GIT ET GITHUB

Mickaël LE BRAS

Basé sur le cours de :

Tiffany Lestroubac

Mila Paul

Luc Bourrat

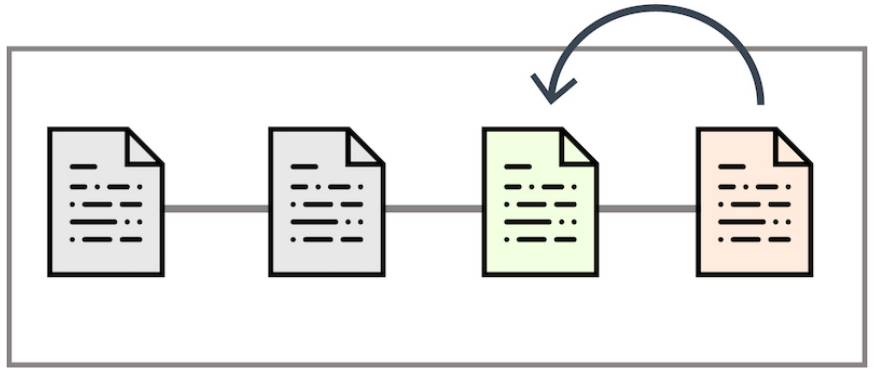
sur Openclassrooms

https://openclassrooms.com/fr/courses/7162856-gerez-du-code-avec-git-et-github/7166041-tirez-le-maximum-de-ce-cours

Le contrôle de versions

Nous allons tout au long de ce cours parler de **Gestionnaire de versions.**

Imaginez, vous avez passé une semaine à écrire votre code, et donc produit votre première version, la V1. Votre collègue vous a fait des suggestions pour l'améliorer. Vous allez donc créer une deuxième version de votre code, la V2. Pour garder un historique de votre travail et éviter les mauvaises surprises (bugs, perte de votre travail), vous aurez besoin d'un **gestionnaire de versions**.



Le gestionnaire de versions

Un gestionnaire de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers.

L'action de contrôler les versions est aussi appelée "versioning" en anglais, vous pourrez entendre les deux termes.

Le gestionnaire de versions permet de garder en mémoire :

- chaque modification de chaque fichier ;
- pourquoi elle a eu lieu et par qui !

Si vous travaillez seul, vous pourrez garder l'historique de vos modifications ou revenir à une version précédente facilement.

Si vous travaillez en équipe, plus besoin de mener votre enquête ! Le gestionnaire de versions fusionne les modifications des personnes qui travaillent simultanément sur un même fichier. Grâce à ça, vous ne risquez plus de voir votre travail supprimé par erreur !

Cet outil a donc trois grandes fonctionnalités :

- Revenir à une version précédente de votre code en cas de problème.
- Suivre l'évolution de votre code étape par étape.
- Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

Pourquoi est-ce utile en travail d'équipe ?

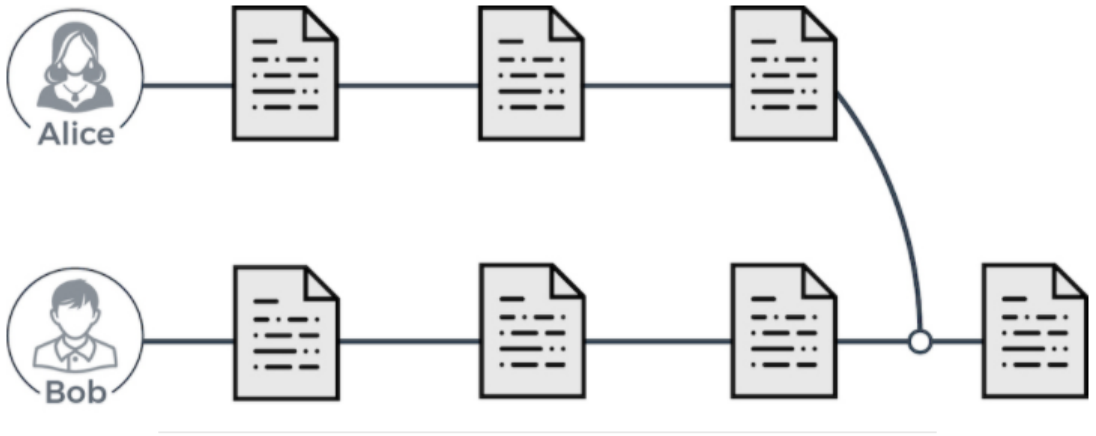
Prenons un exemple concret ! Alice et Bob travaillent sur le même projet depuis un mois. Tout se déroulait à merveille jusqu'à hier : le client leur a demandé de livrer leur projet en urgence. Alice a réalisé en vitesse les dernières modifications, enregistré les fichiers et envoyé le tout au client.

Le lendemain, le client appelle, très énervé : rien ne fonctionne. Alice et Bob ne comprennent pas. Ils s'étaient pourtant bien réparti les tâches et tous les deux avaient fait correctement leur travail.

Le problème ? Sans s'en rendre compte, Alice a écrasé le code de Bob en effectuant ses modifications. Bob n'a pas copié son travail en local, il a donc codé pendant un mois pour rien car il lui est impossible de récupérer son travail.

Tout cela aurait pu être évité avec le gestionnaire de versions !

Si Alice et Bob avaient initialisé Git pour leur projet, ils auraient pu modifier leurs fichiers, envoyer et recevoir les mises à jour à tout moment, sans risque d'écraser les modifications de l'autre. Les modifications de dernière minute n'auraient donc pas eu d'impact sur la production finale !



Alice et Bob ont initialisé Git. Ils travaillent chacun sur leur partie du code. Quand ils les regroupent, leurs versions précédentes sont archivées.

Différence entre GIT et GITHUB

Git et GitHub sont deux choses différentes !

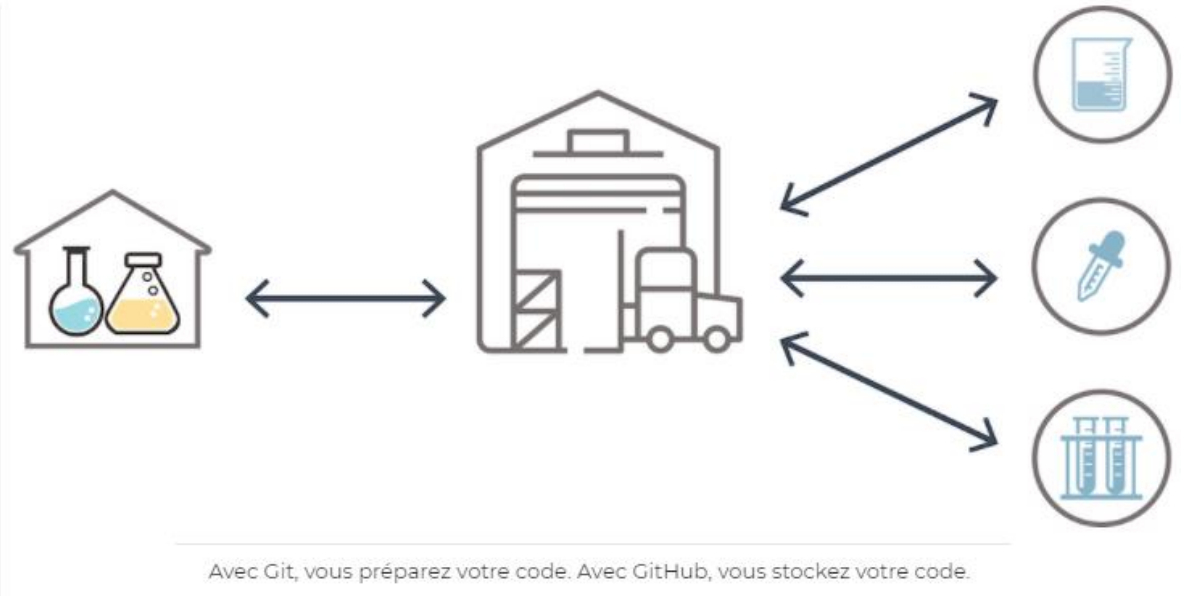
Git est un gestionnaire de versions. Vous l'utiliserez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub est un service en ligne qui va héberger votre dépôt. Dans ce cas, on parle de **dépôt distant** puisqu'il n'est pas stocké sur votre machine.

C'est un peu confus ? Pas de panique, prenons un exemple pour illustrer cela.

Imaginez, vous participez à la préparation d'un parfum. Chez vous, vous créez la base du parfum en mélangeant différents ingrédients. Puis vous envoyez cette base dans un entrepôt où il sera stocké. Cette base de parfum pourra être distribuée telle quelle ou être modifiée en y ajoutant d'autres ingrédients.

Eh bien, c'est la même chose pour Git et GitHub. Git est la préparation que vous avez réalisée chez vous, et GitHub est l'entrepôt où elle peut être modifiée par les autres ou distribuée.



Avec Git, vous préparez votre code. Avec GitHub, vous stockez votre code.

Différence entre un dépot local et distant

Un dépôt est comme un dossier qui conserve un historique des versions et des modifications d'un projet. Il peut être local ou distant.

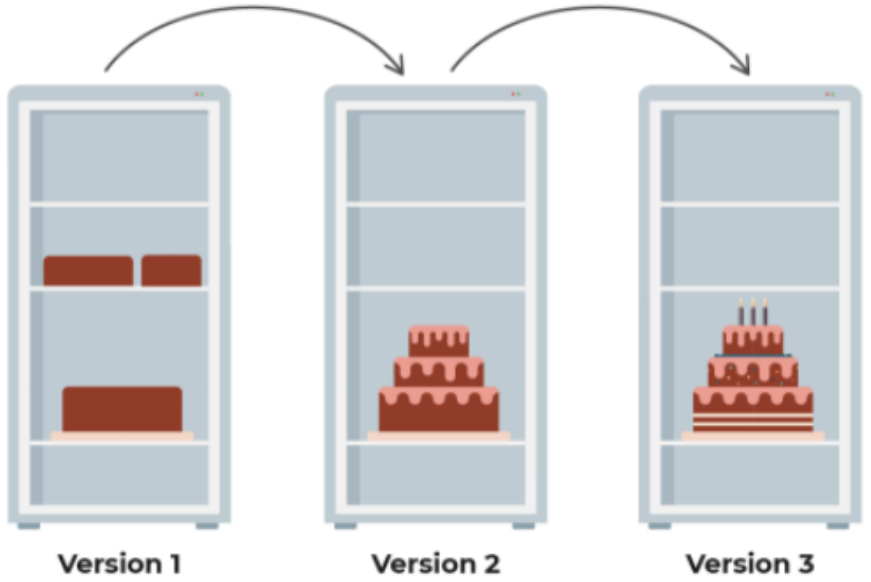
Le dépôt local

Un **dépôt local** est un entrepôt virtuel de votre projet. Il vous permet d'enregistrer les versions de votre code et d'y accéder au besoin.

Pour illustrer cette idée, prenons l'image de la réalisation d'un gâteau. Pour faire un gâteau, vous allez réaliser les étapes suivantes :

- préparer la pâte du gâteau ;
- stocker cette pâte au réfrigérateur ;
- réaliser la crème et en garnir la pâte ;
- stocker le gâteau assemblé au réfrigérateur ;
- décorer votre gâteau ;
- remettre le gâteau au réfrigérateur.

Dans cet exemple, le réfrigérateur est comme un dépôt local : c'est l'endroit où vous stockez vos préparations au fur et à mesure.



Utilisez un dépôt local comme un réfrigérateur et préparez votre gâteau !

Un **dépôt local** est utilisé de la même manière ! On réalise une version, que l'on va petit à petit améliorer. Ces versions sont stockées au fur et à mesure dans le dépôt local.

Le dépôt distant

Le **dépôt distant** est un peu différent. Il permet de stocker les différentes versions de votre code afin de garder un **historique délocalisé**, c'est-à-dire un historique hébergé sur Internet ou sur un réseau. Vous pouvez avoir plusieurs dépôts distants avec des droits différents (lecture seule, écriture, etc.).

Imaginez que votre PC rende l'âme demain, vous aurez toujours vos programmes sur un dépôt distant ! C'est pourquoi, il est recommandé de toujours commencer par copier vos sources sur un dépôt distant lorsque vous commencez un nouveau projet, avec GitHub par exemple ! Vous pourrez aussi les rendre publics, et chacun pourra y ajouter ses évolutions.
Sur un dépôt public, les personnes pourront collaborer à votre projet alors que sur un dépôt privé, vous seul aurez accès à votre travail (et les personnes que vous aurez autorisé) !

Le dépôt distant est donc un type de dépôt indispensable si vous travaillez à plusieurs sur le même projet, puisqu'il permet de centraliser le travail de chaque développeur. C'est aussi sur le dépôt distant que toutes les modifications de tous les collaborateurs seront fusionnées.

Alors, pourquoi créer une copie locale ?

Tout simplement car votre dépôt local est un clone de votre dépôt distant. C'est sur votre dépôt local que vous ferez toutes vos modifications de code.

Ainsi, les dépôts sont utiles si :

- vous souhaitez conserver un historique de votre projet ;
- vous travaillez à plusieurs ;
- vous souhaitez collaborer à des projets open source ;
- vous devez retrouver par qui a été faite chaque modification ;
- vous voulez savoir pourquoi chaque modification a eu lieu.

Le dépôt distant numéro 1 :

Démarrez votre projet avec GitHub

Créez un compte sur GitHub

Pour créer votre compte GitHub, rendez-vous sur la page d'accueil, cliquez sur Sign up. On vous demandera alors de renseigner un nom d'utilisateur, un e-mail et un mot de passe.
Rendez vous sur : https://github.com/ et cliquez sur **Sign Up**  **Choisissez bien entendu la version gratuite**

Votre tableau de bord
Vous pouvez consulter votre **tableau de bord** personnel pour :
- suivre les problèmes et extraire les demandes sur lesquelles vous travaillez ou que vous suivez ;
- accéder à vos principaux repositories et pages d'équipe ;
- rester à jour sur les activités récentes des organisations et des repositories auxquels vous êtes abonné.

Repository est la traduction en anglais du terme "dépôt".

L'interface repository
L'interface Repositories est l'emplacement où vous pourrez créer et retrouver vos dépôts existants.
Pour créer un projet, il suffit de cliquer sur "Start a project".

Votre profil
Sur votre profil, vous pourrez éditer vos informations, mais aussi voir le total de vos contributions sur les différents projets.

L'onglet Pull requests
Il permetde réaliser des demandes de pull. Les demandes de *pull*, ou extractions, vous permettent d'informer les autres des modifications que vous avez appliquées à une branche d'un repository sur GitHub. Mais on y reviendra plus tard.

Créez votre premier dépôt

Pour mettre votre projet sur GitHub, vous devez créer un **repository** dans lequel il pourra être installé.

Cliquez sur le "+" dans le coin supérieur droit, pour faire apparaître l'option New repository.

Choisissez un nom simple pour votre dépôt. Dans le cadre de ce cours, nous allons préparer un dépôt pour le TP Quizz que vous allez réaliser par groupes. Ainsi le dépôt s'appellera "**TPQuizz**".

Puis, choisissez si vous souhaitez créer un dépôt public ou privé. Nous allons choisir de mettre ce dépôt en public pour le TP.

Enfin, initialisez un **readme** et un **gitignore**.

readme est un fichier qui indique les informations clés de votre projet : description, environnement à utiliser, dépendances possibles et droits d'auteurs.

gitignore est un fichier qui permet d'ignorer certains fichiers de votre projet Git. Nous reviendrons là-dessus plus tard.

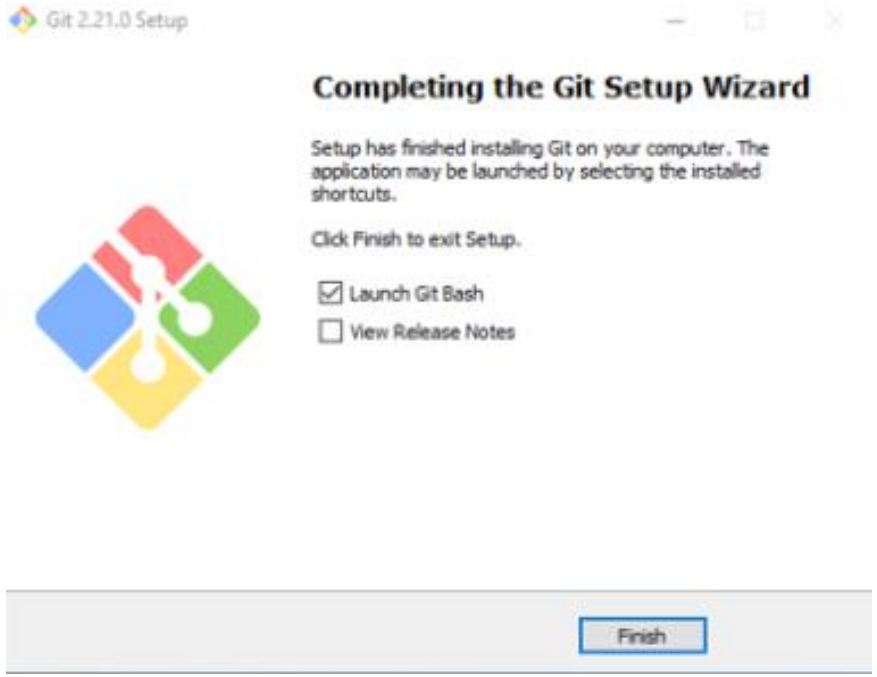
Cliquez ensuite sur "**Créer un dépôt**" ou "**Create repository**".

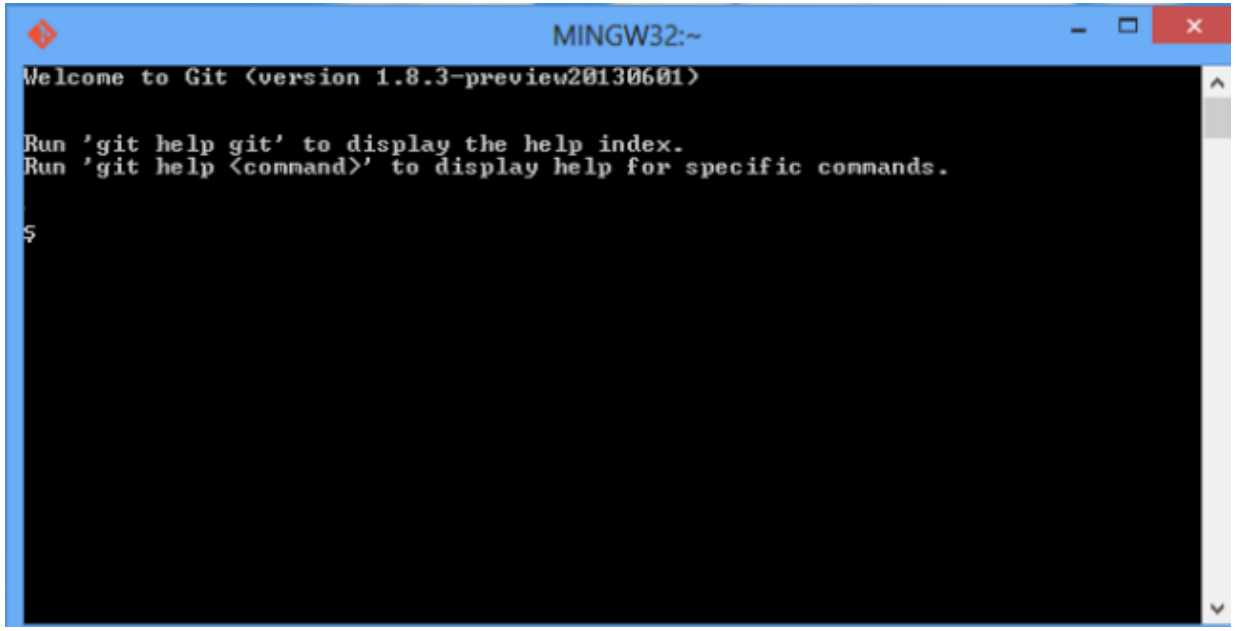
Installez Git sur votre ordinateur

Téléchargez Git en suivant ce lien : https://git-scm.com/downloads. Choisissez la **version** qui correspond à votre installation et téléchargez-la.

Appuyez sur Suivant à chaque fenêtre puis sur **Installer**.
Lors de l'installation, laissez toutes les options par défaut, **SAUF L'ÉDITEUR DE TEXTE (Choisissez VSCODE)**.

Cochez ensuite Launch Git Bash.
Git Bash est l'interface permettant d'utiliser Git en ligne de commande.





Initialisez Git

Pour travailler sur Git, vous devez créer un dépôt local, c'est-à-dire un dossier dans lequel toutes vos modifications seront enregistrées. C'est ce qu'on appelle **initialiser un dépôt Git**.

Configurez votre identité

La première chose à faire est de configurer votre identité.
Commencez par renseigner votre nom et votre adresse e-mail. **C'est une information importante** car vous en aurez besoin pour toutes vos validations dans Git :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Grâce à l'option --global, vous n'aurez besoin de le faire qu'une fois. Si vous souhaitez, pour **un projet spécifique**, changer votre nom d'utilisateur, vous devrez repasser cette ligne mais sans le --global.

Pour vérifier que vos paramètres ont bien été pris en compte, et vérifier les autres paramètres, il suffit de passer la commande `git config –list`

Configurez les couleurs

Je vous recommande d'activer les couleurs afin d'améliorer la lisibilité des différentes branches. Pour cela, passez ces trois lignes dans Git Bash :

```
$ git config --global color.diff auto
$ git config --global color.status auto
$ git config --global color.branch auto
```

Créez votre dépôt local

Les deux méthodes pour créez un dépôt local

Maintenant que vous avez configuré les paramètres de base, vous pouvez **créer votre fameux dépôt local**. Pour ce faire, deux solutions sont possibles :

- **créer un dépôt local** vide pour accueillir un nouveau projet : la procédure est expliquée ci-dessous ;
- **cloner un dépôt distant**, c'est-à-dire rapatrier l'historique d'un dépôt distant en local. Nous aborderons cette méthode dans la deuxième partie du cours.

Initialisez votre dépôt en créant un dépôt local vide

Dans un premier temps, créez un dossier dans votre environnement de travail (avec le nom de notre projet, c'est mieux).
Wamp/www/TPQuizz par exemple

Puis, lancez deux lignes de commandes dans Git Bash :

- `cd [chemin vers wamp]/www/TPQuizz`
- `git init`

La première ligne permet de vous positionner dans le dossier que vous venez de créer sur l'ordinateur. La seconde ligne va initialiser ce "simple dossier" comme un dépôt.

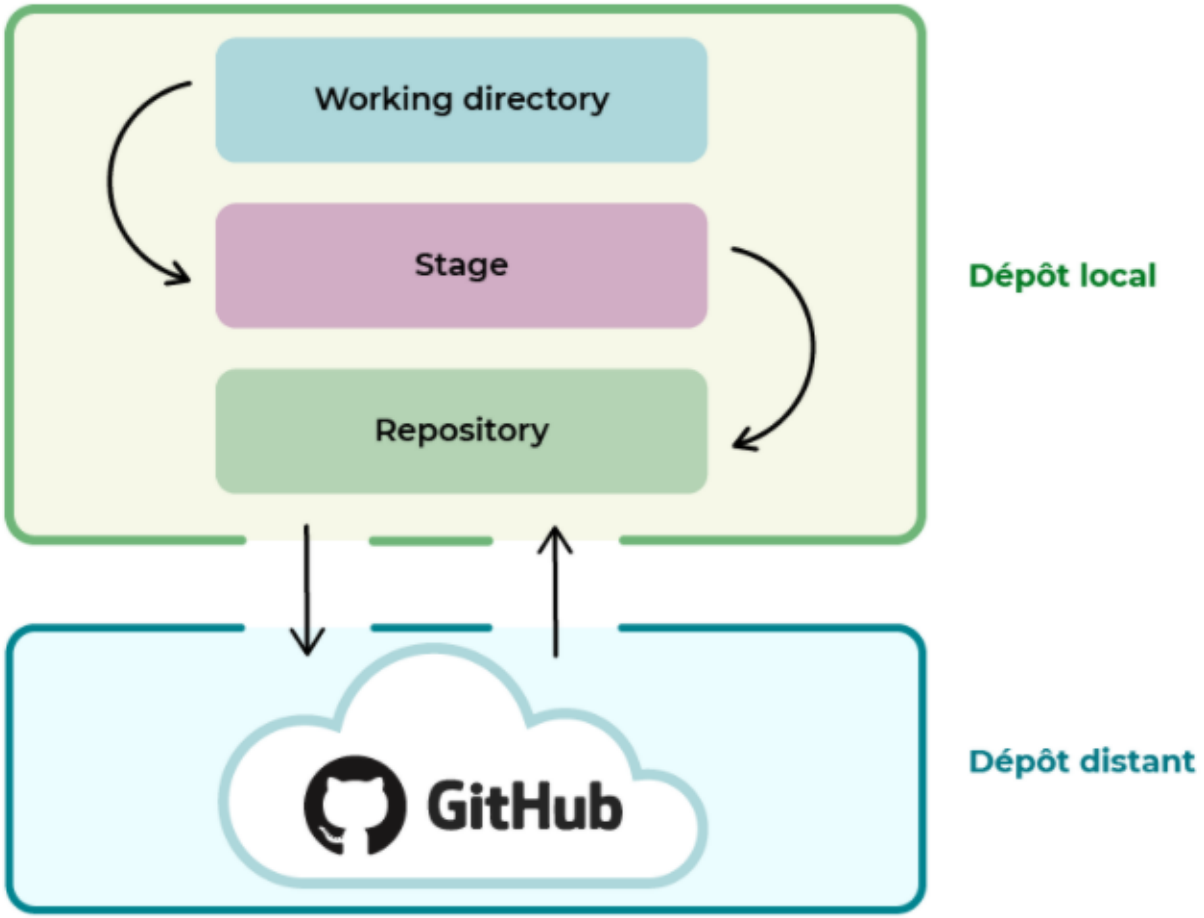
Cette commande a créé un dossier caché dans votre projet nommé .git et qui contient la configuration de git, les "logs" et les branches. On reparlera de ces notions un peu plus tard.

Appréhendez le fonctionnement de Git

Jusqu'ici on a découvert l'utilité du contrôle de version et nous avons initialisé un dépôt Git.

Avant de découvrir les commandes de base, lisons le schéma ci-contre pour mieux appréhender le fonctionnement de Git.

Ce schéma représente le fonctionnement de Git. Il est composé de 3 zones qui forment le dépôt local, et du dépôt distant GitHub.



Le Working directory

Cette zone correspond au dossier du projet sur votre ordinateur.
Souvenez-vous, dans la partie précédente nous avons initialisé le dépôt “TPQuizz”. Eh bien ce dépôt, c’est la zone bleue du schéma.

Le Stage ou index

Cette zone est un intermédiaire entre le working directory et le repository. Elle représente tous les fichiers modifiés que vous souhaitez voir apparaître dans votre prochaine version de code.

Le Repository

Lorsque l’on crée de nouvelles versions d’un projet (vous vous souvenez, les 3 versions différentes du gâteau ?), c’est dans cette zone qu’elles sont stockées.
Ces 3 zones sont donc présentes dans votre ordinateur, en local.

En-dessous, vous trouvez le repository GitHub, c’est-à-dire votre dépôt distant.

Comment ça marche ?

Prenons un exemple pour y voir plus clair !

Imaginez un projet composé de 3 fichiers : fichier1, fichier2 et fichier3.

Nous faisons une modification sur fichier1, puis une modification sur fichier2, depuis le working directory. Super ! Nous avons maintenant une version évoluée de notre projet.

Nous aimerions sauvegarder cette version grâce à Git, c'est-à-dire la stocker dans le repository.

Comment faire ?

Pour commencer, nous allons envoyer les fichiers modifiés (fichier1 et fichier2) du working directory vers l'index. **On dit qu'on va indexer** fichier1 et fichier2. Une fois les fichiers indexés, nous pouvons créer une nouvelle version de notre projet.

Le terme "stage" est aussi beaucoup utilisé par les développeurs à la place du terme "index". On peut dire "indexer un fichier" ou "stage un fichier".

Passons à la pratique

Dans votre Working Directory (votre dossier TPQuizz), mettez en place une base de projet web avec :

- Un fichier index.html

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Un super titre</h1>
</body>
</html>
```

- Un fichier styles.css

```css
h1 {
    color: red;
}
```

Indexez vos fichiers avec la commande git add

Vous avez créé 2 fichiers, index.html et styles.css. Ces fichiers sont situés dans le working directory. Pour créer une nouvelle version de votre projet, vous allez maintenant indexer les fichiers. Pour cela, retournez dans l'outil "Git Bash".

Vous devez vous situer dans le working directory du projet.

Pour le savoir, tapez la commande "pwd" dans Git Bash. Si vous avez créé les mêmes dossiers que moi et que le résultat de la commande "pwd" est : "[chemin vers wamp]/www/TPQuizz".

Si ce n'est pas le cas, utilisez la commande cd pour vous y rendre.

Vous pouvez maintenant faire passer les fichiers index.html et styles.css vers l'index en utilisant la commande "git add" suivie du nom du fichier :

- `git add index.html`
- `git add styles.htm`

Vos deux fichiers sont désormais indexés (sur le schéma étudié précédement, ils sont à la phase "Stage")

Créez une nouvelle version avec la commande git commit

Maintenant que vos fichiers modifiés sont indexés, vous pouvez créer une version, c'est-à-dire archiver le projet en l'état. Pour ce faire, utilisez la commande "git commit" :

- `git commit –m "Ajout des fichiers html et css de base"`

-m (comme message) est ce qu'on appelle un **argument**, qui est ajouté à la commande principale. Ici "-m" permet de définir un message particulier rattaché au commit effectué. Si vous n'utilisez pas cet argument, la commande "git commit" ouvrira un éditeur de texte dans lequel vous pourrez saisir le message de commit.

"Ajout des fichiers html et css de base" est le message rattaché au commit grâce à l'argument "-m".

Vos modifications sont maintenant enregistrées avec la description "Ajout des fichiers html et css de base".

La description est très importante pour retrouver le fil de vos commits, et revenir sur un commit en particulier. Ne la négligez pas ! Il ne serait pas pratique de nommer des commits du type "version 1", "version 2", "version 2.1", car vous seriez alors obligé de lire le fichier pour connaître les modifications réalisées. Avec un message clair, vous pouvez tout de suite identifier les modifications effectuées.

Nous venons donc de créer la toute première version de notre projet. Cette version est pour le moment stockée dans le dépôt local qui correspond au "Repository" dans le schéma vu précédement.

Envoyez votre commit sur le dépôt distant

Créer une version de votre projet, c'est fait. Maintenant il faut passer votre commit du repository (dépôt local) à votre dépôt distant. On dit qu'il faut "pusher" votre commit.

Votre premier push va vous demander un peu de configuration.

Pour commencer, vous allez devoir "relier" votre dépôt local au dépôt distant que vous avez créé sur GitHub précédemment. Pour cela :

- Allez sur GitHub.
- Cliquez sur la petite image en haut à droite.
- Cliquez sur "your repositories".
- Cliquez sur le repository créé dans la première partie du cours, "TPQuizz".

Il va falloir copier le lien qui figure sur l'écran et qui ressemble à : https://github.com/[pseudo]/TPQuizz.git

Maintenant, retournez sur Git Bash et tapez la commande suivante :

- **`git` remote add origin https://github.com/[pseudo]/TPQuizz.git**
- **// Vérifiez à bien remplacer l'url par celle de GitHub**

Ensuite, tapez la commande :

- **`git` branch -M main // Votre repository pourra être divisé sous forme de "branches". Pour le moment on initialise uniquement la branche principale : main.**

Vous pouvez donc envoyer des commits du repository vers le dépôt distant GitHub en utilisant la commande suivante :

- **git push -u origin main**

Vérifiez sur le repository distant si les fichiers index.html et styles.css ont bien été ajoutés !

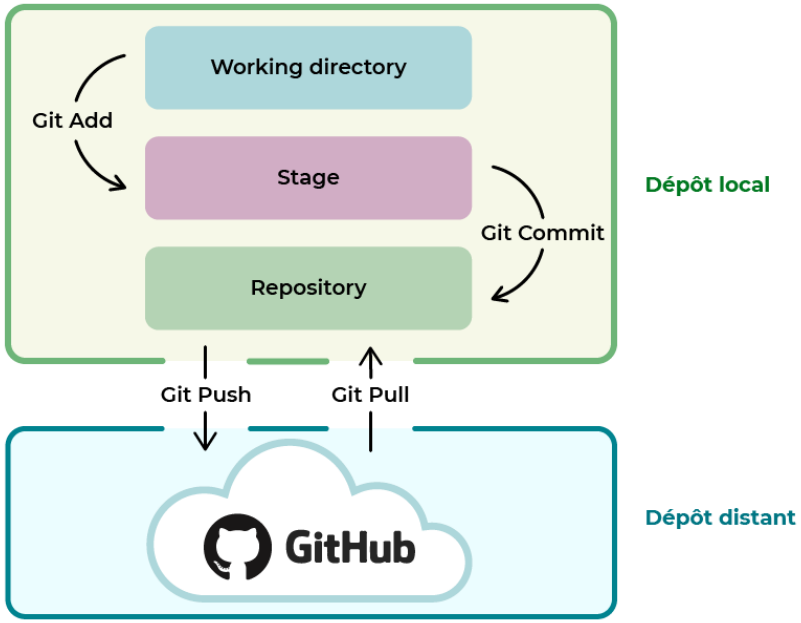
Créez une nouvelle version de votre projet

Retournez dans le fichier index.html de votre working directory et remplacez la chaîne "Un super titre !" Par "TP Quizz"

Pour créer une nouvelle version du projet avec le fichier HTML mis à jour et l'envoyer sur le dépôt GitHub distant :

Petit exercice, revoyez les lignes de commandes écrites précédement pour rédiger les 3 étapes :

1. Indexez le fichier HTML
2. Faites un commit avec comme message "Modification du titre"
3. Envoyez la nouvelle version sur le dépôt distant



Travailler à partir d'un projet existant

Maintenant que vous savez travailler à partir d'un nouveau projet, voyons comment **accéder à un dépôt distant** et le cloner en local.

Cloner consiste à dupliquer un dépôt distant en local.

Il va falloir tout d'abord récupérer l'URL du dépôt distant, et là, cela se passe sur GitHub . Pour cette section, une personne par groupe créera le projet sur GitHub et les autres le cloneront.

Allez dans GitHub et accédez au dépôt distant de votre collaborateur : **https://github.com/[PSEUDO]/TPQuizz**

Cliquez sur le bouton vert "Code" pour cloner le dépôt et copiez l'url HTTPS.

Retournez sur Git Bash et lancez les commandes suivantes :

- **cd [chemin vers wamp]/www**

// On se place à la racine de notre serveur web

- **git remote add TPQuizz https://github.com/[PSEUDO]/TPQuizz**

// TPQuizz représente le nom court que nous utiliseront pour appeler le dépôt distant
// Nous enregistrons le dépôt distant (remote signifie distant) dans une sorte de "Variable" TPQuizz

Cloner le dépôt en local

Maintenant que **notre dépôt local pointe sur le dépôt distant**, nous allons cloner son contenu et le dupliquer en local. Afin de réaliser le clonage, nous allons utiliser la commande :

- **git clone https://github.com/[PSEUDO]/TPQuizz**

Collaborer avec Git

Désormais, vous travaillez à plusieur sur un même respository Git.

Si l'un d'entre vous fait une modification et l'envoie sur le repository distant (s'il push un commit), il faut que les autres puissent mettre à jour leur working directory (leur code en local).

Pour ce faire, placez vous dans votre working directory sur Git Bash et la commande git à utiliser est très simple :

- **git pull**

Appréhender le système de branches

Le principal atout de Git est **son système de branches**. C'est une des notions les plus importantes de Git.

Les différentes branches correspondent à des copies de votre code principal à un instant T, où vous pourrez tester toutes vos idées les plus folles sans que cela impacte votre code principal.

Sous Git, la branche principale est appelée la branche **main**, ou **master** pour les dépôts créés avant octobre 2020.

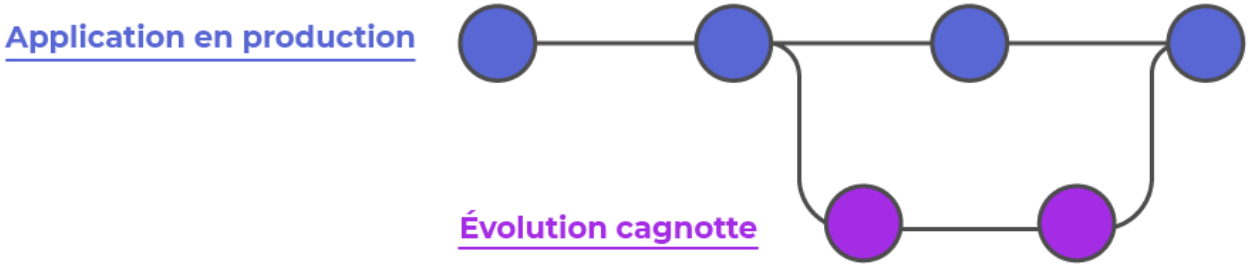
La branche main ou master portera l'intégralité des modifications effectuées. Le but n'est donc pas de réaliser les modifications directement sur cette branche, mais de les réaliser sur d'autres branches, et après divers tests, de les intégrer sur la branche main ou master.

Il faut voir les branches comme autant de dossiers différents. Prenons un exemple concret.

Imaginez que vous ayez réalisé une superbe application bancaire pour M. Robert, et que ce dernier ait une superbe idée de cagnotte à ajouter à son application.

Votre application fonctionne parfaitement, elle est en production, et y toucher serait prendre le risque de tout faire planter. Avec Git et ses fameuses branches, pas de soucis. Vous pouvez créer une branche correspondant à l'évolution "cagnotte", et cela **sans toucher à votre application en cours de production**.

Une fois que toutes vos modifications auront été testées, vous pourrez les envoyer en production sans crainte en les intégrant à la branche main (et dans le pire des cas, revenir en arrière simplement) !



Grâce aux branches, créez l'évolution Cagnotte de votre projet sans toucher à l'application en cours de production

Git va créer une branche virtuelle pour mémoriser tous vos changements, et seulement quand vous le souhaiterez, les ajuster à votre application principale. Il va vérifier s'il n'y a pas de conflits avec d'autres branches et fusionner le tout.

Pour connaître les branches présentes dans notre projet, il faut taper la ligne de commande :

- **git branch**

Dans un premier temps, nous n'auront que : **git branch * main.** C'est normal, l'étoile signifie que c'est la branche sur laquelle vous vous situez actuellement, et que c'est sur elle que vous réalisez des modifications.

Il est fortement conseillé de créer une branche si votre modification est lourde, compliquée, ou si elle risque d'avoir des impacts importants sur votre projet.

Nous avons donc notre branche master, et nous souhaitons maintenant réaliser la fonctionnalité Cagnotte. Pour cela, on tape :

- **git branch cagnotte**

Cette commande va créer la branche Cagnotte en local. Cette dernière ne sera pas dupliquée sur le dépôt distant.

Réaffichez la liste de vos branches avec la commande :

- **git branch**

Vous pouvez maintenant voir votre branche master et votre branche Cagnotte. Comme vous pouvez le voir, la petite étoile est toujours sur la branche master. Nous avons créé la branche Cagnotte, mais nous n'avons pas encore basculé sur celle-ci.
Pour basculer de branche, nous allons utiliser :

- **git checkout cagnotte**

Relancez la commande git branch et regardez où se situe l'étoile.

Réaliser un commit d'une branche, et un push

Après avoir lancé la commande "checkout", Git considère que vous travaillez désormais sur la branche cagnotte. Ainsi, pour commit sur cette branche, il vous suffit simplement de lancer les commandes :

- **git commit -m "Réalisation de la partie cagnotte côté front end"**
- **git push**

Vos modifications sont maintenant enregistrées avec la description "Réalisation de la partie cagnotte côté front end", et ont été publiées sur le repository GitHub.

Si vous souhaitez supprimer une branche, la commande est :

- **git branch -d cagnotte // Si la branche est toute neuve et qu'elle ne contient aucune modification**
- **git branch –D cagnotte // Si la branche contient des modifications par rapport au dépôt. Attention, cette commande supprimera définitivement tous les nouveaux fichiers et modifications non comitées de cette branche**

Fusionner les branches

Après de nombreux commits et pushs, le travail semble terminé. Vous effectuez donc une série de tests pour vérifier le bon fonctionnement de l'évolution “cagnotte” et tout est fonctionnel.

À présent, il faut intégrer l'évolution réalisée dans la branche "cagnotte" à la branche principale "master". Pour cela, vous devez utiliser la commande "git merge".

Cette commande doit s'utiliser à partir de la branche dans laquelle nous voulons apporter les évolutions. Dans notre cas, la commande s’effectuera donc dans la branche master. Pour y retourner, utilisez la commande :

- **git checkout main**

Maintenant que vous êtes dans votre branche principale, vous pouvez fusionner votre branche "cagnotte" à celle-ci grâce à la commande suivante :

- **git merge cagnotte**

Si vous souhaitez supprimer une branche, la commande est :

- **git branch -d cagnotte** // Si la branche est toute neuve et qu'elle ne contient aucune modification
- **git branch –D cagnotte** // Si la branche contient des modifications par rapport au dépôt.
  Attention, cette commande supprimera définitivement tous les nouveaux fichiers et modifications non comitées de cette branche

En résumé

- Une branche est une “copie” d’un projet sur laquelle on opère des modifications de code.
- La branche main ou master est la branche principale d’un projet.
- git checkout permet de basculer d’une branche à une autre.
- git merge permet de fusionner deux branches.

Gérer les erreurs

Revenir à une version précédente

Imaginons maintenant que votre code contienne trop d'erreurs et que vous souhaitiez retourner à une version antérieure de votre projet qui elle fonctionnait correctement.

La solution :

- **git log**

Cette commande va lister tous vos commits par ordre chonologique inversé (du plus récent au plus ancien)

- **git reset --hard ca83a6df**

Cette commande va permettre de retourner à une version antérieure spécifiée (ici le commit portait le code ca83a6df).

Gérer les conflits lors d'un merge

Lors d'un merge entre deux branches, il se peut que la même ligne du même fichier ait été modifiée dans les deux branches. Celà génèrera un conflit qui se présentera de cette manière dans le terminal :

```
git merge cagnotte
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Pour régler ce conflit il va falloir ouvrir le fichier "index.html" qui est indiqué comme conflictuel. Et vous aurez un résultat comme celui-ci:

```
...
<h1>TP Quizz</h1>

<<<<<<< HEAD
<p>Le paragraphe de la branche actuelle (dans laquelle on merge)</p>
=======
<p>Le paragraphe de la branche cagnotte</p>
>>>>>>> cagnotte
...
```

Comment interpréter le contenu de ce fichier ?

- <<<<<<< signifie que les lignes suivantes correspondent aux modifications sur la branche actuelle (HEAD)
- ======= signifie que l'on procède à une séparation. Ce qui se situera en dessous correspond au contenu de la branche conflictuelle
- >>>>>>> signifie que c'est la fin de la 2e section. On voit ensuite le nom de la branche.

Vous allez donc devoir modifier le fichier index.html pour choisir vous même quelle version vous souhaitez garder. Puis vous allez devoir commit les modifications pour terminer le merge.