SAÉ 1.05

Recueil de besoins - GIT

Partie 1 - Présentation de GIT

Introduction

GIT[1] est un Outil de Gestion de Versions qu'il faut absolument connaître et maîtriser.

Un tel outil est très orienté développement, et c'est un outil qui sera obligatoirement un de vos fidèles compagnons durant votre vie professionnelle.

Nous allons aussi essayer d'en faire un compagnon de votre vie d'étudiant car il peut vous aider dans de nombreuses situations dans différentes matières et nous allons voir comment.

Dans ce qui suit, nous allons souvent parler de projets et de groupes d'individus, de développeurs (mais pas uniquement) travaillant sur ces projets.

C'est une façon assez commune de présenter GIT : un outil utilisable dans le cadre de travaux de groupes.

Cependant, il ne faut pas sous-estimer son intérêt en mode solo, juste pour soi. Bien au contraire, une utilisation en solo n'est pas du tout un détournement de la fonction première de GIT.

Utilisez-le dès que vous en avez l'occasion, c'est une excellente habitude !

Outil de Gestion de Versions

Un Outil de Gestion de Versions ou **VCS** en anglais, qui signifie **V**ersion **C**ontrol **S**ystem, est un logiciel qui permet d'assurer plusieurs fonctions, telles que le *Stockage*, la gestion d'*Historique* et le contrôle d'*Intégrité* des données.

[1] **GIT** se prononce comme "Guitare"

Stockage

GIT assure le stockage de fichiers sous une forme qui est généralement le reflet d'une partie de la structure hiérarchique d'un système de fichiers, tout simplement, une arborescence de dossiers et de fichiers, et généralement ces dossiers et fichiers sont regroupés dans ce qu'on appelle un projet.

Mais là où un Outil de Gestion de Versions se différencie d'un système de fichiers traditionnel est dans sa capacité à conserver un historique des évolutions des fichiers, à la fois dans leur contenu mais aussi dans leur organisation dans ce système.

GIT est capable de prendre des sortes de clichés, à un instant T, de l'état des fichiers. Ces instants T sont choisis par le ou les utilisateurs participant au projet. Il s'agit généralement de moments clés dans l'évolution des fichiers au cours de la vie du projet. La succession de ces clichés peut être vue comme une série de photos qui seraient prises durant la vie d'une personne, à des moments importants où on choisit de figer les choses afin de garder la mémoire d'un état et pouvoir y revenir quand on le souhaite.

Historique et navigation

En plus d'assurer le stockage, un Outil de Gestion de Versions permet de naviguer dans l'historique des différentes versions, ces fameux clichés pris au fil du temps.

Revenir en arrière peut être intéressant voire nécessaire pour retrouver une situation où tout fonctionnait bien, quand quelque chose ne tourne plus correctement par exemple.

On peut aussi avoir besoin de corriger un dysfonctionnement dans un logiciel alors qu'on est en plein milieu d'une modification du logiciel qui peut prendre encore plusieurs jours voire plusieurs semaines avant d'être terminée. Si le dysfonctionnement ne peut pas attendre ce délai, il doit être corrigé sur le champ. Un Outil de Gestion de Versions permet de faire facilement ce genre de choses sans risquer de perdre son travail en cours.

Intégrité des données

Un Outil de Gestion de Versions a aussi un rôle capital : celui d'assurer l'intégrité des données, des fichiers d'un projet.

Avec lui il est quasiment impossible d'égarer des modifications, notamment lors de travaux en groupe. Le système d'historique de versions assure que les clichés sont inaltérables et peuvent être restaurés à tout moment.

Pourquoi GIT

Il existe de nombreux Outils de Gestion de Versions. GIT est l'un d'entre eux et connaît un réel succès car il est libre, c'est à dire que son code source est disponible et qu'on peut en faire usage gratuitement. De fait, il est aussi très utilisé dans le monde du logiciel libre dont Linux fait partie.

Pour la petite histoire, GIT a été créé par Linus Torvalds, qui est évidemment connu pour être aussi le papa de Linux. C'est d'ailleurs pour gérer le développement de Linux que GIT a été créé.

Principes et concepts clés

Avant de plonger dans l'utilisation concrète de GIT, il est nécessaire d'introduire quelques notions essentielles de l'environnement GIT.

Fichiers texte

GIT est un outil du monde du développement : il sert à stocker et gérer le code source de logiciels.

L'usage qu'on vous propose d'en faire et évidemment celui-là aussi, mais pas seulement.

Les codes sources sont en principe des fichiers texte. Par exemple du *C*, du *Java*, du *HTML*, du *CSS*, du *Javascript*, du *PHP*, du *SQL*, du script *Bash* etc.

En fait, GIT excelle dans l'organisation et le suivi des versions de **fichiers textes**. Partant de là, il est tout à fait adapté pour gérer d'autres choses que des codes sources informatiques, dès lors qu'il s'agit de fichiers texte. Par exemple, si vous prenez des notes avec un *Notepad*, un *Textedit*, un *Geany* voire *nano* ou un bon vieux *vim* pour les plus geek d'entre vous, vous pouvez aisément les placer dans un GIT. Ça peut aussi servir pour gérer des tableaux de données en CSV ou des fichiers XML. Tout ce qui est fichier texte est un bon candidat pour GIT !

On y revient un peu plus loin, concernant les bons et mauvais usages de GIT.

Projet

Conçu initialement comme l'outil de Gestion de Versions pour le développement de Linux, GIT est généralement associé à la notion de projet informatique, et c'est donc naturellement dans ce cadre que nous allons l'étudier.

Ainsi, dans les explications à venir, on va parler fréquemment de projet.

Ça peut être, par exemple, un site *Web*, le code source d'une application en *C* ou en *Java*, ou même de simples fichiers de documentation en *markdown* ou en texte brut.

Il n'est pas nécessaire qu'un projet contiennent des dizaines ou des centaines de fichiers pour que l'usage de GIT ait un intérêt. Il fonctionnera tout aussi bien sur les 2 ou 3 fichiers sources que vous produisez pour un TP d'Algo par exemple, ou encore pour gérer des fichiers de requêtes SQL en séance de BDD, ou tout simplement pour des prises de notes, des aide-mémoire etc., à partir du moment où ce sont des fichiers textes.

Cependant, il est fortement conseillé qu'un projet tienne dans un unique dossier racine contenant ensuite éventuellement tous les fichiers et les sous-dossiers que vous voulez, sans aucune limite d'imbrication de sous-dossiers.

Historique

GIT est une sorte de machine à remonter le temps, ou plus précisément à se balader dans le temps.

Imaginez-vous en train de feuilleter un album-photo de famille contenant des clichés, des photos prises à des moments clés de votre vie qu'on aurait choisi d'immortaliser, il y a peut-être 5 minutes, hier ou encore il y a un an par exemple.

Imaginez maintenant que vous puissiez choisir l'une de ces photos pour revivre au temps où elle a été prise. Et bien GIT permet de réaliser exactement cela sur les fichiers d'un projet. Il est capable de revenir à un moment clé dans sa frise chronologique et de replacer le projet dans l'état dans lequel il était à un instant T que vous aurez, comme pour l'album de famille, choisi d'immortaliser.

Cette analogie avec le cliché photographique c'est ce qu'on appelle l'historique dans GIT.

Dossiers

Attention, GIT ne sait s'occuper **QUE DE FICHIERS** !

Un dossier n'existe pas vraiment dans GIT. Ce sont uniquement des fichiers qui sont stockés.

Vu de la racine du projet (donc du dépôt local GIT), un fichier possède un chemin relatif qui mène à lui. On sait, depuis **R1.04 Systèmes**, qu'un chemin est une succession de dossiers imbriqués et que la terminaison est, dans ce cas, le fichier en question.

Dans GIT, un dossier n'existe que virtuellement. Il existe parce qu'il apparaît dans un chemin menant vers (au moins) un fichier du projet.

Ainsi, il est totalement impossible de stocker un dossier vide dans un projet GIT. Ça ne signifie pas que DANS votre arborescence sur disque le dossier ne peut pas exister, ça signifie juste que vous ne pourrez pas dire à GIT de prendre en compte ce dossier s'il est

vide car vous ne pouvez lui demander de prendre en compte que des fichiers. Et sans au moins un fichier dans un dossier, on est évidemment coincé.

Si vous avez un peu de mal avec cette notion de dossier que GIT ne sait pas gérer, repensez à cette histoire de clichés : si un dossier est vide c'est qu'il n'y a pas de fichier dans le dossier. GIT ne prend pas de photo de famille sans personne dessus, c'est aussi simple que ça !

Pas d'inquiétude, il existe des solutions, on aura l'occasion d'en reparler plus loin.

Dépôt local

Nous venons de voir ce qu'est l'Historique dans GIT. Voyons maintenant où est placé l'album-photo d'un projet GIT.

Evidemment, dans un album-photo, on n'y colle pas les vraies personnes, on y place des copies *papier* de ces personnes, sous la forme de photos.

Dans GIT c'est pareil, on place des copies des fichiers originaux dans l'album-photo de GIT.

Cet *Album-GIT* porte un nom : on appelle cela un **Dépôt** ou **Repository** en anglais.

Un dépôt est situé dans un sous-dossier du dossier racine d'un projet. Ce sous-dossier s'appelle toujours **`.git`**, et, même s'il cohabite avec les fichiers de votre projet, il est conseillé de ne pas y toucher et de laisser GIT gérer son affaire tout seul.

Nous allons voir plus loin comment ce dépôt est créé.

> Attention, vous avez peut-être déjà utilisé GIT ou entendu parler de dépôts publics GIT tels que *Github* et *Gitlab*.
>
> On y revient plus loin, mais sachez que les dépôts dont on parle pour le moment sont des dépôts locaux à votre ordinateur et, à ce stade, sans aucun rapport avec les dépôts publics. Ne mélangez pas les deux.
>
> Si ça ne vous parle pas, ignorez ce paragraphe pour le moment.

États d'un fichier

Parlons maintenant des états dans lesquels peut se trouver un fichier du projet et de la façon dont on prépare la séance photo de nos fichiers.

Il y a 3 états possibles pour un fichier, et un fichier peut être dans un ou plusieurs de ces états en même temps. Attention, aucun de ces états n'impacte le contenu du fichier. L'état est ce qu'on appelle une méta-donnée, une information annexe sans rapport avec

le contenu du fichier lui-même. Par exemple, la date du fichier est aussi une méta-donnée.

C'est une règle d'or : GIT ne touche jamais au contenu des fichiers, il gère ces états de son côté, dans le dossier **`.git`**.

Modifié ou Nouvellement créé

Un fichier peut être dans un état *modifié* ou *nouvellement créé*.

Mais *modifié* par rapport à quoi ?

Et bien tout simplement, modifié par rapport au dernier cliché qu'on a de lui dans l'album de famille.

Il est dans l'état *nouvellement créé* s'il était absent de la dernière photo de famille, c'est à dire qu'il n'est pas encore pris en charge par GIT dans le projet.

On stage

Un fichier peut être dans le *stage*, ce qu'on traduit en français par l'*index* dans le jargon GIT.

En anglais, *Stage* signifie une phase, une étape, mais il peut aussi signifier la scène, et cette idée de scène va permettre de revenir sur l'analogie des clichés photo : on prend une photo de ce qu'on a placé sur la scène !

NB : c'est une interprétation personnelle. Le terme officiel français reste bien l'*index*.

Les fichiers sont donc *pris en photo* au moment où on les place dans l'index. Mais, pour le moment, la photo n'est pas encore dans l'album, c'est à dire que le fichier n'est pas encore dans le Dépôt local, il est juste dans l'index. Et même s'il est ensuite modifié localement, son état dans l'index est figé.

Par contre, si on décide de reprendre un cliché d'un fichier modifié qui serait toujours présent dans l'index, le nouveau cliché écrase le précédent. De la même manière, on peut aussi supprimer un fichier, de l'index, c'est-à-dire supprimer un cliché avant qu'il n'arrive définitivement dans l'album, comme si on n'avait jamais pris la photo. Cette action s'appelle : *unstage*, qu'on pourrait donc traduire par *désindexer*.

Insistons bien sur le fait que *désindexer* un fichier ne le supprime pas du dépôt, on le retire juste de l'*index*, du *stage*, le dernier cliché de modification qu'on a pris de lui.

Dans le dépôt local

Le troisième et dernier état possible pour un fichier est d'être dans le *dépôt local*, c'est-à-dire dans l'album de famille du projet.

Un fichier ne peut arriver dans le *dépôt local* que s'il a été placé dans l'*index* avant tout.

C'est lors de la mise dans l'*index* que l'état du fichier est figé. Le déplacement de l'*index* au *dépôt local* est simplement l'action de placer la photo dans l'album.

C'est le seul état qui bénéficie de l'historique car une photo n'écrase pas la précédente, elle s'ajoute à la frise chronologique. L'état du fichier figé est déplacé de l'*index* vers le *dépôt local*, il disparaît donc de l'*index*, qui n'est finalement qu'un état transitoire.

Le passage de l'*index* vers le *dépôt local* se fait de façon globale et atomique pour tous les fichiers qui sont en attente dans l'*index* au moment du passage. Cette action s'appelle le *commit*, qu'on pourrait traduire en français par le *transfert*, mais il n'y a pas réellement de traduction officielle, donc on utilise généralement le terme anglais de *commit* qu'on trouve aussi sous la forme du verbe *commiter*.

Vous pouvez et même vous devez *commiter* aussi souvent que possible. Évidemment on ne *commite* pas tout et n'importe quoi. L'Atelier sur le **`commit`** va vous détailler les règles à respecter.

Un dernier point concernant les états d'un fichier. Il n'y a pas d'obligation que les fichiers situés dans le dossier servant à votre projet soient tous pris en charge par GIT. Certains d'entre eux peuvent être non suivi, dans le jargon de GIT on dit **`untracked`**, tout simplement parce qu'il n'y a aucune raison d'en conserver une trace ou un historique. Par exemple, si votre application génère des fichiers de **`log`**, des traces d'exécution qui servent généralement au débuggage, ces fichiers n'ont qu'une utilité locale et ponctuelle. Il n'y a donc aucune raison de les ajouter au GIT de votre projet.

Local vs Distant

On a vu les trois états d'un fichier sous le contrôle de GIT. En fait, il en existe un quatrième mais il est optionnel.

Un *dépôt* GIT peut être autonome et isolé, placé localement dans le dossier d'un de vos projets, sur votre disque dur. C'est ce qu'on appelle un *dépôt local* et c'est d'ailleur, pour le moment, le seul type de dépôt qu'on a évoqué.

Mais un *dépôt local* peut aussi être lié à un *dépôt externalisé*, placé sur un serveur dédié.

Un tel *dépôt* s'appelle un *Dépôt Distant* ou *Remote Repository* en anglais.

Il existe des serveurs publics qui permettent d'accueillir des dépôts distants. Les deux plus connus sont Github (**`https://github.com`**) et Gitlab (**`https://gitlab.com`**).

Il est aussi possible de créer soi-même un serveur privé qui offre les mêmes prestations qu'un serveur public. A l'IUT nous disposons d'un tel serveur.

Les deux dépôts, le local et le distant, peuvent être synchronisés à tout moment dès lors que l'un des deux a reçu de nouveaux clichés du projet.

Le *dépôt distant* a plusieurs fonctions. Il sert déjà de copie de secours à votre *dépôt local*. Il sert aussi de plateforme de distribution quand plusieurs personnes travaillent en

parallèle sur un même projet. Chacune synchronise son propre *dépôt local* avec le *dépôt distant* qui est unique et commun à tous.

L'avantage de ce *dépôt distant* est qu'il est tout à fait possible de récupérer un projet, à tout moment, sur un ordinateur quelconque, à condition évidemment de disposer des droits pour accéder au *dépôt distant*, si nécessaire.

Pour utiliser GIT pour un projet sur son ordinateur, on peut avoir un *dépôt local* sans nécessiter d'avoir un *dépôt distant*, mais l'inverse n'est pas possible. Il faudra obligatoirement passer par la création locale d'une copie d'un *dépôt distant* avant de pouvoir faire quoi que ce soit, localement, avec GIT.

Si vous faites bon usage de cet outil, il vous sera très aisé de récupérer, à tout moment et où vous voulez, notamment chez vous, vos travaux réalisés à l'IUT. Ce qui est valable dans ce sens l'est évidemment aussi dans l'autre, et ce que vous aurez fait chez vous, vous pourrez aussi le retrouver à l'IUT.

Tout ceci, sans avoir à vous préoccuper de balader une clé USB à la fiabilité très relative, et qui demanderait, en plus, de faire des transferts de fichiers avec le risque d'en écraser certains par des versions plus anciennes, ce qui arrivera fatalement tôt ou tard, c'est garanti, et avec cette technique, pas d'historique, juste vos yeux pour pleurer…

Finies aussi les excuses de la clé USB oubliée chez soi ! 😈 😇

Branches

Avant de passer aux ateliers de mise en pratique, nous devons parler d'une dernière chose essentielle dans le cadre d'une utilisation de GIT sur des projets de groupe, c'est à dire quand plusieurs personnes sont amenées à modifier des choses sur les mêmes fichiers, en même temps, mais chacune dans son *dépôt local*.

Vous imaginez aisément que si plusieurs personnes modifient les mêmes fichiers, au moment où tout ceci va remonter vers le dépôt distant qui, rappelons-le, est unique et commun à tous, ça risque de se télescoper fortement. Comment peut-on faire pour réduire les risques ? Les branches apportent un début de solution à ce problème.

Une branche est une sorte de copie d'un projet à une étape précise de son historique. Notez que l'étape choisie n'a pas l'obligation d'être la plus récente, on peut créer une branche à partir de n'importe quelle étape, du présent ou du passé.

Cette copie s'appelle une branche car elle va servir de point de départ à une évolution du projet qui va se faire de manière parallèle et indépendante du projet initial.

L'intérêt des branches est de permettre justement à chaque membre d'une équipe de pouvoir travailler sur des fichiers communs sans perturber le travail des autres. A un moment, ces différentes évolutions des fichiers pourront être mises en commun pour

reconstituer une version synthétisant les apports des uns et des autres. Cette action s'appelle la *fusion*. C'est une étape importante qui peut demander du temps et du travail. Utiliser des branches permet ainsi de choisir quand on souhaite faire cette fusion au lieu de se retrouver confronté au problème à chaque synchronisation entre le dépôt distant et le dépôt local de chaque membre de l'équipe du projet.

Une branche permet aussi d'expérimenter des changements, qui pourraient casser ce qui est actuellement propre et fonctionnel dans le projet. En travaillant sur une branche, vous pourrez continuer sur cette branche l'esprit tranquille. Si votre travail ou vos expérimentations aboutissent à quelque chose d'acceptable, vous pourrez fusionner cette branche secondaire avec celle qui avait servi de point de départ, comme si vous aviez travaillé directement sans créer de branche, c'est-à-dire sans filet de sécurité. Mais si vous n'êtes pas satisfait de vos changements, le filet de sécurité que constitue cette branche sera très appréciable, il suffira simplement de scier la branche, de la supprimer et le projet retrouvera l'état originel qu'il avait avant la création de la branche, comme si rien ne s'était passé entre temps. C'est très confortable et rassurant.

S'il faut retenir une chose quand on travaille en équipe mais aussi en solo, c'est de créer des branches aussi souvent que nécessaire, ça vous évitera beaucoup de soucis.

Notez qu'un projet a toujours une branche principale qu'on appelle le **`master`**. C'est un peu comme pour un arbre, il a toujours une branche principale, qui s'appelle le tronc. De plus en plus souvent on trouve aussi le nom de **`main`** à la place de **`master`**.

Pour créer une branche, on a dit qu'il fallait partir d'un point de l'historique. Cette nouvelle branche est donc attachée à ce point de départ et va évoluer à partir de là. Il convient de préciser que, comme dans un arbre, une branche peut avoir son origine, son point de départ, sur le tronc, c'est à dire sur la branche **`master`**, mais elle peut tout aussi bien avoir son origine sur une autre branche, c'est même quelque chose de très fréquent sur les projets impliquant de nombreux membres actifs.

Partie 2 - Ateliers

Atelier 1 - Préparation d'un dépôt local

Pour pouvoir utiliser GIT, deux situations peuvent se présenter.

La première est que vous voulez créer un tout nouveau projet qui n'a pas encore de dépôt ni localement ni distant.

La seconde situation possible est que vous voulez créer un *dépôt local* à partir d'un *dépôt distant* déjà existant. C'est le cas notamment quand un projet est déjà initié et que vous venez vous greffer au projet ou que vous souhaitez utiliser une nouvelle machine que celle que vous utilisez d'habitude, ou simplement parce que vous avez changé d'ordinateur et que vous devez réinstaller votre environnement de développement.

Ces deux situations se gèrent différemment uniquement pour le démarrage, mais ensuite vous utiliserez exactement les mêmes commandes pour gérer votre projet au quotidien avec GIT.

Préparation

Les ateliers utilisent la même base de départ pour vos expérimentations.

Voici donc comment créer cette base de départ :

- Ouvrez un Terminal et placez-vous où vous souhaitez travailler.
- Créez un dossier de travail **`atelier_git/`**. Dans la suite on appellera ce dossier le *dossier de l'atelier*.
- Créez un dossier **`atelier_git/site_web`**.
- Créez un fichier **`atelier_git/site_web/index.html`** avec un contenu **`HTML`** de base (à vous de le saisir).
- Tout le reste se passe dans le dossier de l'atelier : **`atelier_git/`**

Afin de pouvoir revenir facilement consulter ce sujet référence, vous trouverez une section, en début de chaque atelier, qui annonce les sujets étudiés dans l'atelier.

Cas 1 : Initialisation de GIT sur un nouveau projet

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- Initialisation d'un dépôt vide : **`git init`**
- Affichage de l'état du dépôt local, du projet : **`git status`**

Expérimentations

Le dossier de l'atelier contient un dossier **`site_web/`** qui est un site Web embryonnaire avec juste un **`index.html`**

Avec un :

```
ls -la
```

On observe qu'il n'y a strictement rien d'autre dans le dossier de l'atelier.

On décide donc d'utiliser le dossier de l'atelier comme base de départ d'un nouveau projet.

Pour cela, il suffit d'initialiser le projet avec la commande :

```
git init .
```

Le **`.`** indique la racine du projet à initialiser, ici c'est le dossier courant, mais on aurait aussi pu choisir un nom de dossier existant, comme **`site_web`**. GIT aurait alors utilisé le dossier **`site_web`** comme racine du projet dans ce cas.

Avec un autre :

```
ls -la
```

On observe maintenant la présence d'un nouveau dossier **`.git`**, c'est le dépôt local !

Jetez un œil au contenu de ce dossier :

```
ls -la .git
```

Il contient tout le paramétrage du projet GIT ainsi que les fichiers du dépôt, les clichés de l'album photos du projet.

> On évitera désormais de toucher à quoi que ce soit dans ce dossier **`.git`** sans y avoir été invité.
>
> Si ce dossier disparaît ou est altéré, c'est tout le dépôt local qui disparaît aussi. C'est une sorte de base de données du projet pour GIT.

On peut maintenant observer l'état des fichiers du projet à l'aide d'une nouvelle commande qui est :

```
git status
```

NB : Les commandes **`git`** peuvent s'exécuter depuis n'importe quel dossier ou sous-dossier du projet, pas besoin d'être sur la racine, GIT remontera l'arborescence jusqu'à rencontrer un dossier **`.git`** qu'il considérera alors comme la racine du projet.

On peut voir que le dossier **`site_web/`** est **`untracked`**. Ca peut sembler étrange que GIT nous parle ici d'un dossier, car on a vu en introduction que GIT ne s'occupe que de fichiers et pas des dossiers. Quand GIT affiche qu'un dossier n'est pas *tracké*, il faut en fait comprendre que tous les fichiers du dossier en question sont **`untracked`**. C'est sa façon de résumer cela. Ça lui évite de lister, comme étant **`untracked`**, tous les fichiers contenus dans le dossier.

Effectivement, **`index.html`** qui est l'unique fichier de **`site_web/`** est actuellement **`untracked`**, non supervisé par GIT, il ne fait pas partie de la famille, il n'est donc pas encore dans l'album de famille ni évidemment sur aucune photo.

C'est normal, on vient tout juste de créer le *dépôt local*, et on n'a encore rien mis dedans.

Le fait qu'un fichier soit présent dans le dossier du projet supervisé par GIT ne suffit pas à ce que le fichier soit pris en charge par GIT. On se rappelle des différents états d'un fichier. Celui-ci est simplement **`untracked`**, non supervisé.

Pour le moment, on le laisse ainsi, on va voir dans un prochain atelier comment le placer sous la supervision de GIT.

On peut aussi créer un nouveau dépôt local sans avoir de dossier projet déjà existant. Il suffit de faire ceci dans un dossier qui n'est pas encore sous contrôle de GIT (ne le faites donc pas en étant dans le dossier **`atelier_git`** !!!) :

```
git init nom_projet
```

où **nom_projet** est un dossier inexistant. Ca créera un dossier **nom_projet** racine du nouveau projet et GIT y placera aussi son dossier **.git**, mais le projet sera simplement vide.

Rappel : Ne créez pas des projets GIT dans un autre projet GIT. Pas de projets imbriqués, c'est source de problèmes !

Cas 2 : Initialisation de GIT par clonage d'un projet existant

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- Cloner un dépôt : **git clone**
- Synchroniser le dépôt local vers le dépôt distant : **git push**
- Synchroniser le dépôt distant vers le dépôt local : **git pull**

Expérimentations

Pour le moment vous ne pouvez pas expérimenter ce qui suit (d'ici jusqu'au paragraphe **Atelier 2**) car vous n'avez pas encore accès à votre compte Gitlab sur les serveurs de l'IUT. Lisez donc attentivement ces quelques pages (jusqu'à l'**Atelier 2**), sans taper aucune commande. Vous pourrez y revenir très prochainement, quand on vous aura donné accès à vos comptes.

Cependant, si vous le souhaitez vous pourrez aussi, après la fin de ce TP, créer un compte sur GitLab.com ou GitHub.com et expérimenter par vous même à l'IUT, chez vous et, intérêt principal, entre ces deux endroits. GitLab est peut-être le meilleur choix car il permet de créer autant de projets privés que vous voulez. Privilégiez plutôt des projets privés pour vos travaux (TP, TD etc).

Introduction

Cloner un projet depuis un Dépôt Distant a plusieurs avantages :

- Pouvoir travailler et synchroniser son travail sur plusieurs machines : par exemple à l'IUT et chez vous.

- Pouvoir travailler en groupe, chacun sur sa ou ses machines.
- Pouvoir sauvegarder son dépôt local.

Attention, les dépôts distants sont généralement publics. Un projet peut être privé sur certains dépôts publics mais le bon sens et la prudence sont évidemment de rigueur.

> Ne placez **JAMAIS** de données sensibles sur un dépôt public ! Pas de clés SSH privées, de mots de passe, d'informations sensibles sur l'infrastructure de votre réseau, de votre architecture logicielle etc. Bref, rien que vous ne souhaiteriez pas voir divulgué !

Cloner en SSH

Pour des contraintes techniques, vous ne pourrez pas utiliser SSH à l'IUT, même s'il s'agit de la façon de faire la plus simple et la plus conviviale.

Elle est totalement compatible avec la méthode suivante (HTTPs), donc si vous souhaitez l'expérimenter chez vous, voici un tuto (à regarder chez vous) :

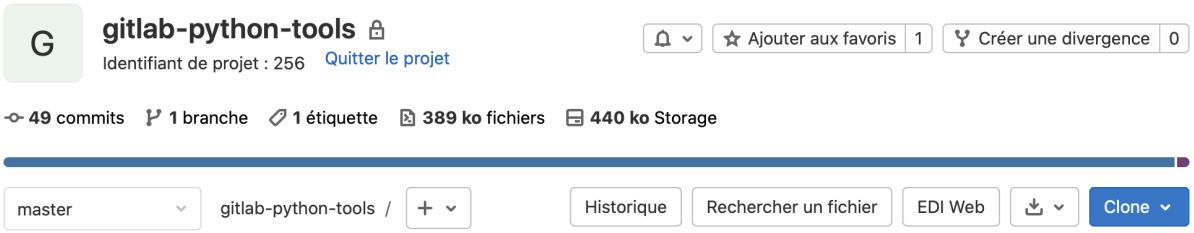
https://www.youtube.com/watch?v=TzPf1UtAnlg

Cloner en HTTPs

Pour cloner un projet, la commande à utiliser est :

```
git clone https://url_vers_le_projet_git(lab|hub)
```

Vous trouverez le **`https://url_vers_le_projet_git(lab|hub)`** du projet en cliquant sur le bouton **`Clone`** de votre dépôt distant. Exemple :



Si vous spécifiez un paramètre supplémentaire après l'URL, dans la commande **`git clone`**, ça vous permet d'indiquer un nom de dossier à créer pour héberger votre projet local, sinon il portera le même nom que le projet du dépôt distant.

Cas particulier du serveur IUT : il utilise un certificat SSL auto-signé, vous devrez donc ajouter ceci à la commande git clone :

```
git clone -c http.sslVerify=false https://...
```

Pour un dépôt sur un serveur public (GitLab, GitHub), vous n'aurez pas cette particularité.

Vous devrez ensuite saisir vos identifiants (login et mot de passe) de compte sur le serveur du dépôt distant.

Vous devrez saisir ces identifiants à chaque action vers ou depuis ce serveur, ce qui peut rapidement être ennuyeux (vous n'aurez pas ce soucis avec SSH). Pour éviter cela, deux solutions :

- Stocker vos identifiants dans un fichier local. C'est fortement déconseillé, et de ce fait, nous n'entrerons pas en détail sur la façon de procéder. Tout est décrit ici si besoin : **https://git-scm.com/book/fr/v2/Utilitaires-Git-Stockage-des-identifiants**
- Garder en mémoire (uniquement) les login et mot de passe durant un temps limité, par exemple 1 ou 2H. Cette conservation en mémoire sera perdue au logout.

  Pour ce faire, voici la commande à saisir :

  ```
  git config --global credential.helper cache
  ```

  par défaut le timeout sera de 15 min. Pour une valeur différente, exprimée en secondes, utilisez (exemple pour **30000** secondes = env. 8H) :

  ```
  git config --global credential.helper "cache --timeout 30000"
  ```

  Entre deux sessions vos identifiants sont oubliés mais la configuration (**--global**) est conservée. Vous n'aurez donc à saisir vos identifiants qu'une seule fois au début de session.

Synchronisation

Pour synchroniser un projet du local vers le dépôt distant, la commande à utiliser est :

```
git push
```

Pour synchroniser un projet du dépôt distant vers le local, la commande à utiliser est :

```
git pull
```

Attention, la synchronisation ne s'occupe que de ce qui est commité !

Cycle d'activité classique :

- Vous arrivez (à l'IUT ou chez vous) : **`git pull`**
- Dans la journée, vous travaillez sur votre dépôt local : **`git add`** et **`git commit`** (2 commandes vues plus loin dans l'**Atelier 2**)
- Avant de partir (de l'IUT ou de chez vous) : **`git push`**

Pour les projets, vous aurez des dépôts dédiés qu'on vous créera.

Pour vos TP/TD, vous pouvez créer un seul et gros dépôt globalisant vos travaux (sur GitLab par exemple), en les structurant par dossiers de matière et sous-dossiers. Ainsi en un seul **`git push/pull`** vous synchronisez tous vos travaux personnels de la journée.

Cas 3 : Rattachement d'un projet GIT local (Cas 1) à un dépôt externe

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- Ajouter un dépôt distant : **`git remote add origin`**
- Synchroniser vers un dépôt distant : **`git push origin`**

Expérimentations

Aucune pour ce TP.

Explications

Si vous n'êtes pas parti sur le clonage d'un dépôt existant mais directement sur la création d'un dépôt local, ça ne signifie pas que vous ne pouvez pas bénéficier d'un dépôt distant et de tous ses avantages.

Il vous suffit simplement de rattacher votre dépôt local à un dépôt distant, que vous aurez pris soin de créer au préalable sur le service de votre choix.

Pour ce faire, utilisez la commande suivant, depuis votre dossier de projet local :

```
git remote add origin https://...
```

L'URL est la même que dans le **Cas 2** (ici en HTTPs mais ça fonctionne aussi en SSH).

Ensuite les **`push/pull`** fonctionneront aussi comme expliqué dans le **Cas 2**. Cependant, il est possible que le 1er push nécessite cette syntaxe :

```
git push origin master
```

Atelier 2 - Actions de base sur le dépôt local

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- Affichage du résumé de l'état local du projet : **`git status`**
- Ajout d'un nouveau fichier dans l'index : **`git add`** sur fichier et sur dossier
- Envoi de l'index dans le dépôt local : **`git commit`**
- Renseignement de l'identité de l'utilisateur :
  - **`git config --global user.name "Votre nom"`**
  - **`git config --global user.email "votre@email"`**
  - **`git config --global core.editor nano`**
- Suppression d'un fichier de l'index : **`git rm --cached fichier`**

Expérimentations

On poursuit dans le dossier **`site_web/`** de l'Atelier 1

Vérifions immédiatement quel est l'état de notre projet :

```
cd site_web/
git status
```

Le projet est bien sous la supervision de GIT, et on observe que le dossier **.** qui est **site_web/** et qui contient le fichier **index.html**, est tel qu'on l'avait laissé, il est **untracked** , c'est à dire *non supervisé par GIT*.

On se rappelle qu'un fichier **untracked** qu'on souhaite voir supervisé par GIT doit être placé sur le **stage**, qu'on appelle aussi **index**, et que ça revient à prendre un cliché du fichier tel qu'il est maintenant.

Pour ce faire, il faut utiliser la commande :

```
git add index.html
```

On vérifie maintenant le nouvel état de notre projet avec encore une fois un :

```
git status
```

C'est une commande que vous utiliserez très fréquemment.

GIT nous dit maintenant que le fichier **index.html** fait partie de la famille, fait partie du projet. Il s'agit même d'un nouveau venu puisqu'il est marqué **new file**.

Attention, pour le moment il est uniquement dans le **stage**, il n'est pas encore arrivé dans le *dépôt local*, qui, rappelons-le, est le seul état à proposer un historique et à être synchronisable avec un dépôt distant, mais ça, on n'y est pas encore. Intéressons-nous déjà au dépôt local.

Pour envoyer dans le dépôt local tous les fichiers du **stage**, c'est-à-dire, pour le moment, uniquement le fichier **index.html**, on va utiliser une nouvelle commande GIT qui est :

```
git commit
```

A ce stade, il est possible que vous ayez une erreur qui surviendra uniquement lors de votre 1ère utilisation de GIT.

Si vous n'avez pas d'erreur, sautez à la partie préfixée d'un 👉 un peu plus loin.

L'erreur indique en principe que GIT ne vous connaît pas encore et il a absolument besoin d'avoir 2 informations vous concernant : votre nom et votre adresse mail. Pour quelle raison ? Simplement parce que ces informations seront associées à chaque

changement que vous ferez dans le dépôt et c'est important, surtout quand vous travaillez en équipe car ça permettra de savoir qui a modifié quoi et quand. Ce sont des informations qui apparaîtront dans l'historique du projet, dans la frise chronologique notamment.

Renseignez ces informations tel qu'il le suggère, de cette manière :

```
git config --global user.email "votre_email_UR1"
git config --global user.name "vos_nom_prenom"
git config --global core.editor nano
```

L'option **`--global`** permet d'écrire ces informations dans un fichier de configuration qui sera personnel mais valable pour tous les projets que vous utiliserez. Le fichier de configuration est **`~/.gitconfig`**

Maintenant vous pouvez refaire votre **`git commit`**.

👉 Cette fois-ci ça fonctionne et on se retrouve dans un éditeur de texte.

Pourquoi arrive-t-on dans un éditeur de texte ? Tout simplement parce qu'un **`git commit`** doit toujours s'accompagner d'un message décrivant la raison du *commit*, et GIT vous donne la possibilité de saisir ça dans un éditeur de texte. Au passage, il vous indique différentes informations, et notamment la liste des fichiers du **`stage`** qui vont être *commités*. Ces informations sont précédés d'un **`#`** mais ne figureront pas dans le message associé au *commit*.

Vous devez impérativement saisir un commentaire quelconque dans cet éditeur de texte sinon le **`commit`** sera abandonné. Vous pouvez le placer en 1ère ligne et supprimer ou laisser les autres lignes actuelles qui, puisqu'elles commencent toutes par un **`#`**, ne seront, de ce fait, pas utilisées pour constituer le commentaire.

Une fois le commentaire saisi, il faut quitter l'éditeur en sauvant et en utilisant le nom de fichier proposé par l'éditeur. Ne cherchez pas à renommer ce fichier-commentaire car il s'agit d'un fichier temporaire que Git a choisi pour vous et qu'il s'attend donc à retrouver (rempli) quand vous allez quitter l'éditeur de texte. Sinon ça ne marchera pas et le *commit* n'aura pas lieu en quittant l'éditeur.

Par défaut, l'éditeur de texte est **`nano`**. Ça pourra se modifier ultérieurement.

Rappel : en quittant l'éditeur de texte, le *commit* sera effectué, mais uniquement à la condition d'avoir saisi un message dans l'éditeur, sinon le *commit* sera abandonné.

Cependant, il existe une façon plus rapide de procéder.

Modifiez un peu votre **`index.html`**

Vous devez de nouveau envoyer votre modification sur le **`stage`**, car rappelons qu'envoyer un fichier sur le **`stage`** revient à figer son image, c'est-à-dire son contenu, à

l'instant où on effectue cette action. Comme on vient de re-modifier **index.html**, on doit donc refaire un :

```
git add index.html
```

Un **git status** nous le montre cette fois-ci comme **modified** puisqu'il fait partie de la famille, de notre projet, depuis le **commit** précédent. Il n'est donc plus **new**.

Il ne nous reste plus qu'à *commiter* une nouvelle fois, mais cette fois-ci, on va écrire la commande différemment, de cette manière :

```
git commit -m "Un message detaillant la modif faite..."
```

De cette manière, on ne passe plus par un éditeur de texte. Le **-m** permet d'indiquer directement le message associé au **commit**.

Vérifiez ensuite que tout est bien *commité* : **git status**. Rien d'autre ne s'affiche, ce qui signifie que le **stage** a bien été commité.

Maintenant, supposons que vous ayez ajouté au **stage** un fichier par erreur et que vous souhaitiez le retirer, car sinon il sera ajouté au dépôt et ce sera définitif. Ça peut être un fichier supervisé par GIT et dont la photo que vous avez prise ne vous convient pas, parce qu'elle n'est pas encore dans l'état souhaité ou pire, ça peut être un fichier qui ne doit jamais faire partie du projet dans GIT. Vous devez alors le retirer du **stage**.

Dans le dossier de l'atelier, faites ceci :

```
echo "La cle est sous le pot de fleurs" > secret
git add secret
git status
```

On peut voir que le fichier **secret** est un nouveau fichier pour notre projet et, comme son nom l'indique, il contient des choses secrètes. Il vient d'être placé sur le **stage**.

Placer dans GIT des fichiers confidentiels est une très mauvaise idée.

La commande à utiliser pour le retirer du stage (avant qu'il ne soit trop tard) est :

```
git rm --cached secret
```

Avec un **`git status`** on vérifie que le fichier secret est de nouveau **`untracked`**.

Le **`rm`** de cette commande peut éventuellement faire peur, mais il ne supprime pas le fichier, il le retire simplement du **`stage`**.

> Une fois *commité* ce sera tard, même si on retire le fichier du projet ensuite, il restera toujours dans son historique.

Pour en terminer avec **`add`** et **`commit`** qui sont les actions les plus fréquentes que vous aurez à faire dans GIT, vous rencontrerez certainement souvent le cas où vous avez à ajouter en même temps un grand nombre de de fichiers modifiés ou créés.

On pourrait les ajouter à la main un par un en faisant des **`git add`** sur chaque fichier mais c'est très fastidieux. il y a une autre solution qui est de faire, dans le dossier qui contient les fichiers, un :

```
git add .
```

ou un :

```
git add dossier_contenant_les_fichiers
```

Attention, ça va ajouter tous les fichiers du dossier en question. Y compris des fichiers que vous ne souhaitez pas voir pris en compte par GIT. A utiliser avec parcimonie et en bonne connaissance et compréhension de ce que vous faites !

Précautions

Pour terminer, quelques règles d'or concernant le **`commit`** :

Le **`commit`** ne doit contenir que des fichiers en rapport avec l'état qu'on souhaite figer dans l'historique. Par exemple, ne mettez pas dans un même commit des fichiers modifiés ou ajoutés pour une nouvelle fonctionnalité sur laquelle vous êtes en train de travailler, et des fichiers concernés par une correction de bug sans aucun rapport avec cette fonctionnalité. Faites deux commits ou même plus si besoin.

Commitez fréquemment. Rappelez-vous que vos commits sont comme des photos de votre projet. Si ces photos sont suffisamment proches les unes des autres, vous avez maintenant un film de votre projet. Avec un film, il est beaucoup plus facile de remonter et de s'arrêter à un instant exact de la vie de votre projet plutôt que de faire de grands bonds qui ne seront jamais les bons moments qui vous intéressent, ce sera toujours soit trop tôt soit trop tard. Il vous faudra alors reconstruire ou déconstruire en tâtonnant pour rétablir cet instant qui n'existe pas en tant que tel dans l'historique.

Accompagnez votre **`commit`** d'un message significatif. Pas besoin de raconter votre vie, mais évitez les commentaires vides ou sans intérêt. Par exemple *Modif*, *Nouvelle version* ou *Correction bug* sont des commentaires sans aucun intérêt. Pensez à vos commits comme à des boîtes d'archive et le commentaire comme à une étiquette sur la boîte. Si l'étiquette dit *Documents* vous voilà bien parti pour savoir ce que contient votre boîte sans l'ouvrir ! Pour GIT c'est pareil. Ecrivez plutôt *Ajout images du menu*, *Fonction de tri des articles* ou encore *Corrige débordement tableau produits*. Même si vous utilisez un style télégraphique, le temps perdu à rédiger un bon commentaire sera gagné au centuple quand vous chercherez la bonne boîte !

Pour commiter, il n'y a pas de règle en terme de temps passé ou de taille de code tapé. La règle est simplement contextuelle. Quand vous avez fait une modification qui est terminée et qu'on peut la décrire simplement par un commentaire, on *commite*. Cependant, même s'il n'y a pas de règle de temps passé, sauf à s'endormir sur son clavier, ne pas avoir *commité* 2-3 fois en une heure est assez rare. Généralement on arrive quand même à produire 2-3 fonctions qui feront d'excellentes candidates au **`commit`**, et si c'est plus c'est très bien aussi. En tout cas, si vous travaillez sur un projet 3 semaines et que vous *commitez* une fois à la fin, vous avez dû rater un truc.

Un autre moment clé pour *commiter* est quand vous sentez que vous allez vous lancer dans une modification hasardeuse et que vous aimeriez bien pouvoir revenir en arrière, c'est certainement un bon moment pour *commiter*, fiez-vous à votre instinct qui vous dit que cette modification que vous vous apprêtez à faire va certainement partir en sucette !

Un dernier mot au sujet des *commits* : GIT est très intelligent dans son organisation des dépôts. Un **`commit`** n'est pas plus gros que la somme des seuls fichiers modifiés. Même si un projet contient des milliers de fichiers et qu'un seul d'entre eux est modifié pour un **`commit`**, la boîte représentant ce commit, même si elle représente l'intégralité du projet, c'est à dire des milliers de fichiers, cette boîte aura la taille du seul fichier modifié pour ce **`commit`**, pas plus.

Donc n'ayez pas peur de commiter. GIT est fait pour ça.

Atelier 3 - Historique

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- Affichage de l'historique :
  - **`git log`**
  - **`git log --oneline`**
  - **`git log --oneline --graph`**
  - **`git log -- `*`nom_fic`***
  - **`git show `*`commit_id`***

Expérimentations

On a vu comment ajouter des fichiers nouveaux ou modifiés au **`stage`**, puis comment envoyer le **`stage`** dans le *dépôt local* par des **`commits`** qui sont toujours accompagnés d'un commentaire qui doit être quelque chose de parlant.

Voyons maintenant comment on peut consulter l'historique des commits.

La commande de base pour accéder à l'historique est :

```
git log
```

L'historique qui s'affiche doit être assez court, et cet historique est à lire à l'envers, en haut se trouve le dernier commit.

Chaque ligne d'historique indique le message associé au **`commit`**, accompagné du nom de l'auteur, c'est à dire du nom et du mail de celui qui a effectué le **`commit`** ainsi que la date et heure de ce **`commit`**. On parle bien de celui qui a *commité* et pas du moment ni de qui a effectué les modifications sur les fichiers, mais en général c'est bien sûr la même personne. On peut aussi observer que chaque **`commit`** possède un **`commit ID`**. qui identifie de façon unique un **`commit`**. C'est une longue chaîne alphanumérique.

Un historique peut-être et est souvent beaucoup plus gros que ces quelques **`commits`** de notre exemple. Si la liste est longue, l'affichage sera paginé comme avec un **`less`**.

Il y a un moyen de faire plus condensé en utilisant la commande :

```
git log --oneline
```

Là encore l'affichage sera paginé mais il sera beaucoup plus concis. On a aussi l'**ID** qui s'affiche de façon tronquée avec seulement les 7 premiers caractères de chaque **ID**. Généralement c'est largement suffisant pour ne pas avoir d'**ID** commençant par la même séquence de 7 caractères. On va voir un peu plus loin à quoi peuvent servir ces **IDs**.

Un **commit** peut concerner des dizaines, voire des centaines ou même des milliers de fichiers à la fois. Supposons que vous ayez introduit une erreur dans un de vos fichiers à un moment donné, mais que vous ne savez pas exactement quand ni dans quel **commit** c'est arrivé. Il y a une commande qui va vous être très utile pour ça.

Supposons qu'il s'agisse du fichier **index.html** de votre petit site Web. Avec la commande :

```
git log -- index.html
```

vous pouvez afficher l'historique et filtrant uniquement sur les **commits** où **index.html** a été impliqué. Bien entendu, on peut faire un :

```
git log --oneline -- index.html
```

pour avoir quelque chose de plus concis. Attention, les **--** devant le **index.html** doivent être obligatoirement encadrés par un espace devant et un espace derrière. Sans entrer dans les explications détaillées, sachez que même s'ils ne sont pas toujours nécessaires, il est conseillé de toujours mettre ces **--**.

Maintenant, si vous voulez avoir un aperçu de ce qui a évolué pour le fichier **index.html** dans un **commit** donné, on peut utiliser la commande :

```
git show ID -- index.html
```

En principe, on peut voir ce que vous avez modifié dans ce fichier à l'Atelier 2. L'ancien contenu est affiché en rouge et le nouveau en vert. Notez aussi que seules les portions modifiées sont affichées, et elles le sont en contexte, c'est-à-dire avec 2-3 lignes avant et après le bloc modifié. Notez aussi que l'**ID** n'a pas besoin d'être donné dans sa version intégrale, en utilisant les quelques 1ers caractères (au moins 4), c'est généralement suffisant. Là encore les **--** sont fortement recommandés, si ce n'est même obligatoires.

Atelier 4 - Branches

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- **`HEAD`**
- **`commit IDs`**

Expérimentations

En introduction, on a évoqué les branches comme étant des filets de sécurité pour expérimenter des choses sans risquer de casser ce qui est actuellement propre et fonctionnel.

Pour le moment, quand on affiche l'historique, on est sur une branche qui est la branche principale de notre projet. Cette branche s'appelle le **`master`** ou le **`main`**. Nous l'appellerons le **`master`** dans ce qui suit.

Supposons qu'on veuille modifier notre projet sans perturber l'état du **`master`** qui pourra aussi évoluer de son côté. La solution déjà évoquée est donc de créer une branche pour qu'on puisse travailler tranquillement de notre côté. Pour notre exemple l'objectif est de modifier le fichier **`index.html`** pour lui ajouter le support de styles CSS alors que pour le moment, dans le **`master`**, le CSS n'est pas encore pris en compte dans le **`index.html`**.

Nous allons créer une branche dédiée qu'on va appeler *ajout_style* :

```
git branch ajout_style
```

Immédiatement après, on refait un :

```
git status
```

et on s'aperçoit qu'on est toujours sur le **`master`**. Mais qu'a donc fait ce **`git branch`** ?

Il a effectivement créé la branche *ajout_style* qui est une copie complète de la branche sur laquelle on est actuellement, c'est-à-dire le **`master`**, et… c'est tout. On est, pour le moment, encore assis sur la branche **`master`**, ça veut donc dire que si on fait des modifications, on va les faire sur le **`master`**. Et ce n'est pas ce qu'on veut faire !

Il nous faut donc changer de branche. Ça se fait par la commande :

```
git checkout ajout_style
```

Un **`git status`** nous confirme cette fois-ci le changement. Mais à ce stade, notre projet est dans le même état que le **`master`**, sauf que nos modifications à venir se feront dans cette branche qui va donc vivre sa vie de branche en toute autonomie par rapport au **`master`**.

Apportez maintenant quelques modifications à **`index.html`** comme convenu :

```
<link rel="stylesheet" href="style.css">
```

Créez aussi un fichier **`style.css`** et placez-y un petit peu de CSS.

Il nous reste à ajouter ces modifications au projet :

```
git add index.html style.css
git commit -m "Prise en charge CSS dans index"
```

Voyons l'état de l'historique maintenant :

```
git log --oneline
```

On peut voir que notre dernier **`commit`** est bien dans la branche *ajout_style* et que le **`master`** est un cran en arrière dans l'historique.

Il faut savoir que le **`master`** représente généralement la branche stable d'un projet, celle qui peut même être éventuellement la version de production, c'est-à-dire, dans le cas de notre site Web, la version du site actuellement en ligne.

Maintenant, supposons qu'un événement particulier nous demande de modifier le **`index.html`** du **`master`**. Or nous sommes en train de travailler sur notre branche *ajout_style* et le travail est encore loin d'être terminé.

Comment peut-on faire dans ce cas pour apporter les modifications demandées sur le **`master`** ? Est-ce que la solution est de créer un autre dossier et de récupérer le projet dans sa version **`master`** ? Evidemment non, le principe des branches est justement là pour nous sauver la vie !

Deux situations peuvent se présenter dans ce cas :

Branche commitée

Notre branche *ajout_style* est dans un état commité, c'est à dire qu'aucun fichier n'est en cours de modification. C'est la situation la plus simple mais pas forcément la plus fréquente. Dans ce cas, il suffit de faire un :

```
git checkout master
```

pour revenir à la branche **master**. Mais qu'est-ce que ça signifie pour nos fichiers ? Testons :

```
cat index.html
```

On a, semble-t-il, perdu la modification d'ajout du style !

Heureusement non, cette modification est dans le dépôt local, bien au chaud, dans sa branche. Il se trouve juste qu'on a quitté l'autre branche, donc on ne voit plus les choses qui sont spécifiques à cette branche là.

Le projet dans notre dossier est toujours le reflet de la branche sur laquelle on est. Cela va pour les modifications des fichiers communs aux deux branches, mais si des fichiers sont présents sur une branche et absents sur une autre, il apparaîtront et disparaîtront du projet au gré des **checkout** qui nous feront passer d'une branche à l'autre. C'est donc ce qui vient de se passer par le **git checkout master**.

Branche non commitée

Deuxième situation, nous sommes en cours de modifications de plusieurs fichiers. Première solution, on *commite* les fichiers tels qu'ils sont actuellement. C'est pas super propre de faire ainsi. En tout cas, sur le **master**, on ne fait pas ce genre de choses, on *commite* uniquement des états propres.

Par exemple, on ne *commite* pas du code où on aurait commencé à écrire une fonction sans la terminer. Dans notre situation, on n'est pas sur le **master**, donc on peut éventuellement *commiter* ce qui est en cours et pas fini. Mais il y a une autre solution qui est d'utiliser le **stash**.

Le **stash** est une sauvegarde locale des modifications faites sur des fichiers depuis le dernier **commit**. Une fois la sauvegarde faite, les fichiers sont restaurés dans la version du dernier **commit**, ce qui va permettre de nous retrouver finalement dans la situation n°1, celle où le projet est dans un état totalement *commité*.

Testons.

On revient sur la branche `ajout_style` qu'on avait quittée précédemment :

```
git checkout ajout_style
```

Modifiez le **index.html** et ajoutez aussi un fichier **menu.html** (peu importe son contenu). On regarde l'état du projet :

```
git status
```

on a vraiment un projet en pleine modification. On va *stasher* ces modifications avec un :

```
git stash
```

puis un **git status** nous montre bien que les changements apportés à **index.html** n'apparaissent plus, et on vérifie avec un **cat index.html** qu'effectivement les modifications ont disparu, le fichier est de retour à son état du dernier **commit**.

Par contre, le fichier **menu.html** est toujours présent. Cela s'explique simplement parce qu'il ne fait pas partie du projet GIT, donc un **stash** ne s'occupe de sauvegarder que les fichiers en cours de modification qui sont supervisés par GIT. Il n'y a pas de problème à ça puisque le **checkout** ne supprimera pas non plus ces fichiers non supervisés, ils sont là et le restent.

Faisons maintenant notre :

```
git checkout master
```

On se retrouve dans la même situation que la 1ère.

Nous pouvons maintenant apporter notre modification au fichier **index.html** du **master**.

Puis mettre sur le **stage** nos modifications et enfin *commiter*.

A vous de faire ces différentes étapes.

Enfin, il faut maintenant revenir sur notre branche de travail *ajout_style* :

```
git checkout ajout_style
```

On revient évidemment au dernier **commit** de la branche *ajout_style*, dans laquelle il nous manque bien entendu les modifications qu'on était en train de faire et qu'on a abandonnées temporairement en les *stashant*.

Il nous reste donc à restaurer le **stash** à l'aide de la commande :

```
git stash apply
```

qui restaure le dernier **stash**.

En effet, les **stash** peuvent être multiples et peuvent aussi être conservés.

La commande :

```
git stash list
```

affiche la liste des **stash** de notre projet.

Il faut noter que les **stash** sont des sauvegardes locales et personnelles, elles ne sont pas partagées entre les membres d'une équipe. Comme un **stash** reste présent, même après avoir fait un **git stash apply** pour le restaurer, il ne faut donc pas oublier de le supprimer s'il ne sert plus à rien, ce qui est le cas pour nous maintenant qu'on a restauré notre branche dans son état en cours de modification. On supprime un **stash** avec la commande :

```
git stash drop
```

Vous voilà expert en gestion de branches !

Atelier 5 - Fusion

Fera l'objet d'un complément ultérieur.

Atelier 6 - Remonter le temps

Sujets étudiés

Donné à titre de référence, voici ce qui est vu dans cet atelier. Ne tapez aucune de ces commandes pour le moment.

- **`HEAD`**
- **`commit IDs`**
- **`git checkout -b`**

Expérimentations

On va maintenant voyager dans le temps. Vous avez pu observer que dans l'affichage d'un **`git log --oneline`** on a généralement une ligne qui indique **`HEAD`**. Mais que signifie ce **`HEAD`** ?

Il s'agit de la branche sur laquelle on se trouve dans l'historique et à chaque **`commit`**, le **`HEAD`** va avancer.

Quand on fait un **`checkout`** d'une branche, le **`HEAD`** est repositionné automatiquement sur le bout de la branche par GIT.

Mais il est aussi possible de faire un **`checkout`** non pas en indiquant une branche mais en indiquant un **`ID`** de **`commit`**. Et dans ce cas, le **`HEAD`** se retrouve détaché de toute branche, ce que GIT appelle un *Detached HEAD*. Ce n'est pas une solution idéale car si on *commite* quoi que ce soit, on va donc le faire de façon détaché d'une branche, et on risque fort de perdre son **`commit`** dans la nature car le seul lien qu'on aura sur lui est son **`ID`**, qui est quand même assez complexe à mémoriser.

Mais, dans ce cas, à quoi peut donc servir un **`checkout`** sur un **`commit ID`** ? Et bien ça peut servir à voir le projet dans un état antérieur sur une de ses branches pour éventuellement analyser les modifications apportées par ce **`commit`**.

Cependant, il est déconseillé d'apporter des modifications et des **`commits`** à partir de cet état. Par contre, il y a un autre usage plus intéressant qui est de créer une nouvelle branche à partir d'un ancien **`commit`**. Par exemple pour expérimenter une modification à partir d'un état intermédiaire, celui du **`commit`** en question.

Pour ce faire, on utilise la commande :

```
git checkout -b nom_de_nv_branche ID_commit
```

Faisons un essai.

On affiche déjà l'historique pour repérer le **commit** qui va nous servir de point de départ de la nouvelle branche :

```
git log --oneline
```

On va simplement partir du 1er commit (remplacez **commitID** par l'**ID** que vous aurez trouvé dans la liste précédente) :

```
git checkout commitID
```

On peut déjà observer que GIT nous prévient qu'on est effectivement avec un *Detached HEAD*.

Revenons sur le master :

```
git checkout master
```

et utilisons la commande (remplacez **commitID** par l'**ID** souhaité) :

```
git checkout -b test ID
```

Maintenant un :

```
git branch
```

nous permet de voir qu'on a bien une nouvelle branche dont le point de départ est bien le 1er commit de votre projet.

Et voilà, c'est ça la machine à remonter le temps de GIT.

Atelier 7 - Dépôt distant

Même s'il s'agit d'un élément essentiel pour le travail en groupe et/ou pour pouvoir synchroniser vos fichiers de l'IUT avec votre ordinateur personnel, le sujet sera abordé très prochainement dans un complément à ce TP.

Dossiers

Retour sur les dossiers vides...

On a évoqué au début de ce TP qu'un dossier ne peut exister dans GIT que parce qu'il contient des fichiers. En gros, GIT ne sait pas prendre de photo vide, il doit toujours y avoir un sujet dans la photo, c'est-à-dire un fichier. Partant de là, une astuce si vous voulez que GIT prenne en compte un dossier vide est tout simplement qu'il ne soit plus vide en y plaçant un fichier fictif et en prenant un cliché de ce fichier et non pas du dossier. Généralement on place un fichier vide et nommé, par exemple, **`.keep`** ou **`.dummy`** ou **`.vide`** si vous préférez le français. Le fait de le nommer **`.quelquechose`** permettra de le masquer un peu puisqu'il ne sert vraiment à rien d'autre qu'à être *photographiable* si on peut dire, par GIT.

Tips

Voici un site pratique pour trouver la bonne commande quand on est perdu !

https://gitexplorer.com/