Git : installation et prise en main.

Plan du cours :

I. Présentation

Git, un logiciel open source de gestion de versions qui est très populaire chez les personnes amenées à manipuler du code : **les développeurs, les personnes DevOps,** mais également les personnes adeptes de **Scripting**.

Sans un outil adapté, la gestion des versions pour un projet de développement ou même un script, cela peut vite devenir compliqué sur la durée... D'autant plus si l'on travaille à plusieurs sur ce projet. Par exemple, cela oblige à créer soi-même des copies de ses fichiers afin de pouvoir revenir en arrière si cela ne se passe pas bien.

Git apporte une réponse à cette problématique, car il va assurer un suivi sur tous les fichiers de votre projet afin de permettre une gestion des versions. Git est ce que l'on appelle un système de contrôle de version décentralisé (DVCS - Distributed Version Control System).

Git s'utilise aussi bien sur Linux, Windows que macOS et pour le stockage des données des différents projets, il y a plusieurs manières de voir les choses. Tout d'abord, on peut utiliser Git en local, c'est-à-dire que l'on installe l'outil sur sa machine et on stocke le code en local pour gérer les versions de son projet. Ensuite, on peut utiliser Git au travers des services en ligne comme **GitHub** ou **GitLab**.

GitHub est idéal pour les projets publics tandis que pour des projets privés, privilégiez plutôt **GitLab** et **Bitbucket**. Il est également possible de créer son propre serveur **Git** sur une machine **Linux**, ou sur un **NAS** comme sur **Synology** où il y a un paquet **"Git Server"**. Pour les utilisateurs du **Cloud Azure de Microsoft**, sachez qu'**Azure DevOps** intègre une fonctionnalité nommée **"Azure Repos"** qui sert à gérer des dépôts Git privés.

Git possède **deux structures de données** (En informatique, une structure de données est une manière d'organiser les données pour les traiter plus facilement) : **une base d'objets** et **un cache de répertoires**.

Il existe quatre types d'objets :

1. L'objet blob (**pour binary large object désignant un ensemble de données brutes**), qui représente le contenu d'un fichier ;

2. L'objet tree (**mot anglais signifiant arbre**), qui décrit une arborescence de fichiers. Il est constitué d'une liste d'objets de type blobs et des informations qui leur sont associées, tel que le nom du fichier et les permissions. Il peut contenir récursivement d'autres trees pour représenter les sous-répertoires ;

3. L'objet commit (**résultat de l'opération du même nom signifiant « valider une transaction »**), qui correspond à une arborescence de fichiers (**tree**) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc. Il pointe également vers un ou plusieurs objets commit parents pour former un graphe d'historiques ;

4. L'objet tag (**étiquette**) qui est une manière de nommer arbitrairement un commit spécifique pour l'identifier plus facilement. Il est en général utilisé pour marquer certains commits, par exemple par un numéro ou un nom de version (2.1 ou bien Lucid Lynx).

Vous pouvez alors utiliser l'outil **git bash**, c'est également un interpréteur de commande mais plus proche de celui de Linux car il utilise MinGW(Minimalist GNU for Windows) ou **PowerShell** ou **CMD**. A vous de choisir la plateforme que vous préférez.

II. Git : le vocabulaire à connaître !

Pour bien comprendre le fonctionnement de **Git**, d'être capable de bien interpréter la documentation, d'orienter ses recherches sur le Web, et être capable d'utiliser les bons termes, vous devez connaître absolument ces termes :

Repository

En français, un repository est un dépôt et il correspond à un espace de stockage sur lequel vous allez venir stocker les fichiers de votre projet. Autrement dit, c'est un dossier connecté à **Git**, car **Git** va surveiller ce dossier notamment pour identifier les changements, l'historique, etc. Un dépôt peut être local ou distant (**GitHub, GitLab, etc**.).

Master / Main

Lorsque l'on travaille sur un projet, il y a toujours (en principe) une branche principale nommée **"Master"** c'est-à-dire **"maître"** (cette branche peut aussi s'appeler **main**) : il s'agit de la branche principale du projet qui correspond toujours à la dernière version d'une application (ou d'un script, etc.). Néanmoins, il est possible d'utiliser un autre nom !

Branch

L'une des forces de **Git**, c'est de permettre la création et la gestion de plusieurs branches qui vont venir s'ajouter à la branche **"Master"** évoquée précédemment. Ainsi, on peut créer une nouvelle branche correspondante à un nouvel axe de développement pour travailler sur une nouvelle fonctionnalité, la correction d'un bug, etc... Cela va permettre de bénéficier de plusieurs versions des fichiers.

Pour manipuler le dépôt et les branches avec **Git**, on a différentes actions à notre disposition, dont :

Merge

Merge, ou fusionner est l'action que l'on utilise pour fusionner une branche annexe (par exemple, une branche utilisée pour développer une nouvelle fonctionnalité) avec la branche principale "Master" pour intégrer les modifications à la version principale via une action de fusion.

Commit

Quand on travaille sur un ou plusieurs fichiers associés à un dépôt Git, il faut synchroniser les modifications vers le dépôt afin qu'elles soient publiées en quelque sorte. L'action "commit" permet de valider les modifications avant de les envoyer au serveur où se situe le dépôt. C'est une action de contrôle.

Push

À la suite d'un commit, on effectue une action de type "push" pour envoyer les fichiers modifiés sur le serveur.

Pull

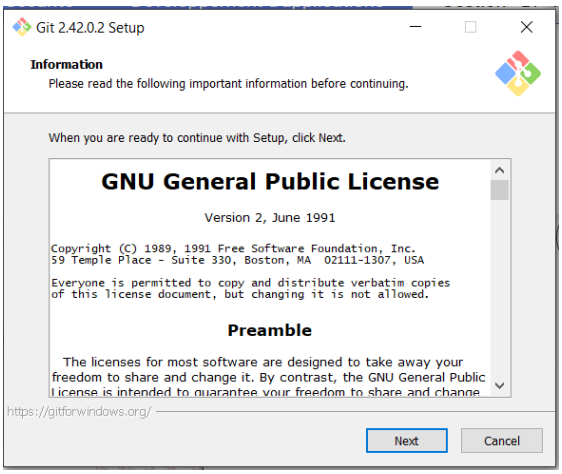
À l'inverse, on peut télécharger des fichiers à partir du dépôt distant sur notre machine en local pour travailler dessus. On parle d'une action "pull" qui se traduit par "tirer" en français, car on tire les fichiers du dépôt vers l'hôte local.

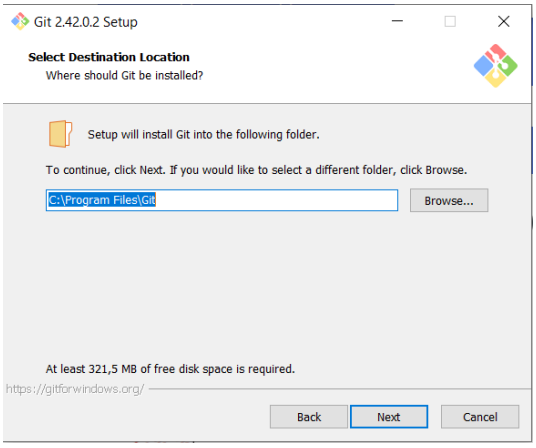
Clone

Quand on copie un projet situé sur un dépôt distant en local pour la première fois, on parle d'un clone du projet. Si l'on effectue des mises à jour par la suite, pour récupérer les derniers fichiers modifiés, on effectuera une action "pull". Cette action est aussi un moyen de télécharger une copie locale d'un projet complet pour lancer l'installation de l'application (par exemple un projet hébergé sur GitHub).

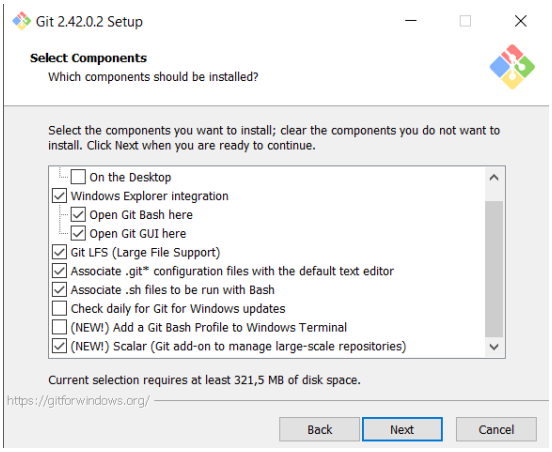
III. Installation de Git sous Windows

Git est officiellement pris en charge sur Windows et les binaires sont disponible sur le site officiel, à l'adresse suivante : **git-scm.com/download/win**.

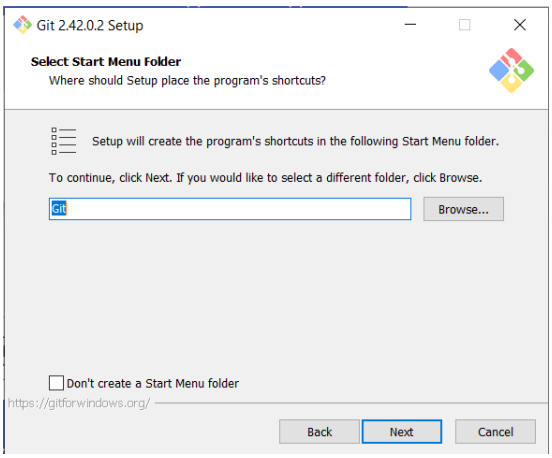




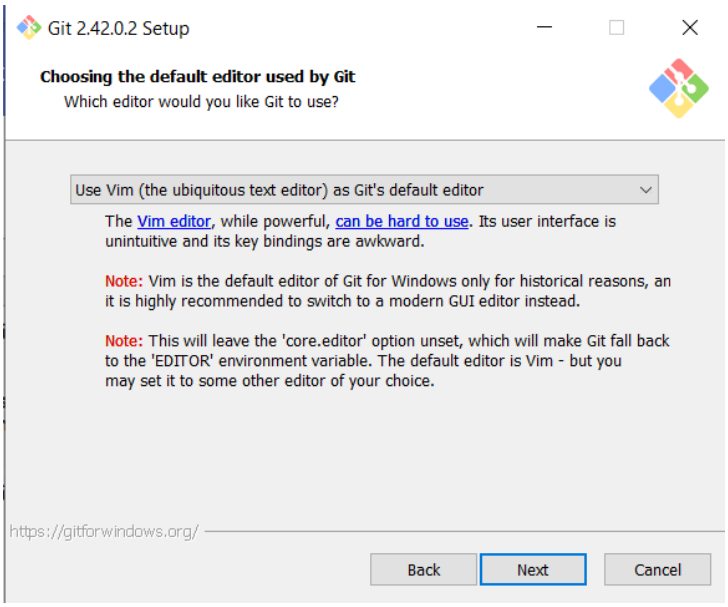
La troisième fenêtre permet de définir les éléments complémentaires à ajouter.



La quatrième permet de personnaliser l'affichage de Git dans le menu Démarrer de Windows.

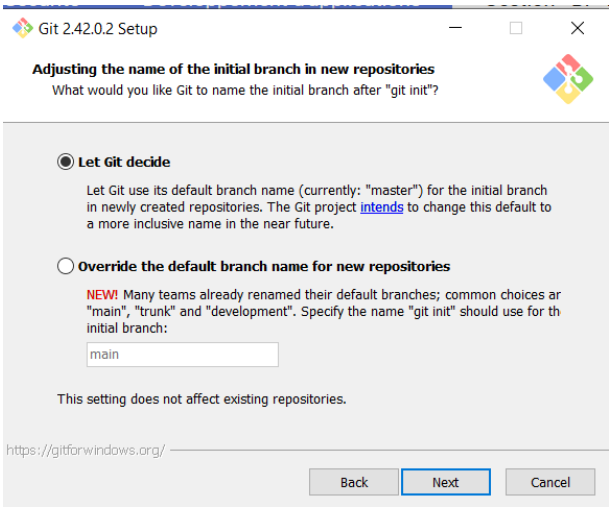


La cinquième fenêtre permet de définir l'éditeur de texte par défaut utilisé par Git (pour éditer le message de commit, par exemple). Vim est généralement difficile à manipuler par les néophytes mais est souvent présent sur les postes de travail.



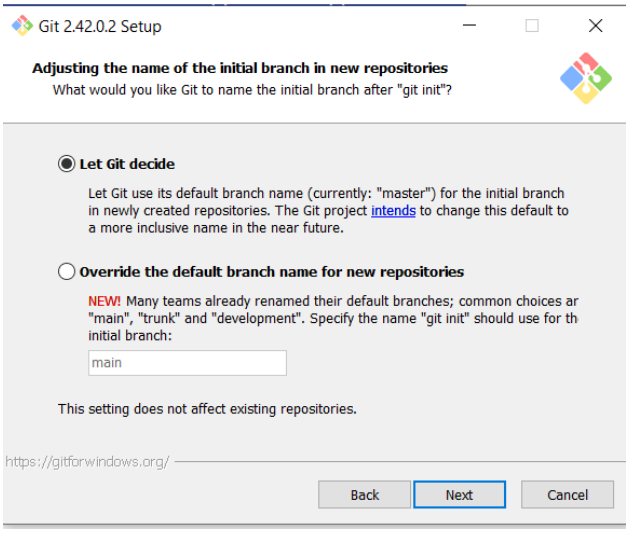
La sixième fenêtre permet de personnaliser le nom de la branche initiale d'un projet Git. Cette option de configuration pourra être personnalisée par la suite. Cette personnalisation est très récente. Les

équipes ne souhaitant pas nécessairement personnaliser le nom de la branche, cette option de configuration doit être prise conjointement avec les autres membres de l'équipe.



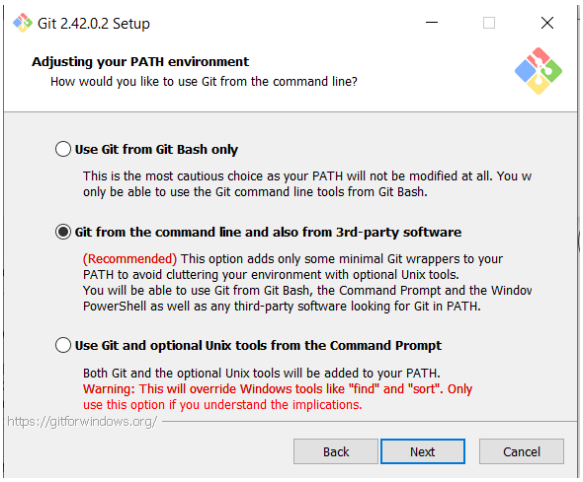
La septième fenêtre permet de personnaliser ce qui sera ajouté au PATH de Windows. Cela va modifier la façon d'utiliser Git dans Windows.

Git Bash est un outil qui permet de simuler un environnement bash UNIX dans Windows et utiliser les commandes qui lui sont propres (ls, mkdir, etc.) et d'utiliser le séparateur de chemin /.

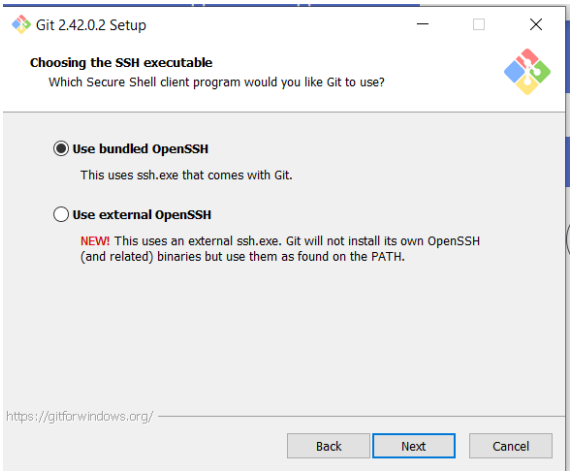


La septième fenêtre permet de personnaliser ce qui sera ajouté au PATH de Windows. Cela va modifier la façon d'utiliser Git dans Windows.

Git Bash est un outil qui permet de simuler un environnement bash UNIX dans Windows et utiliser les commandes qui lui sont propres (ls, mkdir, etc.) et d'utiliser le séparateur de chemin /.



La huitième fenêtre permet de personnaliser le protocole de transport HTTPS pour les communications de Git.



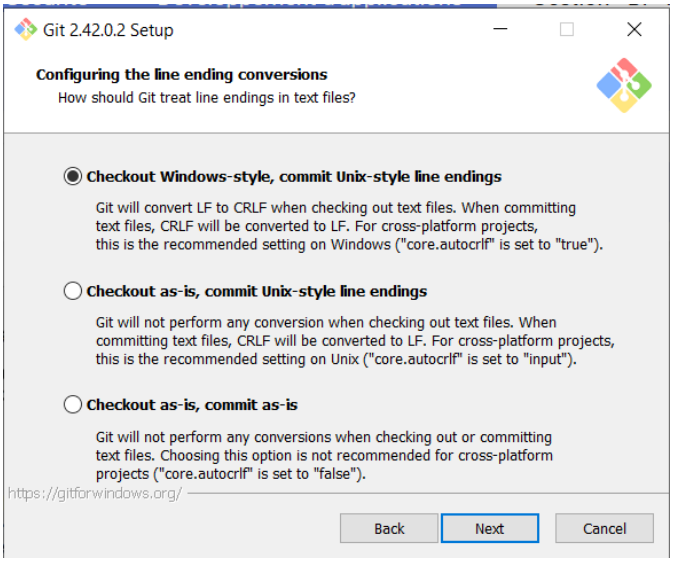
La neuvième fenêtre permet de configurer la gestion des sauts de ligne (capture ci-dessous). Il faut savoir que les systèmes UNIX et les systèmes Windows n'utilisent pas les mêmes caractères pour effectuer un saut de ligne. En effet, Windows utilise le mode CRLF (Carriage Return Line Feed), c'est-à-dire que dans un fichier de texte brut, Windows insère entre deux lignes un retour chariot puis un saut de ligne. Les systèmes UNIX utilisent le mode LF, c'est-à-dire qu'ils utilisent uniquement un saut de ligne entre deux lignes.

Git permet d'effectuer certaines actions sur les sauts de ligne. En effet, si un utilisateur travaillant sur UNIX crée un fichier et l'ajoute au dépôt, puis qu'un utilisateur Windows ouvre le fichier, le fichier sera perçu comme ayant été entièrement modifié étant donné que chaque saut de ligne aura été converti par Windows. Git va donc proposer plusieurs modes pour permettre aux différents systèmes d'être interopérables avec Git.

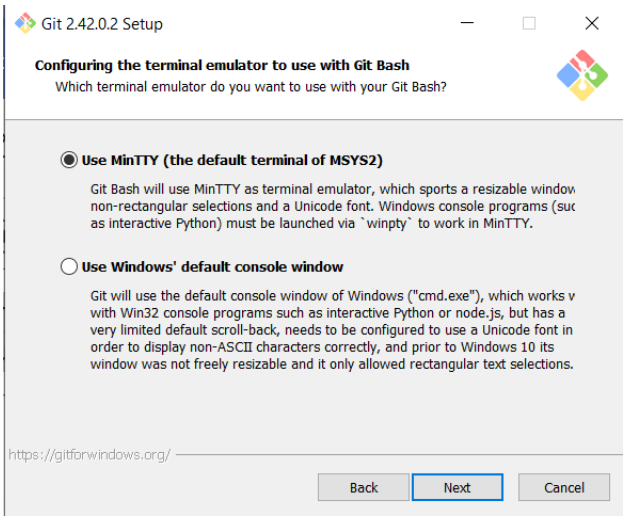
Le premier mode est le mode défini par défaut. Lorsque Git enregistrera des fichiers sur votre système Windows, il convertira les sauts de ligne UNIX (LF) en sauts de ligne Windows (CRLF). Git fera l'inverse lorsqu'il enregistrera le contenu des fichiers dans le dépôt. C'est le mode recommandé pour les utilisateurs de Windows.

Dans le deuxième mode, Git ne modifie pas les sauts de ligne des fichiers qu'il enregistre sur le système de fichiers. Inversement, lorsque l'utilisateur commitera des fichiers sur le dépôt, les sauts de ligne de type Windows seront convertis en sauts de ligne UNIX.

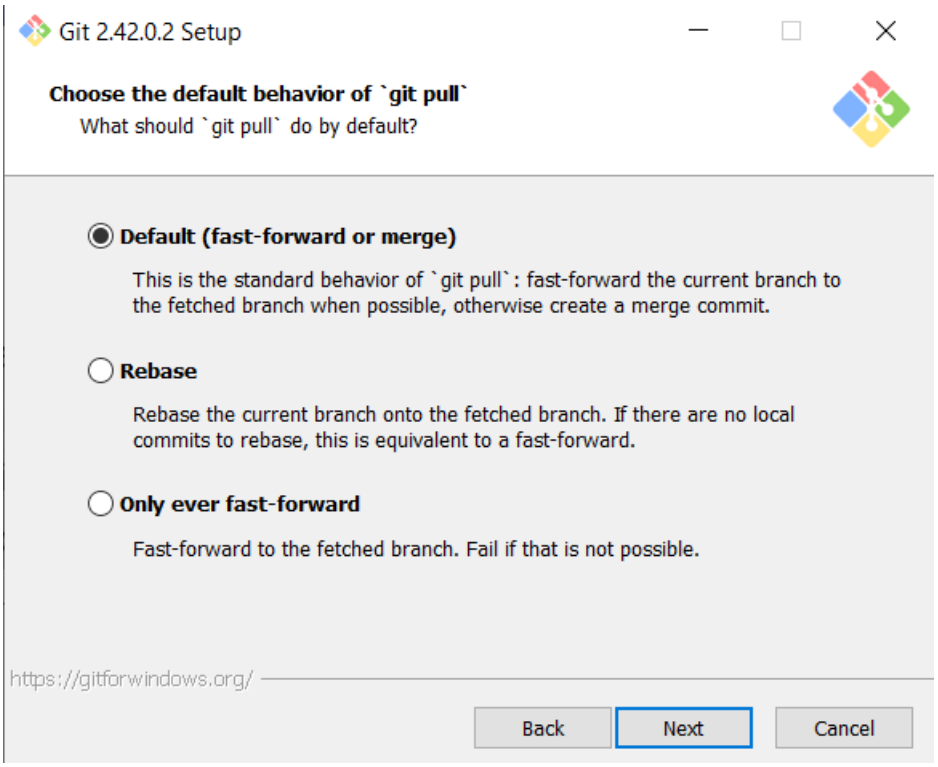
Le dernier mode n'effectue aucune conversion de saut de ligne. Normalement, si tous les utilisateurs de Git utilisent le même système d'exploitation, ce type ne pose pas de problème particulier. C'est tout de même se priver de nombreuses possibilités que de restreindre l'utilisation du dépôt à un seul type de système. C'est la raison pour laquelle cette option est fortement déconseillée lorsqu'on gère un projet développé sur des systèmes Windows et UNIX.



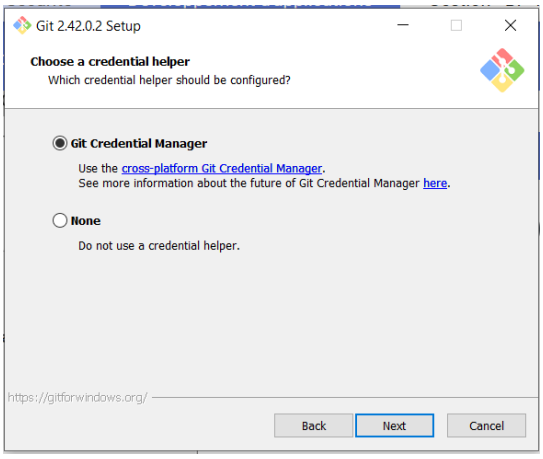
La dixième fenêtre permet de définir l'émulateur de terminal de Git Bash. MinTTY est une option généralement plus confortable au quotidien parce que cmd.exe reste limité.



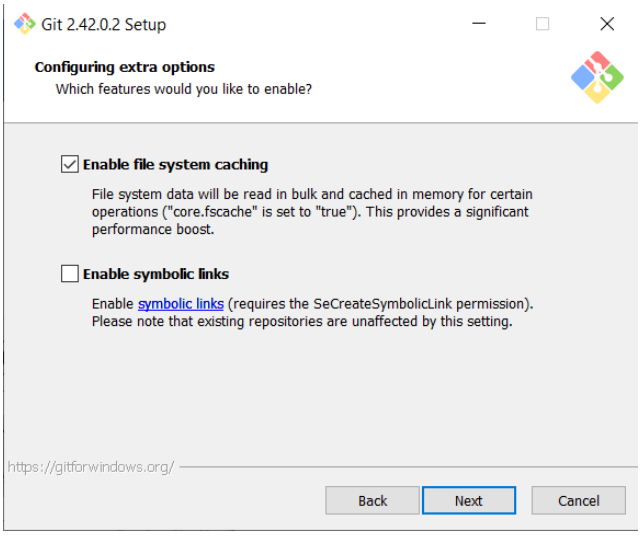
La onzième fenêtre permet de configurer le comportement par défaut de git pull. Pour en apprendre plus sur git pull et sur les fusions de branches, il faut lire les chapitres Les branches et les tags et Partager un dépôt. Cette option de configuration doit être prise de façon collégiale dans une équipe. Utiliser différentes stratégies de fusion au sein d'une équipe peut générer des problèmes et incompréhensions.



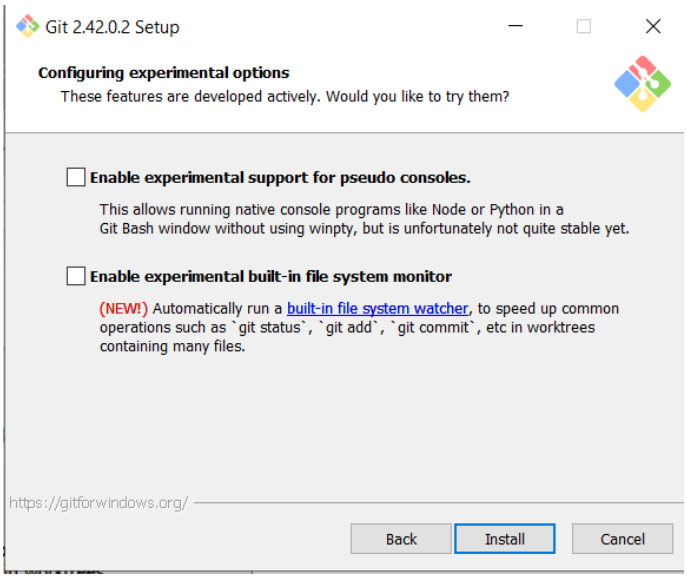
La douzième fenêtre permet de configurer la mise en cache des identifiants des dépôts. Pour en apprendre plus sur la mise en cache des identifiants, il est possible de lire la section Configurer le cache de l'authentification du chapitre Création d'un dépôt.



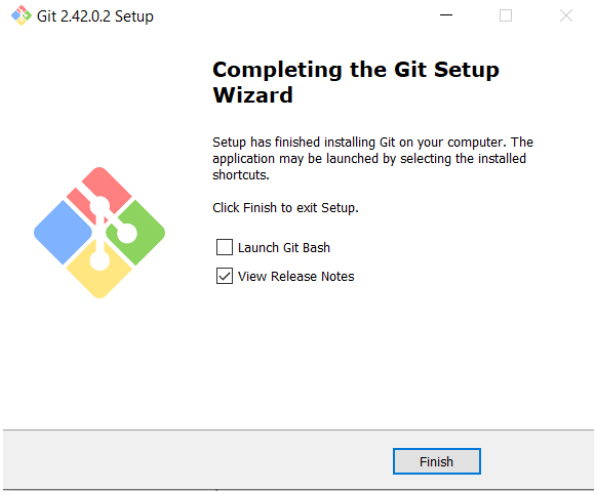
La treizième fenêtre permet d'activer ou non la mise en cache de certains fichiers et d'activer ou non l'utilisation de liens symboliques.



La quatorzième fenêtre permet d'activer un mode expérimental permettant de brancher des programmes sur Git Bash. Il faut éviter d'activer ce mode, excepté si le développeur prévoit de coder ce type de programme.



La dernière fenêtre permet de lancer Git Bash directement et finalise l'installation.



Une fois que Git a été installé, deux raccourcis ont été créés dans la liste des programmes. Un raccourci appelé Git Bash redirige vers une interface en ligne de commande gérée par Cygwin. Un autre appelé Git GUI redirige vers une interface permettant d'utiliser Git.

Pour toute la suite de ce livre, il faudra utiliser Git Bash qui permet d'effectuer beaucoup plus d'actions et qui permet également de mieux comprendre le fonctionnement de Git.

Après avoir cliqué sur le raccourci Git Bash, une interface en ligne de commande s'affiche. Cette interface permet d'utiliser Git ainsi qu'un nombre important de commandes bash.

Pour vérifier l'installation de Git, il est possible d'utiliser la commande suivante :

git --version

IV. Premiers pas avec Git

A. Configurer votre profil utilisateur

Pour commencer, avant même de créer un nouveau projet, nous allons définir deux options dans la configuration de **Git** : le **nom d'utilisateur** et l'**adresse e-mail.** Ainsi, les différentes actions, notamment les **"commit"** sur les fichiers seront associés à cet utilisateur, ce qui est important pour le suivi même si pour le moment vous travaillez seul sur ce projet.

git config --global user.name **"John Begood"**
git config --global user.email **"begood.nostress.gonabeok@gmail.com"**

```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t
$ mkdir workspace

DIDICOF@DESKTOP-0SDG95G MINGW64 /t
$ ls
 Bitlocker/  'System Volume Information'/   TPM/   workspace/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t
$ cd workspace/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace
$ mkdir cours_git

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace
$ cd cours_git/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git
$ git config --global user.name "John Begood"

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git
$ git config --global user.email begood.nostress.gonabeok@gmail.com

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git
$ git config --global --list
user.name=John Begood
user.email=begood.nostress.gonabeok@gmail.com
```

Une fois que c'est fait, vous pouvez afficher la configuration actuelle avec la commande ci-dessous pour valider que c'est bien pris en compte.
git config --global --list

B. Initialiser un nouveau projet Git

1. CREER UN DEPOT LOCAL

Désormais, nous devons indiquer à Git où se situe notre projet sur notre espace de stockage local (ici, nous travaillons sur un projet local). Il peut s'agit d'un répertoire existant qui contient déjà les fichiers de votre projet, ou d'un nouveau répertoire créé pour l'occasion. Pour tout nouveau projet que l'on souhaite **versionne**r, il est nécessaire de créer un nouveau dépôt. C'est ensuite dans ce dépôt que Git stockera toutes nos informations.

On va créer un dossier avec la commande mkdir **mon_site_web**

Il faut se placer dans ce dossier **c'est important de vous positionner à la racine de votre projet pour l'initialiser**. Exécuter la commande suivante : **git init**

```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git
$ mkdir mon_site_web

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git
$ cd mon_site_web/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web
$ git init
Initialized empty Git repository in T:/workspace/cours_git/mon_site_web/.git/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Git confirme la création du dépôt avec le message suivant : **Initialized empty Git repository in ….**

Cette action va créer un fichier(caché) nommé **.git** à la racine du projet.

Par défaut, la création d'un dépôt crée automatiquement **une branche master**.

2. LE CONTENU DU DOSSIER .GIT

Le fichier **.git** héberge tout le contenu du dépôt utilisé par Git. De manière pragmatique, nous allons visualiser les dossiers et fichiers présents dans ce fichier. Pour cela, il faut taper **ls .git**

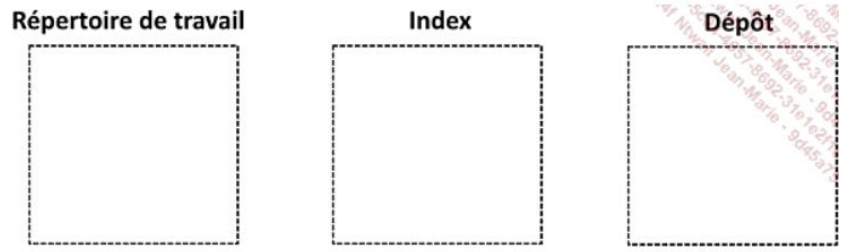
```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ ls .git/
COMMIT_EDITMSG  config       hooks/  info/  objects/
HEAD            description  index   logs/  refs/

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

C. Ajouter un fichier à notre projet Git

1. GESTION DES FICHIERS ET COMMIT

Un fichier peut se trouver à trois endroits différents : **le répertoire de travail, l'index et le dépôt.**
Le fichier va se trouver dans les différentes zones selon son avancée dans le projet.

Le schéma suivant montre un projet totalement vierge dans lequel un **dépôt Git** vient d'être initialisé, c'est-à-dire que nous n'avons enregistré aucun fichier dans le projet. La seule action effectuée sur ce projet est un **git init**.



Pour visualiser l'état des fichiers, nous allons utiliser la commande suivante : **git status**

On peut voir également qu'il n'y a pas encore eu de commit via la phrase **"*No commits yet*"**.

2. LE REPERTOIRE DE TRAVAIL

Cette zone correspond au répertoire du système de fichiers sur lequel travaille le développeur.

C'est le dossier du projet tel qu'il est stocké sur le disque dur. Les fichiers qui se trouvent dans cette zone peuvent être connus de Git selon qu'ils ont été ajoutés au moins une fois dans Git ou non. Un fichier qui se trouve uniquement dans cette zone est un fichier totalement inconnu pour Git.

Ce genre de fichier est également appelé fichier non suivi **(untracked file en anglais)**.

Pour arriver à ce résultat, vous pouvez utiliser les commandes : echo "<html>Le fichier est dans le répertoire de travail</html>" **hello.html** ou **style.css** comme exemple.

La commande **git status** nous indique que le fichier est visualisé en tant que fichier non suivi.

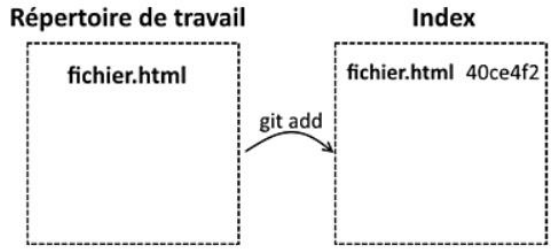
```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.html
        style.css

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

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Le fichier **hello.html** n'est pas suivi par **Git**. Il n'est pas encore identifié par **Git** et ne possède donc pas encore d'identifiant sous forme de **hash** (Le hash ou condensat ou signature est une empreinte numérique unique d'un fichier sous la forme d'une série alphanumérique). Un exemple de **hash** : **5819778898df55e3a762f0c5728b457970d72cae.**

3. L'INDEX

Cette zone utilisée par **Git** est très particulière. C'est en quelque sorte une zone d'attente de commit indépendante du répertoire de travail. Nous allons y placer les différents fichiers que nous voulons intégrer au prochain commit.

Un fichier qui a été ajouté dans l'index ne sera plus considéré comme un fichier non suivi.



Pour arriver à ce résultat, nous pouvons utiliser la commande suivante :

git add fichier.html

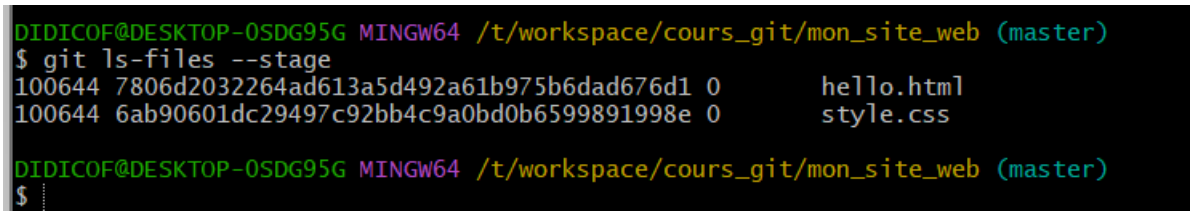
git add permet d'ajouter un fichier à l'index. Aussi, nous pouvons désindexer notre fichier avec la commande **git reset fichier.html**

Si ces plusieurs projets qui sont à la racine, nous pouvons utiliser la commande **git add .** **point qui veut dire toutes les modifications qui se trouvent dans workspace.**

La commande **git status** nous indique que le fichier est prêt à être **commité.**

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git add hello.html style.css
warning: in the working copy of 'hello.html', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'style.css', LF will be replaced by CRLF the next time Git touches it

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   hello.html
        new file:   style.css


DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Cette indication de Git selon laquelle les fichiers sont prêts à être commités signifie qu'ils se situent dans l'index, qui est la zone d'attente avant le commit. Voici la commande qui va nous permettre d'afficher ces éléments : **git ls-files --stage**



D. Effectuer l'action de commit

4. LE DEPOT

Pour que le fichier passe de l'index au dépôt, il faut **le commiter**. Commiter revient à enregistrer un ensemble de modifications. Le commit est le concept fondamental, de Git.

Une fois qu'un fichier a été commité, il passe alors dans la zone du dépôt.

Dans cette zone, le fichier est **versionné par Git** qui en contient au moins une version.

Le fichier est suivi **(tracked)** par **Git** et sera considéré comme modifié si le **hash** du fichier dans le répertoire de travail est différent du **hash** enregistré dans le dépôt.

La zone de dépôt contient tous les commits du dépôt. Cela signifie que tout l'historique du projet se situe dans le dépôt.

Pour enregistrer le fichier **hello.html et style.css** dans le dépôt, il faut utiliser la commande **git commit** :

git commit -m "exemple chemin hello.html ou style.css"

Cette commande va créer un paquet virtuel (également appelé c**ommit [snapshot]**) contenant un ensemble de modifications avec le titre défini par **l'argument -m**.

En fait, en déclenchant **un commit on va effectuer en quelque sorte un snapshot de notre code à instant *t***. Ainsi, si l'on effectue des modifications mais que finalement on souhaite revenir en arrière, il sera possible de **restaurer le fichier dans l'état qu'il était au moment du commit** (snapshot).
Avec l'option "**-m**", on peut **ajouter un commentaire qui sera associé à ce commit**. C'est très important pour apporter des précisions et être capable de s'y retrouver dans tous les commits.

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git commit -m"mon premier commit"
[master (root-commit) f8f323c] mon premier commit
 2 files changed, 102 insertions(+)
 create mode 100644 hello.html
 create mode 100644 style.css

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Lorsqu'on exécute cette commande, **Git** récupère le fichier placé dans l'index et l'ajoute dans le dépôt. Ensuite, **Git** supprime le fichier de l'index, car étant donné que celui-ci est une zone d'attente, le fichier n'a plus de raison de s'y trouver.

Il est intéressant de vérifier ce qu'affiche la commande **git status** :

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git status
On branch master
nothing to commit, working tree clean

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Git nous indique cela car aucun fichier n'est présent dans la zone d'index et la version du projet contenue dans le répertoire de travail correspond à la dernière version du dépôt.

> **En réalité, pour vérifier s'il y a des modifications, Git compare le répertoire de travail avec HEAD. HEAD est une référence (c'est-à-dire un raccourci de Git) qui pointe vers le commit le plus récent de la branche courante. Expliqué plus simplement, HEAD est la version courante enregistrée dans le dépôt.**

Pour afficher les détails du commit (ainsi que son identifiant), il est possible d'utiliser la commande suivante : **git log** ou **git log -1**

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git log -1
commit eaa02f1ae6c015d52998cbd65bf0f94af963eca3 (HEAD -> master)
Author: John Begood <begood.nostress.gonabeok@gmail.com>
Date:   Wed Oct 11 22:15:08 2023 +0200

    ajout du top banner

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

La commande ci-dessous sert à visualiser les logs de façon plus synthétique, pour une branche spécifique, ici "master".

git log --oneline master

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git log --oneline master
8348f48 (HEAD -> master) mon excel
c24352a ajout top-banner
eaa02f1 ajout du top banner
f8f323c mon premier commit
```

De la même manière que lorsque le fichier se trouvait dans l'index, il est possible de l'afficher maintenant qu'il est dans le dépôt avec la commande suivante : **git ls-tree -r HEAD**

```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git ls-tree -r HEAD
100644 blob 7806d2032264ad613a5d492a61b975b6dad676d1    hello.html
100644 blob 6ab90601dc29497c92bb4c9a0bd0b6599891998e    style.css

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Un élément supplémentaire est affiché dans cette sortie : le mot **blob**. Celui-ci signifie **binary large object**, c'est un type d'objet interne à Git qui est chargé de stocker l'intégralité du contenu d'un fichier.

Le schéma ci-dessous récapitule l'état des trois zones ainsi que le détail des éléments stockés dans le dépôt.

Le schéma ci-dessous montre ce qui se passe si le développeur modifie le fichier, l'ajoute à l'index mais ne le commite pas. Le fichier se retrouve donc dans un état distinct entre l'index et le dépôt, ce qui est à l'origine des deux identifiants différents.

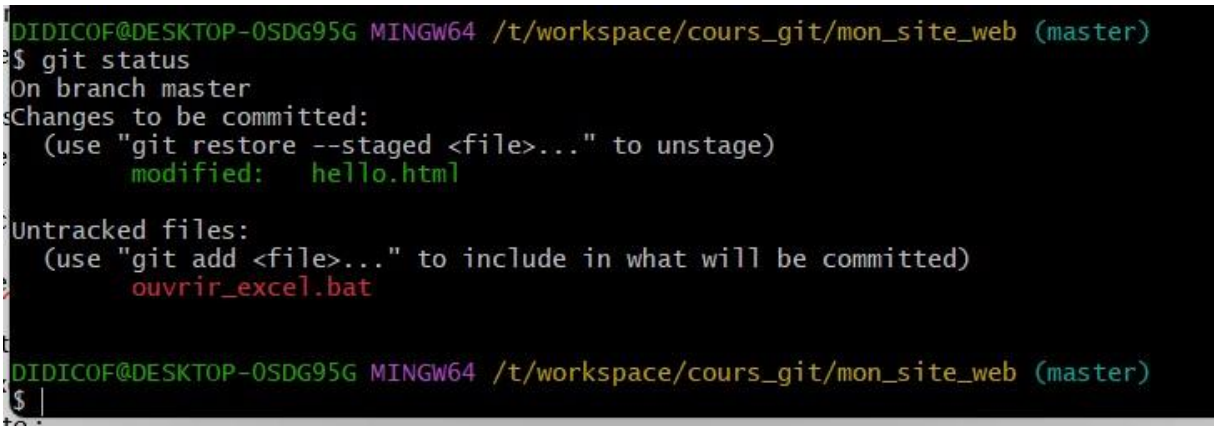
E. MANIPULER LES FICHIERS

1. AJOUTER DES FICHIERS DANS L'INDEX

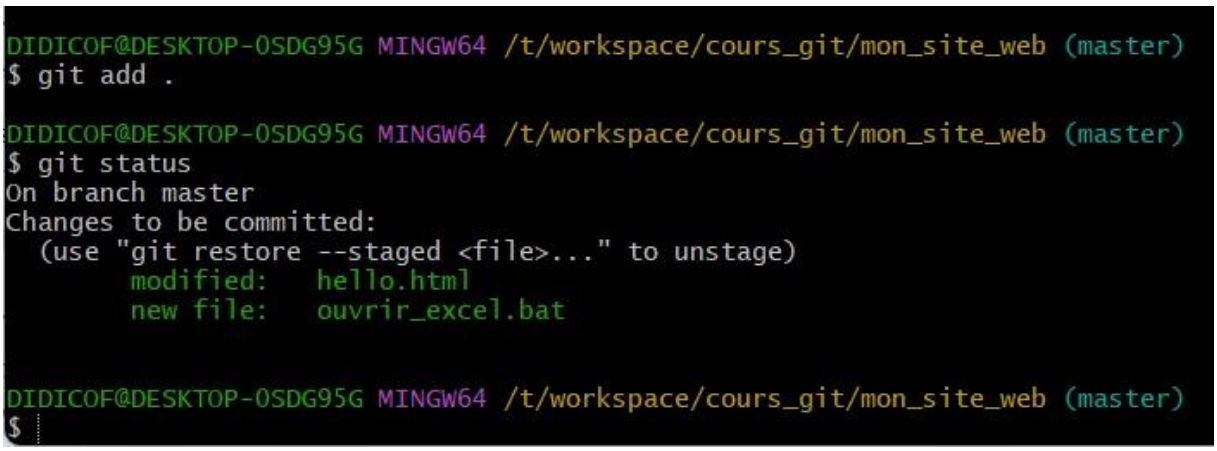
Il existe **deux** cas où il est utile d'ajouter un fichier à l'index.

Le premier cas est celui où le fichier est nouveau et inconnu de Git. Ajouter ce nouveau fichier dans l'index va permettre de prévenir Git que ce fichier doit être pris en compte. Comme expliqué précédemment, ce fichier passera d'un état non suivi à un état suivi par Git.

Ex : ici je viens d'ajouter un fichier.bat et il est dans l'état non suivi.



Maintenant je le passe à l'état suivi par Git avec la commande **git add .**



Le deuxième cas est celui où le fichier est déjà **versionné**, mais que le développeur souhaite mettre les modifications effectuées dans ce fichier dans l'index pour les commiter.

Pour ajouter un fichier dans l'index, il faut utiliser la commande **git add** avec la syntaxe suivante :

git add nom fichier

Il existe également des manières d'ajouter plusieurs fichiers dans l'index à l'aide d'une seule commande. Par exemple, pour ajouter tous les fichiers dans le dépôt, vous devez utiliser la commande suivante : **git add -A**

2. DEPLACER OU RENOMMER DES FICHIERS

Git possède une commande qui permet **de renommer ou de déplacer un fichier**. Voici la syntaxe de cette commande : **git mv ancien_fichier nouveau_fichier**

Voici par exemple comment renommer le fichier **ouvrir_excel.bat en open_excel.bat** :

```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git mv ouvrir_excel.bat open_excel.bat

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ ls
hello.html  open_excel.bat  style.css

DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

La commande **git status** nous affiche la sortie suivante :

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        Renamed :    ouvrir_excel.bat -> open_excel.bat
```

```
DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    ouvrir_excel.bat -> open_excel.bat


DIDICOF@DESKTOP-0SDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$
```

Cette sortie indique que le fichier a été ajouté à l'index et qu'il est détecté comme étant renommé.

La commande ls -l open_excel.bat qui permet de visualiser la liste de fichiers et de dossiers du répertoire courant affiche la sortie suivante :

```
$ ls -l open_excel.bat
-rw-r--r-- 1 DIDICOF 197121 28 Sep 24 21:19 open_excel.bat
```

Cette sortie indique que Git a bien renommé le fichier dans le répertoire de travail en plus de changer l'état du fichier dans l'index.

3. SUPPRIMER DES FICHIERS

Pour supprimer un fichier suivi par Git, il ne faut pas uniquement le supprimer dans le système de fichiers. Git nous propose une commande qui permet de supprimer le fichier dans le répertoire de travail, mais également de placer cette suppression dans la zone d'index pour qu'au prochain commit Git sache que le fichier ne doit plus être suivi. Il faut utiliser la syntaxe suivante :

git rm nom_fichier

Par exemple, dans le cas où un fichier nommé fichier.html existe dans le répertoire de travail :

git rm fichier.html

La commande git status indique que le fichier a été supprimé et que cette suppression est prête à être commitée.

Pour obtenir le même comportement, il est également possible d'utiliser les deux commandes suivantes :

rm fichier.html

git add fichier.html

La première commande supprime le fichier dans le répertoire de travail et la deuxième commande permet d'ajouter cette suppression dans l'index.

4. ARRETER DE SUIVRE UN FICHIER

Plusieurs raisons peuvent pousser un développeur à vouloir arrêter de suivre un fichier sans le supprimer. Un exemple typique de ce genre de cas est lorsqu'un développeur ajoute par erreur un fichier de log (ou n'importe quel autre type de fichier qu'on ne versionne pas) dans le dépôt. Le développeur a envoyé son commit au serveur central et à partir de ce moment-là, il ne peut pas revenir en arrière, il doit donc arrêter de suivre le fichier ajouté par erreur sans pour autant le supprimer.

Pour cela, il faut utiliser la syntaxe suivante :

git rm --cached nom_fichier

Par exemple, dans le cas où un fichier nommé fichier.html est présent dans le dépôt et dans le répertoire de travail :

git rm --cached fichier.html

5. IGNORER DES FICHIERS

Dans le chapitre Création d'un dépôt qui explique comment créer et configurer un dépôt, la section Ignorer des fichiers présente de quelle manière configurer le fichier .gitignore permettant d'ignorer un certain nombre de fichiers.

En effet, il existe plusieurs types de fichiers qui ne doivent pas être commités. Voici une liste non exhaustive des fichiers qui ne doivent pas être versionnés :

- Les fichiers de logs.
- Les fichiers résultants d'une compilation.
- Les fichiers des bibliothèques (ce cas particulier est abordé dans le chapitre Les outils de Git, à la section Dépôts intégrés avec submodules).
- Les fichiers de configuration (surtout lorsqu'ils contiennent des données sensibles).

Ces fichiers seront spécifiés dans le **fichier .gitignore** qui sera stocké à la racine du dépôt. Les règles de syntaxe du **fichier .gitignore** sont les suivantes :

- Les lignes vides sont ignorées. Elles peuvent donc servir de séparateur pour aérer le fichier.
- Les lignes débutant par un dièse # sont considérées comme des commentaires et n'ont aucune incidence sur les fichiers ignorés.
- Une ligne contenant un chemin complet suivi du nom d'un fichier exclut uniquement le fichier ciblé.
- Il est possible d'utiliser un ou plusieurs astérisques pour spécifier plusieurs noms de fichier à l'instar de ce qui est possible avec une interface en ligne de commande.
- Une ligne composée uniquement d'un nom de fichier ou de dossier exclut les fichiers portant ce nom quel que soit le dossier dans lequel ils se trouvent. Pour spécifier un fichier se situant à la racine du dépôt, il faut le préfixer par un slash comme si le chemin absolu du fichier débutait au niveau du dépôt.

- Un point d'exclamation en début de ligne signifie que les fichiers ciblés ne seront pas ignorés.

Ci-dessous un exemple de **fichier .gitignore** simple :

```
# Ignorer tous les dossiers lib (qui contiennent des bibliothèques)
lib/
# Autoriser le versionnement des bibliothèques internes
!lib/interne/
# Ignorer tous les fichiers Python semi-compilés
*.pyc
__pycache__
# Ignorer tous les exports de documentation, mais pas les originaux
/documentation/*
!/documentation/*.md
```

Le fichier .gitignore doit être enregistré à la racine du dépôt pour être interprété par **Git**. Ce fichier sera versionné, car il correspond à toutes les exclusions qui doivent être faites pour le projet.

Il existe un autre type de fichier servant à ignorer des fichiers dont la configuration a été évoquée dans le chapitre Création d'un dépôt à la section Ignorer des fichiers. Ce fichier est global et sert à ignorer les fichiers propres à l'utilisateur. Par exemple, ceux qui utilisent des **IDE tels que PyCharm** ont l'habitude de voir apparaître à la racine de leur projet **un dossier .idea** contant les fichiers utilisés par **PyCharm** pour gérer le projet. Ce n'est pas **au fichier .gitignore** d'ignorer ce type de fichiers, car ces fichiers ne dépendent pas du tout du projet, mais de l'utilisateur. En effet, rien n'empêche un développeur d'utiliser un autre éditeur comme **Sublime Text** ou encore **Eclipse**. Pour cela, il faudra utiliser le **fichier .gitignore_global** défini avec l'option de configuration **core.excludesfile**. Ce fichier possède exactement la même syntaxe que le **fichier .gitignore**.

F. Basculer d'une branche à une autre

Pour le moment, nous travaillons sur une branche unique nommée "master". Imaginons que l'on souhaite travailler sur de nouvelles fonctionnalités à inclure à notre script, et que l'on ait besoin de rentrer en phase de développement, tout en gardant de côté la branche principale. Dans ce cas, on peut **créer une nouvelle branche nommée "dev" (ou avec un autre nom) qui sera utilisée pour développer les nouvelles fonctionnalités.**
Pour créer une nouvelle branche **"dev",** on peut utiliser cette commande : **git branch dev**

Ou, directement la commande **git switch -c dev** qui va créer la branche et basculer dessus, tandis que la commande **git branch dev** crée seulement la branche.
Pour basculer d'une branche à une autre, c'est simple il suffit d'utiliser **"git switch"** et de spécifier le nom de la branche.

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git switch -c dev
Switched to a new branch 'dev'

DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (dev)
```

git switch <nom de la branche>
git switch master : La branche actuelle est mise en évidence, comme c'est le cas ici avec la branche "***master***".

La commande "***git switch***" est une commande qui remplace "***git checkout***" même si cette dernière fonctionne toujours. Si vous souhaitez obtenir la liste de vos branches, utilisez cette commande : **git branch**

```
DIDICOF@DESKTOP-OSDG95G MINGW64 /t/workspace/cours_git/mon_site_web (master)
$ git branch
  dev
* master
```

Le sujet des branches est vaste et mérite un article dédié, mais sachez qu'il est possible de supprimer une branche (***git branch -d <nom de la branche>***), de fusionner des branches (***git merge <nom de la branche>***), etc.

G. Restaurer un fichier

Dans le cas exemple où l'on travaille sur notre fichier **"hello.html ou style.css ou open_excel.bat"** mais que l'on a cassé une partie du code et que l'on souhaite revenir en arrière, à l'état tel qu'était le fichier lors du dernier commit, il suffit d'utiliser la commande "***git restore***" de cette façon :
git restore hello.html ou style.css ou open_excel.bat
Cette commande va restaurer le fichier immédiatement, sans même que l'on ait besoin de confirmer. A utiliser avec prudence, donc.