# Git et Github

Les projets informatiques sont parfois chaotiques, alors comment garder une trace de tout son travail ?  
Comment revenir à une version précédente ?  
Et comment réparer ses erreurs ?  
  
Git permet de **suivre les modifications et organiser un projet**. C’est un outil essentiel, que l’on **travaille seul**, **en équipe**, ou même **sur un projet en open source**.

Les objectifs de ce document sont …

* d'**utiliser** **les** **commandes** de base de Git ;
* de **corriger les erreurs courantes** sur GitHub ;
* de **gérer plusieurs versions** sur GitHub ;
* de **collaborer grâce à GitHub** en utilisant les **flux de travail** (*workflows*).

## Contrôle de versions

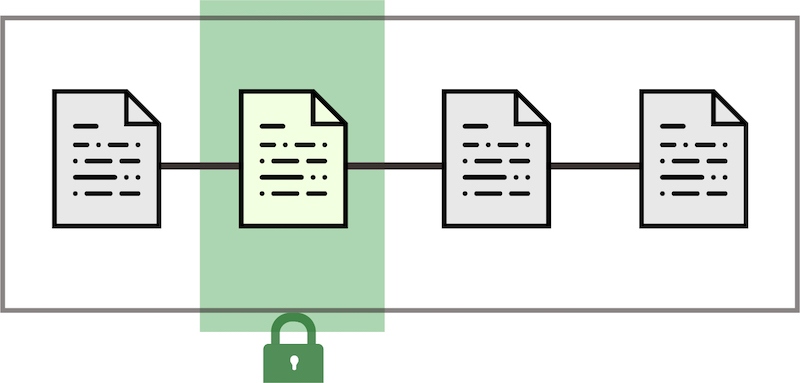
Quel développeur n’a jamais été confronté à un de ces problèmes …

* Ma modification n’a pas fonctionné et j’ai oublié de sauvegarder une copie de mon code avant d’effectuer les modifications ;
* Qui a modifié mon fichier ? Il existe maintenant un bogue ;
* À quoi servent ces nouveaux fichiers ?
* Il ne faut surtout pas toucher pas à ce fichier, je suis en train de le modifier.

Il existe également bien d’autres problèmes, que tout développeur sera susceptible de rencontrer un jour.

### Qu'est-ce que le contrôle de versions ?

Un **contrôleur de versions** est un **programme qui permet aux développeurs de conserver un historique** **des** **modifications** et des **versions de tous les fichiers**.



Si on est seul à travailler sur un projet, le contrôle de versions sera fort utile.   
Il permettra de **garder l’historique des modifications de tous les fichiers**.

Le **contrôle de versions permet de garder en mémoire** **chaque modification de chaque fichier** qui a eu lieu, **pourquoi elle a eu lieu** et **par qui**.

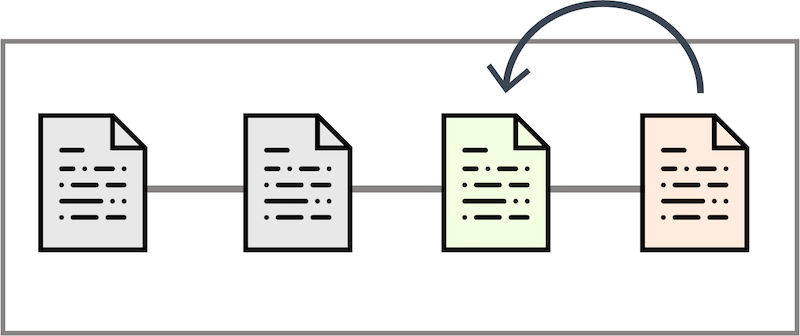
Le contrôle de versions permet d’**assembler les modifications de deux personnes travaillant simultanément sur un même fichier**, afin d’**éviter** **d’écraser** **le travail des autres**.

Ces outils ont donc **trois grandes fonctionnalités** …

* **travailler à plusieurs** sans risquer de supprimer les modifications des autres collaborateurs ;
* **revenir en arrière** en cas de problème ;
* **suivre l’évolution étape par étape d’un code source** pour retenir les modifications effectuées sur chaque fichier.

### Retour à une ancienne version

L’intérêt de ce type d'outil est donc de pouvoir **revenir sur n’importe quelle version en cas de bogue dans l’application**.



Le contrôle des versions est un outil extrêmement utile dans le cadre d'un développement personnel comme dans le cadre d’un projet mutualisé.

|  |  |
| --- | --- |
|  | **Git est de loin le système de contrôle de versions le plus largement utilisé aujourd'hui.** |

Par sa **structure décentralisée**, Git illustre parfaitement ce qu'est un système de contrôle de versions décentralisé. **Plutôt que de consacrer un seul emplacement pour l'historique complet des versions** du logiciel, dans Git, chaque **copie de travail du code est également un dépôt qui contient l'historique complet de tous les changements**.

### Git et le travail d’équipe

**Voici un exemple concret** …

Alice et Bob travaillent sur un même projet depuis un mois et jusque-là tout se passait bien.   
Hier, leur client leur a demandé de livrer en production leur travail en urgence.   
Alice a réalisé au plus vite les dernières modifications, a enregistré les fichiers et a envoyé le tout au client.

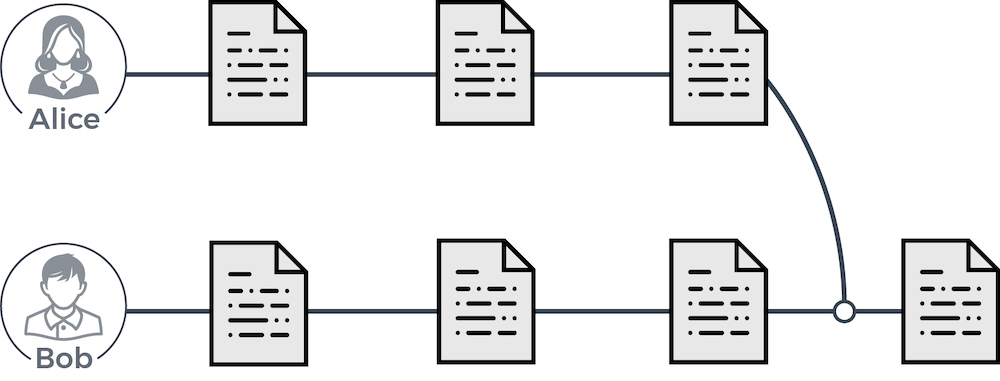
Le lendemain, le client les appelle, très énervé : rien ne fonctionne comme prévu.   
Alice et Bob ne comprennent pas, ils avaient séparé les tâches et tous les deux avaient fait correctement le travail.   
Toutefois, Alice, sans le savoir, a écrasé le code qu’avait réalisé Bob lorsqu’elle a fait ses modifications de dernière minute ; et en enregistrant, elle a perdu le travail de Bob.   
Bob n’ayant pas de copie sur disque local, il a travaillé pendant un mois pour rien, car il lui est impossible de récupérer son travail.

Sur disque **local signifie sur sa station de travail machine**, par **opposition en ligne**.

Cela aurait pu être évité avec un gestionnaire de codes sources.

**Voici le même exemple concret … révisé ...**

Avec un gestionnaire de codes sources (Git). **Alice et Bob travaillent sur un même projet** et ont **initialisé Git pour leur projet**.   
Grâce à Git, **chacun modifie ses fichiers**, et **chacun peut envoyer** **et recevoir les mises à jour des fichiers à n’importe quel moment**, et cela **sans écraser les modifications de l’autre**.   
Des modifications même en urgence n’auront aucun impact sur le travail de l’autre !



**Travail mutualisé**

Un des aspects passionnants lorsque l’on fait du développement, c'est que l’on peut **apporter sa pierre à plein d'édifices en contribuant à des projets open source**.

**Open source signifie que le code source d'un logiciel est public et accessible**.   
Le logiciel en question peut alors être **modifié et diffusé par n'importe quel individu**.

Travailler sur un projet open source est stimulant et permet de développer rapidement ses compétences.   
Que ce soient le codage, la conception de l’interface utilisateur, la conception graphique, la rédaction ou l’organisation, il existe une tâche pour tous sur un projet open source.

### Git ou GitHub

**Git est l'outil qui permet de créer un dépôt local** et de **gérer les versions de ses fichiers**, alors que **GitHub est un service en ligne qui permet héberger son dépôt**, qui sera par conséquent **distant** (puisqu'il ne sera pas sur la station de travail locale).

Un projet réalisé sous GitHub permet de **suivre les étapes suivantes pour collaborer sur un projet** open source …

* le premier réflexe est de **regarder la documentation**.   
  Il y a souvent tout un tas d’informations sur la manière de collaborer au mieux au projet ;
* dans un second temps, on devra **rapatrier le dépôt distant sur son dépôt local**.   
  Sur le dépôt local, on pourra réaliser les modifications ;
* une fois **toutes les modifications réalisées**, on pourra **envoyer les modifications en ajoutant des messages de description**.   
  Il faut que la personne gérant le dépôt distant comprenne les modifications qui ont été faites.   
  En allant de nouveau sur le dépôt distant, on pourra maintenant **soumettre ses modifications**.

# Utilité des dépôts

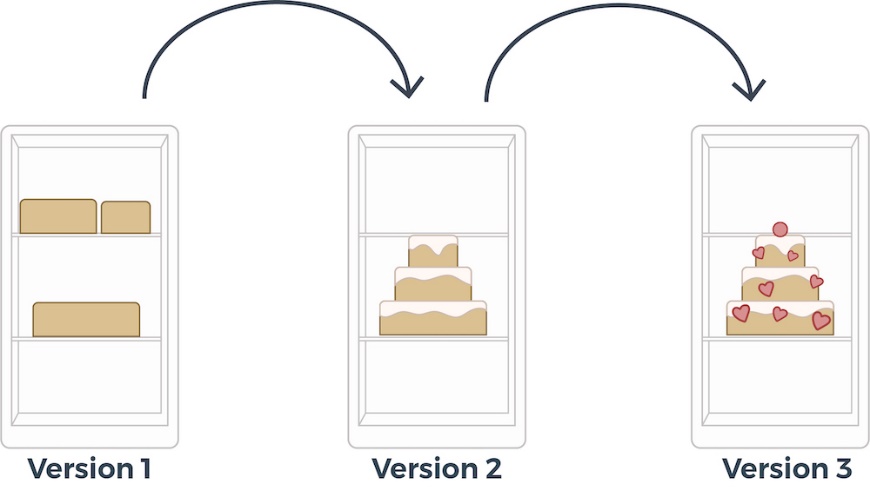
## Différence entre dépôt local et dépôt distant

Voici un premier exemple : La réalisation d'un gâteau.

On va dire que le réfrigérateur est le dépôt local. C'est l'endroit où l'on va stocker ses préparations au fur et à mesure.

Dans un premier temps, on réalise la pâte, on la stocke au réfrigérateur, puis on réalise la crème, on l'assemble avec la pâte et on stocke l'ensemble au réfrigérateur.

Finalement, on réalise la décoration du gâteau, on finalise le gâteau en y ajoutant les décorations et on le remet au réfrigérateur.



Ceci résume le **fonctionnement d’un dépôt local**.   
On **réalise une version**, que l'on va **petit à petit venir améliorer en stockant toutes ces versions**.

Voici un second exemple : L'image d'un manuel de classe …

Sur un manuel, il y a souvent écrit première édition ou cinquième édition.   
L'éditeur a réalisé une version et petit à petit, il a corrigé l'orthographe, ou modifié le contenu.

Les versions dans Git fonctionnent de la même manière. On **réalise une première version**, que l'on **améliore au fil du temps**.

|  |  |
| --- | --- |
|  | Le **dépôt distant est un peu différent**. Il **permet de stocker certaines versions** qu'on lui aura envoyées, afin de **garder un historique délocalisé**.  L’avantage d’un dépôt est qui **si une version locale malencontreusement est modifiée ou encore supprimée**, une **copie distante existera toujours sur GitHub**. Mais en plus de les stocker, il est aussi possible de les **rendre publics afin que cloud** et **chacun pourra alors venir y ajouter ses évolutions**. |

Lors de la **collaboration à des projets**, il est **nécessaire de disposer de dépôts distants**. Le dépôt distant est un **historique du projet hébergé sur Internet** ou **sur un réseau**. On peut avoir **plusieurs dépôts distants** avec des **droits** **différents** (**lecture seule**, **écriture**, …).

Un dépôt Git est donc un **entrepôt virtuel pour un projet**. Il permet d'**enregistrer** **les** **versions** **du** **code** et d'y **accéder au besoin**.

Le dépôt distant est un type **de dépôt qui devient réellement important** (voire indispensable) lorsque l’on **travaille à plusieurs sur le même projet**, puisqu’il permet de **centraliser le travail de chaque développeur**.   
C’est pourquoi il est fortement recommandé d’utiliser GitHub pour ses dépôts distants.

Sur GitHub, on peut créer des **dépôts distants publics**, **mais aussi privés**.

Sur un **dépôt public**, les **personnes pourront collaborer au projet** alors que sur un **dépôt privé**, un **seul développeur possède un accès à son travail**.

Le **principal intérêt de Git** est le **suivi des modifications**, mais aussi la **sauvegarde de ces projets**.   
C’est pourquoi il est conseillé de toujours commencer par **copier ses sources sur un dépôt distant**, si possible **situé à l’extérieur de ses locaux**.  
C’est aussi sur le **dépôt distant que toutes les modifications de tous les collaborateurs seront fusionnées**. La **majeure partie du travail se fera sur le dépôt local** qui est un **clone du dépôt distant**. C’est sur ce **dépôt local que l’on fait toutes les modifications de codes**, les **créations de branches** (il y en a aussi sur le dépôt distant), et **ses validation (*commits*)** ; et seulement **lorsque les modifications seront prêtes à être partagées à l’équipe**, on les **téléverse sur le dépôt distant**.

Les **dépôts sont utiles** si …

* on **travaille à plusieurs sur un projet** ;
* on **souhaite collaborer à des projets open source** ;
* on **souhaite conserver un historique de ses projet** ;
* on **veut pouvoir retrouver par qui a été faite chaque modification** ;
* on veut **savoir pourquoi chaque modification a eu lieu**.

### Outils existants

Il existe **plusieurs outils intéressants** (GitHub, GitLab, Bitbucket, SourceForge) …

|  |  |
| --- | --- |
|  | **GitHub** GitHub est un outil de communication et de collaboration entre plusieurs développeurs (ou tout autre personne qui écrit du texte). C’est une **interface web créée pour faciliter l’interaction avec Git**. L’avantage de GitHub, c’est que depuis quelques années GitHub est **devenu le portfolio des développeurs**. GitHub est considéré comme un **véritable réseau social** et permet de **contribuer à des projets open source**. GitHub **fonctionne par abonnement** mais il existe un **abonnement gratuit**. |
|  | **GitLab** Gitlab est la **principale alternative à GitHub** depuis le rachat de GitHub par Microsoft. GitLab **fonctionne avec une version gratuite à installer** **sur son propre serveur** ou une **version cloud payante**. |
|  | **BitBucket**  BitBucket est la version de Atlassian. **Payante**, elle plaira néanmoins aux **habitués de la gestion de projet sous Atlassian**.  BitBucket conviendra aussi bien aux **étudiants** ou **petites équipes** qu’aux **grands groupes**. |
|  | **SourceForge** SourceForge est l’**ancêtre** **dans** le **domaine**.  Il a été créé **10 ans avant les autres**, afin de **gérer à la base des projets open source**.  SourceForge intègre un outil de **suivi des bogues et un répertoire de code intégré**.  Il **n’est plus très populaire** depuis ces dernières années. |

# Github

## Création d’un compte GitHub

**GitHub est un service en ligne** permettant d’héberger ses dépôts distants.

Pour **créer un compte GitHub**, on accède à leur site web …  
**https://github.com/**

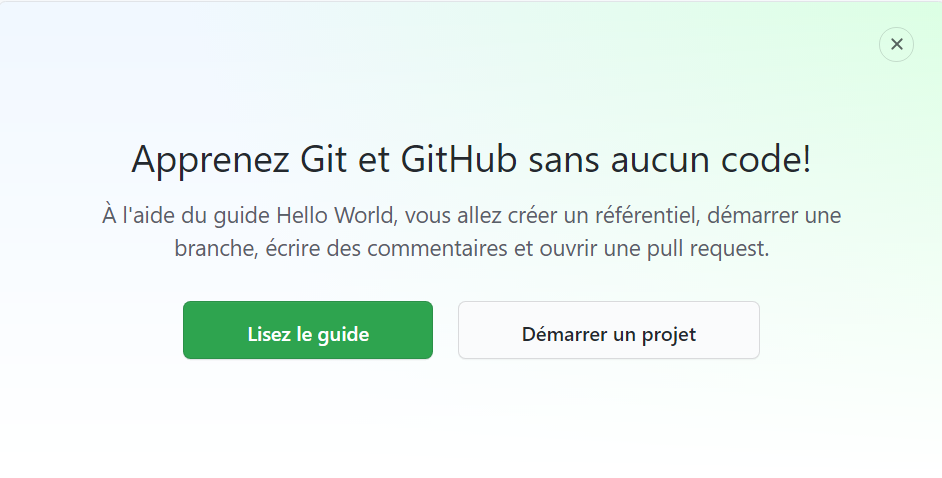
Le **bouton S’inscrire** (*Sign up*) permettra de **créer un compte**.  
Pour la **création d’un compte**, on demandera un **nom d’utilisateur**, un **courriel** et un **mot de passe**.

Une fois ces informations remplies, on doit choisir le **type d’abonnement** (**gratuit** ou **pro**).   
La **principale différence entre les deux offres** est que la **première est destinée aux particuliers**, ou aux **équipes de moins de 3 collaborateurs**, alors que la **seconde offre** est destinée aux **plus grandes équipes**.   
S’il s’agit, toutefois, d’un **projet open source**, il n’y a **aucune limitation sur le nombre de collaborateurs**.

## Tour de GitHub

GitHub est assez facile à prendre en main et simple d’utilisation.

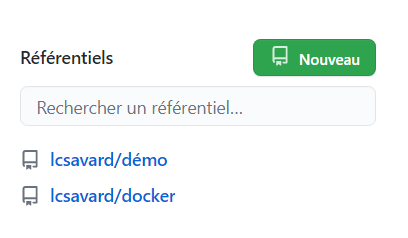
L’accueil propose la **consultation d’un guide ou de démarrer un nouveau projet**.



On peut consulter le **tableau de bord personnel** afin de …

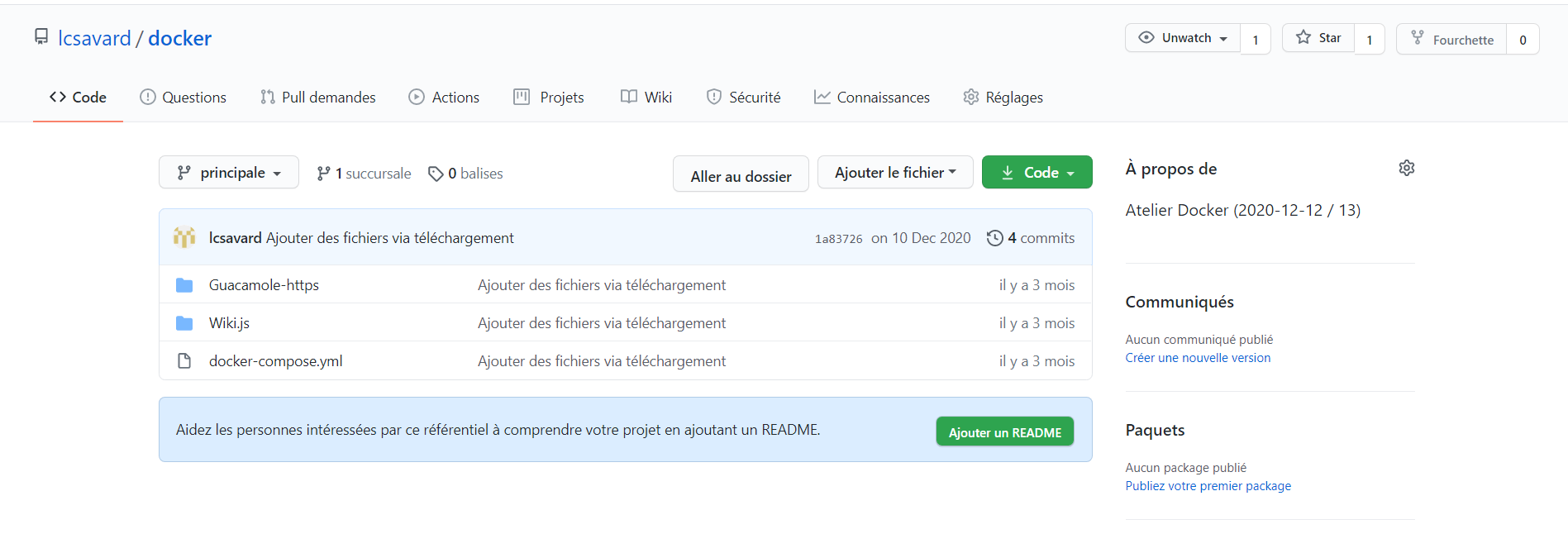
* **suivre les problèmes** et **extraire les demandes sur lesquelles on travaille ou que l’on suit** ;
* **accéder aux principaux référentiels** **et** **pages d'équipe** ;
* **rester à jour sur les activités récentes des organisations et des référentiels** auxquels on est abonné   
  et
* **explorer les référentiels recommandés**.

L’interface **Repositories** est **l’emplacement où l’on peut créer et retrouver ses dépôts existants**.



Pour **créer un projet**, il suffit de cliquer sur **nouveau**.

Pour **consulter un projet existant**, il suffit de **le sélectionner dans la liste** …



Sur son **profil**, il est possible d’**éditer** **ses** **informations**, mais aussi voir le **total de ses contributions sur les différents projets.**

L’**onglet Pull requests**, quant à lui, permet de **réaliser des demandes de téléversement** (*pull*).   
Les **demandes de pull** (extractions) permettent d'**informer** **les** **autres** **sur** **les** **modifications** que l’on **appliquées à une branche d'un référentiel** sur GitHub.   
Une fois qu'une demande d'extraction est ouverte, on peut discuter et examiner les modifications éventuelles avec les collaborateurs, et ajouter des validations de suivi avant que ses modifications ne soient fusionnées dans la branche de base.

Dans la **section Activité récente** de son fil d’actualité, on peut rapidem**ent rechercher et suivre les problèmes récemment mis à jour**, et **extraire les demandes** sur lesquelles on travaille.   
Sous **Activité récente**, il est possible de prévisualiser jusqu'à 12 mises à jour récentes effectuées au cours des deux dernières semaines.

Une **activité est récente lorsque** …

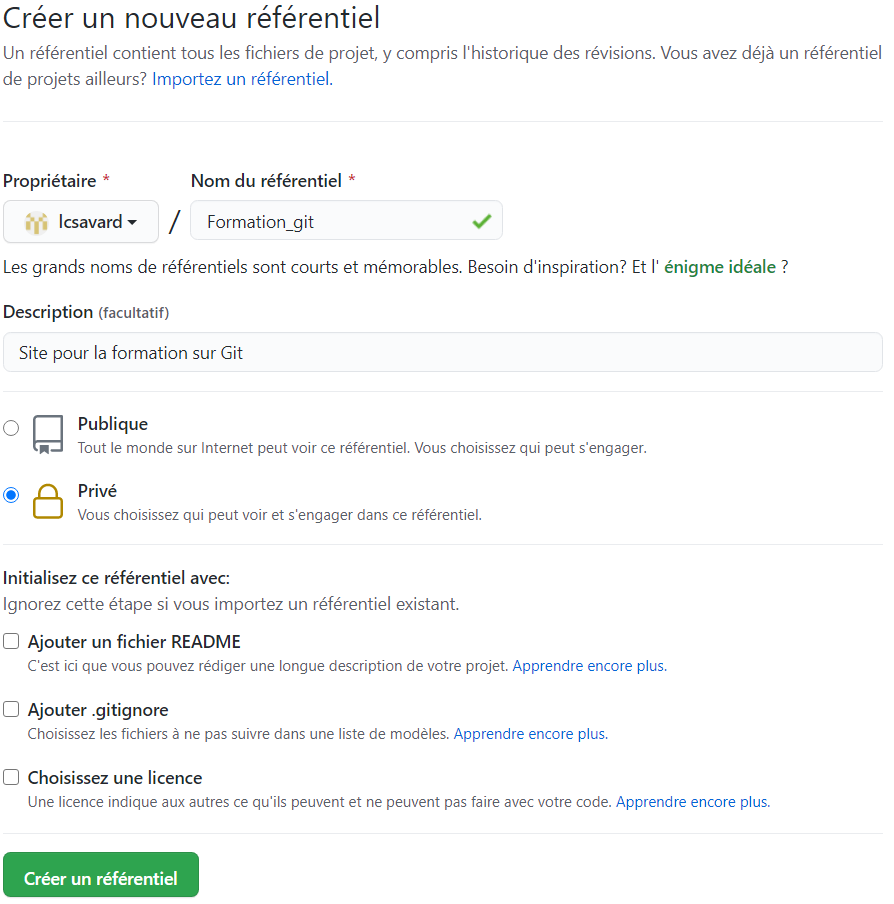
* **on a ouvert un problème** ou une **demande d'extraction** ;
* **quelqu'un a commenté un problème** ou **tiré une demande** que l’on a ouverte ;
* **son problème ou demande** **d'extraction** a été **rouvert** ;
* **un avis a été demandé** sur une **demande de tirage** ;
* **on a été affecté à un problème** ou à **une demande d'extraction** ;
* on a **référencé un problème** ou une **requête d'extraction à l’aide un commit** ;
* on a **commenté un problème** ou **une demande d'extraction**.

Un des derniers points importants sur GitHub est la fonctionnalité **Explore**.

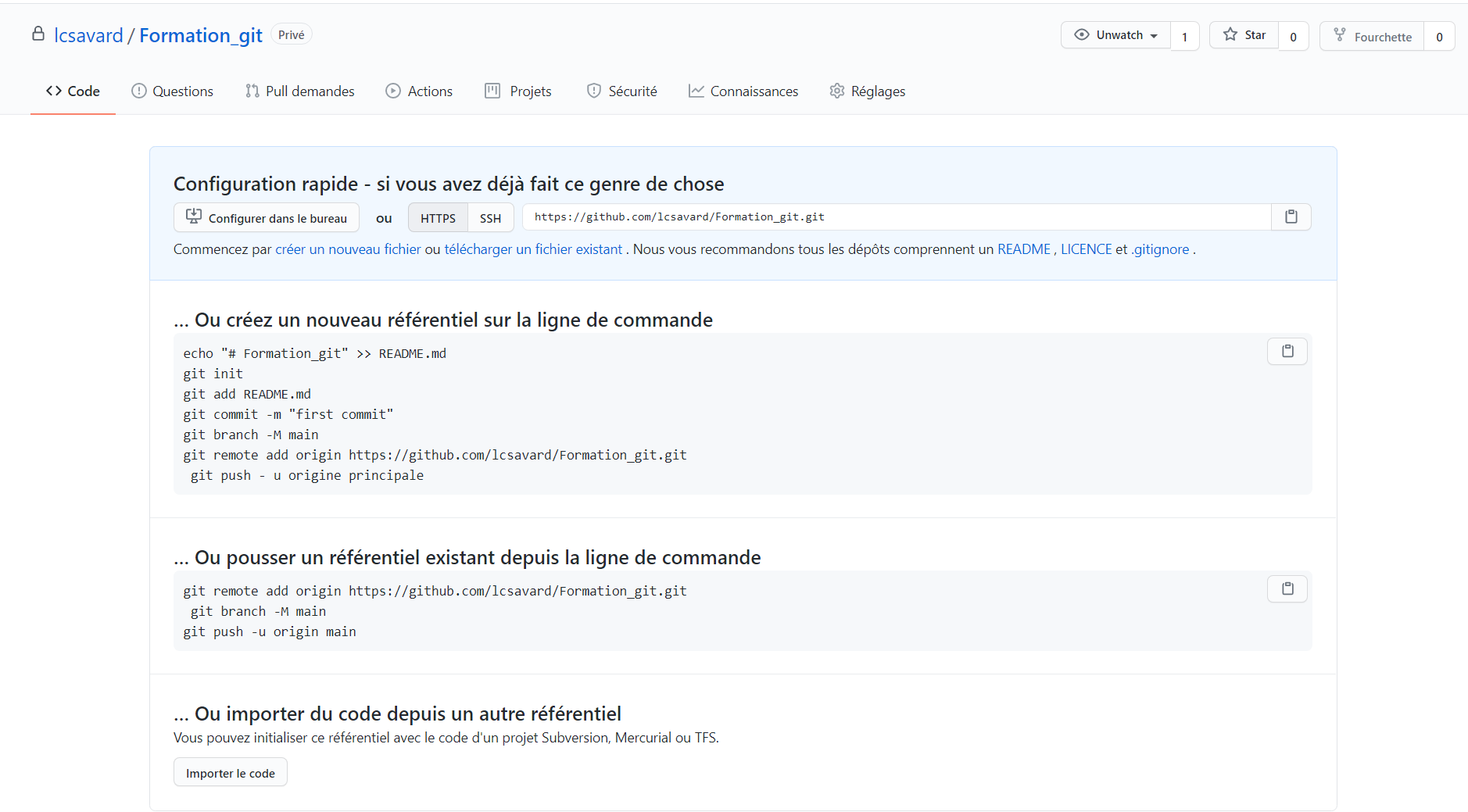
À l’aide d’Explore, on peut **trouver de nouveaux projets open source intéressants** sur lesquels travailler, en parcourant les projets recommandés, en se **connectant à la communauté GitHub** et en **recherchant des référentiels par sujet ou par libellé**.

## Création d’un nouveau projet

Pour **déposer un projet sur GitHub**, on doit **créer un référentiel** dans lequel il pourra être installé.  
Il suffit de **répondre aux différentes questions**.

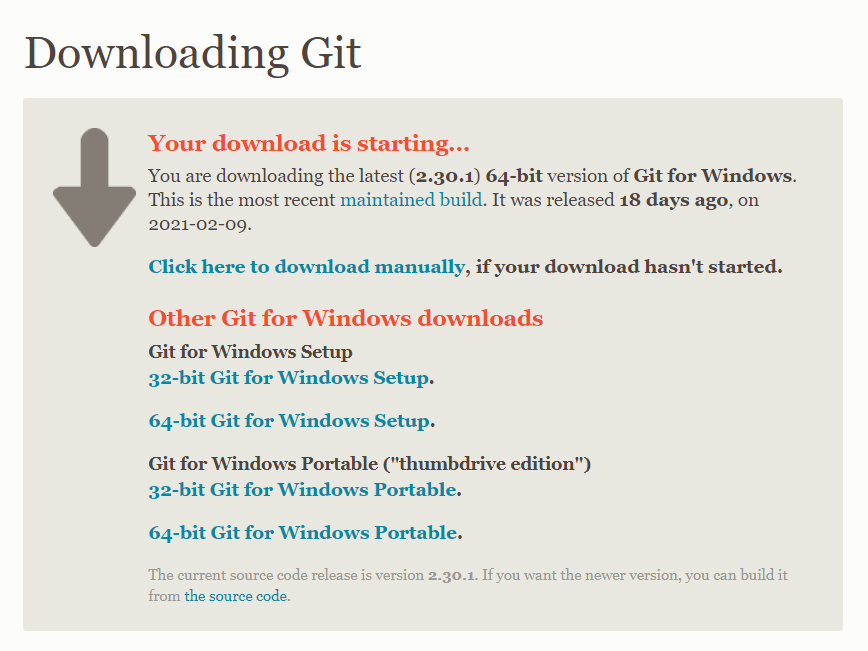


Voici le **résultat à la suite de la création** …



# Mise en place d’un référentiel git

Afin d'**installer** **Git**, il faut dans un premier temps le **télécharger**.  
On peut trouver l’application sur le site …  
<https://git-scm.com/downloads>

**Pour Windows** …  


L’installation est intuitive.  
Les **options par défaut sont habituellement les plus standards**.

**Pour Linux** …

Il est plus simple d'installer Git sur Linux en utilisant le **gestionnaire de paquets de sa distribution Linux**.   
Il est aussi **possible de le construire à partir des sources**. On trouve les **archives tar sur kernel.org**.   
La **dernière version est la 2.30.1**.

**Debian/Ubuntu**Pour la dernière version stable de votre version de Debian/Ubuntu …  
**>> apt install git**

Pour Ubuntu, ce PPA fournit la dernière version stable en amont de Git …  
**>> add-apt-repository ppa:git-core/ppa  
>> apt update; apt install git**

**RHEL**Pour les distributions RHEL et ses dérivés …  
**>> yum install git**ou  
**>> dnf install git**

**Arch Linux**Pour la distribution Arch Linux …  
**>> pacman -S git**

**openSUSE**Pour les distributions SUSE et openSUSE …  
**>> zypper install git**

Pour la version Windows, il existe deux interfaces …

* **Interface CLI** …  
  git bash ;
* **Interface GUI** …  
  git GUI.

Il est préférable d’utiliser l’**interface** **CLI** **car il est identique sur toutes les plates-formes**.

**Sous Windows** …  
On recherche **git bash**.

**Sous Linux** …  
**>> git  
git >>**



**Initialisation de Git**

La première chose à faire est de **configurer son identité**.   
Ces **informations se retrouve en haut à droite de la page principale de Git**.



Dans un premier temps, il faut s’identifier …

Il faut donner son **nom d’utilisateur et son adresse de courriel** …

**>> git config --global user.name <Nom d’utilisateur>  
>> git config --global user.name tux  
>> git config --global user.email <Adresse de courriel>  
>> git config --global user.email tux@cmaisonneuve.qc.ca**

**Remarques** …  
Il est possible d’utiliser la **touche TAB pour la complétion des commandes**.  
Grâce à l’**option** --**globa**l, on n’aura **besoin de le faire qu'une fois**.

C'est une **information importante car toutes les validations dans Git** utilisent cette information et elle est **indélébile dans toutes les validations que l’on pourra réaliser**.

Afin de vérifier que ses **paramètres aient bien été pris en compte**, et **vérifier les autres paramètres** …  
**>> git config --list**

Il est recommandé d’activer les couleurs afin d’améliorer la lisibilité des différentes branches.   
Pour cela, on peut entrer les trois lignes suivantes dans Git Bash …  
**>> git config --global color.diff auto  
>> git config --global color.status auto  
>> git config --global color.branch auto**

Par défaut, Git utilisera vim comme éditeur et vimdiff comme outil de convergence (*merge*).   
On peut les modifier …  
**>> git config --global core.editor notepad++  
>> git config --global merge.tool vimdiff**

Si on travaille sous Linux, on peut choisir nano ou pico à la place de vim.

**Après avoir modifié les paramètres de base**, on peut **créer un premier dépôt local**.   
Pour ce faire, **deux solutions possibles** …

* **créer un dépôt local vide** pour accueillir un nouveau projet ;
* **cloner un dépôt distant**, c’est-à-dire rapatrier tout l’historique d’un dépôt distant en local, afin de pouvoir le travailler.

Pour le logiciel de gestion de versions Git, un **dépôt représente une copie du projet**.  
Chaque station de travail d’un développeur qui travaille sur le projet possède donc une copie du dépôt.  
Dans chaque dépôt, on trouve les fichiers du projet ainsi que leur historique.

### Création d’un dépôt vide

On va créer, dans un premier temps, un **dossier sur le disque** (nommé Projets) …

Dans l’interface GLI de git …

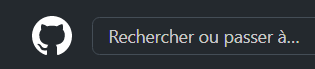
On **accède au dossier et on initialise le projet** …  
**>> mkdir –parent --verbose documents/projet  
>> cd documents/projet  
~/Documents/Projets/ >>  
>> git init  
Initialized empty Git repository in C:/Users/TUX/Documents/Projets/.git/**

Le **dépôt est initialisé**. Un **dossier caché .git a été créé**.

# Accès à un dépôt distant

Pour **accéder à un dépôt distant** et d’en **télécharger une copie** (cloner) **en local**, il faut, dans un premier temps, **récupérer l’URL du dépôt distant**.

On effectue la recherche d’un projet …



On recherche …  
**Fomation\_git**

On copie l’URL …  
**https://github.com/lcsavard/Formation\_git.git**

Dans git bash, on entre la commande suivante …  
**>> git remote add tux https://github.com/lcsavard/Formation\_git.git**

**tux représente le nom court que l’on utilisera ensuite pour appeler le dépôt**.   
Un **nom court et simple est toujours plus facile**.

Cette **ligne ne permet pas de cloner le dépôt**, mais permet de **dire au dépôt que l’on pointe vers le dépôt distant**.

### Téléchargement du dépôt en local

Maintenant que le dépôt local pointe sur le dépôt distant, il faut **cloner son contenu et le dupliquer en local**.

Afin de réaliser le clonage …  
**>> git clone** [**https://github.com/lcsavard/Formation\_git.git**](https://github.com/lcsavard/Formation_git.git) **Cloning into 'Formation\_git'...  
warning: You appear to have cloned an empty repository.**

Cela devrait afficher quelques lignes confirmant le résultat de la commande.

On devrait maintenant avoir un **nouveau répertoire** (ayant le nom du projet) et dans ce dossier, tous les fichiers.   
La **commande ls permet de vérifier le tout** …  
**>> ls  
Formation\_git**

## Système de branches

Le principal atout de Git est son système de branches.  
Ce **système de branches est une des principales fonctionnalités de** git.

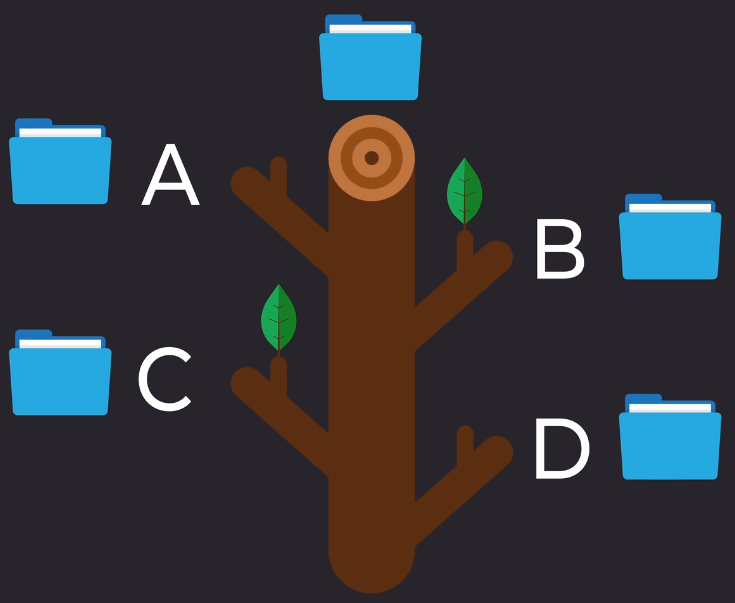
Les **différentes branches correspondent à des copies du code principal à un instant** **T**.  
Cette fonctionnalité permet de **tester tous les changements sans que cela impacte le code principal**.

Sous Git, la **branche principale est appelée la branche master**.   
Lorsque l’on **commence à faire des commits**, la **branche principale pointe toujours vers le dernier commit** que l’on a **effectué**.   
**À la suite de chaque commit**, le pointeur **de la branche principale avance automatiquement**.

Toutefois, la **branche principale de Git n'est pas une branche spéciale**.

C’est cette **branche qui représente la version finale du projet**, **la plus à jour**, celle qui **inclut toutes les modifications**.   
Le but est de **ne surtout pas réaliser les modifications directement sur cette branche**, mais de **réaliser les modifications sur d’autres branches**, et **après tests**, les **intégrer sur la branche principale**.

Si une application fonctionne parfaitement et elle est en production ; y toucher serait prendre le risque de la déstabiliser.   
C’est ici que l’on peut se rendre compte de l’utilité des branches. Il est **possible de créer une branche sans toucher à l’application en production qui fonctionne parfaitement**. Une fois que toutes les **modifications auront été testées**, on pourra les **déployer sans crainte** (et dans le **pire des cas**, **revenir en arrière simplement**).



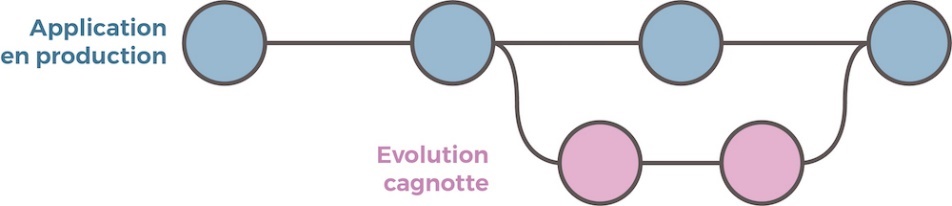
### Branche principale

Avec Git, **aucun problème de fusion**.  
On n’a **pas besoin de connaître tous les bouts de code qui ont été modifiés**.

git va créer une **branche virtuelle**, **mémoriser tous les changements**, et seulement quand on le souhaite, les **ajouter à l’application principale**. Il va **vérifier s'il n’y a pas de conflits avec d’autres fusions**.

Afin de **connaître les branches présentes pour le projet** …  
**>> git branch  
\* main**

L’**étoile signifie** **que c’est la branche sur laquelle on se situe** et que c’est sur celle-ci qu'actuellement on réalise ses modifications.



**Branche des modifications**

**Application en production**

Il est **fortement conseillé de créer une branche si une modification va être longue**, qu’elle peut **avoir des impacts**, qu’elle n’est pas simple ou que on ne voit pas tout de suite comment faire la modification.

Il est souvent **préférable de créer une branche pour une modification**. La **création prend peu de temps** et **économise** **beaucoup de temps de galère** si on fait des **bêtises sur la branche principale** (*master*).

Pour **créer une branche** …  
**>> git branch documents**Cette commande va **créer la branche modification en local** (elle ne va pas être dupliquée sur le dépôt distant).

**>> git branch  
 documents  
\* main**

On peut maintenant **voir la nouvelle branche et la branche master**. La petite **étoile est toujours sur la branche principale** (*master*). La **branche a été créée mais on n’a pas encore basculé** sur celle-ci.

Pour **basculer de branche** …  
**>>** **git checkout documents**  
**Switched to branch 'documents'**

**>>** **git branch  
master  
\* documents**

Une **branche fonctionne comme un dossier virtuel**.   
Avec la **commande git checkout**, on va être **téléporté dans le dossier virtuel documents**.   
On **demeure physiquement dans le dossier premier\_projet**, mais **virtuellement on se situe dans la branche documents**.

On peut **désormais réaliser le travail** (les changements) **sans toucher à la branche principale** qui abrite le code principal (les fichiers).

On peut **rebasculer si besoin à tout moment sur la branche principale**, **sans impacter les modifications de la branche documents**.

### Réalisation d’un commit

Après avoir réalisé des changements (ajouts de fichiers par exemple) sur la branche documents et il faut **demander à Git de les enregistrer**.

Si on a ajouté des fichiers il faut effectuer une opération supplémentaire …  
**>> git add <Fichier >   
>> git add base.rexte**

**Remarque** …  
Si notre projet contient plusieurs nouveaux fichiers, il faut les ajouter à l’index (stage)  
On utilise alors la commande …  
**>>** **git add -A**ou  
**>>** **git add --all**   
Il est possible d’utiliser les globs.  
Exemple …   
**>> git add \***Il est aussi possible de passer la commande.

Un **commit est tout simplement un enregistrement de son travail à un instant T sur la branche courante où l’on est situé**.

Pour ce faire, on passe à git la **commande de confirmation** …  
**>> git commit -m "<Message>"  
>> git commit -m "Ajout des documents de base"**

**Remarque** …  
L’**option -m** ou **–message=<Description>** permet d’ajouter un message de validation.

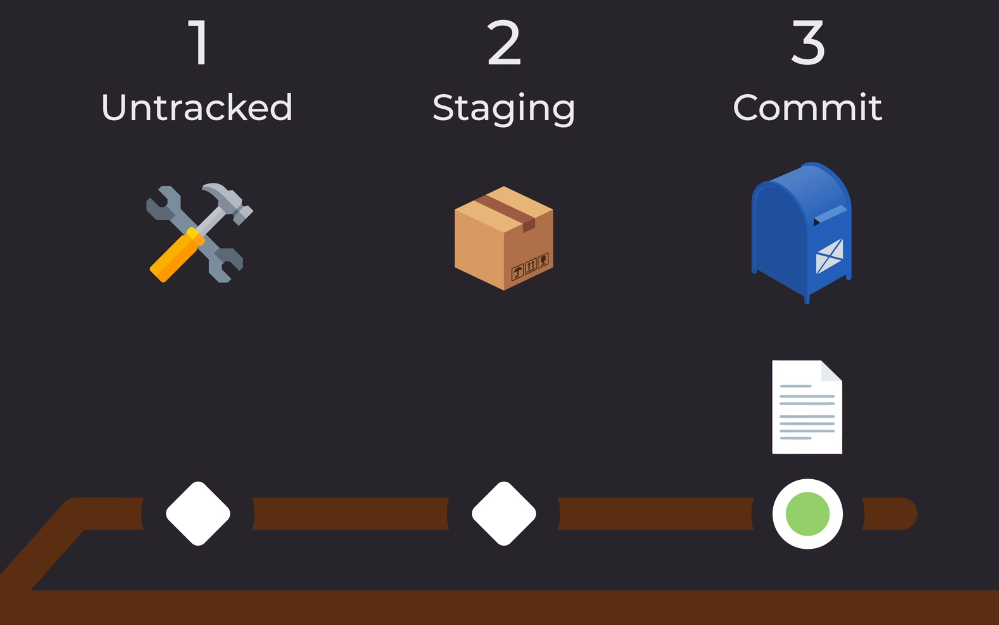
Les **modifications sont maintenant enregistrées** avec comme description ou message “ Ajout des documents de base ”.   
La **description est très importante pour retrouver le fil de les commits**, et **revenir sur un commit en particulier**.

**Aide**

Il est toujours possible de demander à git de l’aide …

**>> git --help**Affiche les sous-commandes de git.  
**>> git commit --help**Affiche de l’aide sur la commande git commit.

Voici un **résumé des commandes de mise à jour** …



1. **Untracked**  
   On effectue les modifications au projet ;
2. **Staging**  
   On emballe les modifications  
   On doit utiliser la comme **git add** ;
3. **Commit**  
   On effectue la confirmation des modifications …   
   On doit utiliser la commande **git commit "<Description>"**.  
   Il est important de donner une description.

# Corrections des erreurs en local

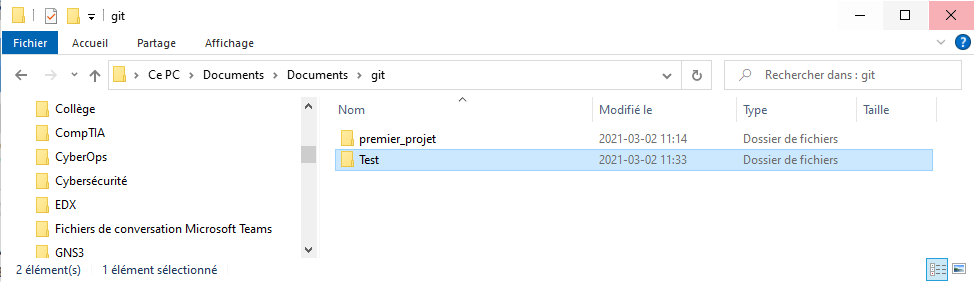
Git est un outil performant, mais on a vite fait …

* de **créer une branche alors qu’on ne le souhaitait pas** ;
* de **modifier la branche principale** (*master*)   
  ou encore
* d’**oublier** **des fichiers dans ses commits**.

## Corrections des erreurs

Voici un exemple.

**Création d’un dépôt Git nommé Test** …

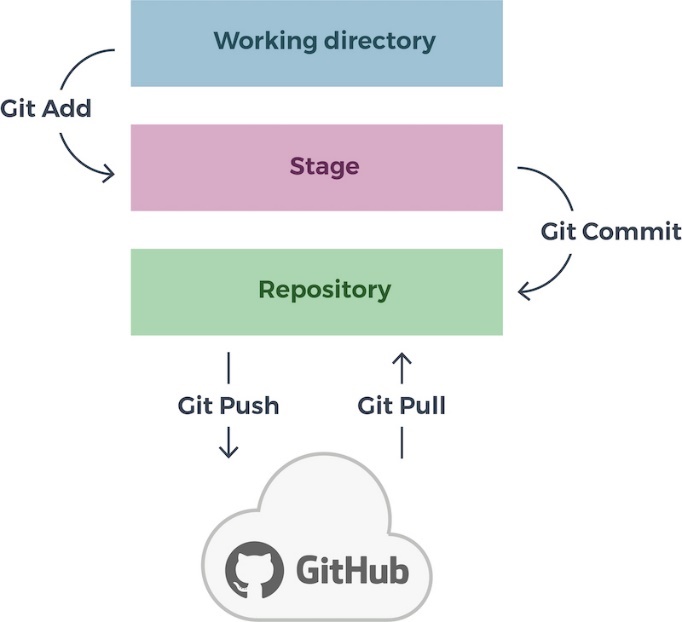


On initialise le dépôt git …  
On se déplace dans le dossier et …  
 **>> git init**

Le dépôt est maintenant initialisé, et si on fait apparaître les dossiers masqués, on peut voir le dossier .git.

Git **gère les versions des travaux locaux à travers 3 zones locales majeures** …

* le **répertoire de travail** (*working directory/WD*) ;
* l’**index** (*stage*) ;
* le **dépôt local** (*Git directory/repository*).



# Téléversement du projet

**git status**

La **commande git push permet d'envoyer** (ou de pousser) **les valdations** (commits) **d’une branche locale depuis le référentiel Git local vers le référentiel distant**.

Pour pouvoir pousser vers le référentiel distant, on **doit s’assurer que toutes les modifications apportées au référentiel local sont validées** …  
**>>** **git status  
>> git push <repo name> <branch name>**

## Téléverser vers un référentiel et une branche distants spécifiques

Afin de pousser le code, on doit d'abord cloner un référentiel sur la machine locale …  
**>>** **git clone https://github.com/<git-user>/<repo-name> && cd <repo-name>**  
**>>** **git clone** [**https://github.com/lcsavard/docker.git**](https://github.com/lcsavard/docker.git) **&& cd docker**

On vérifie le résultat de la commande précédente …  
**>>** **ls -l  
total 8  
drwxr-xr-x 1 LSAVARD 197121 0 Mar 2 12:33 Guacamole-https/  
drwxr-xr-x 1 LSAVARD 197121 0 Mar 2 12:33 Wiki.js/  
-rw-r--r-- 1 LSAVARD 197121 601 Mar 2 12:33 docker-compose.yml**

On fait les modifications souhaitées …  
**>>** **nano description.texte**  
Écrire une description  
**>>** **nano version.texte**Écrire des commentaires

On ajoute les fichiers à l’index …  
**>>** **git add --all**

On valide les modifications au projet …  
**>>** **git commit -m "Ajout du fichier description.texte au projet"**

On vérifie si le dépôt local est propre (clean) …  
**>>** **git status**  
**On branch main  
Your branch is ahead of 'origin/main' by 1 commit.  
 (use "git push" to publish your local commits)  
nothing to commit, working tree clean**

On téléverse les modifications au dépôt distant …  
**>>** **git push -u origin main**

Il suffit de retourner sur le site web de Gtihub afin de vérifier le résultat.

# Résumé des commandes



# Références

Ce document est un dérivé du document disponible sur Openclassroom …  
<https://openclassrooms.com/fr/courses/5641721-utilisez-git-et-github-pour-vos-projets-de-developpement>

## Autres références

Fusions (merges)  
<https://www.atlassian.com/fr/git/tutorials/using-branches/git-merge>

# Licence

