



INTRODUCTION

Qu'est-ce que Git?

Git est un système de contrôle de version distribué open source. Il est conçu pour gérer des projets mineurs à majeurs avec une vitesse et une efficacité élevées. Il est développé pour coordonner le travail entre les développeurs. Le contrôle de version nous permet de suivre et de travailler avec les membres de notre équipe dans le même espace de travail.

Git est la base de nombreux services comme GitHub et GitLab, mais nous pouvons utiliser Git sans utiliser d'autres services Git. Git peut être utilisé en privé et en public.

Git a été créé par Linus Torvalds en 2005 pour développer le noyau Linux. Il est également utilisé comme un important outil de contrôle de version distribué pour le DevOps.

Git est facile à apprendre et offre des performances rapides. Il est supérieur aux autres outils **SCM** (Supply Chain Management) tels que Subversion, CVS (Concurrent Versions System), Perforce et ClearCase.

La différence entre Git, Github ou Gitlab

Il est important de comprendre la différence entre Git et Github ou Gitlab.

Git est un outil qui permet de gérer différents projets en les envoyant sur un serveur. Ce dernier est connecté à l'ordinateur d'autres développeurs qui envoient leur code et récupèrent le vôtre. Toute personne qui travaille sur un projet est connectée avec les autres, tout est synchronisé. Quant à **Github** et **Gitlab**, il s'agit d'un logiciel, ou plateforme. Leur rôle est d'héberger ces différents projets qui utilisent Git.

Concrètement, ils proposent une interface graphique qui vous en simplifie l'utilisation. Par exemple, chez **Phoenix**, nous utilisons un Gitlab sur un serveur privé, qui est lui-même sauvegardé sur un serveur miroir quotidiennement. Cela nous permet de garder une trace de toutes les mises à jour que nous effectuons sur les projets de nos clients.

Pourquoi utiliser GIT ?

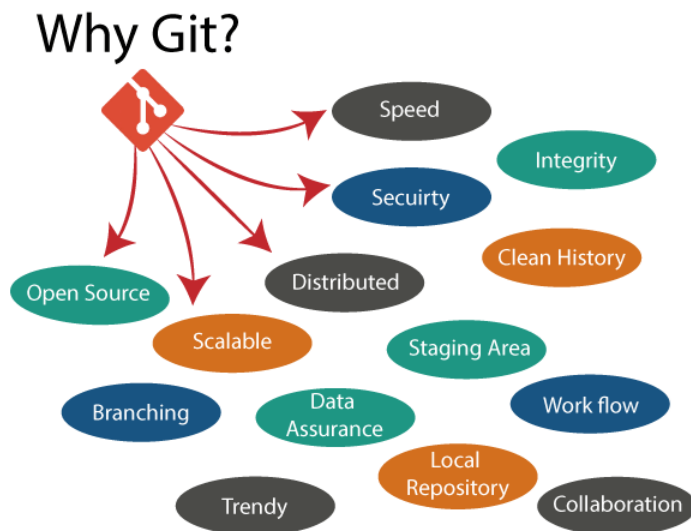
Git est un outil qui permet de gérer différents projets en les envoyant sur un serveur. Ce dernier est connecté à l'ordinateur d'autres développeurs qui envoient leur code et récupèrent le vôtre. Toute personne qui travaille sur un projet est connectée avec les autres, tout est synchronisé.

Dans la suite de ce cours, Nous parlerons des nombreuses fonctionnalités et avantages de Git qui démontrent sans aucun doute Git en tant que système de contrôle de version leader. Maintenant, nous allons discuter d'autres points sur les raisons pour lesquelles nous devrions choisir Git.

Si vous hésitez à franchir le pas et commencer à utiliser Git, j'espère que ces trois points pourront vous aider :

- L'**aspect backup** régulier du code, le versioning, est essentiel dans le développement web. Une panne de disque dur pourrait être fatale pour vous, par exemple, les délais de livraison. C'est un risque à ne pas prendre.
- L'**aspect collaboratif** qu'offre Git devient rapidement essentiel lorsqu'il y a plus d'un développeur dans le projet. Aujourd'hui, comme un certain nombre de développeurs, je pense que chez Phoenix, nous ne pourrions plus nous en passer.
- Enfin, il permet de **garder une trace** des versions antérieures du site. Très utile en cas de debug.

Pour résumer, Git c'est tout simplement parfait pour manager vos projets et éviter des erreurs qui peuvent coûter très cher. Il faut que cela devienne un automatisme dans votre manière de travailler. Vous y gagnerez à la fois en temps et en qualité, faites-nous confiance. Il est possible que dans un prochain article nous voyions comment l'utiliser en ligne de commande ainsi que les solutions alternatives existantes.



Intégrité Git

Git est **développé pour garantir la sécurité et l'intégrité** du contenu contrôlé par version. Il utilise la somme de contrôle pendant le transit ou la falsification du système de fichiers pour confirmer que les informations ne sont pas perdues. En interne, il crée une valeur de somme de contrôle à partir du contenu du fichier, puis la vérifie lors de la transmission ou du stockage des données.

Système de contrôle de version à la mode

Git est le **système de contrôle de version le plus utilisé**. Il a un **maximum de projets** parmi tous les systèmes de contrôle de version. En raison de son flux de travail et de ses fonctionnalités incroyables, c'est un choix préféré des développeurs.

Tout est local

Presque toutes les opérations de Git peuvent être effectuées localement; c'est une raison importante pour l'utilisation de Git. Nous n'aurons pas à assurer la connectivité Internet.

Collaborez à des projets publics

Il existe de nombreux projets publics disponibles sur le GitHub. Nous pouvons collaborer sur ces projets et montrer notre créativité au monde. De nombreux développeurs collaborent sur des projets publics. La collaboration nous permet d'être aux côtés de développeurs expérimentés et d'apprendre beaucoup d'eux; ainsi, il prend nos compétences en programmation au niveau suivant.

Impressionnez les recruteurs

Nous pouvons impressionner les recruteurs en mentionnant Git et GitHub sur notre CV. Envoyez votre lien de profil GitHub aux ressources humaines de l'organisation que vous souhaitez rejoindre. Montrez vos compétences et influencez-les à travers votre travail. Cela augmente les chances d'être embauché.

Installation

➤ Comment installer Git sur Windows

Pour utiliser Git, vous devez l'installer sur votre ordinateur. Même si vous avez déjà installé Git, c'est probablement une bonne idée de le mettre à niveau vers la dernière version. Vous pouvez soit l'installer sous forme de package ou via un autre installateur, soit le télécharger depuis son site officiel.

Maintenant, la question se pose de savoir comment télécharger le package d'installation de Git. Vous trouverez ci-dessous le processus d'installation par étapes qui vous aide à télécharger et à installer Git.

Comment télécharger Git?

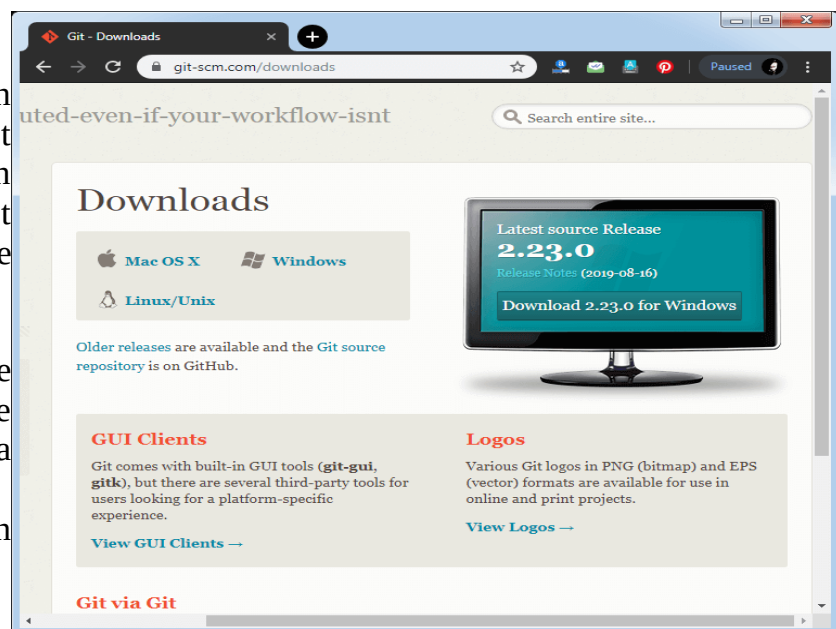
Étape 1

Pour télécharger le programme d'installation de Git, visitez le site officiel de Git et accédez à la page de téléchargement. Le lien vers la page de téléchargement est <https://git-scm.com/downloads>. La page ressemble à

Cliquez sur le package donné sur la page en téléchargement 2.23.0 pour Windows. Le téléchargement commencera après la sélection du package.

- Maintenant, le package d'installation de Git a été téléchargé.

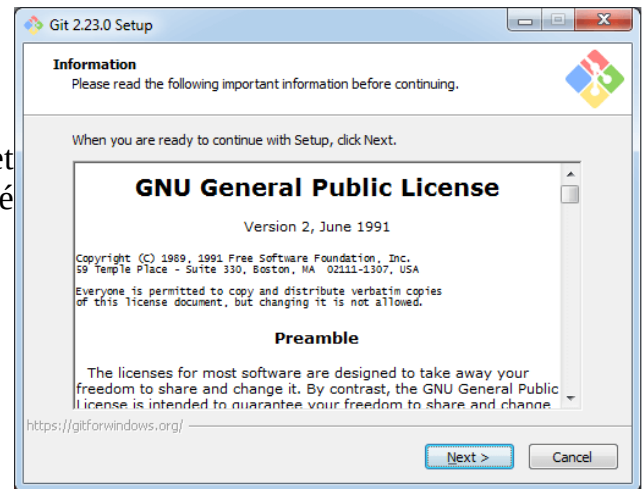
Installez Git



Étape 2

Cliquez sur le fichier d'installation téléchargé et sélectionnez Oui pour continuer. Après avoir sélectionné oui, l'installation commence et l'écran ressemblera à

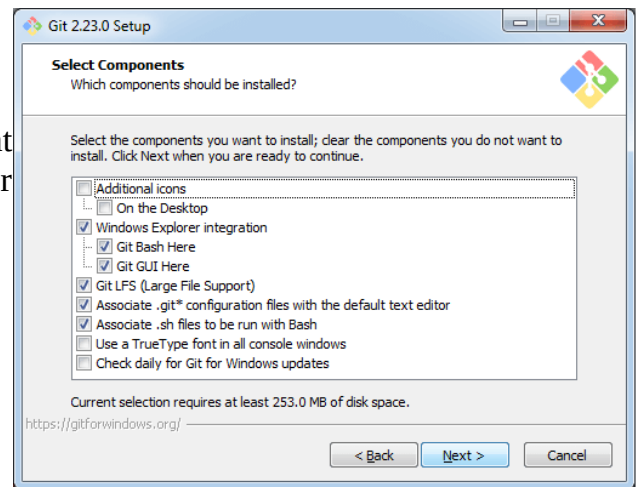
- Cliquez sur **suivant** pour continuer.



Étape 3

Les composants par défaut sont automatiquement sélectionnés à cette étape. Vous pouvez également choisir la pièce souhaitée.

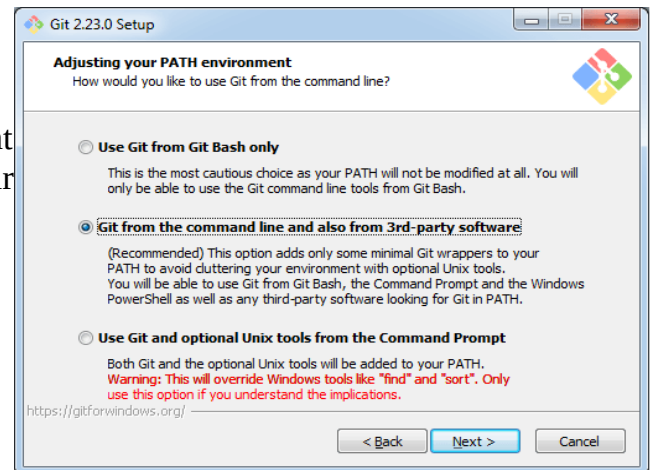
- Cliquer sur **Suivant** pour continuer.



Étape 4

Les options de ligne de commande Git par défaut sont sélectionnées automatiquement. Vous pouvez choisir votre choix préféré.

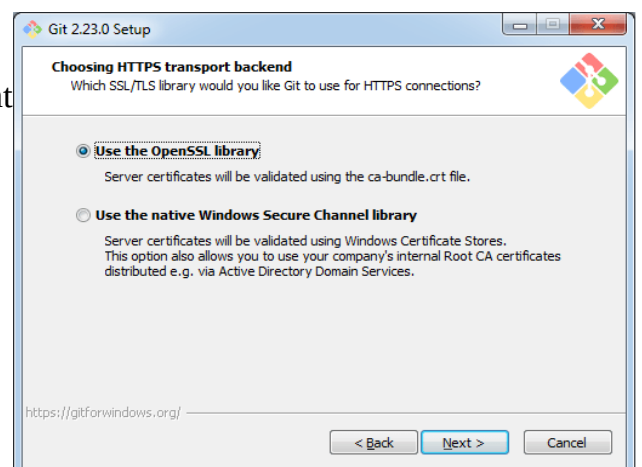
- Cliquer sur **Suivant** pour continuer.



Étape 5

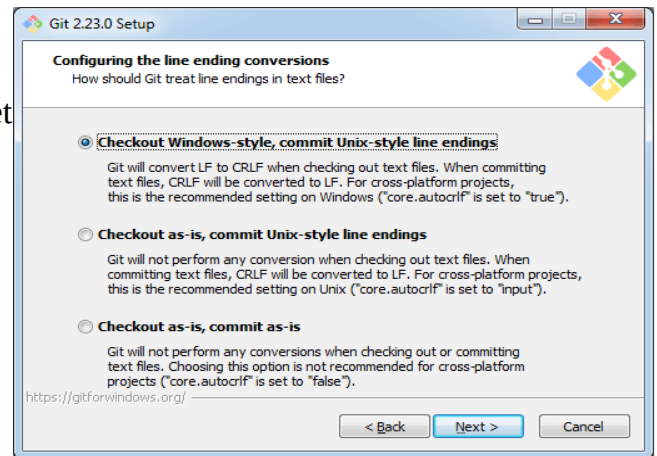
Les options de backend de transport par défaut sont sélectionnées à cette étape.

- Cliquer sur Suivant pour continuer.



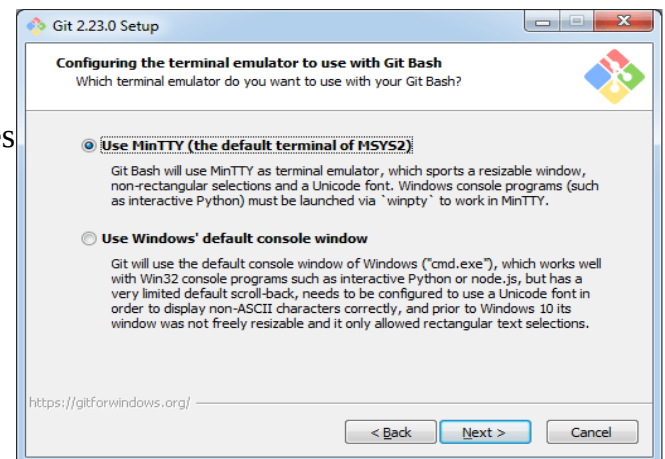
Étape 6

Sélectionnez l'**option** de fin de ligne souhaitée et cliquez sur **Suivant** pour continuer.



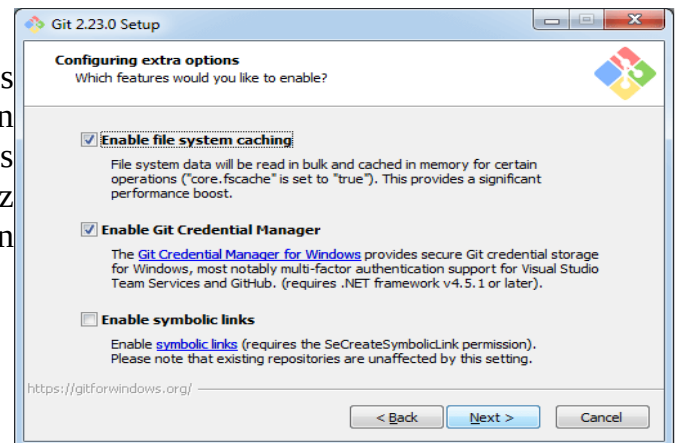
Étape 7

Sélectionnez les clics d'émulateur de terminal préférés sur le **suivant** pour continuer.



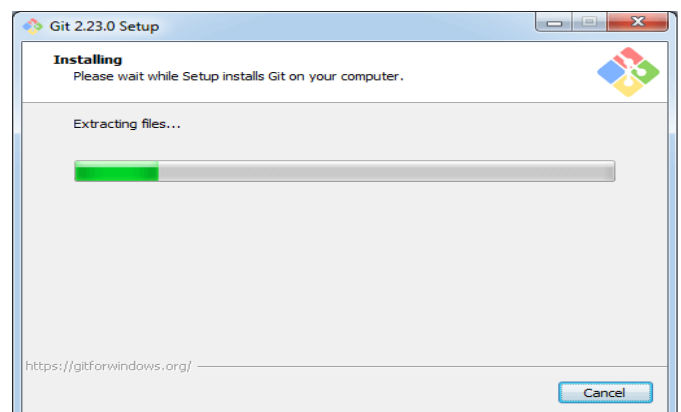
Étape 8

Il s'agit de la dernière étape qui fournit des fonctionnalités supplémentaires telles que la mise en cache du système, la gestion des informations d'identification et le lien symbolique. Sélectionnez les fonctionnalités requises et cliquez sur l'option **suivante**.



Étape 9

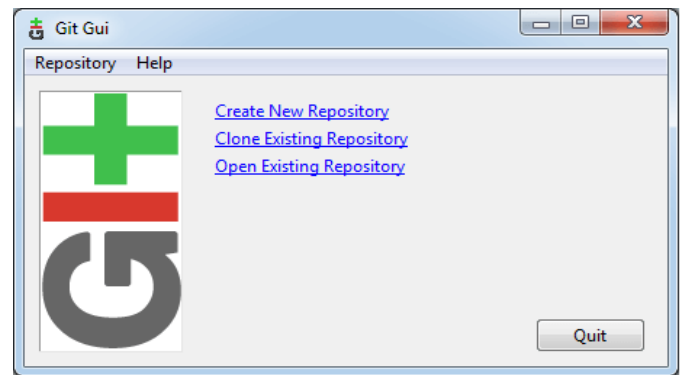
Les fichiers sont en cours d'extraction à cette étape.



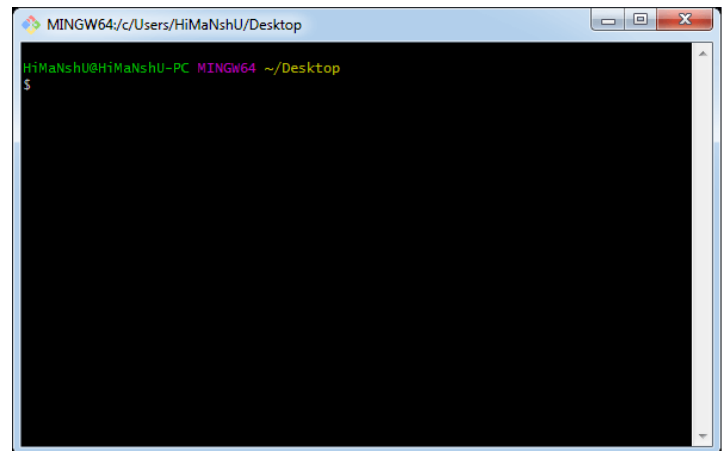
Par conséquent, l'installation de Git est terminée. Vous pouvez maintenant accéder à Git Gui et Git Bash.

Le **Git Gui** ressemble à
Il facilite avec trois fonctionnalités.

- Créer un nouveau référentiel
- Cloner le référentiel existant
- Ouvrir le référentiel existant



Le **Git Bash** ressemble à



➤ **Installer Git sur Linux**

Pour installer Git sous linux, il existe deux solutions vraiment très simples:

Si vous êtes sur un système basé sur Debian, comme par exemple Ubuntu, utiliser le gestionnaire de paquets apt-get :

Pour installer uniquement Git, exécutez la commande suivante: **\$ apt-get install git**

Pour installer Git et ses deux interfaces graphiques de base, exécutez la commande suivante:
\$ apt-get install git-all

Si vous souhaitez installer Git via un installateur d'application, vous pouvez le faire via le système de gestion de paquets de base fourni avec votre distribution. exemple avec yum sous Fedora, exécutez la commande suivante: **\$ yum install git**

➤ **Installez Git sur Ubuntu**

Git est un système de contrôle de version distribué open source qui est disponible pour tout le monde à un coût nul. Il est conçu pour gérer des projets mineurs à majeurs avec rapidité et efficacité. Il est développé pour coordonner le travail des programmeurs. Le contrôle de version vous permet de suivre et de travailler avec les membres de votre équipe dans le même espace de travail.

Git est la gestion du code source (SCM) la plus courante et couvre plus d'utilisateurs que les systèmes VCS antérieurs comme SVN. Comprenons comment installer Git sur votre serveur Ubuntu.

J'ai fait cette installation sur Ubuntu 16.04 LTS. Mais les commandes données devraient également fonctionner avec les autres versions.

Voici les étapes pour installer le Git sur le serveur Ubuntu:

Étape 1: Démarrez la mise à jour générale du système d'exploitation et du package

Tout d'abord, nous devons démarrer les mises à jour générales du système d'exploitation et des packages. Pour ce faire, exécutez la commande suivante: **\$ apt-get update**

Nous avons maintenant commencé les mises à jour générales du système d'exploitation et des packages. Après cela, nous exécuterons les mises à jour générales sur le serveur afin de pouvoir commencer à installer Git. Pour ce faire, exécutez les commandes suivantes:

Étape 2: Installez Git

Pour installer Git, exécutez la commande suivante: **\$ apt-get install git-core**

La commande ci-dessus installera le Git sur votre système, mais elle peut vous demander de confirmer le téléchargement et l'installation.

Étape 3: Confirmez l'installation de Git

Pour confirmer l'installation, appuyez sur la touche «y» de l'éditeur. Maintenant, Git est installé et prêt à être utilisé.

Une fois l'installation centrale terminée, vérifiez d'abord que le fichier exécutable est configuré et accessible. La meilleure façon de faire est la commande git version. Il sera exécuté comme: **\$ git --version**

Sur votre Terminal, vous verrez apparaître la version de votre Git comme suit: **git version 2.24.0**

Étape 4: Configurez le Git pour la première utilisation

Vous pouvez maintenant commencer à utiliser Git sur votre système. Vous pouvez explorer de nombreuses fonctionnalités du système de contrôle de version. Pour utiliser Git, vous devez configurer le processus d'accès utilisateur initial. Cela peut être fait avec la commande git config.

Supposons que je veuille enregistrer un utilisateur dont le nom d'utilisateur est "**Phoenix**" et l'adresse e-mail "**phoenix@gmail.com**", alors ce sera fait comme suit:

Pour enregistrer un nom d'utilisateur, exécutez la commande ci-dessous:

```
$ git config --global user.name "Phoenix"
```

Pour enregistrer une adresse e-mail pour l'auteur donné, exécutez la commande ci-dessous:

```
$ git config --global user.email "phoenix@gmail.com"
```

Vous avez maintenant enregistré avec succès un utilisateur pour le système de contrôle de version.

Il est important de comprendre que l'outil git config fonctionne sur un utilisateur en fonction de l'utilisateur. Par exemple, si nous avons un utilisateur "**Toure**" enregistré sur Git. Ensuite, il peut y avoir un autre utilisateur "**Mike**" sur la même machine enregistrée sur Git. Pour ce faire, **Albert** doit exécuter la même commande depuis son compte utilisateur. Les commits effectués par les deux utilisateurs seront effectués sous leurs coordonnées dans Git.

➤ Installez Git sur Mac

Il existe plusieurs façons d'installer Git sur mac. Il est intégré à Xcode ou à ses autres outils de ligne de commande. Pour démarrer le Git, ouvrez le terminal et entrez la commande ci-dessous:

```
$ git --version
```

La commande ci-dessus affichera la version installée de Git.

Production: **git version 2.24.0 (Apple Git-66)**

Si vous ne l'avez pas déjà installé, il vous demandera de l'installer.

Apple fournit un support pour Git, mais il est en retard par plusieurs versions majeures. Nous pouvons installer une version plus récente de Git en utilisant l'une des méthodes suivantes:

Installer Git pour Mac

Ce processus est le moyen le plus simple de télécharger la dernière version de Git. Visitez la [page officielle](#) des téléchargements git. Choisissez l'option de téléchargement pour **Mac OS X**.



Le fichier d'installation sera téléchargé sur votre système. Suivez les invites, choisissez l'option d'installation requise. Une fois le processus d'installation terminé, vérifiez que l'installation a réussi en exécutant la commande suivante sur le terminal: **\$ git --version**

La commande ci-dessus affichera la version installée de Git. Considérez la sortie ci-dessous.

Production: git version 2.24.0 (Apple Git-66)

Maintenant, nous avons installé avec succès la dernière version sur notre Mac OS. Il est temps de configurer le système de contrôle de version pour la première utilisation.

Pour enregistrer un nom d'utilisateur, exécutez la commande ci-dessous:

\$ git config --global user.name "Phoenix"

Pour enregistrer une adresse e-mail pour l'auteur donné, exécutez la commande ci-dessous:

\$ git config --global user.email "phoenix@gmail.com"

Installation via MacPorts

Parfois, MacPorts faisait également référence à DarwinPorts. Il facilite l'installation de logiciels sur les systèmes d'exploitation Mac OS et Darwin. Si nous avons installé MacPorts pour gérer les packages sous OS X, suivez les étapes ci-dessous pour installer Git.

Étape 1: mettre à jour MacPorts

Pour mettre à jour MacPorts, exécutez la commande suivante: **\$ sudo port selfupdate**

Étape 2: recherchez les derniers ports

Pour rechercher les ports et variantes Git disponibles les plus récents, exécutez la commande ci-dessous:

- **\$ port recherche git**
- **\$ port variantes git**

La commande ci-dessus recherchera le dernier port et les dernières options disponibles et l'installera.

Étape 3: Installez Git

Pour installer Git, exécutez la commande suivante: **\$ sudo port installer git**

Nous pouvons également installer des outils supplémentaires avec Git. Ces outils peuvent aider Git de différentes manières. Pour installer Git avec bash-completion, svn et les documents, exécutez la commande suivante: **\$ sudo port installer git + svn + doc + bash_completion + gitweb**

Maintenant, nous avons installé avec succès Git avec l'aide de MacPorts sur notre système.

Étape 4: Configurer Git

La prochaine étape pour la première utilisation est la configuration de git.

Nous allons configurer le nom d'utilisateur et l'adresse e-mail Git comme indiqué ci-dessus.

Pour enregistrer un nom d'utilisateur, exécutez la commande ci-dessous:

```
$ git config --global user.name "Phoenix"
```

Pour enregistrer une adresse e-mail pour l'auteur donné, exécutez la commande ci-dessous:

```
$ git config --global user.email "phoenix@gmail.com"
```

Installez Git via Homebrew

Homebrew est utilisé pour simplifier l'installation du logiciel. Si nous avons installé Homebrew pour gérer les packages sur OS X, suivez les étapes ci-dessous pour aller avec Git:

Étape 1: installez Git

Ouvrez le terminal et exécutez la commande ci-dessous pour installer Git en utilisant Homebrew: **\$ brew install git**

La commande ci-dessus installera le Git sur notre machine. L'étape suivante consiste à vérifier l'installation.

Étape 2: vérifier l'installation

Il est essentiel de s'assurer que le processus d'installation a réussi ou non.

Pour vérifier si l'installation a réussi ou non, exécutez la commande suivante: **\$ git --version**

La commande ci-dessus affichera la version qui a été installée sur votre système. Considérez la sortie suivante: **git version 2.24.0**

Étape 3: Configurer Git

Nous allons configurer le nom d'utilisateur et l'adresse e-mail Git comme indiqué ci-dessus.

Pour enregistrer un nom d'utilisateur, exécutez la commande ci-dessous:

```
$ git config --global user.name "Phoenix"
```

Pour enregistrer une adresse e-mail pour l'auteur donné, exécutez la commande ci-dessous:

```
$ git config --global user.email "phoenix@gmail.com"
```

Configuration

Configuration de l'environnement Git

L'environnement de tout outil se compose d'éléments qui prennent en charge l'exécution avec le logiciel, le matériel et le réseau configurés. Il comprend les paramètres du système d'exploitation, la configuration matérielle, la configuration logicielle, les terminaux de test et d'autres supports pour effectuer les opérations. C'est un aspect essentiel de tout logiciel.

Cela vous aidera à comprendre comment configurer Git pour la première utilisation sur diverses plates-formes afin que vous puissiez lire et écrire du code en un rien de temps.

La commande Git config

Git prend en charge une commande appelée **git config** qui vous permet d'obtenir et de définir des variables de configuration qui contrôlent toutes les facettes de l'apparence et du fonctionnement de Git. Il est utilisé pour définir les valeurs de configuration Git au niveau d'un projet global ou local.

Les paramètres **user.name** et **user.email** sont les options de configuration nécessaires car votre nom et votre adresse e-mail apparaîtront dans vos messages de validation.

Définition du nom d'utilisateur

Le nom d'utilisateur est utilisé par le Git pour chaque commit.

```
$ git config --global user.name "Phoenix"
```

Définition de l'identifiant de messagerie

Le Git utilise cet identifiant de messagerie pour chaque commit.

```
$ git config --global user.email "phoenix@gmail.com"
```

Il existe de nombreuses autres options de configuration que l'utilisateur peut définir.

Éditeur de paramètres

Vous pouvez définir l'éditeur de texte par défaut lorsque Git a besoin que vous saisissiez un message. Si vous n'avez sélectionné aucun des éditeurs, Git utilisera l'éditeur de votre système par défaut.

Pour sélectionner un autre éditeur de texte, tel que Vim,

```
$ git config --global core.editor Vim
```

Vérification de vos paramètres

Vous pouvez vérifier vos paramètres de configuration; vous pouvez utiliser la commande **git config --list** pour lister tous les paramètres que Git peut trouver à ce stade.

```
$ git config -list
```

Cette commande listera tous vos paramètres. Voir la sortie de ligne de commande ci-dessous.

Production

Phoenix @ Phoenix-PC MINGW64 ~ / Bureau

```
$ git config --list
core.symlinks = false
core.autocrlf = vrai
core.fscache = true
color.diff = auto
color.status = auto
color.branch = auto
color.interactive = true
help.format = html
rebase.autosquash = true
http.sslcainfo = C: / Program Files / Git / mingw64 / ssl / certs / ca-bundle.crt
http.sslbackend = openssl
diff.astextplain.textconv = astextplain
filter.lfs.clean = git-lfs nettoyer -% f
filter.lfs.smudge = git-lfs smudge --skip -% f
filter.lfs.process = processus de filtrage git-lfs --skip
filter.lfs.required = true
credential.helper = gestionnaire
gui.recentrepo = C: / Git
user.email=dav.phoenix@gmail.com
user.name = phoenix
```

Sortie colorée

Vous pouvez personnaliser votre sortie Git pour afficher un thème de couleur personnalisé. La **configuration git** peut être utilisée pour définir ces thèmes de couleurs.

Color.ui

\$ Git config -global color.ui true

La valeur par défaut de color.ui est définie sur **auto**, ce qui appliquera des couleurs au flux de sortie du terminal immédiat. Vous pouvez définir la valeur de couleur sur true, false, auto et always.

Niveaux de configuration Git

La commande git config peut accepter des arguments pour spécifier le niveau de configuration. Les niveaux de configuration suivants sont disponibles dans la configuration Git.

- local
- global
- système

--local

C'est le niveau par défaut dans Git. Git config écrira au niveau local si aucune option de configuration n'est donnée. Les valeurs de configuration locales sont stockées dans le **répertoire .git / config** sous forme de fichier.

--global

La configuration de niveau global est une configuration spécifique à l'utilisateur. Spécifique à l'utilisateur signifie, il est appliqué à un utilisateur du système d'exploitation individuel.

Les valeurs de configuration globales sont stockées dans le répertoire de base d'un utilisateur.

~ /.**gitconfig** sur les systèmes UNIX et **C: \ Users \ \ .gitconfig** sous Windows en tant que format de fichier.

--système

La configuration au niveau du système est appliquée à tout un système. L'ensemble du système signifie tous les utilisateurs d'un système d'exploitation et tous les référentiels. Le fichier de configuration au niveau du système est stocké dans un fichier gitconfig hors du répertoire système. **\$ (prefix) / etc / gitconfig** sur les systèmes UNIX et **C: \ ProgramData \ Git \ config** sur Windows.

L'ordre de priorité de la configuration Git est respectivement local, global et système. Cela signifie que lors de la recherche d'une valeur de configuration, Git démarrera au niveau local et remontera au niveau du système.

Premiers pas avec GIT

Outils Git

Pour explorer les fonctionnalités robustes de Git, nous avons besoin de quelques outils. Git est livré avec certains de ses outils comme Git Bash, Git GUI pour fournir l'interface entre la machine et l'utilisateur. Il prend en charge les outils intégrés et tiers.

Git est livré avec des outils d'interface graphique intégrés tels que **git bash**, **git-gui** et **gitk** pour la validation et la navigation. Il prend également en charge plusieurs outils tiers pour les utilisateurs à la recherche d'une expérience spécifique à la plate-forme.

Outils de package Git

Git fournit des fonctionnalités puissantes pour l'explorer. Nous avons besoin de nombreux outils tels que les commandes, la ligne de commande, l'interface graphique Git. Comprenons quelques outils de package essentiels.

Gitbash

Git Bash est une application pour l'environnement Windows. Il est utilisé comme ligne de commande Git pour Windows. Git Bash fournit une couche d'émulation pour une expérience de ligne de commande Git. Bash est une abréviation de **Bourne Again Shell**. Le programme d'installation du package Git contient Bash, les utilitaires bash et Git sur un système d'exploitation Windows.

Bash est un shell standard par défaut sur Linux et macOS. Un shell est une application de terminal utilisée pour créer une interface avec un système d'exploitation via des commandes.

Par défaut, le package Git Windows contient l'outil Git Bash. Nous pouvons y accéder par un clic droit sur un dossier dans l'Explorateur Windows.

Commandes Git Bash

Git Bash est livré avec des commandes supplémentaires qui sont stockées dans le **répertoire /usr / bin** de l'émulation Git Bash. Git Bash peut fournir une expérience de shell robuste sur Windows. Git Bash est livré avec des commandes shell essentielles telles que **Ssh**, **scp**, **cat**, **find**.

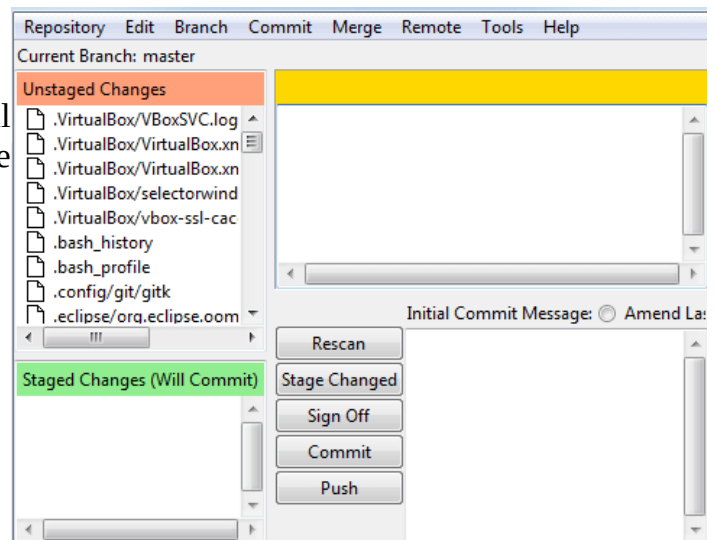
Git Bash comprend également l'ensemble complet des commandes principales de Git telles que **git clone**, **git commit**, **git checkout**, **git push**, etc.

Interface graphique Git

Git GUI est une alternative puissante à Git BASH. Il offre une version graphique de la fonction de ligne de commande Git, ainsi que des outils de comparaison visuelle complets. Nous pouvons y accéder par un simple clic droit sur un dossier ou un emplacement dans l'explorateur Windows. En outre, nous pouvons y accéder via la ligne de commande en tapant ci-dessous la commande.

\$ git gui

Une fenêtre contextuelle s'ouvrira en tant qu'outil d'interface graphique Git. L'interface de l'interface graphique Git ressemble à ceci:



Git facilite avec certains outils d'interface graphique intégrés pour la validation (git-gui) et la navigation (gitk), mais il existe de nombreux outils tiers pour les utilisateurs à la recherche d'une expérience spécifique à la plate-forme.

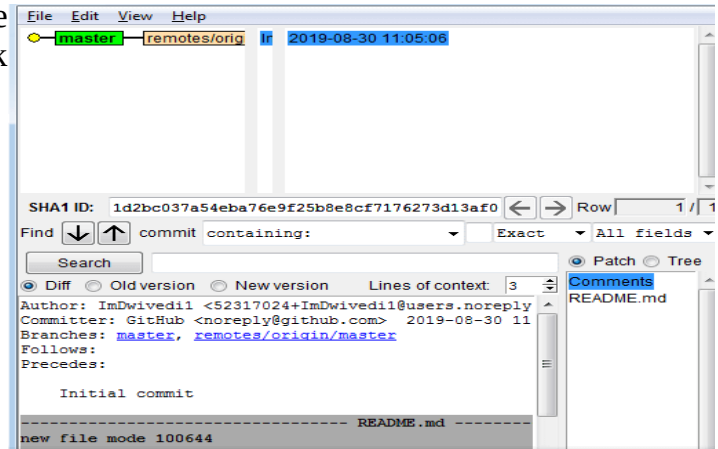
Gitk

gitk est un outil de visualisation d'historique graphique. C'est un shell GUI robuste sur git log et git grep. Cet outil est utilisé pour trouver quelque chose qui s'est passé dans le passé ou pour visualiser l'historique de votre projet.

Gitk peut appeler à partir de la ligne de commande. Changez simplement de répertoire dans un référentiel Git et tapez:

\$ gitk [options git log]

Cette commande appelle l'interface graphique de gitk et affiche l'historique du projet. L'interface Gitk ressemble à ceci:



Outils tiers Git

De nombreux outils tiers sont disponibles sur le marché pour améliorer les fonctionnalités de Git et fournir une interface utilisateur améliorée. Ces outils sont disponibles pour des plateformes distinctes telles que Windows, Mac, Linux, Android, iOS.

Voici quelque outils Git tiers populaires:

SourceTree, Bureau GitHub, TortueGit, Extensions Git, GitKraken, SmartGit, La tour, Git Up, GitEye, gitg, Git2Go, GitDrive, GitFinder, SnailGit, Pocket Git, Fusion sublime ...

Pour réaliser nos **premiers pas avec GitHub**, il faut s'assurer que les parties suivantes antérieurement vue dans ce cours ont été bien pris en compte. Pour réaliser cette activité, vous devez :

- Connaître les bases du fonctionnement de Git.
- Avoir installé Git sur votre machine.
- Posséder un compte GitHub.

Allons-y !!!

- (Optionnel) Lancez l'installation de GitHub Desktop pour votre environnement.
- Sans attendre la fin de l'installation, connectez-vous sur github.com et créez un nouveau dépôt (repository) nommé **hello-world-github** avec les paramètres ci-dessous.

Owner: bpesquet / Repository name: hello-world-github ✓

Great repository names are short and memorable. Need inspiration? How about **squealing-barnacle**.

Description (optional): My first repository!

☒ **Public**
Anyone can see this repository. You choose who can commit.

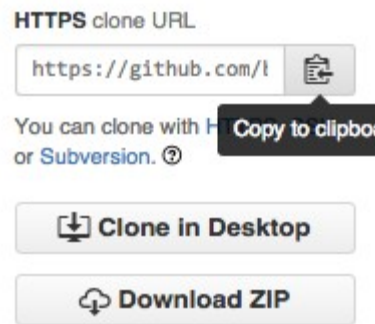
☐ **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

- Copiez l'URL du dépôt GitHub (zone en bas à droite).



Sur votre poste de travail, ouvrez un terminal puis déplacez-vous dans votre répertoire de travail.

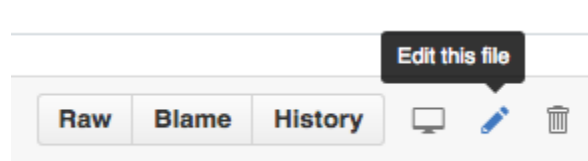
- Depuis le terminal, lancez la commande `git config --global user.email <votre courriel GitHub>`.
- Vérifiez le résultat en lançant la commande `git config --global user.email`. Elle doit maintenant afficher le courriel de votre compte GitHub.
- Depuis le terminal, clonez le dépôt avec la commande: `git clone <URL copiée>`. Un répertoire `hello-world-github` contenant le dépôt est créé.
- Dans ce répertoire, ouvrez le fichier `README.md` avec votre éditeur de texte favori et donnez-lui le contenu suivant :

hello-world-github

Ceci est mon premier dépôt GitHub.

Depuis le terminal, déplacez-vous dans ce répertoire `hello-world-github` puis committez votre modification (`git add` puis `git commit`) dans votre dépôt local avec le commentaire `Modification README`.

- Lancez la commande `git push`. Votre modification est poussée sur votre dépôt GitHub.
- Depuis `github.com`, cliquez sur le fichier `README.md` puis cliquez sur le bouton d'édition.



Ajoutez au fichier le contenu suivant : **Mais pas le dernier !**

- Committez votre modification avec le commentaire `Ajout d'une ligne`.
- Depuis le terminal, lancez la commande `git pull` pour récupérer la modification depuis GitHub.
- Lancez la commande `git log` pour afficher l'historique des modifications.
- Depuis `github.com`, modifiez le fichier `README.md` de la manière suivante.

hello-world-github BIS

- Commitez vos modifications avec le commentaire “Modification titre GitHub”.
- Avant d’effectuer un `git pull`, modifiez le fichier `README.md` sur votre machine locale de la manière suivante.

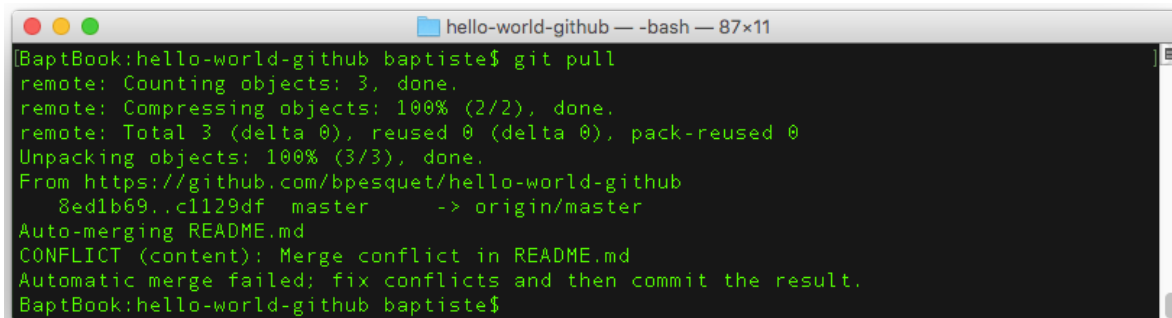
hello-world-github TER

- Commitez vos modifications dans votre dépôt local avec le commentaire “Modification titre locale”.
- Tentez de pusher votre modification locale vers GitHub. Vous obtenez un message d’erreur : vous devez d’abord intégrer les modifications faites sur le dépôt GitHub.

A terminal window titled 'hello-world-github — -bash — 87x10'. The user runs 'git push'. The output shows the push attempt to 'https://github.com/bpesquet/hello-world-github.git' being rejected. The error message states: 'error: failed to push some refs to 'https://github.com/bpesquet/hello-world-github.git''. The hint explains that updates were rejected because the remote contains work that the user does not have locally, suggesting a 'git pull' before pushing again.

```
[BaptBook:hello-world-github baptiste$ git push
To https://github.com/bpesquet/hello-world-github.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/bpesquet/hello-world-github.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
BaptBook:hello-world-github baptiste$
```

- Effectuez un `git pull` pour récupérer localement les modifications du dépôt GitHub : un conflit est apparu sur le fichier `README.md`, qui a été modifié des deux côtés et qui ne peut pas être fusionné automatiquement par Git. Une intervention manuelle est nécessaire.

A terminal window titled 'hello-world-github — -bash — 87x11'. The user runs 'git pull'. The output shows the pull operation from 'https://github.com/bpesquet/hello-world-github' (commit 8ed1b69) to the local 'origin/master' branch. It indicates that 'Auto-merging README.md' failed due to a 'CONFLICT (content): Merge conflict in README.md'. The message instructs the user to 'fix conflicts and then commit the result'.

```
[BaptBook:hello-world-github baptiste$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/bpesquet/hello-world-github
 8ed1b69..c1129df master    -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
BaptBook:hello-world-github baptiste$
```

- Ouvrez le fichier `README.md` avec un éditeur de texte. Les zones en conflit sont délimitées par des marqueurs `<<<<<<<` et `>>>>>>>`. La zone `HEAD` correspond à la modification faite localement. L’autre zone correspond à la modification réalisée sur `github.com`.

```
<<<<<<< HEAD # hello-world-github TER ===== # hello-world-github BIS >>>>>>>
c1129dfbbe585fc94978be38625b5ae7f63474bf
```

- Résolvez le conflit en modifiant le titre et en supprimant les marques de conflit dans le fichier.

hello-world-github FINAL

- Faites un `git add` pour indiquer la résolution puis committez le fichier fusionné avec le commentaire “Résolution conflit”.
- Poussez les modifications sur GitHub pour voir apparaître le nouveau titre.
- (Optionnel) Lorsque l’installation de GitHub Desktop est terminée, lancez-le puis choisissez d’ajouter un nouvel dépôt (*Add repository*). Sélectionnez le répertoire `hello-world-github` puis validez.
- (Optionnel) Observez l’historique des modifications sur le dépôt avec GitHub Desktop.
- (Optionnel) Ajoutez localement une nouvelle ligne de votre choix au fichier README.md.
- (Optionnel) Depuis GitHub Desktop, committez votre modification avec un commentaire puis cliquez sur le bouton Sync en haut à droite. La nouvelle modification est poussée sur GitHub.

REPOSITORY GIT

Définition

Les référentiels dans GIT (Repository Git) contiennent une collection de fichiers de différentes versions d'un projet. Ces fichiers sont importés du référentiel dans le serveur local de l'utilisateur pour d'autres mises à jour et modifications du contenu du fichier.

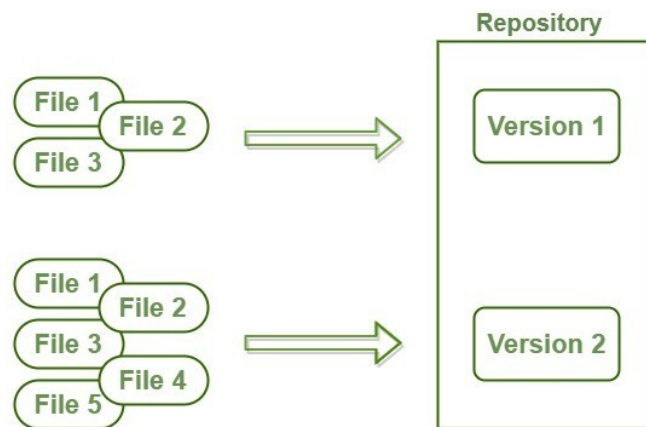
Un **VCS** ou le **système de contrôle de version** est utilisé pour **créer** ces versions et les stocker dans un endroit spécifique appelé référentiel. Le processus de copie du contenu d'un référentiel Git existant à l'aide de divers outils Git est appelé **clonage**. Une fois le processus de clonage terminé, l'utilisateur obtient le référentiel complet sur sa machine locale. Git suppose par défaut que le travail à effectuer sur le référentiel se fait en tant qu'utilisateur, une fois le clonage terminé.

Les utilisateurs peuvent également créer un nouveau référentiel ou supprimer un référentiel existant. Pour supprimer un référentiel, la méthode la plus simple consiste simplement à supprimer le dossier contenant le référentiel.

Les référentiels peuvent être divisés en deux types en fonction de l'utilisation sur un serveur. Ceux-ci sont:

Référentiels nus: ces référentiels sont utilisés pour partager les modifications effectuées par différents développeurs. Un utilisateur n'est pas autorisé à modifier ce référentiel ou à créer une nouvelle version pour ce référentiel en fonction des modifications effectuées.

Référentiels non dénudés: les référentiels non dénudés sont conviviaux et permettent donc à l'utilisateur de créer de nouvelles modifications de fichiers et de créer également de nouvelles versions pour les référentiels. Le processus de clonage par défaut crée un référentiel non nu si aucun paramètre n'est spécifié pendant l'opération de clonage.



Zone de travail ou mise en place d'un référentiel Git

Une arborescence de travail dans un référentiel Git est la collection de fichiers qui proviennent d'une certaine version du référentiel. Il aide à suivre les modifications effectuées par un utilisateur spécifique sur une version du référentiel. A chaque fois qu'une opération est validée par l'utilisateur, Git recherchera uniquement les fichiers présents dans la zone de travail, et non

tous les fichiers modifiés. Seuls les fichiers présents dans la zone de travail sont pris en compte pour l'opération de validation.

L'utilisateur de l'arborescence de travail peut modifier les fichiers en modifiant les fichiers existants et en supprimant ou en créant des fichiers.

Il y a quelques étapes d'un fichier dans l'arborescence de travail d'un référentiel:

Non suivi: à ce stade, le référentiel Git est incapable de suivre le fichier, ce qui signifie que le fichier n'est jamais mis en scène ni validé ou commité.

Suivi: lorsque le référentiel Git suit un fichier, ce qui signifie que le fichier est validé mais n'est pas stocké dans le répertoire de travail.

Staged: à ce stade, le fichier est prêt à être validé et est placé dans la zone de staging en attente de la prochaine validation.

Modifié / sale: lorsque les modifications sont apportées au fichier, c'est-à-dire que le fichier est modifié mais que le changement n'est pas encore mis en scène.

Une fois les modifications effectuées dans la zone de travail, l'utilisateur peut mettre à jour ces modifications dans le référentiel GIT ou annuler les modifications.

Travailler avec un référentiel

Un référentiel GIT permet d'effectuer diverses opérations dessus pour créer différentes versions d'un projet. Ces opérations incluent l'ajout de fichiers, la création de nouveaux référentiels, la validation d'une action, la suppression d'un référentiel, etc. Ces modifications entraîneront la création de différentes versions d'un projet.

Ajout à un référentiel

Après avoir effectué diverses modifications sur un fichier dans la zone de travail, GIT doit suivre deux étapes supplémentaires pour enregistrer ces modifications dans le référentiel local. Ces étapes sont:

- Ajout des modifications à l'index (zone de transit)
- Validation des modifications indexées dans le référentiel

Ajout de modifications à l'index

Ce processus est effectué à l'aide de la commande **git add**. Lorsque les modifications ont été apportées dans l'arborescence/la zone de travail. Ces modifications doivent être ajoutées à la zone de transit pour une modification ultérieure du fichier. La commande git add ajoute le fichier dans le référentiel local. Cela les met en scène pour le processus de validation.

Syntaxe:

\$ git add Nom-fichier

Différentes façons d'utiliser la commande add:

\$ git add

- Pour ajouter une liste spécifique de fichiers à la zone de préparation.
\$ **git add - -all**
- Pour ajouter tous les fichiers du répertoire actuel à la zone de préparation.
\$ **git add *.txt**
- Pour ajouter tous les fichiers texte du répertoire actuel à la zone de préparation.
\$ **git add docs/*.txt**
- Pour ajouter tous les fichiers texte d'un répertoire particulier (docs) à la zone de préparation.
\$ **git add docs/**
- Pour ajouter tous les fichiers d'un répertoire particulier (docs) à la zone de préparation.
\$ **git add "**.txt"**

Pour ajouter des fichiers texte de l'ensemble du projet à la zone de préparation.

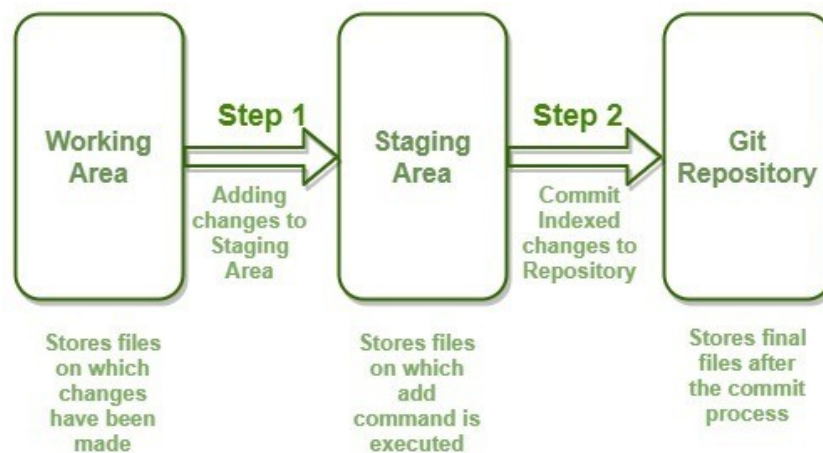
Validation des modifications à partir de l'index

Le processus de validation est effectué dans la zone de préparation sur les fichiers qui sont ajoutés à l'index après l'exécution de la commande **git add**. Ce processus de validation est effectué à l'aide de la commande **git commit**. Cette commande valide les modifications par étapes dans le référentiel local.

Syntaxe:

\$ **git commit -m "Add existing file"**

Cette commande de validation est utilisée pour ajouter l'un des fichiers suivis à la zone de préparation et les valider en fournissant un message à mémoriser.



Synchronisation avec les référentiels distants

Git permet aux utilisateurs d'effectuer des opérations sur les référentiels en les clonant sur la machine locale. Cela se traduira par la création de différentes copies du projet. Ces copies sont stockées sur la machine locale et, par conséquent, les utilisateurs ne pourront pas synchroniser leurs modifications avec d'autres développeurs. Pour surmonter ce problème, Git permet d'effectuer la synchronisation de ces référentiels locaux avec les référentiels distants.

Cette synchronisation peut être effectuée par l'utilisation de deux commandes dans le Git. Ces commandes sont:

- Push (pousser)
- Pull (tirer)

Push: Cette commande est utilisée pour pousser tous les commits du référentiel actuel vers le référentiel distant suivi. Cette commande peut être utilisée pour pousser votre référentiel vers plusieurs référentiels à la fois.

Syntaxe:

\$ git push -u origin master

Pour pousser tout le contenu de notre référentiel local qui appartient à la branche maître vers le serveur (référentiel global).

Pull: La commande Pull est utilisée pour récupérer les validations d'un référentiel distant et les stocke dans les branches distantes. Il peut arriver que d'autres utilisateurs modifient leur copie de référentiels et les téléchargent avec d'autres référentiels distants. Mais dans ce cas, votre copie du référentiel deviendra obsolète. Par conséquent, pour resynchroniser votre copie du référentiel avec le référentiel distant, l'utilisateur doit simplement utiliser la commande git pull pour récupérer le contenu du référentiel distant.

Syntaxe:

\$ git pull

Création

Git Init

La commande **git init** est la première commande que vous exécuterez sur Git. La commande git init est utilisée pour créer un nouveau référentiel vide. Il est utilisé pour créer un projet existant en tant que projet Git. Plusieurs commandes Git s'exécutent à l'intérieur du référentiel, mais la commande init peut être exécutée en dehors du référentiel.

La commande git init crée un sous-répertoire **.git** dans le répertoire de travail actuel. Ce sous-répertoire nouvellement créé contient toutes les métadonnées nécessaires. Ces métadonnées peuvent être classées en objets, références et fichiers temporaires. Il initialise également un pointeur HEAD pour la branche maître du référentiel.

Création du premier référentiel

Le système de contrôle de version Git vous permet de partager des projets entre les développeurs. Pour apprendre Git, il est essentiel de comprendre comment créer un projet sur Git. Un référentiel est un répertoire qui contient toutes les données relatives au projet. Il peut également y avoir plusieurs projets sur un même référentiel.

Nous pouvons créer un référentiel pour les projets vierges et existants. Comprenons comment créer un référentiel.

Créez un référentiel pour un projet vierge (nouveau):

Pour créer un référentiel vide, ouvrez la ligne de commande sur le répertoire souhaité et exécutez la commande init comme suit: **\$ git init**

La commande ci-dessus créera un référentiel **.git** vide. Supposons que nous voulions créer un référentiel git sur notre bureau. Pour ce faire, ouvrez Git Bash sur le bureau et exécutez la commande ci-dessus. Considérez la sortie ci-dessous:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git init
Initialized empty Git repository in C:/Users/HiMaNshU/Desktop/.git/

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

La commande ci-dessus initialisera un référentiel **.git** sur le bureau. Nous pouvons maintenant créer et ajouter des fichiers sur ce référentiel pour le contrôle de version.

Pour créer un fichier, exécutez la commande **cat** ou **touch** comme suit:

\$ touch <nom du fichier>

Pour ajouter des fichiers au référentiel, exécutez la commande **git add** comme suit:

\$ git add <nom de fichier>

Créer un référentiel pour un projet existant

Si vous souhaitez partager votre projet sur un système de contrôle de version et le contrôler avec Git, parcourez le répertoire de votre projet et démarrez la ligne de commande git (Git Bash pour Windows) ici. Pour initialiser un nouveau référentiel, exécutez la commande ci-dessous:

Syntaxe: **\$ git init**

Production:

```
HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project
$ git init
Initialized empty Git repository in C:/My Project/.git/

HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project (master)
$ |
```

La commande ci-dessus créera un nouveau sous-répertoire nommé **.git** contenant tous les fichiers de référentiel nécessaires. Le sous-répertoire .git peut être compris comme un squelette de référentiel Git. Considérez l'image ci-dessous:

.git	10/12/2019 4:04 PM	File folder	
design	9/19/2019 6:10 PM	Cascading Style S...	1 KB
design2	10/6/2019 5:21 PM	Cascading Style S...	1 KB
index	9/19/2019 6:10 PM	JSP File	2 KB
master	9/19/2019 6:10 PM	JSP File	1 KB
merge the branch	9/20/2019 6:05 PM	File	1 KB
newfile	10/4/2019 2:10 PM	Text Document	1 KB
newfile1	10/4/2019 2:10 PM	Text Document	1 KB
newfile2	10/9/2019 12:26 PM	Text Document	0 KB
README	9/19/2019 6:10 PM	MD File	1 KB

Un référentiel vide .git est ajouté à mon projet existant. Si nous voulons démarrer le contrôle de version pour les fichiers existants, nous devons suivre ces fichiers avec la commande git add, suivie d'un commit.

Nous pouvons lister tous les fichiers non suivis par la commande git status.

\$ git status

Considérez la sortie ci-dessous:

```
HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md
    design.css
    design2.css
    index.jsp
    master.jsp
    merge the branch
    newfile.txt
    newfile1.txt
    newfile2.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Dans la sortie ci-dessus, la liste de tous les fichiers non suivis est affichée par la commande git status.

Nous pouvons suivre tous les fichiers non suivis par la commande Git Add.

Créer un référentiel et un répertoire ensemble

La commande git init nous permet de créer un nouveau référentiel vide et un répertoire ensemble. Le référentiel vide .git est créé sous le répertoire. Supposons que je veuille créer un référentiel vide avec un nom de projet, alors nous pouvons le faire par la commande git init. Considérez la commande suivante: **\$ git init NewDirectory**

La commande ci-dessus créera un référentiel .git vide sous un répertoire nommé **NewDirectory**. Considérez la sortie ci-dessous:

```
HiManShU@HiManShU-PC MINGW64 ~/Desktop (master)
$ git init
Initialized empty Git repository in C:/Users/HiManShU/Desktop/.git/

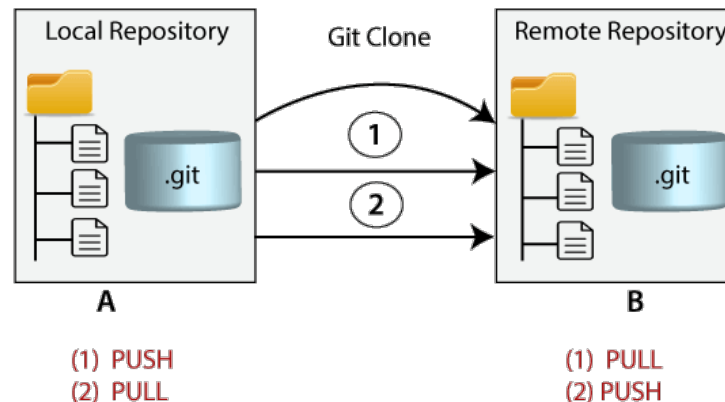
HiManShU@HiManShU-PC MINGW64 ~/Desktop (master)
$ git init NewDirectory
Initialized empty Git repository in C:/Users/HiManShU/Desktop/NewDirectory/.git/
```

Dans la sortie ci-dessus, le répertoire et le référentiel sont tous deux créés. Par conséquent, nous pouvons créer un référentiel à l'aide de la commande `git init`.

Clonage d'un dépôt

Clone Git

Dans Git, le clonage consiste à créer une copie de n'importe quel référentiel cible. Le référentiel cible peut être distant ou local. Vous pouvez cloner votre référentiel à partir du référentiel distant pour créer une copie locale sur votre système. En outre, vous pouvez synchroniser entre les deux emplacements.



Commande Git Clone

Le **clone git** est un utilitaire de ligne de commande qui est utilisé pour faire une copie locale d'un référentiel distant. Il accède au référentiel via une URL distante.

Habituellement, le référentiel d'origine est situé sur un serveur distant, souvent à partir d'un service Git comme GitHub, Bitbucket ou GitLab. L'URL du référentiel distant fait référence à l'**origine**.

Syntaxe: `$ git clone<URL du référentiel>`

Dépôt de clones Git

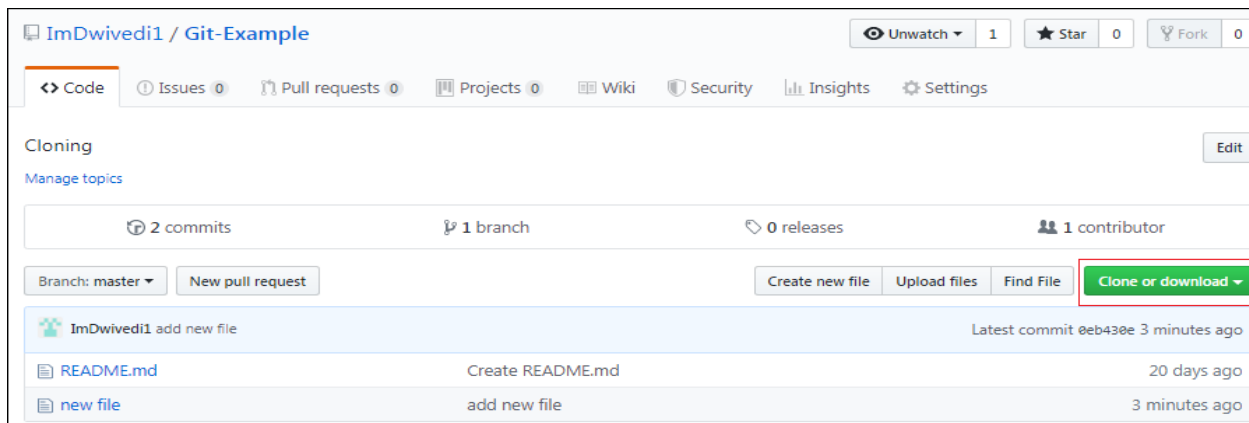
Supposons que vous souhaitiez cloner un référentiel à partir de GitHub, ou avoir un référentiel existant appartenant à tout autre utilisateur auquel vous souhaitez contribuer. Les étapes pour cloner un référentiel sont les suivantes:

Étape 1:

Ouvrez GitHub et accédez à la page principale du référentiel.

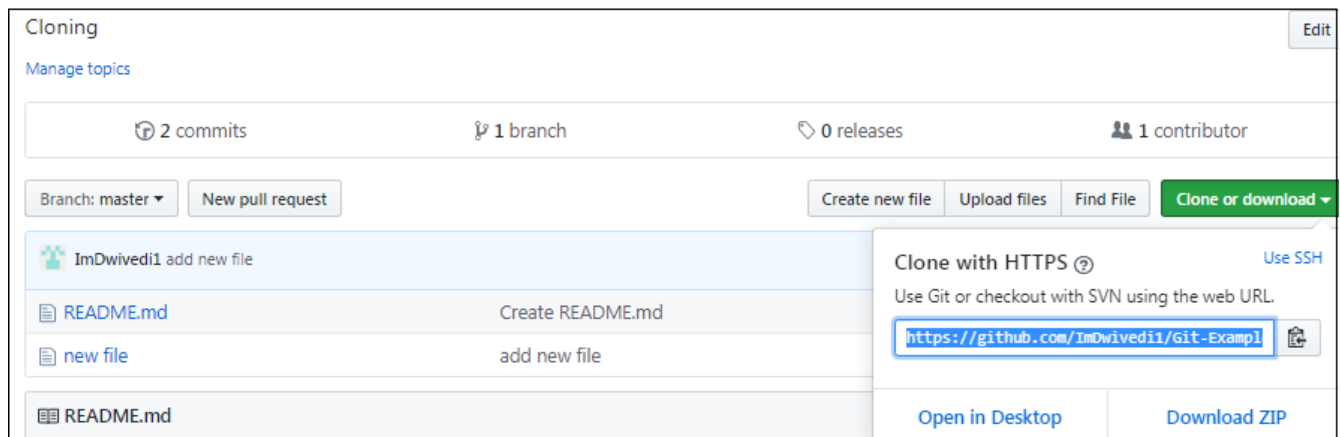
Étape 2:

Sous le nom du référentiel, cliquez sur **Cloner** ou **télécharger**.



Étape 3:

Sélectionnez la section **Cloner avec HTTP** et copiez l'**URL de clonage** du référentiel. Pour le référentiel vide, vous pouvez copier l'URL de la page du référentiel à partir de votre navigateur et passer à l'étape suivante.



Étape 4:

Ouvrez Git Bash et modifiez le répertoire de travail actuel vers l'emplacement souhaité où vous souhaitez créer la copie locale du référentiel.

Étape 5:

Utilisez la commande git clone avec l'URL du référentiel pour créer une copie du référentiel distant. Voir la commande ci-dessous:

\$ git clone https://github.com/ImDwivedi1/Git-Example.git

Maintenant, appuyez sur Entrée. Par conséquent, votre référentiel cloné local sera créé. Voir la sortie ci-dessous:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ cd "new folder"

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder (master)
$
```

Clonage d'un référentiel dans un dossier local spécifique

Git permet de cloner le référentiel dans un répertoire spécifique sans basculer vers ce répertoire particulier. Vous pouvez spécifier ce répertoire comme argument de ligne de commande suivant dans la commande git clone. Voir la commande ci-dessous:

\$ git clone https://github.com/ImDwivedi1/Git-Example.git "nouveau dossier (2)"

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"
Cloning into 'new folder(2)'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

La commande donnée fait la même chose que la précédente, mais le répertoire cible est basculé vers le répertoire spécifié.

Git a un autre protocole de transfert appelé protocole SSH. L'exemple ci-dessus utilise le protocole git: //, mais vous pouvez également utiliser http (s): // ou user @ server: /path.git, qui utilise le protocole de transfert SSH.

Branche de clone Git

Git permet de ne faire une copie que d'une branche particulière à partir d'un référentiel. Vous pouvez créer un répertoire pour la branche individuelle en utilisant la commande git clone. Pour créer une branche clonée, vous devez spécifier le nom de la branche avec la commande

-b. Voici la syntaxe de la commande pour cloner la branche git spécifique:

Syntaxe: \$ git clone -b< Nom de la **branche** > <URL du **référentiel**>

Voir la commande ci-dessous:

```
$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git "nouveau dossier (2)"
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ |
```

Dans la sortie donnée, seule la branche master est clonée à partir du référentiel principal Git-Example.

Récupération d'un dépôt existant

Obtention et création des projets

Il y a deux façons de récupérer un dépôt Git. L'une consiste à le **copier d'un dépôt** existant sur le réseau ou ailleurs et l'autre est d'en **créer un nouveau** dans un dossier existant.

git init

Pour transformer un dossier en un nouveau dépôt Git afin que vous puissiez commencer sa gestion de version, vous pouvez simplement lancer la commande **git init**.

git clone

La commande **git clone** sert en fait à englober plusieurs autres commandes. Elle crée un nouveau dossier, va à l'intérieur de celui-ci et lance **git init** pour en faire un dépôt Git vide, ajoute un serveur distant (**git remote add**) à l'URL que vous lui avez passée (appelé par défaut **origin**), lance **git fetch** à partir de ce dépôt distant et ensuite extrait le dernier **commit** dans votre répertoire de travail avec **git checkout**.

Bon à savoir: **git init** et **git clone** sont facilement confondues. À un niveau supérieur, les deux commandes permettent d'« initialiser un nouveau dépôt Git ». Toutefois, **git clone** dépend de **git init**. **git clone** permet de créer une copie d'un dépôt existant. En interne, **git clone** appelle d'abord **git init** pour créer un dépôt. Elle copie ensuite les données du dépôt existant, puis fait un **checkout** d'un nouvel ensemble de fichiers de travail.

Mise à jour, export d'un dépôt

Partage et mise à jour de projets

Il n'y a pas vraiment beaucoup de commandes dans Git qui accèdent au réseau; presque toutes les commandes agissent sur la base de données locale. Quand vous êtes prêt à partager votre travail ou à tirer les changements depuis ailleurs, il y a une poignée de commandes qui échangent avec les dépôts distants.

git fetch

La commande **git fetch** communique avec un dépôt distant et rapporte toutes les informations qui sont dans ce dépôt qui ne sont pas dans le vôtre et les stocke dans votre base de données locale.

Récupérer et tirer depuis des dépôts distants

Comme vous venez tout juste de le voir, pour obtenir les données des dépôts distants, vous pouvez lancer: **\$ git fetch [remote-name]**

Cette commande s'adresse au dépôt distant et récupère toutes les données de ce projet que vous ne possédez pas encore. Après cette action, vous possédez toutes les références à toutes les branches contenues dans ce dépôt, que vous pouvez fusionner ou inspecter à tout moment.

Si vous clonez un dépôt, le dépôt distant est automatiquement ajouté sous le nom «**origin**».

Donc, **git fetch origin** récupère tout ajout qui a été poussé vers ce dépôt depuis que vous l'avez cloné ou la dernière fois que vous avez récupéré les ajouts. Il faut noter que la commande **fetch** tire les données dans votre dépôt local mais sous sa propre branche.

Elle ne les fusionne pas automatiquement avec aucun de vos travaux ni ne modifie votre copie de travail. Vous devez volontairement fusionner ses modifications distantes dans votre travail lorsque vous le souhaitez.

Si vous avez créé une branche pour suivre l'évolution d'une branche distante, vous pouvez utiliser la commande **git pull** qui récupère et fusionne automatiquement une branche distante dans votre branche locale. Ce comportement peut correspondre à une méthode de travail plus confortable, sachant que par défaut la commande **git clone** paramètre votre branche locale pour qu'elle suive la branche **master** du dépôt que vous avez cloné (en supposant que le dépôt distant ait une branche **master**). Lancer **git pull** récupère généralement les données depuis le serveur qui a été initialement cloné et essaie de les fusionner dans votre branche de travail actuel.

Scénario 1: pour récupérer le référentiel distant

Nous pouvons récupérer le référentiel complet à l'aide de la commande **fetch** à partir d'une URL de référentiel comme le fait une commande **pull**. Voir la sortie ci-contre:

Syntaxe: \$ git fetch <URL du dépôt>

Production:

Dans la sortie ci-dessus, le référentiel complet a été extrait d'une URL distante.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch https://github.com/ImDwivedi1/Git-Example.git
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/ImDwivedi1/Git-Example
* branch          HEAD       -> FETCH_HEAD
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ |
```

Scénario 2: Pour récupérer une branche spécifique
Nous pouvons récupérer une branche spécifique à partir d'un référentiel. Il n'accédera à l'élément qu'à partir d'une branche spécifique. Voir la sortie ci-contre:
Syntaxe: `$ git fetch <URL de la branche> <nom de la branche>`

Production:

```
HiManShU@HiManShU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch https://github.com/ImDwivedi1/Git-Example.git Test
warning: no common commits
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch                Test      -> FETCH_HEAD
$ |
```

Dans la sortie donnée, le test de branche spécifique a été extrait d'une URL distante.

Scénario 3: Pour récupérer toutes les branches simultanément:

La commande `git fetch` permet de récupérer toutes les branches simultanément à partir d'un référentiel distant. Voir l'exemple ci-contre:

Syntaxe: `$ git fetch -all`

Production:

```
HiManShU@HiManShU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch --all
Fetching origin
From https://github.com/ImDwivedi1/Git-Example
 * [new branch]      master    -> origin/master
 * [new branch]      Test      -> origin/Test
$ |
```

Dans la sortie ci-dessus, toutes les branches ont été extraites du référentiel Git-Example.

Scénario 4: pour synchroniser le référentiel local:

Supposons que le membre de votre équipe ait ajouté de nouvelles fonctionnalités à votre référentiel distant. Donc, pour ajouter ces mises à jour à votre référentiel local, utilisez la commande `git fetch`. Il est utilisé comme suit.

Syntaxe: `$ git fetch origin`

Production:

```
HiManShU@HiManShU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin
HiManShU@HiManShU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
 * [new branch]      test2      -> origin/test2
$ |
```

Dans la sortie ci-dessus, les nouvelles fonctionnalités du référentiel distant ont été mises à jour sur mon système local. Dans cette sortie, la branche **test2** et ses objets sont ajoutés au référentiel local.

Le **git fetch** peut récupérer soit à partir d'un seul référentiel ou URL nommé soit à partir de plusieurs référentiels à la fois. Il peut être considéré comme la version sûre des commandes `git pull`.

Git fetch télécharge le contenu distant mais ne met pas à jour l'état de fonctionnement de votre dépôt local. Lorsqu'aucun serveur distant n'est spécifié, par défaut, il récupère la télécommande d'origine.

Différences entre git fetch et git pull

Pour comprendre les différences entre fetch et pull, connaissons les similitudes entre ces deux commandes. Les deux commandes sont utilisées pour télécharger les données à partir d'un référentiel distant. Mais ces deux commandes fonctionnent différemment. Comme lorsque vous effectuez un pull git, il obtient toutes les modifications du référentiel distant ou central et les met à disposition de votre branche correspondante dans votre référentiel local. Lorsque vous effectuez une extraction git, il récupère toutes les modifications du référentiel distant et les stocke dans une branche distincte de votre référentiel local. Vous pouvez refléter ces changements dans vos branches correspondantes en fusionnant.

Donc en gros,

git pull = git fetch + git merge

git pull

La commande **git pull** est essentiellement une combinaison des commandes **git fetch** et **git merge**, où Git ira chercher les modifications depuis le dépôt distant que vous spécifiez et essaie immédiatement de les fusionner dans la branche dans laquelle vous vous trouvez.

git push

La commande **git push** est utilisée pour communiquer avec un autre dépôt, calculer ce que votre base de données locale a et que le dépôt distant n'a pas, et ensuite pousser la différence dans l'autre dépôt. Cela nécessite un droit d'écriture sur l'autre dépôt et donc normalement de s'authentifier d'une manière ou d'une autre.

git remote

La commande **git remote** est un outil de gestion pour votre base de dépôts distants. Elle vous permet de sauvegarder de longues URLs en tant que raccourcis, comme «**origin**», pour que vous n'ayez pas à les taper dans leur intégralité tout le temps. Vous pouvez en avoir plusieurs et la commande **git remote** est utilisée pour les ajouter, les modifier et les supprimer.

Elle est aussi utilisée dans presque tous les chapitres suivants du livre, mais toujours dans le format standard **git remote add <nom> <URL>**.

Vérifiez votre télécommande

Pour vérifier la configuration du serveur distant, exécutez la commande **git remote**. La commande git remote permet d'accéder à la connexion entre distant et local. Si vous voulez voir l'existence d'origine de votre référentiel cloné, utilisez la commande git remote. Il peut être utilisé comme:

Syntaxe: **\$ git remote**

Production:

```
HiManShU@HiManShU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote
origin
```

La commande donnée fournit le nom distant comme origine. Origin est le nom par défaut du serveur distant, qui est donné par Git.

Git remote -v:

Git remote prend en charge une option spécifique -v pour afficher les URL que Git a stockées sous forme de nom court. Ces noms courts sont utilisés lors de l'opération de lecture et d'écriture. Ici, -v signifie **verbeux**. Nous pouvons utiliser **--verbose** à la place de -v. Il est utilisé comme:

Syntaxe: \$ git remote -v

Ou alors

\$ git remote --verbose

Production:

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
origin https://github.com/ImDwivedi1/GitExample2.git (fetch)
origin https://github.com/ImDwivedi1/GitExample2.git (push)
```

La sortie ci-dessus fournit des connexions à distance disponibles. Si un référentiel contient plusieurs connexions distantes, cette commande les répertorie toutes.

Ajouter à distance Git

Lorsque nous récupérons un référentiel implicitement, git ajoute une télécommande pour le référentiel. De plus, nous pouvons ajouter explicitement une télécommande pour un référentiel. Nous pouvons ajouter une télécommande comme surnom ou nom court. Pour ajouter remote comme nom court, suivez la commande ci-dessous:

Syntaxe: \$ git remote add <nom court> <URL distante>

Production:

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$ git remote add hd https://github.com/ImDwivedi1/hello-world

HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$ git remote -v
hd      https://github.com/ImDwivedi1/hello-world (fetch)
hd      https://github.com/ImDwivedi1/hello-world (push)

HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$
```

Dans la sortie ci-dessus, j'ai ajouté un référentiel distant avec un référentiel existant comme nom court "**hd**". Maintenant, vous pouvez utiliser "**hd**" sur la ligne de commande à la place de l'URL entière. Par exemple, vous souhaitez extraire le référentiel, considérez la sortie ci-dessous:

J'ai extrait un référentiel en utilisant son nom court au lieu de son URL distante. Désormais, la branche principale du référentiel est accessible via un nom court.

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$ git pull hd
warning: no common commits
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), done.
From https://github.com/ImDwivedi1/hello-world
* [new branch]      master    -> hd/master
You asked to pull from the remote 'hd', but did not specify
a branch. Because this is not the default configured remote
for your current branch, you must specify a branch on the command line
.
```


Récupération et extraction de la branche distante

Vous pouvez récupérer et extraire des données du référentiel distant. La commande d'extraction et d'extraction est envoyée à ce serveur distant et récupère toutes les données de ce projet distant que vous n'avez pas encore. Ces commandes nous permettent de récupérer les références à toutes les branches de cette télécommande.

Pour **récupérer les données** de vos projets distants, exécutez la commande ci-dessous:

```
$ git fetch <remote>
```

Pour **cloner le référentiel** distant à partir de vos projets distants, exécutez la commande ci-dessous:

```
$ git clone <remote>
```

Lorsque nous clonons un référentiel, le référentiel distant est ajouté par un nom par défaut «origine». Donc, la plupart du temps, la commande est utilisée comme origine git fetch.

L'origine git fetch récupère les mises à jour qui ont été apportées au serveur distant depuis que vous l'avez cloné. La commande git fetch télécharge uniquement les données dans le référentiel local; il ne fusionne ni ne modifie les données tant que vous n'opérez pas. Vous devez le fusionner manuellement dans votre référentiel quand vous le souhaitez.

Pour extraire le référentiel, exécutez la commande ci-dessous:

```
$ git pull <remote>
```

La commande git pull récupère automatiquement puis fusionne les données distantes dans votre branche actuelle. La traction est un flux de travail plus facile et confortable que la récupération. Parce que la commande git clone configure votre branche maître locale pour suivre la branche maître distante sur le serveur que vous avez cloné.

Pousser vers une branche distante

Si vous souhaitez partager votre projet, vous devez le pousser en amont. La commande git push est utilisée pour partager un projet ou envoyer des mises à jour au serveur distant. Il est utilisé comme:

```
$ git push <remote> <branch>
```

Pour mettre à jour la branche principale du projet, utilisez la commande ci-dessous:

```
$ git push origin master
```

Il s'agit d'un utilitaire de ligne de commande spécial qui spécifie la branche et le répertoire distants. Lorsque vous avez plusieurs branches sur un serveur distant, cette commande vous aide à spécifier votre branche principale et votre référentiel.

Généralement, le terme **origin** désigne le référentiel distant et **master** est considéré comme la branche principale. Ainsi, l'instruction entière "**git push origin master**" a poussé le contenu local sur la branche master de l'emplacement distant.

Git Remove Remote

Vous pouvez supprimer une connexion à distance d'un référentiel. Pour supprimer une connexion, exécutez la commande `git remote` avec l'option **remove** ou **rm** . Cela peut être fait comme:

Syntaxe:

\$ git remote rm <destination>

Ou alors

\$ git remote remove <destination>

Prenons l'exemple ci-dessous:

Supposons que vous soyez connecté à un serveur distant par défaut «origin». Pour vérifier la télécommande en détail, exécutez la commande ci-dessous:

\$ git remote -v

Production:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
origin https://github.com/ImDwivedi1/GitExample2.git (fetch)
origin https://github.com/ImDwivedi1/GitExample2.git (push)
```

La sortie ci-dessus répertorie le serveur distant disponible. Maintenant, effectuez l'opération de suppression comme mentionné ci-dessus. Considérez la sortie ci-dessous:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote rm origin

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

Dans la sortie ci-dessus, j'ai supprimé le serveur distant "**origin**" de mon référentiel.

Renommer à distance Git

Git permet de renommer le nom du serveur distant afin que vous puissiez utiliser un nom court à la place du nom du serveur distant. La commande ci-dessous est utilisée pour renommer le serveur distant:

Syntaxe:

\$ git remote renommer <ancien nom> <nouveau nom>

Production:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote rename origin hd

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
hd https://github.com/ImDwivedi1/GitExample2 (fetch)
hd https://github.com/ImDwivedi1/GitExample2 (push)
```

Dans la sortie ci-dessus, j'ai renommé l'origine du nom de mon serveur par défaut en hd. Maintenant, je peux opérer en utilisant ce nom à la place de l'origine. Considérez la sortie ci-dessous:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git pull hd master
From https://github.com/ImDwivedi1/GitExample2
 * branch          master      -> FETCH_HEAD
 * [new branch]     master      -> hd/master
Already up to date.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git pull origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Dans la sortie ci-dessus, j'ai extrait le référentiel distant en utilisant le nom de serveur hd. Mais, lorsque j'utilise l'ancien nom de serveur, il génère une erreur avec le message " **'origin' ne semble pas être un référentiel git.**" Cela signifie que Git n'identifie pas l'ancien nom, donc toutes les opérations seront effectuées sous un nouveau nom.

Git Show Remote

Pour afficher des informations supplémentaires sur une télécommande particulière, utilisez la commande git remote avec la sous-commande show. Il est utilisé comme:

Syntaxe:

\$ git remote show <remote>

Il en résultera des informations sur le serveur distant. Il contient une liste de branches liées à la télécommande et également les points de terminaison attachés pour la récupération et la transmission.

Production:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote show origin
* remote origin
Fetch URL: https://github.com/ImDwivedi1/GitExample2
Push URL: https://github.com/ImDwivedi1/GitExample2
HEAD branch: master
Remote branches:
  BranchCherry    new (next fetch will store in remotes/orig
in)
  PullRequestDemo new (next fetch will store in remotes/orig
in)
  master          tracked
Local ref configured for 'git push':
  master pushes to master (up to date)
```

La sortie ci-dessus répertorie les URL du référentiel distant ainsi que les informations de la branche de suivi. Ces informations seront utiles dans divers cas.

Git Change Remote (Modification de l'URL d'une télécommande)

Nous pouvons changer l'URL d'un référentiel distant. La commande `git remote set` est utilisée pour modifier l'URL du référentiel. Il modifie une URL de référentiel distant existante.

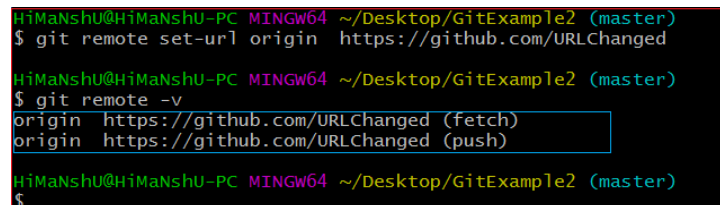
Ensemble à distance Git:

Nous pouvons changer l'URL distante simplement en utilisant la commande `git remote set`. Supposons que nous voulions créer un nom unique pour notre projet afin de le spécifier. Git nous permet de le faire. C'est un processus simple. Pour modifier l'URL distante, utilisez la commande ci-dessous:

\$ git remote set-url <nom distant> <newURL>

La **commande set-url distante** prend deux types d'arguments. Le premier est <nom distant>, c'est votre nom de serveur actuel pour le référentiel. Le deuxième argument est <newURL>, c'est votre nouveau nom d'URL pour le référentiel. La <nouvelle URL> doit être au format suivant: **https://github.com/URLChanged**

Considérez l'image ci-contre:



```
HIMANSHU@HIMANSHU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote set-url origin https://github.com/URLChanged

HIMANSHU@HIMANSHU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
origin https://github.com/URLChanged (fetch)
origin https://github.com/URLChanged (push)

HIMANSHU@HIMANSHU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

Dans la sortie ci-dessus, j'ai changé l'URL de mon référentiel existant en tant que `https://github.com/URLChanged` à partir de `https://github.com/ImDwivedi1/GitExample2`. Il peut être compris par mon nom d'URL que j'ai changé cela. Pour vérifier la dernière URL, exécutez la commande ci-dessous:

\$ git remote -v

git archive

La commande **git archive** est utilisée pour créer un fichier d'archive d'un instantané spécifique du projet.

Préparation d'une publication

Maintenant, vous voulez publier une version. Une des étapes consiste à créer une archive du dernier instantané de votre code pour les malheureux qui n'utilisent pas Git. La commande dédiée à cette action est **git archive** :

\$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz

\$ ls *.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz

Lorsqu'on ouvre l'archive, on obtient le dernier instantané du projet sous un répertoire **projet**. On peut aussi créer une archive au **format zip** de manière similaire en passant l'option **format=zip** à la commande **git archive** :

\$ git archive master --prefix='project/' --format=zip > `git describe master`.zip

Voilà deux belles archives **tar.gz** et **zip** de votre projet prêtes à être téléchargées sur un site web ou envoyées par courriel.

git submodule

La commande **git submodule** est utilisée pour gérer des dépôts externes à l'intérieur de dépôts normaux. Cela peut être pour des bibliothèques ou d'autres types de ressources partagées. La commande **submodule** a plusieurs sous-commandes (**add**, **update**, **sync**, etc) pour la gestion de ces ressources.

Il arrive souvent lorsque vous travaillez sur un projet que vous deviez utiliser un autre projet comme dépendance. Cela peut être une bibliothèque qui est développée par une autre équipe ou que vous développez séparément pour l'utiliser dans plusieurs projets parents. Ce scénario provoque un problème habituel:

vous voulez être capable de gérer deux projets séparés tout en utilisant l'un dans l'autre.

Voici un exemple. Supposons que vous développez un site web et que vous créez des flux Atom. Plutôt que d'écrire votre propre code de génération Atom, vous décidez d'utiliser une bibliothèque. Vous allez vraisemblablement devoir soit inclure ce code depuis un gestionnaire partagé comme CPAN ou Ruby gem, soit copier le code source dans votre propre arborescence de projet. Le problème d'inclure la bibliothèque en tant que bibliothèque externe est qu'il est difficile de la personnaliser de quelque manière que ce soit et encore plus de la déployer, car vous devez vous assurer de la disponibilité de la bibliothèque chez chaque client. Mais le problème d'inclure le code dans votre propre projet est que n'importe quelle personnalisation que vous faites est difficile à fusionner lorsque les modifications du développement principal arrivent.

Git gère ce problème avec les sous-modules. Les sous-modules vous permettent de gérer un dépôt Git comme un sous-répertoire d'un autre dépôt Git. Cela vous laisse la possibilité de cloner un dépôt dans votre projet et de garder isolés les **commits** de ce dépôt.

Démarrer un sous-module

Détaillons le développement d'un projet simple qui a été divisé en un projet principal et quelques sous-projets.

Commençons par ajouter le dépôt d'un projet Git existant comme sous-module d'un dépôt sur lequel nous travaillons. Pour ajouter un nouveau sous-module, nous utilisons la commande **git submodule add** avec l'URL du projet que nous souhaitons suivre. Dans cette exemple, nous ajoutons une bibliothèque nommée «**DbConnector**».

```
$ git submodule add https://github.com/chaconinc/DbConnector
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Dépaquetage des objets: 100% (11/11), fait.
Vérification de la connectivité... fait.
```

Par défaut, les sous-modules ajoutent le sous-projet dans un répertoire portant le même nom que le dépôt, dans notre cas «**DbConnector**». Vous pouvez ajouter un chemin différent à la fin de la commande si vous souhaitez le placer ailleurs.

Si vous lancez **git status** à ce moment, vous noterez quelques différences.

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

Modifications qui seront validées :
  (utilisez "git reset <fichier>..." pour désindexer)

    nouveau fichier :   .gitmodules
    nouveau fichier :   DbConnector
```

Premièrement, un fichier **.gitmodules** vient d'apparaître. C'est le fichier de configuration qui stocke la liaison entre l'URL du projet et le sous-répertoire local dans lequel vous l'avez tiré.

```
$ cat .gitmodules
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

Si vous avez plusieurs sous-modules, vous aurez plusieurs entrées dans ce fichier. Il est important de noter que ce fichier est en gestion de version comme vos autres fichiers, à l'instar de votre fichier **.gitignore**. Il est poussé et tiré comme le reste de votre projet. C'est également le moyen que les autres personnes qui clonent votre projet ont de savoir où récupérer le projet du sous-module.

Bon à savoir: Comme l'URL dans le fichier **.gitmodules** est ce que les autres personnes essaieront en premier de cloner et de tirer, assurez-vous que cette URL est effectivement accessible par les personnes concernées. Par exemple, si vous utilisez une URL différente pour pousser que celle que les autres utiliseront pour tirer, utilisez l'URL à laquelle les autres ont accès. Vous pouvez surcharger cette URL localement pour votre usage propre avec la commande **git config submodule.DbConnector.url PRIVATE_URL**.

L'autre information dans la sortie de **git status** est l'entrée du répertoire du projet. Si vous exécutez **git diff**, vous verrez quelque chose d'intéressant:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Même si **DbConnector** est un sous-répertoire de votre répertoire de travail, Git le voit comme un sous-module et ne suit pas son contenu (si vous n'êtes pas dans ce répertoire). En échange, Git l'enregistre comme un **commit** particulier de ce dépôt.

Si vous souhaitez une sortie diff plus agréable, vous pouvez passer l'option **--submodule** à **git diff**.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Au moment de valider, vous voyez quelque chose comme:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Remarquez le mode 160000 pour l'entrée **DbConnector**. C'est un mode spécial de Git qui signifie globalement que vous êtes en train d'enregistrer un **commit** comme un répertoire plutôt qu'un sous-répertoire ou un fichier.

Enfin, poussez ces modifications :

```
$ git push origin master
```

Cloner un projet avec des sous-modules

Maintenant, vous allez apprendre à cloner un projet contenant des sous-modules. Quand vous récupérez un tel projet, vous obtenez les différents répertoires qui contiennent les sous-modules, mais encore aucun des fichiers:

```
$ git clone https://github.com/chaconinc/MainProject
Clonage dans 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Dépaquetage des objets: 100% (14/14), fait.
Vérification de la connectivité... fait.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x  9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x  7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--  1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x  2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--  1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x  3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

Le répertoire **DbConnector** est présent mais vide. Vous devez exécuter deux commandes: **git submodule init** pour initialiser votre fichier local de configuration, et **git submodule update** pour tirer toutes les données de ce projet et récupérer le **commit** approprié tel que listé dans votre super-projet:

```
$ git submodule init
Sous-module 'DbConnector' (https://github.com/chaconinc/DbConnector)
enregistré pour le chemin 'DbConnector'
$ git submodule update
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```


Votre répertoire **DbConnector** est maintenant dans l'état exact dans lequel il était la dernière fois que vous avez validé.

Il existe une autre manière plus simple d'arriver au même résultat. Si vous passez l'option **--recurse-submodules** à la commande **git clone**, celle-ci initialisera et mettra à jour automatiquement chaque sous-module du dépôt.

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject
Clonage dans 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Dépaquetage des objets: 100% (14/14), fait.
Vérification de la connectivité... fait.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered
for path 'DbConnector'
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Dépaquetage des objets: 100% (11/11), fait.
Vérification de la connectivité... fait.
chemin du sous-module 'DbConnector' :
'c3f01dc8862123d317dd46284b05b6892c7b29bc' extrait
```

Si vous avez déjà cloné le projet et oublié **--recurse-submodules**, vous pouvez combiner les étapes **git submodule init** et **git submodule update** en lançant **git submodule update --init**. Pour initialiser, récupérer et extraire aussi tous les sous-modules récursivement, vous pouvez utiliser la commande complète **git submodule update --init --recursive**.

Travailler sur un projet comprenant des sous-modules

Nous avons à présent une copie d'un projet comprenant des sous-modules, et nous allons collaborer à la fois sur le projet principal et sur le projet du sous-module.

Tirer des modifications amont

Le modèle le plus simple d'utilisation des sous-modules est le cas de la simple consommation d'un sous-projet duquel on souhaite obtenir les mises à jour de temps en temps mais auquel on n'apporte pas de modification dans la copie de travail. Examinons un exemple simple.

Quand vous souhaitez vérifier si le sous-module a évolué, vous pouvez vous rendre dans le répertoire correspondant et lancer **git fetch** puis **git merge** de la branche amont pour mettre à jour votre code local.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master    -> origin/master
$ git merge origin/master
Mise à jour c3f01dc..d0354fc
Avance rapide
scripts/connect.sh | 1 +
src/db.c           | 1 +
2 files changed, 2 insertions(+)
```

Si vous revenez maintenant dans le projet principal et lancez **git diff --submodule**, vous pouvez remarquer que le sous-module a été mis à jour et vous pouvez obtenir une liste des **commits** qui y ont été ajoutés. Si vous ne voulez pas taper **--submodule** à chaque fois que vous lancez **git diff**, vous pouvez le régler comme format par défaut en positionnant le paramètre de configuration **diff.submodule** à la valeur «**log**».

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
 > more efficient db routine
 > better connection routine
```

Si vous validez à ce moment, vous fixez la version du sous-module à la version actuelle quand d'autres personnes mettront à jour votre projet.

Il existe aussi un moyen plus facile, si vous préférez ne pas avoir à récupérer et fusionner manuellement les modifications dans le sous-répertoire. Si vous lancez la commande **git submodule update --remote**, Git se rendra dans vos sous-modules et réalisera automatiquement le **fetch** et le **merge**.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master    -> origin/master
chemin du sous-module 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644' extrait
```

Cette commande considère par défaut que vous souhaitez mettre à jour la copie locale vers la branche **master** du dépôt du sous-module. Vous pouvez, cependant, indiquer une autre branche. Par exemple, si le sous-module **DbConnector** suit la branche **stable** du dépôt amont, vous pouvez l'indiquer soit dans votre fichier **.gitmodules** (pour que tout le monde le suive de

même) ou juste dans votre fichier local **.git/config**. Voyons ceci dans le cas du fichier **.gitmodules**:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
 27cf5d3..c87d55d  stable -> origin/stable
chemin du sous-module 'DbConnector' :
'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687' extrait
```

Si vous ne spécifiez pas la partie **-f .gitmodules**, la commande ne fera qu'une modification locale, mais il semble plus logique d'inclure cette information dans l'historique du projet pour que tout le monde soit au diapason.

Quand vous lancez **git status**, Git vous montrera que nous avons de nouveaux **commits** («**new commits**») pour le sous-module.

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

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans
la copie de travail)

   modifié :   .gitmodules
   modifié :   DbConnector (new commits)

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou
"git commit -a")
```

Si vous activez le paramètre de configuration **status.submodulesummary**, Git vous montrera aussi un résumé des modifications dans vos sous-modules:

```
$ git config status.submodulesummary 1

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans
la copie de travail)

      modifié :   .gitmodules
      modifié :   DbConnector (new commits)

Sous-modules modifiés mais non mis à jour :

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Ici, si vous lancez **git diff**, vous pouvez voir que le fichier **.gitmodules** a été modifié mais aussi qu'il y a un certain nombre de **commits** qui ont été tirés et sont prêts à être validés dans le projet du sous-module.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

C'est une information intéressante car vous pouvez voir le journal des modifications que vous vous apprêtez à valider dans votre sous-module. Une fois validées, vous pouvez encore visualiser cette information en lançant **git log -p**.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
    > catch non-null terminated lines
    > more robust error handling
    > more efficient db routine
    > better connection routine
```

Par défaut, Git essaiera de mettre à jour **tous** les sous-modules lors d'une commande **git submodule update --remote**, donc si vous avez de nombreux sous-modules, il est préférable de spécifier le sous-module que vous souhaitez mettre à jour.

Tirer des modifications amont depuis le serveur distant

Glissons-nous maintenant dans les habits d'un collaborateur, qui a son propre clone local du dépôt ProjetPrincipal. Lancer simplement **git pull** pour obtenir les nouvelles modifications validées ne suffit plus:

```
$ git pull
From https://github.com/chaconinc/ProjetPrincipal
   fb9093c..0a24cfc  master    -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
   c3f01dc..c87d55d  stable    -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules          | 2 +-
 DbConnector          | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)

$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/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:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine

no changes added to commit (use "git add" and/or "git commit -a")
```

Par défaut, la commande **git pull** récupère les modifications des sous-modules récursivement, comme nous pouvons le voir sur la première ligne ci-dessus. Cependant, elle ne met pas les sous-modules à jour. C'est affiché à la sortie de la commande **git status**, qui indique que le sous-module est «**modified**», et a des nouveaux commit (**new commits**). De plus, les chevrons indiquant les nouveaux commits pointent à gauche (<), ce qui signifie que ces commits sont enregistrés dans ProjetPrincipal, mais ne sont pas présents dans l'extraction locale de DbConnector. Pour finaliser la mise à jour, vous devez lancer **git submodule update**:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Notez que pour rester en sécurité, vous devriez lancer **git submodule update** avec le drapeau **--init** au cas où les commits de ProjetPrincipal que vous venez de tirer ont ajouté des nouveaux sous-modules, et le drapeau **--recursive** si certains sous-module ont des sous-modules imbriqués.

Si vous souhaitez automatiser ce processus, vous pouvez ajouter le drapeau **--recurse-submodules** à la commande **git pull** (depuis Git 2.14). Cela forcera Git à lancer **git submodule update** juste après le tirage, de manière à mettre à jour les sous-modules dans l'état correct. De plus, si vous voulez que Git tire toujours avec **--recurse-submodules**, vous pouvez régler l'option de configuration **submodule.recurse** à true (cela marche pour **git pull**

depuis Git 2.15). Cette option forcera Git à utiliser le drapeau `--recurse-submodules` pour toutes les commandes qui le supportent (à part **clone**).

Il y a une situation spéciale qui peut arriver lors du tirage depuis le **super-projet**; le dépôt amont peut avoir modifié l'**URL** du sous-module dans le fichier **.gitmodules** dans un des commits qui vous tirez. Cela peut arriver si le projet du sous-module change de plate-forme d'hébergement. Dans ce dernier cas, il est possible pour **git pull --recurse-submodules**, or **git submodule update**, d'échouer si le super-projet fait référence à un commit d'un sous-module qui n'est pas trouvé dans le serveur distant du sous-module configuré localement dans votre dépôt. Pour corriger cette situation, la commande **git submodule sync** est nécessaire:

```
# copier la nouvelle URL dans votre config locale
$ git submodule sync --recursive
# mettre à jour le sous-module depuis la nouvelle URL
$ git submodule update --init --recursive
```

Travailler sur un sous-module

Il y a fort à parier que si vous utilisez des sous-modules, vous le faites parce que vous souhaitez en réalité travailler sur le code du sous-module en même temps que sur celui du projet principal (ou à travers plusieurs sous-modules). Sinon, vous utiliseriez plutôt un outil de gestion de dépendances plus simple (tel que Maven ou Rubygems).

De ce fait, détaillons un exemple de modifications réalisées dans le sous-module en même temps que dans le projet principal et de validation et de publication des modifications dans le même temps.

Jusqu'à maintenant, quand nous avons lancé la commande **git submodule update** pour récupérer les modifications depuis les dépôts des sous-modules, Git récupérait les modifications et mettait les fichiers locaux à jour mais en laissant le sous-répertoire dans un état appelé «HEAD détachée». Cela signifie qu'il n'y pas de branche locale de travail (comme **master**, par exemple) pour y valider les modifications. Donc, toutes les modifications que vous y faites ne sont pas suivies non plus.

Pour rendre votre sous-module plus adapté à la modification, vous avez besoin de deux choses. Vous devez vous rendre dans chaque sous-module et extraire une branche de travail. Ensuite vous devez dire à Git ce qu'il doit faire si vous avez réalisé des modifications et que vous lancez **git submodule update --remote** pour tirer les modifications amont. Les options disponibles sont soit de les fusionner dans votre travail local, soit de tenter de rebaser le travail local par dessus les modifications distantes.

En premier, rendons-nous dans le répertoire de notre sous-module et extrayons une branche.

```
$ git checkout stable
Basculement sur la branche 'stable'
```

Attaquons-nous au choix de politique de gestion. Pour le spécifier manuellement, nous pouvons simplement ajouter l'option `--merge` à l'appel de `update`. Nous voyons ici qu'une modification était disponible sur le serveur pour ce sous-module et qu'elle a été fusionnée.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
  c87d55d..92c7337  stable      -> origin/stable
Mise à jour de c87d55d..92c7337
Avance rapide
  src/main.c | 1 +
  1 file changed, 1 insertion(+)
chemin du sous-module 'DbConnector': fusionné dans
'92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Si nous nous rendons dans le répertoire **DbConnector**, les nouvelles modifications sont déjà fusionnées dans notre branche locale **stable**. Voyons maintenant ce qui arrive si nous modifions localement la bibliothèque et que quelqu'un pousse une autre modification en amont dans le même temps.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Maintenant, si nous mettons à jour notre sous-module, nous pouvons voir ce qui arrive lors d'un rebasage de deux modifications concurrentes.

```
$ git submodule update --remote --rebase
Premièrement, rembobinons head pour rejouer votre travail par-dessus...
Application : unicode support
chemin du sous-module 'DbConnector': rebasé dans
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Si vous oubliez de spécifier `--rebase` ou `--merge`, Git mettra juste à jour le sous-module vers ce qui est sur le serveur et réinitialisera votre projet à l'état «**HEAD détachée**».

```
$ git submodule update --remote
chemin du sous-module 'DbConnector' :
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94' extrait
```


Si cela arrive, ne vous inquiétez pas, vous pouvez simplement revenir dans le répertoire et extraire votre branche (qui contiendra encore votre travail) et fusionner ou rebaser **origin/stable** (ou la branche distante que vous souhaitez) à la main.

Si vous n'avez pas validé vos modifications dans votre sous-module, et que vous lancez une mise à jour de sous-module qui causerait des erreurs, Git récupérera les modifications mais n'écrasera pas le travail non validé dans votre répertoire de sous-module.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
 5d60ef9..c75e92a  stable    -> origin/stable
error: Vos modifications locales seraient écrasées par checkout:
      scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Impossible d'extraire 'c75e92a2b3855c9e5b66f915308390d9db204aca' dans le
chemin du sous-module 'DbConnector'
```

Si vous avez réalisé des modifications qui entrent en conflit avec des modifications amont, Git vous en informera quand vous mettrez à jour.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (contenu): Conflit de fusion dans scripts/setup.sh
La fusion automatique a échoué ; réglez les conflits et validez le résultat
Impossible de fusionner 'c75e92a2b3855c9e5b66f915308390d9db204aca' dans le
chemin du sous-module 'DbConnector'
```

Vous pouvez vous rendre dans le répertoire du sous-module et résoudre le conflit normalement.

Publier les modifications dans un sous-module

Nous avons donc des modifications dans notre répertoire de sous-module, venant à la fois du dépôt amont et de modifications locales non publiées.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnessesary method
  > add new option for conn pooling
```

Si nous validons dans le projet principal et que nous le poussons en amont sans pousser les modifications des sous-modules, les autres personnes qui voudront essayer notre travail vont avoir de gros problèmes vu qu'elles n'auront aucun moyen de récupérer les modifications des sous-modules qui en font partie. Ces modifications n'existent que dans notre copie locale.

Pour être sûr que cela n'arrive pas, vous pouvez demander à Git de vérifier que tous vos sous-modules ont été correctement poussés avant de pouvoir pousser le projet principal. La commande **git push** accepte un argument **--recurse-submodules** qui peut avoir pour valeur «**check**» ou «**on-demand**». L'option «**check**» fera échouer **push** si au moins une des modifications des sous-modules n'a pas été poussée.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

Comme vous pouvez le voir, il donne aussi quelques conseils utiles sur ce que nous pourrions vouloir faire ensuite. L'option simple consiste à se rendre dans chaque sous-module et à pousser manuellement sur les dépôts distants pour s'assurer qu'ils sont disponibles publiquement, puis de réessayer de pousser le projet principal.

L'autre option consiste à utiliser la valeur «**on-demand**» qui essaiera de faire tout ceci pour vous.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Décompte des objets: 9, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (8/8), fait.
Écriture des objets: 100% (9/9), 917 bytes | 0 bytes/s, fait.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Décompte des objets: 2, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (2/2), 266 bytes | 0 bytes/s, fait.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/ProjetPrincipal
   3d6d338..9a377d1  master -> master
```

Comme vous pouvez le voir, Git s'est rendu dans le module **DbConnector** et l'a poussé avant de pousser le projet principal. Si la poussée du sous-module échoue pour une raison quelconque, la poussée du projet principal sera annulée.

Fusion de modifications de sous-modules

Si vous changez la référence d'un sous-module en même temps qu'une autre personne, il se peut que cela pose problème. Particulièrement, si les historiques des sous-modules ont divergé et sont appliqués à des branches divergentes dans un super-projet, rapprocher toutes les modifications peut demander un peu de travail.

Si un des **commits** est un ancêtre direct d'un autre (c'est-à-dire une fusion en avance rapide), alors Git choisira simplement ce dernier pour la fusion et cela se résoudra tout seul.

Cependant, Git ne tentera pas de fusion, même très simple, pour vous. Si les **commits** d'un sous-module divergent et doivent être fusionnés, vous obtiendrez quelque chose qui ressemble à ceci:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Dépaquetage des objets: 100% (2/2), fait.
From https://github.com/chaconinc/ProjetPrincipal
 9a377d1..eb974f8  master    -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Fusion automatique de DbConnector
CONFLICT (sous-module): Conflit de fusion dans DbConnector
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Donc, ce qui s'est passé en substance est que Git a découvert que les deux points de fusion des branches à fusionner dans l'historique du sous-module sont divergents et doivent être fusionnés. Il l'explique par «**merge following commits not found**» (fusion suivant les **commits** non trouvée), ce qui n'est pas clair mais que nous allons expliquer d'ici peu.

Pour résoudre le problème, vous devez comprendre l'état dans lequel le sous-module devrait se trouver. Étrangement, Git ne vous donne pas d'information utile dans ce cas, pas même les SHA-1 des **commits** des deux côtés de l'historique. Heureusement, c'est assez facile à comprendre. Si vous lancez **git diff**, vous pouvez obtenir les SHA-1 des **commits** enregistrés dans chacune des branches que vous essayiez de fusionner.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Donc, dans ce cas, `eb41d76` est le **commit** dans notre sous-module que **nous** avons et `c771610` est le **commit** amont. Si nous nous rendons dans le répertoire du sous-module, il devrait déjà être sur `eb41d76` parce que la fusion ne l'a pas touché. S'il n'y est pas, vous pouvez simplement créer et extraire une branche qui pointe dessus.

Ce qui importe, c'est le SHA-1 du **commit** venant de l'autre branche. C'est ce que nous aurons à fusionner. Vous pouvez soit essayer de fusionner avec le SHA-1 directement ou vous pouvez créer une branche à partir du **commit** puis essayer de la fusionner. Nous suggérons d'utiliser cette dernière méthode, ne serait-ce que pour obtenir un message de fusion plus parlant.

Donc, rendons-nous dans le répertoire du sous-module, créons une branche basée sur ce second SHA-1 obtenu avec `git diff` et fusionnons manuellement.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Nous avons eu un conflit de fusion ici, donc si nous le résolvons et validons, alors nous pouvons simplement mettre à jour le projet principal avec le résultat.

```
$ vim src/main.c (1)
$ git add src/main.c
$ git commit -am 'fusion de nos modifications'
[master 9fd905e] fusion de nos modifications

$ cd .. (2)
$ git diff (3)
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
-Subproject commit c77161012afbbee1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector (4)

$ git commit -m "Fusion du travail de Tom" (5)
[master 10d2c60] Fusion du travail de Tom
```

1. Nous résolvons le conflit
2. Ensuite, nous retournons dans le projet principal
3. Nous pouvons revérifier les SHA-1
4. Nous résolvons l'entrée en conflit dans le sous-module
5. Enfin, nous validons la résolution.

Cela peut paraître compliqué mais ce n'est pas très difficile.

Curieusement, il existe un autre cas que Git gère seul. Si un **commit** de fusion existe dans le répertoire du sous-module qui contient **les deux commits** dans ses ancêtres, Git va le suggérer comme solution possible. Il voit qu'à un certain point de l'historique du projet du sous-module, quelqu'un a fusionné les branches contenant ces deux **commits**, donc vous désirerez peut-être utiliser celui-ci.

C'est pourquoi le message d'erreur précédent s'intitulait «**merge following commits not found**», parce que justement, il ne pouvait pas trouver **le commit de fusion**. C'est déroutant car qui s'attendrait à ce qu'il **essaie** de le chercher?

S'il trouve un seul **commit** de fusion acceptable, vous verrez ceci:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
  "DbConnector"

which will accept this suggestion.
Fusion automatique de DbConnector
CONFLIT (submodule): Conflit de fusion dans DbConnector
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Ce qu'il suggère de faire est de mettre à jour l'index comme si vous aviez lancé **git add**, ce qui élimine le conflit, puis de valider. Vous ne devriez cependant pas le faire. Vous pouvez plus simplement vous rendre dans le répertoire du sous-module, visualiser la différence, avancer en avance rapide sur le **commit**, le tester puis le valider.

```
$ cd DbConnector/
$ git merge 9fd905e
Mise à jour eb41d76..9fd905e
Avance rapide

$ cd ..
$ git add DbConnector
$ git commit -am 'Avance rapide sur un fils commun dans le sous-module'
```

Cela revient au même, mais de cette manière vous pouvez au moins vérifier que ça fonctionne et vous avez le code dans votre répertoire de sous-module quand c'est terminé.

Trucs et astuces pour les sous-modules

Il existe quelques commandes qui permettent de travailler plus facilement avec les sous-modules.

Submodule foreach

Il existe une commande submodule **foreach** qui permet de lancer une commande arbitraire dans chaque sous-module. C'est particulièrement utile si vous avez plusieurs sous-modules dans le même projet.

Par exemple, supposons que nous voulons développer une nouvelle fonctionnalité ou faire un correctif et que nous avons déjà du travail en cours dans plusieurs sous-modules. Nous pouvons facilement remiser tout le travail en cours dans tous les sous-modules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Ensuite, nous pouvons créer une nouvelle branche et y basculer dans tous nos sous-modules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Basculement sur la nouvelle branche 'featureA'
Entering 'DbConnector'
Basculement sur la nouvelle branche 'featureA'
```

Vous comprenez l'idée. Une commande vraiment utile permet de produire un joli diff unifié des modifications dans le projet principal ainsi que dans tous les sous-projets.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

    commit_pager_choice();

+   url = url_decode(url_orig);
+
    /* build alias_argv */
    alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
    alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+   return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;
```

Ici, nous pouvons voir que nous définissons une fonction dans un sous-module et que nous l'appelons dans le projet principal. C'est un exemple exagérément simplifié, mais qui aide à mieux comprendre l'utilité de cette commande.

Alias utiles

Vous pourriez être intéressé de définir quelques alias pour des commandes longues pour lesquelles vous ne pouvez pas régler la configuration par défaut. Voici un exemple d'alias que vous pourriez trouver utiles si vous voulez travailler sérieusement avec les sous-modules de Git.

```
$ git config alias.sdiff '!git diff && git submodule foreach 'git diff''
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

De cette manière, vous pouvez simplement lancer `git supdate` lorsque vous souhaitez mettre à jour vos sous-module ou `git spush` pour pousser avec une gestion de dépendance de sous-modules.

Les problèmes avec les sous-modules

Cependant, utiliser des sous-modules ne se déroule pas sans accroc.

Commuter des branches

Commuter des branches qui contiennent des sous-modules peut également s'avérer difficile. Si vous créez une nouvelle branche, y ajoutez un sous-module, et revenez ensuite à une branche dépourvue de ce sous-module, vous aurez toujours le répertoire de ce sous-module comme un répertoire non suivi:

```
$ git --version
git version 2.12.2

$ git checkout -b add-crypto
Basculement sur la nouvelle branche 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Fichiers non suivis :
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    CryptoLibrary/

aucune modification ajoutée à la validation mais des fichiers non suivis sont
présents (utilisez "git add" pour les suivre)
```


Supprimer le répertoire n'est pas difficile, mais sa présence est assez déroutante. Si vous le supprimez puis que vous rebasculez sur la branche qui contient le sous-module, vous devrez lancer **submodule update --init** pour le réalimenter.

```
$ git clean -ffdx
Suppression de CryptoLibrary/

$ git checkout add-crypto
Basculement sur la branche 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out
'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile      includes      scripts      src
```

Une fois de plus, ce n'est pas réellement difficile, mais cela peut être déroutant.

Les nouvelles versions de Git (Git >= 2.13) simplifie tout ceci en ajoutant le drapeau **--recurse-submodules** à la commande **git checkout**, qui s'occupe de placer les sous-modules dans le bon état pour la branche sur laquelle nous commutons.

```
$ git --version
git version 2.13.3

$ git checkout -b add-crypto
Basculement sur la branche 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Clonage dans 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

Utiliser le drapeau `--recurse-submodules` de `git checkout` peut aussi être utile quand vous travaillez sur plusieurs branches dans les super-projet, chacune aillant votre sous-module pointant sur des commits différents. En fait, si vous commutez de branches qui enregistrent le sous-module à différents commits, à l'exécution de `git status` le sous-module apparaîtra comme «modified», et indique «new commits» (nouveaux commits). C'est parce que l'état du sous-module n'est pas géré par défaut lors du basculement de branches.

Cela peut être vraiment déroutant, donc c'est toujours une bonne idée de toujours lancer `git checkout --recurse-submodules` quand votre projet a des sous-modules. Pour les versions anciennes de Git qui n'ont pas de drapeau `--recurse-submodules`, après l'extraction, vous pouvez utiliser `git submodule update --init --recursive` pour placer les sous-modules dans le bon état.

Par chance, vous pouvez indiquer à Git (≥ 2.14) de toujours utiliser le drapeau `--recurse-submodules` en paramétrant l'option de configuration `submodule.recurse`: `git config submodule.recurse true`. Comme noté ci-dessus, cela forcera aussi Git à parcourir récursivement les sous-modules pour toute commande qui accepte l'option `--recurse-submodules` (excepté `git clone`).

Basculer d'un sous-répertoire à un sous-module

Une autre difficulté commune consiste à basculer de sous-répertoires en sous-modules. Si vous suiviez des fichiers dans votre projet et que vous voulez les déplacer dans un sous-module, vous devez être très prudent ou Git sera inflexible. Présunons que vous avez les fichiers dans un sous-répertoire de votre projet, et que vous voulez les transformer en un sous-module. Si vous supprimez le sous-répertoire et que vous exécutez `submodule add`, Git vous hurle dessus avec:

```
$ rm -Rf CryptoLibrary/  
$ git submodule add https://github.com/chaconinc/CryptoLibrary  
'CryptoLibrary' already exists in the index
```

Vous devez d'abord supprimer le répertoire `CryptoLibrary` de l'index. Vous pourrez ensuite ajouter le sous-module:

```
$ git rm -r CryptoLibrary  
$ git submodule add https://github.com/chaconinc/CryptoLibrary  
Cloning into 'CryptoLibrary'...  
remote: Counting objects: 11, done.  
remote: Compressing objects: 100% (10/10), done.  
remote: Total 11 (delta 0), reused 11 (delta 0)  
Unpacking objects: 100% (11/11), done.  
Checking connectivity... done.
```

Maintenant, supposons que vous avez fait cela dans une branche. Si vous essayez de basculer dans une ancienne branche où ces fichiers sont toujours dans l'arbre de projet plutôt que comme sous-module, vous aurez cette erreur:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by
checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

Vous pouvez le forcer à basculer avec **checkout -f**, mais soyez attentif à ce qu'il soit propre ou les modifications seraient écrasées.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Basculement sur la branche 'master'
```

Ensuite, lorsque vous rebasculez, vous aurez un répertoire **CryptoLibrary** vide et **git submodule update** pourrait ne pas le remettre en état. Vous allez devoir vous rendre dans le répertoire de votre sous-module et lancer **git checkout** pour retrouver tous vos fichiers. Vous pouvez lancer ceci dans un script **submodule foreach** dans le cas de multiples sous-modules.

Il est important de noter que depuis les versions de Git récentes, les sous-modules conservent leurs données Git dans le répertoire **.git** du projet principal, ce qui à la différence des versions antérieures, permet de supprimer le dossier du sous-module sans perdre les **commits** et les branches qu'il contenait.

Avec ces outils, les sous-modules peuvent être une méthode assez simple et efficace pour développer simultanément sur des projets connexes mais séparés.

Collaboration en dépôt distant

Les **dépôts** – ou **repo**, de l'anglais repository – distants permettent la collaboration sur des projets. Chaque projet Git peut être synchronisé avec un ou plusieurs dépôts distants. Sur chacun d'eux, vous pouvez avoir des droits en lectures, en écriture ou les deux. La collaboration consiste donc à récupérer le code depuis un dépôt distant, à effectuer des modifications puis à pousser – ou push – les commits de modifications sur le repo distant ; de manière à ce que tout le monde puisse y accéder.

Le bare repository

Lorsque l'on parle de dépôt distant, de l'autre côté du tuyau se trouve toujours un **bare repository**, ou dépôt nu. Ce type de dépôt a pour unique fonction d'être un réceptacle, le point central de synchronisation entre différents dépôts de travail.

Il s'agit d'un dépôt qui ne contient pas de répertoire de travail. Contrairement à un dépôt "normal", il ne contient pas de répertoire.**git** mais place tout le contenu de ce répertoire directement à la racine. Par ailleurs, par convention, ce type de répertoire possède l'extension.**git: projet-dingue.git**.

Ce type de dépôt ne possède pas de répertoire de travail car personne ne travaille directement dedans. Un dépôt **bare** a simplement vocation à être la source de **git clone** et la cible de **git push**.

Un repos de ce type se créer aussi facilement qu'un repo classique : **git init --bare**

Il est également possible d'en créer un directement depuis un dépôt existant. Avec la commande suivante: **git clone --bare <repo-address>**

Les protocoles de Git

On a parlera ici essentiellement de deux protocoles qui interviennent dans l'installation de Git, il s'agit notamment de : SSH et HTTPS. Ce sont en effet les deux plus courants mais pas les seuls.

HTTPS (HyperText Transfer Protocol Secure)

Le HTTPS est le protocole de synchronisation le plus courant. Il offre en effet la gestion la plus souple de l'authentification. Elle est possible, mais non obligatoire. Ainsi, il est possible pour quiconque possède l'url de cloner le dépôt sans avoir à s'authentifier, tout en requérant une authentification par login et mot de passe pour ajouter des modifications.

C'est la méthode par défaut de toutes les plateformes collaboratives. Lorsque l'on collabore activement à un projet, il devient vite fastidieux d'avoir à entrer un couple login/mot de passe pour chaque push. Comme nous l'avons vu lors de la configuration de Git, il devient alors pertinent de mettre en place une gestion automatique de l'authentification.

Git supporte aussi le HTTP, son fonctionnement est le même que le HTTPS, la sécurité en moins. Il est également bon de savoir que dans les versions plus anciennes de Git (antérieures à 1.6.6), le protocole HTTP – aujourd'hui dénommé *Dumb* HTTP, par opposition à la gestion actuelle nommée *Smart* – ne permettait pas l'authentification. Le dépôt du projet était simplement servi tel quel par un serveur web et récupéré par Git à travers ce protocole.

SSH (Secure Shell)

Le SSH a comme atout une gestion intégrée de l'authentification, une sécurisée via un chiffrement natif des communications sans recours à des certificats SSL et une présence quasi systématique sur les serveurs.

Lorsqu'un dépôt distant est configuré sur un serveur, ce dernier a 99% de chance d'avoir déjà SSH de configuré. Le principal inconvénient du SSH est qu'il nécessite la configuration d'une clef SSH pour tous les utilisateurs qui voudraient accéder au dépôt, même en lecture.

Cela ne présente aucun problème pour les dépôts privés, mais pour les dépôts publics et les projets open source, c'est plus embêtant. En SSH, vous n'auriez ainsi pas été en mesure de cloner le projet Git emojis hook sans d'abord enregistrer votre clef SSH sur GitHub. D'ailleurs, GitHub ne vous propose l'option que si vous êtes connectés et qu'une clef SSH est associée à votre compte.

La clef possède en revanche l'énorme avantage d'unifier l'authentification entre les différents systèmes. Il n'y a pas de mot de passe à taper et ce n'est pas à Git de se souvenir d'un mot de passe. Qu'il s'agisse de Windows, Linux ou macOS, on utilise la même stratégie et on peut même partager une clef entre plusieurs systèmes pour s'authentifier de manière transparente.

Les clefs SSH sont supportées par tous les systèmes. Pour les créer sous macOS, Linux et Unix,

Le protocole local

Le protocole local est le plus basique de tous. Il permet de cloner un dépôt accessible localement, c'est à dire sur un système de fichiers local. Il est la plupart du temps utilisé lorsque tous les collaborateurs ont accès à un système de fichiers partagé, tel que NFS.

```
# On précise simplement la chemin d'accès  
git clone /chemin/vers/repo
```

Dans cette configuration, Git active automatiquement l'option **--local**. Ainsi, Git gagne en rapidité en réalisant simplement une copie des dossiers **refs** et **objets** du répertoire **.git**. En outre, pour économiser de la place, les éléments contenus dans **objects** sont des liens physiques et partagent donc le même contenu.

Cela se vérifie facilement en affichant les inodes des fichiers.

```
# Dépôt source git-emojis-clone  
ls -li git-emojis-clone/.git/objects  
3131811 info 3131810 pack  
# On créer un autre dossier et on s'y place  
mkdir test && cd test  
# On clone le répertoire d'origine  
git clone ../git-emojis-clone  
# On affiche les inodes  
ls -li git-emojis-clone/.git/objects  
3131811 info 3131810 pack
```

On voit que les inodes sont les mêmes. Cela indique bien qu'ils sont partagés. Si l'on veut simuler une copie comme à travers le réseau, on peut passer l'argument **--no-local**. Cela permettra de forcer un import propre, notamment si le dépôt d'origine est un import depuis un autre VCS par exemple.

Le protocole Git

Le protocole Git n'utilise ni serveur web, ni le SSH. Il fonctionne via un démon Git qui écoute sur un port dédié (le port 9418 par défaut) et fournit un service assez similaire au SSH mais sans aucune authentification.

Dans la mesure où il n'y a aucune sécurité, lorsque ce protocole est utilisé, c'est simplement pour le clonage et l'ajout de modifications est désactivé. Il est toutefois possible de l'activer sur un réseau local par exemple.

Étant donné qu'il s'agit d'un démon spécifique écoutant sur un port dédié, sa mise en place n'est pas standard et de ce fait très peu répandue.

Lister les remotes

Lorsque l'on crée un projet localement, il n'y a par défaut pas de dépôt distant, cela semble logique. En revanche, si le dépôt est cloné depuis une source, alors la source est directement référencée comme dépôt distant.

```
# Test sur le dépôt localement créé dans le chapitre précédent
git remote
```

La commande ne retourne rien, c'est qu'il n'y a aucune remote... Testons donc maintenant sur le repo [Git emojis hook](#) cloné depuis GitHub.

```
git remote
origin
```

```
# L'option -v nous permet d'obtenir un peu plus de précisions
```

```
git remote -v
origin git@github.com:Buzut/git-emojis-hook.git (fetch)
origin git@github.com:Buzut/git-emojis-hook.git (push)
```

On voit cette fois que la remote nommée **origin** a pour origine le dépôt à l'adresse **git@github.com:Buzut/git-emojis-hook.git**, pour les push et les pull. Votre dépôt distant est automatiquement nommé **origin**. C'est un choix par défaut de Git au même titre que le nom de **master** pour la branche par défaut.

Lorsque les adresses sont du type **user@url** c'est qu'il s'agit du protocole SSH. Si vous utilisez HTTPS, l'adresse commencera par HTTPS.

En savoir plus sur une remote

Lorsque vous voulez plus d'information sur une remote en particulier, vous pouvez le demander à Git.

On prend cette fois pour exemple le repo que j'utilise pour le code de ce site

git remote show origin

* remote origin

Fetch URL: git@git.buzut.net:Buzut/buzut-blog.git

Push URL: git@git.buzut.net:Buzut/buzut-blog.git

HEAD branch: master

Remote branches:

formations tracked

master tracked

Local branches configured for 'git pull':

formations merges with remote formations

master merges with remote master

Local refs configured for 'git push':

formations pushes to formations (up to date)

master pushes to master (fast-forwardable)

Nous avons de nouveau l'adresse du dépôt, mais en plus des informations détaillées sur les branches locales et distantes. On sait donc que j'ai deux branches locales configurées pour traquer les branches distantes du même nom et dans notre cas, la réciproque est vraie pour le push.

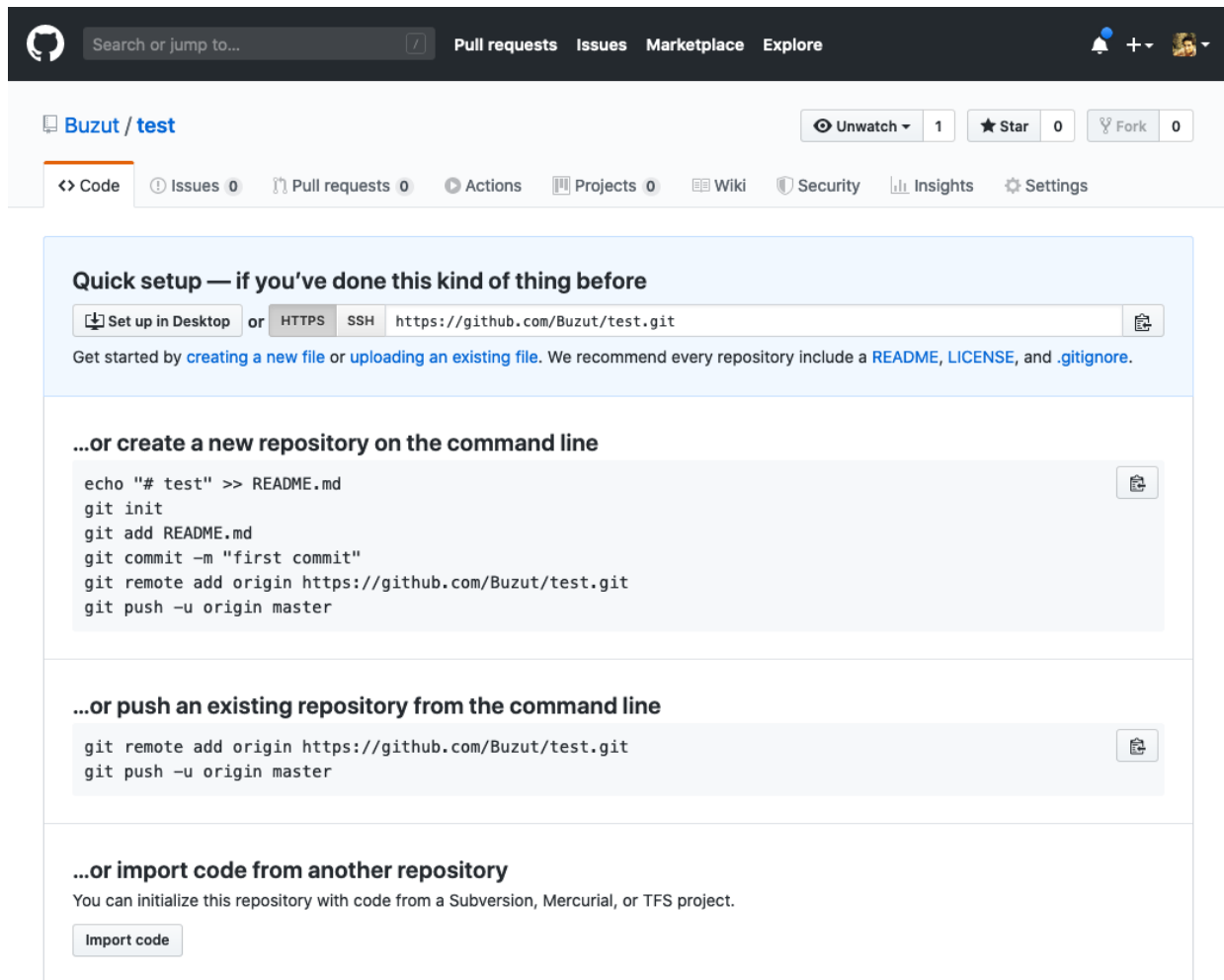
On apprend également que la branche locale “**formations**” est à jour avec la version distante. En revanche, la **master** locale possède des commits qui ne sont pas encore sur la **master** distante. Git nous spécifie cependant que ces changements sont **fast-forwardable**; il n'y aura pas de merge à prévoir en amont.

Cette commande nous montrerait également si nous avions des branches distantes qui n'ont pas été récupérées localement ou encore des branches distantes effacées mais toujours présentes localement.

Ajouter un dépôt

Chacun des dépôts distants possède un nom. Cela permet de facilement préciser à Git si vous voulez effectuer une action sur une remote plutôt qu'une autre.

Que votre dépôt possède un dépôt distant ou qu'il n'en possède encore aucun, vous pouvez toujours en ajouter de nouveaux. Par exemple, bien que mon repo **Git emojis hook** soit publiquement accessible en lecture, vous ne pourrez pas y pusher de modifications. Si vous voulez y effectuer des modifications et les partager, il faudra que vous créiez votre propre dépôt distant.



Pour les besoins du test, je crée sur mon compte GitHub un dépôt vide, créativement nommé **"test"**

Ajout de ce dépôt distant au dépôt local

La commande générique est : `git remote add <name> <url>`

git remote add testrepo git@github.com:Buzut/test.git

On confirme l'ajout en affichant les remotes

git remote -v

origin git@github.com:Buzut/git-emojis-hook.git (fetch) origin git@github.com:Buzut/git-emojis-hook.git (push) testrepo git@github.com:Buzut/test.git (fetch)

testrepo git@github.com:Buzut/test.git (push)

Notre repo est bien là. Cependant, si nous demandons plus d'infos à Git, on réalise qu'il ne sait pas ce que doit traquer **testrepo**.

git remote show testrepo

```
* remote testrepo
Fetch URL: git@github.com:Buzut/test.git
Push URL: git@github.com:Buzut/test.git
HEAD branch: (unknown)
```

Lors du premier push, il faudra donc indiquer à Git quel est l'**upstream** par défaut, c'est à dire la branche distante principale. On fait pour cela un **git push** en passant l'option **-u**. Comme son nom l'indique dans sa version longue, **--set-upstream**, permet de définir le dépôt **upstream**.

```
# La commande générique : git push -u <upstream> <branch>
```

git push -u testrepo master

```
# On vérifie de nouveau l'état de notre remote
```

git remote show testrepo

```
* remote testrepo
Fetch URL: git@github.com:Buzut/test.git
Push URL: git@github.com:Buzut/test.git
HEAD branch: master
Remote branch:
master tracked
Local ref configured for 'git push':
master pushes to master (up to date)
```

Récupérer et envoyer les données

Lorsque vous travaillez avec des dépôts distants, il faut régulièrement les synchroniser : récupérer les modifications effectuées par d'autres et pousser les vôtres.

Push : envoyer vos modifications

Lorsque vous clonez un dépôt, ce dernier est automatiquement ajouté comme dépôt distant sous le nom d'*origin* et vos branches traquent automatiquement les branches de ce dépôt.

Par défaut, lors du clonage, vous êtes sur la **master**, si vous y effectuez des modifications et faites un **git push**, celles-ci seront automatiquement envoyées sur la branche *master* du repo par défaut (**souvent origin**). **git push** fait donc en réalité implicitement **git push origin master**.

git push

```
Total 0 (delta 0), reused 0 (delta 0)
To git@git.buzut.net:Buzut/buzut-blog.git
018dc17..58e4a6b master -> master
```

Comprendre le résultat du push

On s'aperçoit que Git envoie ici la *master* locale vers la *master* distante, sur le repo configuré comme le dépôt distant par défaut. La première ligne, sans ambiguïté, indique sur quel dépôt distant est poussée la branche, la seconde ligne comporte quatre informations :

- un flag (espace, +, -, *, !, =),
- le résumé sous la forme <old-commit>..<new-commit> ,
- les branches sous la forme <localbranch> -> <remotebranch> (dans le cas d'un effacement, seule la branche effacée est listée),
- en cas d'échec, la raison de ce dernier.

Le flag prend une des quatre valeurs suivantes :

- espace : si l'update est un simple *fast-forward* ;
- +: s'il s'agit d'un update forcé ;
- -: pour une référence effacée ;
- !: lorsqu'il s'agit d'un échec ;
- =: lorsqu'il n'y a eu aucun changements.

Si vous créez une nouvelle branche et que vous faites de nouveau un push, cette nouvelle branche sera aussi envoyée sur **origin** car c'est la remote par défaut. Vous pouvez cependant en décider autrement en spécifiant à Git le comportement à tenir.

Commande push plus détaillée

git push [<upstream>] [<branch>]

Vous voyez ici qu'il n'est donc pas nécessaire d'être sur la branche que vous voulez pusher. Si vous êtes sur une autre branche, vous devez alors spécifier à Git qu'il faut pusher une branche tierce plutôt que l'actuelle.

Par la suite, Git se souvient de la remote associée à une branche. Sauf mention contraire, un push ultérieur associera donc automatiquement la branche distante sur laquelle envoyer les données.

Par ailleurs, si vous tentez de pousser vos modifications mais que la branche distante a été modifiée entre temps, Git refusera d'intégrer vos modifications tant que vous ne les intégrez par d'abord à votre historique.

fetch: récupérer l'état du dépôt distant

Cette commande permet de récupérer l'ensemble des branches disponibles sur le dépôt distant.

git fetch testrepo

remote: Enumerating objects: 4, done.

remote: Counting objects: 100% (4/4), done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 Unpacking objects: 100% (3/3), done.

From github.com:Buzut/test

1f80958..a0e6f51 master -> testrepo/master

Les modifications ne sont en revanche pas intégrées aux branches locales. Elles sont placées dans des branches spécifiques reflétant l'état du dépôt distant. C'est ce qui nous est ici indiqué sur la dernière ligne du **git fetch**. On le constate par ailleurs en affichant **toutes** les branches locales.

```
# L'option -a pour "all"
git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/testrepo/master
# Un git log nous confirme que rien n'a bougé
git log --oneline
1f80958 (HEAD -> master, origin/master, origin/HEAD)    update licence year
```

Directement depuis l'interface en ligne de GitHub, j'ai ajouté un fichier. La branche distante possède donc un commit d'avance sur la branche locale. Nous allons donc nous placer sur la branche **remotes/testrepo/master** et observer ce qu'il s'y passe.

```
# Changeons maintenant de branche
git switch remotes/testrepo/master Note: switching to 'remotes/testrepo/master'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
git switch -c <new-branch-name>
Or undo this operation with:
git switch -
Turn off this advice by setting config variable advice.detachedHead to false
HEAD is now at a0e6f51 Create DOC.md
```

Git nous indique ici que nous ne pouvons pas directement effectuer de modifications dans cette branche – elle reflète l'état de la remote, ça n'aurait donc pas de sens. On peut regarder ce qui s'y passe et créer une nouvelle branche à partir de cette dernière si l'on veut sauvegarder des modifications que l'on aurait faites ou simplement créer une branche locale qui reflète l'état de la remote.

En outre, si nous voulons intégrer ces modifications directement dans la *master* (ou une autre branche locale), on peut tout à fait fusionner cette branche avec une branche locale déjà existante.

pull : intégrer des modifications distantes

Lorsque vous souhaitez récupérer des modifications effectuées par d'autres sur la branche courante, vous pouvez effectuer un **git pull**. Deux cas se présentent :

- les modifications sont des enfants directs du dernier commit de votre branche locale,
- la branche distante et votre branche locale ont chacune des modifications que l'autre n'a pas.

La premier cas est simple, les modifications distantes sont récupérées et les changements sont simplement ajoutées à la branche locale.

Dans le second cas, un merge doit avoir lieu. **git pull** invoque **merge** par défaut. En réalité, cette commande est un raccourci permettant de faire en une seule fois **git fetch** et **git merge**.

Suivi des branches

Comme nous l'avons vu plus haut, lorsqu'un dépôt upstream est défini, Git présume que toute nouvelle branche suivra automatiquement la branche du même nom de l'upstream par défaut. Deux cas sont alors couramment rencontrés :

- Lorsqu'une branche locale du nom d'une branche distante est créée, la branche locale suit automatiquement la branche distante. Ainsi si le **fetch** a récupéré **origin/feature**, lors du **git switch feature**, la branche locale **feature** reflètera l'état de **origin/feature**. Subséquemment, tout **git push** enverra les modifications sur la branche distante **origin/feature** et tout **git pull** intégrera les ajouts de **origin/feature** à **feature**.
- Lorsqu'une nouvelle branche est créée, tout **git push** sans plus de précisions créera alors une branche du nom de la branche locale sur la remote par défaut.

Il est courant d'avoir des branches suivant différentes remotes. Nous l'avons vu, lorsqu'il n'y a pas encore de remote par défaut, on indique à Git lors du push quelle branche distante traquer avec **git push -u**. Lorsqu'une branche locale est créée et qu'elle n'a pas d'équivalent sur une remote, il est alors possible de lui spécifier quelle remote traquer lors du push en utilisant **git push -u <upstream>**.

Par ailleurs, on peut déterminer la branche distante à suivre directement lors de la création de cette dernière via la commande **branch**.

branch permet de créer la branche en précisant l'upstream

Si la branche existe déjà, la configuration de l'upstream sera simplement ajoutée

git branch -u <upstream> <branch>

Ces paramètres sont enregistrés dans la configuration locale du projet, dans le fichier **.git/config**. On peut facilement obtenir des informations via **git config**.

```
# Connaître la remote d'une branche (la master dans notre exemple)
# Ici nous allons constater que la remote par défaut est gogs et non origin
git config branch.master.remote
gogs
# La configuration nous permet aussi d'éditer ces préférences
git config branch.master.remote origin
# On vérifie de nouveau
git config branch.master.remote
origin
```

Il est également possible de dissocier la cible du push et du pull. On peut donc vouloir récupérer les modifications depuis un dépôt donné et les envoyer sur un autre.

```
# Définir une cible pour le push autre que la "remote" par défaut
git config branch.<name>.pushRemote
# Définir une cible pour fetch/pull/rebase autre que la remote par défaut
git config branch.<name>.merge
```

Limiter les branches suivies

Lors de l'ajout d'une remote,

Il est possible de limiter les branches traquées (et donc récupérées par fetch). On utilise pour cela **set-branches**.

De nombreuses autres options sont disponibles. Plus rarement utilisées, nous ne les listerons pas ici. Cependant, pour des besoins spécifiques, ne perdez pas les bons réflexes et référez-vous [à la doc](#).

Limiter les données récupérées

Lorsque l'on travaille avec de gros projets, il est possible de vouloir n'en récupérer qu'une partie pour s'épargner un lent téléchargement et gagner un peu d'espace disque.

Lorsque l'on est pas encore en possession du repo, tout commence avec la commande **clone**.

```
# Cloner une branche seulement
# Si la branche n'est pas précisée, la branche référencée par HEAD (probablement master) sera
récupérée
git clone [--branch <branch-name>] --single-branch <repo_url>
# Cloner seulement depuis une date donnée
git clone --shallow-since=<date> <repo-url>
# On peut alternativement spécifier un nombre de commits
# Penser à préciser --branch si la cible est autre que master
# Ou spécifier --no-single-branch pour récupérer toutes les branches
git clone --depth <number> <repo-url>
```

fetch permet ensuite de travailler avec ce type de repo partiel. On trouve bien entendu les paramètres **--depth** et **--shallow-since** qui permettent soit de récupérer partiellement une autre branche ou de modifier la taille de l'historique d'une branche déjà récupérée.

On trouve également **--unshallow** qui permet de récupérer l'ensemble de l'historique.

```
# Par ailleurs, dans le cas où l'on voudrait ne récupérer qu'une branche supplémentaire du projet
# On le spécifie à fetch
git fetch <remote-name> <branch>
```

On peut également limiter le tracking des branches directement lors de l'emploi de **git remote**.

```
# Tracking d'une seule branches distante git remote add -t <name> <branch>
# Modifications des branches suivies (écrase les réglages précédents)
git remote set-branches <remote-name> <branch-name> [<branch-name2>]
# On peut également ajouter une tracking plutôt que de tout écraser
git remote set-branches --add <remote-name> <branch-name>
```

Supprimer et renommer une remote

Il arrive que l'on veuille renommer un dépôt distant, il arrive aussi qu'un dépôt distant ne soit plus d'aucune utilité, dans ce cas, on le supprime. Dans un cas comme dans l'autre, ce sont des actions très simples.

```
# On renome le dépôt testrepo en secondary
git remote rename testrepo secondary
# On vérifie si la commande a bien été exécutée
git remote
origin
secondary
# Pour supprimer le repo, c'est tout aussi simple
git remote rm secondary
# Vérification
git remote
origin
```

Pour supprimer une branche distante, cela semble parfois contre-intuitif car il faut pusher.

```
# Suppression de la branche "unebranche"
git push [<upstream>] -d unebranche
```

Les plateformes web

Que vous vouliez collaborer ou bénéficier d'une copie distante de votre travail pour une plus grande sécurité, de nombreuses options existent. Le plupart des plateformes incluent, en plus de Git, des outils de collaborations avancés : gestion des tickets, gestion des modifications, gestion avancée des droits, création de wiki, etc.

Les deux plateformes les plus connues sont **GitHub** et **GitLab**. GitHub est vraiment la plus grosse plateforme d'hébergement de code d'Internet. Elle héberge de très nombreux projets open source et propose un niveau d'utilisation gratuit.

Elle a été rachetée par Microsoft, on peut donc s'attendre à un haut niveau d'intégration avec Azure, le cloud de Microsoft. GitHub possède de [nombreuses fonctionnalités](#), mais son plus gros atout est à mon avis sa communauté de développeur.

GitLab est le challenger de GitHub. Dans l'ensemble, GitLab offre plus de fonctionnalités et prend une posture [résoluement DevOps](#), qui en fait une plateforme très complète. De plus, GitLab a la particularité d'avoir une version [communautaire open source](#). Il est donc possible d'installer GitLab sur ses propres serveurs et de garder le contrôle sur l'hébergement de ses données. La version SaaS possède toutefois [plus de fonctionnalités](#).

En dernier lieu, il faut mentionner [Bitbucket](#), la plateforme d'Atlassian. Elle propose également un large éventail de fonctionnalités et s'intègre particulièrement bien avec les autres services de l'entreprise, notamment Jira et Trello.

GitLab est à mon sens la plateforme la plus riche et flexible, cependant, si vous êtes hésitant, GitLab propose des comparatifs avec GitHub et Bitbucket :

- [GitHub vs GitLab](#),
- [Bitbucket vs GitLab](#).

Auto-hébergement

J'ai mentionné le fait que GitLab peut être librement installé afin d'en posséder sa propre instance privée. Le principal inconvénient de GitLab dans ce cas est son besoin élevé en ressources. Alternativement, Git possède lui-même un serveur web intégré permettant de parcourir ses repos depuis un navigateur web. Cependant, il est très limité et son intérêt en est donc réduit.

[Gogs](#) est une plateforme open source écrite en Go. Elle est facile à installer et consomme très peu de ressources, à tel point qu'elle peut se contenter d'un petit VPS ou d'un Raspberry Pi. Malgré sa faible consommation en ressources, elle offre un grand nombre de fonctionnalités et constitue une alternative viable aux plateformes SaaS.

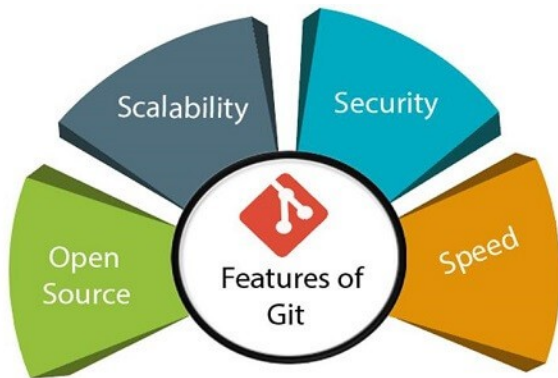
Il faut également mentionner [Gitea](#), un fork de Gogs développé par la communauté et possédant plus de fonctionnalités.

Quel que soit votre besoin, vous avez maintenant les clefs pour choisir la bonne solution et la mettre en œuvre sans plus attendre. Dans le prochain chapitre, nous allons explorer les usages avancés de Git : les commandes que nous n'utilisons que rarement mais qui s'avèrent bien pratiques dans certaines situations.

FONCTIONNALITÉS GIT

Fonctionnalités de Git

Voici quelques fonctionnalités remarquables de Git:



- **Open source**

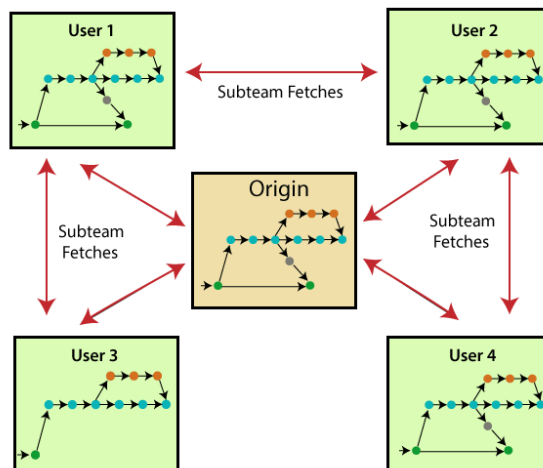
Git est un outil open source. Il est publié sous la licence GPL (General Public License).

- **Évolutif**

Git est évolutif, ce qui signifie que lorsque le nombre d'utilisateurs augmente, le Git peut facilement gérer de telles situations.

- **Distribué**

L'une des grandes fonctionnalités de Git est qu'il est **distribué**. Distribué signifie qu'au lieu de basculer le projet sur une autre machine, nous pouvons créer un "clone" de l'ensemble du référentiel. De plus, au lieu de n'avoir qu'un seul référentiel central auquel vous envoyez les modifications, chaque utilisateur dispose de son propre référentiel qui contient l'intégralité de l'historique des validations du projet. Nous n'avons pas besoin de nous connecter au référentiel distant; le changement est simplement stocké sur notre référentiel local. Si nécessaire, nous pouvons pousser ces modifications vers un référentiel distant.



- **Sécurité**

Git est sécurisé. Il utilise la SHA1 (Secure Hash Function) pour nommer et identifier les objets dans son référentiel. Les fichiers et les validations sont vérifiés et récupérés par sa somme de contrôle au moment du paiement. Il stocke son historique de telle manière que l'ID de commits particuliers dépend de l'historique complet du développement menant à ce commit. Une fois publié, on ne peut pas modifier son ancienne version.

- **La vitesse**

Git est très rapide, il peut donc accomplir toutes les tâches en un moment. La plupart des opérations git sont effectuées sur le référentiel local, donc cela fournit une **vitesse énorme**. En outre, un système de contrôle de version centralisé communique en permanence avec un serveur quelque part.

Les tests de performances menés par Mozilla ont montré qu'il était **extrêmement rapide par rapport aux autres VCS**. La récupération de l'historique des versions à partir d'un référentiel stocké localement est beaucoup plus rapide que de la récupérer à partir du serveur distant. **La partie principale de Git est écrite en C**, ce qui **ignore** les surcoûts d'exécution associés à d'autres langages de haut niveau.

Git a été développé pour fonctionner sur le noyau Linux; par conséquent, il est suffisamment **capable de gérer efficacement de grands référentiels**. Depuis le début, la **vitesse** et les **performances** ont été les principaux objectifs de Git.

- **Prend en charge le développement non linéaire**

Git prend en charge le **branchement et la fusion transparents**, ce qui aide à visualiser et à naviguer dans un développement non linéaire. Une branche dans Git représente un seul commit. Nous pouvons construire la structure de branche complète à l'aide de son commit parental.

- **Branchement et fusion**

Le branchement et la fusion sont les **grandes fonctionnalités** de Git, ce qui le distingue des autres outils SCM. Git permet la création de plusieurs branches sans affecter les uns les autres. Nous pouvons effectuer des tâches telles que la création, la suppression et la fusion sur les branches, et ces tâches ne prennent que quelques secondes. Voici quelques fonctionnalités qui peuvent être obtenues par branchement:

1. Nous pouvons **créer une branche séparée** pour un nouveau module du projet, le valider et le supprimer quand nous le voulons.
2. Nous pouvons avoir une **branche de production**, qui a toujours ce qui entre en production et qui peut être fusionnée pour des tests dans la branche de test.
3. Nous pouvons créer une **branche de démonstration** pour l'expérience et vérifier si cela fonctionne.
4. Nous pouvons également le **supprimer** si nécessaire.

Le principal avantage du branchement est que si nous voulons pousser quelque chose vers un référentiel distant, nous n'avons pas à pousser toutes nos branches. Nous pouvons sélectionner quelques-unes de nos succursales, ou toutes ensemble.

- **Assurance des données**

Le modèle de données Git garantit l'intégrité cryptographique de chaque unité de notre projet. Il fournit un ID de validation unique à chaque validation via un algorithme SHA. Nous pouvons récupérer et mettre à jour le commit par ID de com

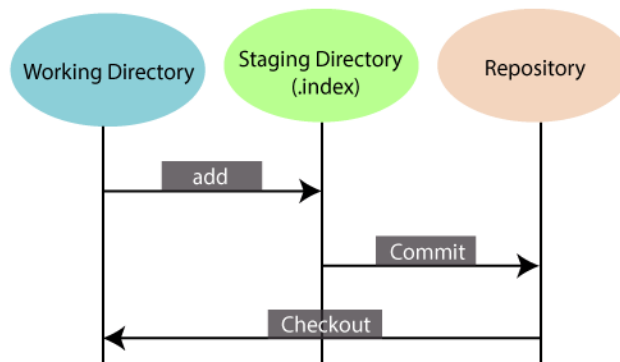
Pourquoi Git?

Nous avons discuté de nombreuses fonctionnalités et avantages de Git qui démontrent sans aucun doute Git en tant que système de contrôle de version leader. Maintenant, nous allons discuter d'autres points sur les raisons pour lesquelles nous devrions choisir Git. La plupart des systèmes de contrôle de version centralisés ne fournissent pas une telle intégrité par défaut.

- **Zone de transit**

La **zone de préparation** est également une **fonctionnalité unique** de Git. Il peut être considéré comme un **aperçu de notre prochain commit**, de plus, une **zone intermédiaire** où les commits peuvent être formatés et révisés avant la fin. Lorsque vous effectuez un commit, Git prend les modifications qui se trouvent dans la zone de préparation et les fait comme un nouveau commit. Nous sommes autorisés à ajouter et supprimer des modifications de la zone de préparation. La zone de préparation peut être considérée comme un endroit où Git stocke les modifications.

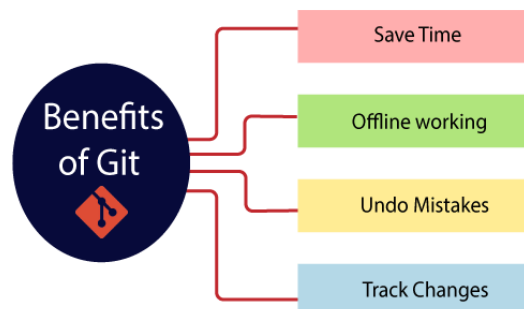
Bien que Git ne dispose pas d'un répertoire intermédiaire dédié dans lequel il peut stocker certains objets représentant les modifications de fichier (blobs). Au lieu de cela, il utilise un fichier appelé index.



Avantages de Git

Une application de contrôle de version nous permet de garder une trace de toutes les modifications que nous apportons aux fichiers de notre projet. Chaque fois que nous apportons des modifications aux fichiers d'un projet existant, nous pouvons les pousser dans un référentiel. Les autres développeurs sont autorisés à extraire vos modifications du référentiel et à continuer à travailler avec les mises à jour que vous avez ajoutées aux fichiers de projet.

Certains avantages importants de l'utilisation de Git sont les suivants:



- **Gain de temps**

Git est une technologie ultra-rapide. Chaque commande ne prend que quelques secondes à exécuter, ce qui nous permet de gagner beaucoup de temps par rapport à la connexion à un compte GitHub et à découvrir ses fonctionnalités.

- **Travail hors ligne**

L'un des avantages les plus importants de Git est qu'il prend en charge le travail hors ligne. Si nous sommes confrontés à des problèmes de connectivité Internet, cela n'affectera pas notre travail. Dans Git, nous pouvons presque tout faire localement. Comparativement, d'autres CVS comme SVN sont limités et préfèrent la connexion avec le référentiel central.

- **Annuler les erreurs**

Un avantage supplémentaire de Git est que nous pouvons annuler les erreurs. Parfois, l'annulation peut être une option de sauvetage pour nous. Git fournit l'option d'annulation pour presque tout.

- **Suivre les changements**

Git facilite avec quelques fonctionnalités intéressantes telles que Diff, Log et Status, qui nous permettent de suivre les changements afin que nous puissions vérifier le statut, comparer nos fichiers ou nos branches.

Visualisation

Visualiser l'historique des validations

Après avoir créé plusieurs **commits** ou si vous avez cloné un dépôt ayant un historique de **commits**, vous souhaitez probablement revoir le fil des événements. Pour ce faire, la commande **git log** est l'outil le plus basique et le plus puissant.

Enregistrement des modifications

git add

Lorsque vous faites une modification sur un fichier, ce dernier n'est pas encore indexé. Cela signifie que les modifications de ce fichier ne seront pas prises en compte lors de votre prochain commit. La commande **git add** vous permet d'ajouter le fichier modifié dans ce «sas» afin qu'il soit pris en compte lors du prochain commit. exécutez la commande suivante:

git add [fichier]

Cette commande indexe le fichier spécifié. exécutez la commande suivante:

git add [répertoire]

Indexe le répertoire et ce qu'il contient. exécutez la commande suivante: **git add ***

Indexe tous les fichiers et répertoires qui ont été modifiés ou créés. exécutez la commande suivante: **git add '*.txt'**

Indexe tous les fichiers dont l'extension est .txt.

Pour voir dans quel état se trouvent vos fichiers (modifications indexées et non-indexées), vous devez lancer la commande **git status**.

git commit

Cette commande permet de valider les modifications indexées. En aucun cas cette commande n'envoie les modifications sur le dépôt distant. exécutez la commande suivante: **git commit**

Cette commande commit toutes les modifications indexées. Une fois lancés, cette commande ouvrira un éditeur dans lequel vous devrez saisir le message de votre commit. Il vous suffit de renseigner les informations souhaitées puis de quitter l'éditeur pour que votre commit soit créé. Pour aller plus vite, vous pouvez utiliser la commande suivante :

git commit -m "Mon message"

Cette commande vous permet de renseigner directement le message du commit sans passer par un éditeur.

Astuces

Plutôt que de lancer 2 commandes pour indexer vos fichiers puis commiter, vous pouvez lancer une unique commande: **git commit -am "Mon commit"**

Cette commande indexe tous les fichiers modifiés ou supprimés (option **-a**) puis commit. Attention, seuls les fichiers déjà versionnés sont pris en compte. Pour les fichiers nouvellement créés, il faudra utiliser la commande **git add** avant de faire un commit.

git stash

Le remisage (stash en anglais) permet de stocker les modifications effectuées pour les ré-appliquer plus tard. Il est utile lorsque vous souhaitez changer de branche sans perdre vos modifications.

Imaginez que vous travaillez sur une branche pour ajouter une grosse fonctionnalité qui touche à de nombreux fichiers. D'un seul coup vous souhaitez corriger une petite fonctionnalité qui se trouve sur une autre branche. Et bien vous ne pourrez pas changer de branche tant que vous avez des modifications en cours.

Vous pourriez faire un commit de ces modifications mais ce n'est pas très propre si vous n'avez pas fini ce que vous êtes en train de faire. Un commit doit comporter des modifications qui ne rendront pas l'application instable. exécutez la commande suivante: **git stash**

Cette commande permet de garder dans un coin les modifications en cours. Maintenant, si vous lancez la commande **git status**, vous vous apercevrez que toutes les modifications ont disparues. Vous pouvez donc changer de branche et faire ce que bon vous semble.

Lorsque vous avez terminé, il vous suffit de lancer la commande suivante pour ré-appliquer les modifications que vous aviez stockées : **git stash apply**

Vous pouvez voir la liste des remisages que vous avez effectués avec la commande : **git stash list**

Vous pourrez voir que le remisage que vous avez fait est toujours présent. Pour supprimer le dernier stash, lancez la commande : **git stash drop**

Astuces

Vous pouvez combiner les 2 commandes **apply** et **drop** en utilisant la commande **git stash pop**. La commande est la suivante : **git stash pop**

Cette commande ré-applique les modifications remisées puis supprimer le remisage.

Retrouver et mettre en attente des modifications

Parfois, vous souhaitez changer de branche, mais vous travaillez sur une partie incomplète de votre projet actuel. Vous ne voulez pas commettre un travail à moitié fait. **Git stashing** vous permet de le faire. La commande **git stash** vous permet de changer de branche sans valider la branche actuelle.

Syntaxe de révision

C'est l'ensemble des commandes Git mis en exécution au cours de la manipulation des données.

Les patches

Maintenance d'un projet

En plus de savoir comment contribuer efficacement à un projet, vous aurez probablement besoin de savoir comment en maintenir un. Cela peut consister à accepter et appliquer les patches générés via **format-patch** et envoyés par courriel, ou à intégrer des modifications dans des branches distantes de dépôts distants. Que vous mainteniez le dépôt de référence ou que vous souhaitiez aider en vérifiant et approuvant les patches, vous devez savoir comment accepter les contributions d'une manière limpide pour vos contributeurs et soutenable à long terme pour vous.

```
git format-patch [-k] [(--o|--output-directory) <dir> | --stdout]
                  [--no-thread | --thread[=<style>]]
                  [(--attach|--inline) [=<boundary>] | --no-attach]
                  [-s | --signoff]
                  [--signature=<signature> | --no-signature]
                  [--signature-file=<file>]
                  [-n | --numbered | -N | --no-numbered]
                  [--start-number <n>] [--numbered-files]
                  [--in-reply-to=<message id>] [--suffix=.<sfx>]
                  [--ignore-if-in-upstream]
                  [--cover-from-description=<mode>]
                  [--rfc] [--subject-prefix=<subject prefix>]
                  [(--reroll-count|-v) <n>]
                  [--to=<email>] [--cc=<email>]
                  [--[no-]cover-letter] [--quiet]
                  [--[no-]encode-email-headers]
                  [--no-notes | --notes[=<ref>]]
                  [--interdiff=<previous>]
                  [--range-diff=<previous> [--creation-factor=<percent>]]
                  [--filename-max-length=<n>]
                  [--progress]
                  [<common diff options>]
                  [ <since> | <revision range> ]
```

Les branches et les sous-branches

Une branche dans Git est simplement un pointeur léger et déplaçable vers un de ces commits. La branche par défaut dans Git s'appelle **master**. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. À chaque validation, le pointeur de la branche master avance automatiquement.

git branch

La commande **git branch** est en fait une sorte d'outil de gestion de branche et de sous branche. Elle peut lister les branches que vous avez, créer une nouvelle branche, supprimer des branches et renommer des branches.

Les fusions

Si vous avez besoin de les y rapatrier, vous pouvez fusionner la branche master dans la branche iss53 en lançant la commande **git merge master**, ou vous pouvez retarder l'intégration de ces modifications jusqu'à ce que vous décidiez plus tard de rapatrier la branche iss53 dans master. Pour faire fusionner deux branches, on utilise la commande suivante: **\$ git merge iss53**

.....Conclusion.....

Git est un système de contrôle de version distribué open source. **Git** est un outil qui permet de gérer différents projets en les envoyant sur un serveur. Ce dernier est connecté à l'ordinateur d'autres développeurs qui envoient leur code et récupèrent le vôtre. Toute personne qui travaille sur un projet est connectée avec les autres, tout est synchronisé. Quant à **Github** et **Gitlab**, il s'agit d'un logiciel, ou plateforme. Leur rôle est d'héberger ces différents projets qui utilisent Git.

Git est un outil qui permet de gérer différents projets en les envoyant sur un serveur. Il est compatible avec plusieurs SE et il est facile d'utiliser ce système de versioning pour la gestion de nos projets informatiques. Il est open source et sa communauté de supervision met à notre disposition une multitude de commandes pour faciliter le bon déroulement de la conservation de nos données sous une plateforme bien sécurisée.