**La gestion de version**

Sommaire :

* **À propos de la gestion de version**
* Type de système de gestion de fichiers
  + **Les systèmes de gestion de version locaux**
  + **Les systèmes de gestion de version centralisés**
  + **Les systèmes de gestion de version distribués**
  + **Exemple de logiciels**
* **Git** 
  + **Historique**
  + Rudiment de git :
  + Base de Git
  + **Les branches sur git** 
    - **Ce qu'est une branche**
    - **Differente type de branches**

**. !! Find a title to this section (like wrapping ….)**

* **GitHub** 
  + **GitHub c’est Quoi** ?
  + **Statistiques**
  + **Codage social**
* **Comment mettre tout ça en œuvre.**

**À propos de la gestion de version**

Qu'est-ce que la gestion de version et pourquoi devriez-vous vous en soucier ? Un gestionnaire de version est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment. Bien que la gestion de version est utilisé généralement pour les codes sources, mais en réalité on puisse l'utiliser avec pratiquement tous les types de fichiers d'un ordinateur*.*

Si vous êtes un dessinateur, développeur web ou développeur logiciel, et que vous voulez conserver toutes les versions d'une image ou d'un code source, un système de gestion de version (VCS en anglais pour Version Control System) est un outil qu'il est très sage d'utiliser. Il vous permet de ramener un fichier à un état précédent, de ramener le projet complet à un état précédent, de visualiser les changements au cours du temps, de voir qui a modifié quelque chose qui pourrait causer un problème, qui a introduit un problème et quand, et plus encore. Utiliser un VCS signifie aussi généralement que si vous vous trompez ou que vous perdez des fichiers, vous pouvez facilement revenir à un état stable. De plus, vous obtenez tous ces avantages avec peu de travail additionnel.

**Les systèmes de gestion de version locaux**

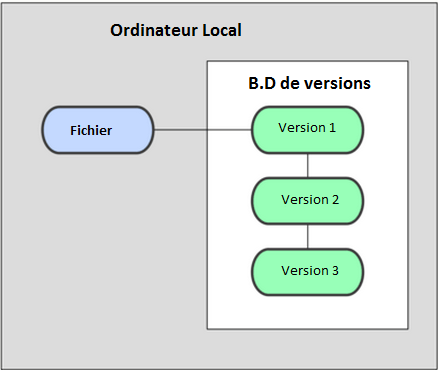
La méthode courante pour la gestion de version est généralement de recopier les fichiers dans un autre répertoire qui peut-être avec un nom incluant la date dans le meilleur des cas. Cette méthode est la plus courante parce que c'est la plus simple, mais c'est aussi la moins fiable. Il est facile d'oublier le répertoire dans lequel vous êtes et d'écrire accidentellement dans le mauvais fichier ou d'écraser des fichiers que vous vouliez conserver. Pour traiter ce problème, les programmeurs ont développé il y a longtemps des VCS locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier (voir figure 1-1).

Figure 1-1 : Modélisation d’un système de gestion de version local

Un des systèmes les plus populaires était RCS, qui est encore distribué avec de nombreux systèmes d'exploitation aujourd'hui. Même le système d'exploitation populaire Mac OS X inclut le programme RCS lorsqu'on installe les outils de développement logiciel. Cet outil fonctionne en conservant des ensembles de patchs (c'est-à-dire la différence entre les fichiers) d'une version à l'autre dans un format spécial sur disque ; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

**Les systèmes de gestion de version centralisés**

Le problème majeur que les gens rencontrent est qu'ils ont besoin de collaborer avec des développeurs sur d'autres ordinateurs. Pour traiter ce problème, les systèmes de gestion de version centralisés (CVCS en anglais pour Centralized Version Control Systems) furent développés. Ces systèmes tels que CVS, Subversion, et Perforce, mettent en place un serveur central qui contient tous les fichiers sous gestion de version, et des clients qui peuvent extraire les fichiers de ce dépôt central. Pendant de nombreuses années, cela a été le standard pour la gestion de version (voir figure 1-2).

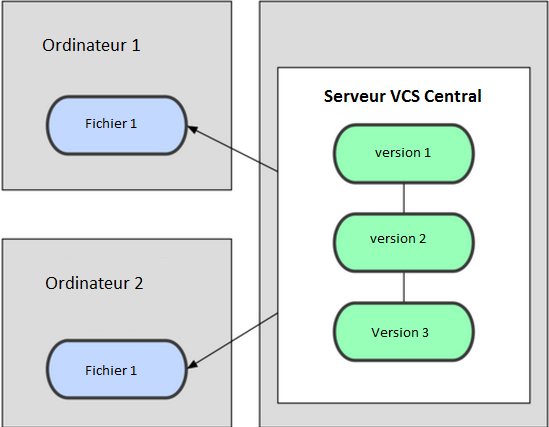


Figure 1-2 : Modélisation d’un système de gestion de version centraliser

Ce schéma offre de nombreux avantages par rapport à la gestion de version locale. Par exemple, chacun sait jusqu'à un certain point ce que tous les autres sont en train de faire sur le projet. Les administrateurs ont un contrôle fin des permissions et il est beaucoup plus facile d'administrer un CVCS que de gérer des bases de données locales.

Cependant ce système a aussi de nombreux défauts. Le plus visible est le point unique de panne que le serveur centralisé représente. Si ce serveur est en panne pendant une heure, alors durant cette heure, aucun client ne peut collaborer ou enregistrer les modifications issues de son travail. Si le disque dur du serveur central se corrompt, et s'il n'y a pas eu de sauvegarde, vous perdez absolument tout de l'historique d'un projet en dehors des sauvegardes locales que les gens auraient pu réaliser sur leur machines locales. Les systèmes de gestion de version locaux souffrent du même problème — dès qu'on a tout l'historique d'un projet sauvegardé à un endroit unique, on prend le risque de tout perdre.

**Les systèmes de gestion de version distribués**

C'est à ce moment que les systèmes de gestion de version distribués entrent en jeu (DVCS en anglais pour Distributed Version Control Systems). Dans un DVCS (tel que Git, Mercurial, Bazaar ou Darcs), les clients n'extraient plus seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données (voir figure 1-3).

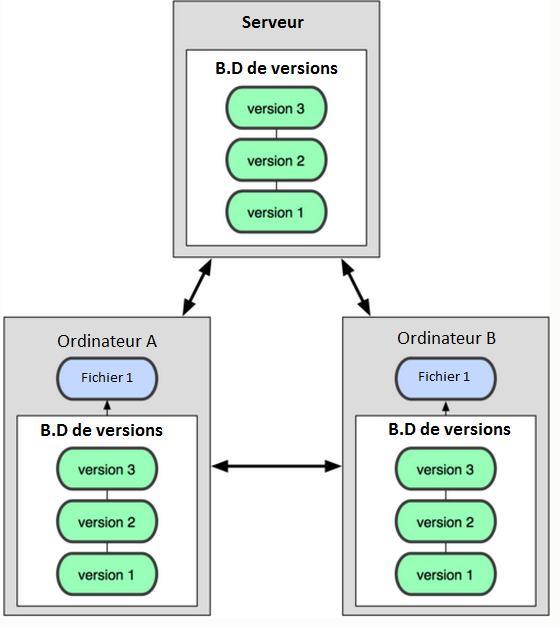


Figure 1-3 : Modélisation d’un système de gestion de version distribuée

De plus, un grand nombre de ces systèmes gère particulièrement bien le fait d'avoir plusieurs dépôts avec lesquels travailler, vous permettant de collaborer avec différents groupes de personnes de manières différentes simultanément dans le même projet. Cela permet la mise en place de différentes chaînes de traitement qui ne sont pas réalisables avec les systèmes centralisés, tels que les modèles hiérarchiques.

**Quelque logiciels de gestion de version :**

|  |  |  |
| --- | --- | --- |
|  | **Logiciels libres** | **Logiciels propriétaires** |
| **Gestion Locale** | GNU RCS (1892) | PVCS (1985), SCCS(1972) |
| **Client-Serveur** | CVS(1990), CVSNT(1992), SVN(2000) | CMVC(1994), sourceanywhere(2003) |
| **Décentralisé** | GNU arch(2001), Fossil(2007), Git(2005) | BitKeeper(1998), plastic SCM(2007) |

Git

Historique.

Git est né avec une dose de destruction créative et de controverse houleuse. Le noyau Linux est un projet libre de grande envergure. Pour la plus grande partie de sa vie (1991–2002), les modifications étaient transmises sous forme de patchs et d'archives de fichiers. En 2002, le projet du noyau Linux commença à utiliser un DVCS propriétaire appelé BitKeeper.

En 2005, les relations entre la communauté développant le noyau Linux et la société en charge du développement de BitKeeper furent rompues, et le statut de gratuité de l'outil fut révoqué. Cela poussa la communauté du développement de Linux et plus particulièrement Linus Torvalds, le créateur de Linux à développer son propre outil en se basant sur les leçons apprises lors de l'utilisation de BitKeeper.

Depuis sa naissance en 2005, Git a évolué et mûri pour être facile à utiliser tout en conservant ses qualités initiales. Il est incroyablement rapide, il est très efficace pour de grands projets et il a un incroyable système de branches pour des développements non linéaires .

Rudiment de git :

### [Des instantanés, pas des différences](http://git-scm.com/book/fr/D%C3%A9marrage-rapide-Rudiments-de-Git#Des-instantanés,-pas-des-différences)

La différence majeure entre Git et les autres VCS (Subversion et autres) réside dans la manière dont Git considère les données. Au niveau conceptuel, la plupart des autres VCS gèrent l'information comme une liste de modifications de fichiers. Ces systèmes (CVS, Subversion, Perforce, Bazaar et autres) considèrent l'information qu'ils gèrent comme une liste de fichiers et les modifications effectuées sur chaque fichier dans le temps, comme illustré en figure 1-4.

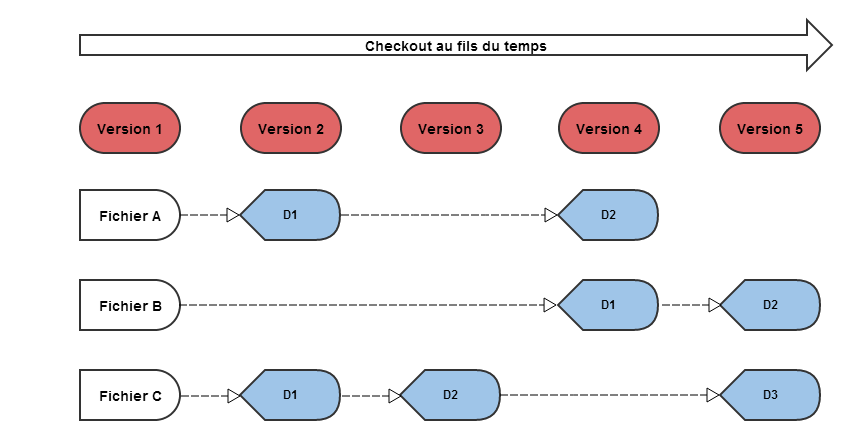


Figure 1-4 : Méthode d`enregistrement pour d’autre logiciel de gestion de version.

Git ne gère pas et ne stocke pas les informations de cette manière. À la place, Git pense ses données plus comme un instantané d'un mini système de fichiers. À chaque fois que vous validez ou enregistrez l'état du projet dans Git, il prend effectivement un instantané du contenu de votre espace de travail à ce moment et enregistre une référence à cet instantané. Pour être efficace, si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qui n'a pas été modifié. Git pense ses données plus à la manière de la figure 1-5.

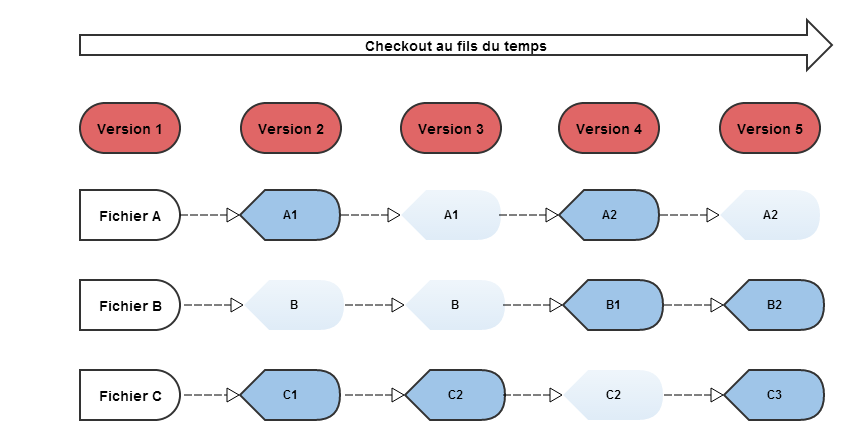


Figure 1-5 : Git stocke les données comme des instantanés du projet au cours du temps.

C'est une distinction importante entre Git et quasiment tous les autres VCS. Git a reconsidéré quasiment tous les aspects de la gestion de version que la plupart des autres systèmes ont copiés des générations précédentes. Cela fait quasiment de Git un mini système de fichiers avec des outils incroyablement puissants construits dessus, plutôt qu'un simple VCS.

### [Presque toutes les opérations sont locales](http://git-scm.com/book/fr/D%C3%A9marrage-rapide-Rudiments-de-Git#Presque-toutes-les-opérations-sont-locales)

La plupart des opérations de Git ne nécessite que des fichiers et ressources locaux — généralement aucune information venant d'un autre ordinateur du réseau n'est nécessaire. Si vous êtes habitué à un CVCS où toutes les opérations sont ralenties par la latence des échanges réseau, cet aspect de Git vous fera penser que les dieux de la vitesse ont octroyé leurs pouvoirs à Git. Comme vous disposez de l'historique complet du projet localement sur votre disque dur, la plupart des opérations semblent instantanées.

Par exemple, pour parcourir l'historique d'un projet, Git n'a pas besoin d'aller le chercher sur un serveur pour vous l'afficher ; il n'a qu'à simplement le lire directement dans votre base de données locale. Cela signifie que vous avez quasi-instantanément accès à l'historique du projet. Si vous souhaitez connaître les modifications introduites entre la version actuelle d'un fichier et son état un mois auparavant, Git peut rechercher l'état du fichier un mois auparavant et réaliser le calcul de différence, au lieu d'avoir à demander cette différence à un serveur ou à devoir récupérer l'ancienne version sur le serveur pour calculer la différence localement.

Cela signifie aussi qu'il y a très peu de choses que vous ne puissiez réaliser si vous n'êtes pas connecté ou hors VPN. Si vous voyagez en train ou en avion et voulez avancer votre travail, vous pouvez continuer à gérer vos versions sans soucis en attendant de pouvoir de nouveau vous connecter pour partager votre travail. Si vous êtes chez vous et ne pouvez avoir une liaison VPN avec votre entreprise, vous pouvez tout de même travailler. Pour de nombreux autres systèmes, faire de même est impossible ou au mieux très contraignant. Avec Perforce par exemple, vous ne pouvez pas faire grand-chose tant que vous n'êtes pas connecté au serveur. Avec Subversion ou CVS, vous pouvez éditer les fichiers, mais vous ne pourrez pas soumettre des modifications à votre base de données (car celle-ci est sur le serveur non accessible). Cela peut sembler peu important a priori, mais vous seriez étonné de découvrir quelle grande différence cela peut constituer à l'usage.

### [Git gère l'intégrité](http://git-scm.com/book/fr/D%C3%A9marrage-rapide-Rudiments-de-Git#Git-gère-l'intégrité)

Dans Git, tout est vérifié par une somme de contrôle avant d'être stocké et par la suite cette somme de contrôle, signature unique, sert de référence. Cela signifie qu'il est impossible de modifier le contenu d'un fichier ou d'un répertoire sans que Git ne s'en aperçoive. Cette fonctionnalité est ancrée dans les fondations de Git et fait partie intégrante de sa philosophie. Vous ne pouvez pas perdre des données en cours de transfert ou corrompre un fichier sans que Git ne puisse le détecter.

Le mécanisme que Git utilise pour réaliser les sommes de contrôle est appelé une empreinte SHA-1. C'est une chaîne de caractères composée de 40 caractères hexadécimaux (de '0' à '9' et de 'a' à 'f') calculée en fonction du contenu du fichier ou de la structure du répertoire considéré. Une empreinte SHA-1 ressemble à ceci :

24b9da6552252987aa493b52f8696cd6d3b00373

Vous trouverez ces valeurs à peu près partout dans Git car il les utilise pour tout. En fait, Git stocke tout non pas avec des noms de fichiers, mais dans la base de données Git indexée par ces valeurs.

### [Généralement, Git ne fait qu'ajouter des données](http://git-scm.com/book/fr/D%C3%A9marrage-rapide-Rudiments-de-Git#Généralement,-Git-ne-fait-qu'ajouter-des-données)

Quand vous réalisez des actions dans Git, la quasi-totalité d'entre elles ne font qu'ajouter des données dans la base de données de Git. Il est très difficile de faire réaliser au système des actions qui ne soient pas réversibles ou de lui faire effacer des données d'une quelconque manière. Par contre, comme dans la plupart des systèmes de gestion de version, vous pouvez perdre ou corrompre des modifications qui n'ont pas encore été entrées en base ; mais dès que vous avez validé un instantané dans Git, il est très difficile de le perdre, spécialement si en plus vous synchronisez votre base de données locale avec un dépôt distant.

Cela fait de l'usage de Git un vrai plaisir, car on peut expérimenter sans danger de casser définitivement son projet

### [Les trois états](http://git-scm.com/book/fr/D%C3%A9marrage-rapide-Rudiments-de-Git#Les-trois-états)

Git gère trois états dans lesquels les fichiers peuvent résider : validé, modifié et indexé. Validé signifie que les données sont stockées en sécurité dans votre base de données locale. Modifié signifie que vous avez modifié le fichier mais qu'il n'a pas encore été validé en base. Indexé signifie que vous avez marqué un fichier modifié dans sa version actuelle pour qu'il fasse partie du prochain instantané du projet.

Ceci nous mène aux trois sections principales d'un projet Git : le répertoire Git, le répertoire de travail et la zone d'index.

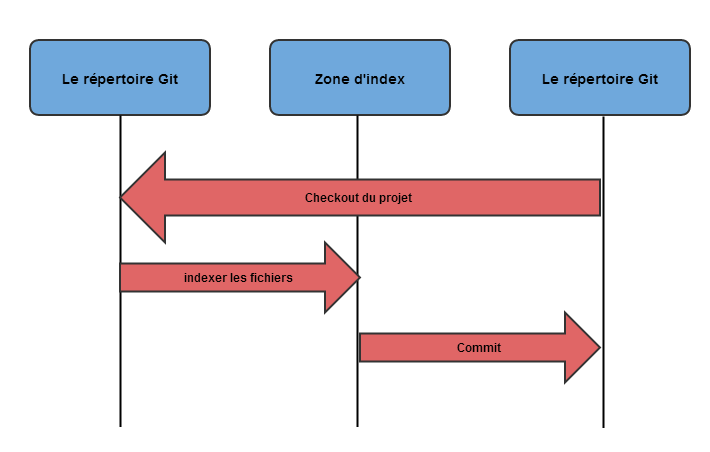


Figure 1-6 : Répertoire de travail, zone d'index et répertoire Git.

Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Le répertoire de travail est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index est un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané.

L'utilisation standard de Git se passe comme suit :

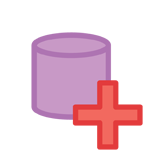
1. vous modifiez des fichiers dans votre répertoire de travail ;
2. vous indexez les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index ;
3. vous validez, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans la base de données du répertoire Git.

Si une version particulière d'un fichier est dans le répertoire Git, il est considéré comme validé. S'il est modifié mais a été ajouté dans la zone d'index, il est indexé. S'il a été modifié depuis le dernier instantané mais n'a pas été indexé, il est modifié. Dans le chapitre 2, vous en apprendrez plus sur ces états et comment vous pouvez en tirer parti ou complètement les occulter.

Base de git :

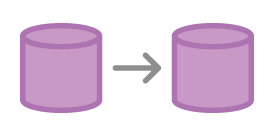
Cette partie donne un aperçu sur les commandes Git les plus importante et explique tous les outils dont vous avez besoin pour démarrer un nouveau projet, créer un dépôt Git, enregistrer des instantanés de votre projet, et consulter l'historique de votre projet sous git. Plus tard dans ce chapitre un tutorial expliquera comment utiliser toutes ces commandes.

\*Configuration d'un dépôt Git

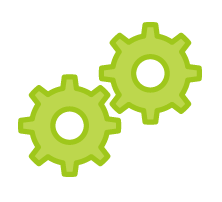
*$ git init*

La commande *git init* initialise un nouveau dépôt Git. Si vous voulez placer

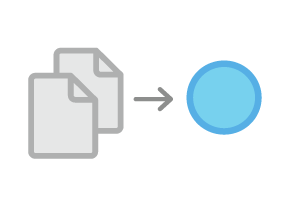
un projet sous contrôle de révision, c'est la première commande que vous devez utiliser.

*$ git clone*

La commande *git clone* crée une copie d'un dépôt Git existant. Le clonage est le moyen le plus courant pour les développeurs afin d'obtenir une copie de travail d'un référentiel central.

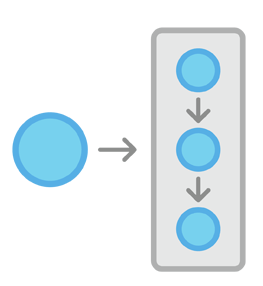
*$ git config*

La commande *git config* est un moyen pratique pour définir les options de configuration de votre installation de Git. Vous aurez généralement seulement besoin de l'utiliser immédiatement après l'installation de Git sur ​​une nouvelle machine de développement.

Enregistrement d’instantané

*$ git add*

La commande *git add* déplace les modifications du répertoire de travail de la zone d'index. Cela vous donne la possibilité de préparer un instantané avant de commiter à l'histoire officielle.



*$ git commit*

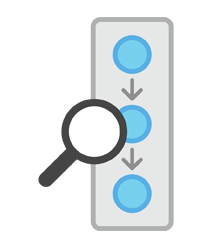
La commande *git commit* prend l’instantané indexé et le commit à l'historique du projet. Combiné avec *git add*, ils définissent le flux de travail de base pour tous les utilisateurs de Git.

L'inspection d'un dépôt Git



$ *git statut*

La commande *git status* affiche l'état du répertoire de travail et l'instantané indexé. Cette commande est très utile en collaboration avec *git add* et *git commit* pour voir exactement ce qui est inclus dans le prochain snapshot.

*$ git log*

La commande *git log* vous permet d'explorer les révisions précédentes du projet. Il offre plusieurs options de mise en forme pour l'affichage que vous pouvez voir à l’aide de la commande *git log –help.*

# Les branches avec Git

Quasiment tous les VCS ont une forme ou une autre de gestion de branche. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans se préoccuper de cette ligne principale. Dans de nombreux outils de gestion de version, cette fonctionnalité est souvent chère en ressources et nécessite de créer une nouvelle copie du répertoire de travail, ce qui peut prendre longtemps dans le cas de gros projets.

De nombreuses personnes font référence au modèle de gestion de branche de Git comme LA fonctionnalité et c'est sûrement la spécificité de Git par rapport à la communauté des gestionnaires de version. Pourquoi est-elle si spéciale ? La méthode de Git pour gérer les branches est particulièrement légère, permettant de réaliser des embranchements quasi instantanément et de basculer entre les branches généralement aussi rapidement. À la différence de nombreux autres gestionnaires de version, Git encourage à travailler avec des méthodes qui privilégient la création et la fusion de branches, jusqu'à plusieurs fois par jour. Bien comprendre et maîtriser cette fonctionnalité est un atout pour faire de Git un outil unique qui peut littéralement changer la manière de développer.

Comment ca marche :

Branches historiques :

Au lieu d'une seule branche maîtresse, Le flux de travail utilise deux branches pour enregistrer l'historique du projet. La branche master stocke l'historique de sortie officielle, et la branche Develop sert comme une branche de l'intégration des fonctionnalités.

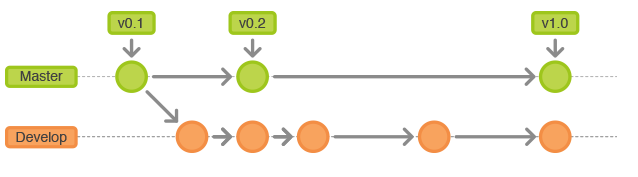


Figure 1-7 : Model de branches historiques

Branches de fonctionnalité

Chaque nouvelle fonctionnalité devrait résider dans sa propre branche, qui peut être poussé vers le répertoire central pour la sauvegarde et la collaboration. Mais, au lieu de brancher le master, les branches de fonctionnalité utilisent Develop comme leur branche mère. Quand une fonction est terminée, elle se fait fusionnée vers Develop. Les nouvelles fonctionnalités ne doivent jamais communiquer directement avec la branche mère.

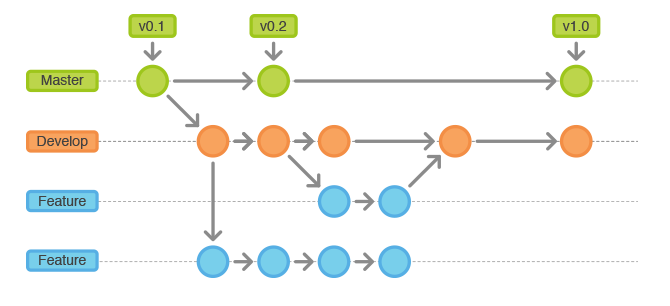


Figure 1-8 : Model de branche de fonctionnalité

**Branche de Release:**

Une fois la branche Develop acquis suffisamment de fonctionnalités pour une version ou une date de sortie prédéterminé est proche, vous débourser une branche de sortie hors de Develop. La création de cette branche commence le prochain cycle de release, donc pas de nouvelles fonctionnalités qui peuvent être ajoutées après ce point uniquement des corrections de bugs, génération de documentation. Une fois que le release est prêt à être expédié, il se fusionne avec le master et étiqueté avec un numéro de version. En outre, il doit être fusionné vers Develop, qui peut avoir évolué depuis que le release a été lancé.

En utilisant une branche dédiée à la préparation des releases, il est possible pour une équipe de raffiner la version actuelle pendant qu'une autre équipe continue à travailler sur les fonctionnalités de la prochaine version.

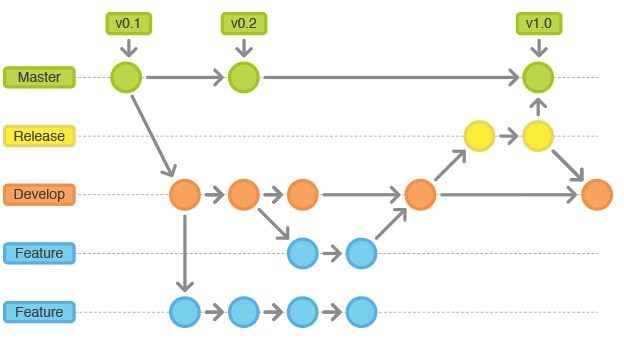


Figure 1-9: Model de branche de release.

**Branche de maintenance :**

Les branches de maintenance ou "hotfix" sont utilisés pour réparer rapidement les versions de production. Il s'agit de la seule branche qui devrait débourser directement sur le master. Dès que la correction est terminée, il devrait être fusionné à la fois avec la master et la develop (ou la branche de version en cours), et le maître doit être étiqueté avec un numéro de version mise à jour.

Avoir une ligne dédiée au développement pour corriger les bugs permet à l’équipes de développement de corriger des bugs sans interrompre le flux de travail ou d'attendre le prochain cycle de release.

 Figure 2 : Model de branches de maintenance.

Conclusion :

Pour conclure Git est un outil très puissant pour la gestion de version de n’importe quel fichier sur un ordinateur. Vu que le principe de git se base sur une base de données distribué, l’utilisation d’un serveur est indispensable. La prochaine partie sera dédier au plus gros hébergeur de projet sous git.

**GitHub**

**GitHub c’est Quoi** ?

GitHub est un service d'hébergement Web pour les projets de développement de logiciels qui utilisent le système de contrôle de révision de Git. GitHub propose deux plans payés référentiels privés, et des comptes gratuits pour les projets open source. En mai 2011, GitHub a été le site de dépôt de code le plus populaire pour les projets open source.

Le site fournit des fonctionnalités de réseau social tel que les commentaire, les fans et le graphe du réseau social pour afficher comment les développeurs travaillent sur leurs versions d'un même répertoire.

**Statistique**

GitHub a été lancé en Avril 2008.

Dans un discours au siège de Yahoo! le 24 Février 2009, les membres de l'équipe GitHub ont annoncé que durant la première année que GitHub était en ligne, il a accumulé 46.000 dépôts publics, 17.000 d'entre eux au cours du mois précédent seul. A cette époque, environ 6.200 dépôts avaient été dupliqué au moins une fois et 4,600 fusionné. Le 5 Juillet 2009, un blog Github a annoncé qu'ils ont atteint la barre des 100.000 utilisateurs.

Dans un autre discours prononcé à Yahoo! le 27 Juillet 2009, Tom Preston-Werner(1) a annoncé que les effectifs avaient augmenté de 90.000 dépôts publics uniques, 12.000 ayant été dupliqué au moins une fois, pour un total de 135.000 dépôts. En Juillet 2010, GitHub a annoncé qu'il accueil 1 million de dépôts. En Avril 2011, GitHub a annoncé qu'il accueil 2 millions de dépôts.

Le 21 Septembre 2011 GitHub a annoncé qu'il avait atteint plus d'1 million d'utilisateurs. [14]

Le 13 Septembre 2012 sur leur page d'accueil, GitHub a annoncé qu'il avait plus de 2,1 millions d'utilisateurs accueillant plus de 3,7 millions de dépôts.

Le 19 Décembre 2012 GitHub a annoncé qu'il avait plus de 2,8 millions d'utilisateurs accueillant plus de 4,6 millions de dépôts.

Le 16 Janvier 2013, GitHub a annoncé qu'il avait franchi la barre des 3 millions d'utilisateurs et accueillant plus de 5 millions de dépôts.

**Social coding:** nouvelle tendance dans le développement logiciel.

Le codage social ou social coding approche le développement de logiciels d'une manière qui met en évidence la collaboration entre les développeurs en leur fournissant un environnement pour discuter, partager et créer.

Appelé aussi cloud coding e  est vraiment une innovation au développement car elle met le travail d'équipe dans un mélange particulier, puisque ça regroupe des développeurs ayant des compétences et des expériences différentes pour résoudre un problème . Grâce à la collaboration, ils sont capables de combiner chacune de leurs spécialités afin d'améliorer des logiciels existants ou développer de nouveaux produits.

Le social coding permet aux développeurs de faire beaucoup plus que de travailler avec un logiciel. Ce qui rend le cœur de toute communauté dépasser la capacité de ses membres, est le fait qu’ils puissent communiquer entre eux . Les développeurs peuvent partager leurs idées et apprendre des autres . Ils peuvent garder un œil sur les tendances importantes ou les nouveautés de l’industrie. Ils peuvent suivre ce que les autres travaillent sur et rester à jour sur les progrès réalisés, tout en apportant des informations sur leurs propres projets.

Pourquoi le social coding est l'avenir du développement ?

Dans le monde numérique d'aujourd'hui en grande partie , les réseaux sociaux sont devenu une partie intégrante pour notre capacité à rester en contact avec les autres. Et, au sein des entreprises , il est de plus en plus une composante essentielle des stratégies de marketing .

Les développeurs sont déjà contestés avec un travail exigeant environnement de travail avec le code n'est pas une tâche simple. Le social coding leur donne un moyen de se connecter avec d'autres développeurs dans un environnement qui encourage la coopération et facilite la collaboration , ce qui se traduira par un meilleur produit plus rapidement.

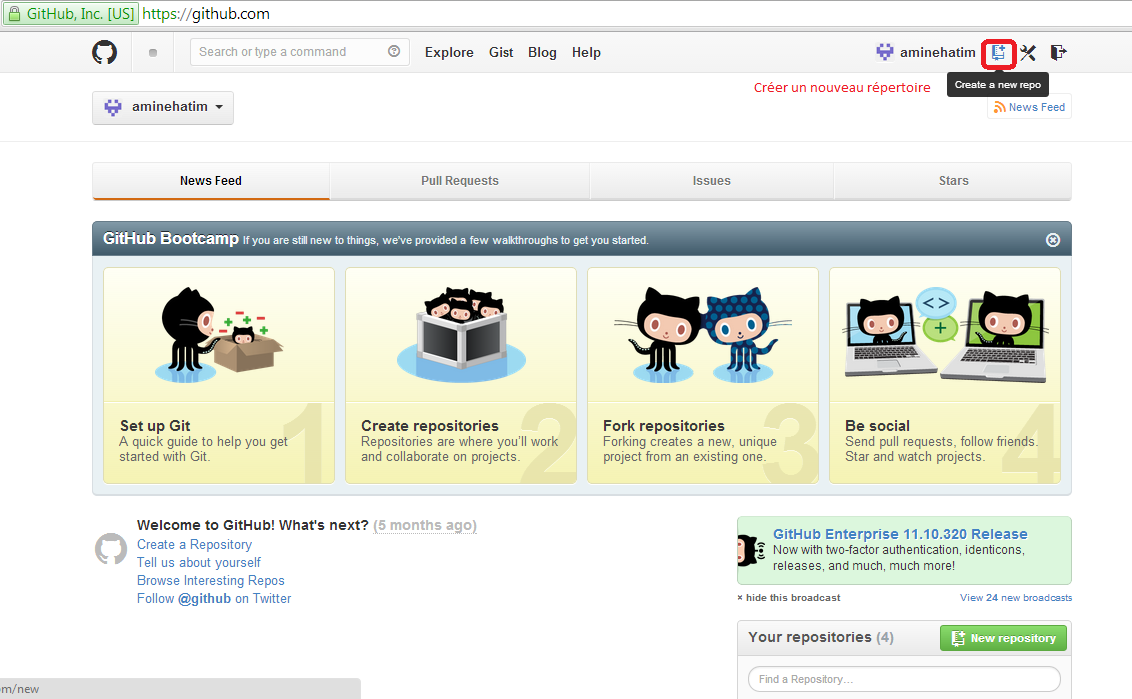
**Comment mettre tout ça en œuvre :**

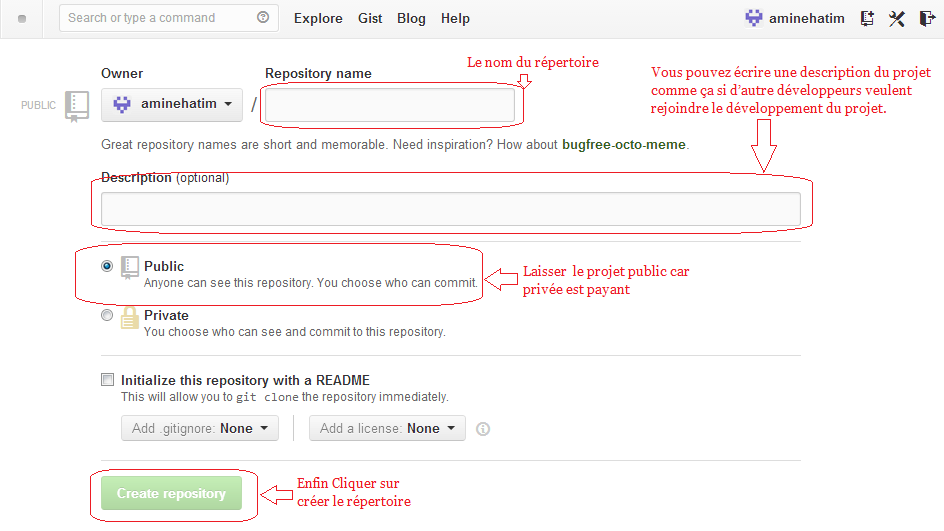
Etape 1 :

On commence par créer un compte gratuit sur github.com, L’interface est assez simple il faut juste un identifient un mot de passe et une adresse email.

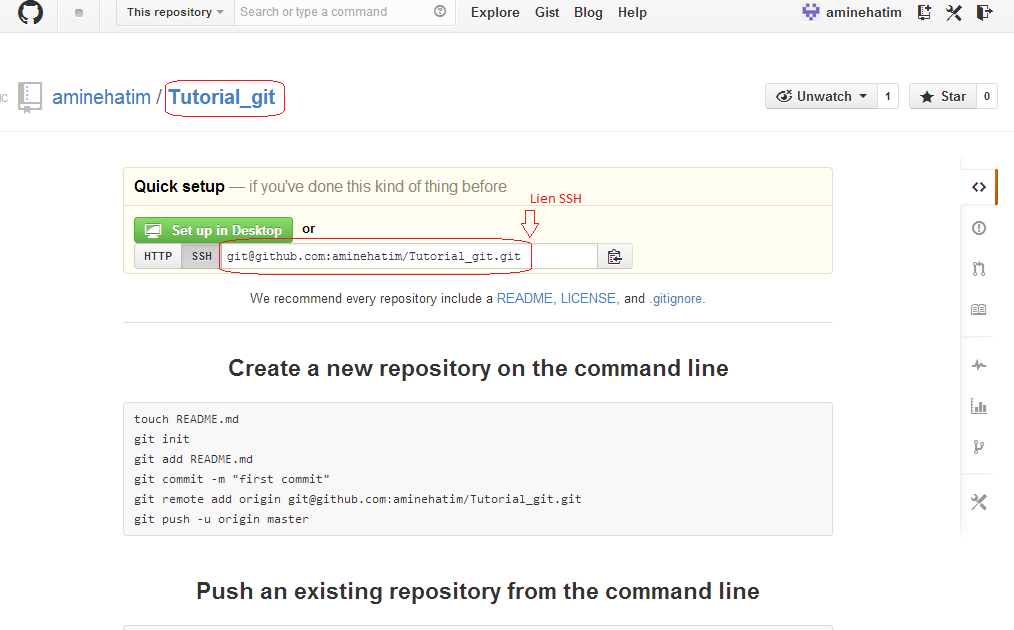
Etape 2 :

Une fois le compte est crée on peut accéder à l’interface de GitHub. Maintenant il faut crée un nouvel répertoire pour contenir notre projet.





Une fois le répertoire sera créer vous pouvez y accéder. Dans la figure qui suit le lien ssh représente l’adresse de notre répertoire ce dernier est essentiel pour synchroniser notre répertoire local a notre répertoire distant.

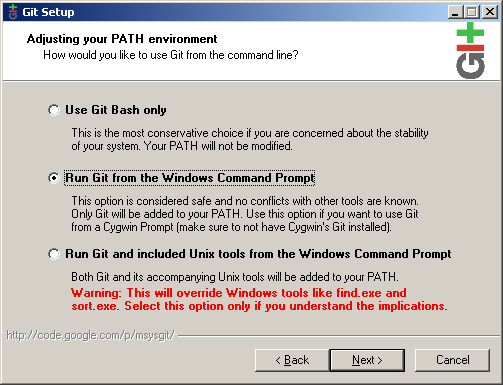


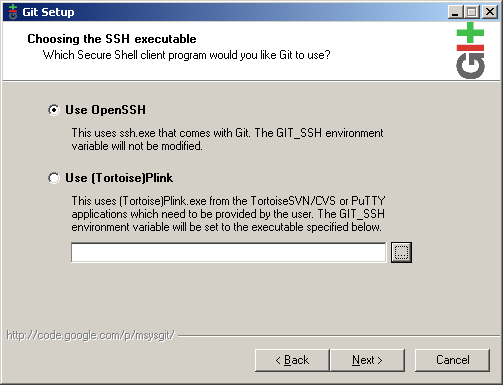
Etape 3 :

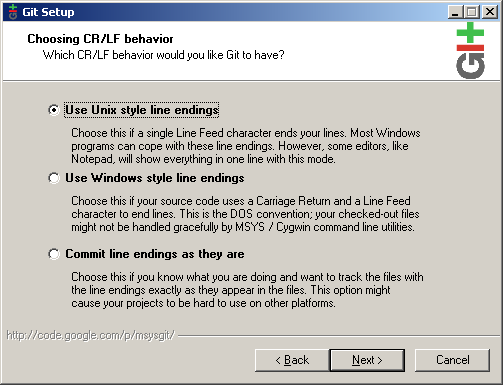
Dans cette étape il faut télécharger et installer Git et TurtoiseGit. TurtoiseGit est simplement une interface graphique pour Git, qui permet d’utiliser git sans se soucier a connaitre le toutes les commandes de Git.

Installation de Git :

Téléchargez msysgit depuis sourceforge.org et puis suivez les étapes suivantes :





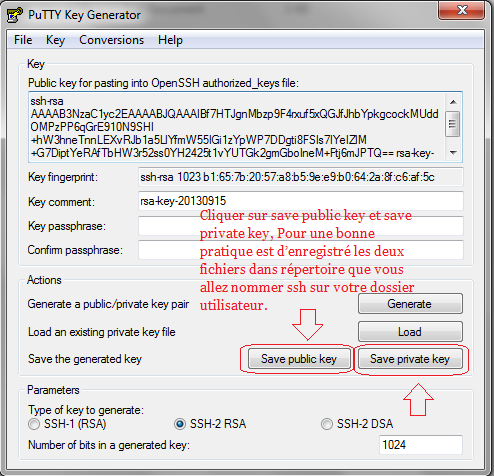
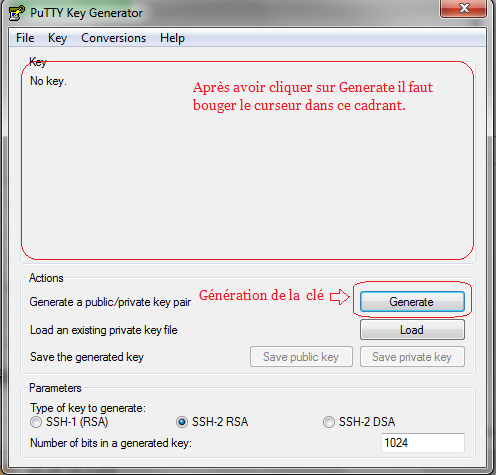


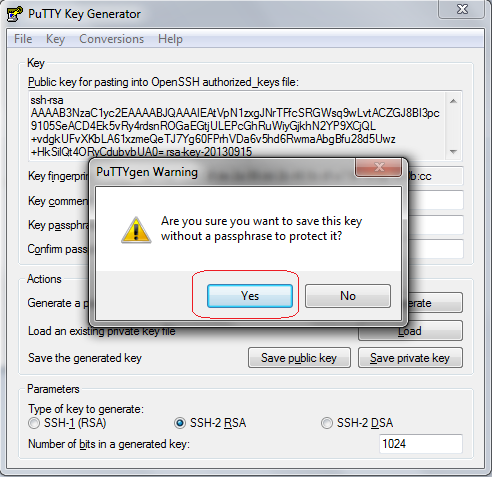
En ce qui concerne TurtoiseGit, il peut être téléchargé depuis sourceforge.org, et pour l’installation suivez les paramètres par défaut.

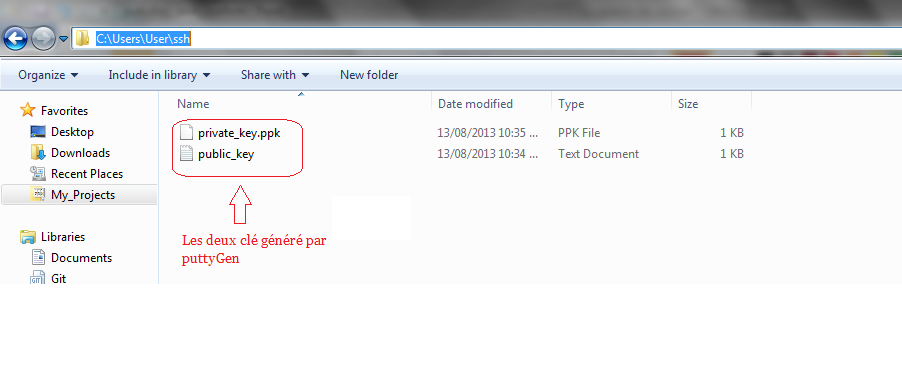
Etape 4 :

Maintenant il faut générer deux clé ssh une privé et une public. Une clé ssh est considérée comme un mot de passe mais un peu plus sécurisé. Cette clé va nous permettre une fois configurer avec Git, une authentification automatique à chaque connexion avec le répertoire distant. La clé privé reste sur votre ordinateur cependant la clé publique va être ajouté au compte GitHub.

Pour cela on va utiliser puttyGen : menu démarrer -> tous les programmes -> TurtoiseGit -> puttyGen.

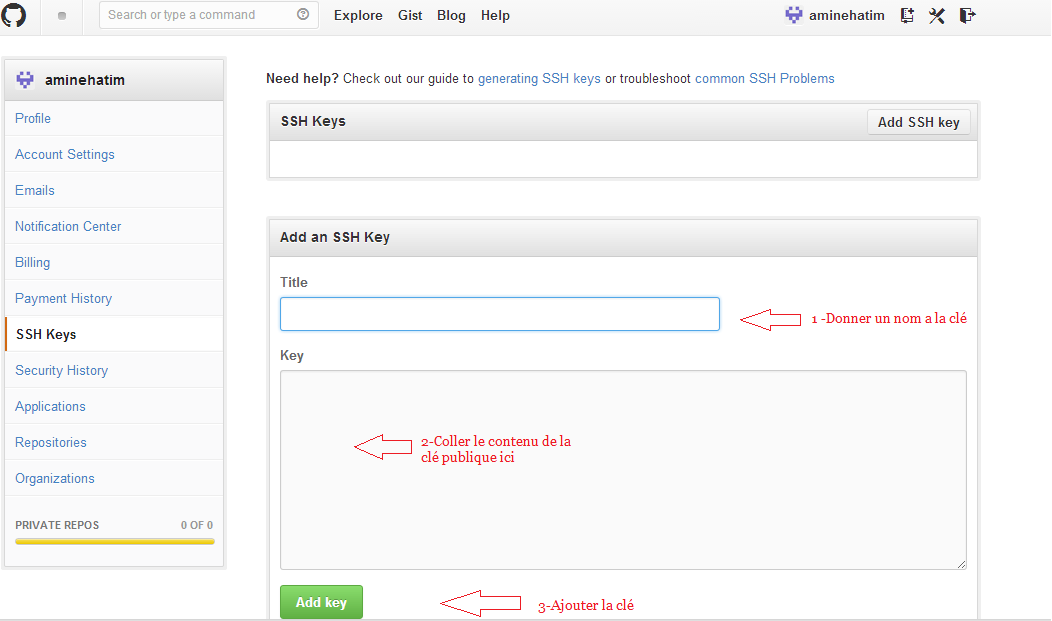
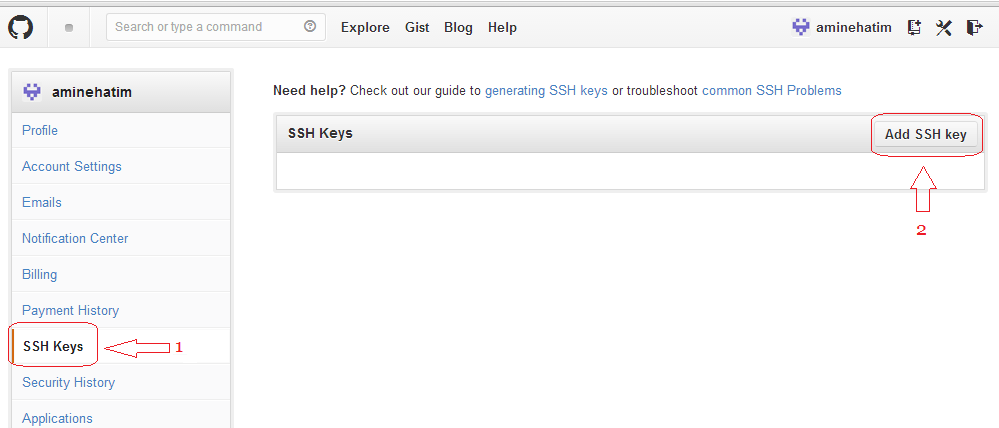
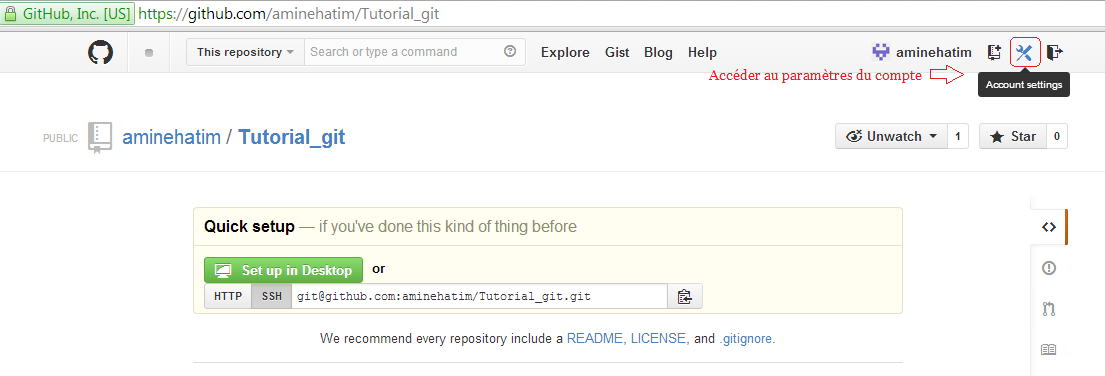






Etape 5 :

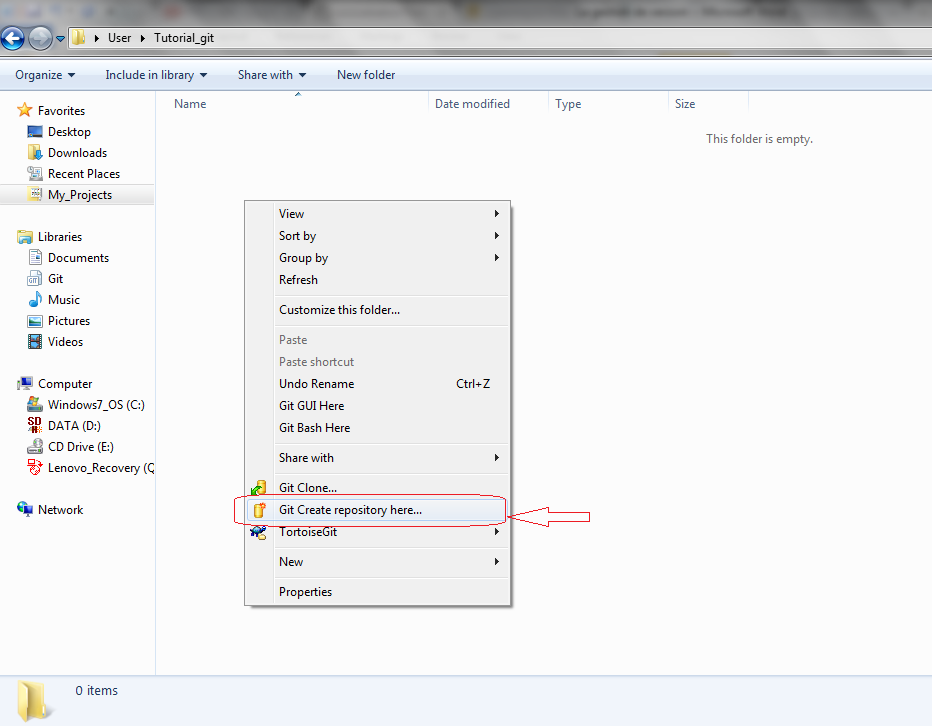
Ajouter la clé publique au compte GitHub. Pour cela copier le contenu de la clé publique (public\_key.txt), et suivez les étapes suivantes :



Etape 7 :

A cette étapes il faut créer un dossier pour le projet et le lier au compte GitHub.

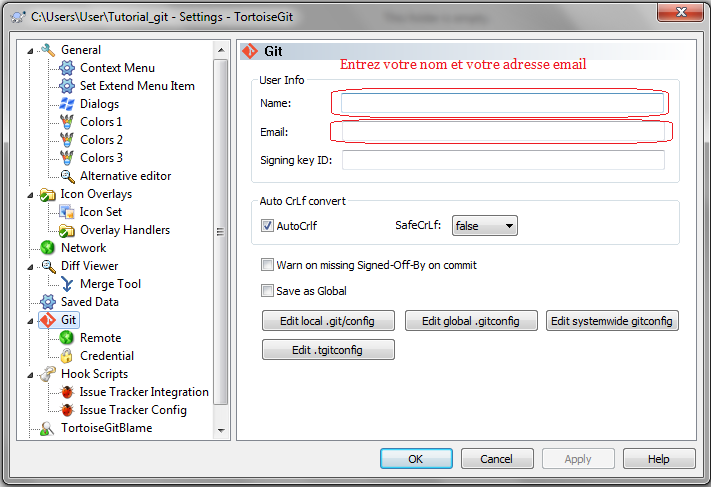
Apres avoir créer un répertoire pour le projet dans ce cas Tutorial\_git. On doit l’initialiser comme répertoire Git.



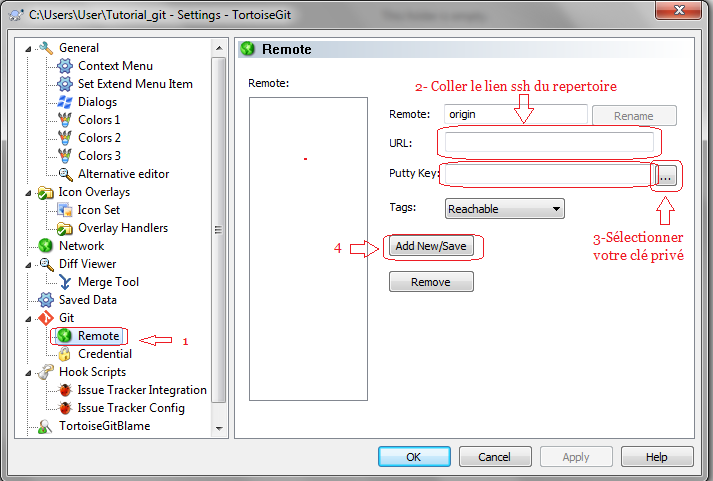


Puisque que c’est dans ce répertoire qu’on va créer notre projet et y travailler dessus, il faut laisser la case décocher. Un ‘Bare directory’ est répertoire de sauvegarde local seulement on ne peut pas faire des modifications sur une copie ‘bare’ du projet.

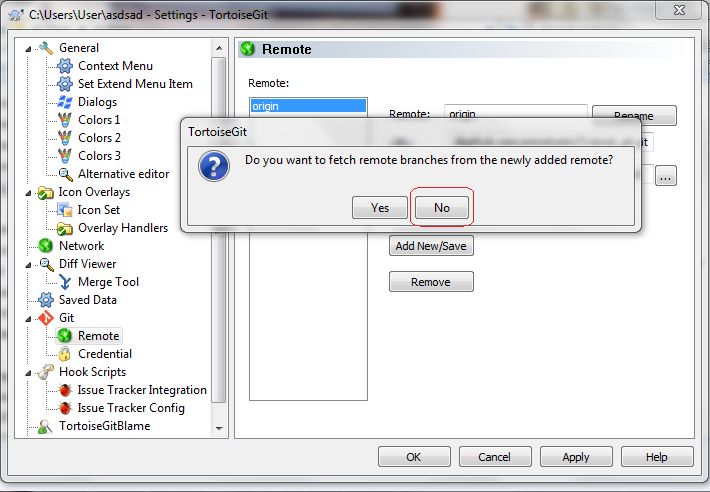
Avant de commencer a jouter des fichier a notre répertoire, il faut créer un utilisateur Git,



Ensuite il faut ajouter le lien ssh de notre répertoire pour que Git puisse synchroniser les deux répertoires distant et locale.



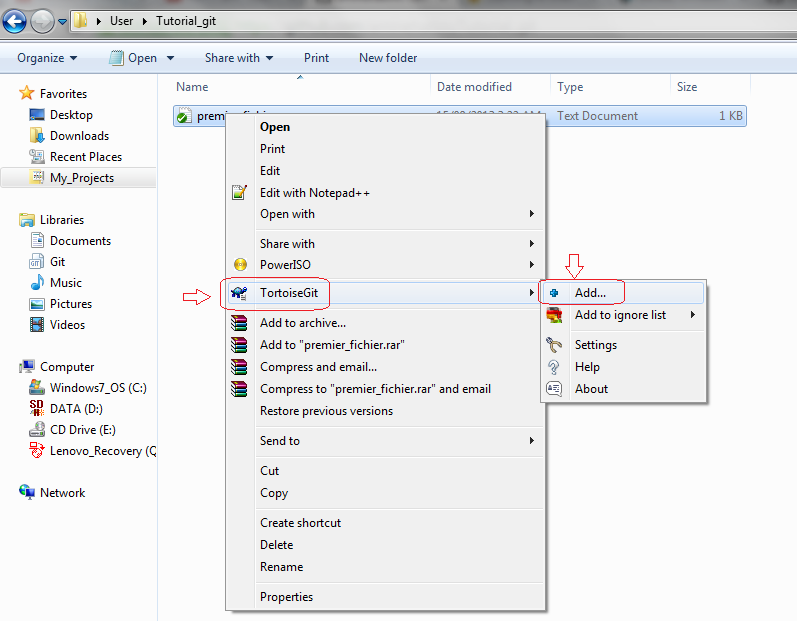
Des qu’on clique sur ‘Add New/Save ‘ une fenetre va apparaitre pour demander de faire un ‘fetch’. La commande ‘fetch’ permet de faire une copie du repertoire distant vers le repertoire local. Dans notre cas puisque le repertoire distant est vide on a pas besoin de faire un ‘fetch’.

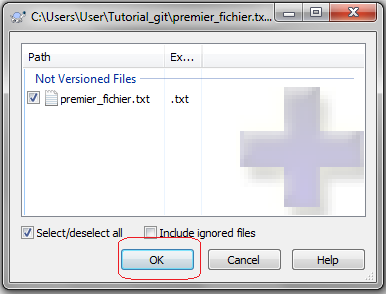


Etape 8 :

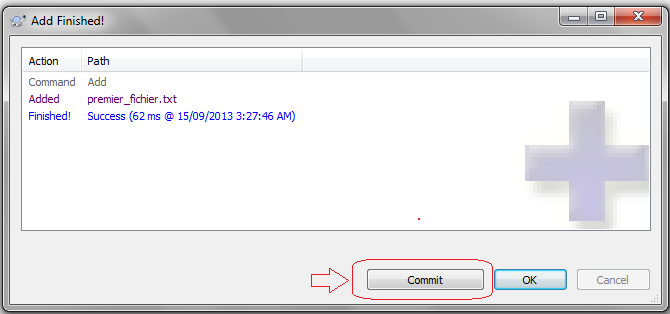
Maintenant tous est configurer, on va créer un nouveau fichier dans le répertoire de projet et puis faire le premier commit.

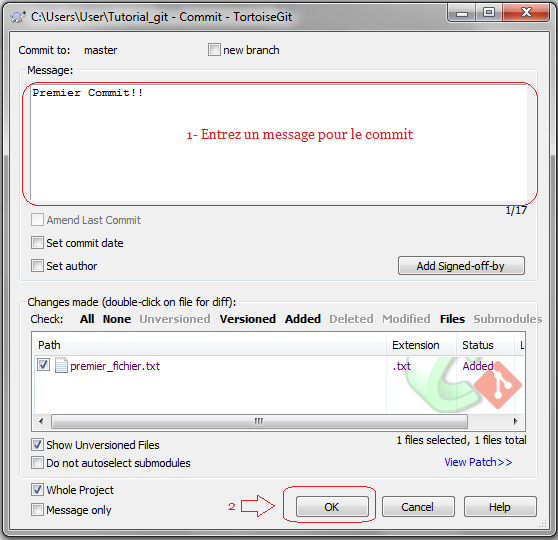
On va commencer par créer un simple fichier texte contenant « Hello world ! ».  A cette etape le repertoire local est synchroniser avec le distant, mais aucun fichier n’est ajouter pour la gestion de version. Donc on va utiliser la commande git add pour ajouter notre nouveau fichier.



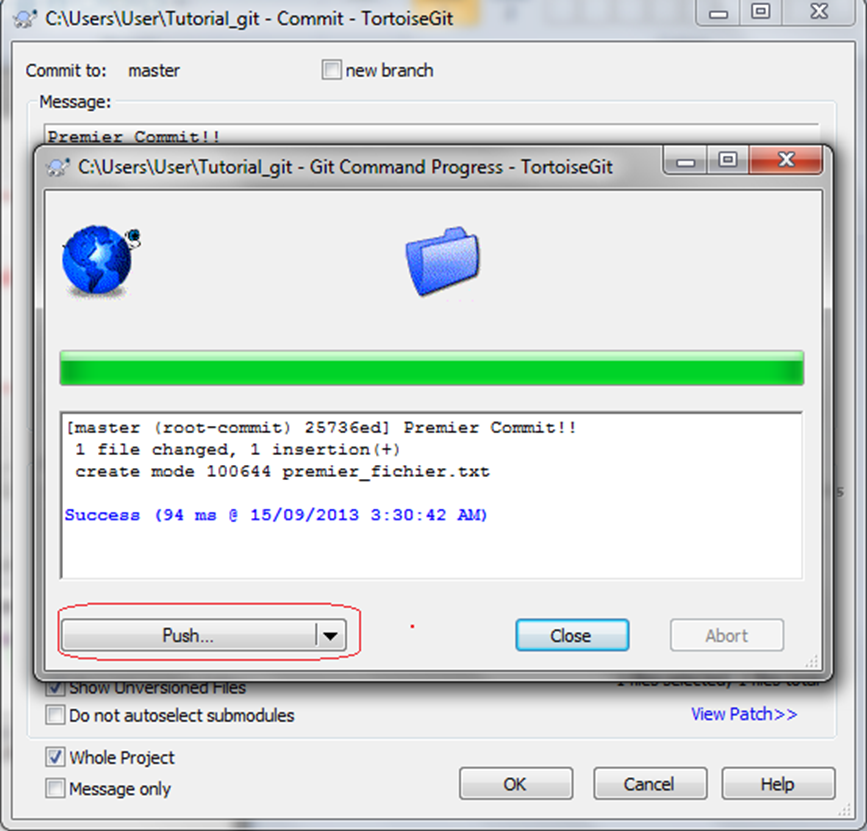
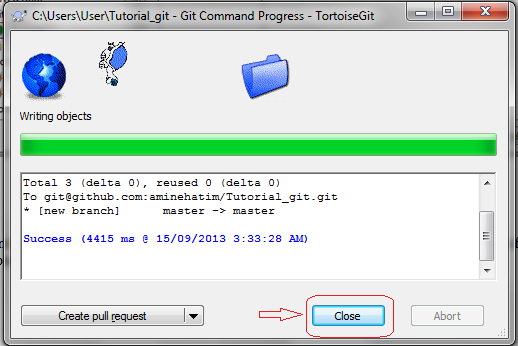


Maintenant git commence a faire un suivi des modification sur ce fichier. Il reste qu’a le comitter vers le repertoire distant.

Le message du commit doit etre claire est explique exactement le changement appliquer par le commit.



A ce nieau Git a enregister les changement sur notre fichier seulment sur la version local du repertoire de travail il reste qu’a l’envoye sur le repertoire distant pour cela on utilise la command ‘push ‘.



Maintenant le fichier existe sur le répertoire local est distant si on retourne au répertoire distant sur GitHub on peut voir que le fichier existe sur le répertoire distant.

