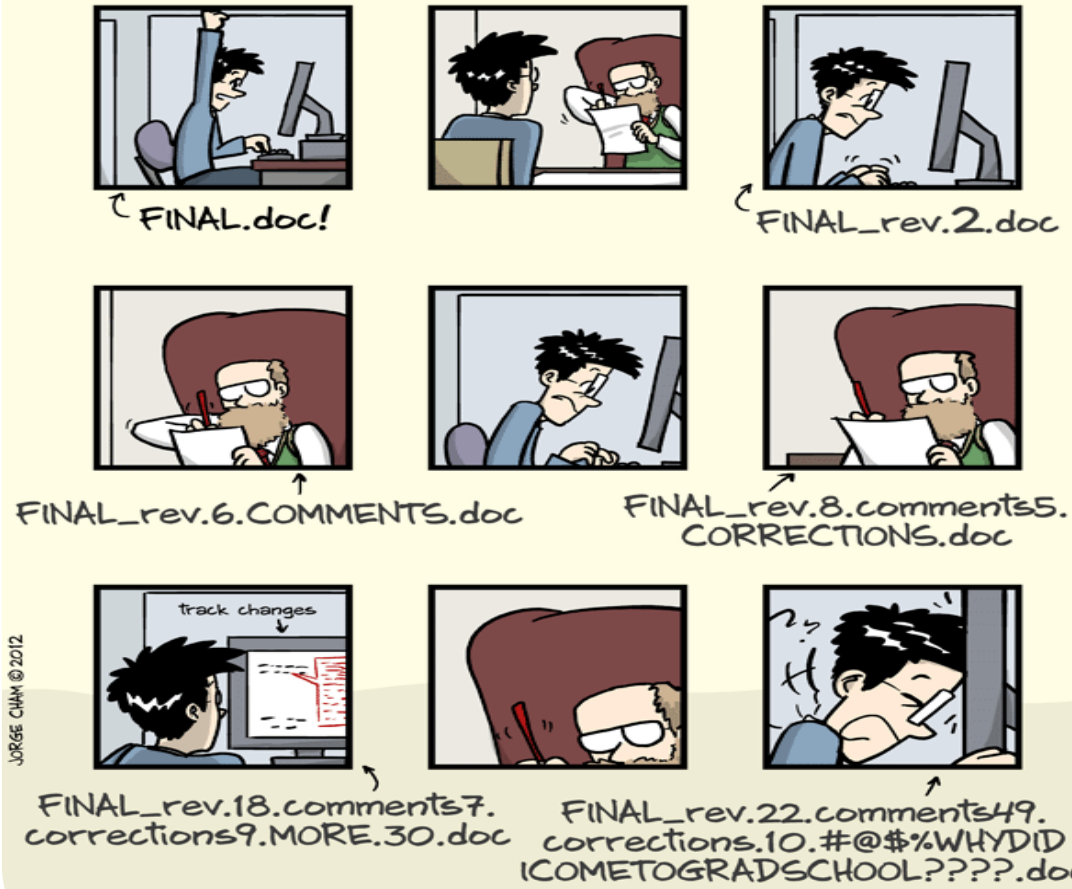
Cours GIT par la pratique

Par Cyrille MBIA

Objectifs

- Comprendre l’intérêt de git et github
- Maîtriser les commandes de base
- Savoir résoudre des conflits
- Principe de collaboration

"FINAL".doc


FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Pourquoi le contrôle de version ?

Collaboration Versioning Rollback Compréhension

Scénario : Plusieurs étudiants réalisent un projet ensemble

Question : Pourquoi pas Google drive & One drive ????

gestion du code source vs gestion du stockage de fichiers

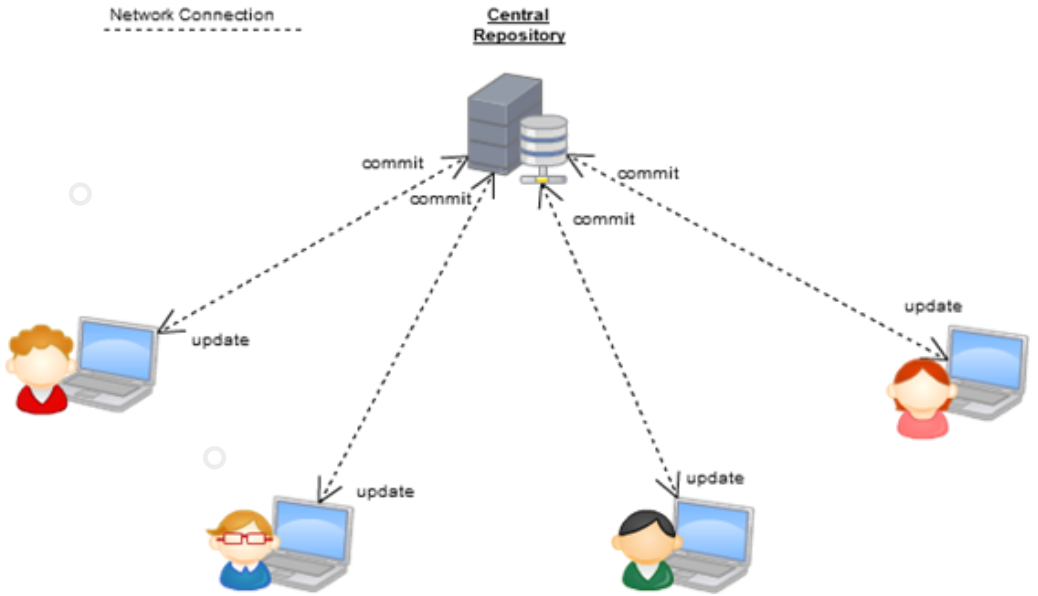
Les gestionnaires de versions

Les gestionnaires de version sont, de nos jours, fortement utilisés par les développeurs :

- Ils permettent d'archiver et de conserver les différentes étapes de développement d'un projet.
- Utilisé pour le travail collaboratif

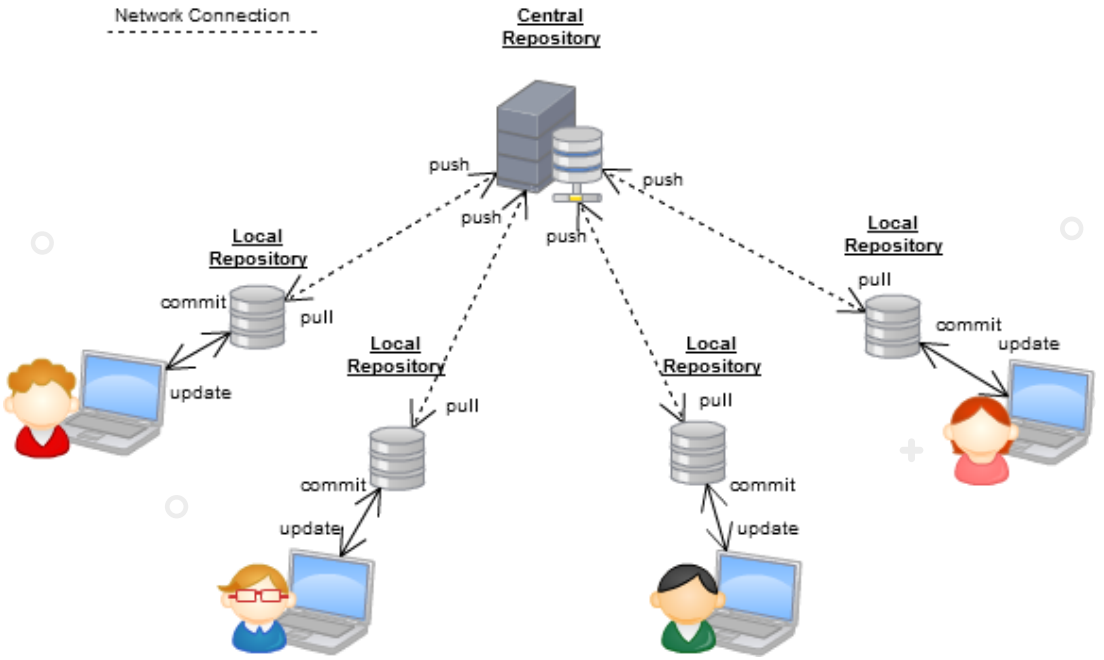
Centralisé/Décentralisé

La plupart des gestionnaires de version comme **Subversion** et **CVS** fonctionnent en mode **centralisé (dépôt centralisé,accès réseau requis ,etc...)**.



Centralisé/Décentralisé

Git lui, est un gestionnaire de version **décentralisée**, c'est à dire qu'il n'est pas nécessaire de disposer d'un serveur maître pour l'utiliser. Chacun des utilisateurs peuvent se synchroniser entre eux.



C’est quoi Git ? Pourquoi Git ?

Git : logiciel le plus couramment utilisé pour suivre les modifications dans n'importe quel ensemble de fichiers, généralement utilisé pour coordonner le travail entre les programmeurs développant en collaboration le code source pendant le développement du logiciel.

Git présente de nombreux avantages par rapport aux systèmes antérieurs tels que **CVS** et **Subversion.**

Histoire de Git

- Issu de la communauté de développement Linux
- **Linus Torvalds**, 2005
- Objectifs initiaux :
  - Vitesse
  - Prise en charge du développement non linéaire (des milliers de branches parallèles)
  - Entièrement distribué
  - Capable de gérer efficacement de grands projets comme Linux

C’est quoi Git ? Pourquoi Git ?

Parmi ses avantages, nous pouvons citer :

- Les identifiants universels de **commits (SHA1)**, permettant ainsi d'identifier un objet git de façon unique. Que ce soit un fichier, un répertoire, un commit etc.
- Il est multi-protocole, les échanges entre repository peuvent se faire via **http(s)**, **ssh**, **rsync**, ou encore en utilisant le protocole **Git** lui même.
- Un stockage des objets efficace grâce a la compression. Permettant ainsi de sauver beaucoup d'espace disque
- Tout le monde dispose du repository entier, et peut ainsi consulter les **logs** et/ou changements entre les révisions sans qu'il soit nécessaire de se connecter a un serveur maître.
- Et enfin, git est très efficace pour la gestion de branches de développement.

Fonctionnement de Git

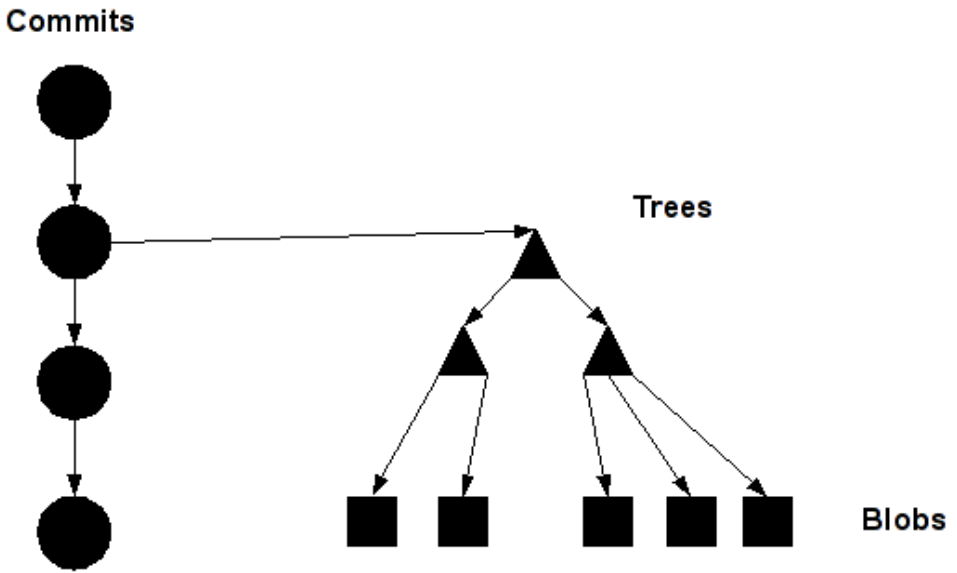
Git pense les données à la manière d'un flux **d'instantanés** ou **snapshots.** Chaque fois qu'on va valider ou enregistrer l'état d'un projet dans Git, il va prendre un instantané du contenu de l'espace de travail à ce moment et va enregistrer une référence à cet instantané pour qu'on puisse y accéder par la suite.

Chaque instantané est stocké dans une base de donnée locale, c'est-à-dire une base de donnée située sur notre propre machine.

Le fait que l'on dispose de l'historique complet d'un projet localement fait que la grande majorité des opérations de Git peuvent être réalisées localement, c'est-à-dire sans avoir à être connecté à un serveur central distant. Cela rend les opérations beaucoup plus rapides et le travail de manière générale beaucoup plus agréable.

Structure d'un repository git

Voici les différents objets stockés dans un repository git.



Structure d'un repository git

Nous avons donc :

- Les **blobs**, qui représente en réalité un fichier, ou plutôt une version bien précise d'un fichier
- Les **trees**, qui sont en fait des répertoires, contenant des objets blobs, comme dans tout système de fichiers
- Les **commits**, le nom parle de lui même
- Les **tags**, un tag peut être associé à un commit, afin de l'identifier plus simplement

Afin de mieux comprendre le fonctionnement de ces objets, voici quelques illustration de leur utilisation. Mais avant tout, il faut savoir que **chaque objet dispose d'un identifiant unique, sous forme de SHA1**.

Structure d'un repository git

Recherchons dans un premier temps un commit sur lequel fonder notre exemple. Pour cela lançons la commande **git log** qui nous affiche les commits du projet, choisissons en un par son identifiant SHA1.

Une fois notre commit exemple choisit, analysons sont contenu par le biais de la commande **git cat-file** avec l'option **"-p"**. Cette commande de git est en quelque sorte une commande de bas niveau, nous permettant d'analyser le contenu brut d'un objet git.

Dans le contenu de ce commit, nous pouvons dans un premier temps remarquer une référence à un commit parent, ce qui parait plutôt logique pour un gestionnaire de version. Et enfin ce qui nous intéresse le plus, la référence **tree**, qui est en fait une référence sur un répertoire contenant les objets de notre commit.

Affichons maintenant les informations sur le contenu de ce **tree.**

Différence entre Git et Github

Le nom git et github ce n’est pas la même chose. **Git** est un outil de gestion de version alors que **github** est une plateforme en ligne qui permet, entre autre, d’héberger des dépôts Git. Ne vous inquiétez pas si cela vous semble encore un peu obscure, nous allons justement éclaircir tout cela dans ce qui suit.

C’est quoi Github? - Fonctionnement

Github est un service en ligne qui permet entre autre d’héberger des dépôts Git.

Il est totalement gratuit pour des projets ouverts au public mais il propose également des formules payantes pour les projets que l’on souhaite rendre privés.

Différence entre Git et Github

Github propose également de nombreux autres services très intéressants comme par exemple:

- **Partager du code source** avec d'autres développeurs.
- Signaler et gérer les problèmes ou bugs de votre code source via les **issues**.
- Partager des portions de code via les **Gists**
- **Proposer des évolutions** pour un projet opensource.
- Et bien plus encore

Installation de git

Installer Git pour Linux

Utilisez le système de gestion de package natif de la distribution Linux pour installer et mettre à jour Git. Par exemple, sur Ubuntu :

sudo apt-get install git

Configurer Git sur Linux

Configurez le nom et l’adresse e-mail avant de commencer à utiliser Git. Git attache ces informations aux modifications et permet à d’autres utilisateurs d’identifier les modifications qui appartiennent aux auteurs.

Configuration initiale de git

Exécutez les commandes suivantes à partir de l'invite de commandes après avoir installé Git pour configurer ces informations :

> git config --global user.name "Cyrille MBIA"

> git config --global user.email "<cyrille@itdreamtech.com>"

Quatre états d’un projet Git:

- **Non suivi**: fichier n’étant (n’appartenant) pas ou plus géré par Git;
- **Non modifié**: fichier sauvegardé de manière sûre dans sa version courante dans la base de données du dépôt;
- **Modifié**: fichier ayant subi des modifications depuis la dernière fois qu’il a été soumis;
- **Indexé**: idem pour **modifié**, sauf qu’il sera pris instantané dans sa version courante de la prochaine soumission (commit).

Zones de travail d’un projet Git:

Les états de fichiers sont liés à des zones de travail dans Git. En fonction de son état, un fichier va pouvoir apparaître dans telle ou telle zone de travail. Tout projet Git est composé de trois sections :

- Le répertoire de travail (working tree)
- La zone d’index (staging area)
- Le répertoire Git (repository).

Zones de travail d’un projet Git:

Le répertoire de travail ou **working tree** correspond à une extraction unique d’une version du projet. Les fichiers sont extraits de la base de données compressée dans le répertoire Git et sont placés sur le disque afin qu’on puisse les utiliser ou les modifier. Lorsque vous ouvrez les fichiers d'un projet géré par Git, vous accédez répertoire de travail.

La zone d’index ou **staging area** correspond à un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané ou du prochain **commit**.

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 principale ou le cœur de Git.

Zones de travail d’un projet Git:

Le processus de travail va ainsi être le suivant :

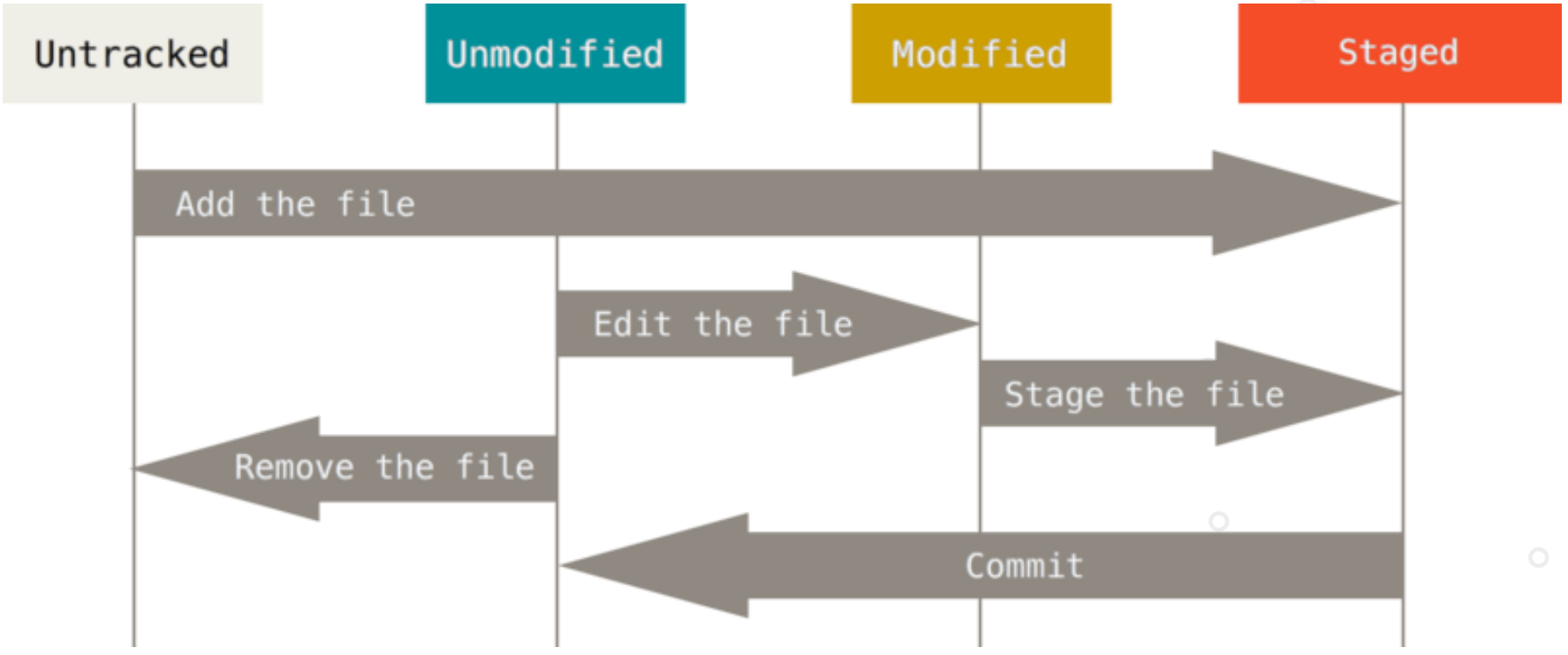
Nous allons travailler sur nos fichiers dans le répertoire de travail.

Lorsqu’on modifie ou crée un fichier, on peut ensuite choisir de l’indexer.

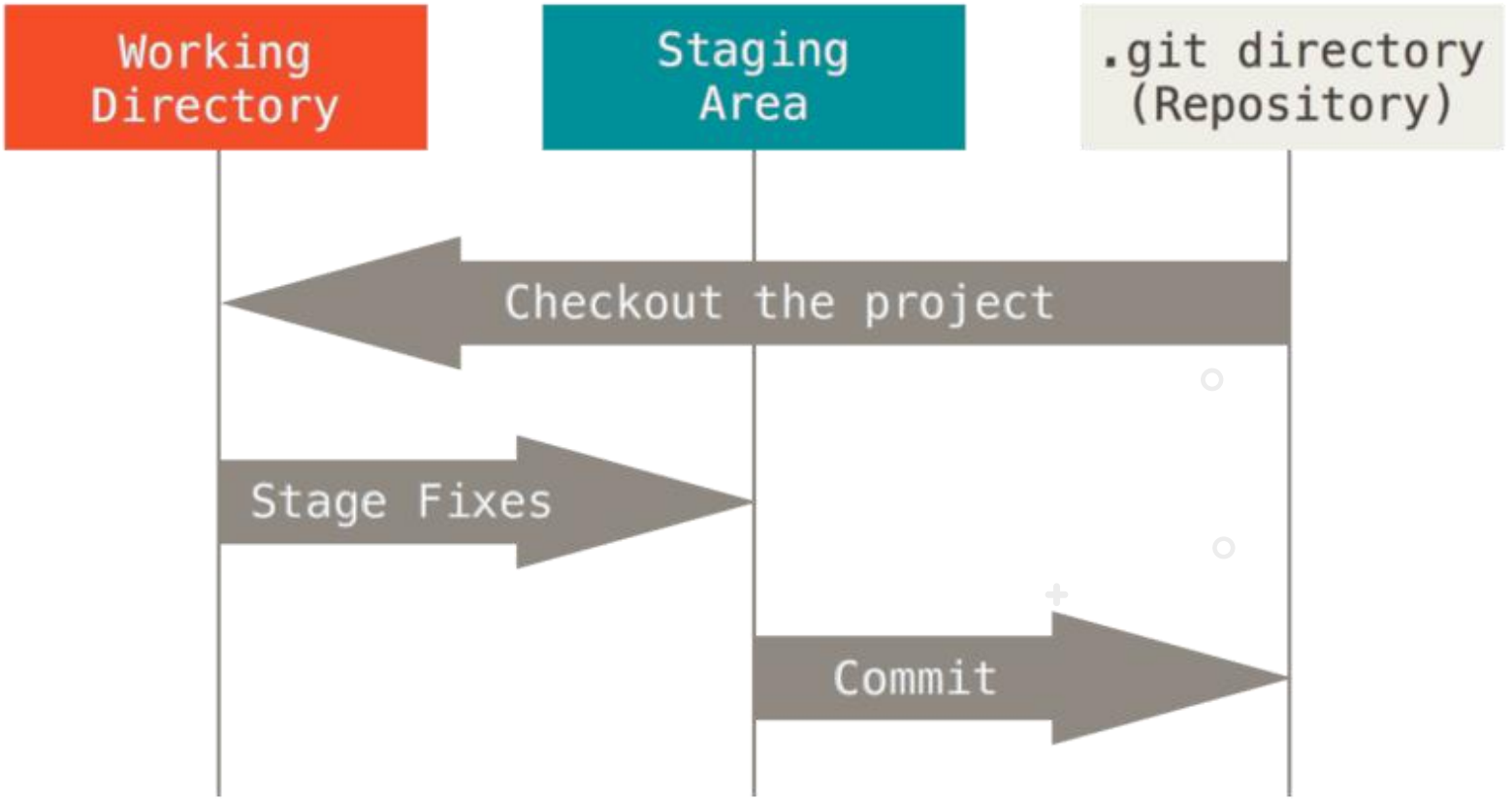
Tant qu’un fichier n’est pas indexé, il possède l’état modifié ou est non suivi si c’est un nouveau fichier.

Dès qu’il est indexé i.e que son nom est ajouté à la zone d’index, il possède l’état indexé. Finalement, on va valider (**commit**) la version indexée de nos fichiers pour les ajouter au répertoire Git.

Zones de travail d’un projet Git:



Zones de travail d’un projet Git:



Les commandes de base de 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 ; (git init)

On peut cloner un dépôt Git déjà existant. (git clone)

Vous clonez un dépôt avec git clone [url]. On a à la fois le clone par l’url ssh qui demande une clé public et par https qui demande à chaque fois l’authentification.

Les commandes de base de Git

a. Vérifier l’état des fichiers

$ **git status**

b. Indexer l’ajout ou les changements d’un fichier avant de soumettre (commit) les modifications

$ **git add -A**

Les commandes de base de Git

c. Valider les modifications

$ **git commit –m « Mon premier commit»**

La commande «commit» est faite pour valider ce qui a été indexé avec «git add».

Pas d’indexe pas de validation.

Après l’option –m est suivi d’un commentaire de l’utilisateur décrivant ce qui a été accompli et le fichier est ajouté au répertoire Git/dépôt (local) mais pas encore sur le dépôt distant.

Les commandes de base de Git

d. Visualiser l’historique des validations

$ **git log**

Par défaut, git log énumère en ordre chronologique inversé des commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier.

Les commandes de base de Git

Empêcher l’indexation de certains fichiers dans Git

Lorsqu’on dispose d’un projet et qu’on souhaite utiliser Git pour effectuer un suivi de version, il est courant qu’on souhaite exclure certains fichiers du suivi de version comme certains fichiers générés automatiquement, des fichiers de configuration, des fichiers sensibles, etc.

On peut informer Git des fichiers qu’on ne souhaite pas indexer en créant un fichier **.gitignore** et en ajoutant les différents fichiers qu’on souhaite ignorer.

Les commandes de base de Git

e. Pousser son travail sur un dépôt distant

$ **git push origin master**

La commande «push» sert à envoyer tous les «commits» effectués se trouvant dans le répertoire Git/dépôt (HEAD) de la copie du dépôt local vers le dépôt distant.

f. Récupérer et tirer depuis des dépôts distants

$ **git pull**

La commande «pull» permet de mettre à jour votre dépôt local des dernières validations (modifications des fichiers). La commande est faite avant l’indexation des modifications.

Les commandes de base de Git

Rapport entre répertoire de travail et commits

Il nous faut préciser un peu mieux le rapport entretenu par Git entre l'historique et le répertoire de travail. De manière conceptuelle, sans trop entrer dans les détails donc, il faut considérer que :

En utilisant la commande **git add**, on demande à Git de suivre certains fichiers, c'est-à-dire de prendre en charge la gestion de l'évolution de ces fichiers,

par la commande **git commit**, nous demandons à Git d'enregistrer un snapshot, c'est-à-dire une photographie instantanée de l'état des fichiers suivis (plus exactement, des modifications qui ont été placées dans l'index, mais il s'agit là en fait d'une facilité), une suite des commits constitue un historique.

Les commandes de base de Git

Rapport entre répertoire de travail et commits

Git va alors voir votre répertoire de travail comme étant en fait composé de :

- l'ensemble des fichiers suivis, dans leur version correspondant à UN SNAPSHOT COURANT (un commit)
- PLUS les modifications existant sur ces fichiers par rapport à cette version spécifique
- PLUS des fichiers non suivis (pour lesquels on n'a pas effectué de git add)

Les commandes de base de Git

Rapport entre répertoire de travail et commits

Pour désigner le **SNAPSHOT COURANT**, Git utilise une référence (c'est à dire, un pointeur vers un commit) qui s'appelle **HEAD**.

Nous avons déjà vu une commande qui modifie ces deux références. En effet la commande git commit effectue deux choses :

- Elle enregistre un snapshot ayant un lien de parenté avec le commit référencé par HEAD
- Elle déplace la référence HEAD et la référence master sur ce nouveau commit.