ou portions de fichiers) sans toucher à votre répertoire de travail.

Certains y verront sans doute un travail superflu bon à satisfaire les instincts pervers des aficionados d'attouchements intimes sur les diptères. Il n'en est rien, et une fois qu'on y a goûté, il est tout simplement impossible de revenir en arrière.

Commandes de base

Le processus de commit avec Git est donc celui-ci :

1. je développe en modifiant / déplaçant / supprimant des fichiers ;
2. quand une série de modification est cohérente et digne d'être commitée, je la place dans la zone de staging ;
3. je vérifie que l'état de ma zone de staging est satisfaisant ;
4. je committe ;
5. et on répète jusqu'à... euh... ben, la fin quoi.

Pour copier un fichier du répertoire de travail vers la zone de staging, on utilise *git add*.

Pour sauvegarder la zone de staging dans le dépôt git et créer un nouveau commit, on utilise *git commit*.

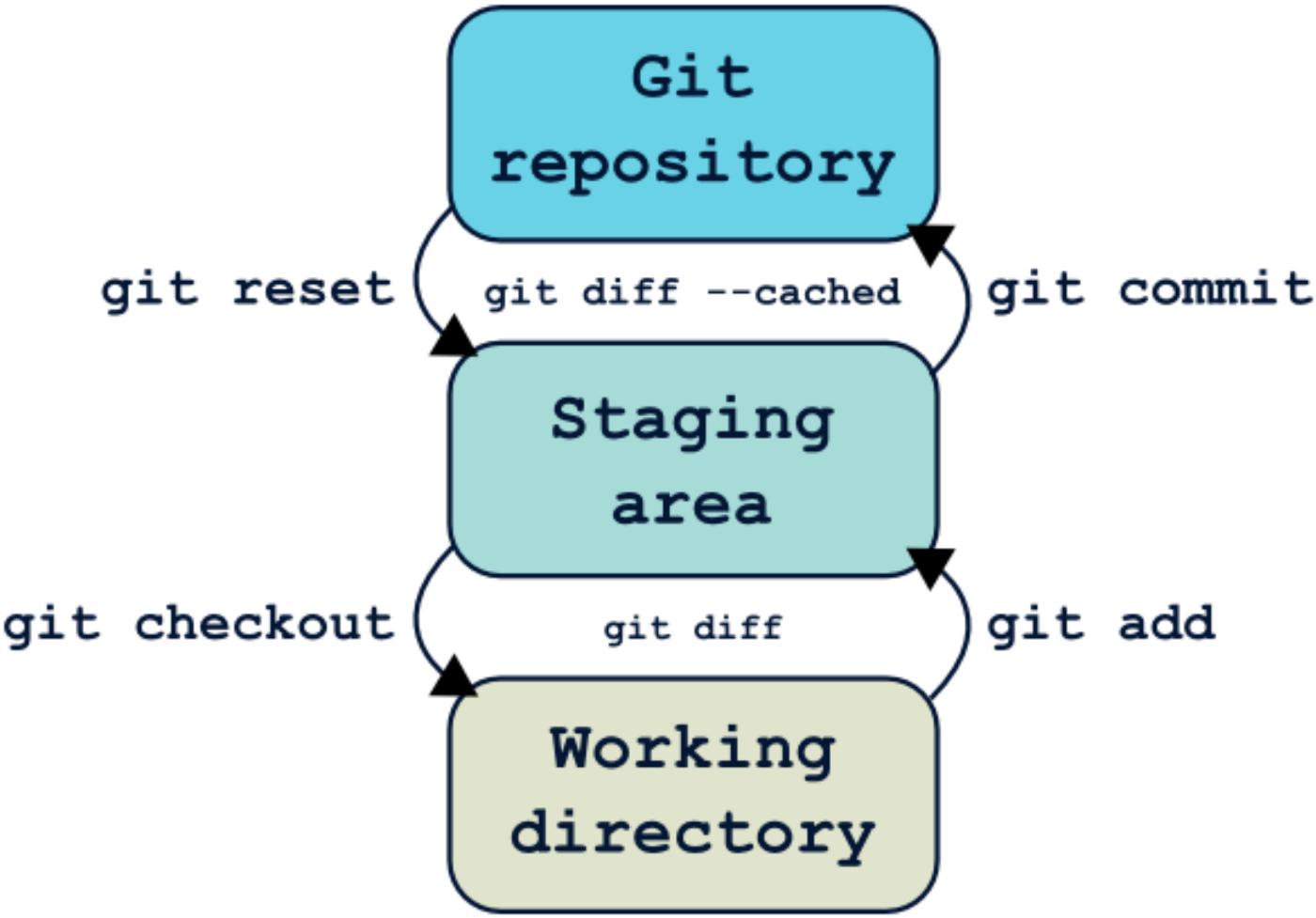
Pour copier un fichier du dépôt Git vers la zone de staging, on utilise *git reset*.

Pour copier un fichier du staging vers le working directory (donc supprimer les

modifications en cours), on utilise *git checkout*.

Pour visualiser les modifications entre le répertoire de travail et la zone de staging, on utilise *git diff*.

Pour visualiser les modifications entre la zone de staging et le dernier commit, on utilise *git diff --cached*.



Et comment sait-on quels fichiers sont différents d'une zone à l'autre ? Grâce à *git status* :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD ..." to unstage)
#
#   modified:   accounts/forms.py
#   modified:   accounts/models.py
#
# Changes not staged for commit:
#   (use "git add ..." to update what will be committed)
#   (use "git checkout -- ..." to discard changes in working d
#
#   modified:   accounts/urls.py
#   modified:   accounts/views.py
#
```

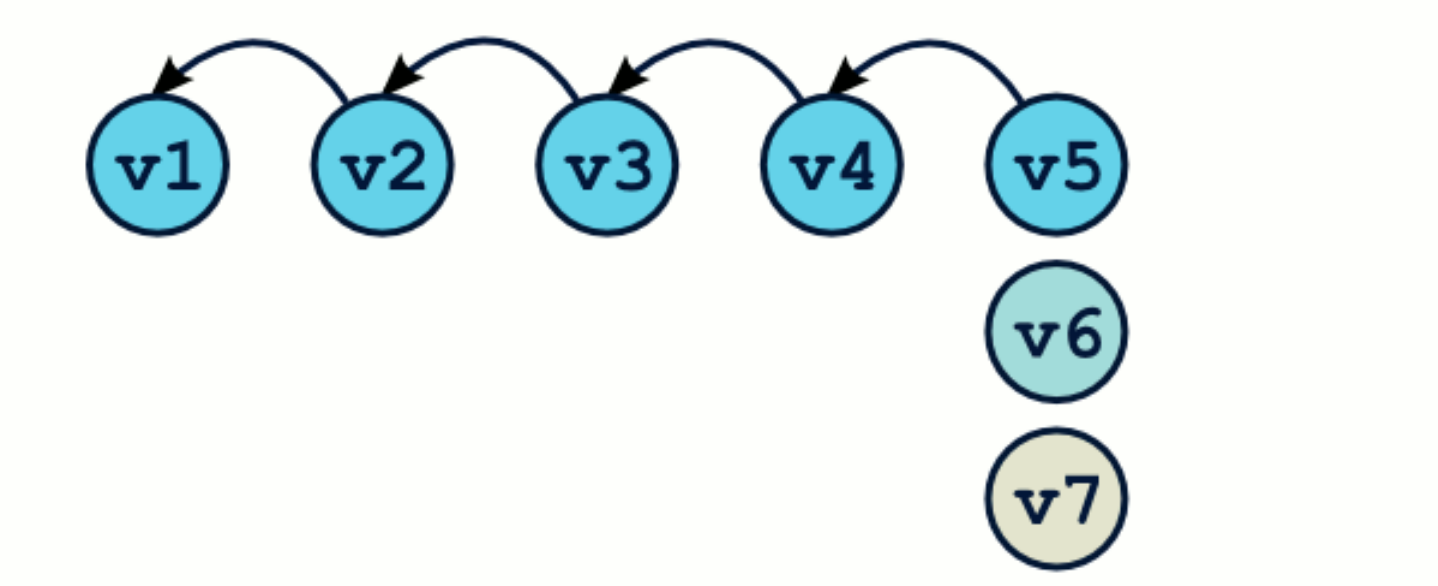
Un arbre de commits

Tout ce pataquès juste pour un commit ? Oui ! Ça peut paraître beaucoup de travail aux habitués de svn (note : ce billet est assez ancien, il a été écrit à une époque ou Git n'était pas encore hégémonique), mais j'affirme que, qu'il travaille seul ou en équipe, un développeur se doit de traiter son historique de projet avec soin, et de s'assurer de la propreté de chaque commit.

Maintenant, que se passe-t-il lorsque les commits s'enchaînent ? En fait, il suffit de comprendre qu'un commit n'est rien d'autre qu'une structure qui contient :

- des méta-données (auteur, date, etc.) ;
- une référence vers un (ou plusieurs) commit parent ;
- une copie de l'arborescence du projet au moment du commit.

Créer un commit, c'est donc rajouter une entrée dans la base de données Git. Et oui, vous avez bien lu : à chaque commit, Git stocke (de manière compressée) l'intégralité des fichiers qui ont été modifiés depuis le commit précédent. Reconstruire le projet à un instant T à partir de l'historique est donc rapide comme l'éclair.



Avec Git, l'historique du projet n'est ni plus ni moins qu'un graphe, qu'on pourra fouiller grâce à la commande *git log*.

```
$ git log
commit 67d6a5214ad4b259407ec7836b9d729f9f7de731
Author: Thibault Jouannic
Date:   Fri Jul 5 10:45:50 2013 +0200

    create reminders admin module

commit 05ca141b9e982c7d04100c37300da4209305b900
Author: Thibault Jouannic
Date:   Fri Jul 5 10:29:56 2013 +0200

    Create user admin module

commit 0fb74654c708a01bfaec8d552437e9f655bd325d
Author: Thibault Jouannic
Date:   Thu Jul 4 15:46:34 2013 +0200

    upgrade pymill version

…
```

Comprendre les branches

Sous d'autres systèmes comme svn, la gestion des branches est souvent pénible et laborieuse, ce qui décourage les développeurs qui ne les utilisent que trés occasionnellement (pour ne pas dire jamais). Avec Git, travailler avec des branches est un tel plaisir qu'on aurait tort de ne pas les utiliser. En fait, les branches sont une fonctionnalité basique, pas un truc « avancé » comme je l'entends parfois.

On utilise les branches tout le temps, ou presque. Pour tester une fonctionnalité ; pour isoler un développement un peu long ; pour mettre quelques commits de côté ; pour développer sans péter la branche principale ; pour corriger un bug sans impacter le développement d'une fonctionnalité parallèle. Bref ! il y a pleins de raisons d'utiliser les branches.

Qu'est-ce qu'une branche ? Attention, accrochez-vous, la définition qui va suivre est difficile à comprendre du premier coup.

Une branche n'est qu'une étiquette qui pointe vers un commit.

Quoi ?! C'est tout ? Et oui ! Si vous ne me croyez pas, tapez la commande suivante à la racine d'un dépôt git :

```
$ cat .git/refs/heads/master
67d6a5214ad4b259407ec7836b9d729f9f7de731
```

La branche master (par défaut la branche principale et seule branche d'un dépôt) n'est qu'une étiquette qui pointe vers un commit. C'est un simple fichier qui ne contient rien d'autre que l'identifiant (un hash SHA1) d'un commit. Dans le jargon Git, cette notion de nom référençant un commit s'appelle une « référence ». Un autre exemple de référence nous est donné par les tags.

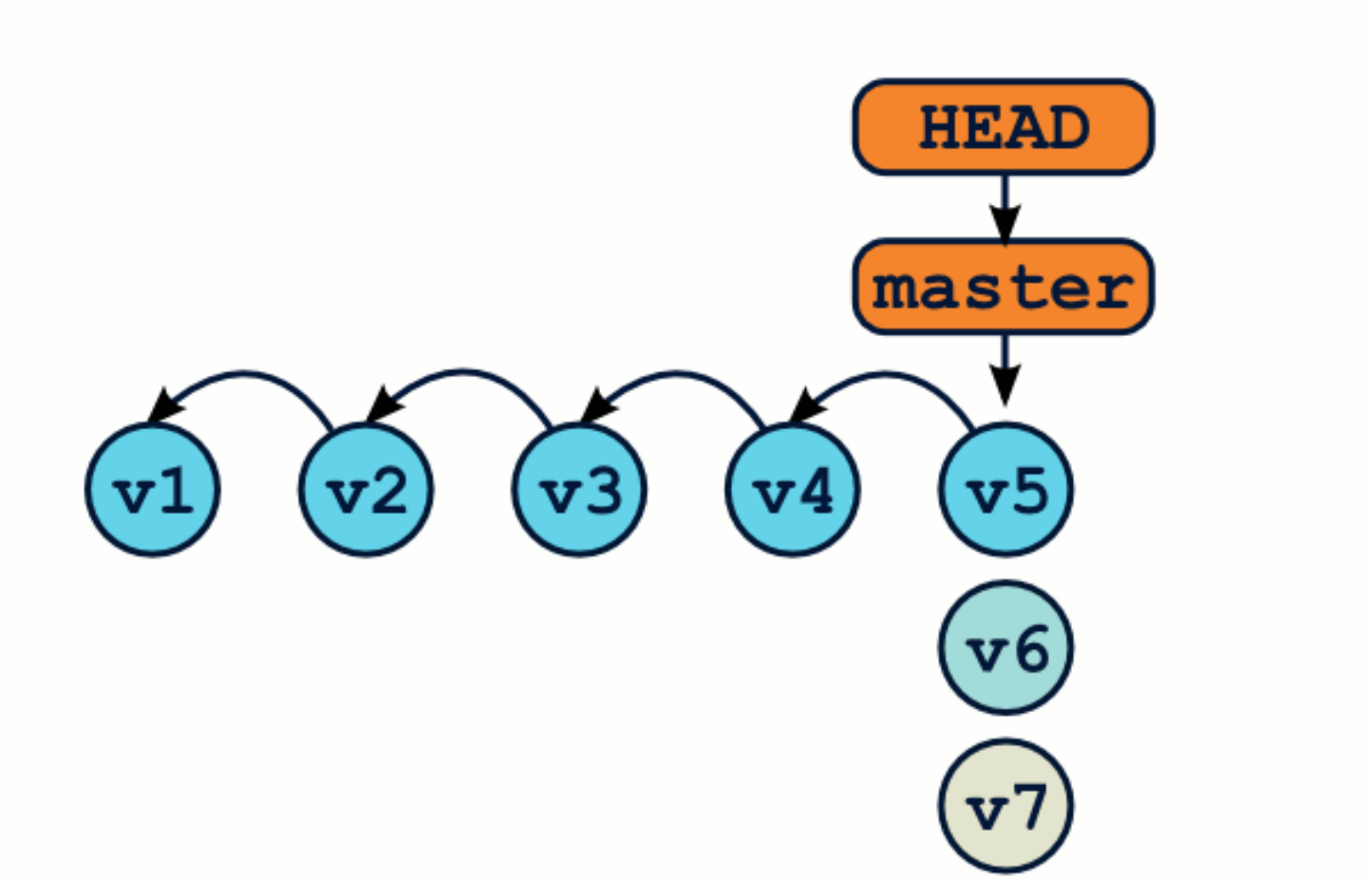
Avec Git, il existe une référence spéciale qui s'appelle *HEAD*. La référence HEAD pointe vers le commit qui sera le parent du prochain commit. C'est clair ? Non ? Vous allez comprendre à la prochaine illustration.

La plupart du temps, HEAD ne pointe pas directement vers un identifiant de commit, mais plutôt vers une branche, e.g *master*.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Quand vous créez un nouveau commit, il se passe ceci :

1. un nouvel objet commit est créé, avec pour parent le commit pointé par HEAD ;
2. la branche pointée par HEAD pointe maintenant vers ce nouveau commit ;
3. et c'est tout.



Manipuler les branches

On va principalement manipuler les branches grâce à deux commandes :

git branch permet de créer, lister et supprimer des branches.

git checkout permet de déplacer la référence HEAD, notamment vers une nouvelle branche.

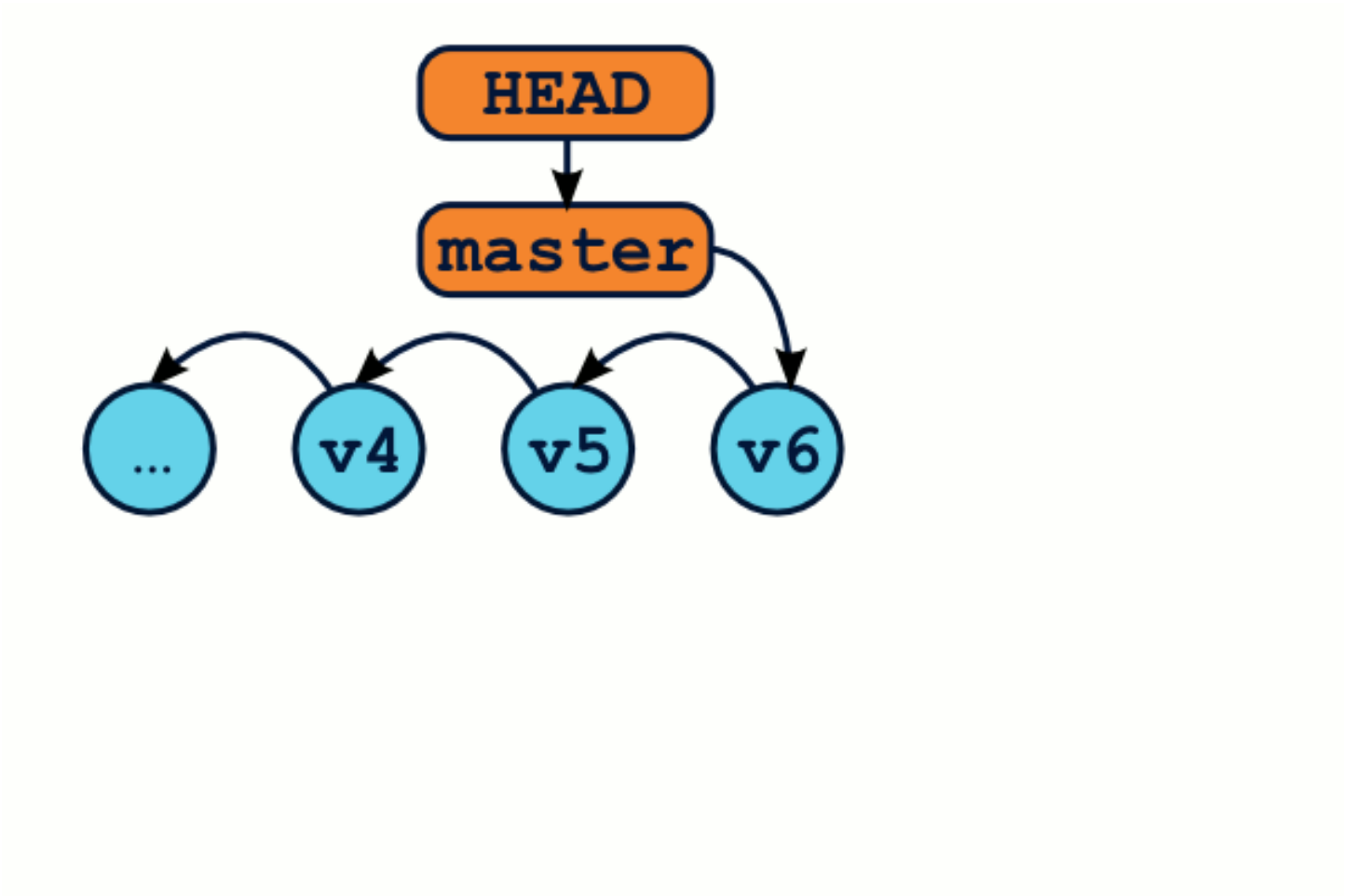
Créer une branche

Voici de manière schématique comment on créé une branche :

```
$ git branch test  # créé la branche "test"
$ git checkout test  # Déplace HEAD sur "test"
```

Notez qu'en général, on utilise le raccourci suivant, qui est trés exactement équivalent aux deux commandes précédentes :

```
$ git checkout -b test
```



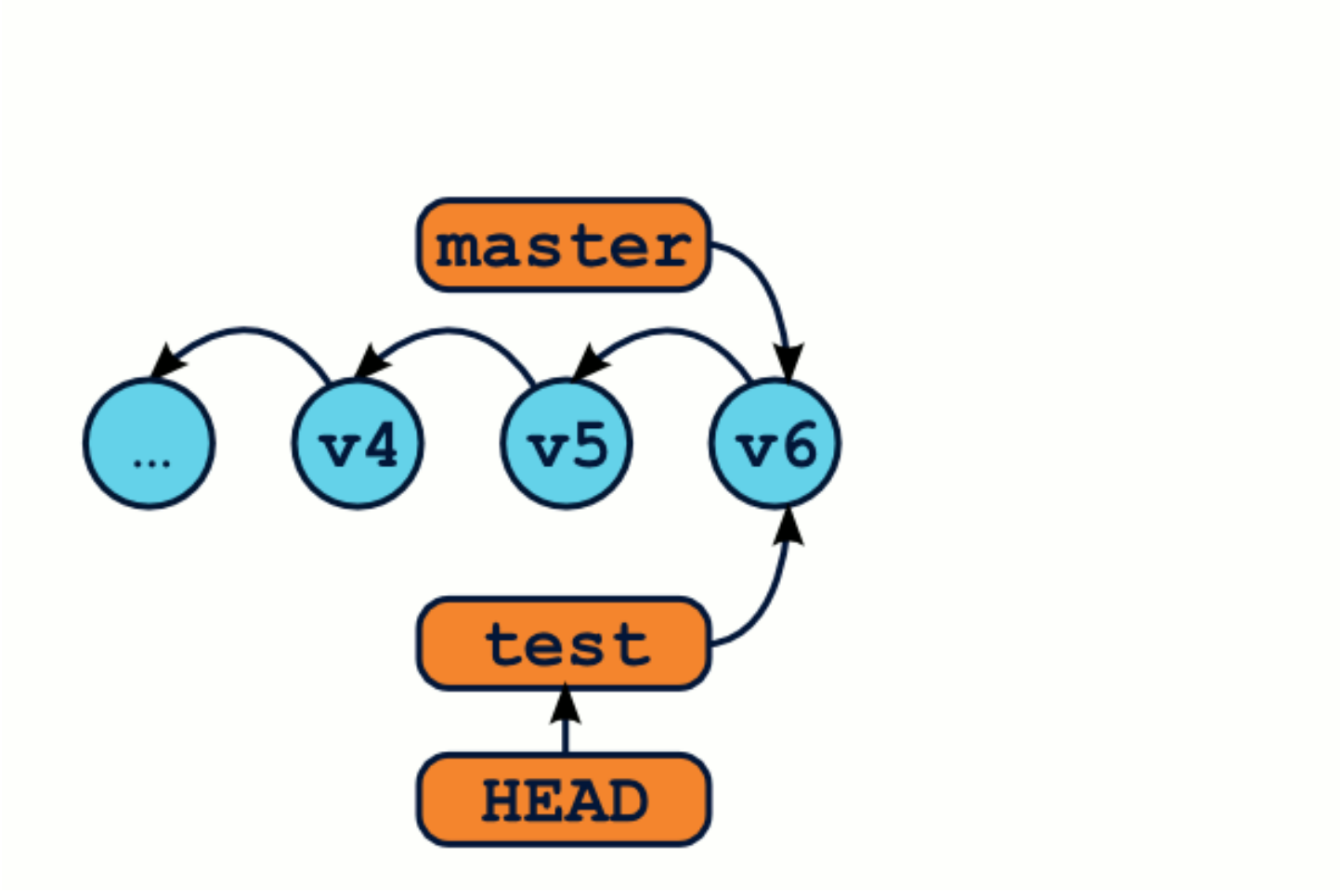
Travailler sur des branches

Créer des branches va nous permettre de créer des flots de développement parallèles. Chaque commit sera ajouté à la chaîne des commits de la branche courante. On connaît la branche courante grâce à la commande *git branch*.

```
$ git branch
* master
  test
```

Voici un workflow de travail classique, ainsi que les commandes associées :

```
$ # Je me trouve actuellement sur la branche "test"
$ git commit …  # Je travaille sur ma branche en créant des co
$ git checkout master  # Retournons sur la branche master
$ git commit …  # je travaille sur ma branche master
```



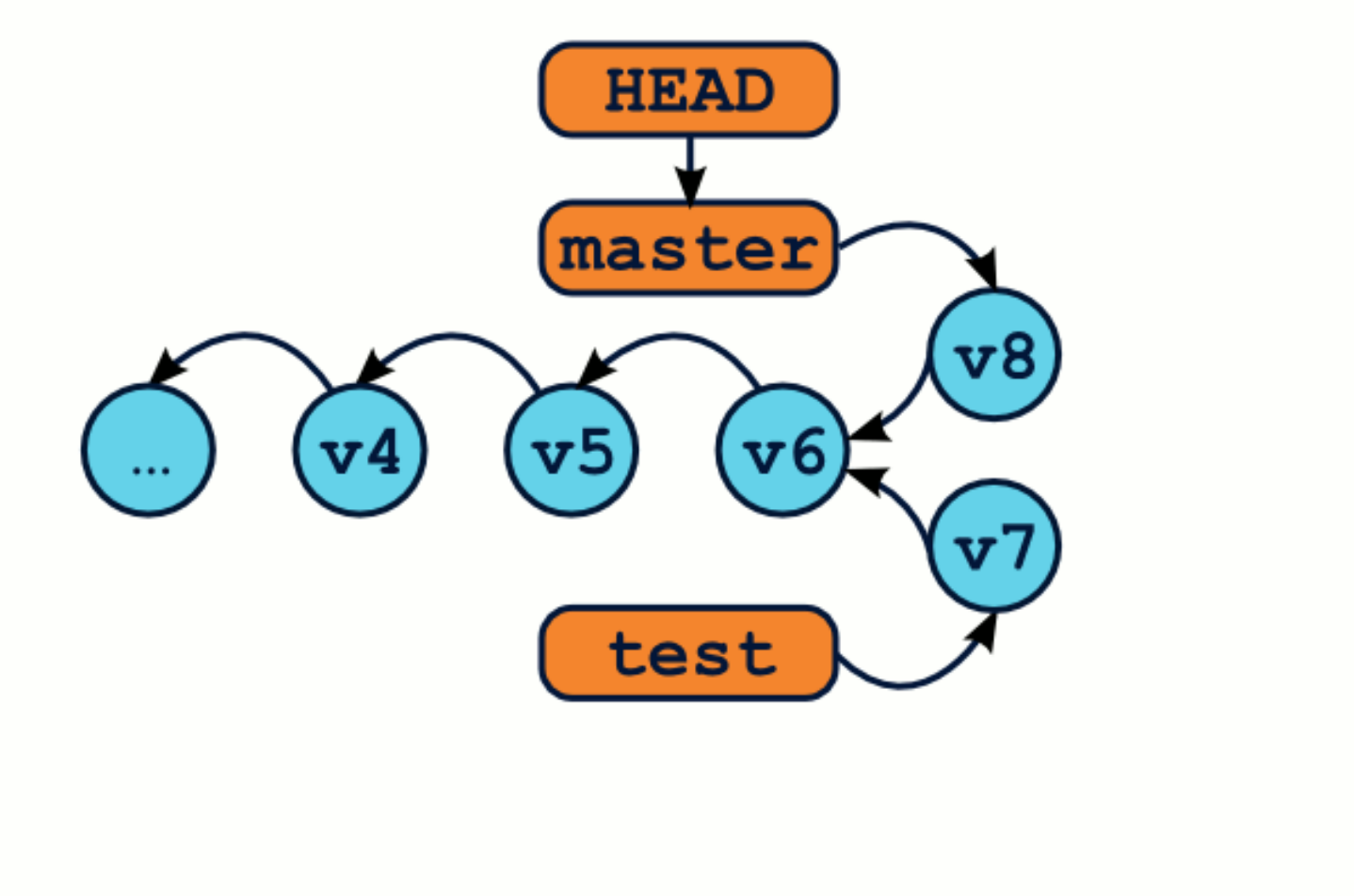
Fusionner des branches

Avoir des branches divergeantes, c'est bien beau, mais il faudra bien fusionner toutes ces modifications dans une seule et même arborescence, n'est-ce pas ? C'est à ça que servent les fusions, ou *merge* dans le jargon.

Lorsqu'on fusionne deux branches, Git va intégrer toutes les modifications contenues sur chaque branche dans une seule et même arborescence. Il va alors créer un commit qui aura deux parents. Une délégation de l'UMP aurait réclamé

l'obligation pour chaque commit de disposer d'un papa et d'une maman, sans effet jusqu'à présent.

```
$ git checkout master  # On se place sur la branche qui va "re
$ git merge test  # FuuuuuuuuuuuuSion !
$ git branch -d test  # Une fois fusionnée, notre branche ne s
```



Résoudre les conflits de fusion

Il peut arriver que la fusion soit impossible à réaliser sans intervention manuelle. Ce sera le cas si chaque branche apporte des modifications différentes au même endroit dans le même fichier. On dit qu'il y a conflit.

Dans ce cas là, Git interrompt le merge et insère des marqueurs dans les fichiers

conflictuels. Il nous faudra éditer ces fichiers manuellement (ou à l'aide d'une interface spécifique), avant de poursuivre la fusion.

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodifie
```

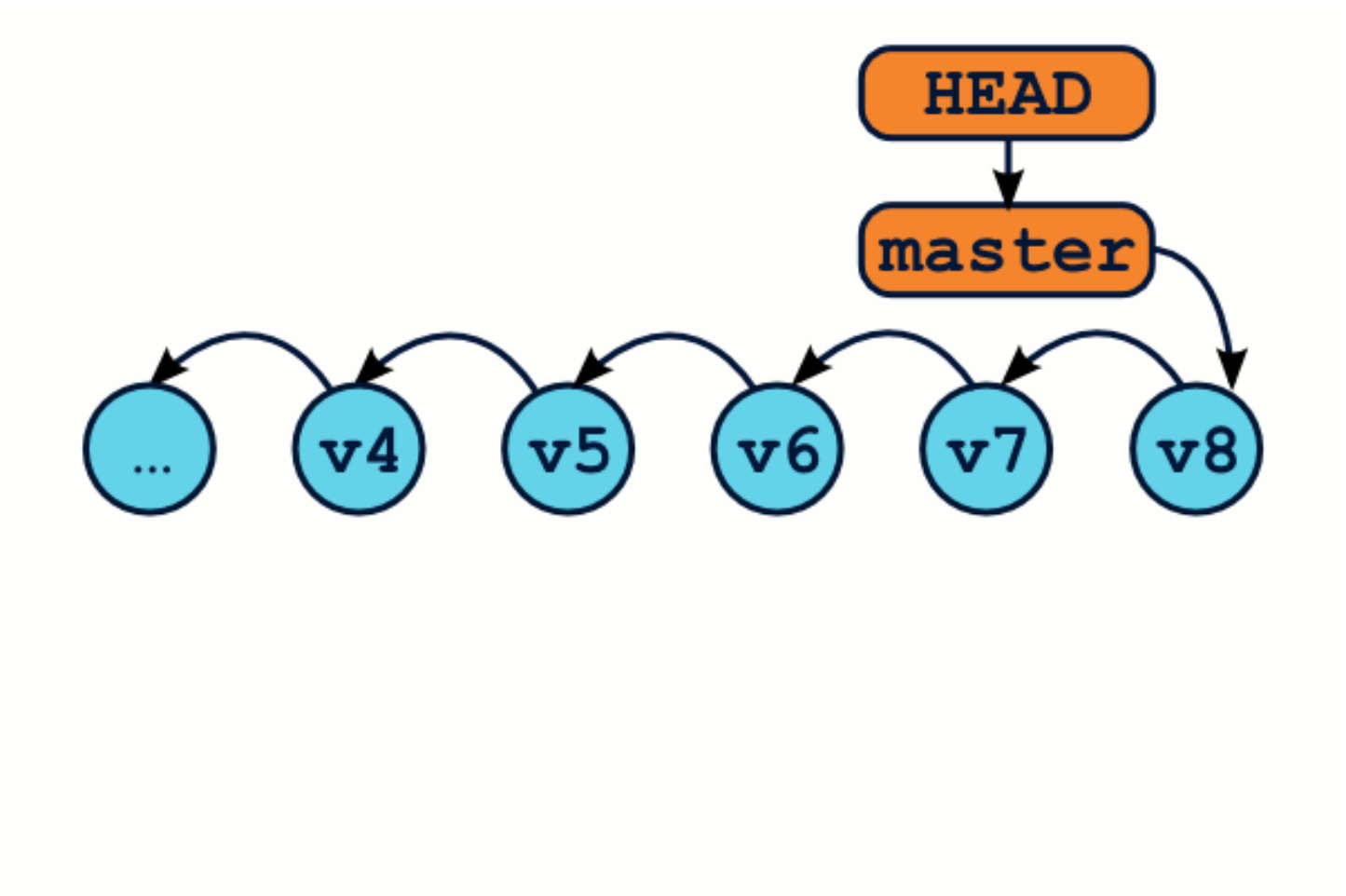
Le protocole précis de résolution est trés bien décrit dans la documentation de la commande *merge*.

```
$ git help merge
```

J'ai des petits problèmes dans ma plantation

Il est trés utile de comprendre que des branches ne sont rien que des étiquettes qui pointent vers des commits. On s'aperçoit alors qu'il est littéralement possible de créer des branches à n'importe quel moment, même depuis un ancien commit.

```
$ git checkout -b test v5
$ Commit… commit… commit…
```

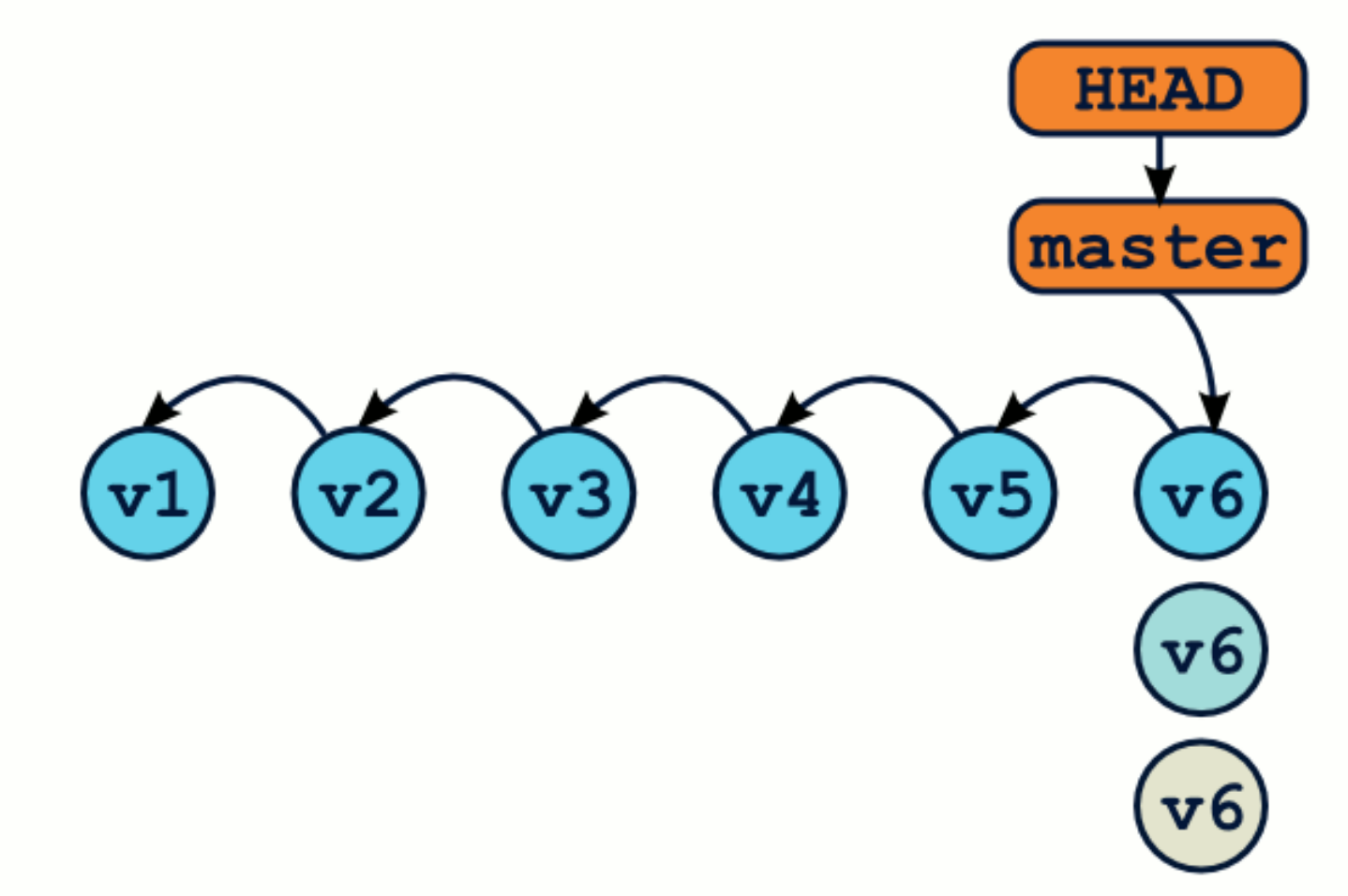


L'état DETACHED HEAD ou *Tête détachée*

Certaines séries de manipulations peuvent parfois laisser votre dépôt Git dans un état dit de *tête détachée*, ce qui est souvent source de confusion. N'y voyons là aucune allusion à un quelconque élément de l'histoire de la révolution française, l'explication est toute autre.

Jusqu'à maintenant, notre référence HEAD a toujours pointé vers une branche, vous vous rappelez ? Ainsi, la commande *git checkout ma_branche* déplace notre HEAD vers la référence « ma_branche ».

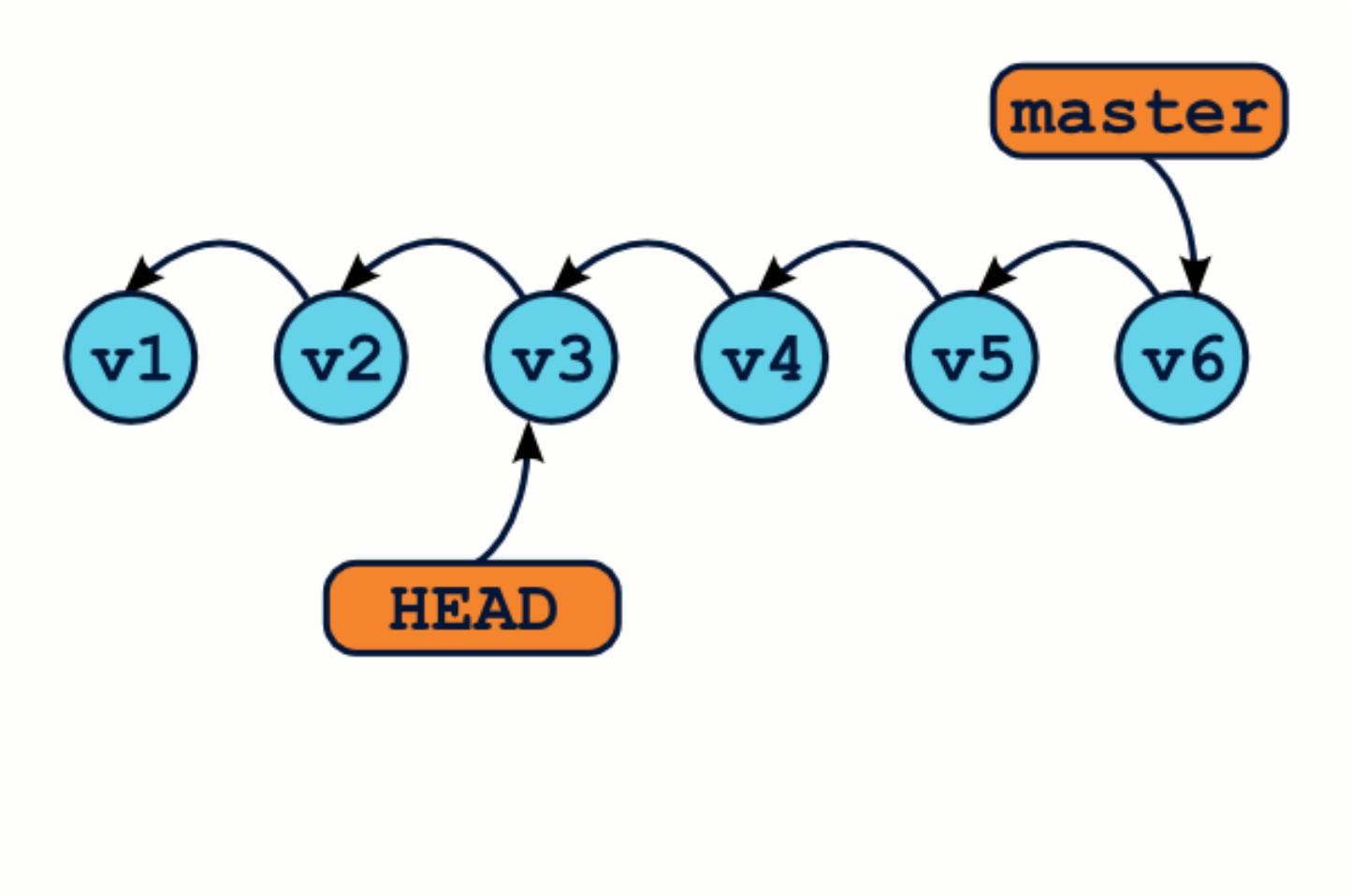
Mais que se passe-t-il si, en lieu et place d'une référence, nous passons directement un identifiant de commit à la commande *git checkout* ?



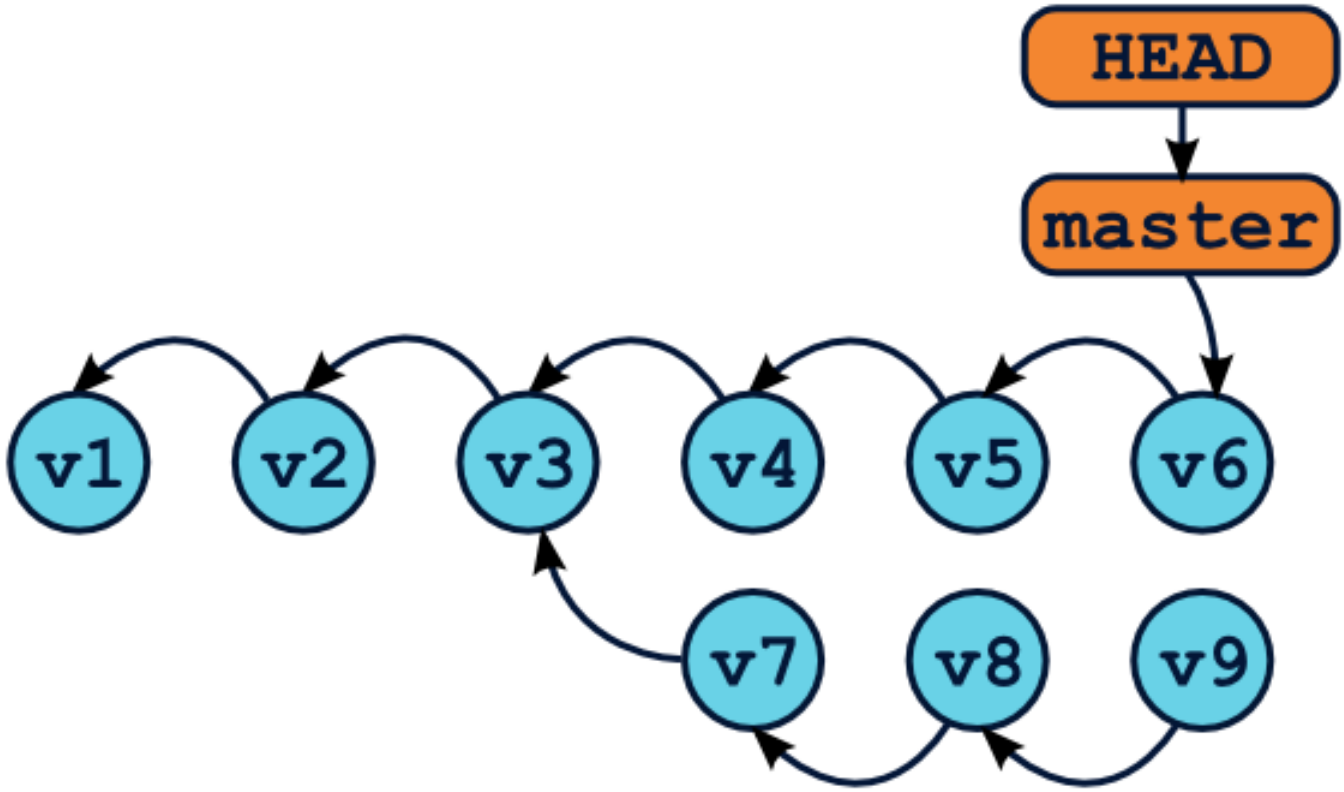
On se retrouve dans ce qu'on appelle l'état *Detached HEAD* : on travaille directement sur un commit, plus sur une branche.

```
$ git branch
* (no branch)
  master
```

L'ajout de nouveaux commits se fera de la manière habituelle.



En revanche, que se passe-t-il si nous utilisons la commande *git checkout* pour retourner sur notre branche *master* ?



OMG ! WTF ! Il semblerait bien que nous ayons perdus des commits ! En effet, la commande *git log* n'affichera que les commits de la branche master (de v1 à v6), mais pas les v7, v8 et v9 ! Et à moins de connaître l'identifiant sha1 exact de ces commits, il semble impossible de les récupérer. Ils ont tout bonnement disparu.

Heureusement, Git dispose d'un outil qui va nous sauver la vie : il s'agit du *reflog*. Git conserve un log de tous les déplacements de la référence HEAD, accessible grâce à la commande *git reflog*.

```
$ git reflog
67d6a52 HEAD@{0}: checkout: moving from 05ca141 to master
05ca141 HEAD@{1}: commit: Create user admin module
0fb7465 HEAD@{2}: commit: upgrade pymill version
8f4c5ba HEAD@{3}: commit: Added log message on new reminder cr
bf15474 HEAD@{4}: checkout: moving from master to bf15474
…
```

Bingo ! La première ligne nous indique la référence du commit juste avant le déplacement de HEAD vers master. Nous pouvons alors créer une branche vers ce commit qui nous permettra de le retrouver plus tard.

```
$ git branch test 05ca141  # Créé une branche, et reste sur ma
```

Si récupérer ces commits ne vous intéresse pas, alors laissez les choses en l'état. Les commits qui ne sont accessibles à travers aucune référence sont régulièrement supprimés par le garbage collector de Git.

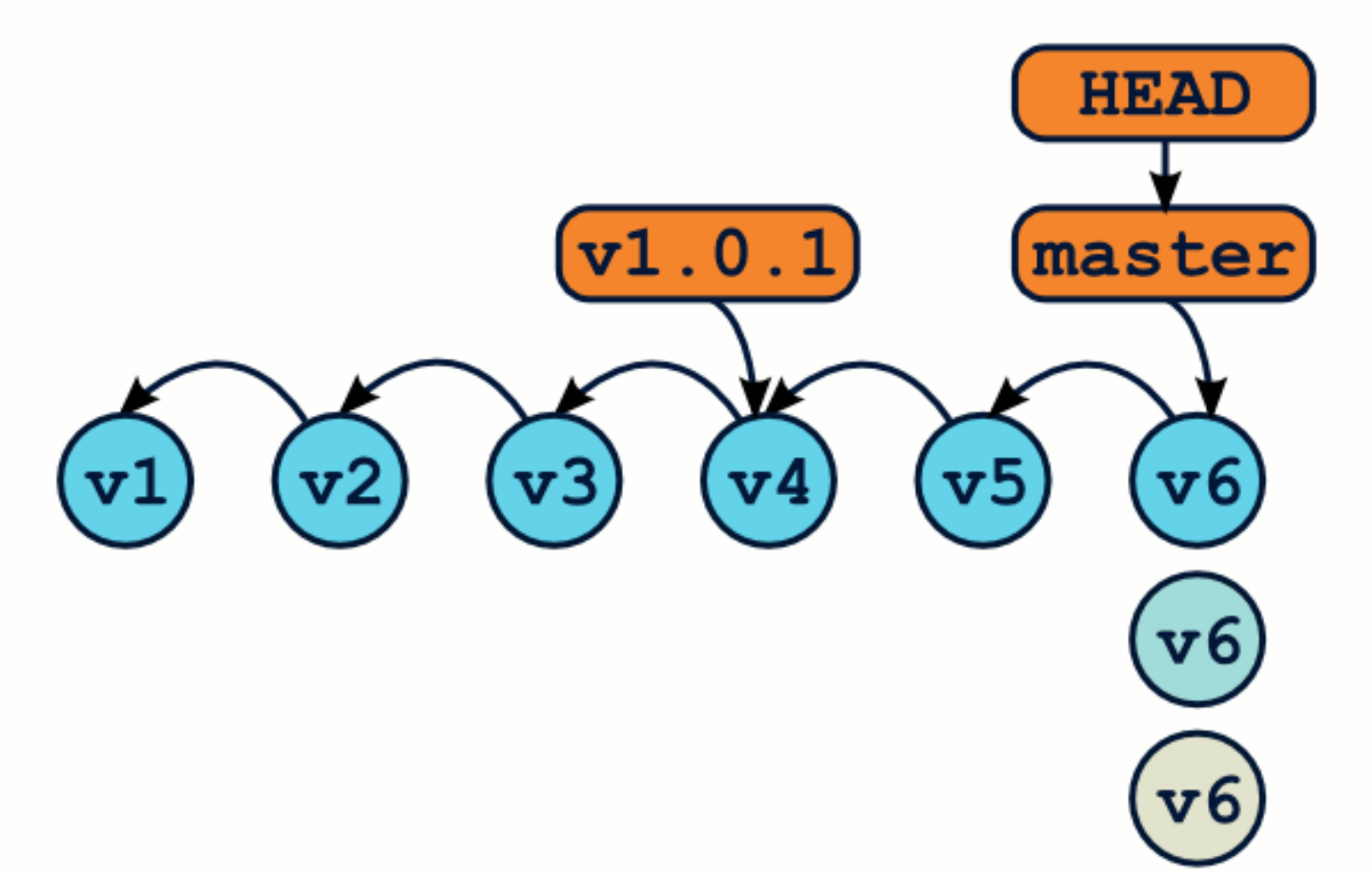
Comprendre cette satanée commande *git checkout*

Si vous avez bien tout suivi, vous avez remarqué que la commande *git checkout* est utilisée à toutes les sauces et semble produire des résultats différents selon les cas.

Il y a en fait deux façons principales d'utiliser la commande *git checkout* :

```
$ git checkout <commit ou branche>  # 1) Avec un identifiant d
$ git checkout  <commit ou branche> <repertoire ou fichier >
$ git checkout  <repertoire ou fichier> # Équivalent à git che
$ git checkout  # Équivalent à git checkout HEAD
```

Dans le premier mode d'utilisation (sans spécifier de fichier ou répertoire), Git va simplement déplacer la référence HEAD, et mettre à jour les deux arborescences de la zone de staging et du répertoire de travail. Si vous avez des modifications en cours, elles ne seront pas écrasées, et Git tentera de fusionner la version courante du fichier avec celle correspondant à la nouvelle position de HEAD.



Ainsi, si vous souhaitez récupérer dans votre répertoire de travail votre projet dans l'état ou il était au moment du tag v1.0.1, vous utiliserez la commande *git checkout*

v1.0.1, et vous pourrez revenir à l'état précédent grâce à *git checkout master*. On comprend ainsi pourquoi *git checkout* permet de passer d'une branche à l'autre. Par ailleurs, utiliser l'option *-b* permet en plus de créer une nouvelle branche au nouvel emplacement de HEAD.

Le deuxième mode d'utilisation (en spécifiant un fichier ou répertoire) fait grosso-modo la même chose, à quelques exceptions près :

1. la référence HEAD n'est pas déplacée ;
2. la zone de staging n'est pas touchée ;
3. le working directory sera mis à jour à partir de la zone de staging ;
4. l'effet de la commande est limitée au fichier ou répertoire spécifié ;
5. les fichiers en cours de modification seront écrasés purement et simplement.

En gros, la commande *git checkout v1.0.1 accounts* écrase le répertoire *accounts* de votre répertoire de travail avec la version de ce répertoire telle qu'elle était au moment du commit correspondant au tag *v1.0.1*. Clair ?

Voici quelques exemples d'utilisation :

```
$ git checkout .  # Supprime toutes les modifications en cours
$ git checkout accounts/  # Supprime toutes les modifications
$ git checkout v1.0.1 accounts/  # Récupère dans le rép. de tr
```

Un peu tordu ? J'en conviens volontiers. Attendez ! attendez ! Il nous reste à nous attaquer à la commende *reset*.

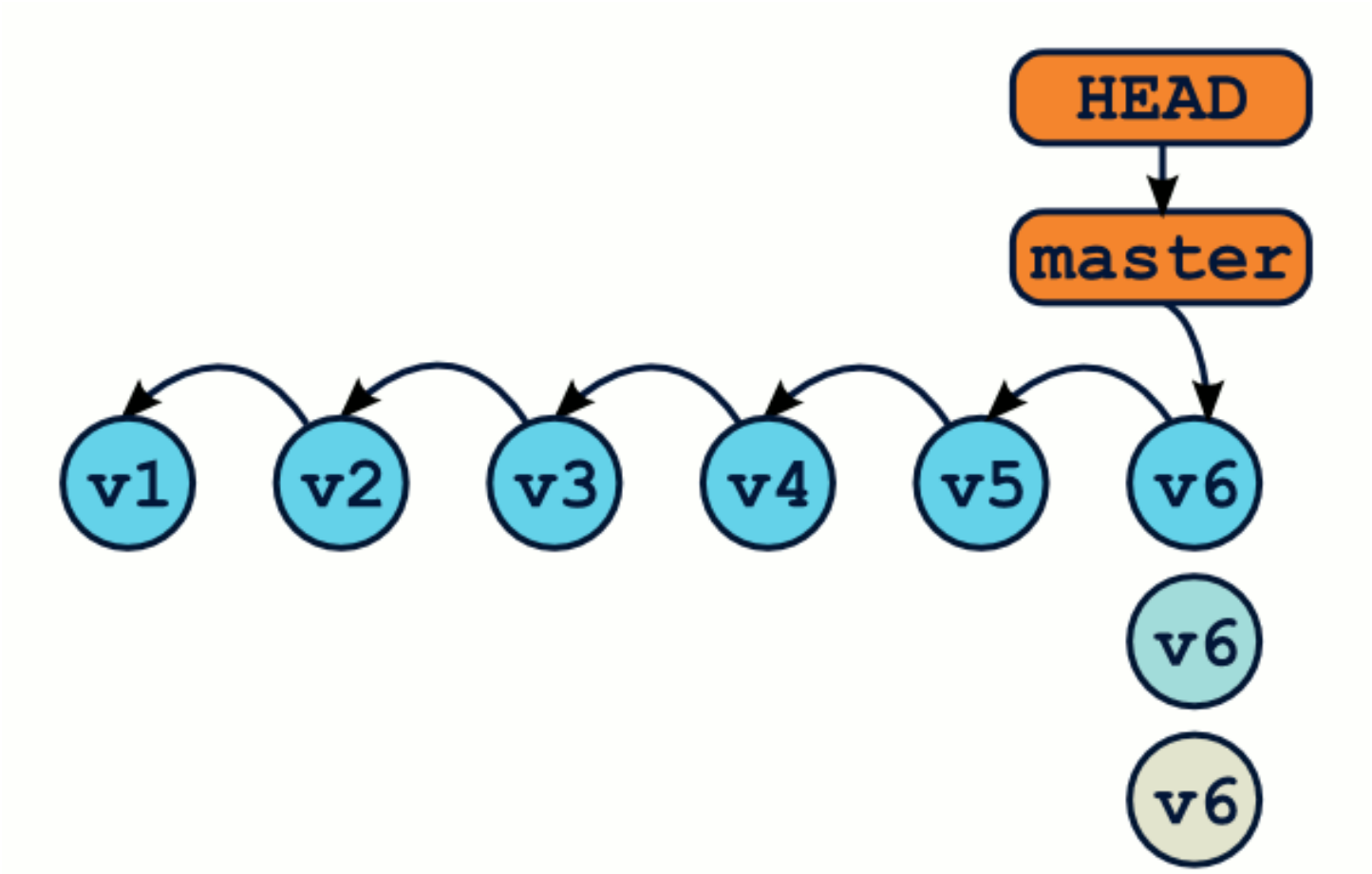
Comprendre la commande *git reset*

Vous avez aimé *git checkout* ? Vous allez adorer *git reset* ! Le principe de cette

commande est grossièrement similaire, sauf qu'en plus de déplacer la référence HEAD, *git reset* déplace également la référence de la branche courante. De même, *git reset* dispose de deux modes d'utilisation : avec ou sans spécifier de chemin de fichier.

Par ailleurs, on peut préciser à la commande quelles zones devront être mises à jour après le déplacement de HEAD.

```
$ git reset v1.0.1 --soft  # Déplace HEAD… et c'est tout !
$ git reset v1.0.1 --mixed  # Déplace HEAD, et met à jour le s
$ git reset v1.0.1 --hard  # Déplace HEAD, met à jour le stagi
```



Attention, contrairement à *git checkout*, *git reset --hard* écrasera votre répertoire

de travail même si vous avez des modifications en cours. Il est donc possible de perdre du travail.

Si vous spécifiez un chemin de fichier en plus, alors la commit fonctionnera de manière similaire, à quelques exceptions près :

1. la référence HEAD ne sera pas déplacée ;
2. les modifications seront limitées au chemin de fichier spécifié ;

Quelques exemples d'utilisation :

```
$ git reset  # Équivalent à git reset --mixed HEAD, supprime t
$ git reset --hard  # Supprime toutes les modifications par ra
$ git reset --hard v1.0.0  # Supprime tous les commits depuis
$ git reset HEAD^  # Annule le dernier commit, mais laisse le
$ git reset accounts/  # Annule les modifications sous account
$ git reset --hard accounts/  # Cette combinaison d'option est
```

C'est plus clair ?

Des questions ?

Bon, j'espère que ces quelques éclaircissements vous auront permis d'appréhender Git d'une manière moins empirique. Il parait aussi que je suis capable de donner de chouettes formations sur Git, alors si vous...

- ...aimeriez mieux comprendre Git,
- ...souhaitez convertir votre entreprise à Git,
- ...galérez pendant cette conversion,

n'hésitez pas, contactez-moi. Et si vous avez d'autres questions, n'hésitez pas à les

poser, je me ferai (peut-être) un plaisir d'y répondre.

Vous aimez ce billet ? Partagez-le !

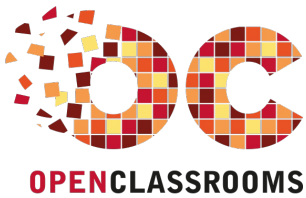
Twitter Facebook Reddit Linkedin Tumblr

Accueil Blog Photos Prestations Twitter

© Thibault Jouannic

Gérez vos codes source avec Git

Par Mathieu Nebra (Mateo21)



www.openclassrooms.com

Licence Creative Commons 6 2.0
Dernière mise à jour le 25/01/2012

Sommaire

Gérez vos codes source avec Git

Par  Mathieu Nebra (Mateo21)

Mise à jour : 25/01/2012
Difficulté : Difficile Durée d'étude : 4 heures

(cc) BY-NC-SA

Si vous avez déjà travaillé sur un projet informatique, que ce soit un petit projet personnel ou un plus gros projet professionnel, vous avez certainement déjà rencontré un de ces problèmes :

- « Qui a modifié le fichier X, il marchait bien avant et maintenant il provoque des bugs ! » ;
- « Robert, tu peux m'aider en travaillant sur le fichier X pendant que je travaille sur le fichier Y ? Attention à ne pas toucher au fichier Y car si on travaille dessus en même temps je risque d'écraser tes modifications ! » ;
- « Qui a ajouté cette ligne de code dans ce fichier ? Elle ne sert à rien ! » ;
- « À quoi servent ces nouveaux fichiers et qui les a ajoutés au code du projet ? » ;
- « Quelles modifications avions-nous faites pour résoudre le bug de la page qui se ferme toute seule ? »

Si ces problèmes-là vous parlent, vous auriez dû utiliser un **logiciel de gestion de versions**. Ce type de logiciel est devenu indispensable lorsqu'on travaille à plusieurs sur un même projet et donc sur le même code source. Même si vous travaillez seuls, vous aurez intérêt à commencer à en utiliser un rapidement car il vous offrira de nombreux avantages, comme la conservation d'un historique de chaque modification des fichiers par exemple.

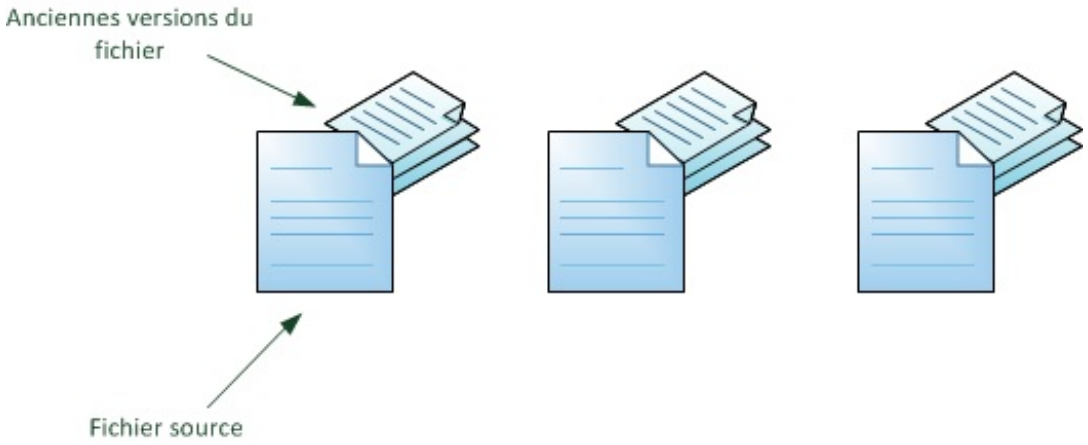
Il existe de nombreux logiciels de gestion de versions, comme SVN (Subversion), Mercurial et Git. Dans ce tutoriel, je vous présenterai **Git** (prononcez « guite ») qui est un des plus puissants logiciels de ce genre. Nous l'utilisons notamment pour gérer le code source du Site du Zéro !
Sommaire du tutoriel :

Qu'est-ce qu'un logiciel de gestion de versions ?

Les logiciels de gestion de versions sont utilisés principalement par les développeurs ; ce sont donc bel et bien des outils pour *geeks*. En effet, ils sont quasi exclusivement utilisés pour gérer des codes sources, car ils sont capables de suivre l'évolution d'un fichier texte ligne de code par ligne de code.
Ces logiciels sont fortement conseillés pour gérer un projet informatique.

Un **projet informatique** correspond aux étapes de création d'un programme, d'un site web ou de tout autre outil informatique. Que vous travailliez seuls sur le développement de ce projet ou à plusieurs, vous avez un objectif (par exemple, « créer un site web ») et vous allez devoir écrire du code source pour parvenir à cet objectif. On dit que vous travaillez sur un **projet**.

Ces outils suivent l'évolution de vos fichiers source et gardent les anciennes versions de chacun d'eux.



S'ils s'arrêtaient à cela, ce ne seraient que de vulgaires outils de *backup* (sauvegarde). Cependant, ils proposent de nombreuses fonctionnalités qui vont vraiment vous être utiles tout au long de l'évolution de votre projet informatique :

- ils retiennent qui a effectué chaque modification de chaque fichier et *pourquoi*. Ils sont par conséquent capables de dire qui a écrit chaque ligne de code de chaque fichier et dans quel but ;
- si deux personnes travaillent simultanément sur un même fichier, ils sont capables d'assembler (de fusionner) leurs modifications et d'éviter que le travail d'une de ces personnes ne soit écrasé.

Ces logiciels ont donc par conséquent deux utilités principales :

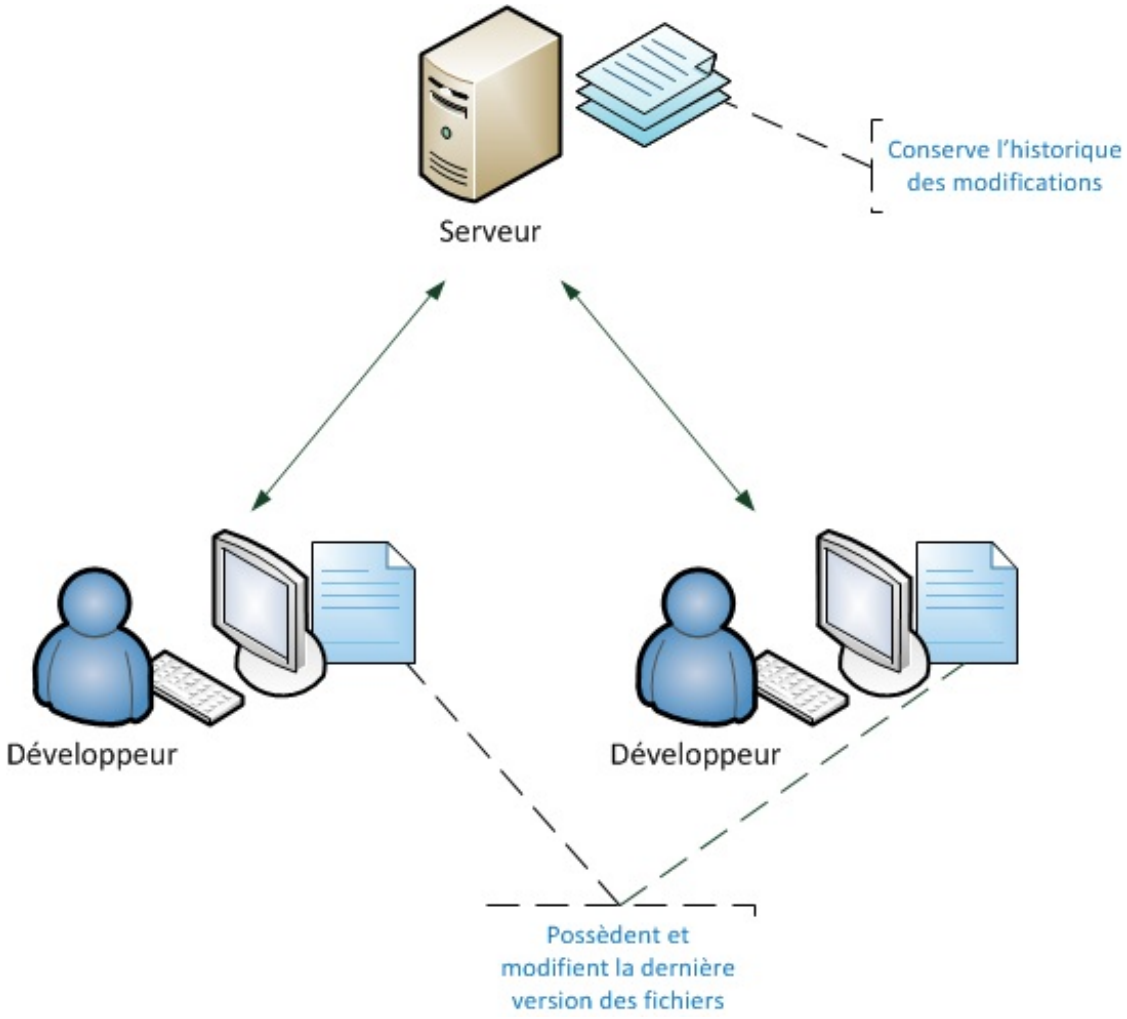
- **suivre l'évolution d'un code source**, pour retenir les modifications effectuées sur chaque fichier et être ainsi capable de *revenir en arrière* en cas de problème ;
- **travailler à plusieurs**, sans risquer de se marcher sur les pieds. Si deux personnes modifient un même fichier en même temps, leurs modifications doivent pouvoir être fusionnées sans perte d'information.

Logiciels centralisés et distribués

Il existe deux types principaux de logiciels de gestion de versions.

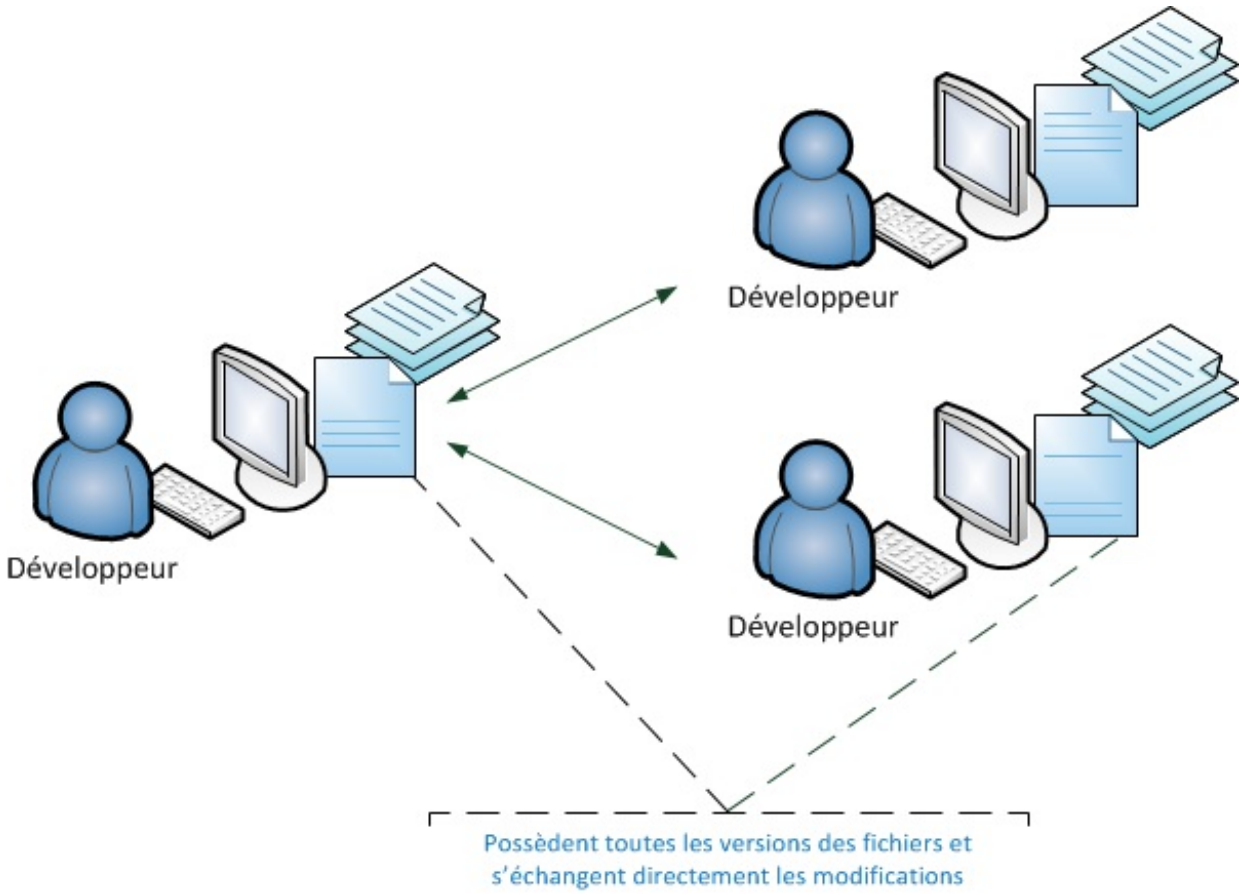
- **Les logiciels centralisés** : un serveur conserve les anciennes versions des fichiers et les développeurs s'y connectent pour prendre connaissance des fichiers qui ont été modifiés par d'autres personnes et pour y envoyer leurs modifications.
- **Les logiciels distribués** : il n'y a pas de serveur, chacun possède l'historique de l'évolution de chacun des fichiers. Les développeurs se transmettent directement entre eux les modifications, à la façon du *peer-to-peer*.

Voici, schématiquement, comment fonctionne un logiciel de gestion de versions centralisé :



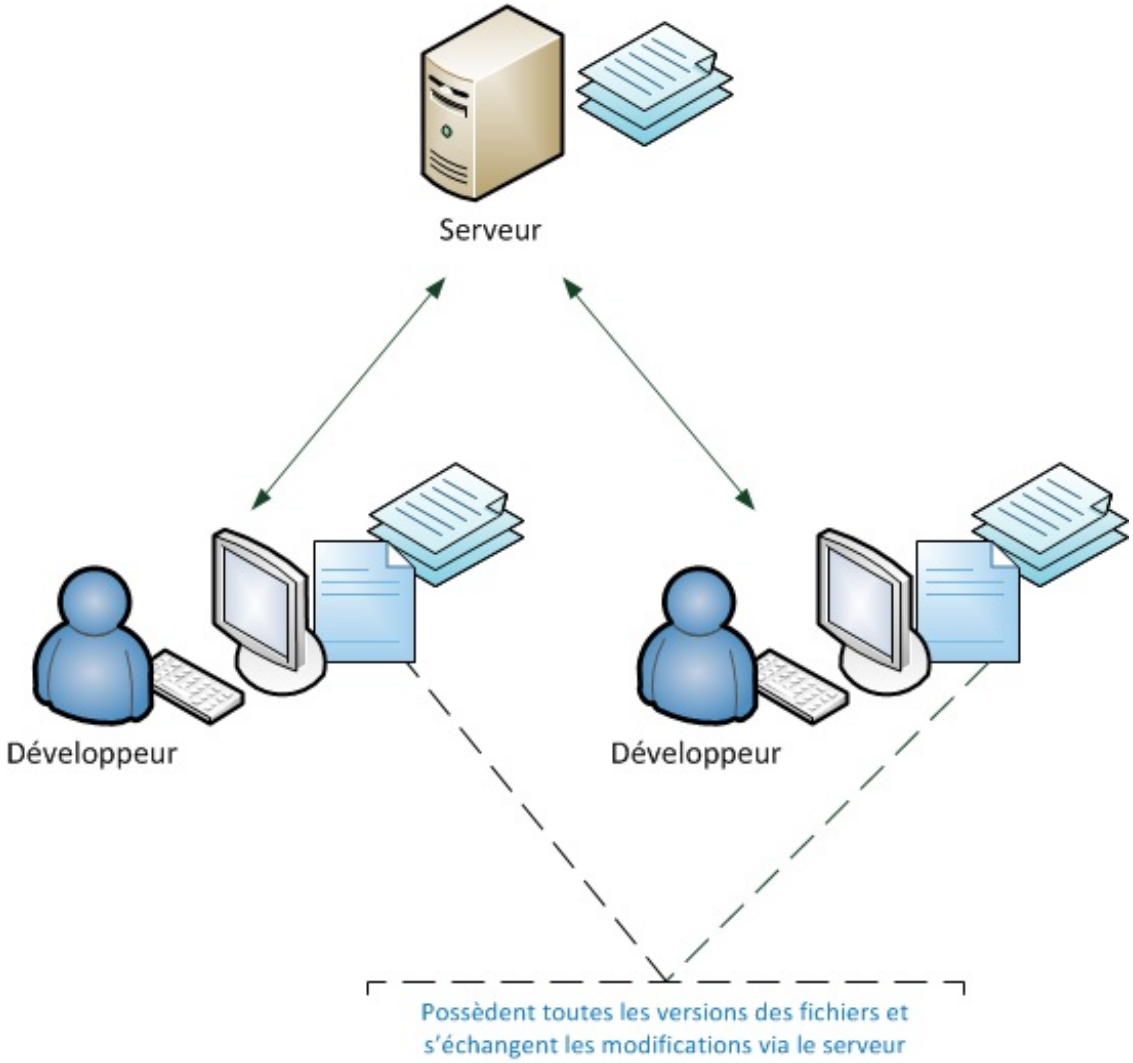
Un logiciel de gestion de versions centralisé.
Le serveur retient les anciennes versions des fichiers et communique les changements aux développeurs.

De même, voici le fonctionnement d'un logiciel de gestion de versions distribué :



Un logiciel de gestion de versions distribué.
Il n'y a pas de serveur. Les développeurs conservent l'historique des modifications et se transmettent les nouveautés.

Dans la pratique, les logiciels distribués sont rarement utilisés comme sur le schéma précédent. Même lorsque les logiciels sont capables de fonctionner en mode distribué, on utilise très souvent un serveur qui sert de point de rencontre entre les développeurs. Le serveur connaît l'historique des modifications et permet l'échange d'informations entre les développeurs, qui eux possèdent également l'historique des modifications.



Un logiciel de gestion de versions distribué avec un serveur.
Le serveur sert de point de rencontre entre les développeurs et possède lui aussi l'historique des versions.

C'est dans ce dernier mode que nous allons fonctionner avec Git.
Il a l'avantage d'être à la fois flexible et pratique. Pas besoin de faire de sauvegarde du serveur étant donné que tout le monde possède l'historique des fichiers, et le serveur simplifie la transmission des modifications.

Git et les autres logiciels de gestion de versions

Les différents logiciels de gestion de versions

Dans ce tutoriel, je vais vous présenter Git, le logiciel de gestion de versions que nous utilisons pour gérer le code source du Site du Zéro.
Néanmoins, il existe d'autres logiciels du même type que je tiens aussi à vous présenter rapidement afin que vous les connaissiez au moins de nom.

Outil	Type	Description	Projets qui l'utilisent
CVS	Centralisé	C'est un des plus anciens logiciels de gestion de versions. Bien qu'il fonctionne et soit encore utilisé pour certains projets, il est préférable d'utiliser SVN (souvent présenté comme son successeur) qui corrige un certain nombre de ses défauts, comme son incapacité à suivre les fichiers renommés par exemple.	OpenBSD…
SVN	Centralis	Probablement l'outil le plus utilisé à l'heure actuelle. Il est assez simple d'utilisation, bien qu'il nécessite comme tous les outils du même type un certain temps d'adaptation. Il a l'avantage d'être bien intégré à Windows avec le	Apache, Redmine

(Subversion)	é	programme Tortoise SVN, là où beaucoup d'autres logiciels s'utilisent surtout en ligne de commande dans la console. Il y a un tutoriel SVN sur le Site du Zéro.	Redmine, Struts…
Mercurial	Distribué	Plus récent, il est complet et puissant. Il est apparu quelques jours après le début du développement de Git et est d'ailleurs comparable à ce dernier sur bien des aspects. Vous trouverez un tutoriel sur Mercurial sur le Site du Zéro.	Mozilla, Python, OpenOffice.org …
Bazaar	Distribué	Un autre outil, complet et récent, comme Mercurial. Il est sponsorisé par Canonical, l'entreprise qui édite Ubuntu. Il se focalise sur la facilité d'utilisation et la flexibilité.	Ubuntu, MySQL, Inkscape…
Git	Distribué	Très puissant et récent, il a été créé par Linus Torvalds, qui est entre autres l'homme à l'origine de Linux. Il se distingue par sa rapidité et sa gestion des branches qui permettent de développer en parallèle de nouvelles fonctionnalités.	Kernel de Linux, Debian, VLC, Android, Gnome, Qt…

Notez qu'il existe d'autres outils de gestion de versions ; je ne vous ai présenté ici que les principaux.
Tous ceux de cette liste sont des logiciels libres, mais il existe aussi des logiciels propriétaires : Perforce, BitKeeper, Visual SourceSafe de Microsoft, etc.

Quelles sont les particularités de Git ?

Je n'entrerai pas dans les détails de la comparaison de Git avec les autres outils concurrents comme SVN et Mercurial. Retenez simplement que :

- CVS est le plus ancien et il est recommandé de ne plus l'utiliser car il est le moins puissant et n'est plus très bien mis à jour ;
- SVN est le plus connu et le plus utilisé à l'heure actuelle, mais de nombreux projets commencent à passer à des outils plus récents ;
- Mercurial, Bazaar et Git se valent globalement, ils sont récents et puissants, chacun a des avantages et des défauts. Ils sont tous distribués, donc chaque développeur possède l'historique des modifications et ils permettent en théorie de se passer de serveur (bien qu'on utilise toujours un serveur pour des raisons pratiques).

Concernant les avantages de Git sur les autres, certains ont fait des listes comparatives intéressantes (bien que toujours criticables).

On retiendra surtout que Git :

- est très rapide ;
- sait travailler par branches (versions parallèles d'un même projet) de façon très flexible ;
- est assez complexe, il faut un certain temps d'adaptation pour bien le comprendre et le manipuler, mais c'est également valable pour les autres outils ;
- est à l'origine prévu pour Linux. Il existe des versions pour Windows mais pas vraiment d'interface graphique simplifiée. Il est donc à réserver aux développeurs ayant un minimum d'expérience et… travaillant de préférence sous Linux.

Une des particularités de Git, c'est l'existence de sites web collaboratifs basés sur Git comme GitHub et Gitorious. GitHub, par exemple, est très connu et utilisé par de nombreux projets : jQuery, Symfony, Ruby on Rails…
C'est une sorte de réseau social pour développeurs : vous pouvez regarder tous les projets évoluer et décider de participer à l'un d'entre eux si cela vous intéresse. Vous pouvez aussi y créer votre propre projet : c'est gratuit pour les projets *open source* et il existe une version payante pour ceux qui l'utilisent pour des projets propriétaires.

GitHub fournit le serveur où les développeurs qui utilisent Git se rencontrent. C'est un excellent moyen de participer à des projets *open source* et de publier votre projet !

Installer et configurer Git

Installation

Nous allons voir ici comment installer Git sous Linux, Windows et Mac OS X. Comme je vous le disais plus tôt, Git est plus agréable à utiliser sous Linux et sensiblement plus rapide, mais il reste néanmoins utilisable sous Windows.

Installer Git sous Linux

Avec un gestionnaire de paquets, c'est très simple :

Code : Console

```
sudo apt-get install git-core gitk
```

Cela installe 2 paquets :

- git-core : c'est git, tout simplement. C'est le seul paquet vraiment indispensable ;
- gitk : une interface graphique qui aide à mieux visualiser les logs. Facultatif.

Il est recommandé de se créer une paire de clés pour se connecter au serveur de rencontre lorsque l'accès à Git se fait via SSH. Un tutoriel du SdZ explique comment créer des clés : http://www.siteduzero.com/tutoriel-3-7 [...] tml#ss_part_5.

Installer Git sous Windows

Pour utiliser Git sous Windows, il faut installer msysgit : http://code.google.com/p/msysgit/.

Cela installe msys (un système d'émulation des commandes Unix sous Windows) et Git simultanément.

Comme vous le voyez, ce n'est pas une « vraie » version pour Windows mais plutôt une forme d'émulation. Je l'ai testé, cela fonctionne, mais c'est loin d'être aussi agréable et rapide que sous Linux. Si la majorité des développeurs de votre projet utilisent Windows, je vous recommanderai donc plutôt d'essayer Mercurial ou Bazaar qui, sous Windows, ont des interfaces graphiques plus adaptées.

Lors de l'installation, laissez toutes les options par défaut, elles conviennent bien.

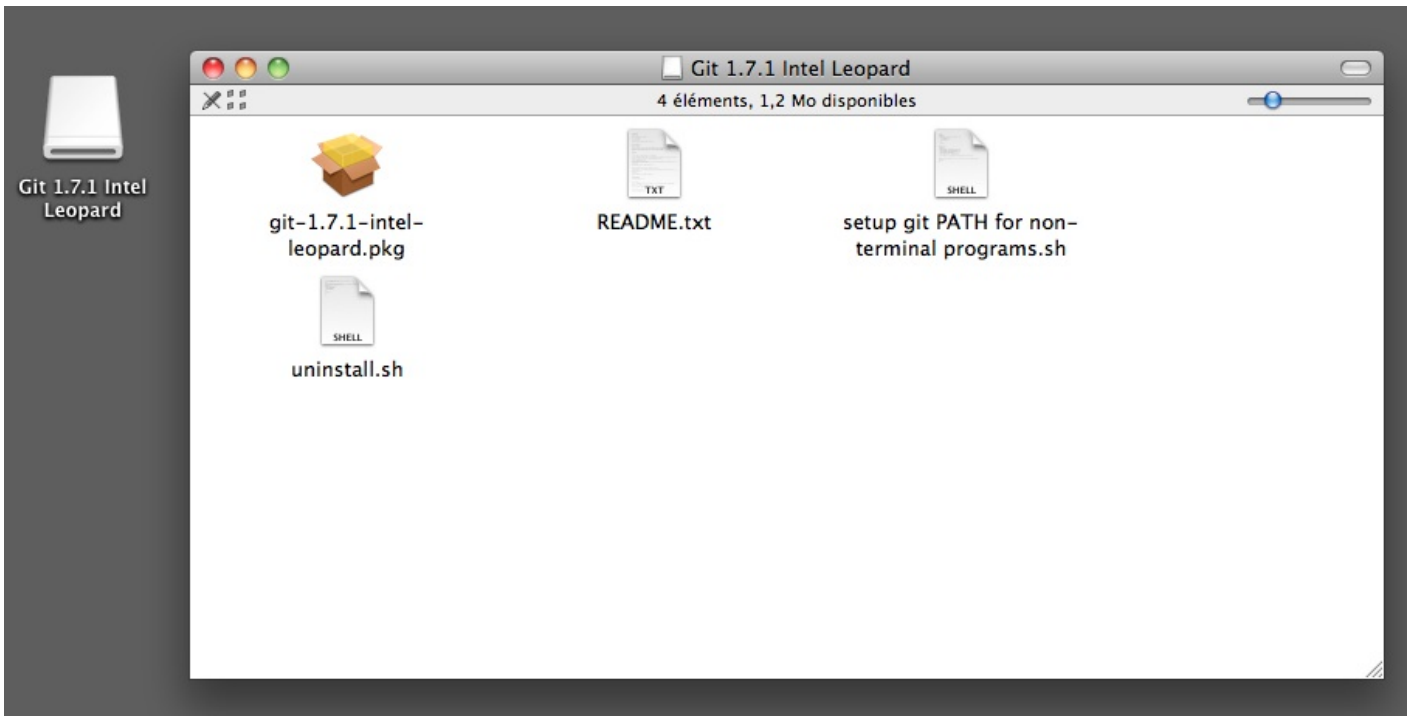
Une fois que c'est installé, vous pouvez lancer une console qui permet d'utiliser Git en ouvrant le programme Git Bash. Les commandes de base d'Unix fonctionnent sans problème : cd, pwd, mkdir, etc.

De la même façon que sous Linux, pour se connecter régulièrement à un serveur, il est recommandé de se créer une paire de clés avec Puttygen et de charger la clé en mémoire avec Pageant (normalement installé avec Putty). Toutes les explications sont dans le tutoriel du SdZ : http://www.siteduzero.com/tutoriel-3-7 [...] tml#ss_part_5.

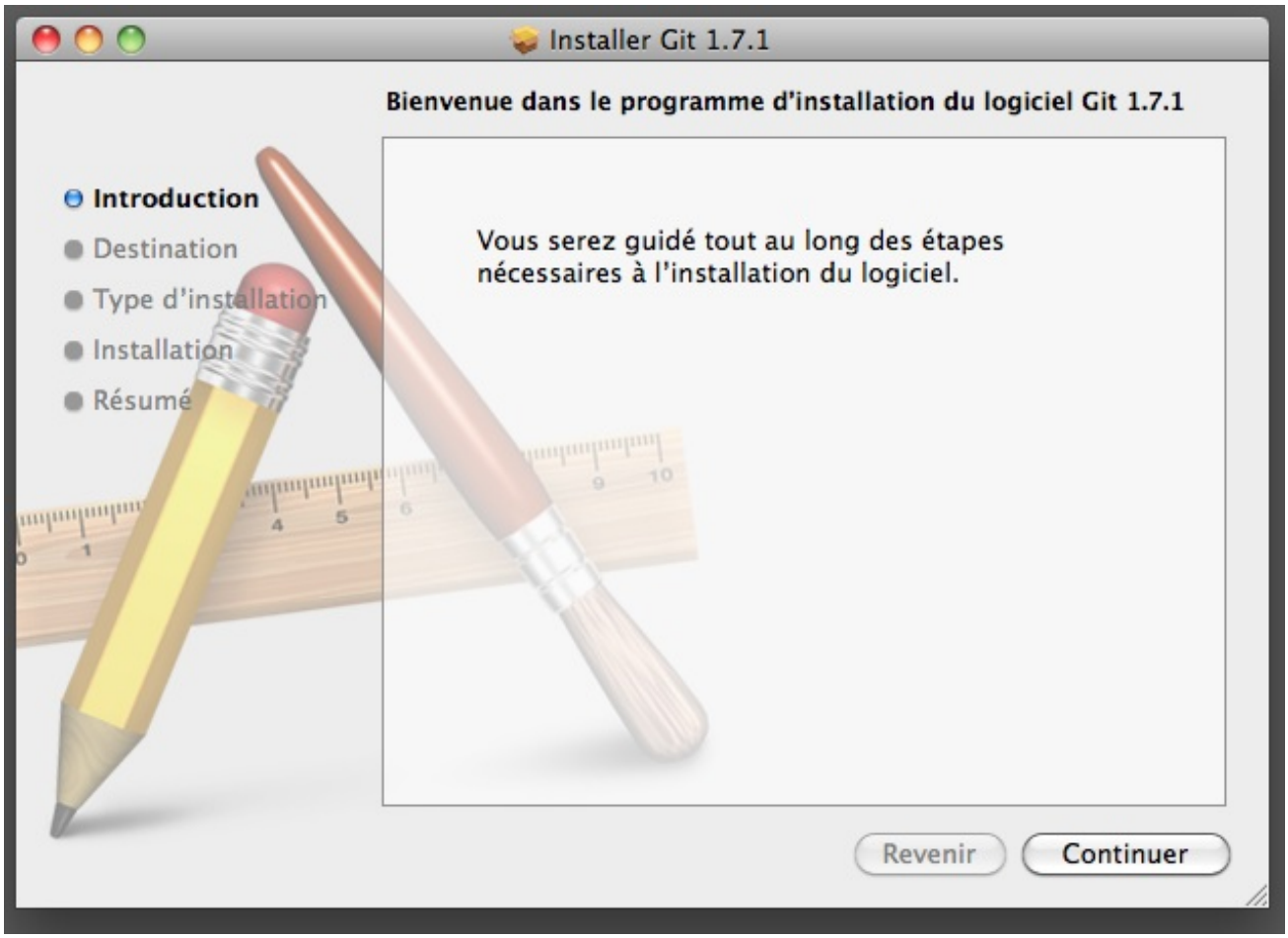
Installer Git sous Mac OS X

Il y a plusieurs façons d'installer Git sous Mac OS X. Le plus simple est de se baser sur cet installeur pour Mac OS X.

Vous allez télécharger une archive .dmg. Il suffit de l'ouvrir pour la monter, ce qui vous donnera accès à plusieurs fichiers :



Ouvrez tout simplement l'archive .pkg qui se trouve à l'intérieur, ce qui aura pour effet d'exécuter le programme d'installation :



Suivez les étapes en laissant les valeurs par défaut (c'est suffisant), et vous aurez installé Git !

Il vous suffit ensuite d'ouvrir un terminal et vous aurez alors accès aux commandes de Git comme sous Linux.

Configurer Git

Maintenant que Git est installé, vous devriez avoir une console ouverte dans laquelle vous allez pouvoir taper des commandes de Git.

Dans la console, commencez par envoyer ces trois lignes :

Code : Console

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

Elles activeront la couleur dans Git. Il ne faut le faire qu'une fois, et ça aide à la lisibilité des messages dans la console.

De même, il faut configurer votre nom (ou pseudo) :

Code : Console

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

Puis votre e-mail :

Code : Console

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

Vous pouvez aussi éditer votre fichier de configuration `.gitconfig` situé dans votre répertoire personnel pour y ajouter une section alias à la fin :

Code : Console

```
vim ~/.gitconfig
```

Code : Autre

```
    [color]
            diff = auto
            status = auto
            branch = auto
    [user]
            name = votre_pseudo
            email = moi@email.com
    [alias]
            ci = commit
            co = checkout
            st = status
            br = branch
```

Ces alias permettent de raccourcir certaines commandes de Git. Ainsi, au lieu d'écrire `git checkout`, vous pourrez écrire si vous le désirez `git co`, ce qui est plus court.

Créer un nouveau dépôt ou cloner un dépôt existant

Pour commencer à travailler avec Git, il y a deux solutions :

- soit vous créez un nouveau dépôt vide, si vous souhaitez commencer un nouveau projet ;
- soit vous clonez un dépôt existant, c'est-à-dire que vous récupérez tout l'historique des changements d'un projet pour pouvoir travailler dessus.

Dans un logiciel de gestion de versions comme Git, un **dépôt** représente une copie du projet. Chaque ordinateur d'un développeur qui travaille sur le projet possède donc une copie du dépôt.
Dans chaque dépôt, on trouve les fichiers du projet ainsi que leur historique (voir les schémas du début du chapitre).

Je vais vous montrer les deux méthodes : la création d'un nouveau dépôt vide et le clonage d'un dépôt existant. À vous de choisir celle que vous préférez, sachant qu'il peut être pratique de commencer par cloner un dépôt existant pour se faire la main sur un projet qui utilise déjà Git. 

Créer un nouveau dépôt

Commencez par créer un dossier du nom de votre projet sur votre disque. Par exemple, je vais créer `/home/mateo21/plusoumoins` pour héberger mon jeu « Plus ou Moins » développé en C. 

Code : Console

```
cd /home/mateo21
mkdir plusoumoins
cd plusoumoins
```

Si votre projet existe déjà, inutile de créer un nouveau dossier. Rendez-vous simplement dans le dossier où se trouve votre projet.
Vous pouvez donc ignorer les étapes précédentes.

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

Code : Console

```
git init
```

C'est tout ! Vous venez de créer un nouveau projet Git dans le dossier où vous vous trouvez. 
Un dossier caché *.git* vient tout simplement d'être créé.

Il faudra ensuite créer les fichiers source de votre projet et les faire connaître à Git en faisant des commits, ce que je vous expliquerai un peu plus loin.

Cloner un dépôt existant

Cloner un dépôt existant consiste à récupérer tout l'historique et tous les codes source d'un projet avec Git.

Pour trouver un dépôt Git, rien de plus facile ! Comme je vous le disais, GitHub est une formidable fourmilière de dépôts Git. Vous y trouverez de nombreux projets connus (certains n'utilisent pas GitHub directement mais on y trouve quand même une copie à jour du projet).

Prenons par exemple Symfony, la nouvelle version du framework PHP qui permet de créer des sites robustes facilement (il s'agit ici de la version *Symfony 2* en cours de développement), que le Site du Zéro utilise, par ailleurs.

Rendez-vous sur la page GitHub du projet. Vous y voyez la liste des fichiers et des derniers changements ainsi qu'un champ contenant l'adresse du dépôt. En l'occurrence, l'adresse du dépôt de Symfony est :

Code : Autre

```
http://github.com/symfony/symfony.git
```

Comme vous le voyez, on se connecte au dépôt en HTTP, mais il existe d'autres méthodes : les protocoles « git:// » et « ssh:// ». Le plus souvent, on utilise SSH car il permet de chiffrer les données pendant l'envoi et gère l'authentification des utilisateurs.

Pour cloner le dépôt de Symfony, il suffit de lancer la commande suivante :

Code : Console

```
git clone http://github.com/symfony/symfony.git
```

Cela va créer un dossier « symfony » et y télécharger tous les fichiers source du projet ainsi que l'historique de chacune de leurs modifications !
Git compresse automatiquement les données pour le transfert et le stockage afin de ne pas prendre trop de place. Néanmoins, le clonage d'un dépôt comme ceci peut prendre beaucoup de temps (ce n'est pas vraiment le cas avec Symfony, mais essayez de cloner le dépôt du Kernel Linux pour voir !).

Les messages suivants devraient apparaître dans la console :

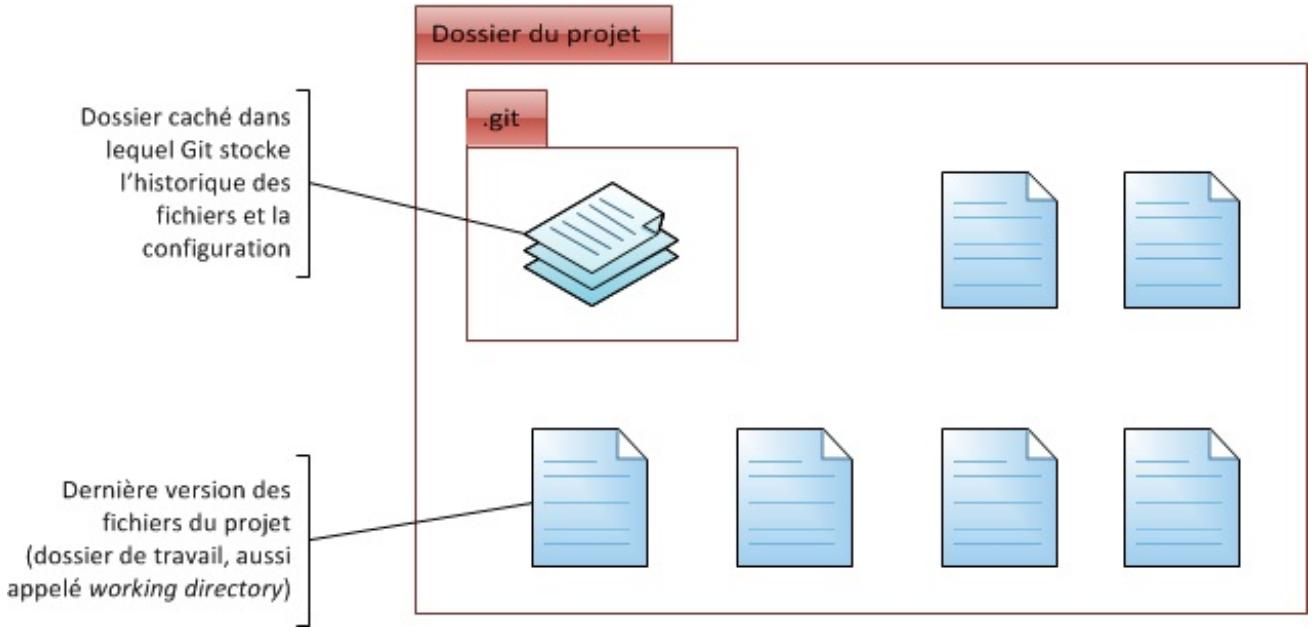
Code : Console

```
$ git clone http://github.com/symfony/symfony.git
Initialized empty Git repository in /home/mateo21/symfony/.git/
remote: Counting objects: 7820, done.
remote: Compressing objects: 100% (2490/2490), done.
remote: Total 7820 (delta 4610), reused 7711 (delta 4528)
Receiving objects: 100% (7820/7820), 1.40 MiB | 479 KiB/s, done.
Resolving deltas: 100% (4610/4610), done.
Checking out files: 100% (565/565), done.
```

Inutile d'essayer d'en comprendre le détail. Vous noterez toutefois que les objets sont compressés avant l'envoi, ce qui accélère le téléchargement. Une fois les fichiers reçus, Git les organise sur votre disque et vous voilà désormais en possession de tous les changements des fichiers du projet ainsi que de leur dernière version !

Vous pouvez ouvrir le dossier « symfony » et regarder son contenu, il y a tout le code source du projet.

Le seul dossier un peu particulier créé par Git est un dossier `.git` (c'est un dossier caché situé à la racine du projet). Il contient l'historique des modifications des fichiers et la configuration de Git pour ce projet.
Lorsque Git crée ou clone un dépôt sur votre ordinateur, il organise les dossiers comme ceci :



En fait, Git crée tout simplement un dossier *.git* caché à la racine du dossier de votre projet. Vous n'aurez pas à vous rendre dans ce dossier caché en temps normal, sauf éventuellement pour modifier le fichier de configuration *.git/config* qui se trouve à l'intérieur.
Mis à part ce dossier un peu « spécial », vous retrouverez tous les fichiers dans leur dernière version dans le dossier du projet. Ce sont eux que vous modifierez.

Sur GitHub, la plupart des dépôts sont en mode lecture seule (*read only*). Vous pouvez télécharger les fichiers, effectuer des modifications sur votre ordinateur (en local) mais pas les envoyer sur le serveur. Ceci permet d'éviter que n'importe qui fasse n'importe quoi sur les gros projets publics.
Si vous faites une modification utile sur le projet (vous résolvez un bug par exemple) il faudra demander à l'un des principaux développeurs du projet de faire un « pull », c'est-à-dire de télécharger vos modifications lui-même afin qu'il puisse vérifier si elles sont correctes.

Créer un dépôt qui servira de serveur

Si vous souhaitez mettre en place un serveur de rencontre pour votre projet, il suffit d'y faire un `git clone` ou un `git init` avec l'option *--bare*. Cela aura pour effet de créer un dépôt qui contiendra uniquement le dossier *.git* représentant l'historique des changements (ce qui est suffisant, car personne ne modifie les fichiers source directement sur le serveur).

Code : Console

```
git init --bare // À exécuter sur le serveur.
```

Pour se connecter au serveur, la meilleure méthode consiste à utiliser SSH. Ainsi, si vous voulez cloner le dépôt du serveur sur votre ordinateur, vous pouvez écrire quelque chose comme :

Code : Console

```
git clone ssh://utilisateur@monserveur.domaine.com/chemin/vers/le/depot/git // À ex
```

Il faudra bien entendu vous identifier en entrant votre mot de passe (sauf si vous avez autorisé votre clé publique).

Modifier le code et effectuer des commits

À ce stade, vous devriez avoir créé ou cloné un dépôt Git. Je vous recommande d'avoir fait un clone afin d'avoir une base de travail, ce sera plus simple que de commencer à zéro pour le moment.

Supposons que vous ayez cloné comme moi le dépôt Git de Symfony. Vous avez sur votre disque dur tous les fichiers source du projet et vous pouvez vous amuser à les modifier avec un éditeur de texte (pas de panique, les changements restent sur votre ordinateur, vous ne risquez pas d'envoyer des bêtises et pouvez donc faire toutes les expériences que vous voulez).

Placez-vous dans le répertoire de base du code, par exemple :

Code : Console

```
cd /home/mateo21/symfony
```

La commande `git status` vous indique les fichiers que vous avez modifiés récemment :

Code : Console

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

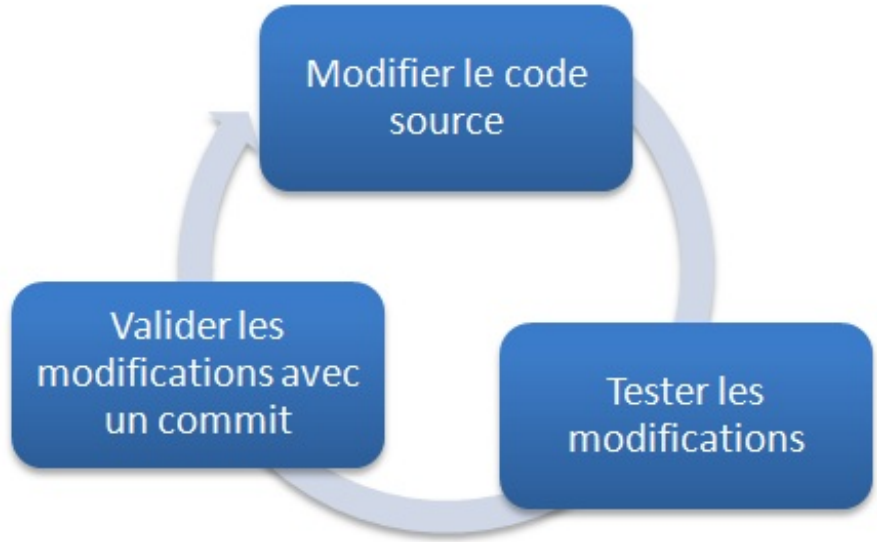
Ce message nous informe que rien n'a été modifié (*nothing to commit*).

Si vous avez défini des alias précédemment, il est plus rapide de taper `git st` que `git status`. C'est une commande que l'on peut être amené à taper plusieurs fois par jour.

Méthode de travail

Lorsqu'on travaille avec Git, on suit en général toujours les étapes suivantes :

1. modifier le code source ;
2. tester votre programme pour vérifier si cela fonctionne ;
3. faire un commit pour « enregistrer » les changements et les faire connaître à Git ;
4. recommencer à partir de l'étape 1 pour une autre modification.



Qu'est-ce qu'on appelle une modification du code source ?

C'est un ensemble de changements qui permet soit de régler un bug, soit d'ajouter une fonctionnalité.

Cela peut aussi bien correspondre à une ligne changée dans un fichier que 50 lignes changées dans un fichier A et 25 lignes dans un fichier B. Un commit représente donc un *ensemble de changements*. À vous de déterminer, dès que vos changements sont stables, quand vous devez faire un commit.

À titre indicatif, si vous travaillez toute une journée sur un code et que vous ne faites qu'un commit à la fin de la journée, c'est qu'il y a un problème (sauf si vous avez passé toute la journée sur le même bug). Les commits sont là pour « valider » l'avancement de votre projet : n'en faites pas un pour chaque ligne de code modifiée, mais n'attendez pas d'avoir fait 50 modifications différentes non plus !

Supposons que vous ayez effectué des modifications dans un des fichiers (par exemple *src/Symfony/Components/Yaml/Yaml.php*). Si vous avez modifié ce fichier et que vous l'avez enregistré, faites un `git status` dans la console pour voir :

Code : Console

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>…" to update what will be committed)
#   (use "git checkout --
 <file>…" to discard changes in working directory)
#
#       modified:   src/Symfony/Components/Yaml/Yaml.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git vous liste tous les fichiers qui ont changé sur le disque. Il peut aussi bien détecter les modifications que les ajouts, les suppressions et les renommages.

Vous pouvez voir concrètement ce que vous avez changé en tapant `git diff` :

Code : Console

```
$ git diff
diff --git a/src/Symfony/Components/Yaml/Yaml.php b/src/Symfony/Components/Yaml/
index fa0b806..77f9902 100644
--- a/src/Symfony/Components/Yaml/Yaml.php
+++ b/src/Symfony/Components/Yaml/Yaml.php
@@ -19,7 +19,7 @@ namespace Symfony\Components\Yaml;
  */
 class Yaml
 {
-    static protected $spec = '1.2';
+    static protected $spec = '1.3';

     /**
      * Sets the YAML specification version to use.
@@ -33,6 +33,8 @@ class Yaml
         if (!in_array($version, array('1.1', '1.2'))) {
             throw new \InvalidArgumentException(sprintf('Version %s of the YAML
         }
+
+        $mtsource = $version;

         self::$spec = $version;
     }
```

Les lignes ajoutées sont précédées d'un « + » tandis que les lignes supprimées sont précédées d'un « - ». Normalement les lignes sont colorées et donc faciles à repérer.
J'ai fait des modifications aléatoires ici mais cela aurait très bien pu correspondre à la correction d'un bug ou l'ajout d'une fonctionnalité.

Par défaut, Git affiche les modifications de tous les fichiers qui ont changé. Dans notre cas, il n'y en avait qu'un, mais s'il y en avait eu plusieurs nous les aurions tous vus.
Vous pouvez demander à Git d'afficher seulement les changements d'un fichier précis, comme ceci :

Code : Console

```
git diff src/Symfony/Components/Yaml/Yaml.php
```

Si les modifications vous paraissent bonnes et que vous les avez testées, il est temps de faire un commit.

Effectuer un commit des changements

En faisant `git status`, vous devriez voir les fichiers que vous avez modifiés en rouge. Cela signifie qu'ils ne seront pas pris en compte lorsque vous allez faire un commit.

Il faut explicitement préciser les fichiers que vous voulez « commiter ». Pour cela, trois possibilités :

- faire `git add nomfichier1 nomfichier2` pour ajouter les fichiers à la liste de ceux devant faire l'objet d'un commit, puis faire un `git commit`. Si vous faites un `git status` après un `git add`, vous les verrez alors en vert ;
- faire `git commit -a` pour « commiter » tous les fichiers qui étaient listés dans `git status` dans les colonnes « *Changes to be committed* » et « *Changed but not updated* » (qu'ils soient en vert ou en rouge) ;
- faire `git commit nomfichier1 nomfichier2` pour indiquer lors du commit quels fichiers précis doivent être « commités ».

J'utilise personnellement la seconde solution lorsque je veux « commiter » tous les fichiers que j'ai modifiés, et la troisième solution lorsque je veux « commiter » seulement certains des fichiers modifiés.
Faire appel à `git add` est indispensable lorsque vous venez de créer de nouveaux fichiers que Git ne connaît pas encore.

Cela lui permet de les prendre en compte pour le prochain commit.

Lorsque la commande *commit* est lancée, l'éditeur par défaut (généralement nano ou Vim) s'ouvre. Vous devez sur la première ligne taper un message qui décrit à quoi correspondent vos changements.

Il est possible de faire un message de commit sur plusieurs lignes. Néanmoins, réservez la première ligne pour une courte description synthétique de vos changements. Si vous ne mettez pas de message de commit, celui-ci sera annulé.

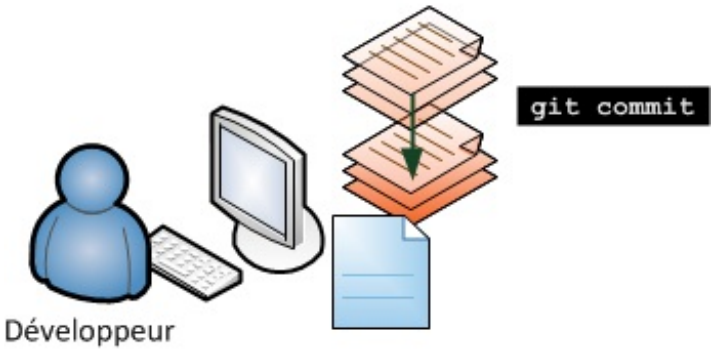
En règle générale, si vous ne pouvez pas résumer vos changements en une courte ligne c'est que vous avez passé trop de temps à faire des changements sans « commiter ».

Exemple de messages de commit d'une ligne corrects :

- « Améliore la visibilité des post-it sur le forum. » ;
- « Simplifie l'interface de changement d'avatar. » ;
- « Résout #324 : bug qui empêchait de valider un tutoriel à plusieurs ».

Comme vous pouvez le constater, on a tendance à écrire les messages de commit au présent. D'autre part, vous remarquerez que la plupart des projets open source sont écrits en anglais, donc il est fréquent de voir des messages de commit en anglais.

Une fois le message de commit enregistré, Git va officiellement sauvegarder vos changements dans un commit. Il ajoute donc cela à la liste des changements qu'il connaît du projet.



git commit *ajoute vos dernières modifications à votre historique des modifications.*

Un commit avec git est **local** : à moins d'envoyer ce commit sur le serveur comme on apprendra à le faire plus loin, personne ne sait que vous avez fait ce commit pour le moment. Cela a un avantage : si vous vous rendez compte que vous avez fait une erreur dans votre dernier commit, vous avez la possibilité de l'annuler (ce qui n'est pas le cas avec SVN !).

Annuler un commit effectué par erreur

Il est fréquent de chercher à comprendre ce qui s'est passé récemment, pourquoi une erreur a été introduite et comment annuler ce changement qui pose problème. C'est même là tout l'intérêt d'un logiciel de gestion de versions comme Git.

Nous allons d'abord apprendre à lire les logs, puis nous verrons comment corriger une erreur.

Que s'est-il passé ? Vérifions les logs

Il est possible à tout moment de consulter l'historique des commits : ce sont les *logs*. Vous pouvez ainsi retrouver tout ce qui a été changé depuis les débuts du projet.

Lorsque vous avez effectué un commit, vous devriez donc le voir dans `git log` :

Code : Console

```
commit 227653fd243498495e4414218e0d4282eef3876e
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date:   Thu Jun 3 08:47:46 2010 +0200

    [TwigBundle] added the javascript token parsers in the helper extension

commit 6261cc26693fa1697bcbbd671f18f4902bef07bc
Author: Jeremy Mikola <jmikola@gmail.com>
Date:   Wed Jun 2 17:32:08 2010 -0400

    Fixed bad examples in doctrine:generate:entities help output.

commit 12328a1bcbf231da8eaf942f8d68c7dc0c7c4f38
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date:   Thu Jun 3 08:42:22 2010 +0200

    [TwigBundle] updated the bundle to work with the latest Twig version
```

Chaque commit est numéroté grâce à un long numéro hexadécimal comme 12328a1bcbf231da8eaf942f8d68c7dc0c7c4f38. Cela permet de les identifier.

Vous pouvez parcourir les logs avec les touches « *Page up* », « *Page down* » et les flèches directionnelles, et quitter en appuyant sur la touche « Q ». Git utilise en fait le programme « less » pour paginer les résultats.

Utilisez l'option *-p* pour avoir le détail des lignes qui ont été ajoutées et retirées dans chaque commit, en tapant `git log -p` :

Code : Console

```
commit 227653fd243498495e4414218e0d4282eef3876e
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date:   Thu Jun 3 08:47:46 2010 +0200

    [TwigBundle] added the javascript token parsers in the helper extension

diff --
git a/src/Symfony/Framework/TwigBundle/Extension/Helpers.php b/src/Symfon
index e4c9cce..57a3d2f 100644
--- a/src/Symfony/Framework/TwigBundle/Extension/Helpers.php
+++ b/src/Symfony/Framework/TwigBundle/Extension/Helpers.php
@@ -33,6 +33,8 @@ class Helpers extends \Twig_Extension
     public function getTokenParsers()
     {
         return array(
+            new JavascriptTokenParser(),
+            new JavascriptsTokenParser(),
             new StylesheetTokenParser(),
```

```
            new StylesheetsTokenParser(),
            new RouteTokenParser(),

commit 6261cc26693fa1697bcbbd671f18f4902bef07bc
Author: Jeremy Mikola <jmikola@gmail.com>
Date:   Wed Jun 2 17:32:08 2010 -0400
```

Vous pouvez avoir un résumé plus court des commits avec `git log --stat` :

Code : Console

```
commit 227653fd243498495e4414218e0d4282eef3876e
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date:   Thu Jun 3 08:47:46 2010 +0200

    [TwigBundle] added the javascript token parsers in the helper extension

 …/Framework/TwigBundle/Extension/Helpers.php     |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

commit 6261cc26693fa1697bcbbd671f18f4902bef07bc
Author: Jeremy Mikola <jmikola@gmail.com>
Date:   Wed Jun 2 17:32:08 2010 -0400

    Fixed bad examples in doctrine:generate:entities help output.

 …/Command/GenerateEntitiesDoctrineCommand.php    |    4 ++--
 1 files changed, 2 insertions(+), 2 deletions(-)
```

Corriger une erreur

Voici différentes méthodes permettant de corriger les erreurs, selon leur ancienneté ou leur importance.

Modifier le dernier message de commit

Si vous avez fait une faute d'orthographe dans votre dernier message de commit ou que vous voulez tout simplement le modifier, vous pouvez le faire facilement grâce à la commande suivante :

Code : Console

```
git commit --amend
```

L'éditeur de texte s'ouvrira à nouveau pour changer le message.

Cette commande est généralement utilisée juste après avoir effectué un commit lorsqu'on se rend compte d'une erreur dans le message. Il est en effet impossible de modifier le message d'un commit lorsque celui-ci a été transmis à d'autres personnes.

Annuler le dernier commit (soft)

Si vous voulez annuler votre dernier commit :

Code : Console

```
git reset HEAD^
```

Cela annule le dernier commit et revient à l'avant-dernier.

Pour indiquer à quel commit on souhaite revenir, il existe plusieurs notations :

- HEAD : dernier commit ;
- HEAD^ : avant-dernier commit ;
- HEAD^^ : avant-avant-dernier commit ;
- HEAD~2 : avant-avant-dernier commit (notation équivalente) ;
- d6d98923868578a7f38dea79833b56d0326fcba1 : indique un numéro de commit précis ;
- d6d9892 : indique un numéro de commit précis (notation équivalente à la précédente, bien souvent écrire les premiers chiffres est suffisant tant qu'aucun autre commit ne commence par les mêmes chiffres).

Seul le commit est retiré de Git ; vos fichiers, eux, restent modifiés.

Vous pouvez alors à nouveau changer vos fichiers si besoin est et refaire un commit.

Annuler tous les changements du dernier commit (hard)

Si vous voulez annuler votre dernier commit **et** les changements effectués dans les fichiers, il faut faire un *reset hard*.

Cela annulera sans confirmation tout votre travail ! Faites donc attention et vérifiez que c'est bien ce que vous voulez faire !

Code : Console

```
git reset --
hard HEAD^   /!\ Annule les commits et perd tous les changements
```

Normalement, Git devrait vous dire quel est maintenant le dernier commit qu'il connaît (le nouveau HEAD) :

Code : Console

```
$ git reset --hard HEAD^
HEAD is now at 6261cc2 Fixed bad examples in doctrine:generate:entities help output
```

Annuler les modifications d'un fichier avant un commit

Si vous avez modifié plusieurs fichiers mais que vous n'avez pas encore envoyé le commit et que vous voulez restaurer un fichier tel qu'il était au dernier commit, utilisez `git checkout` :

Code : Console

```
git checkout nomfichier
```

Le fichier redeviendra comme il était lors du dernier commit.

Annuler/Supprimer un fichier avant un commit

Supposons que vous veniez d'ajouter un fichier à Git avec `git add` et que vous vous apprêtiez à le « commiter ».
Cependant, vous vous rendez compte que ce fichier est une mauvaise idée et vous voudriez annuler votre `git add`.

Il est possible de retirer un fichier qui avait été ajouté pour être « commité » en procédant comme suit :

Code : Console

```
git reset HEAD -- fichier_a_supprimer
```

Télécharger les nouveautés et partager votre travail

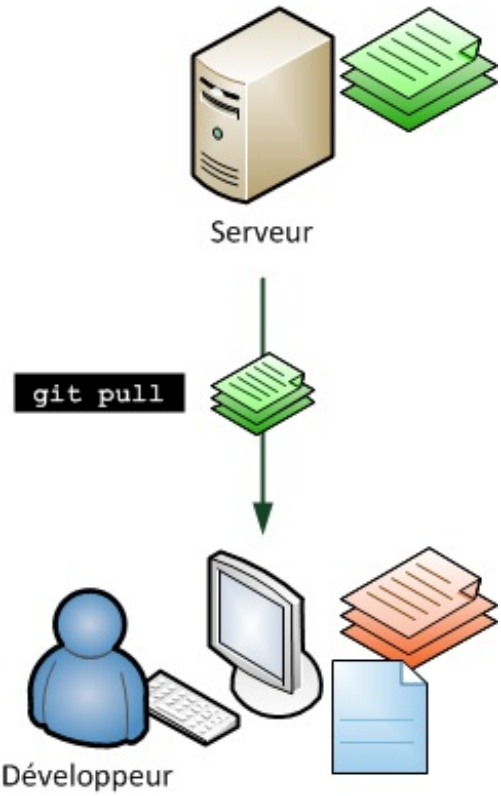
Pour le moment, vous avez tout effectué en local. Comment partager votre travail avec d'autres personnes ?

Télécharger les nouveautés

La commande `git pull` télécharge les nouveautés depuis le serveur :

Code : Console

```
git pull
```



git pull *télécharge les nouvelles modifications effectuées par d'autres personnes.*

Deux cas sont possibles :

- soit vous n'avez effectué aucune modification depuis le dernier pull, dans ce cas la mise à jour est simple (on parle de mise à jour *fast-forward*) ;
- soit vous avez fait des commits en même temps que d'autres personnes. Les changements qu'ils ont effectués sont alors fusionnés aux vôtres automatiquement.

Si deux personnes modifient en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git.

Parfois, mais cela arrive normalement rarement, deux personnes modifient la même zone de code en même temps. Dans ce cas, Git dit qu'il y a un conflit car il ne peut décider quelle modification doit être conservée ; il vous indique alors le nom des fichiers en conflit. Ouvrez-les avec un éditeur et recherchez une ligne contenant « <<<<<<<<< ». Ces symboles délimitent vos changements et ceux des autres personnes. Supprimez ces symboles et gardez uniquement les changements nécessaires, puis faites un nouveau commit pour enregistrer tout cela.

Envoyer vos commits

Vous pouvez envoyer vos commits sur le serveur qui sert de point de rencontre entre les développeurs.

Si vous avez cloné un dépôt Git depuis GitHub comme je l'ai fait précédemment, il y a de fortes chances pour que l'envoi des commits décrit ci-après ne fonctionne pas. Les dépôts publics sont en effet en lecture seule et il faut demander à un développeur officiel du projet de venir récupérer les modifications chez vous (en faisant un pull sur votre ordinateur), afin qu'il puisse vérifier votre travail avant de l'intégrer au projet officiel.

Nous supposons ici que vous avez mis en place un serveur pour votre projet.

Avant d'envoyer vos commits, je vous recommande fortement de consulter votre log local afin de savoir ce que vous vous apprêtez à envoyer :

Code : Console

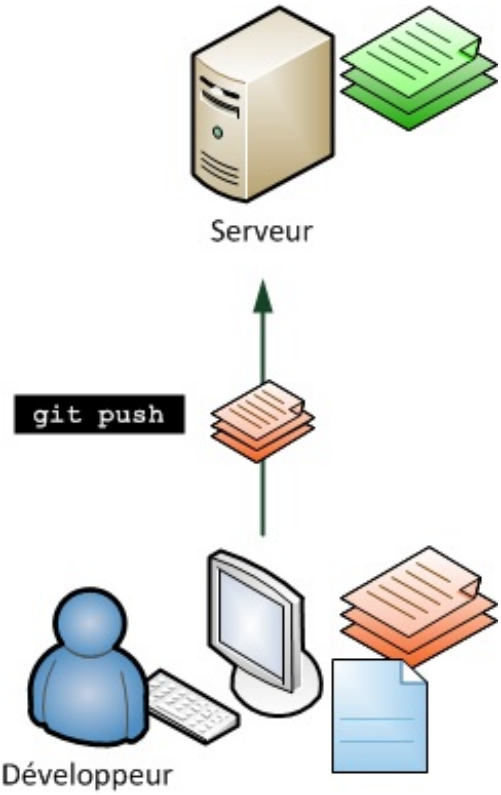
```
git log -p
```

Vérifiez que tout est conforme et qu'il n'y a pas d'erreur qui vous saute aux yeux. Il est encore temps de corriger ces commits, mais une fois envoyés, il sera trop tard !

Une fois que vous êtes sûrs, passez à l'envoi. Vous pouvez envoyer vos commits avec la commande `git push` :

Code : Console

```
git push
```



git push *envoie vos nouvelles modifications (commits) sur le serveur.*

Le changement vers le serveur doit être de type *fast-forward* car le serveur ne peut régler les conflits à votre place s'il y en a. **Personne ne doit avoir fait un push avant vous depuis votre dernier pull.**

Le mieux est de s'assurer que vous êtes à jour en faisant un pull avant de faire un push. Si le push échoue, vous serez de toute façon invités à faire un pull.

Il est recommandé de faire régulièrement des commits, mais pas des push. Vous ne devriez faire un push qu'une fois de temps en temps (pas plus d'une fois par jour en général). Evitez de faire systématiquement un push après chaque commit. Pourquoi ? Parce que vous perdriez la facilité avec laquelle il est possible d'annuler ou modifier un commit.

Un push est irréversible. Une fois que vos commits sont publiés, il deviendra impossible de les supprimer ou de modifier le message de commit ! Réfléchissez donc à deux fois avant de faire un push. C'est pour cela que je recommande de ne faire un push qu'une fois par jour, afin de ne pas prendre l'habitude d'envoyer définitivement chacun de vos commits trop vite.

Annuler un commit publié

Dans le cas malheureux où vous auriez quand même envoyé un commit erroné sur le serveur, il reste possible de l'annuler… en créant un nouveau commit qui effectue l'inverse des modifications (souvenez-vous qu'il n'est pas possible de supprimer un vieux commit envoyé, on ne peut qu'en créer de nouveaux.). Les lignes que vous aviez ajoutées seront supprimées dans ce commit, et inversement.

Jetez tout d'abord un œil à votre `git log` :

Code : Console

```
commit 227653fd243498495e4414218e0d4282eef3876e
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date:   Thu Jun 3 08:47:46 2010 +0200

    [TwigBundle] added the javascript token parsers in the helper extension
```

```
commit 6261cc26693fa1697bcbbd671f18f4902bef07bc
Author: Jeremy Mikola <jmikola@gmail.com>
Date:   Wed Jun 2 17:32:08 2010 -0400

    Fixed bad examples in doctrine:generate:entities help output.
```

Supposons que vous vouliez annuler le commit 6261cc2 dans cet exemple. Il faut utiliser `git revert` qui va créer un commit « inverse » :

Code : Console

```
git revert 6261cc2
```

Il faut préciser l'ID du commit à « revert ». Je vous rappelle qu'il n'est pas obligatoire d'indiquer l'ID en entier (qui est un peu long), il suffit de mettre les premiers chiffres tant qu'ils sont uniques (les 4-5 premiers chiffres devraient suffire).

On vous invite à entrer un message de commit. Un message par défaut est indiqué dans l'éditeur.

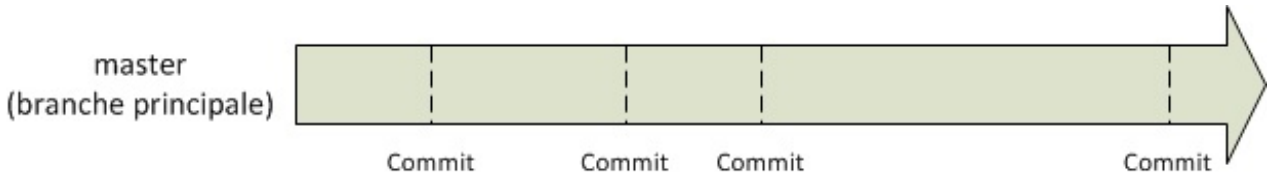
Une fois que c'est enregistré, le commit d'annulation est créé. Il ne vous reste plus qu'à vérifier que tout est bon et à le publier (avec un `git push`).

Travailler avec des branches

Les branches font partie du cœur même de Git et constituent un de ses principaux atouts. C'est un moyen de travailler en parallèle sur d'autres fonctionnalités. C'est comme si vous aviez quelque part une « copie » du code source du site qui vous permet de tester vos idées les plus folles et de vérifier si elles fonctionnent avant de les intégrer au véritable code source de votre projet.

Bien que les branches soient « la base » de Git, je n'en ai pas parlé avant pour rester simple. Pourtant, il faut absolument les connaître et s'en servir. La gestion poussée des branches de Git est la principale raison qui incite les projets à passer à Git, donc il vaut mieux comprendre comment ça fonctionne et en faire usage, sinon on passe vraiment à côté de quelque chose.

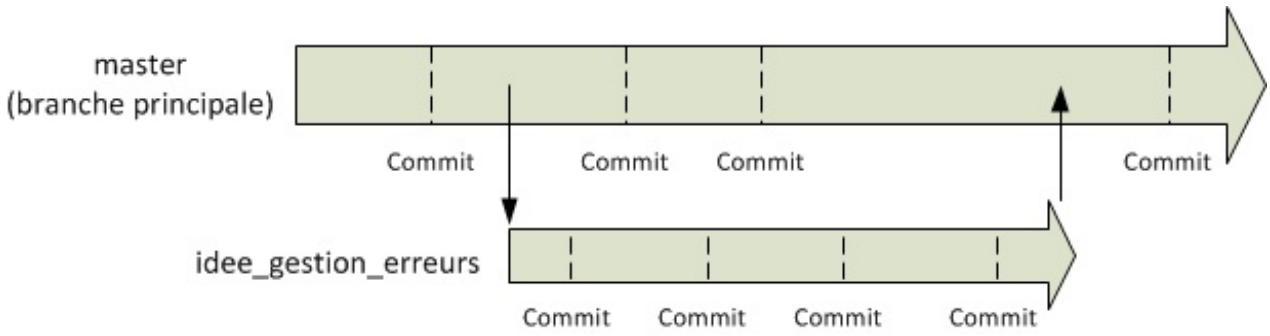
Dans Git, toutes les modifications que vous faites au fil du temps sont par défaut considérées comme appartenant à la branche principale appelée « master » :



On voit sur ce schéma les commits qui sont effectués au fil du temps.

Supposons que vous ayez une idée pour améliorer la gestion des erreurs dans votre programme mais que vous ne soyez pas sûrs qu'elle va fonctionner : vous voulez faire des tests, ça va vous prendre du temps, donc vous ne voulez pas que votre projet incorpore ces changements dans l'immédiat.

Il suffit de créer une branche, que vous nommerez par exemple « idee_gestion_erreurs », dans laquelle vous allez pouvoir travailler *en parallèle* :



À un moment donné, nous avons décidé de créer une nouvelle branche. Nous avons pu y faire des commits, mais cela ne nous a pas empêché de continuer à travailler sur la branche principale et d'y faire des commits aussi.

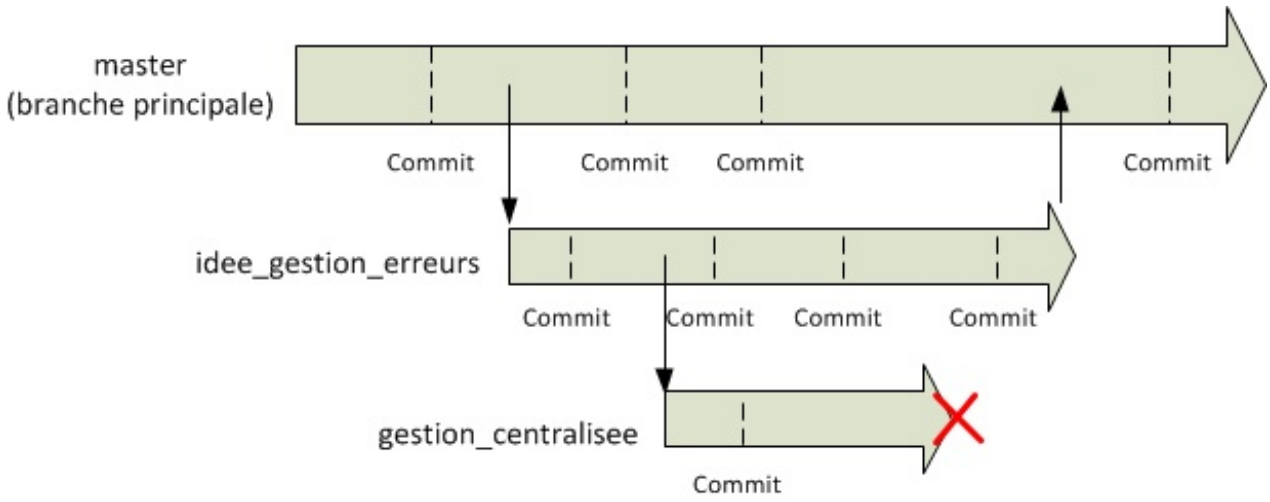
À la fin, mon idée s'est révélée concluante, j'ai largement amélioré la gestion des erreurs et j'ai donc intégré les changements dans la branche principale « master ». Mon projet dispose maintenant de mon idée que j'avais développée en parallèle. Tous les commits de ma branche se retrouvent fusionnés dans la branche principale.

Git n'est pas le seul outil capable de gérer des branches, mais il est le seul à le faire *aussi bien*. En effet, en temps normal vous pourriez tout simplement copier le répertoire de votre projet dans un autre dossier, tester les modifications et les incorporer ensuite dans le véritable dossier de votre projet. Mais cela aura nécessité de copier tous les fichiers et de se souvenir de tout ce que vous avez modifié. Et je ne vous parle même pas du cas où quelqu'un aurait modifié le même fichier que vous en même temps dans la branche principale !

Git gère tous ces problèmes pour vous. Au lieu de créer une copie des fichiers, il crée juste une branche « virtuelle » dans laquelle il retient vos changements en parallèle. Lorsque vous décidez de fusionner une branche (et donc de ramener vos changements dans « master » pour les valider), Git vérifie si vos modifications n'entrent pas en conflit avec des commits effectués en parallèle. S'il y a des conflits, il essaie de les résoudre tout seul ou vous avertit s'il a besoin de votre avis (c'est le cas si deux personnes ont modifié la même ligne d'un même fichier par exemple).

Ce concept de branches très légères qui ne nécessitent pas de copier les fichiers est d'une grande puissance. Cela vous encourage à créer des branches tout le temps, pour toutes les modifications qui pourraient prendre du temps avant d'être terminées.

Vous pouvez même créer une sous-branche à partir d'une branche !



Dans le cas ci-dessus, ma sous-branche ne s'est pas révélée concluante ; j'ai donc dû la supprimer et aucun des commits intermédiaires ne sera finalement incorporé à la branche principale du projet.

Les branches locales

Tout le monde commence avec une seule branche « master » : c'est la branche principale. Jusqu'ici, vous avez donc travaillé dans la branche « master », sur le « vrai » code source de votre projet.

Pour voir toutes vos branches, tapez ceci :

Code : Console

```
git branch
```

Vous verrez normalement uniquement « master » :

Code : Console

```
$ git branch
* master
```

Il y a une étoile devant pour indiquer que c'est la branche sur laquelle vous êtes actuellement.

Pourquoi créer une branche et quand dois-je en créer une ?

Lorsque vous vous apprêtez à faire des modifications sur le code source, posez-vous les questions suivantes :

- « Ma modification sera-t-elle rapide ? » ;
- « Ma modification est-elle simple ? » ;
- « Ma modification nécessite-t-elle un seul commit ? » ;
- « Est-ce que je vois précisément comment faire ma modification d'un seul coup ? ».

Si la réponse à l'une de ces questions est « non », vous devriez probablement créer une branche. Créer une branche est très simple, très rapide et très efficace. Il ne faut donc pas s'en priver.

Créez une branche pour toute modification que vous vous apprêtez à faire et qui risque d'être un peu longue.

Au pire, si votre modification est plus courte que prévu, vous aurez créé une branche « pour pas grand-chose », mais c'est toujours mieux que de se rendre compte de l'inverse.

Créer une branche et changer de branche

Supposons que vous vouliez « améliorer la page des options membres » du code de votre site. Vous n'êtes pas sûrs du temps que cela va prendre, ce n'est pas un changement simple qui consiste à modifier deux-trois liens et vous risquez de faire plusieurs commits. Bref, il faut créer une branche pour cela.

Code : Console

```
git branch options_membres
```

Cela crée une branche appelée « options_membres ». Il est important de noter que cette branche est locale : vous seuls y avez accès (il est néanmoins possible de publier une branche pour que d'autres personnes puissent vous aider, mais ce n'est pas le sujet pour le moment).

Une fois la branche créée, vous devriez la voir quand vous tapez simplement `git branch` :

Code : Console

```
$ git branch
* master
  options_membres
```

Comme vous pouvez le voir, vous êtes toujours sur la branche « master ». Pour aller sur la branche « options_membres », tapez ceci :

Code : Console

```
git checkout options_membres
```

`git checkout` est utilisé pour changer de branche mais aussi pour restaurer un fichier tel qu'il était lors du dernier commit. La commande a donc un double usage.

Qu'est-ce qui se passe lorsque l'on change de branche ? En fait, vous ne changez pas de dossier sur votre disque dur, mais Git change vos fichiers pour qu'ils reflètent l'état de la branche dans laquelle vous vous rendez. Imaginez que les branches dans Git sont comme des dossiers virtuels : vous « sautez » de l'un à l'autre avec la commande `git checkout`. Vous restez dans le même dossier, mais Git modifie les fichiers qui ont changé entre la branche où vous étiez et celle où vous allez.

Faites maintenant des modifications sur les fichiers, puis un commit, puis d'autres modifications, puis un commit, etc. Si vous faites `git log`, vous verrez tous vos récents commits.

Maintenant, supposons qu'un bug important ait été détecté sur votre site et que vous deviez le régler immédiatement. Revenez sur la branche « master », branche principale du site :

Code : Console

```
git checkout master
```

Faites vos modifications, un commit, éventuellement un push s'il faut publier les changements de suite, etc.

Ensuite, revenez à votre branche :

Code : Console

```
git checkout options_membres
```

Si vous faites un `git log`, vous verrez que le commit que vous avez effectué sur la branche « master » n'apparaît pas. C'est en cela que les branches sont distinctes.

Fusionner les changements

Lorsque vous avez fini de travailler sur une branche et que celle-ci est concluante, il faut « fusionner » cette branche vers « master » avec la commande `git merge`.

Avec Git, on peut fusionner n'importe quelle branche dans une autre branche, bien que le plus courant soit de fusionner une branche de test dans « master » une fois que celle-ci est concluante.

Supposons que vous ayez fini votre travail dans la branche « options_membres » et que vous vouliez maintenant le publier. Pour cela, il faut fusionner le contenu de la branche « options_membres » dans la branche principale « master ».

Rendez-vous d'abord dans la branche « master » :

Code : Console

```
git checkout master
```

Demandez ensuite à y intégrer le travail que vous avez fait dans « options_membres » :

Code : Console

```
git merge options_membres
```

Tous vos commits de la branche « options_membres » se retrouvent maintenant dans « master » ! Vous avez pu travailler en parallèle sans gêner la branche principale, et maintenant que votre travail est terminé vous l'avez appliqué sur « master » !

Lorsque vous récupérez les nouveaux commits depuis le serveur avec `git pull`, cela revient en fait à appeler deux commandes différentes : `git fetch`, qui s'occupe du téléchargement des nouveaux commits, et `git merge`, qui fusionne les commits téléchargés issus de la branche du serveur *dans* la branche de votre ordinateur !

Votre branche « options_membres » ne servant plus à rien, vous pouvez la supprimer :

Code : Console

```
git branch -d options_membres
```

Git vérifie que votre travail dans la branche « options_membres » a bien été fusionné dans « master ». Sinon, il vous en avertit et vous interdit de supprimer la branche (vous risqueriez sinon de perdre tout votre travail dans cette branche !).

Si vraiment vous voulez supprimer une branche même si elle contient des changements que vous n'avez pas fusionnés (par exemple parce que votre idée à la base était une erreur), utilisez l'option *-D* (lettre capitale) :

Code : Console

```
git branch -
D options_membres  /!\ Supprime la branche et perd tous les changements.
```

Mettre de côté le travail en cours avant de changer de branche

Cette section est un peu complexe. Je vous recommande de réserver sa lecture pour plus tard.

Avant de changer de branche, vous devez avoir effectué un commit de tous vos changements. En clair, un `git status` ne devrait afficher aucun fichier en cours de modification.

Si vous avez des changements non « commités » et que vous changez de branche, les fichiers modifiés resteront comme ils étaient dans la nouvelle branche (et ce n'est en général pas ce que vous voulez !).

Pour éviter d'avoir à faire un commit au milieu d'un travail en cours, tapez :

Code : Console

```
git stash
```

Vos fichiers modifiés seront sauvegardés et mis de côté. Maintenant, `git status` ne devrait plus afficher aucun fichier (on dit que votre *working directory* est propre).

Vous pouvez alors changer de branche, faire vos modifications, « committer », puis revenir sur la branche où vous étiez.

Code : Console

```
git checkout master
(modifier des fichiers)
git commit -a
git checkout mabranche
```

Pour récupérer les changements que vous aviez mis de côté dans « mabranche », tapez :

Code : Console

```
git stash apply
```

Vos fichiers seront alors restaurés et se retrouveront donc l'état dans lequel ils étaient avant le `git stash` !

Les branches partagées

Il est possible de travailler à plusieurs sur une même branche. En fait, c'est déjà ce que vous faisiez en travaillant sur la branche « master ».

Utilisez `git branch -r` pour lister toutes les branches que le serveur connaît :

Code : Console

```
$ git branch -r
  origin/HEAD
  origin/master
```

origin, c'est le nom du serveur depuis lequel vous avez cloné le dépôt (par exemple celui de GitHub). Vous pouvez en théorie suivre directement les branches de plusieurs personnes (souvenez-vous, Git fonctionne un peu comme le peer-to-peer), mais on travaille plutôt avec un serveur pour suivre les changements.

Si on met de côté « HEAD » qui est un peu particulier, on voit qu'il y a une seule branche sur le serveur : « master ». Le serveur ne connaît donc que l'historique de la branche principale.

Si le serveur possède une autre branche, par exemple « origin/options_membres », et que vous souhaitez travailler dessus, il faut créer une copie de cette branche sur votre ordinateur qui va « suivre » (on dit *tracker*) les changements sur le serveur.

Code : Console

```
git branch --track branchelocale origin/brancheserveur
```

Par exemple, vous pouvez créer une branche « options_membres » locale qui sera connectée à la branche « options_membres »

du serveur :

Code : Console

```
git branch --track options_membres origin/options_membres
```

Lorsque vous ferez un pull depuis la branche « options_membres », les changements seront fusionnés dans votre « options_membres » local. Il est donc important de savoir dans quelle branche vous vous trouvez avant de faire un pull. Un pull depuis la branche « master » met à jour votre branche « master » locale en fonction de ce qui a changé sur le serveur, et il en va de même pour n'importe quelle autre branche.

Ajouter ou supprimer une branche sur le serveur

Il est possible d'ajouter des branches sur le serveur pour y travailler à plusieurs, mais il faut reconnaître que la syntaxe proposée par Git est tout sauf claire à ce niveau.

Voici comment on ajoute une branche sur le serveur :

Code : Console

```
git push origin origin:refs/heads/nom_nouvelle_branche
```

Vous pouvez ensuite créer une branche locale qui « suit » la branche du serveur avec `git branch --track`, comme nous l'avons vu précédemment.

L'inverse est possible : créer une branche locale puis la copier sur le serveur. Pour ce faire, vous devez créer la branche sur le serveur comme expliqué juste au-dessus, mais plutôt que de « tracker » les modifications, modifiez le fichier *.git/config* comme ceci :

- copiez le bloc [branch "master"] ;
- remplacez les occurrences de « master » par le nom de votre branche ;
- enregistrez les modifications.

Faites ensuite un `git pull` et un `git push` ; normalement vos modifications devraient être mises à jour sur le serveur.

Pour supprimer une branche sur le serveur :

Code : Console

```
git push origin :heads/nom_branche_a_supprimer
```

À noter que les « *remote tracking branches* » (celles qui apparaissent lorsqu'on fait `git branch -r`) ne seront pas automatiquement supprimées chez les autres clients. Il faut qu'ils les suppriment manuellement à l'aide de la commande suivante :

Code : Console

```
git branch -r -d origin/nom_branche_a_supprimer
```

Quelques autres fonctionnalités de Git, en vrac

Je vous ai présenté les principales fonctionnalités de Git, mais il est possible de faire bien d'autres choses avec cet outil. Voici, en vrac, quelques-unes de ses autres possibilités utiles à connaître.

Tagger une version

Il est possible de donner un alias à un commit précis pour le référencer sous ce nom. C'est utile par exemple pour dire « À tel commit correspond la version 1.3 de mon projet ». Cela permettra à d'autres personnes de repérer la version 1.3 plus facilement. C'est justement le rôle des tags.

Pour ajouter un tag sur un commit :

Code : Console

```
git tag NOMTAG IDCOMMIT
```

Donc dans le cas présent, on écrirait :

Code : Console

```
git tag v1.3 2f7c8b3428aca535fdc970feeb4c09efa33d809e
```

Un tag n'est pas envoyé lors d'un push, il faut préciser l'option *--tags* pour que ce soit le cas :

Code : Console

```
git push --tags
```

Maintenant, tout le monde peut référencer ce commit par ce nom plutôt que par son numéro de révision.

Pour supprimer un tag créé :

Code : Console

```
git tag -d NOMTAG
```

Rechercher dans les fichiers source

Comme Git connaît tous les fichiers source de votre projet, il est facile de faire une recherche à l'intérieur de tout votre projet.

Par exemple, si vous voulez connaître les noms des fichiers qui contiennent le mot TODO dans le code source, il suffit d'écrire :

Code : Console

```
git grep "TODO"
```

Git accepte les expressions régulières pour les recherches. Si vous connaissez les expressions régulières, sachez donc qu'il est possible de les utiliser.

Si vous voulez connaître les numéros des lignes qui contiennent le mot que vous recherchez, utilisez le paramètre *-n* :

Code : Console

```
git grep -n "TODO"
```

Demander à Git d'ignorer des fichiers (.gitignore)

Pour ignorer un fichier dans git, créez un fichier *.gitignore* (à la racine) et indiquez-y le nom du fichier. Entrez un nom de fichier par ligne, comme ceci :

Code : Autre

```
project.xml
dossier/temp.txt
*.tmp
cache/*
```

Aucun de ces fichiers n'apparaîtra dans `git status`, même s'il est modifié. Il ne paraîtra donc pas dans les commits. Utilisez cela sur les fichiers temporaires par exemple, qui n'ont aucune raison d'apparaître dans Git.

Il est possible d'utiliser une étoile (*) comme joker. Dans l'exemple précédent, tous les fichiers ayant l'extension « .tmp » seront ignorés, de même que tous les fichiers du dossier *cache*.
Git est un outil très complet qui peut parfois se révéler complexe. Comme tout bon outil UNIX, il fait ce qu'on lui demande sans confirmation, ce qui peut être dangereux entre les mains d'un débutant. Commencez donc à l'utiliser doucement et apprenez petit à petit à découvrir ses autres fonctionnalités : vous allez être surpris !

C'est un outil réellement puissant que je recommande aux développeurs, particulièrement ceux qui travaillent sous Linux. Ce n'est pas le seul qui existe (je pense à Mercurial et Bazaar que vous pouvez également essayer et qui sont très bien) mais il a son lot d'avantages, tels que sa rapidité et sa gestion puissante des branches.

Je vous recommande d'essayer de préférence un logiciel de gestion de versions récent (comme Git, Mercurial ou Bazaar) et d'éviter si vous le pouvez SVN, qui commence à se faire vieux. Si vous utilisez déjà SVN pour votre projet, sachez qu'il est possible de migrer facilement à Git avec l'outil `git-svn`. Il sert aussi à faire le pont entre un serveur SVN et Git, ce qui vous permet d'utiliser Git de votre côté même si votre projet utilise officiellement SVN !

Si vous voulez en savoir plus sur Git, je vous recommande le cours Pro Git (en anglais), qui couvre plus en détail le fonctionnement de Git.

Bonne lecture !

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.

Découverte et utilisation de GitHub

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

GitHub est la plus grande plateforme d'hébergement de projets Git au monde. Vous serez probablement amené à travailler avec GitHub et il est donc important de comprendre comment ce service fonctionne.

Commencez déjà par noter que GitHub est un outil gratuit pour héberger du code open source, et propose également des plans payants pour les projets de code privés.

Pour utiliser GitHub, il suffit de créer un compte gratuitement sur le site https://github.com.

Le grand intérêt de GitHub est de faciliter la collaboration à une échelle planétaire sur les projets : n'importe qui va pouvoir récupérer des projets et y contribuer (sauf si le propriétaire du projet ne le permet pas bien entendu).

Sur GitHub, nous allons en effet notamment pouvoir cloner des projets (dépôts) publics, dupliquer ("fork") des projets ou encore contribuer à des projets en proposant des modifications ("pull request").

Contribuer à un projet avec GitHub ou le copier

Sur GitHub, nous allons pouvoir contribuer aux

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Fonctionnement général de Git

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Dans cette leçon, nous allons présenter le fonctionnement général de Git et définir des éléments de vocabulaire qui vont nous être utiles par la suite.

Démarrer un dépôt Git

Un “dépôt” correspond à la copie et à l’importation de l’ensemble des fichiers d’un projet dans Git. Il existe deux façons de créer un dépôt Git :

- On peut importer un répertoire déjà existant dans Git ;
- On peut cloner un dépôt Git déjà existant.

Nous allons voir comment faire cela dans la suite de ce cours. Avant cela, je pense qu’il est bon de comprendre comment Git conçoit la gestion des informations ainsi que le fonctionnement général de Git.

La gestion des informations selon Git

La façon dont Git considère les données est assez différente de la plupart des autres systèmes de gestion de version.

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Créer un dépôt Git

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Dans cette nouvelle leçon, nous allons utiliser nos premières commandes Git afin de créer un dépôt Git de manière pratique.

Pour rappel, il existe deux façons de créer un dépôt Git : on peut soit initialiser un dépôt Git à partir d'un répertoire déjà existant, soit cloner un dépôt Git déjà existant.

Créer un dépôt Git à partir d'un répertoire existant

Lorsqu'on démarre avec Git, on a souvent déjà des projets en cours stockés localement sur notre machine ou sur serveur distant et pour lesquels on aimerait implémenter un système de gestion de version.

Dans ce cas là, nous allons pouvoir importer l'ensemble des ressources d'un projet dans Git. Pour la suite de cette leçon, je vais créer un répertoire "projet-git" qui se trouve sur mon bureau et qui contient deux fichiers texte vides "fichier1.txt" et "README.txt". Ce répertoire va me servir de base pour les exemples qui vont suivre (ce sera le répertoire importé).

Je vous invite à créer le même répertoire sur votre machine. Vous pouvez le faire soit à la main, soit en utilisant la ligne de commandes

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Enregistrer des modifications et modifier un dépôt Git

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Dans la leçon précédente, nous avons initialisé un premier dépôt Git. Dans cette leçon, nous allons apprendre à enregistrer les modifications effectuées sur notre projet dans Git ainsi qu'à modifier un dépôt Git (enlever des fichiers du suivi, renommer un fichier, etc.).

Ajouter ou modifier des fichiers dans un projet et actualiser notre dépôt Git

A ce niveau, nous avons donc d'un côté notre projet contenant un ensemble de fichiers et de ressources sur lesquelles on travaille ainsi qu'un dépôt Git qui sert à gérer les différentes versions de notre projet.

En continuant à travailler sur notre projet, nous allons être amenés à ajouter, modifier, voire supprimer des fichiers. On va indiquer tous ces changements à Git pour qu'il conserve un historique des versions auquel on pourra ensuite accéder pour revenir à un état précédent du projet (dans le cas où une modification entraine un bogue par exemple ou n'amène pas le résultat souhaité).

A chaque fois qu'on souhaite enregistrer une modification de fichier ou un ajout de fichier dans le dépôt Git, on va devoir utiliser les

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Annuler des actions et consulter l’historique des modifications Git

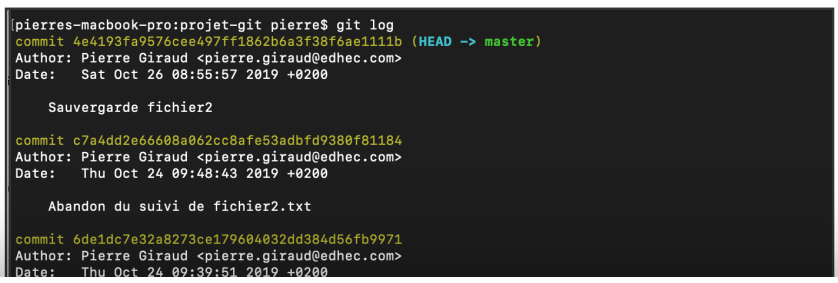
Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Dans la partie précédente, nous avons découvert quelques commandes Git nous permettant de modifier des fichiers et d'enregistrer des modifications dans notre dépôt Git.

Dans cette nouvelle leçon, nous allons apprendre à consulter l'historique des commits réalisés, à remplacer un ancien commit et allons voir comment annuler des modifications apportées à un fichier, c'est-à-dire comment restaurer une version précédente d'un fichier.

Consulter l'historique des modifications Git

La manière la plus simple de consulter l'historique des modifications Git est d'utiliser la commande git log. Cette commande affiche la liste des commits réalisés du plus récent au plus ancien. Par défaut, chaque commit est affiché avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.



Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Les branches git

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Qu'est ce qu'une branche ?

Créer une branche, c'est en quelque sorte comme créer une "copie" de votre projet pour développer et tester de nouvelles fonctionnalités sans impacter le projet de base.

Dans la plupart des système de contrôle de version, justement, une copie physique de la totalité du répertoire de travail est effectuée, ce qui rend la création de branches contraignante et en fait une opération lourde.

Git a une approche totalement différente des branches qui rend la création de nouvelles branches et la fusion de branche très facile à réaliser. Une branche, dans Git, est simplement un pointeur vers un commit (une branche n'est qu'un simple fichier contenant les 40 caractères de l'empreinte SHA-1 du commit sur lequel elle pointe).

Pour rappel, lorsqu'on effectue un commit, Git stocke en fait un objet commit qui contient les nom et prénom de l'auteur du commit, le message renseigné lors de la création du commit ainsi qu'un pointeur vers l'instantané du contenu indexé et des pointeurs vers le ou les commits précédant directement le commit courant.

Un pointeur est un objet qui contient l'adresse

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Fusion et rebasage avec Git

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Dans la partie précédente, on a vu comment créer différentes branches pour un projet. généralement, on va créer de nouvelles branches pour travailler sur de nouvelles fonctionnalités pour notre projet sans impacter la ligne de développement principale (représentée par notre branche principale).

On va donc développer nos fonctionnalités sur des branches connexes et les tester jusqu'à ce qu'on soit sûrs qu'il n'y a pas de problème et on va finalement réintégrer ces fonctionnalités développées au sein de notre ligne de développement principale.

Pour faire cela, il va falloir rapatrier le contenu des branches créées dans la branche principale. On peut faire ça de deux manières avec Git : en fusionnant les branches ou en rebasant.

Fusionner des branches

Commençons par nous concentrer sur sur la fusion de branches.

Dans la leçon précédente, on avait fini avec deux branches master et test divergentes. On parle de divergence car les deux branches possèdent un ancêtre commit en commun mais pointent chacune vers de nouveaux

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Travailler avec des dépôts Git distants

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Jusqu'à présent, nous avons utilisé Git pour mettre en place un système de gestion de version local. En pratique, vous allez souvent être amené à travailler à plusieurs sur un même projet.

Pour collaborer sur un projet et mettre en place un système de gestion de version, vous allez devoir utiliser des dépôts distants c'est-à-dire des dépôts hébergés sur serveur distant (serveurs accessibles via Internet ou via votre réseau d'entreprise).

La gestion de dépôts distants est plus complexe que la gestion d'un dépôt local puisqu'il va falloir gérer les permissions des différents utilisateurs, définir quelles modifications doivent être adoptées ou pas, etc.

Dans le cas où on travaille à plusieurs sur un dépôt distant, nous n'allons jamais directement modifier le projet distant. Nous allons plutôt cloner le dépôt distant localement (sur notre ordinateur donc), effectuer nos modifications puis les pousser vers le dépôt distant. Ces modifications pourront ensuite être acceptées ou pas.

Cloner, afficher et renommer un dépôt distant

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

Laisser un commentaire

Vous devez être connecté pour publier un commentaire.

© Pierre Giraud - Toute reproduction interdite

Ce site utilise des cookies pour vous fournir la meilleure expérience de navigation possible. En continuant sur ce site, vous acceptez l'utilisation des cookies. Réglages ACCEPTER

GitHub Pour les Nuls : Pas de Panique, Lancez-Vous ! (Première Partie)

christophe ducamp 15 décembre 2013 - 3000 mots

Traduction d'un article original de [Lauren Orsini] (http://otakujournalist.com/about-the-author/) publié le 30 septembre 2013 pour ReadWriteWeb. Seul le [lien original fait référence](http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1).
La traduction reste à raffiner avec la pratique de cet outil. Seconde partie en cours d'étude pour me lancer sous peu dans les premiers *commits* à la ligne de commande. Mise en forme prévue pour le plan de route indieweb 2014. Merci. - xtof_fr

GitHub est plus qu'un simple outil de programmation. Si vous voulez collaborer sur n'importe quoi, vous devriez l'essayer. 1ère Partie pour apprendre à démarrer sur GitHub.

Nous sommes en 2013 et pas moyen d'y échapper : vous devrez apprendre comment utiliser GitHub.

Pourquoi ? Parce que c'est un réseau social qui change drastiquement notre façon de travailler. Ayant démarré sous forme de plateforme collaborative pour développeurs, GitHub est désormais le plus grand espace de stockage de travaux collaboratifs dans le monde. Que vous soyez intéressé(e) pour participer à ce cerveau global ou tout simplement pour partir à la

recherche de cet énorme réservoir de connaissances, vous vous devez d'y être.

En étant simplement membre, vous pourrez croiser le fer avec ce qu'aiment Google et Facebook. Avant que Github n'existe, les grandes sociétés créaient leurs bases de connaissance surtout en privé. Mais lorsque vous accédez à leurs comptes GitHub, vous êtes libres de télécharger, étudier et construire dessus tout ce que vous voulez sur ce qu'elles ajoutent sur ce réseau. Aussi, qu'attendez-vous ?

Chercher des Réponses GitHub

Aussi gênant que cela puisse paraître, j'ai écrit ce tutoriel parce que je me sentais vraiment perdue dans tous les articles de type "GitHub pour Débutants". Probablement parce qu'à la différence de la plupart des utilisateurs de Github, je manque de bagage solide en programmation. Et je ne m'y retrouvais pas non plus dans les tutoriels d'utilisation de Github, pour construire une vitrine de quelques travaux de programmation.

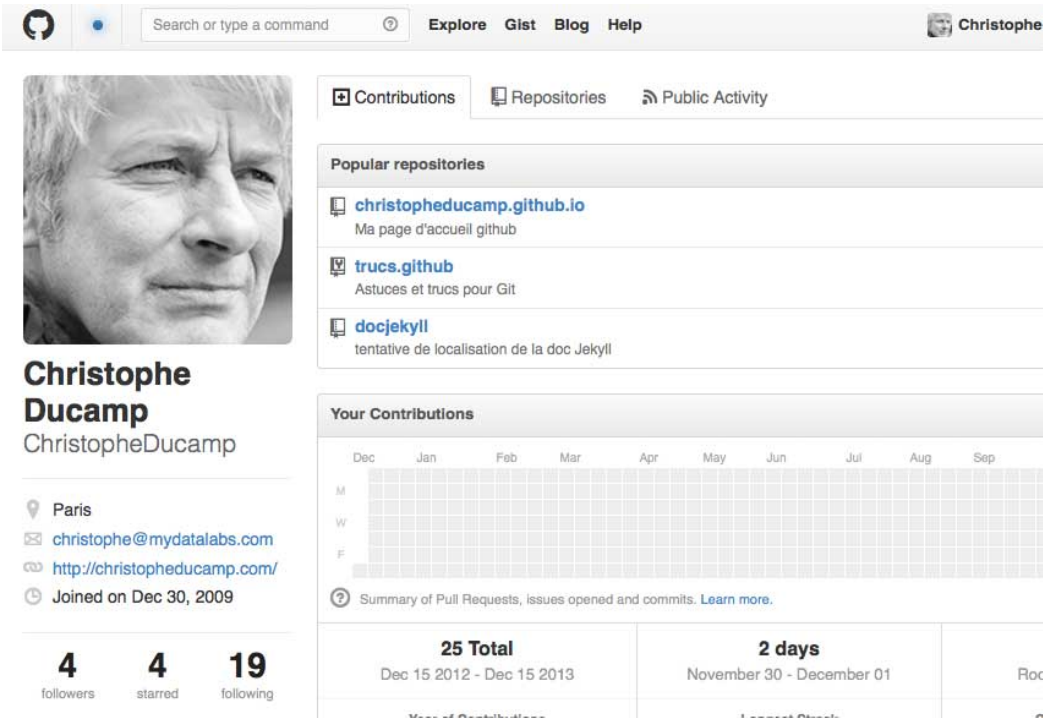
Voir aussi : Tom Preston-Werner de Github : Comment Nous Sommes Devenus Mainstream

Ce que vous pourriez ignorer, c'est qu'il existe plein de raisons d'utiliser GitHub même si vous n'êtes pas programmeur. Selon les vidéos de tutoriels GitHub, tout travailleur du savoir peut en tirer profit, "knowledge worker" s'entendant ici pour désigner la plupart des professionnels faisant usage d'un ordinateur.

Par conséquent, si vous avez déjà lâché prise sur la compréhension d'utilisation de Github, cet article est pour vous.

L'un des principaux malentendus concernant GitHub est que c'est un outil de développement, faisant partie de la panoplie de tout

programmeur, comme le sont les langages de programmation et les compilateurs. Cependant, GitHub en lui-même n'est rien de plus qu'un réseau social comme Facebook ou Flickr. Vous construisez un profil, vous y déposez des projets à partager et vous vous connectez avec d'autres utilisateurs en suivant leurs comptes. Même si la plupart des utilisateurs y déposent des projets de programmes ou de code, rien ne vous empêche d'y placer des textes ou tout type de fichier à présenter dans vos répertoires de projets.



Vous avec peut-être déjà plus d'une dizaine de comptes sociaux… et voici pourquoi vous devriez être sur Github : il dispose des meilleures Conditions Générales d'Utilisation. Si vous regardez la section F des conditions générales, vous verrez que Github fait tout pour vous assurer que vous conservez la propriété complète de tous les projets que vous déposez sur le site :

> *We claim no intellectual property rights over the material you provide to the Service. Your profile and materials uploaded remain yours.*

> *conditions générales GitHub*

En outre, vous pouvez véritablement utiliser GitHub sans connaître UNE SEULE LIGNE de code. Vous n'avez pas besoin de tutoriel pour vous enregistrer et vous promener. Mais mon point de vue est que GitHub a le mérite de nous apprendre à faire les choses dures en premier, ce qui veut dire, utiliser le bon vieux code Git. Après tout, GitHub est parvenu à produire l'une des interfaces graphiques sans effort pour le langage de programmation Git.

C'est Quoi Git ?

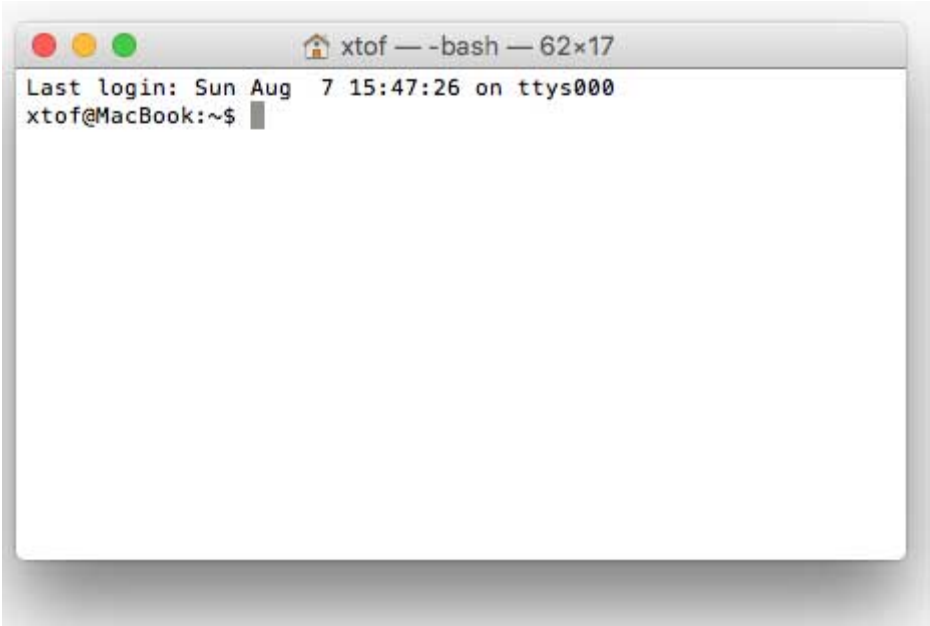
Remercions le célèbre développeur de logiciel Linus Torvalds pour Git, le logiciel qui fait tourner le coeur de GitHub. (Et tant que vous y êtes, remercions-le aussi pour le système d'exploitation Linux). **Git est un logiciel de contrôle de version**, ce qui signifie qu'il gère les modifications d'un projet sans écraser n'importe quelle partie du projet. Et il ne risque pas de disparaître, tout particulièrement parce que Torvalds et ses collègues développeurs du noyau utilisent Git pour aider à développer le noyau coeur de Linux.

Pourquoi utiliser quelque chose comme Git ? Supposons que vous mettiez à jour avec un collègue des pages sur le même site web. Vous faites des modifications, vous les sauvegardez et les versez sur le site. À ce stade, tout va bien. Le problème survient quand votre collègue travaille sur la même page que vous en même temps. L'un de vous va voir son travail écrasé.

Une application de contrôle de version comme Git empêche ça d'arriver. Vous et votre collègue pouvez chacun de votre côté verser vos révisions sur la même page, et Git sauvegardera deux copies. Plus tard, vous pourrez fusionner vos modifications sans perdre le travail dans le processus. Vous pouvez même revenir en arrière à tout moment, parce que Git conserve une « copie instantanée » de

tous les changements produits.

Le problème avec Git est qu'il est vieux. Si vieux que nous devons utiliser la ligne de commande -ou l'application Terminal si vous êtes sur Mac - afin d'y accéder, et y taper dedans des bouts de code comme les hackers des années 90. Ceci peut être une proposition difficile pour les utilisateurs d'ordinateurs modernes. C'est là où Github entre dans la danse.



GitHub facilite l'utilisation de Git sur deux points. Premièrement, si vous téléchargez le logiciel GitHub sur votre ordinateur, GitHub fournit une interface visuelle pour vous aider à gérer localement vos projets avec les contrôles de version. Deuxièmement, créer un compte sur GitHub.com apporte les contrôles de versions à vos projets web, et leur confère des fonctionnalités de réseaux sociaux.

Vous pouvez parcourir les projets d'autres utilisateurs de Github, et même y télécharger des copies pour vous-même afin de les modifier, apprendre ou les enrichir. D'autres utilisateurs peuvent faire la même chose avec vos projets publics, repérer vos erreurs et suggérer des corrections. De toute façon, aucune donnée ne se perd parce que Git enregistre un "instantané" de chaque

modification.

Bien qu'il soit possible d'utiliser GitHub sans apprendre Git, il y a une énorme différence entre l'utilisation et la compréhension. Avant de connaître Git, je savais utiliser GitHub, mais je ne comprenais pas vraiment pourquoi. Dans ce tutoriel, nous allons apprendre à utiliser Git à la ligne de commande.

Les Mots que les Personnes Utilisent quand Elles Parlent de Git

Dans ce tutoriel, il y a quelques mots que j'utiliserai à plusieurs reprises, aucun d'eux dont je n'avais entendu parler avant d'avoir démarré l'apprentissage. Voici les plus connus :

Ligne de Commande : Le programme de l'ordinateur que nous utilisons pour entrer des commandes Git. Sur un Mac, ça s'appelle Terminal. Sur un PC, c'est un programme non-natif que vous téléchargez lorsque vous téléchargez Git pour la première fois (nous allon faire ça dans la section suivante). Dans les deux cas, vous tapez à l'écran des commandes à base de texte, appelées invites de commande, au lieu d'utiliser une souris.

Dépôt : Un répertoire ou de l'espace de stockage où vos projets peuvent vivre. Parfois les utilisateurs GitHub raccourcissent ça en « repo ». Il peut être local sur un répertoire de votre ordinateur, ou ce peut être un espace de stockage sur GitHub ou un autre hébergeur en ligne. À l'intérieur d'un dépôt, Vous pouvez conserver des fichiers de code, des fichiers texte, des images.

Contrôle de Version : Fondamentalement, l'objectif pour lequel Git a été conçu. Quand vous avez un fichier Microsoft Word, vous l'écrasez à chaque fois que vous faites une nouvelle sauvegarde, ou vous sauvegardez plusieurs versions. Avec Git, vous n'êtes plus obligé de faire ça. Git conserve des « instantanés » de chaque point

dans l'historique d'un projet, de sorte que vous ne pouvez jamais le perdre ou l'écraser.

Commit : C'est la commande qui donne à Git toute sa puissance. Quand vous « committez », vous prenez un « instantané », une « photo » de votre dépôt à ce stade, vous donnant un point de contrôle que vous pouvez ensuite réévaluer ou restaurer votre projet à un état précédent.

Branche : Comment faire travailler plusieurs personnes sur un projet en même temps sans que Git ne s'embrouille ? Habituellement, elles se « débranchent » du projet principal avec leurs propres versions complètes des modifications qu'elles ont chacune produites de leur côté. Après avoir terminé, il est temps de « fusionner » cette branche pour la ramener vers la branche « master », le répertoire principal du projet.

Commandes Spécifiques Git

Le fait que Git ait été conçu avec un grand projet comme Linux, il existe beaucoup de commandes Git. Toutefois, pour utiliser les bases de Git, vous aurez seulemenet besoin de connaître quelques termes. Ils commencent tous de la même façon avec le mot « git ».

`git init` : Initialise un nouveau dépôt Git. Jusqu'à ce que vous exécutiez cette commande dans un dépôt ou répertoire, c'est juste un dossier ordinaire. Seulement après avoir entré cette commande, il accepte les commandes Git qui suivent.

`git config` : raccourci de "configurer," ceci est tout particulièrement utile quand vous paramétrez Git pour la première fois.

`git help` : Oublié une commande ? Tapez-ça dans la ligne de commande pour afficher les 21 commandes les plus courantes de

Git. Vous pouvez aussi être plus spécifique et saisir “git help init” ou tout autre terme pour voir comment utiliser et configurer une commande spécifique git.

`git status` : Vérifie le statut de votre repository. Voir quels fichiers sont à l’intérieur, quelles sont les modifications à *commiter*, et sur quelle branche du repository vous êtes en train de travailler.

`git add` : Ceci n’ajoute *pas* de nouveaux fichiers dans votre repository. Au lieu de cela, ceci porte de nouveaux fichiers à l’attention de Git. Après avoir ajouté des fichiers, ils sont inclus dans les « instantanés » du dépôt Git.

`git commit` : la commande la plus importante de Git. Après avoir effectué toute sorte de modification, vous entrez ça afin de prendre un “instantané” du dépôt. Généralement cela s’écrit sous la forme `git commit -m “Message ici“`. Le -m indique que la section suivante de la commande devrait être lue comme un message.

`git branch` : Vous travaillez avec plusieurs collaborateurs et vous voulez produire des modifications de votre côté ? Cette commande vous permet de construire une nouvelle branche, ou une chronologie des commits, des modifications et des ajouts de fichiers qui sont complètement les vôtres. Votre titre va après la commande. Si vous vouliez créer une nouvelle branche appelée « chats », vous saisiriez `git branch chats`.

`git checkout` : Permet littéralement de vérifier un dépôt dans lequel vous n’êtes pas. C’est une commande de navigation qui vous permet de vous déplacer vers le répertoire que vous voulez vérifier. Vous pouvez utiliser cette commande sous la forme `git checkout master` pour regarder la branche master, ou `git checkout chats` pour regarder une autre branche.

`git merge` : Lorsque vous avez fini de travailler sur une branche, vous pouvez fusionner vos modifications vers la branche master, qui est visible pour tous les collaborateurs. `git merge chats` prendrait toutes les modifications que vous avez apportées à la branche “chats” et les ajoutera à la branche master.

`git push` : Si vous travaillez sur votre ordinateur local, et voulez que vos commits soient visibles aussi en ligne sur Github, vous « push »ez les modifications vers Github avec cette commande.

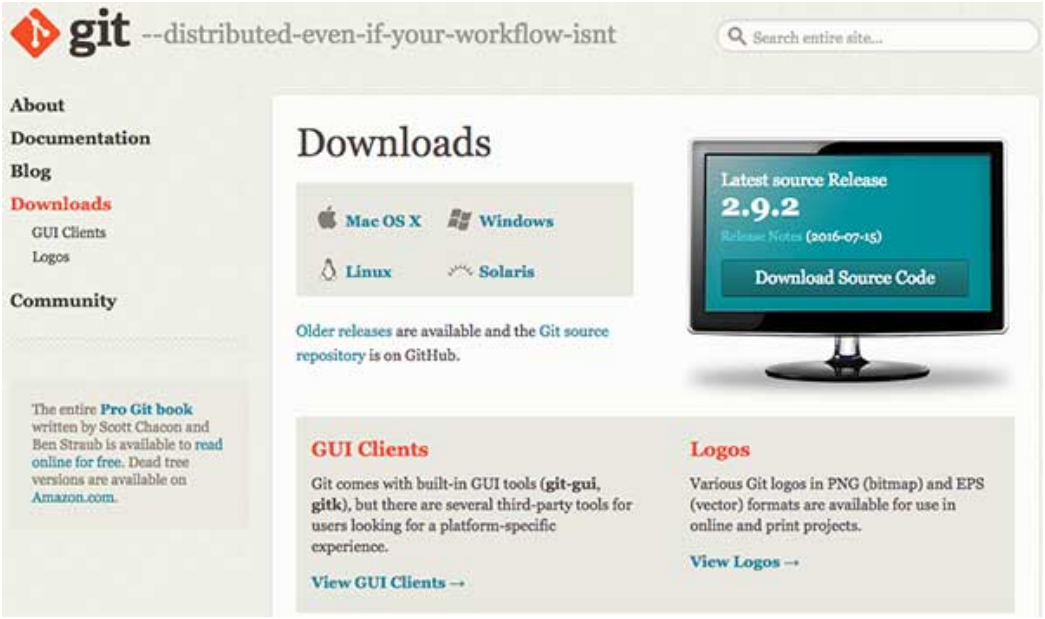
`git pull` : Si vous travaillez sur votre ordinateur local, et que vous voulez la version la plus à jour de votre repository pour travailler dessus, vous “pull”ez (tirez) les modifications provenant de Github avec cette commande.

Paramétrer GitHub ET Git Pour La Première Fois

Premièrement, vous devrez vous enregistrer pour disposer d’un compte sur GitHub.com. C’est aussi simple que de s’enregistrer sur n’importe quel autre réseau social. Conservez l’e-mail que vous avez choisi à portée de main ; nous en aurons besoin de nouveau.

Vous pourriez vous arrêter là et github fonctionnerait bien. Mais si

vous voulez travailler sur votre projet sur votre ordinateur local, vous devez avoir installé Git. En fait, Github ne fonctionnera pas sur votre ordinateur local si vous n'installez pas Git. Téléchargez et installez la dernière version de Git pour Windows, Mac ou Linux selon votre machine.



Maintenant, il est temps de passer à la ligne de commande. Sur Windows, ça veut dire démarrer l'application Git Bash que vous venez d'installer, et sur MacOSX, c'est le bon vieux Terminal. Il est temps de vous présenter à Git. Saisissez le code qui suit :

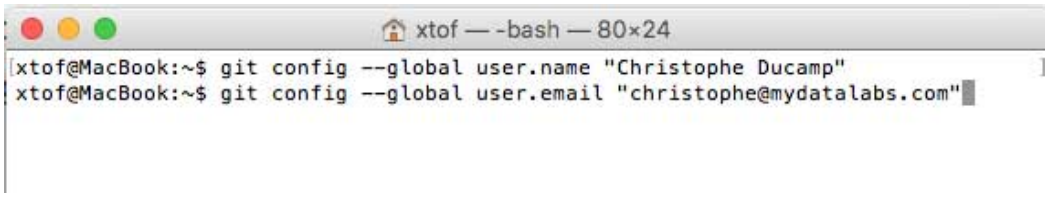
```
git config --global user.name "Votre Nom Ici"
```

Vous aurez bien sûr besoin de remplacer "Votre Nom Ici" par votre propre nom entre guillemets. Ce peut être votre nom légal, votre pseudo en ligne ou tout ce que vous voudrez. Git s'en moque, il a juste besoin de savoir à qui créditer les commits et projets futurs.

Ensuite, indiquez-lui votre adresse de courrier électronique et assurez-vous que c'est le même email que vous avez utilisé pour enregistrer votre compte Github. Faites comme suit :

```
git config --global user.email "votre_email@votre
```

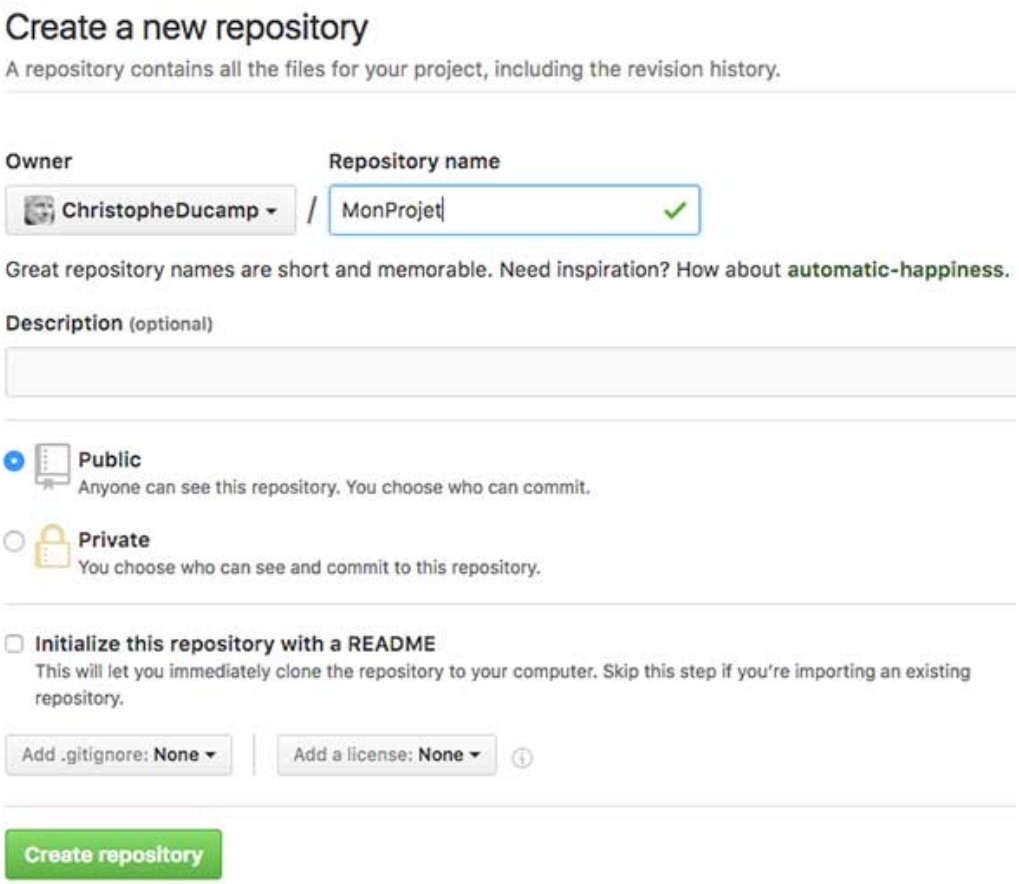
C'est tout ce que vous devez faire pour commencer à utiliser Git sur votre ordinateur. Cependant, puisque vous venez de configurer un compte Github.com, il est probable que vous ne souhaitiez pas gérer simplement votre projet localement, mais aussi en ligne. Si vous le souhaitez, vous pouvez également configurer Git pour qu'il ne vous demande pas de vous connecter à votre compte Github à chaque fois que vous voulez lui parler. Pour les besoins de ce tutoriel, ce n'est pas un grand problème parce que nous ne lui parlerons qu'une fois. Le tutoriel complet pour faire ça est situé sur Github.



Créer Votre Repo En Ligne

Maintenant que vous avez tout mis en place, il est temps de créer un endroit pour placer votre projet à faire vivre. Git et Github appellent cela un repository ou « repo » pour faire court, un répertoire numérique ou un espace de stockage où vous pouvez accéder à votre projet, ses fichiers, et toutes les versions de ses fichiers que Git sauvegarde.

Retournons sur GitHub.com et cliquez sur la petite icône de texte à côté de votre nom d'utilisateur. Ou allez vers la nouvelle page repository si toutes les icônes sont les mêmes. Donnez à votre dépôt un nom court et mémorisable. Allez-y et rendez-le public, pourquoi cacher votre tentative d'apprendre Github !



.

Ne vous inquiétez pas de cliquer sur le bouton radio à côté intitulé “Initialize this repository with a README.” Un fichier Readme est généralement un fichier texte qui explique sommairement le projet. Mais nous pouvons produire localement notre propre fichier `Readme` pour l’entraînement.

Cliquez sur le bouton vert “Create Repository” et c’est fait. Vous avez maintenant un espace en ligne pour votre projet restant à faire vivre.

Créer Votre Repository Local

Ainsi nous venons juste de créer un espace pour votre projet en ligne, mais ce n’est pas l’endroit où nous travaillerons dessus. La majeure partie de votre travail sera faite sur votre ordinateur. Nous

devons donc en fait refléter ce repository que nous venons juste de produire sous un répertoire local.

C'est –là où nous faisons quelques saisies de ligne de commande– la partie de chaque tutoriel Git qui me chahute le plus, aussi j'irai vraiment lentement.

Saisissez d'abord :

```
mkdir ~/MonProjet
```

`mkdir` est le raccourci de « make directory ». Ce n'est en réalité pas une ligne de commande Git, mais une commande générale de navigation provenant du temps avant les interfaces ordinateurs visuelles. Le `~/` veille à vous assurer de construire le repository au niveau supérieur de la structure de fichiers de notre ordinateur, au lieu d'un répertoire coincé dans quelque autre répertoire qui serait plus difficile à retrouver. En fait, si vous saisissez `~/` dans la fenêtre de votre navigateur, cela vous ramènera vers le répertoire local le plus haut de votre ordinateur. Pour moi, en utilisant Chrome sur un Mac, cela affiche mon dossier Users.

Remarquez aussi que je l'ai appelé `MonProjet`, le même nom donné au dépôt Github que nous avons produit précédemment. Assurez-vous aussi de garder une cohérence sur votre nom.

Puis, saisissez :

```
cd ~/MonProjet
```

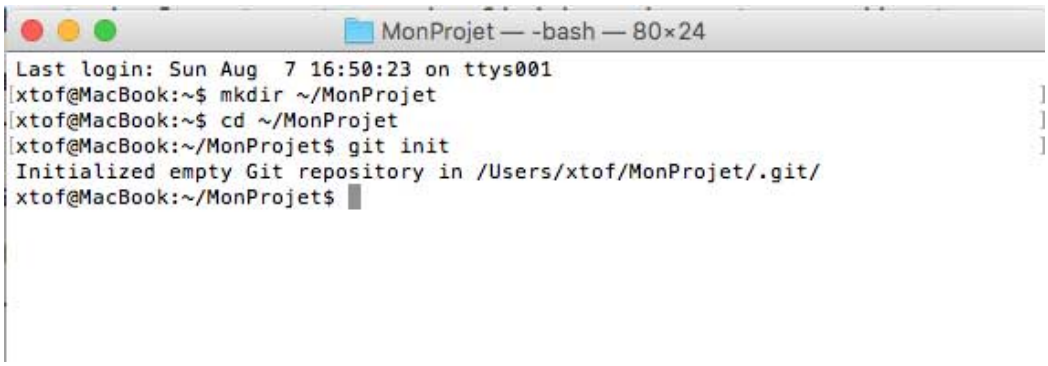
`cd` signifie change directory, et c'est aussi une commande de navigation. Nous venons de produire un répertoire, et nous voulons maintenant passer à ce répertoire et aller dedans. Une fois que

nous avons tapé cettte commande, nous sommes transportés à l'intérieur de MonProjet.

Maintenant, nous allons pour finir utiliser une commande Git. Pour votre prochaine ligne, saisissez :

```
git init
```

Vous savez que vous utilisez une commande Git car elle démarre toujours par `git`. Init signifie « initialiser ». Souvenez-vous que les deux précédentes commandes que nous avons saisies étaient des termes généraux de ligne de commande. Quand nous tapons ce code à l'intérieur, cela dit à l'ordinateur de reconnaître ce répertoire comme un dépôt local Git. Si vous ouvrez le répertoire, il ne s'affichera pas différemment, parce que ce nouveau répertoire Git est un fichier caché à l'intérieur du dépôt dédié.



Cependant, votre ordinateur réalise maintenant que ce répertoire est *prêt-pour-git*, et vous pouvez commencer à entrer des commandes Git. Maintenant, vous avez à la fois un dépôt local et un repo en ligne pour votre projet. Dans la seconde partie, vous apprendrez comment faire votre premier commit vers des dépôts locaux et sur Github. Et vous en saurez plus sur quelques ressources géniales de Github.

Voir aussi

- GitHub pour les Débutants : Consignez, Poussez et Foncez
- *How the Heck Do I Use Github?* (Lifehacker Adam Dachis 2013-12-02)

github git tutoriel

"GitHub Pour les Nuls : Pas de Panique, Lancez-Vous ! (Première Partie)" a été mise à jour le : 15 décembre 2013

AMÉLIOREZ CETTE PAGE

Salvador MEES

Réseau social GitHub == christopheducamp.com/2013/12/15/git… #reseau #travail #collaboratif #programmation

5 years ago

Hannibule

Merci @xtof_fr pour ce tuto en français sur #Git et #GitHub christopheducamp.com/2013/12/15/git… Du fond du cœur, merci !

5 years ago

Juan

Apprentissage personnel du matin - GitHub pour les Débutants : Consignez, Poussez et Foncez ! christopheducamp.com/2013/12/15/git…

5 years ago

Gaby Wald

#ChristopheDucamp #blog "GitHub Pour les Nuls : Pas de Panique, Lancez-Vous !" #Git #PourLesNuls #GitHub ... christopheducamp.com/2013/12/15/git…

7 weeks ago

Copyright © christophe ducamp

GitHub pour les Débutants : Consignez, Poussez et Foncez !

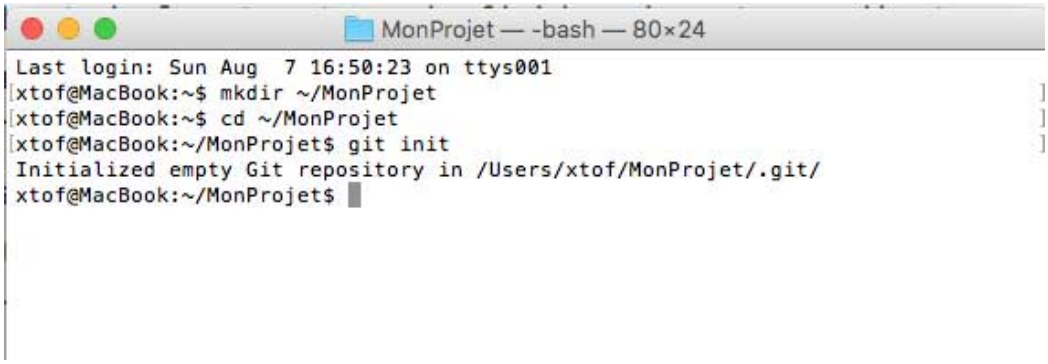
christophe ducamp 16 décembre 2013 - 1900 mots

Traduction -à des fins d'étude et de mémo- d'un article original de Lauren Orsini publié le 2 octobre 2013 pour ReadWriteWeb. Seul le lien original fait référence. - xtof_fr

Maintenant que nous connaissons les concepts Git, il est temps de jouer. Voici venue la deuxième partie de notre série.

Dans la 1ère partie de ce tutoriel GitHub en deux parties, nous avons examiné les principales utilisations de GitHub, commencé le processus d'enregistrement d'un compte GitHub et pour finir, nous avons créé notre propre repository local pour le code.

Maintenant que ces premières étapes ont été accomplies, ajoutons la première partie de votre projet en **produisant votre premier « commit » sur GitHub.** Lorsque nous nous sommes quittés dans la première partie, nous avions créé un repo local appelé `MonProjet`, qui, vu à la ligne de commande, ressemble à cette capture d'écran :



Toujours en fenêtre Terminal, à la prochaine ligne, entrez :

```
$ touch Readme.txt
```

Une fois de plus, *ceci n'est pas* une commande Git. C'est simplement une autre invite de commande standard de navigation. `touch` signifie en fait « créer ». Tout ce que vous écrivez après ce qui suit, c'est le nom de la chose créée. Si vous allez sur votre répertoire local en utilisant le Finder ou le menu Démarrer, vous verrez qu'un fichier vide intitulé `Readme.txt` se niche désormais dedans. Vous auriez pu varier le plaisir avec quelque chose comme "Lisez-moi.doc" ou "Kiwi.gif," .

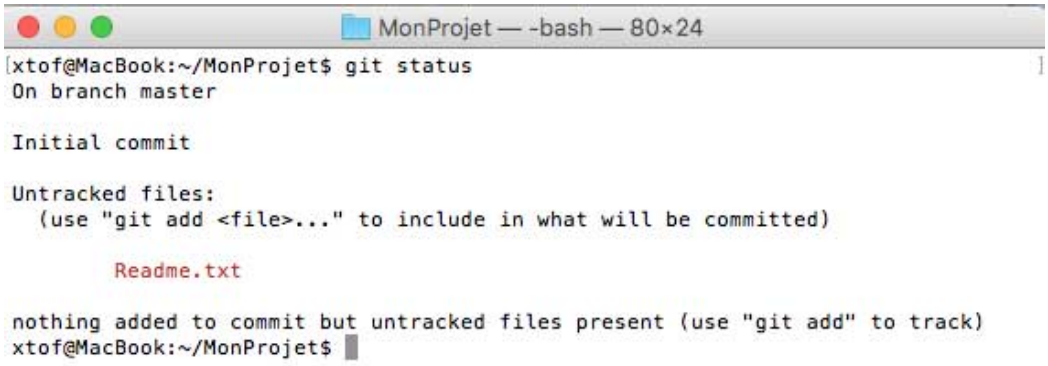
Vous pouvez clairement voir votre nouveau fichier `Readme` . Mais Git le peut-il aussi ? Regardons ça. Tapez :

```
$ git status
```

La ligne de commande, généralement passive jusqu'à ce stade, vous renvoie quelques lignes de texte similaires à ce qui suit :

```
# On branch master
#
# Untracked files:
#
```

```
#   (use "git add ..." to include in what will be
#
#         Readme.txt
```



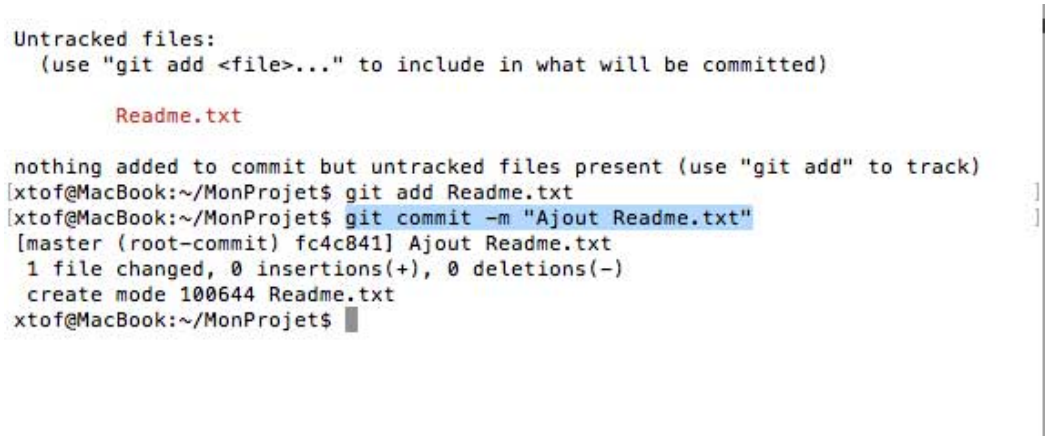
Qu'est-ce qui se passe ?

Tout d'abord, vous êtes sur la branche `master`, la branche principale de votre projet, ce qui est logique puisque nous n'avons pas « bifurqué » du projet. Il n'y a aucune raison faire ça puisque nous travaillons seul. Deuxièmement, `Readme.txt` est répertoriré comme un fichier "untracked", ce qui signifie que Git l'ignore pour l'instant. Pour signaler à Git que le fichier est là, tapez :

```
$ git add Readme.txt
```

Remarquez comment la ligne de commande vous glisse un truc ici ? Très bien, nous avons ajouté notre premier fichier, aussi est-il temps à ce stade de prendre un « instantané » du projet, ou de le « consigner » :

```
$ git commit -m "Ajout Lisez-moi.txt"
```



Le marqueur `m` comme indiqué dans le répertoire des définitions en 1ère partie, indique simplement que le texte qui suit doit être lu comme un message. Notez que le message de `commit` est écrit au présent. Vous devriez **toujours écrire vos commandes au temps présent** parce que le contrôle de version ne traite que de flexibilité dans le temps. Vous n'écrivez pas pour dire ce qu'un commit a fait précédemment, parce que vous pouvez toujours revenir à la version précédente. **Vous écrivez ce que fait un commit**.

Maintenant que nous avons fait un petit travail en local, il est temps de pousser (de "push"er) notre premier « commit » sur GitHub.

"Attendez, on n'a jamais connecté mon dépôt en ligne à mon dépôt local, » pourriez-vous penser. Et vous avez raison. En fait, votre dépôt local et votre dépôt en ligne ne communiquent que par de courtes rafales, lorsque vous confirmez les ajouts et les modifications au projet. Passons maintenant à votre première véritable connexion.

Connecter Votre Dépôt Local À Votre Dépôt GitHub

Avoir à la fois un dépôt local et un dépôt à distance (en ligne), c'est tout bonnement le meilleur des deux mondes. Vous pouvez bricoler tout ce que vous aimez sans même être connecté à internet, tout en présentant votre travail fini sur Github afin que tout le monde puisse le voir.

Cette configuration facilite aussi le fait d'avoir plusieurs collaborateurs travaillant sur le même projet. Chacun de vous peut travailler seul sur son propre ordinateur, mais téléverser ou « push »er vos modifications vers le dépôt Github quand elles sont prêtes. Aussi allons-y.

Tout d'abord, nous devons dire à Git qu'un dépôt distant existe quelque part en ligne. Nous faisons ça en ajoutant ça à la connaissance de Git. Tout comme Git ne reconnaît pas nos fichiers jusqu'à ce que nous utilisions la commande `git add`, il ne reconnaîtra pas non plus notre dépôt distant à cette heure.

Supposons que nous ayons un dépôt GitHub appelé "MonProjet" situé sur `https://github.com/nomutilisateur/MonProjet.git`. Bien sûr, `nomutilisateur` devrait être remplacé par votre véritable nom d'utilisateur Github, et `MonProjet` devrait être remplacé par le véritable titre que vous avez donné à votre premier dépôt GitHub.

```
$ git remote add origin https://github.com/nomut
```

```
xtof@MacBook:~/MonProjet$ git remote add origin https://github.com/ChristopheDucamp/MonProjet.git
xtof@MacBook:~/MonProjet$
```

La première partie est connue ; nous avons déjà utilisé `git add` avec les fichiers. Nous avons ajouté après le mot `origin` pour indiquer un nouvel endroit à partir duquel viendront les fichiers. `remote` est un descripteur de `origin`, pour indiquer que l'original n'est *pas* sur l'ordinateur, mais quelque part en ligne.

Git sait désormais qu'il existe un dépôt distant et que c'est là où vous voulez envoyer vos modifications du dépôt local. Pour confirmer, saisissez cela pour déposer :

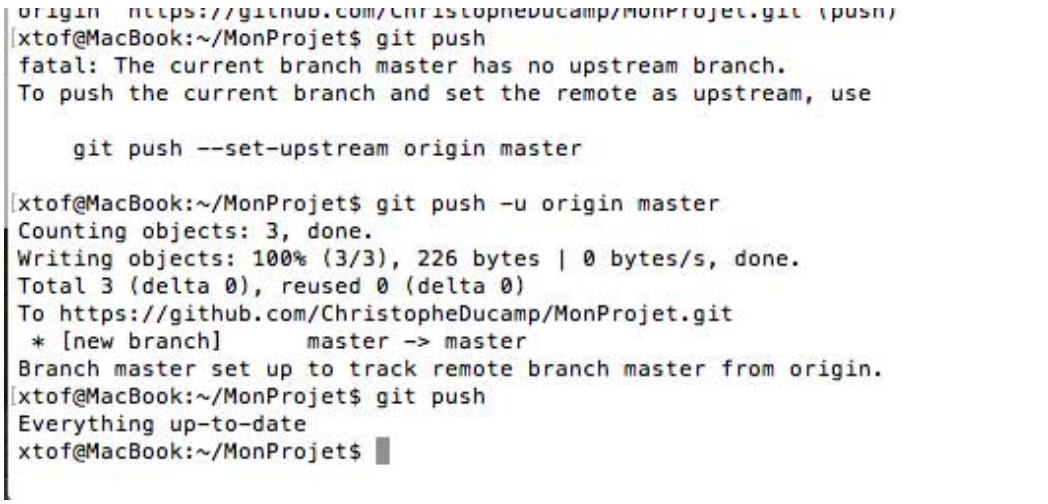
```
$ git remote -v
```

Cette commande vous donne une liste de toutes les origines distantes connues par votre dépôt local. En supposant que vous m'ayez suivi jusque là, il ne devrait y en avoir qu'un, le `MonProjet.git` que nous venons d'ajouter. Il est listé deux fois, ce qui signifie qu'il est disponible pour y *« push »*er de l'information, et pour y extraire (`fetch`) de l'information.

Maintenant, nous voulons téléverser, ou « `push` « er, nos modifications vers le dépôt distant Github. C'est facile. Tapez simplement :

```
$ git push
```

La ligne de commande vous soufflera plusieurs lignes, et le mot final qu'elle recrachera ressemblera à quelque chose comme "everything up-to-date."



Git m'a renvoyé dans mon cas un paquet d'avertissements parce

que j'avais simplement produit la commande simple. Pour être plus spécifique, j'ai saisi `git push --set-upstream origin master`, pour spécifier que je voulais dire la branche `master` de mon dépôt.

Connectez-vous de nouveau à GitHub. Vous remarquerez que GitHub suit désormais combien de commits vous avez produits aujourd'hui. Si vous avez suivi simplement ce tutoriel, ceci devrait être exactement un. Cliquez sur votre dépôt et il aura un fichier identique `Readme.txt` car nous l'avons construit précédemment à l'intérieur de votre dépôt local.

Tous ensemble Maintenant !

Bravo, vous êtes officiellement un utilisateur Git ! Vous pouvez créer des dépôts et *committer* des modifications. C'est là où s'arrête ce tutoriel de débutant.

> *Regardez aussi : Tom Preston-Werner de Github : How We Went Mainstream*

Cependant, vous pouvez avoir cette lancinante impression de ne pas vous sentir comme un expert. Bien sûr, vous avez réussi à suivre quelques étapes, mais êtes-vous prêt à y aller seul ? Je n'aurai nullement cette prétention.

Afin d'être plus à l'aise avec Git, avançons sur un workflow imaginaire tout en utilisant les quelques points que nous avons appris. Vous êtes désormais salarié dans l'agence « 123 Web Design », où vous construisez un nouveau site web pour le Magasin de Glaces de Jimmy avec quelques-uns de vos collègues.

Vous étiez un peu nerveux quand votre patron vous a demandé de participer au projet de redesign de refonte du site du Magasin de Glaces de Jimmy. Après tout, vous n'êtes pas programmeur ; vous

êtes designer graphique. Mais votre patron vous a assuré que tout le monde peut utiliser Git.

Vous avez créé quelques nouvelles illustrations d'un sundae à la crème et il est temps de les ajouter au projet. Vous les avez enregistrées dans un dossier de votre ordinateur, appelé "icecream", pour éviter de vous emmêler.

Ouvrez la Ligne de Commande et changez le répertoire jusqu'à ce vous soyez dans le répertoire `icecream`, là où est stocké votre design.

```
$ cd ~/icecream
```

Puis, réinitialisez Git de manière à pouvoir démarrer en utilisant des commandes Git à l'intérieur du répertoire. Le dossier est désormais un dépôt Git.

```
$ git init
```

Attendez, est-ce le bon fichier ? Voici comment vérifier et vous assurer que c'est bien l'endroit où vous avez stocké votre design :

```
$ git status
```

Et c'est ce que Git vous dira en retour :

```
# Untracked files:
#   (use "git add ..." to include in what will be
#

#       chocolat.jpeg
```

Ayé ils sont ici ! Ajoutez-les dans votre dépôt local Git pour qu'ils soient suivis par Git.

```
$ git add chocolat.jpeg
```

Maintenant, faites un "instantané" du dépôt tel qu'il est maintenant avec la commande commit :

```
$ git commit -m "Ajoute chocolat.jpeg."
```

Bravo ! Mais vos collègues, acharnés au boulot dans leurs propres dépôts locaux, ne peuvent pas voir votre tout nouveau design ! Ceci parce que le projet principal est stocké dans le compte Github de la société (nom d'utilisateur : 123WebDesign) dans le dépôt intitulé "icecream."

Parce que vous ne vous êtes pas encore connecté au dépôt GitHub, votre ordinateur ne sait même pas qu'il existe. Aussi, déclarez votre dépôt local :

```
$ git remote add origin https://github.com/123Web
```

Et double-checkez pour vous assurer qu'il le connaît :

```
$ git remote -v
```

Pour finir, c'est le moment que vous attendiez. Téléversez ce délicieux sundae sur le projet :

```
git push
```

Tut tut ! Avec tous ces outils à portée de mains, il est clair que Git et

le service GitHub ne sont pas que pour les programmeurs.

Les Ressources Git

Git est dense, je sais. J'ai fait de mon mieux pour produire un tutoriel qui pourrait même m'aider à savoir comment l'utiliser, mais nous n'apprenons pas tous de la même manière.

En plus de mon anti-sèche pour la ligne de commande, voici quelques ressources que j'ai trouvées utiles tout en apprenant personnellement à utiliser Git et Github durant l'été :

- **Pro Git**. Voici un livre complet open source sur l'apprentissage et l'utilisation de Git. Il peut paraître long, mais je n'ai pas eu besoin de lire quoi que ce soit après le chapitre trois pour apprendre les fondamentaux.
- **Try Git**. CodeSchool et GitHub ont fait équipe pour produire ce tutoriel rapide. Si vous voulez un peu plus de pratique avec les fondamentaux, ceci devrait vous aider. Et si vous avez un peu d'argent en plus et que vous vouliez apprendre tout ce qu'il faut savoir sur Git, Git Real de Code School devrait faire l'affaire.
- **GitHub Guides**. Si vous êtes plutôt visuel, le canal officiel de GitHub vaut le coup d'oeil. J'ai particulièrement beaucoup **appris de la série Git Basics** en quatre parties.

- **Git Reference**. Vous avez les basiques mais vous oubliez toujours les commandes ? Ce site pratique est génial comme glossaire de référence.
- **Git - le petit guide**. Ce tutoriel est court et délicieux, mais il allait un peu trop vite pour moi. Si vous voulez vous rafraîchir sur les fondamentaux de Git, ceci devrait faire tout ce dont vous avez besoin.

hack github howto versioncontrol

🔗 "GitHub pour les Débutants : Consignez, Poussez et Foncez !" a été mise à jour le : 16 décembre 2013

AMÉLIOREZ CETTE PAGE

E-mail / Twitter / Facebook / Instagram / GitHub / GitLab /
Copyright © christophe ducamp

Instantly share code, notes, and snippets.

YannBouyeron / **git.md**

Last active 28 days ago

Embed <script src="https://gi Download ZIP

Utilisation basique de Git

git.md

Git

Initier un projet.

Vous n'avez encore rien fait dans votre projet.

```
git init mon_projet
```

Cela va créér un répertoire "mon_projet" dans le dossier courant.

Vous avec déjà débuté votre projet.

Votre répertoire "mon_projet" existe donc déjà, et il contient déjà des fichiers. Placez vous dans votre répertoire et exécutez la commande suivante:

```
git init
```

Ajoutez des fichiers à votre git.

Votre répertoire contient des fichiers, il faut les ajouter à git pour qu'il puisse tenir compte de ces fichiers.

```
git add my_file
```

Ou si plusieurs fichiers:

```
git add my_file1 my_file2
```

Ou pour ajouter tous les fichiers du répertoire:

```
git add .
```

Afficher le statut de votre git.

```
git status
```

Vous obtiendrez ainsi la liste des fichiers non encore ajoutés à votre git, et la liste des fichiers déjà ajoutés, mais n'ayant pas encore été mis a jour dans votre git depuis leur dernière modification.

Faire un commit.

Vous avez modifié un fichier (par exemple my_file1), il faut alors faire un commit pour que git enregistre ces modifications:

```
git commit my_file1 -m "ajout de la fonction md2html"
```

L'argument -m permet d'ajouter un bref commentaire décrivant votre modification. Ce commentaire est obligatoire; si vous n'ajoutez pas l'argument -m et son commentaire, l'éditeur nano s'ouvrira alors pour que vous puissiez ajouter votre commentaire.

Obtenir une liste de tous vos commit et de leurs commentaires.

```
git log
```

Pour chaque commit, la première ligne correspond au sha du commit.

Revenir à un ancien commit.

Il faut faire un `git log` pour connaitre son sha.

```
git checkout sha_du _vieux_commit
```

Pour revenir au dernier commit (le plus récent):

```
git checkout master
```

Créér une branche.

Vous voulez faire une modification sur l'un de vos fichiers tout en conservant votre dernier commit intacte; il faut pour cela créér une branche.

```
git branch nom_de_la_branche
```

Pour savoir dans quelle branche vous vous situez:

```
git branch
```

Vous verrez alors vos differentes branches: la branche master (c'est la branche principale), et votre nouvelle branche. L'astérix devant master signifie que vous êtes toujours dans la branche master. Il faut alors changer de branche avant de faire vos modifications:

```
git checkout nom_de_la_branche
```

Pour créér une branche en se plaçant directement dans celle ci:

```
git checkout -b nom_de_la_branche
```

Merger une branche avec le master.

Les modifications apportées dans la branche vous conviennent. Il faut alors fusionner votre branche et votre master. Placer vous dans la branche master:

```
git checkout master
```

Puis :

```
git merge nom_de_la_branche
```

Pour effacer la branche devenue inutile:

```
git branch -D nom_de_la_branche
```

Cloner un repository ou un gist

```
git clone path_du_repository_ou_du_gist path_du_repertoire_a_creer_en_local
```

Push vers github

Relier votre git local à votre repository github (à faire une seule fois):

Créer un repository sur votre compte github, puis placez vous dans votre dossier local contenant votre git et votre projet local. Puis entrez la commande suivante:

```
git remote add origin https://github.com/nomutilisateur/MonProjet.git
```

Faire un push (pensez à faire un commit avant !)

```
git push
```

Ou:

```
git push --set-upstream origin master
```

Il faudra alors entrez votre nom d'utilisateur et mot de passe sauf si vous avez installé un certificat ssh

Push vers gist

- Créer un gist depuis votre compte github.
- Clonez votre gist en local
- Placer votre fichier dans votre git local
- Faire un commit puis un git push

Pull de github vers votre repo local

Pour récupérer les dernières modifications de votre repository distant (sur github) vers votre repository local:

Placez vous dans votre repository local

```
git pull origin master
```

leilachebbah commented on Dec 4, 2019

Thank you for your post

Découverte :
Gérer les versions de vos fichiers : premiers pas avec git

mer 20 Mai 2015 Article CollaboratifDécouverte, Didacticiel 4

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Creative Commons
Attribution 3.0 Unported
License

Git est un logiciel de contrôle de versions de fichiers. Il est distribué sous licence GNU GPLv2 et est disponible sur les principaux systèmes d'exploitation.

Cet article est le premier d'une série de deux. Nous allons voir ici (1) à quoi sert le contrôle de versions, (2) comment configurer git et (3) les bases de son utilisation.

Dans l'article suivant, nous verrons comment cloner un projet (par exemple pour travailler à plusieurs ou pour faire des sauvegardes) et comment synchroniser les différentes copies.

Contexte : le contrôle de versions

Pourquoi faire ?

En bioinformatique comme dans d'autres domaines, nous sommes tous concernés par des fichiers/programmes qui évoluent, que ce soient nos scripts ou nos fichiers textes personnels. Parmi les versions successives, on a typiquement besoin d'**identifier la plus à jour ou celles qui correspondent à des versions majeures** afin de les distinguer par rapport à toutes celles qui correspondent des changements mineurs.

De plus, l'**enchaînement des versions n'est pas toujours strictement linéaire** et il peut se produire des branchements où une version d'un document sert de point de départ à deux copies qui évoluent indépendamment l'une de l'autre.

Enfin, il est parfois nécessaire d'aller au delà des versions successives d'un seul document et d'**administrer les versions successives de plusieurs documents** qui correspondent ensemble à un projet.

Les problèmes précédents se compliquent lorsque l'on a besoin de **maintenir à jour une copie de ces fichiers à différents endroits ou avec différentes personnes**, et c'est encore pire si plusieurs de ces personnes peuvent faire des modifications et les partager à leur tour. Il faut alors gérer la synchronisation et les accès concurrents.

Logiciels de contrôle de version

Les logiciels de contrôle de versions servent précisément à gérer les versions successives d'un ensemble de documents, ainsi que leur partage, leur mise à jour entre plusieurs utilisateurs en gérant les conflits potentiels lorsque plusieurs personnes font des modifications concurrentes sur le même document. Les plus connus sont CVS, subversion (svn), bazaar (bzr), mercurial, ou git que nous allons présenter ici. On a souvent une vision un peu rébarbative des logiciels de contrôle de versions alors qu'ils sont en général simples à prendre en main et se révèlent très utiles. Cet article a justement pour but de vous aider à faire le premier pas.
Il existe par ailleurs une quantité impressionnante de didacticiels sur git. Souvent en anglais, mais promis, nous allons faire un effort ici.
CVS et SVN utilisent un modèle en étoile avec un répertoire maître sur lesquel se font toutes les mises à jour et sur lequel se synchronise chaque utilisateur. Inversement, mercurial, bazaar et git reposent sur une approche décentralisée : après duplication, la copie a le même statut que l'original (on utilise souvent la métaphore du clonage).
Initialement codé par Linus Torvalds pour gérer les versions de son célèbre noyau, git est aujourd'hui massivement utilisé dans le monde de l'open-source, notamment via des plateformes centralisées (bizarre, hein ?) telles que github ou bitbucket. CVS et SVN étaient la référence il y a quelques années, et même s'ils restent largement utilisés, git est aujourd'hui l'outil de gestion de versions le plus classique (on n'a pas dit le meilleur).
Bien sûr, comme nous ne sommes point intégristes, un petit encart montre l'équivalent sans git au début de chaque partie.
Pour un usage efficace d'un tutoriel, il est généralement conseillé de faire ce que personne ne fait jamais : reproduire les commandes et autres manipulations chez vous. (présence d'un adulte non nécessaire)
Et, n'oubliez jamais : un gestionnaire de versions se souvient de tout. NO EXCEPTION

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Rien à faire pour cette étape. Cool, non ?

Avec git

Vous savez ce qu'on dit : un bon outil est difficile à prendre en main, mais une fois cela fait rien ne le remplace. C'est pareil pour git : 3 commandes suffisent à le configurer. Il faut d'abord ouvrir un terminal, et installer git :

```
# Vous pouvez utiliser les paquets de votre distribution, ou à défaut télécharger git depuis http://git-scm.com.
# Ici on se simplifie la vie et on prend la première option.
# Adaptez selon votre distribution.
sudo apt-get install git # debian, ubuntu et consorts
sudo pacman -S git # archlinux
# pour les autres, soit vous savez, soit vous êtes sur mac ou windows…
```

Ensuite, avant que git ne vous le demande officiellement, vous devriez lancer ces deux commandes, très gitesques :

```
git config --global user.name "Gérard Menvuça"
git config --global user.email "super.gege@wanadold.fr"
```

Pourquoi ?

Parce que c'est ainsi que git vous identifiera. Après avoir entré ces commandes, vous pourrez voir dans votre fichier `~/.gitconfig` que les valeurs sont enregistrées ici… C'est en fait ici que git cherchera si, dans le projet dans lequel vous travaillez, aucune de ces informations n'est renseignée. Pour les trois de devant qui suivent : oui, cela permet d'avoir plusieurs identités.

Le cas Windows/OSX

Git est un programme suffisamment démocratisé pour se retrouver partout. Sous tout OS Unix-like (donc OSX aussi), git est utilisable dans la console, fût-elle bien cachée, et installable d'une manière ou d'une autre.

Pour Windows, on nous souffle en coulisse que Powershell gère l'affaire avec la même interface que les autres. Youpi !

Git pour remplir 2 Gb de RAM

Pour les plus allergiques à la ligne de commande, ou pour ceux qui n'intègrent pas cet outil à leur workflow, il existe une quantité importante d'interfaces graphiques qui encapsulent les commandes détaillées dans ce tuto.

Elles nécessitent toujours une compréhension de ce qu'est git et de ce qu'il fait (du moins cela ne peut que aider), ne vous attendez donc pas à vous économiser la totalité de ce tutoriel en vous contentant d'un clicodrôme (d'autant que certaines fonctionnalités avancées dont vous pourriez un jour avoir besoin pourraient nécessiter l'intervention en ligne de commande).

Créer un projet vide

Sans git

Un `mkdir` et pouf, c'est fini !

Avec git

Un "`git init`" et pouf, c'est fini aussi. On en profitera pour voir également "`git status`".

git init

Cette commande crée un dépôt (aussi appelé *repository*), c'est à dire un répertoire que git surveillera et gèrera, pendant que vous y ajouterez ou modifierez vos fichiers. Dans le jargon git, cela se traduit par la création d'un dossier caché nommé "`.git`" à l'endroit précis où vous vous trouvez quand vous lancez "`git init`"

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
$ mkdir /some/path/to/your/project # création du répertoire
$ cd /some/path/to/your/project # on va dedans
$ ls -a # le répertoire est bien vide
. ..
$ git init # bon, ben, créons un dépôt git vide initialisé dans /some/path/to/your/project/.git/
$ ls -a # la commande "git init" crée un répertoire caché ".git"
. .. .git
$ ls .git # on jette un oeil dans le répertoire
branches config description HEAD hooks info objects refs
```

Comme vous pouvez le constater, il y a déjà plein de trucs dans le `.git` alors qu'on n'a encore rien écrit... Rien d'anormal ; c'est comme quand vous venez d'installer un système : il y a déjà plein de bazar dans des répertoires inutiles tels que lib, bin ou dev, users, programs,... mais si vous tripotez trop à ce qui s'y trouve sans savoir ce que vous faites, eh bien... vous aurez quelques problèmes avec votre install. Git c'est pareil. Laissez-le gérer ses fichiers, il est meilleur que vous à ce niveau (pour les vôtres aussi d'ailleurs). Néanmoins, vous serez peut-être amenés à modifier un peu les fichiers "`.git/config`" et "`.git/description`" selon ce que vous voudrez, plus tard, quand cet article n'aura plus de secrets pour vous.

git status

Cette commande ne modifie rien, elle ne fait que parler. Et pourtant, vous l'adorerez : elle renseigne sur l'état actuel du dépôt. Voyez plutôt :

```
$ git status
 On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

Ici, git nous indique clairement qu'on devrait se mettre au boulot. Il dit même comment (vous verrez, git est très malin à ce petit jeu), ce qui spoile un peu la prochaine partie de cet article.
Plus tard, cette commande nous resservira. C'est l'occasion de revenir sur le répertoire caché "`.git`" créé lors du "`git init`" : lorsque vous faite une commande git, par exemple "`git status`", git va chercher le premier répertoire "`.git`" qu'il trouvera en partant du répertoire courant puis en remontant successivement. De fait, lorsque vous serez la tête dans votre projet avec votre `pwd` quelque part dans la hiérarchie, git trouvera seul le dépôt, et trouvera les infos nécessaires dans le "`.git`". Par contre, lorsque vous n'êtes pas dans un dépôt et faites "`git status`", git va vous sermoner. La preuve : git est remonté jusqu'à la racine système sans trouver le moindre "`.git`".

Principes de base (pour travailler seul(e))

Sans git

Codez.
Si vous supprimez un fichier par erreur : perdez votre travail et priez pour `foremost` sans passer par la case départ.
Si vous voulez faire des sauvegardes de version, faites des dumps de dossiers avec des noms que vous ne comprendrez plus dans 2h, et que de toute façon vous ne réutiliserez jamais, sauf la semaine prochaine après le problème de sauvegarde ; mais de toute façon vu que vous aviez tout refactorisé, autant tout reprendre de zéro : effacez les dossiers de sauvegarde faits à la main, allez demander de l'aide sur stackoverflow. Expliquez à votre boss que vous n'auriez pas une semaine de retard si `mv` et `rm` n'était pas aussi proches syntaxiquement, apprenez l'existance d'un truc qui s'appelle gestionnaire de versions, prenez peur en lisant la page wikipédia, découvrez cet article. Expirez profondément, vous êtes sur la bonne voie et tout va déjà mieux.

Avec git

git status

Après la création du dépôt, "`git status`" vous disait que tout allait bien, et que la prochaine chose à faire est de créer des fichiers, puis d'utiliser "`git add`". Nous allons donc maintenant créer un fichier "`firstFile.txt`" et voir ce qui se passe.

```
$ vi firstFile.txt
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

gestion des conflits, qui sera vu plus tard)
Maintenant que nous avons créé le fichier, voici la réaction de git :

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

git add

Dans l'exemple précédent, on voit que créer le fichier `firstFile.txt` dans le dépôt ne suffit pas pour que git le prenne en charge automatiquement, et qu'il faut pour cela faire un "`git add`".

```
$ git add firstFile.txt
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: firstFile.txt
$
```

Cela peut sembler inutilement compliqué et malcommode, mais c'est nécessaire afin d'indiquer quels sont les fichiers intéressants. On verra dans la section sur `.gitignore` un peu plus loin comment traiter les fichiers pas intéressants.
Après avoir explicitement inclus `firstFile.txt` dans la liste des fichiers que git doit gérer, le dernier "`git status`" indique que le fichier est bien pris en compte. À ce stade, git sait qu'il doit surveiller ce fichier et il en a pris une empreinte de la version courante (un *snapshot* dans la documentation en anglais), qu'il stocke dans une zone temporaire appelée l'*index*. Si maintenant on modifie le fichier `firstFile.txt`, git sait qu'il doit le surveiller et indique que le snapshot qu'il a pris lors du "`git add`" n'est plus à jour. Il faut donc faire un nouveau "`git add`" pour réactualiser le snapshot. L'exemple suivant illustre ce principe : le premier "`git status`" reprend l'étape précédente et indique que tout va bien ; on modifie ensuite le fichier et le "`git status`" qui suit indique que git dispose bien d'un snapshot dans sa zone temporaire (c'est la ligne "`new file: firstFile.txt`" qui est similaire à ce qu'on avait avant), mais que ce snapshot n'est plus à jour car le fichier a été modifié depuis (ce sont les lignes à partir de "`Changes not staged for commit:`"). On fait de nouveau un "`git add firstFile.txt`" pour mettre à jour le snapshot et le troisième git status montre que tout va bien à nouveau et que les changements sont prêts pour un `commit` (cf. section suivante).

```
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line

$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: firstFile.txt
$ vi firstFile.txt
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: firstFile.txt
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: firstFile.txt
$ git add firstFile.txt
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: firstFile.txt
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
$ ls
firstFile.txt
$ vi README.txt
$ vi licence.txt
$ mkdir doc
$ vi doc/documentation.txt
$ vi doc/otherDocumentation.txt
$ find . -name .git -a -type d -prune -o -print
.
./README.txt
./doc
./doc/documentation.txt
./doc/otherDocumentation.txt
./firstFile.txt
./licence.txt
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: firstFile.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.txt
        doc/
        licence.txt
$ git add README.txt licence.txt doc
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: README.txt
        new file: doc/documentation.txt
        new file: doc/otherDocumentation.txt
        new file: firstFile.txt
        new file: licence.txt
```

À propos de la discrimination des dossiers vides, car, diantre ! Cela peut être ennuyeux pour les dossiers de logs, et il ne s'agit pas d'un bug de fonctionnement :
dans la vraie vie réelle de l'internet, il existe plusieurs moyens de garder un dossier vide (comme ici ou là), et c'est a vous de choisir votre solution préférée (3615 ma life de lucas: j'ai une préférence pour le bricolage de .gitignore, puisqu'il permet de spécifier des patterns de fichier à ignorer, typiquement les fichiers avec une extension log, swp,...). L'usage du gitignore sera détaillé plus loin.

git commit (+ git status)

À ce stade, git surveille bien les fichiers et en prend des snapshots, mais l'index où ils sont stockés n'est qu'une zone temporaire. Lorsque les fichiers sont à jour, le "`git status`" indique qu'ils sont maintenant prêts à être archivés de façon définitive avec un "`git commit`". Typiquement, on travaille sur certains fichiers en faisant des modifications (éventuellement plusieurs, comme nous l'avons fait aux figures précédentes) jusqu'à réussir à ajouter une nouvelle fonctionnalité ou jusqu'à en avoir débuggué une. Le `git commit` permet alors de dire "Voilà, cette version des fichiers correspond à telle action". Cette version devient la nouvelle version à jour qui peut être partagée.
Remarquez que le seul changement de version "*officielle*" se fait d'un commit à l'autre, sans que personne ne voit la longue série de modifications plus ou moins habiles qui figurait dans les "`git add`" et qui vous a permis d'y parvenir.

```
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: README.txt
        new file: doc/documentation.txt
        new file: doc/otherDocumentation.txt
        new file: firstFile.txt
        new file: licence.txt
$ git commit -m "First commit"
[master (root-commit) e26b158] First commit
 5 files changed, 10 insertions(+)
 create mode 100644 README.txt
 create mode 100644 doc/documentation.txt
 create mode 100644 doc/otherDocumentation.txt
 create mode 100644 firstFile.txt
 create mode 100644 licence.txt
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

pas facile sur le premier `commit`). Si vous pensez qu'il faut donner des détails, commencez quand même par une phrase brève, laissez une ligne blanche, puis racontez ce que vous voulez.
Si vous ne donnez pas l'option m et le message de commit, git va appeler votre éditeur de texte par défaut (souvent nano si vous n'avez rien changé, définit par la variable d'environnement $EDITOR), ou un autre si vous avez utilisé la commande `"git config --global core.editor "vim""` ou équivalent. Une fois démarré, l'éditeur de texte vous laissera écrire le message de commit.
Un parallèle avec l'industrie du bâtiment est possible : imaginez un commit comme une brique utilisée pour construire un mur. C'est l'ensemble des briques posées les unes sur les autres qui rendent le projet final. Plus tard, nous verrons qu'il est possible de cloner un mur en construction pour apposer ses propres briques, et de proposer ensuite une fusion des murs : le début du travail collaboratif.
Pour récapituler, il faut donc faire une combinaison d'une suite de "`git add`" pour ajouter les fichiers pertinents à l'index, et "`git commit`" pour valider les changements des fichiers de l'index.
Lorsque vous commencez à savoir ce que vous faites, un "`git commit -a -m`" combine les deux commandes précédentes en ajoutant les fichiers qui ont changé (mais pas les nouveaux) à l'index et en les validant. Il est donc aisé de commiter plusieurs fichiers simultanément. Néanmoins, cela va à l'encontre de la philosophie de git : **un commit devrait toujours être une modification atomique**, pas une pleine brouette de code désorganisée et obèse. Une exception notable est cependant lorsque vous créez un dépôt pour un projet, comme Linus avant tous, le premier `commit` (nommé souvent "`initial commit`") est généralement un gros dump de ce qui a été fait auparavant. Par exemple, cela permet simplement de suivre avec un échantillonnage élevé l'évolution d'un projet, et donc de contrôler facilement les ajouts de fonctionnalités, les versions,...
Exactement ce pourquoi git existe. Et en plus c'est mieux pour faire de jolis graphiques !

Graphe d'activité du dépôt git du projet Julia

Donc : **préférez des `commit`s atomiques et simples, touchant le minimum de fichiers à chaque fois, avec un message de `commit` qui répond au *pourquoi*, et non au *comment*** dont la réponse est déjà dans le `commit`. Le dépôt du langage Julia présenté dans un autre article est un bon exemple : des milliers de `commit`s, et même les tous premiers sont atomiques :

```
git clone https://github.com/JuliaLang/julia
cd julia
git log --reverse # afficher les commits du plus ancien au plus récent
```

git log pour suivre l'historique des commit

La commande "`git log`" vous permet de lister l'auteur, la date et la description succinte (celle qui figurait après le -m) des "`git commit`" successifs.

La commande git log mériterait un tuto à elle toute seule. Dans un premier temps, la connaitre et l'utiliser sans arguments suffit largement.
Disons que les trois lignes précédentes seront votre premier TP, si par malheur vous ne reproduisez pas le tuto à la lecture.

git diff

Au cours du cycle typique "modification de fichier(s) <-> `git add` <-> `git commit`", git permet plusieurs points de contrôle (le "`git add`" met à jour l'index temporaire et le "`git commit`" met à jour le dépôt à partir de l'index) et on peut avoir besoin d'analyser ce qui a changé de l'un à l'autre grâce à la commande "`git diff`".

- Par défaut, "**`git diff`**" permet de comparer la version courante du fichier avec la dernière version prise en compte par l'index (donc ce qui a changé dans la version actuelle du fichier depuis le dernier "`git add`")
- Par contre, "**`git diff --cached`**" permet de comparer la dernière version prise en compte par l'index avec l'état lors du dernier commit (donc ce qui va être modifié lors du prochain `commit`)

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

et ne met en valeur que la première modification et ignore la seconde. Si ça vous semble normal, c'est que vous avez bien assimilé le principe de git ; sinon relisez les sections sur "`git add`" et "`git commit`".

```
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

$ git status
On branch master
nothing to commit, working directory clean
$ echo "Adding a fourth line" >> firstFile.txt
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified: firstFile.txt
no changes added to commit (use "git add" and/or "git commit -a")
$ git diff
diff --git a/firstFile.txt b/firstFile.txt
index 9a4e002..d5078b7 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -3,3 +3,4 @@ This is the first line of my first file...
 Now we add another line

+Adding a fourth line
$ git add firstFile.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified: firstFile.txt
$ git diff
$ git diff --cached
diff --git a/firstFile.txt b/firstFile.txt
index 9a4e002..d5078b7 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -3,3 +3,4 @@ This is the first line of my first file...
 Now we add another line


+Adding a fourth line
$ echo "Adding a fifth line" >> firstFile.txt
$ git diff
diff --git a/firstFile.txt b/firstFile.txt
index d5078b7..ba75664 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -4,3 +4,4 @@ Now we add another line


 Adding a fourth line
+Adding a fifth line
$ git diff --cached
diff --git a/firstFile.txt b/firstFile.txt
index 9a4e002..d5078b7 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -3,3 +3,4 @@ This is the first line of my first file...
 Now we add another line


+Adding a fourth line
$ git commit -m "this commit only adds the fourth line; the fifth is ignored as it is not yet in the index"
[master 61b5f1d] this commit only adds the fourth line; the fifth is ignored as it is not yet in the index
 1 file changed, 1 insertion(+)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified: firstFile.txt
no changes added to commit (use "git add" and/or "git commit -a")
$ git diff
diff --git a/firstFile.txt b/firstFile.txt
index d5078b7..ba75664 100644
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
$ git diff --cached
$
```

Le dernier "`git status`" indique que `firstFile.txt` a été modifié depuis le dernier snapshot (ben oui, on vient d'y ajouter successivement deux lignes pour illustrer "`git diff`" et "`git diff --cached`"). On en profite pour mettre de l'ordre tout en récapitulant avec un "`git add`" pour mettre le snapshot à jour, puis un "`git commit`".

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified: firstFile.txt
no changes added to commit (use "git add" and/or "git commit -a")
$ git add firstFile.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified: firstFile.txt
$ git commit -m "added two more lines to explain how git diff works"
[master fcae267] added two more lines to explain how git diff works
 1 file changed, 1 insertion(+)
$ git status
On branch master
nothing to commit, working directory clean
$
```

gitignore

Dans un projet, il y a des fichiers ou des répertoires qu'il n'est pas utile d'archiver ou de partager comme par exemple des fichiers intermédiaires de compilation, des fichiers de log, les fichiers `.bak` générés par certains éditeurs... Tant qu'on ne les ajoute pas grâce à un "`git add`" ce n'est pas bien grave, mais ils ont quand même l'inconvénient de polluer les "`git status`". De plus, on a vu précédemment que lors d'un "`git add`" sur un répertoire, git ajoute à l'index tous les fichiers du répertoire, y compris les fichiers indésirables.
En plus des fichiers inutiles, il est également bon d'éviter d'ajouter des fichiers binaires ou des fichiers compressés à un projet si ceux-ci peuvent être ammenés à changer. En effet, git est bien adapté aux fichiers textes mais pas trop aux autres types de fichiers (c'est logique, souvenez-vous du `git diff` de la section précédente). Normalement, à chaque modification, git ne stocke que ce qui a changé depuis la dernière version. Dans le cas des fichiers binaires, il est donc obligé de stocker la version complète du fichier à chaque modification, ce qui conduit rapidement à une augmentation importante de la taille du dépôt.
Il est possible d'**indiquer à git qu'il doit ignorer certains fichiers** (en donnant leur nom), ou certains types de fichiers (à l'aide d'expressions régulières). Pour cela, créez un fichier nommé "`.gitignore`" à la racine du projet (dans le répertoire où il y a aussi le répertoire "`.git`" généré lors du "`git init`"). **Dans le fichier "`.gitignore`", mettez un nom de fichier ou une expression régulière par ligne**.

```
$ git status
On branch master<br>nothing to commit, working directory clean
$ touch firstFile.txt.bak
$ mkdir log
$ touch log/example.log
$ touch log/example.log.gz
$ git status
On branch master<br>Untracked files:
  (use "git add <file>..." to include in what will be committed)
        firstFile.txt.bak
        log/
nothing added to commit but untracked files present (use "git add" to track)
$ vi .gitignore
$ cat .gitignore
*.bak
log/
$ On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
nothing added to commit but untracked files present (use "git add" to track)
$ ls -a
. .. doc firstFile.txt firstFile.txt.bak .git .gitignore licence.txt log README.txt
```

Le dernier "`git status`" indique deux choses :

- les fichiers se terminant en "`.bak`" et le répertoire "`log`" existent bien mais sont ignorés par git

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file: .gitignore
$ git commit -m "added a .gitignore file to ignore backup files and the log directory"
[master 7deabd7] added a .gitignore file to ignore backup files and the log directory
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
$ git status
On branch master
nothing to commit, working directory clean<
$
```

Il faut bien comprendre que le contenu du fichier "`.gitignore`" est lu par git et qu'il indique **les fichiers dont git ne tient pas compte lors de la mise à jour de l'index**. Si vous avez l'esprit taquin, vous pouvez évidemment mentionner "`.gitignore`" dans votre "`.gitignore`". Git continuera bien à en lire le contenu, mais n'ajoutera simplement pas votre "`.gitignore`" à l'index.
Ce n'est sans doute pas une bonne idée si plusieurs personnes collaborent au projet. En effet, vos partenaires ne récupèreront donc pas votre "`.gitignore`" avec les autres fichiers du projet. Ils risquent alors d'ajouter des fichiers indésirables lorsqu'ils feront un "`git commit`" (mais grâce à la nature décentralisée de git, vous aurez quand même le loisir de rejeter leur `commit`, il vous faudra juste refaire le ménage à la main).
On a évidemment tendance à conserver les mêmes `.gitignore` d'un projet à l'autre, donc c'est un bon investissement. Il existe également des générateurs de `.gitignore` qui vous aident dans cette tâche.
Une application très parlante du gitignore dans tout projet Python est d'ignorer l'ensemble des fichiers issus de la compilation :

```
*.pyc
__pycache__/
```

On peut également utiliser le gitignore pour conserver un dossier vide, comme expliqué dans la partie dédiée à git add, tout en évitant de tracker l'ensemble des fichiers de logs.

Note importante : avoir un gitignore n'est pas obligatoire. Un gitignore n'affecte que le dossier où il se trouve ainsi que ses sous-dossiers. Vous pouvez avoir plusieurs gitignore.

Conclusion

Les outils de gestion de versions comme git sont bien plus simples à utiliser qu'on ne le pense (du moins pour une utilisation de base).

Nous avons vu comment créer un dépôt, sélectionner les fichiers et les répertoires que l'on veut prendre en compte et enregistrer leurs versions successives. Dans le prochain article, nous verrons comment utiliser git dans un contexte collaboratif, détecter et résoudre les conflits que cela peut entrainer, créer des branches et indiquer les étapes qui correspondent à des versions.

L'excellent site first aid git recense les questions les plus fréquentes sur git (et leurs réponses)

Remerciements

Merci à Hautbit, NiGoPol et Slybzh pour les commentaires et discussions lors de l'édition de cet article.

À propos de cet article

Cet article et le suivant ont étés adaptés à partir d'un cours donné par Lucas Bourneuf.

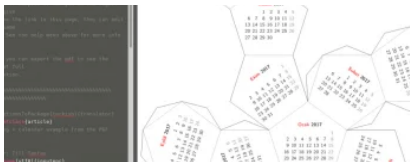
Il a été rédigé par Lucas Bourneuf et Olivier Dameron.

Partager :

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Git est un logiciel de contrôle de versions de fichiers. Il est distribué sous licence GNU GPLv2 et est disponible sur les principaux systèmes d'exploitation. Dans l'article
jeu 11 Juin 2015
Dans "Didacticiel"



Créez vos documents collaboratifs en LaTeX
jeu 29 Juin 2017
Dans "Découverte"

S'outiller et s'organiser pour mieux travailler
mer 5 Oct 2016
Dans "Astuce"

- **À propos de Article Collaboratif**
-

Catégorie: Découverte, Didacticiel | Tags: contrôle de version, git, programmation
S'abonner au flux de commentaires de cet article

4 commentaires sur “Gérer les versions de vos fichiers : premiers pas avec git”

1.

Pierre Marijon

mai 20, 2015 à 10:00

On aurait pu aussi parler de forge logiciel (site proposant les mêmes fonctionnalités que github) libre, je citerais mais ce ne sont pas les seuls :

* gitlab http://gitlab.org/ une instance tenue par framasoft : https://git.framasoft.org/
* gogs http://gogs.io/ la démo : https://try.gogs.io/

Il y en a bien d'autres et beaucoup que j'ai vu passé mais ne retrouve plus le nom, l'offre de forge logiciel est assez impressionnante.

Pour ceux qui voudraient retrouver l'aspect décentralisé de git je vous conseille d'aller voir du coté de gitchan http://gitchain.org/

Répondre

2.

Serious Lee

mai 20, 2015 à 11:28

Très bon tuto. Un petit firstFile.py à un moment sème le doute 😉

Répondre

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

C'est corrigé !

Répondre

2.

hedjour

mai 22, 2015 à 5:15

Pour mémoire et ceux qui n'en ont pas 😉

Répondre

Laisser un commentaire

Entrez votre commentaire...

← Précédent Suivant →

Recherche

La boutique officielle de Bioinfo-fr.net !

Portez haut les couleurs de votre blog de bioinfo préféré !

Les profits nous aideront à maintenir le blog et à vous faire de beaux cadeaux :)

Les snippets de Bioinfo-fr

Partagez vos bouts de code !

canSnippet

Articles au hasard

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

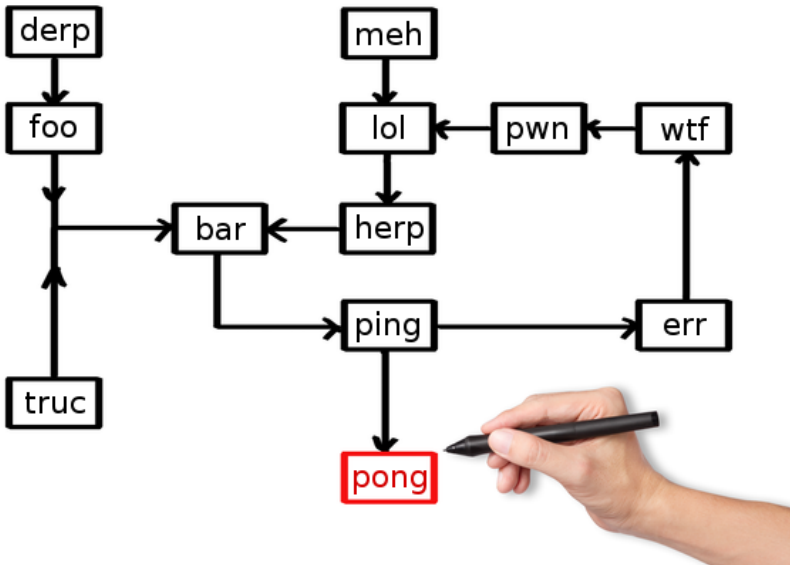
- [RNA-seq : plus de profondeur ou plus d'échantillons ?]
  mer 8 Avr 2015

```
usage: monScript.py [-h] [-t] [--path PATH] input output

positional arguments:
  input         Input File
  output        Output File

optional arguments:
  -h, --help    show this help message and exit
  -t            Print time
  --path PATH   Path to software
```

- [Ajoutez une interface graphique à votre script en 4 lignes avec Gooey]
  mer 6 Juil 2016
- [SIB (Swiss Institute of Bioinformatics)]
  mer 5 Fév 2014
- [Gephi pour la visualisation et l'analyse de graphes]
  mer 5 Mar 2014

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok



- Formaliser ses protocoles avec Snakemake
  mer 26 Fév 2014

Liens

- BioStar
- canSnippet
- JeBiF
- Les Bioinformations
- SFBI

Catégories

- Actualité (17)
- Astuce (36)
- Bioinformatique (1)
- Brèves (9)
- Conférence (13)
- Contribuer (1)
- Découverte (116)
- Didacticiel (59)
- Editorial (58)
- En image (11)
- Entretien (13)
- Formation (14)
- J'ai lu (9)
- Journal Club (5)
- Opinion (31)
- Suivez l'guide (27)

Commentaires récents

- L'art de dessiner des graphes – archivEngines dans Tour d'horizon des outils de visualisation des réseaux biologiques
- Léopold Carron dans Hi-C: Quelques bases
- Bioss dans Hi-C: Quelques bases
- blob dans LaTeX : automatisez le traitement des CSV
- lhtd dans Covid-19 : Des liens utiles pour aider à la recherche contre le SARS-CoV-2

Étiquettes

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

SFBI SQL statistiques strip séquençage thèse Tips tutoriel vacances visualisation

Sauf mention contraire, tous les contenus sont publiés sous licence CC-by-SA 2.0 et supérieure.

Pour les scripts et binaires, référez-vous à la licence associée attribuée par l'auteur.

Pour tout renseignement supplémentaire : admin AT bioinfo-fr.net.

Design: imago-fr.org.
Aller en haut

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Didacticiel :
Git : cloner un projet, travailler à plusieurs et créer des branches

jeu 11 Juin 2015 Article CollaboratifDidacticiel 5

Logo de git (http://git-scm.com) Git Logo by Jason Long is licensed under the Creative Commons Attribution 3.0 Unported License

Git est un logiciel de contrôle de versions de fichiers. Il est distribué sous licence GNU GPLv2 et est disponible sur les principaux systèmes d'exploitation.

Dans l'article précédent, nous avions vu comment installer et configurer git, comment créer un dépôt pour un projet, ainsi que les

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Cloner un projet (par ex. pour travailler à plusieurs ou pour sauvegarder)... et synchroniser les différentes copies

git clone pour dupliquer un dépôt

La commande "`git clone </path/to/existing/git/repository> </path/where/it/should/be/cloned>`" permet de cloner un dépôt existant (en local ou à travers un réseau si le chemin est une URL) à un autre endroit. La métaphore du clonage doit être comprise ici comme une copie exacte : après "`git clone`" les deux dépôts sont les images parfaites l'une de l'autre : ils possèdent le même historique, et il n'y en a pas un qui est plus légitime que l'autre (en fait, on verra dans la section sur "`git pull`" que suite au clonage la branche master du clone fait référence à la branche master du cloné). Si plusieurs personnes travaillent sur un projet, elles peuvent ainsi en obtenir chacune une copie.
Dans un premier temps, nous allons synchroniser les dépôts, puis nous verrons dans un second temps comment détecter et gérer les conflits qui peuvent survenir lors de modifications concurrentes. Nous supposerons que le projet initial était dans le répertoire `repoA`, et qu'on le clone dans un autre répertoire `repoB` situé n'importe où sauf sous l'arborescence de `repoA` évidemment. Pour simplifier nous prendrons `repoA` et `repoB` au même niveau, même si en pratique ça n'a pas grand intérêt.

```
$ git status
On branch master
nothing to commit, working directory clean
$ cd ..
$ mkdir repoB
$ git clone repoA repoB
Cloning into 'repoB'...
done.
$ ls -a repoA
. .. doc firstFile.txt firstFile.txt.bak .git .gitignore licence.txt log README.txt
$ ls -a repoB
. .. doc firstFile.txt .git .gitignore licence.txt README.txt
```

Au passage, vous remarquerez que le fichier `.gitignore` (c'est le même fichier qu'au premier article) dans `repoA` a bien été pris en compte et que le fichier `firstFile.txt.bak` ainsi que le répertoire `log` ont bien été ignorés lors du clonage dans `repoB`.

git pull pour maintenir à jour une copie locale d'un dépôt de référence

Nous allons commencer par une configuration simple où `repoA` est le dépôt de référence d'un projet où se font toutes les modifications, et vous souhaitez uniquement maintenir à jour votre copie locale `repoB` sans que vous y apportiez la moindre modification.

Maintenant que le dépôt d'origine `repoA` a été cloné, nous allons commencer par y faire des modifications puis un nouveau `commit`.

```
$ cd repoA
$ git status
On branch master
nothing to commit, working directory clean
$ echo "modification from repoA" >> firstFile.txt
$ git add firstFile.txt
$ git commit -m "some improvement from repoA"
$ git status
On branch master
nothing to commit, working directory clean
```

Maintenant, un "`git status`" dans `repoB` montre que celui-ci est à jour (par rapport à lui-même), mais que les modifications que nous venons de faire dans `repoA` n'ont pas été reportées dans `repoB`.

```
$ cd ../repoB
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$ diff ../repoA/firstFile.txt ./firstFile.txt
8d7
< modification from repoA
```

Pour synchroniser son dépôt avec celui que nous avions cloné (ou un autre), il faut faire un appel explicite à "`git pull`". Ce dernier ne se fait pas automatiquement (heureusement, imaginez un peu la pagaille). Ci-dessous, la ligne 19 lorsque l'on regarde le contenu du fichier `firstFile.txt` après le "`git pull`" montre que la modification que nous avions faite dans `repoA` a bien été répercutée dans `repoB`.

Remarquez que dans `repoB`, nous faisons simplement "`git pull`" sans lui dire explicitement qu'il faut aller chercher dans `repoA`. En fait, le "`git clone repoA repoB`", git a bien indiqué dans `repoB` que ce dernier avait été généré à partir de `repoA` en le désignant comme "origin" (comparez donc les fichiers `repoA/.git/config` et `repoB/.git/config`). Consultez la documentation sur les dépôts distants pour plus de précisions.

```
1  $ git pull
2  remote: Counting objects: 3, done.
3  remote: Compressing objects: 100% (2/2), done.
4  remote: Total 3 (delta 0), reused 0 (delta 0)
5  Unpacking objects: 100% (3/3), done.
6  From /home/olivier/tmp/jnk/repoA
7   c939f1f..b88eea6 master -> origin/master
8  Updating c939f1f..b88eea6
9  Fast-forward
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
17 Adding a fourth line
18 Adding a fifth line
19 modification from repoA
20 $ git status
21 On branch master
22 Your branch is up-to-date with 'origin/master'.
23 nothing to commit, working directory clean
```

Pour récapituler :

- Le dépôt de référence fait donc un cycle classique :
  1. modifications,
  2. `git add`,
  3. `git commit`.
- Le dépôt clone se maintient à jour en faisant "`git pull`".

git push pour propager sur le dépôt d'origine les modifications que vous avez faites en local

Alors que dans la section précédente les modifications se faisaient toujours dans le même sens, nous allons maintenant voir que git vous permet de propager sur le dépôt d'origine des modifications que vous avez faites sur son clone. Puisque les modifications se font en sens inverse de celles du "`git pull`", il faut évidemment utiliser "`git push`" depuis `repoB`. Afin d'éviter les conflits potentiels lorsque plusieurs utilisateurs font des modifications sur leur clone, nous allons créer une branche spécifique dans laquelle nous allons faire nos modifications, puis nous allons envoyer cette branche dans `repoA` avec un "`git push`", et enfin fusionner cette branche avec la branche en cours de `repoA` avec un "`git merge`". La section suivante revient plus en détail sur les branches.

Dans l'exemple suivant, nous nous plaçons dans `repoB`, nous créons une nouvelle branche `myBranchFromB` (ligne 6) dans laquelle nous faisons des modifications sur le fichier `firstFile.txt` (lignes 8 à 10). Remarquez que lors du "`git commit`" lignes 11 puis 14, ces opérations se font sur la nouvelle branche `myBranchFromB`. Enfin, lors du "`git push`" les lignes 22 et 23 montrent que la nouvelle branche est transmise à `repoA`.

```
1  $ git pull
2  $ git status
3  On branch master
4  Your branch is up-to-date with 'origin/master'.
5  nothing to commit, working directory clean
6  $ git checkout -b myBranchFromB
7  Switched to a new branch 'myBranchFromB'
8  $ echo "modifications from repoB" >> firstFile.txt
9  $ git add firstFile.txt
10 $ git commit -m "improvements from repoB"
11 [myBranchFromB 098fc69] improvements from repoB
12  1 file changed, 1 insertion(+)
13 $ git status
14 On branch myBranchFromB
15 nothing to commit, working directory clean
16 $ git push origin myBranchFromB
17 Counting objects: 3, done.
18 Delta compression using up to 4 threads.
19 Compressing objects: 100% (3/3), done.
20 Writing objects: 100% (3/3), 328 bytes | 0 bytes/s, done.
21 Total 3 (delta 2), reused 0 (delta 0)
22 To /home/olivier/tmp/jnk/repoA/
23  * [new branch] myBranchFromB -> myBranchFromB
```

En revenant à `repoA`, nous pouvons vérifier que tant que l'on est sur la branche `master`, les modifications ne sont pas prises en compte (lignes 3 et 6 à 12), mais qu'elles sont bien dans la branche `myBranchFromB` que nous venons d'envoyer avec le "`git push`" (lignes 14, 16 et 19 à 26). Nous revenons donc à la branche `master` (ligne 27) pour incorporer les changements de la branche `myBranchFromB` avec un "`git merge`" (ligne 29).

```
1  $ cd ../repoA
2  $ git status
3  On branch master
4  nothing to commit, working directory clean
5  $ cat firstFile.txt
6  This is the first line of my first file...
7  ... and this is the second line
8  Now we add another line
9
10 Adding a fourth line
11 Adding a fifth line
12 modification from repoA
13 $ git checkout myBranchFromB
14 Switched to branch 'myBranchFromB'
15 $ git status
16 On branch myBranchFromB
17 nothing to commit, working directory clean
18 $ cat firstFile.txt
19 This is the first line of my first file...
20 ... and this is the second line
21 Now we add another line
22
23 Adding a fourth line
24 Adding a fifth line
25 modification from repoA
26 modifications from repoB
27 $ git checkout master
28 Switched to branch 'master'
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
35 This is the first line of my first file...
36 ... and this is the second line
37 Now we add another line
38
39 Adding a fourth line
40 Adding a fifth line
41 modification from repoA
42 modifications from repoB
```

À ce point, nous avons effectué des modifications dans la branche `myBranchFromB` de `repoB`, envoyé cette branche dans `repoA` et nous l'y avons fusionné avec la branche principale (appelée `master`) de `repoA`. Néanmoins, la branche `master` de `repoB` n'a pas encore été mise à jour. Il nous faut donc enfin faire un "`git pull`" depuis `repoB`.

```
$ cd ../repoB
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

Adding a fourth line
Adding a fifth line
modification from repoA
$ git pull
From /home/olivier/tmp/jnk/repoA
 842cc68..09805d8 master -> origin/master
Updating 842cc68..09805d8
Fast-forward
 firstFile.txt | 1 +
 1 file changed, 1 insertion(+)
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

Adding a fourth line
Adding a fifth line
modification from repoA
modifications from repoB
```

Pour récapituler :

- depuis le dépôt clone
  - n'oubliez pas de commencer par un "`git pull`" pour être certain de partir de la dernière version de référence
  - créez une nouvelle branche avec `git checkout -b nomNouvelleBranche`
  - faites un ou plusieurs cycles classiques
    - modifications,
    - `git add`
    - `git commit`
  - faites un "`git push`" (comme nous l'avons vu précédemment, il est possible de faire plusieurs cycles de "`commit`" atomiques puis un "`git push`" lorsque l'on est satisfait)
- depuis le dépôt origine
  - assurez-vous d'être sur la bonne branche (par exemple `master`) avec un "`git checkout nomBonneBranche`"
  - faites un "`git merge nomNouvelleBranche`" de la branche que vous venez de pousser depuis le dépôt clone
  - résolvez les conflits éventuels (cf. section suivante)
  - faites un "`git branch -d nomNouvelleBranche`" pour supprimer la branche qui est devenue inutile maintenant que vous l'avez intégrée
- depuis le dépôt clone
  - repassez dans la bonne branche (par exemple `master`)
  - faites un "`git pull`" pour récupérer la dernière version à jour avec les modifications que vous venez de pousser, les modifications éventuelles d'autres utilisateurs et la résolution des conflits éventuels.
  - faites un "`git branch -d nomNouvelleBranche`" sur la nouvelle branche pour la supprimer également.

Détection et résolution de conflits en cas de modifications concurrentes sur deux instances

Évidemment, la situation que nous venons de voir peut se révéler problématique si pendant que vous êtes occupé à faire vos modifications entre votre "`git pull`" et votre "`git push`" quelqu'un fait d'autres modifications sur le dépôt d'origine (ou fait un autre "`git push`" avant le vôtre).

Une fois de plus, git est là pour vous sauver la vie :

- si les modifications concurrentes portent sur des fichiers différents ou même des endroits différents d'un même fichier, git se débrouille pour tout intégrer (alors qu'avec de simples copies de fichiers, les modifications du dernier écraseraient les modifications antérieures)
- si les modifications concurrentes portent sur les mêmes endroits dans un fichier,
  - git vous **signale qu'il y a un conflit** (et vous dit où)
  - git vous **prépare la zone** en délimitant les deux modifications concurrentes
  - git vous **laisse gérer la résolution**

Là encore, notez bien l'intérêt de ne travailler sur des changements les plus petits possibles. Il est bien plus facile de fusionner deux branches ayant peu de modifications que deux branches ayant fortement divergé et comportant des différences aux mêmes endroits

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
$ echo "conflict from repoB" >> firstFile.txt
$ git add firstFile.txt
$ git commit -m "improvement from repoB likely to cause conflict"
[myBranchFromB db36634] improvement from repoB likely to cause conflict
 1 file changed, 1 insertion(+)
$ cd ../repoA
$ echo "conflict from repoA" >> firstFile.txt
$ git add firstFile.txt
$ git commit -m "improvement from repoA likely to cause conflict"
[master 348ff6b] improvement from repoA likely to cause conflict
 1 file changed, 1 insertion(+)
$ cd ../repoB
$ git push origin myBranchFromB
Counting objects: 26, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (26/26), 2.12 KiB | 0 bytes/s, done.
Total 26 (delta 11), reused 0 (delta 0)
To /home/olivier/tmp/jnk/repoA/
 * [new branch] myBranchFromB -> myBranchFromB
```

À ce point, nous avons ajouté une ligne dans le fichier `firstFile.txt` dans `repoB` et une autre ligne au même endroit du fichier `firstFile.txt` dans `repoA`. Nous procèdons ensuite comme à la section précédente pour envoyer sur `repoA` la branche `myBranchFromB` de `repoB`.

Il ne reste plus qu'à fusionner la branche `myBranchFromB` avec la branche principale de `repoA` (et les ennuis commencent).

```
1  $ cd ../repoA
2  $ git merge myBranchFromB
3  Auto-merging firstFile.txt
4  CONFLICT (content): Merge conflict in firstFile.txt
5  Automatic merge failed; fix conflicts and then commit the result.
6  $ cat firstFile.txt
7  This is the first line of my first file...
8  ... and this is the second line
9  Now we add another line
10
11 Adding a fourth line
12 Adding a fifth line
13 modification from repoA
14 modifications from repoB
15 <<<<<<< HEAD
16 conflict from repoA
17 =======
18 conflict from repoB
19 >>>>>>> myBranchFromB
```

Lors du "`git merge`", git détecte bien le conflit (ligne 4) et nous dit dans quel fichier cela s'est produit. Dans le fichier, git a ajouté les deux portions qui posent problème en les délimitant par des lignes de chevrons et en les séparant par une ligne de '=' (lignes 15 à 19). C'est alors à l'utilisateur d'éditer la zone à la main et de supprimer les délimitations.

```
$ vi firstFile.txt
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

Adding a fourth line
Adding a fifth line
modification from repoA
modifications from repoB
Manually-resolved conflict from repoA and repoB
$ git add firstFile.txt
$ git commit -m "Fixed conflicts from repoA and repoB"
[master ab5a47f] Fixed conflicts from repoA and repoB
$ git status
On branch master
nothing to commit, working directory clean
$ git branch -d myBranchFromB
Deleted branch myBranchFromB (was db36634).
```

Enfin, il ne reste plus qu'à mettre à jour `repoB`.

```
$ cd ../repoB
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 3 commits, and can be fast-forwarded.
 (use "git pull" to update your local branch)
$ git pull
Updating 09805d8..ab5a47f
Fast-forward
 firstFile.txt | 1 +
 1 file changed, 1 insertion(+)
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

Adding a fourth line
Adding a fifth line
modification from repoA
modifications from repoB
Manually-resolved conflict from repoA and repoB
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Créer des branches et des versions

Intérêt des branches

Dans la section sur "`git push`", nous avons eu un premier aperçu de la notion de branche. En fait, git permet de créer autant de branches que nous en avons besoin. La documentation de git comporte des explications très claires sur la notion de branche qui en plus abordent des points un peu plus techniques comme la notion de HEAD.

Les branches sont au cœur d'organisation des projets avec git. Un exemple d'usage est le marquage de version. Les tags, que nous verrons plus loin, c'est cool, mais si vous passez en version majeure suivante (donc, brisure de compatibilité), il faut aussi penser à ceux qui ne suivront pas l'update et resteront à la version «ancienne», et qui voudront des corrections de bugs et même des mises à jour. Un exemple parlant est python : il y a deux versions qui cohabitent, et même si l'une est destinée à mourir bientôt, elle est encore développée et enrichie. Même si la réalité est plus complexe, nous pouvons imaginer qu'un repo git de Python serait composé de deux branches principales, «Python2» et «Python3», qui permettent au deux versions de cohabiter et d'évoluer indépendamment. Il y en aurais bien d'autres (une pour chaque fonctionnalité en cours de développement), mais ces deux là seraient les «master».

Un vrai exemple du monde de la vérité vraie : la SFML, une librairie graphique initiée et maintenue par un français (cocorico), et qui initialement était au C++ ce que la SDL est au C. Les branches sont ici utilisées pour les différents portage de la librairies sur différents langages, et github permet de les visualiser joliment. Nous voyons également très bien toutes les branches dont le nom commence par «bugfix», qui sont en fait créées uniquement pour résoudre un des problèmes identifiés par le mécanismes de gestion de problèmes, qui ne sera pas abordé ici, car cela n'a rien à voir avec git (pas directement). Nous trouvons également les branches ayant pour nom «2.3.x» par exemple, et nous pouvons constater que de nombreuses personnes créèent leur propre branche (ça s'appelle un «fork», et c'est une notion à la base du développement de logiciel libre) pour développer une fonctionnalité ou corriger un bug, puis effectuent un merge de leur branche avec la branche principale (via les pull requests, où une personne demande à ce qu'une de ses branches soit mergée avec une branche d'un autre repo qui ne lui appartient pas).

En général, un dépôt git est composé d'au moins deux branches, souvent nommées master et dev. Master est la branche dite principale, où le code est stable. Autrement dit, lorsque vous récupérez les fichiers de la branche master d'un projet, vous devriez avoir le code le plus fonctionnel possible. (c'est souvent la release qui est stockée en master)

La branche dev, quant à elle, est une branche où le code est en développement. C'est à partir de celle-ci que se créent des branches telles que «bugfix53» qui va implémenter la correction du bug numéro 53. Lorsque la branche dev est stable et pleine de nouvelles fonctionnalités, elle est mergée avec master, qui est alors passé à la version suivante. (et il est possible de placer des tags sur les derniers commits pour montrer où en est la version)

Quand le projet devient plus gros, ou, mieux, qu'une nouvelle version importante va sortir, c'est une bonne idée de créer un fork de master nommé selon la version (par exemple «4.x», ou «5.3.x»). Dés lors, comme la branche dev continue de merger avec le master, les fonctionnalités n'ayant pas été ajoutées avant devront attendre la prochaine version du logiciel (et donc la prochaine branche) pour être utilisées. Avec ce système, il est possible de se passer de la branche dev et créer les nouvelles fonctionnalités directement en forkant le master.

Autre exemple : sur le repo de Julia, nous observons plus de 183 branches et 192 pull requests. Chaque branche est créée par un contributeur avec un nom adapté au problème auquel cette branche va répondre, et lorsque c'est terminé, la branche sera «mergée» à la branche principale.

Souvent, il y a un peu de discussion pour que le code ajouté soit parfait, juste, en accord avec le projet, n'introduise pas de conflit, ou pour avoir des messages de commit de meilleure qualité.

Certaines personnes attendent de leurs contributeurs qu'ils suivent une ligne de conduite parfois assez stricte. Cela peut sembler tyrannique, mais c'est comme obliger un style de code dans un projet à plusieurs : sans uniformisation, le code est d'autant plus dur à lire et hétérogène dans sa construction, alors autant se forcer à écrire les choses de la même façon dés le début.

Le scénario suivant est fortement inspiré de la documentation et vous donne l'idée générale :

- vous faites une branche pour ajouter une nouvelle fonctionnalité
- en plein milieu on vous demande de corriger un bug en urgence
    - vous ne le faites pas dans votre nouvelle branche car le code n'est pas encore fonctionnel
    - vous ne le faites pas non plus dans la branche principale car celle-ci ne doit pas être utilisée comme branche de développement
        - vous créez une seconde branche depuis la branche principale pour corriger le bug
        - vous fusionnez cette seconde branche avec la branche principale une fois la correction terminée
        - vous terminez votre ajout de fonctionnalité sur la première branche
        - vous fusionnez votre première branche avec la (nouvelle) branche principale en réglant les conflits éventuels

Commandes utiles pour manipuler les branches

La commande "`git branch`" permet de lister les branches existantes. Elle a notamment deux filtres :

- "`git branch --merged`" indique uniquement les branches qui ont déjà été fusionnées.
- "`git branch --no-merged`" indique uniquement les branches qui n'ont pas encore été fusionnées.

La commande "`git checkout nomDeLaBranche`" permet de passer sur la branche nomDeLaBranche (qui doit déjà exister), et qui devient alors la branche courante.

La commande "`git branch -b nomDeLaBranche`" permet de créer une nouvelle branche. La branche ainsi créé est identique à la branche

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

action ne sera opérée, et git expliquera la marche à suivre pour résoudre le problème ; en général cela arrive quand vous essayez de détruire une branche que vous n'avez pas mergée. Détruire malgré tout la branche (avec l'option -D, comme l'expliquera git) supprimera DÉFINITIVEMENT les modifications n'ayant pas été rapatriées sur une autre branche. Ce cas de figure arrive typiquement lorsque vous développez une correction de bug sur une branche dédiée, et que le bug est corrigé par quelqu'un d'autre : votre branche est inutile, elle peut donc être détruite.

Enfin, la commande "`git merge nomDeLaBranche`" permet de fusionner la branche nomDeLaBranche avec la branche courante, en détectant éventuellement les conflits. La branche cible (nomDeLaBranche) ne sera pas modifiée lors de cette opération.

Utiliser des tags pour indiquer les versions

Nous avons vu que `git log` donne l'historique de tous les `commit`. Nous avons aussi lourdement insisté sur les avantages de faire de nombreux `commit` atomiques plutôt qu'un gros commit chaque fois que vous produisez une nouvelle version stable. On comprend donc bien que repérer parmi cette longue liste *le* `commit` qui correspond à la version 5.2 de votre projet ne va pas être facile, d'autant plus que le message associé à chaque commit décrit ce qui a changé, et pas forcément le fait que ce commit constitue une étape importante pour le projet.

En plus du message de description, vous pouvez associer un tag à un `commit`. En fait git supporte deux types de tag:

- un **tag léger** (*lightweight tag*) est simplement une association entre une chaîne de caractères (le tag) et un commit ;
- un **tag annoté** (*annotated tag*) est un objet qui contient la chaîne de caractères ainsi que la date, l'identité de l'auteur, etc. Il peut éventuellement être signé.

On utilise principalement les tags légers pour la maintenance, les besoins temporaires et les affaires internes, et les tags annotés pour les annotations importantes.

La commande `git tag` renvoie la liste de tous les tags d'un projet, triés par ordre alphabétique.

La commande `git show someTagValue` renvoie les informations du tag et une description du commit associé.

La commande `git tag -d someTagValue` détruit le tag `someTagValue`.

Créer un tag léger

La commande "`git tag someTagValue`" permet de créer un tag léger. Le tag est alors associé au dernier `commit`.

La commande `git tag someTagValue commitCheckSum` permet d'associer un tag léger à un `commit` antérieur. La commande `git log` (mais c'est assez verbeux), ou mieux `git log --pretty=oneline` est alors bien pratique pour retrouver la valeur de checksum de chaque commit en faisant.

```
$ git tag
$ git log --pretty=oneline
ab5a47f1438ac91c1403cfe348b1ffe17a517646 Fixed conflicts from repoA and repoB
348ff6b484cb63d5b526fa160773fc19b2b0c388 improvement from repoA likely to cause conflict
db36634059cb8a18707b440cf76fc7aab648685c improvement from repoB likely to cause conflict
09805d8e4a418e2dfcda0bc1c6f7a70fa4c7c8d4 improvements from repoB
842cc6890d9bf62cd0817a7cb14ecf81c525ebaf some improvement from repoA
7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9 added a .gitignore file to ignore backup files and the log directory
fcae2678d2c8c19933c8d64aeb3dff205b654421 added two more lines to explain how git diff works
61b5f1da1168e3bfd89a10c65dd1dab6ee25d3fd this commit only adds the fourth line; the fifth is ignored as it is not yet in the index
e26b158e016d50b75a551438ccaa936b59afb799 First commit
$ git tag afterConflictResolution
$ git tag
afterConflictResolution
$ git tag gitignore 7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9
$ git tag
afterConflictResolution
gitignore
$ git show gitignore
commit 7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9
Author: Gérard Menvuça <super.gege@wanadold.fr>
Date: Sat Apr 11 00:28:19 2015 +0200

 added a .gitignore file to ignore backup files and the log directory
```

Créer un tag annoté

la commande `git tag -a someTagValue -m "some description for the tag"` permet de créer un tag annoté. Là encore, cela associe le tag annoté au dernier `commit`, et il suffit de préciser le checksum si on veut associer le tag à un `commit` antérieur.

Dans l'exemple ci-dessous, nous associons le tag annoté `v2.0` au dernier `commit`, et le tag `v1.0` au `commit` concernant le fichier `.gitignore`, juste avant la partie sur la résolution de conflits. Remarquez que le tag léger `gitignore` et le tag annoté `v1.0` sont associés au même commit, et que les commandes `git show` respectives montrent les informations supplémentaires associées au tag annoté.

```
$ git log --pretty=oneline
ab5a47f1438ac91c1403cfe348b1ffe17a517646 Fixed conflicts from repoA and repoB
348ff6b484cb63d5b526fa160773fc19b2b0c388 improvement from repoA likely to cause conflict
db36634059cb8a18707b440cf76fc7aab648685c improvement from repoB likely to cause conflict
09805d8e4a418e2dfcda0bc1c6f7a70fa4c7c8d4 improvements from repoB
842cc6890d9bf62cd0817a7cb14ecf81c525ebaf some improvement from repoA
7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9 added a .gitignore file to ignore backup files and the log directory
fcae2678d2c8c19933c8d64aeb3dff205b654421 added two more lines to explain how git diff works
61b5f1da1168e3bfd89a10c65dd1dab6ee25d3fd this commit only adds the fourth line; the fifth is ignored as it is not yet in the index
e26b158e016d50b75a551438ccaa936b59afb799 First commit
$ git tag -a v2.0 -m "version 2.0 after conflict resolution"
```

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

```
afterConflictResolution
gitignore
v1.0
v2.0
$ git show v1.0
tag v1.0
Tagger: Gérard Menvuça <super.gege@wanadold.fr>
Date: Fri Apr 17 01:59:54 2015 +0200

version 1.0 including .gitignore

commit 7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9
Author: Gérard Menvuça <super.gege@wanadold.fr>
Date: Sat Apr 11 00:28:19 2015 +0200

 added a .gitignore file to ignore backup files and the log directory
$ git show gitignore
commit 7deabd7f55558ab1ce5d7f5c55d13e358f4f51c9
Author: Gérard Menvuça <super.gege@wanadold.fr>
Date: Sat Apr 11 00:28:19 2015 +0200

 added a .gitignore file to ignore backup files and the log directory
```

Les forges logicielles

Sous ce nom barbare débordant de sens complexes et pointus se cachent des sites que beaucoup de monde utilise parfois sans le remarquer. L'un des exemples les plus connus est sourceforge, qui est utilisée par beaucoup de logiciels pour y être... forgés.

Le principe d'une forge logicielle est de permettre à un (groupe d')utilisateur(s) ou une association de travailler collaborativement sur un projet, avec une uniformisation des outils de gestion. Et dans gestion, il y a gestion de versions, dont parle cet article, mais pas que. Une forge logicielle propose en général un wiki intégré, un système de tickets, voire des logiciels de gestion de projet pour générer des diagrammes de gantt, de l'UML,... En d'autres termes, la forge logicielle est le terrain de jeu de la science du génie logiciel, et en utiliser les bases ne peut qu'être utile et efficace.

Par exemple, dans le dépôt sourceforge de Potassco, on trouve les fichiers dans un dépôt svn (un autre gestionnaire de versions), deux mailing lists, un système de tickets pour remonter les bugs et proposer des modifications (de futures branches, pour enrichir de futures versions !), un wiki,...

Dans ce tuto, nous avons beaucoup vu de liens vers une des plus fameuses forges : github. Bien qu'elle ne soit en elle-même pas libre (son code source n'est pas distribué ni sous licence libre), c'est une des plateformes principales de l'univers du libre. Elle propose, gratuitement pour les projets open-source, payant sinon, un espace de stockage illimité pour les projets, un unique gestionnaire de versions (git !) avec une interface graphique dédiée à la visualisation des dépôts et pull requests, un système de wiki, de gestion de problèmes,...

Il est à noter qu'au moment où ces lignes sont écrites, deux forges logicielles ont disparu en peu de temps : google code et gitorious ont été respectivement fermée et rachetée, car trop forte concurrence ou absorption par un autre projet de forge. Une autre, et pas des moindres, sourceforge, commence sérieusement à inquiéter de par son comportement que d'aucun diront amoral, au point que de nombreux projets, suite aux déboires du projet The Gimp, migrent vers d'autres forges (ce dernier lien montre au passage une bonne quantité de forges, avec les gestionnaires de versions supportés par chacune)

Les forges logicielles forment un sujet suffisamment vaste pour être traité dans un article dédié. En général, regarder les offres et fonctionnalités de chacune d'entre elles suffit à se faire une idée. Parmi les autres alternatives :

- gitlab, une solution efficace pour l'auto-hébergement, ou comme solide alternative à github.
- framagit, tenu par l'association Framasoft et basée sur gitlab. Le nombre de dépôts est limité, mais c'est une asso française et, pour le coup, des plus libristes qui soient !
- bitbucket, qui propose notamment une offre gratuite pour les petits dépôts privés.
- gogs, versé dans l'auto-hébergement (merci Pierre Marijon).
- gitchain, un nouveau projet orienté pure décentralisation, ce qui parait important, dans ce monde où l'on a trop tendance à faire confiance au «cloud».
- redmine, une forge très complète, utilisée dans le domaine du génie logiciel.
- SourceSup, la forge de Renater.

Choisir une forge n'est pas une décision primordiale : vous arriverez certainement à travailler sur plusieurs projets, tous hébergés dans une forge différente. Il peut également arriver qu'un projet change de forge ; d'abord parce que rien ne l'interdit, mais aussi parce que parfois, comme pour sourceforge, la philosophie d'une forge est remise en question et que les créateurs d'un projet préfèrent rejoindre une forge plus en adéquation avec leurs besoins. Pour passer d'une forge à l'autre, quelques commandes suffisent souvent.

Sachez qu'avec tout ce que vous savez de git, il ne vous est pas difficile de créer un compte sur gitlab ou framagit, et de vous lancer ! Les seules commandes qui changeront seront les premiers appels à git clone, git push et git pull. Il faudra donner des URL ou des arguments particuliers, qu'en général la forge vous donne directement pour vous faire gagner du temps.

Aujourd'hui, il est rare qu'un projet n'ait pas une visibilité sur une ou plusieurs forges logicielles. Les implémentations de langages, par exemple, sont très visibles par ce biais, ainsi que le noyau linux ou galaxy.

Au niveau de l'enseignement en informatique, certains professeurs observent les statistiques offertes par la forge pour juger rapidement du partage des tâches et de la régularité de travail des membres d'une équipe ; par exemple, github se prête assez bien à ce jeu car fournissant une grande quantité de graphiques colorés et rigolos.

Tuner son git

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Je suis certain que beaucoup d'entre vous trouvent extrêmement long les commandes de git, telles que git commit, avec 10 touches à frapper sans compter «enter», ou git pull, qui donne l'impression de ne fonctionner qu'en hiver. Eh bien, c'est comme le bashrc pour remplacer «ls» par «l» et «cd ..» par «c.» : nous allons faire du tuning !

À ce stade de l'article, vous devriez avoir un fichier ~/.gitconfig. Sinon, créez-le et demandez-vous si vous n'avez pas sauté la première partie. C'est ce fichier que nous allons modifier sauvagement :

```
[user]
    name = Gerard
    email = gerard.menvuca@wanadold.py
[alias]
    st = status
    ci = commit
    cim = commit -m
    cam = commit -Am
    cia = commit --amend
        yolo = commit -Am "deal with it"
    br = branch
    co = checkout
    df = diff
    dfs = diff --staged
    lg = log -p
    lo = log --decorate
    lol = log --graph --decorate --pretty=oneline --abbrev-commit
    lola = log --graph --decorate --pretty=oneline --abbrev-commit --all
    pa = add --patch
        a = add
        ap = add -p
    ai = add -i
    who = shortlog -s --
    ps = push
    pl = pull
    diffstat = diff --stat -r
        ss = stash
        sa = stash apply
        sp = stash pop
        sl = stash list
        sd = stash drop
```

Ça c'est du tuning ! Les trois premières lignes devraient déjà exister, et d'autres aussi éventuellement (certaines ont été retirées dans l'exemple, car ce fichier contient généralement des valeurs privées pour authentification auprès de forges). Si elle n'existe pas déjà, créez la section [alias] et mettez-y des alias cools et musclés pour vous simplifier la vie. Des plus utiles au moins utiles (d'après l'auteur) :

- ceux qui font économiser deux touches : «a», «df», «ps», «pl», voire 4 pour «ci», «br» et «co»
- ceux qui évitent d'oublier des caractères éloignés sur le clavier : «cim», «cia», «ap», «dfs»
- ceux qui permettent d'être efficace avec les stashs : «s*» (voir les liens plus bas)
- ceux qui font éviter des options zarbi : «lol», «who»
- ceux qu'on n'utilise jamais : «yolo», «who», «diffstat» (trop longue à taper…)

Voilà ! Libre à vous de recopier ces lignes, d'en enlever, d'en ajouter,… Bref, de les adapter à votre usage. Toutes ne sont pas utiles, mais elles donnent des idées et des pistes pour ceux qui veulent booster un peu leur productivité.

En prime, voilà quelques alias à mettre dans son bashrc/zshrc, qui se couplent assez bien avec le gitconfig :

```
alias 'g'='git'
alias 'gps'='g ps'
alias 'gpl'='g pl'
… etc
```

La carrosserie

Moulte chemins mènent à l'épanouissement de git dans votre outils de travail : si par chance vous n'utilisez pas d'interface graphique autre que celle de votre (émulateur de) terminal, il existe de nombreux modules, packages, layouts, hack propres et moins propres qui permettent à git. d'être un poil plus visuel, coloré ou rigolo.

De plus, de nombreux éditeurs de textes et IDE proposent des facilités d'usage de git. Entre autres :

- vim propose trois modules d'intérêt (il y en a beaucoup d'autres) qui permettent de jouer avec git simplement : le puissant fugitive, l'utile extradite et l'informatif gitgutter,
- bash et zsh peuvent être tunés avec oh my git.
- python possède un module pour intéragir avec git : pygit.

Le mot de la fin

Tuner son git, c'est comme tuner son bashrc : c'est gagner beaucoup de temps en s'amusant, et c'est nécessaire pour faire de son environnement de travail un lieu productif et simple. Cela mériterait un article à part entière, mais ici nous nous contenterons de ce simple énoncé : que ce soit avec les alias ou les gestionnaires de versions, le principe est le même : faites bosser l'ordinateur, surtout pas vous !

Soyez fainéants, et vous vous simplifierez la vie.

Du git dans les Internets

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

- une explication par bitbucket des contributions [EN] : via
- une question de fond par framasoft sur l'usage des forges comme github, problématique à laquelle répond en partie gitchain : via
- comment contribuer sur github (ça s'applique à toutes les autres forges) : via
- il existe aussi pas mal de tuto vidéo, essentiellement en anglais : via
- stackoverflow propose un tag git, et un tas de réponses à des questions allant du simple au complexe : via (ou ici pour les gestionnaires de versions en général, ou là pour une vision plus abstraite)
- un article court mais intense sur les bases : via
- il existe des centaines de *cheatsheet* sur internet, certaines interactives : via via via (merci hedjour)
- plus d'info sur sourceforge et ses nouvelles manies [EN] : via (réponse de sourceforge)
- configurer git pour utiliser un éditeur particulier : via
- si vous voulez jouer avec git : via ou découvrir un autre usage des branches : via (merci Yoann M.)

Conclusion

Nous avons vu comment cloner un projet, faire des modifications sur l'un ou l'autre des clones (voire sur les deux) et propager ces modifications en tenant compte de la nature décentralisée de git. Cela illustre trois des intérêts majeurs des gestionnaires de versions : permettre à plusieurs personnes de partager une base commune, la modifier selon leurs besoins tout en en faisant profiter les autres, et enfin de détecter les conflits qui peuvent survenir lors de modifications concurrentes.

Enfin, nous avons rapidement abordé l'aspect communautaire que permettent les gestionnaires de versions. Même si il y avait emphase sur git, c'est la même chose pour l'ensemble des gestionnaires de versions : puisque le partage de code est simple et structuré, il devient aisé de contribuer à un projet sans bouger de son pc. Et avec le principe des forks et des pull requests, vous avez les bases pour vous lancer dans la contribution effrénée et dans les tutoriaux plus avancés : il est bien sûr possible de faire des choses beaucoup plus raffinées. Ces deux articles présentent les principes de base permettant de se sortir de la plupart des situations pas trop tordues. Nous sommes convaincus qu'une utilisation même limitée d'un gestionnaire de versions vaut mieux que pas de gestionnaire de versions du tout.

Remerciements

Merci à Estel, nallias, hedjour et Yoann M pour les commentaires et discussions lors de l'édition de cet article.

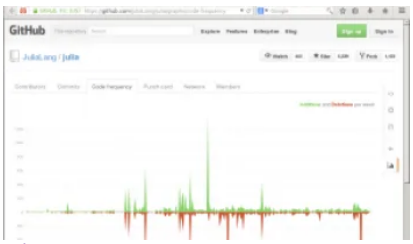
À propos de cet article

Cet article et le précédent ont étés adaptés à partir d'un cours donné par Lucas Bourneuf.

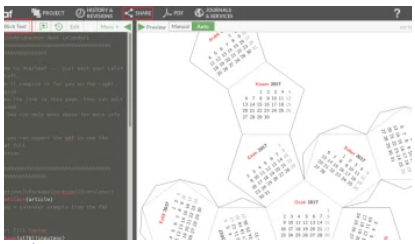
Il a été rédigé par Lucas Bourneuf et Olivier Dameron.

Partager :

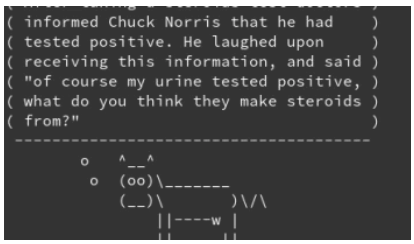
Articles similaires



Gérer les versions de vos fichiers : premiers pas avec git
mer 20 Mai 2015
Dans "Découverte"



Créez vos documents collaboratifs en LaTeX
jeu 29 Juin 2017
Dans "Découverte"



Pimp my workstation: avoir le terminal le plus classe du bureau !
jeu 15 Juin 2017
Dans "Didacticiel"

- **À propos de Article Collaboratif**
-

Catégorie: Didacticiel | Tags: gestionnaire de version, git, programmation
S'abonner au flux de commentaires de cet article

5 commentaires sur “Git : cloner un projet, travailler à plusieurs et créer des

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

1.

Pierre

juin 11, 2015 à 10:01

Je m'insurge devant la politique pro-vim de ce billet il n'a été fait nullement mention d'emacs et des merveilleux module pour jouer avec git dans emacs. Tout est là http://emacswiki.org/emacs/Git ça va du plus simple et user friendly au plus complexe.

Répondre

- **Olivier**

  juin 11, 2015 à 10:38

  C'est parce que c'est un article sérieux, Môssieur, et quitte à expliquer autant ne pas parler des éditeurs en plastique

  Répondre

- **lucas**

  juin 17, 2015 à 11:17

  Ce n'est pas si pro-vim que ça : on aborde nano dans la partie précédente… (c'est l'éditeur par défaut de git !)

  Répondre

3.

Lucas Bourneuf

mars 22, 2016 à 8:17

Un tutoriel en anglais pour git et les designers :
https://medium.com/@dfosco/git-for-designers-856c434716e#.vdeneq3py

Répondre

4.

Pascal

juin 24, 2016 à 8:35

Ce tutoriel est très réussi. Merci pour ce partage. Moi, j'ai trouvé très intéressant les fonctionnalités avancés du Git sur http://www.alphorm.com/tutoriel/formation-en-ligne-git-fonctionnalites-avancees.

Répondre

Laisser un commentaire

Entrez votre commentaire...

← Précédent Suivant →

Recherche

La boutique officielle de Bioinfo-fr.net !

Portez haut les couleurs de votre blog de bioinfo préféré !

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Les profits nous aideront à maintenir le blog et à vous faire de beaux cadeaux :)

Les snippets de Bioinfo-fr

Partagez vos bouts de code !

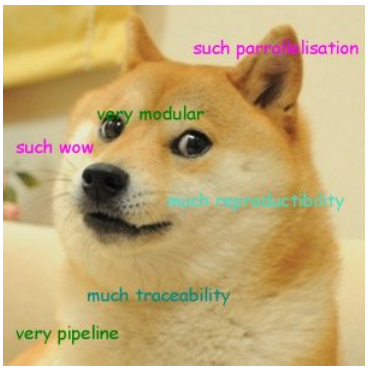
canSnippet

Articles au hasard

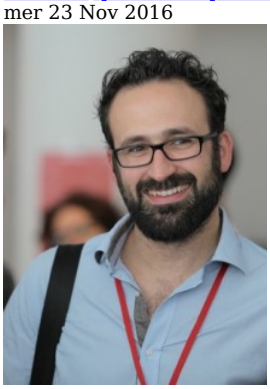
- Doctorant dis toi que...
  mer 29 Jan 2020

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok



- Nextflow, pour votre prochain pipeline ?
mer 23 Nov 2016



- Questions à... Julien Maupetit
mer 14 Oct 2015

- The Bio Code : guide du bon broinformaticien
mer 21 Mai 2014

- Julia: le successeur de R ?
mer 23 Oct 2013

Liens

- BioStar
- canSnippet
- JeBiF
- Les Bioinformations
- SFBI

Catégories

- Actualité (17)
- Astuce (36)

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

- Editorial (58)
- En image (11)
- Entretien (13)
- Formation (14)
- J'ai lu (9)
- Journal Club (5)
- Opinion (31)
- Suivez l'guide (27)

Commentaires récents

- L'art de dessiner des graphes – archivEngines dans Tour d'horizon des outils de visualisation des réseaux biologiques
- Léopold Carron dans Hi-C: Quelques bases
- Bioss dans Hi-C: Quelques bases
- blob dans LaTeX : automatisez le traitement des CSV
- lhtd dans Covid-19 : Des liens utiles pour aider à la recherche contre le SARS-CoV-2

Étiquettes

ADN analyse base de données bioinformatique code Communauté concours conférence débutant Découverte edito emploi formation GenBank Génomique Interview JeBiF JOBIM langage LaTeX master modélisation Métagénomique NCBI NGS outil Perl phylogénie pipeline programmation protéine Python R rentrée script SFBI SQL statistiques strip séquençage thèse Tips tutoriel vacances visualisation

Sauf mention contraire, tous les contenus sont publiés sous licence CC-by-SA 2.0 et supérieure.

Pour les scripts et binaires, référez-vous à la licence associée attribuée par l'auteur.

Pour tout renseignement supplémentaire : admin AT bioinfo-fr.net.

Design: imago-fr.org.
Aller en haut

Nous utilisons des cookies pour vous garantir la meilleure expérience sur notre site. Si vous continuez à utiliser ce dernier, nous considérerons que vous acceptez l'utilisation des cookies.

Ok

Les commandes Git indispensables

Laisser un commentaire

Incontestablement, Git est un atout majeur dans le développement. Sa force réside dans sa souplesse d'usage et sa grande puissance. Bien qu'il possède de très nombreuses commandes, seule une partie d'entre elles sont réellement indispensables dans un usage quotidien. Focus sur les commandes Git les plus courantes.

Au programme

Format des options

Vous verrez tout au long de ce livre que les options existent en deux formats :

```
# Format strict
git <command> --<option>=<param>
```

```
# Format light
git <command> --<option> <param>
```

En général, Git supporte les deux. Si vous obtenez une erreur, essayez l'autre format.

Configuration de Git

Avant toute chose, configurons rapidement Git. Cela nous permettra de gagner du temps par la suite.

La première chose à faire est de définir notre identité : l'email et le nom d'utilisateur utilisé lorsque nous faisons un commit.

```
git config --global user.name "Buzut"
git config --global user.email "buzut@mail.com"
```

Votre nom est de forme libre : "Pseudo" ou "Prénom Nom" par exemple. Nous verrons plus tard qu'il est également possible d'appliquer des configurations différentes à l'échelle d'un projet.

On passe ici le *flag* `--global`, pour dire à Git que l'on définit la configuration globale pour l'utilisateur courant. Il

existe aussi le paramètre `--system`, lequel définit la configuration pour tous les utilisateurs du système.

Windows est un peu spécifique quant à la localisatin des fichiers de configuration, mais dans les cas macOS, Linux et Unix :

- la configuration système se trouve dans `/etc/gitconfig`,
- la configuration globale se trouve dans `~/.gitconfig`,
- la configuration d'un projet se trouve à la racine du projet dans `.git/gitconfig`.

La configuration est consultable par un simple `git config -l`. Cela vous permettra de connaître la stratégie utilisée par Git pour la gestion des mots de passe ; le paramètre s'appelle `credential.helper`.

S'il n'est pas renseigné, cela veut dire que vous aurez à taper le mot de passe à chaque fois si vous utilisez du HTTPS. Si vous êtes sous Mac, il y a de grandes chances que la stratégie soit `osxkeychain`.

Si vous souhaitez utiliser le stockage natif du mot de passe par Git, il suffit de le lui demander.

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

Dans le cas des tratégies propres à Windows et macOS, une fois installée (nous avons abordé le sujet dans le chapitre précédent), vous utilisez la commande précédente en remplaçant `cache` par la stratégie souhaitée.

Créer des alias

Git permet la création d'alias, très commode pour les commandes souvent utilisées ou pour les options que l'on voudrait associer par défaut à une commande.

Comme pour tout élément de la configuration, vous pouvez les définir au niveau système, global ou local.

```
# Ajoutons quelques commandes en alias
git config --global alias.aa git add -a
git config --global alias.ca git commit -a
git config --global alias.st git status
```

Vous pouvez maintenant faire un `git add -a` en tapant `git aa` et `git status` en tapant `git st`. Toutefois, n'oubliez pas qu'en plus des alias de Git – qui requièrent

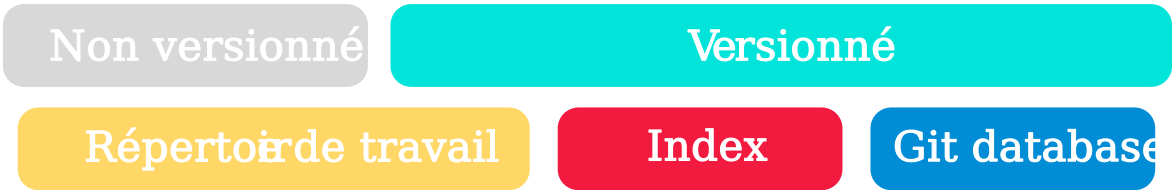
toujours que vous invoquiez `git` – vous pouvez aussi tirez parti des alias de votre shell. Ainsi, pour ma part, le simple fait d'écrire `git xxx` s'avère parfois déjà contraignant pour une commande que je tape très souvent. Via un alias de Bash, `git add -a` se transforme en `gaa`, bien plus concis !

Demander de l'aide

Notez que pour obtenir de l'aide sur une commande en particulier, vous pouvez toujours le demander à Git.

```
git commande --help

# ou
git help commande
```

Création d'un dépôt Git



Nous connaissons déjà ce schéma, je le remets ici afin que vous l'ayez bien en tête. Dans un flux de travail classique, lorsque l'on commence un nouveau projet, on créé alors un dépôt vide.

```
git init
```

À ce stade, aucun fichier n'est alors suivi par Git. Nous sommes dans le cas de notre schéma "Non versionné".

Ajout de fichiers à l'index

```
# Dire à git de suivre un fichier
git add <file> <otherfile>

# On peut utiliser les patterns habituels
git add *.js *.jpe?g

# Ajoute tous les fichiers
```

```
git add .
```

À ce stade, le fichier est alors indexé. Vous pouvez dès lors effectuer d'autre modifications en sachant qu'elles n'impacteront pas celles que vous avez déjà enregistrées dans la staging area.

Interprétation du shell

Que vous utilisiez bash, zsh ou autre, le shell procède automatiquement à l'expansion des patterns contenant "*". De ce fait, il est recommandé de le mettre entre guillemets si vous souhaitez qu'il soit passé tel quel à Git.

Dans un répertoire contenant `main.js`, `test.js` et `es6/main.js` ; `git add *.js` sera implicitement transformé en `git add main.js test.js` tandis que `git add "*.js"` sera passé tel quel par Git : `es6/main.js` sera alors également ajouté.

Si vous souhaitez enlever un fichier de la zone d'index, il faut utiliser la commande `restore`, deux options sont alors possibles.

```
# Simple annulation du git add
```

```
git restore --staged fichier

# Restauration du fichier dans son état avant mo
git restore fichier
```

La première version fait exactement l'inverse de `git add`, la deuxième version supprime totalement les dernières modifications non commitées, soyez donc vigilent ! Cette commande prend obligatoirement un chemin de fichier. Si vous voulez tout restaurer, pensez au caractère `.`.

`restore` et `reset`

Historiquement, on utilise à la commande `reset` pour annuler des changements et enlever des fichiers de l'index. Cependant, cette commande possède de très nombreuses possibilités et, mal comprise, elle pouvait mener à des effets désastreux.

Git a ainsi introduit dans la version 2.24 la commande `restore` afin d'annuler des changements. Toutefois, la commande `reset` reste tout à fait fonctionnelle. Nous verrons dans un prochain chapitre ses possibilités.

Comme il y a de grandes chances que vous soyez confronté à la commande `reset`, notamment sur les forums, voici comment l'utiliser pour annuler des changements.

```
# Enlève simplement les modifications de l'index
# Si un fichier est précisé, cela affecte juste le fi
# Sinon, cela affecte tout le projet
git reset [fichier]

# Restaure le répertoire de travail dans son état d'a
git reset --hard
```

Par ailleurs, `reset --hard` ne peut s'utiliser en précisant un chemin de fichier, cela s'applique donc forcément à tout le projet.

Sauvegarder les modifications

Nous sommes satisfaits de nos modifications, il est donc temps d'effectuer notre premier commit !

```
git commit
```

Cette action lance l'éditeur par défaut du système d'exploitation – il est possible de le préciser à Git, c'est l'option `core.editor`. Vous n'avez plus qu'à entrer votre

message de commit et à valider. Il existe aussi une commande courte permettant d'effectuer cette action sans passer par l'éditeur.

```
# Nous voulons commiter des modifications du REA
git commit -m "add documentation to README"
```

Il est aussi possible de sauter l'étape de staging, pratique lorsque l'on a travaillé sur quelques modifications qui sont toutes à inclure dans le même commit.

```
# Toutes les modifications des fichiers trackés
git commit -a

# On peut bien entendu combiner les options
git commit -am "add documentation to README"
```

Effacer et déplacer des fichiers

```
# Effacer un fichier (arrête le suivi et efface
git rm monfichier
```

```
# Arrêter de suivre un fichier mais ne pas l'eff
git rm --cached monfichier

# Déplacer des fichiers
git mv fichierSrc fichierDest
```

En réalité, de ces trois commandes, seule la désindexation est réellement utile. En effet, Git est bien assez sophistiqué pour se rendre compte par lui même qu'un fichier a été déplacé, a changé de nom ou a été supprimé.

Modifier le dernier commit

Il arrive fréquemment qu'on réalise après avoir commité que l'on veut modifier le message de commit ou modifier notre dernier snapshot. C'est possible, il suffit de commiter en passant l'option `--amend`.

Si des fichiers ont été placés dans la staging, ils seront alors ajoutés au dernier commit. Dans le cas où aucun fichier n'a été ajouté à l'index, l'option `--amend` servira simplement à modifier le message de commit.

```
# Ajoute les nouvelles modifications au dernier
git commit --amend
```

```
# Idem mais en modifiant le message du commit
git commit --amend -m "nouveau message de commit
```

Statut du dépôt

Comme on a tendance a faire beaucoup de choses, il est facile de ne plus trop savoir où l'on en est. `git status` est là pour ça.

```
# On voit ici que nous n'avons encore rien ajout
# Et que nous avons des fichier non suivis
git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what wi
    README.md
    index.html

nothing added to commit but untracked files pres
```

On obient en quelque sorte un rapport de situation. Ajoutons donc de nouvelles modifications.

```
# On modifie un fichier déjà tracké
echo "console.log('hello world')" >> main.js

# On affiche le statut de nouveau
git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will b
  (use "git restore <file>..." to discard change
    modified:   main.js

Untracked files:
  (use "git add <file>..." to include in what wi
    README.md
    index.html

no changes added to commit (use "git add" and/or

# On ajoute donc les nouveaux fichiers et le fic
git add .

# Une fois de plus, on demande à Git quelle est
git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unsta
    new file:   README.md
    new file:   index.html
```

```
    modified:   main.js

# On commit alors le tout
git commit -m "ajout du hello world"

# Dernier coup de status pour voir ce qu'il en e
git status
On branch master
nothing to commit, working tree clean
```

Lorsque vous voulez profiter d'un statut plus concis, l'option `-s`, `--short` en version longue, est la bienvenue.

Ignorer des fichiers

Les exemples de situations dans lesquelles vous ne voulez pas inclure un fichier ne manquent pas : fichier de config avec informations sensibles ou unique à un environnement précis, fichiers propres à un OS qui n'apportent rien au projet (hello les *.DS_store* et *Thumbs.db*), mais aussi fichiers de build (on ne voudra stocker que les fichiers sources). Bref, les exemples sont multiples.

Il existe un fichier de configuration dédié à cet effet : le

`.gitignore`. Ce fichier se trouve n'importe où dans le répertoire d'un dépôt Git et s'appliquera au répertoire courant et à tous ses sous-répertoires.

Tous les fichiers et dossiers listés dans le `.gitignore` seront tout bonnement ignorés par Git. La syntaxe est simple :

- un fichier/dossier par ligne,
- toute ligne commençant par `#` est ignorée (c'est un commentaire),
- les *patterns* `*` et `?`, ainsi que la notation de plage `[a-zA-Z]` sont pris en compte de manière classique,
- toute ligne commençant par `!` réfute une règle précédente (`*.min.js` ignore tous les fichiers ayant `.min.js` comme sufixe ; `!vendors.min.js` permet tout de même d'inclure `vendors.min.js`).

Enfin, au delà du `.gitignore` par projet, il est possible d'en créer directement au niveau de la configuration globale. Cela permet par exemple d'ignorer des fichiers et répertoires plus en lien avec le machine d'un développeur que du projet en lui-même. Il s'agit de la configuration `core.excludesfile`, on lui passe alors le chemin absolu du fichier – souvent nommé `.gitignore_global` placé dans le répertoire de l'utilisateur (`~/home`).

Suivi des répertoires vides

À ce stade, nul n'ignore le `.gitignore`. Mais saviez-vous que Git ne stocke pas les répertoires vides ? Pour palier à cela, lorsqu'un répertoire possède une importance mais est vide dans le répo du projet – `logs/` ou `config/` sont assez courants – on peut placer un fichier vide afin de forcer Git à prendre le répertoire en compte.

Ce fichier vide est par convention nommé `.gitkeep`. Il n'a aucune fonction particulière. Pour Git, c'est un fichier comme un autre. Cependant, il est là et indique par son nom que sa seule fonction est de permettre l'indexation de son répertoire parent.

Afficher l'historique des commits

Cette commande est essentielle au flux de travail avec Git. Quels sont les derniers commits ? Quels en sont les auteurs ? Quand cela a-t-il été fait ?

```
# Exemple du code de ce site
git log
commit 50fce567e20beb8ed0b48562082e4db2c646e327
Author: Buzut <buzut@email.com>
Date:   Wed Sep 11 22:48:38 2019 +0200

    🚧 reorganize css for improved clarity

commit b29c78cb832d69970b79179f6e31c6deab4bcbe8
Author: Buzut <buzut@email.com>
Date:   Wed Sep 11 20:49:53 2019 +0200

    💄 avoid using background shorthand

commit 61c115d9f2d9e109c435a89e0e04f1aaa4d95650
Author: Buzut <buzut@email.com>
Date:   Tue Sep 10 18:55:57 2019 +0200

    💄 replace a color value by a variable

commit 8b5c0e02231c1a2c1ad05a37c6a0a87dbbb825e5
Author: Buzut <buzut@email.com>
Date:   Sun Aug 11 17:31:47 2019 +0200

    🌟 add pages into search (for courses)

commit 36c61b3d91164723bdf13854799da2dc72277aa8
Author: Buzut <buzut@email.com>
```

```
Date:   Sun Aug 11 17:31:09 2019 +0200

    🌟 adjust layout with courses for mobile
```

Par défaut, `git log` énumère les commits en ordre antéchronologique avec, pour chaque commit, sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur et la date et le message du commit.

Par ailleurs, il est assez courant de ne vouloir afficher que les modifications relatives à un fichier. Pour ce faire, on passe le chemin du fichier en paramètre de la commande.

Enfin, une option assez intéressante consiste à demander à Git d'afficher les modifications apportées par le commit directement dans le log, il s'agit de l'option `-p` – `--patch` dans sa version longue.

```
# On affiche ici l'historique du header de ce si
# avec les hashs courts, des dates relatives et
git log --abbrev-commit --date=relative -p theme
commit 50fce56
Author: Buzut <buzut@email.com>
Date:   3 months ago

    🚧 reorganize css for improved clarity
```

```
diff --git a/themes/buzut/source/styles/header.l
index dc2e379..6c204f6 100644
--- a/themes/buzut/source/styles/header.less
+++ b/themes/buzut/source/styles/header.less
@@ -1,4 +1,4 @@
-.header {
+.main-header {
     display: block;
     height: 378px;
     padding: 0;

commit b29c78c
Author: Buzut <buzut@email.com>
Date:   3 months ago

    🌟 avoid using background shorthand

diff --git a/themes/buzut/source/styles/header.l
index fc5c815..dc2e379 100644
--- a/themes/buzut/source/styles/header.less
+++ b/themes/buzut/source/styles/header.less
@@ -134,7 +134,7 @@ map {
         overflow-y: auto;
         font-size: .7em;
         line-height: 1.4em;
-        background: @white;
+        background-color: @white;
         border: 1px solid @lightGray;
```

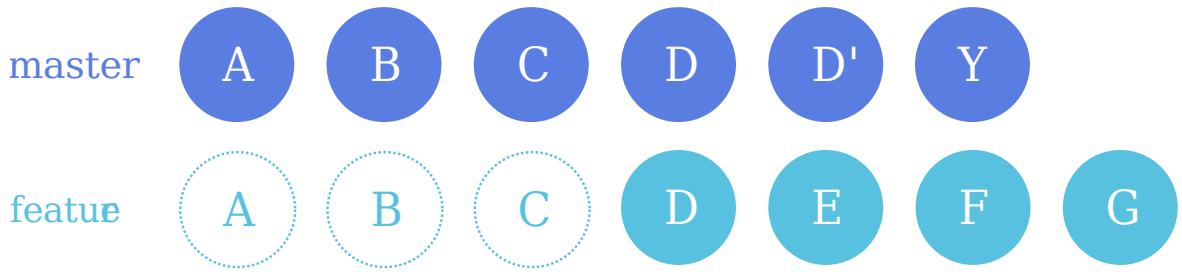
```
          border-top: none;
          border-radius: 0 0 3px 3px;
@@ -212,7 +212,7 @@ map {
                      height: 2px;
                      visibility: hidden;
                      content: '';
-                     background: #eb5055;
+                     background-color: #eb5055;
                      transition: all .1s ease;
                      transform: scaleX(0);
                  }
```

Bien entendu, de nombreuses autres options existent tant l'historique est riche. Nous l'explorerons plus en détails dans le chapitre qui lui est consacré.

Utiliser les branches

Les branches constituent une fonctionalité centrale de Git et sont un réel atout pour les développeurs. Dans la gestion de version, deux branches se comportent comme deux univers parallèles. Lors de la création de la branche, celle-ci est dupliquée depuis la branche source. Elles sont en tous points identiques.

En revanche, à partir de cet instant, leurs existances sont indépendantes. Les modifications de la branche A n'impactent pas la branche B et vice-versa.



Dans l'exemple ci-dessus, nous n'avons au début qu'une seule branche, la *master*, sur laquelle sont créés les commits A, B, C et D. La branche *feature* est créé depuis la *master* et possède donc exactement le même historique. Cependant, toute modification apportée à partir de cet instant à l'une des branches n'a plus aucune incidence sur l'autre.

Ce mécanisme permet par exemple de développer une nouvelle fonctionalité sur une branche dédiée sans risquer d'altérer le code de la branche principale, qui reste stable à tout moment. Les cas d'usage sont multiples et dépendent beaucoup du type de projet, du nombre de collaborateurs etc.

Voyons donc les principales commandes utiles pour travailler avec les branches. Nous pouvons utiliser le dépôt précédemment initialisé.

Lister et créer des branches

```
# Lister les branches
git branch
* master

# Créer une branche
git branch <newbranch>

# On liste de nouveau les branches
git branch
* master
  nouvelle_branche
```

Vous notez que l'étoile à gauche du nom de la branche indique sur quelle branche on se trouve présentement. Par ailleurs, par défaut, lors de l'initialisation du dépôt, Git créé la branche *master*. Elle n'a aucune propriété particulière et peut même être supprimée par la suite. C'est juste la branche par défaut, par conséquent, la plupart des projets possèdent une *master*.

Tout projet Git doit impérativement posséder au moins une branche, sans quoi le code n'a nulle part où être. Donc si vous désirez supprimer *master*, il faut créer une nouvelle branche au préalable.

Changer de branche

Pour changer de branche, il faut impérativement être dans un état propre, c'est à dire que le répertoire de travail ne peut contenir aucun fichier dans un état modifié. Si c'est le cas, il faut d'abord commiter les changements pour pouvoir changer de branche.

Étant donné que le changement de branche modifie le répertoire de travail, tout changement non commité mènerait à la perte des modifications non sauvegardées.

Le changement de branche s'effectue avec la commande `switch`. Cette commande sert à modifier l'état du répertoire de travail, c'est exactement ce en quoi consiste le changement de branche !

```
# Passage de la branche master à nouvelle_branch
git switch nouvelle_branche
Switched to branch 'nouvelle_branche'
```

Vous pouvez effectuer des modifications sur n'importe quel fichier, ajouter ou supprimer des fichiers. Une fois commités, si vous retournez sur la branche *master*, vous retrouverez votre espace de travail comme vous l'aviez laissé.

`switch` et `checkout`

Historiquement, c'est avec la commande `checkout` que l'on change de branche. Cependant, tout comme `reset`, cette commande possède de très nombreux usages qui changent selon le contexte. Cela semait la confusion chez bon nombre d'utilisateurs de Git.

Ainsi, depuis la version 2.23, Git propose l'usage de `switch` au lieu de `checkout` pour le changement de branche. La commande `checkout` reste cependant bien présente et tout à fait fonctionnelle.

Fusionner deux branches

Il arrive un moment où l'on souhaite que les changements effectués dans une branche intègrent ou ré-intègrent une autre branche. Cela fait partie du flux de travail classique de Git. La difficulté de cette opération est variable. Si les deux branches ont subi des modifications aux mêmes endroits, Git ne peut pas deviner quels changements doivent être gardés et lesquels sont à jeter.

Il y a alors des conflits et nous devons les résoudre manuellement avant d'indiquer à Git de fusionner les

branches. Nous traiterons la gestion des conflits dans le chapitre dédié aux branches. Pour la fusion classique, sans conflit, il n'y a qu'une seule commande à connaître : `merge`.

```
# On s'assure d'abord de se placer sur la branch
git switch master

# Puis on demande à Git de fusionner les modific
git merge <newbranch>
Updating 872ff95..8410530
Fast-forward
 newfile | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile
```

Git nous indique les modifications apportées par cette fusion ainsi que la stratégie qu'il a utilisé. Il s'agit ici du *fast-forward*, c'est à dire que le branche fusionnée ne comportait que des commits supplémentaires par rapport à la branche cible. Il lui a donc suffit d'ajouter ces commits à notre branche cible.

Supprimer une branche

Dans sa forme basique, la suppression est très simple,

elle se résume à utiliser l'option `-d`.

```
git branch -d <branch>
```

Il ne faut évidemment pas être sur la branche en question lors de sa suppression, auquel cas Git retournera une erreur. Nous verrons d'autres options utiles dans le chapitre dédié aux branches.

Travailler avec les dépôts distants

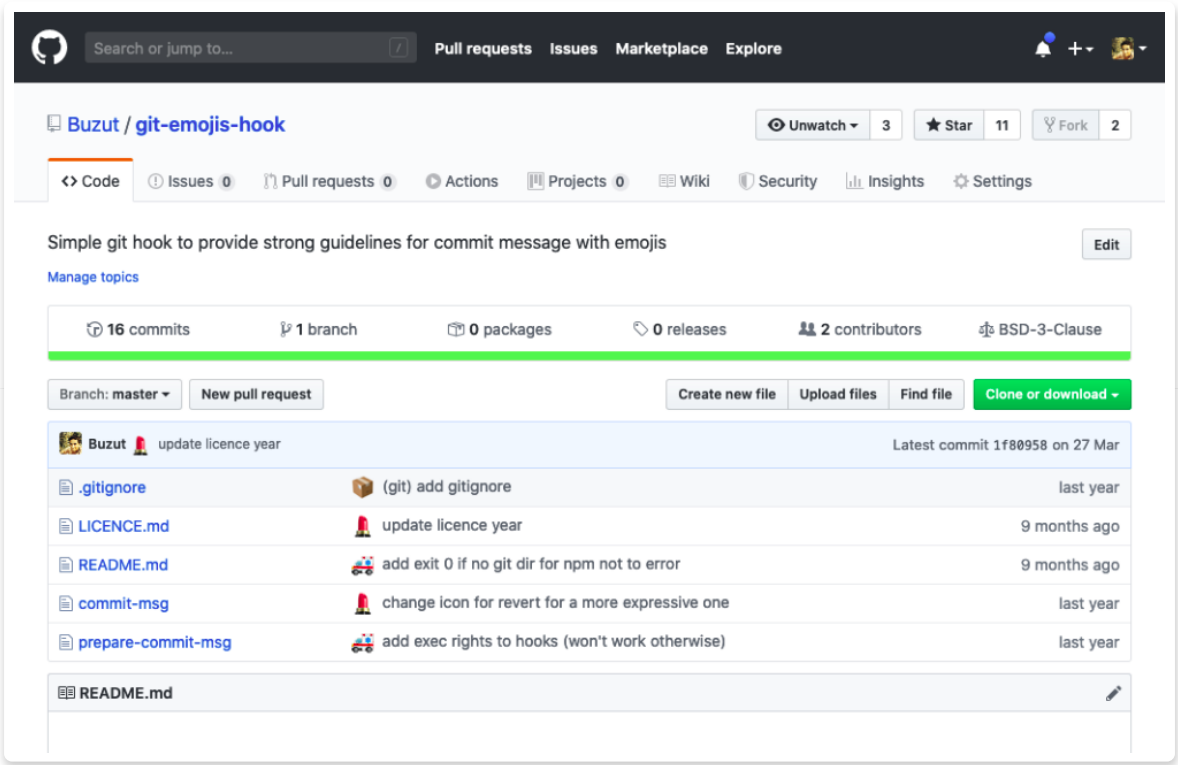
Les dépôts distants occupent une place centrale dans l'usage collaboratif de Git. Aussi, leur usage est très répandu et présentent de très nombreuses possibilités. D'ailleurs, le prochain chapitre leur est entièrement consacré. Nous allons ici aborder les actions les plus courantes.

Cloner depuis un dépôt distant

L'action la plus couramment effectuée avec les dépôts distants consite à récupérer, ou cloner, un dépôt. Pour ce

faire, vous n'avez besoin que des accès en écriture – disponibles pour tous dans le cas de projets publics – et de l'adresse du projet.

En exemple, nous allons cloner le dépôt du projet Git emojis hook. Vous l'avez peut-être constaté dans les exemples précédents, mes commits sont toujours agrémentés d'Emojis. C'est ce projet qui permet d'en automatiser l'insertion. Il est publiquement disponible sur GitHub.



Page d'accueil d'un projet sur GitHub

Toutes les palteformes de gestion de projets Git possèdent une fonctionalité pour cloner les projets. Sur les projets publics comme celui-ci, vous n'avez pas besoin

d'être identifié. En revanche, sur un projet privé, dont l'accès est restreint, il faudra montrer pate blanche.

Vous choisissez le protocole de clonage qui vous convient le mieux : HTTPS ou SSH, vous copiez le lien puis vous importez le repo, cela se fait au moyen de la commande `clone` de Git. Le dépôt est alors cloné dans le répertoire courant.

```
git clone https://github.com/Buzut/git-emojis-ho

# Vérification du clonage
ls git-emojis-hook
LICENCE.md        README.md        commit-msg
```

Télécharger n'est pas cloner

Attention, bien que la plupart des plateformes permettent le téléchargement, il ne s'agit pas de clonage. Si vous cliquez sur "Download ZIP" dans GitHub, vous téléchargez le dernier snapshot du projet mais **vous n'avez pas de répertoire `.git`** et donc aucune des informations ni fonctionalités apportées par Git.

Pusher ses modifications

Une fois que nous avons ajoutés des commits au projet, on peut les pousser sur le dépôt distant afin de rendre nos modifications accessibles aux autres. Si vous avez les droits en écriture sur le projet, le partage de vos modifications se résumera à une commande.

```
git push
```

C'est tout ! Le push envoie les modifications relatives à la branche en cours, donc si vous voulez pusher les modifications de la branche `feature`, il faut être sur cette dernière.

Nous avons vu dans ce chapitre 80% des commandes dont nous nous servons constamment avec Git. Dans le chapitre suivant, nous allons explorer en détails et découvrir les usages avancés des dépôts distants.

Commentaires

Esperancia dit – September 26, 2017

Merci pour ce tutoriel. mais ce qui est rarement dit dans les tutoriels de git c'est comment revenir à un commit spécifique sur la branche courante. Exemple: je veux revenir au commit 1a48dbd sur la branche master. Evidemment j'ai eu plusieurs autres commit récents après

celui là. Alors comment suis-je censée y arriver? Pour ma part j'ai fait : git checkout 1a48dbd . Puis: git commit -m "Revenir sur mon commit 1a48dbd qui fixe le bug de ..." Je crois que ça a bien marché. et que c'est ce qu'il faut. Donc essayez et dites moi!

Répondre

> Buzut dit – September 26, 2017
>
> Hello,
>
> C'est pourtant ce que j'explique dans Voyager dans le temps si je comprends bien ta question ;)
>
> Répondre

David dit – October 20, 2017

Merci beaucoup ! :)

Répondre

Rejoignez la discussion !

Ce que vous voulez partager avec nous

Votre message

Vous pouvez utiliser Markdown pour les liens [ancre de lien](url), la mise en *italique* et en **gras**. Enfin pour le code, vous pouvez utiliser la syntaxe `inline` et la syntaxe bloc

```
ceci est un bloc
de code
```

Votre prénom ou pseudo

Pseudo

Votre email (un email vous sera envoyé pour valider le commentaire)

Email

Soumettre le commentaire

Quentin Busuttil – Licence Creative Commons BY-NC-ND

BLOG (https://karac.ch/blog)

MAÎTRISER L'ESSENTIEL DE GIT EN QUELQUES MINUTES

(https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fkarac.ch%2Fblog%2Fmaitriser-essentiel-de-git-en-quelques-minutes)
(https://twitter.com/share?text=Maîtriser l'essentiel de Git en quelques minutes&url=https%3A%2F%2Fkarac.ch%2Fblog%2Fmaitriser-essentiel-de-git-en-quelques-minutes)

TECH WEB (https://karac.ch/blog/categorie/tech-web)

30.05.2013 (https://karac.ch/blog/maitriser-essentiel-de-git-en-quelques-minutes#)

Git est un logiciel de gestion de versions à destination des développeurs. S'il est envisageable de se passer d'un tel logiciel, celui-ci pourrait bien vous tirer de quelques misères, particulièrement si vous travaillez en équipe.

Au cours de cet article, nous allons voir comment utiliser Git à l'aide de scénarios pour que cela soit le plus concret possible.

Ce que Git va faire pour vous :

- stocker le code de votre projet sur un serveur en ligne et ainsi vous permettre de reprendre ce projet depuis n'importe quel ordinateur
- conserver les anciennes versions de chaque fichier du projet
- vous permettre de travailler à plusieurs dans le code du projet en étant certain de ne pas écraser le code des autres
- vous permettre de savoir qui a fait telle ou telle modification

Installation

Git est à l'origine conçu pour Linux et plus particulièrement pour une utilisation via le terminal de Linux. Il a d'ailleurs été développé par Linus Torvald, auteur du noyau Linux. Il peut cependant être installé sur Windows via un système d'émulation des commandes Unix, mais aussi sur Mac via un installateur, ou en passant par le gestionnaire de paquets "Homebrew".

Installation sur Linux

Je pars du principe que vous êtes sur une distribution Ubuntu, Debian, ou du moins que vous avez le gestionnaire de paquets "apt-get". Cependant, vous pouvez très bien l'installer avec n'importe quelle autre configuration.

Ouvrez un terminal et entrez ces commandes :

BLOG (https://karac.ch/blog)


(https://karac.ch/blog/auteur/joel-buchs)

Joël Bu
(https:/
/blog/a
buchs)
"Many
all of th
withou
it is not
after."
Thorea
joel@k
(mailto

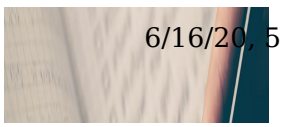
(/)

NEWSLETTER

Abonne-toi à notre Newsletter !

Adresse OK

PEUT VOUS INTÉRESSER



© 2020 karac.ch (https://karac.ch) - info@karac.ch (mailto:info@karac.ch) -
BLOG (https://karac.ch/blog) LIKE @HTTPS://WWW.FACEBOOK.COM/KARACWEB/).

Mini-tutoriel sur Git

Ce tutoriel explique la base pour savoir se servir de Git. Cela vous permettra de cloner les projets présent sur ce site, mais étant donnés qu'ils sont en lecture seule, les opérations de mise à jour du dépôt inspyration.org ne seront pas réalisables.

Configurer GIT

Pour faire des validation qui soient lisibles, il faut que chacun déclare son nom, pour que chaque validation soit associé à la personne qui l'a conduite.

```
$ git config --global user.name "Prénom NOM"
$ git config --global user.email "adresse.courriel@serveur.com"
```

Transformer un répertoire en un dépôt

```
$ git init
```

Ceci crée un dépôt vide, ne contenant rien, quelque soit l'état du répertoire. Le dépôt est géré par l'entremise du répertoire **.git** que la commande vient de créer.

On peut alors rajouter une description succincte de ce que contient le dépôt :

```
$ vim .git/description
```

Rajouter un fichier ou un répertoire au dépôt

Le dépôt dispose d'un index des fichiers qu'il contient. Pour rajouter un fichier ou un répertoire, il suffit de faire ceci :

```
$ git add path
```

Si le chemin est un répertoire, le répertoire lui-même ainsi que tout son contenu (récursivement) est alors rajouté dans l'index.

Si, à l'intérieur d'un répertoire, on ne souhaite pas rajouter un fichier ou un répertoire, il faut alors créer un fichier **.gitignore** et y rajouter la liste de ce que l'on ne souhaite pas voir :

```
!.gitignore                          (ne pas ignorer le .gitignore)
.~lock.*                             (ignorer les fichiers temporaires libreoffice)
.buildpath                           (pour Eclipse)
.project
.settings
file                                 (exemple d'un nom de fichier)
```

```
folder                                  (exemple d'un nom de répertoire)
```

Puis, il faut rajouter le fichier **.gitignore** dans l'index des fichiers du dépôt.

```
$ git add path/to/.gitignore
```

Valider un ou plusieurs fichiers / répertoires

```
$ git commit file1 file2 folder1 folder2
```

La validation est un processus local. Les différentes versions des fichiers sont conservés localement.

On peut utiliser l'option -m pour rajouter le message de validation directement. Sans cela, un éditeur va s'ouvrir pour recueillir ce message. S'il est vide, le processus de validation est abandonné.

Valider tous les fichiers indexés

```
$ git commit -a
```

Attention, les fichiers non indexés (non rajoutés dans le dépôt avec la commande **git add**) ne seront pas validés.

La validation est un processus local. Les différentes versions des fichiers sont conservés localement.

Lorsque l'on utilise l'option **-a**, il est déconseillé d'utiliser l'option **-m**, ne serait-ce que pour vérifier la liste des fichiers réellement validés avant validation.

Vérifier le statut du dépôt

```
$ git status
```

Cela indique la liste des fichiers modifiés et la liste des fichiers ne faisant partie ni du dépôt, ni de la liste des fichiers à ignorer.

Voir le différentiel entre les fichiers actuels et la dernière validation

```
$ git diff
```

Il faut alors savoir interpréter un fichier diff pour visualiser les différences ou utiliser un lecteur de fichiers diff.

Avant toute validation, il est possible de réaliser un patch de la validation en faisant :

```
$ git diff > path/to/patch
```

On peut également le faire après coup. A tout moment, on peut faire un différentiel entre deux validations distinctes.

Supprimer toutes les modifications depuis le dépôt

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

Le dépôt revient à son état après la dernière validation, mais cette commande ne touche pas les fichiers ne faisant pas partie du dépôt. Si un tel fichier a été supprimé, il est définitivement perdu, s'il a été modifié, les modifications perdurent et s'il a été créé, il reste toujours là.

Cloner un dépôt

```
$ git clone url_depot
```

Voici un exemple concret pour un dépôt en lecture écriture :

```
$ git clone git+ssh://inspyration.org/nom_depot.git
```

Voici un exemple concret pour un dépôt en lecture seule :

```
$ git clone http://inspyration.org/nom_depot.git
```

On peut rajouter un argument supplémentaire qui serait le nom que prendrait le dépôt local.

Rajouter un serveur distant

La validation de son code se fait toujours sur le dépôt local. Il est possible de transmettre ses validations à un serveur distant et de récupérer les validations faites par les autres sur son propre serveur.

La première étape consiste à créer le lien entre son dépôt et celui disponible sur le serveur distant :

```
$ git remote add nom_distant url
```

Voici un exemple :

```
$ git remote add origin git+ssh://inspyration.org/nom_depot.git
```

Bien sûr, il faut que le dépôt ait été créé et initialisé à vide coté serveur.

Pousser ses validations sur un serveur distant :

```
$ git push origin master
```

Ici, **origin** est le nom du serveur distant (voir paragraphe précédent) et master est le nom de la branche courante.

Rapatrier les validations sur le serveur vers son dépôt local

```
$ git pull origin master
```

Lister les branches disponibles sur le dépôt local

```
$ git branch
```

Attention, ce n'est pas parce qu'une branche existe sur un dépôt local qu'elle existe sur un dépôt distant et vice-versa. Chaque branche doit être poussée ou rapatriée indépendamment.

Lister les branches disponibles sur les dépôts local et distants

```
$ git branch -a
```

Pour les dépôts distants, on voit apparaître **remotes/nom_distant /nom_branche_distante**, comme, par exemple **remotes/origin/master**.

Créer une nouvelle branche

```
$ git checkout -b nom_nouvelle_branche
```

Cette commande crée la branche à partir de là où l'on se trouve et bascule vers elle.

Il faut que le dépôt soit dans un état propre et stable avant de créer une nouvelle branche.

Basculer vers une branche

```
$ git checkout nom_branche
```

Là encore, avant de basculer, il faut s'assurer d'être dans un état propre et stable.

Merger une branche vers la branche courante

En étant dans la branche courante :

```
$ git merge nom_branche
```

Résoudre un conflit de merge

```
$ git merge nom_branche
$ git status
$ git mergetool
$ git commit -a
```

-

Mots-clés associés : tutoriel git, Git