# [Utilisez Git et GitHub pour vos projets de développement](https://openclassrooms.com/fr/courses/5641721-utilisez-git-et-github-pour-vos-projets-de-developpement)

Table des matières

[Utilisez Git et GitHub pour vos projets de développement 1](#_Toc57651379)

[Tirez le maximum de ce cours ! 6](#_Toc57651380)

[Découvrez la magie du contrôle de versions 7](#_Toc57651381)

[**Qu'est-ce que le contrôle de versions ?** 7](#_Toc57651382)

[**Pourquoi est-ce utile dans le travail d’équipe ?** 9](#_Toc57651383)

[**Comment l’utiliser dans des projets open source ?** 10](#_Toc57651384)

[**Saisissez l'utilité des dépôts distants sur GitHub** 12](#_Toc57651385)

[**Faites la différence entre dépôt local et dépôt distant** 12](#_Toc57651386)

[**Pourquoi utiliser un dépôt ?** 13](#_Toc57651387)

[**Quels sont les outils existants ?** 14](#_Toc57651388)

[**Démarrez votre projet avec GitHub** 15](#_Toc57651389)

[**Créez un compte GitHub** 15](#_Toc57651390)

[**Faites un petit tour de GitHub** 16](#_Toc57651391)

[**Créez votre propre dépôt** 18](#_Toc57651392)

[**Installez Git sur votre ordinateur** 21](#_Toc57651393)

[**Initialisez Git** 22](#_Toc57651394)

[Utilisez les commandes de base de Git ! 26](#_Toc57651395)

[Accédez à un dépôt distant OC 26](#_Toc57651396)

[Clonez le dépôt en local 27](#_Toc57651397)

[Appréhendez le système de branches 28](#_Toc57651398)

[Réalisez un commit 30](#_Toc57651399)

[Réalisez votre premier push 30](#_Toc57651400)

[Démarrez avec Git 32](#_Toc57651401)

[Compétences évaluées 32](#_Toc57651402)

[ Question 1 32](#_Toc57651403)

[ Question 2 32](#_Toc57651404)

[ Question 3 33](#_Toc57651405)

[ Question 4 33](#_Toc57651406)

[ Question 5 33](#_Toc57651407)

[ Question 6 34](#_Toc57651408)

[ Question 7 34](#_Toc57651409)

[ Question 8 35](#_Toc57651410)

[Corrigez vos erreurs sur votre dépôt local 36](#_Toc57651411)

[J’ai créé une branche que je n’aurai pas dû créer 38](#_Toc57651412)

[J’ai modifié la branche principale 40](#_Toc57651413)

[Je souhaite changer le message de mon commit 43](#_Toc57651414)

[J’ai oublié un fichier dans mon dernier commit 44](#_Toc57651415)

[Corrigez vos erreurs sur votre dépôt distant 45](#_Toc57651416)

[Corrigez vos erreurs en local et à distance 45](#_Toc57651417)

[L'accès à distance ne fonctionne pas 45](#_Toc57651418)

[Modifiez vos informations d'identification et supprimez la clé 46](#_Toc57651419)

[**Utilisez Git reset** 49](#_Toc57651420)

[**Les trois types de réinitialisation de Git** 49](#_Toc57651421)

[**Oups, j'ai des conflits !** 50](#_Toc57651422)

[**La machine à remonter le temps ! J’ai ajouté le mauvais fichier au commit** 51](#_Toc57651423)

[**Corrigez un commit raté** 53](#_Toc57651424)

[**Un trou de mémoire ? Git reflog !** 53](#_Toc57651425)

[**Qui s’est amusé dans mon dépôt ? Git blame** 54](#_Toc57651426)

[**Il me faut ce commit ! Vite Git cherry-pick** 54](#_Toc57651427)

[Réparez les erreurs les plus courantes 55](#_Toc57651428)

[Compétences évaluées 55](#_Toc57651429)

[ Question 1 55](#_Toc57651430)

[ Question 2 55](#_Toc57651431)

[ Question 3 56](#_Toc57651432)

[ Question 4 56](#_Toc57651433)

[ Question 5 56](#_Toc57651434)

[ Question 6 57](#_Toc57651435)

[ Question 7 58](#_Toc57651436)

[Identifiez la structure de fichier de Git 60](#_Toc57651437)

[Découvrez l’arbre Git et sa structure 60](#_Toc57651438)

[Représentation cryptographique d'un commit 61](#_Toc57651439)

[Comment fonctionne la fusion sous Git ? 61](#_Toc57651440)

[Les options Git pull/Git push 62](#_Toc57651441)

[À quoi sert Git fetch ? 63](#_Toc57651442)

[**Modifiez vos branches avec Rebase** 65](#_Toc57651443)

[**Comment fonctionne Git rebase ?** 65](#_Toc57651444)

[**Réécrivez l’historique avec l’interactif Rebase de Git** 65](#_Toc57651445)

[**Modifiez l'ordre des commits** 66](#_Toc57651446)

[**Modifiez les messages de validation** 66](#_Toc57651447)

[**Utilisez des techniques de nettoyage de branche** 68](#_Toc57651448)

[**Comment utiliser Rebase pour nettoyer votre branche ?** 68](#_Toc57651449)

[**Supprimez les branches non suivies** 68](#_Toc57651450)

[**Squash ? Débarrassez-vous des commits à tout va !** 69](#_Toc57651451)

[**Trouvez l’origine d’un bug avec Git bisect** 69](#_Toc57651452)

[**Intégrez les dépôts d’autres personnes dans le vôtre** 71](#_Toc57651453)

[**Définissez les sous-arborescences et les sous-modules** 71](#_Toc57651454)

[Gérez plusieurs versions avec Git 73](#_Toc57651455)

[Compétences évaluées 73](#_Toc57651456)

[ Question 1 73](#_Toc57651457)

[ Question 2 73](#_Toc57651458)

[ Question 3 74](#_Toc57651459)

[ Question 4 74](#_Toc57651460)

[ Question 5 74](#_Toc57651461)

[ Question 6 75](#_Toc57651462)

[ Question 7 75](#_Toc57651463)

[Travaillez en équipe en utilisant un workflow 77](#_Toc57651464)

[Qu'est-ce que le flux de travail ou workflow? 77](#_Toc57651465)

[Appréhendez le fonctionnement de GitFlow 78](#_Toc57651466)

[Que signifient tous les types de branches et quelles sont leurs utilités ? 78](#_Toc57651467)

[**Améliorez Git avec des outils supplémentaires** 80](#_Toc57651468)

[**Exploitez au mieux la marketplace de GitHub** 80](#_Toc57651469)

[**Quelques outils et leurs fonctionnalités** 80](#_Toc57651470)

[**Comparez deux fichiers avec une interface graphique** 80](#_Toc57651471)

[Utilisez le GitLab intégration continue (IC) 82](#_Toc57651472)

[Qu'est-ce que GitLab ? 82](#_Toc57651473)

[Où l'IC se situe-t-elle dans le flux de travail et que fait-il ? 82](#_Toc57651474)

[Quels sont les autres outils de CI ? 83](#_Toc57651475)

[Fonctionnement de GitLab CI 84](#_Toc57651476)

[Gérez les demandes de pull 87](#_Toc57651477)

[Pourquoi est-il important de gérer les demandes de pull régulièrement pour maintenir un projet actif ? 87](#_Toc57651478)

[Étiquetez votre projet avec des badges 87](#_Toc57651479)

[Gérez les corrections de bugs avec la console GitHub 88](#_Toc57651480)

[Quand devriez-vous fusionner dans la branche principale ? 90](#_Toc57651481)

[Récapitulez ce que vous avez appris 91](#_Toc57651482)

[Gérez un travail d’équipe 92](#_Toc57651483)

[Compétences évaluées 92](#_Toc57651484)

[ Question 1 92](#_Toc57651485)

[ Question 2 93](#_Toc57651486)

[ Question 3 93](#_Toc57651487)

[ Question 4 93](#_Toc57651488)

[ Question 5 94](#_Toc57651489)

[ Question 6 95](#_Toc57651490)

[ Question 7 95](#_Toc57651491)

[ Question 8 96](#_Toc57651492)

Vous êtes un développeur débutant et vous souhaitez vous lancer sur **des projets plus ambitieux**?

Vous êtes au bon endroit !

Les projets informatiques sont parfois chaotiques, alors comment garder une trace de tout son travail ?  
Comment revenir à une version précédente ?  
Et comment réparer ses erreurs ?

Dans ce cours, je vous donnerai toutes les réponses pour gérer et déployer tous vos projets informatiques avec **Git**. :D  
  
Git permet de suivre les modifications et organiser votre projet. C’est un outil essentiel, que vous travailliez **seul**, en **équipe**, ou même sur **un projet en open source**!

Dans ce cours, vous installerez et configurerez **Git** et son ami **GitHub**. Vous découvrirez les **commandes de base** avant d'apprendre à corriger vos erreurs simplement et efficacement. Vous découvrirez la **structure de l'arbre Git** et comment garder vos **branches** propres. Enfin, vous apprendrez à utiliser **GitFlow** et à gérer les demandes de **pull**.

À la fin de ce cours, vous serez capable de :

* utiliser les commandes de base de Git ;
* corriger les erreurs courantes sur GitHub ;
* gérer plusieurs versions sur GitHub ;
* collaborer grâce à GitHub en utilisant les workflows.

Prérequis : pour suivre ce cours, vous devez savoir écrire une ligne de commande. Si ce n'est pas encore le cas, je vous invite à suivre le cours "[**Apprenez à utiliser la ligne de commande dans un terminal**](https://openclassrooms.com/fr/courses/6173491-apprenez-a-utiliser-la-ligne-de-commande-dans-un-terminal)" au préalable.

## Tirez le maximum de ce cours !

Bonjour et bienvenue dans ce cours !

Je m'appelle **Tiffany Lestroubac,**je suis développeuse sur les technologies Microsoft depuis maintenant 6 ans et mentor chez OpenClassrooms. J'utilise Git au quotidien !

**Mila Paul** est l'auteure du cours anglais sur Git et GitHub. Elle a contribué à ce cours, notamment pour le contenu des vidéos.

Avant de démarrer, je vous invite à regarder la vidéo ci-dessus pour comprendre le fonctionnement de ce cours et en tirer le maximum. :D

*Alors, prêt ? C'est parti !*

## Découvrez la magie du contrôle de versions

Dans ce cours, je vais vous présenter la clé à molette de tout bon développeur : **le gestionnaire de code source**.

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

Ma modif n’a pas marché et j’ai oublié de sauvegarder une copie de mon code avant de le bidouiller...

Qui a touché à mon fichier ? Il présente un bug maintenant !

À quoi servent ces nouveaux fichiers ?

Ne touche surtout pas à ce fichier, je suis en train de le modifier.

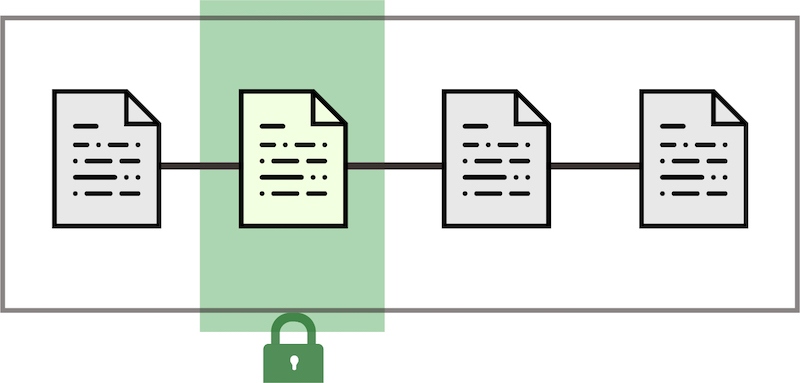
Et bien d’autres problèmes, que tout développeur sera susceptible de rencontrer sur son parcours. Si maintenant, je vous disais que vous ne rencontrerez plus jamais ces problèmes grâce au contrôle de versions…. C’est parti ! :soleil:

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

Le nom n’a rien de magique, je vous l’accorde, et pourtant ce petit outil vous deviendra vite **indispensable** ! Si, si !

Mais déjà, qu’est-ce qu’un contrôleur de versions ?

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



Contrôleur de versions

**Si vous êtes seul** à travailler sur votre projet, le contrôle de versions vous sera d’une grande aide. Il vous permettra de garder l’historique des modifications de tous vos fichiers. Je vous conseille donc fortement de l’utiliser ! Le contrôle de versions permet de garder en mémoire chaque modification de chaque fichier qui a eu lieu, pourquoi elle a eu lieu et par qui ! Plus besoin de mener son enquête pour connaître les dernières modifications. Le contrôle de versions permet d’assembler les modifications de deux personnes travaillant simultanément sur un même fichier, afin d’éviter d’écraser le travail des autres.

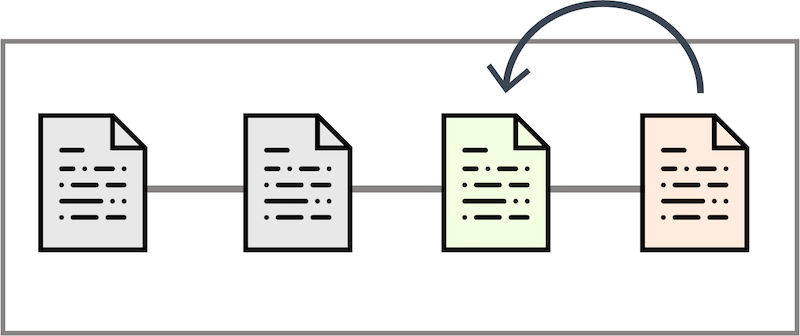
 Ces outils ont donc trois grandes fonctionnalités :

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

Prenons un exemple simple, imaginez que vous remportiez un projet de création d’un formulaire de recrutement. Youpiii ! :D

Vous avez réussi à créer le formulaire et votre client est super content ! Voyant que son projet a bien avancé, il a l’idée d’ajouter une fonctionnalité d’envoi de mails. Il vous demande donc de le faire au plus vite. Ni une ni deux, vous vous lancez dans le code.

Malheureusement, la modification que vous avez faite a entraîné des **régressions** et le formulaire ne fonctionne plus. Votre client n’est pas content et souhaite que vous fassiez un retour arrière sur votre code pour que le formulaire fonctionne de nouveau. Avec un contrôle de versions, il n’y a aucun problème. :) Vous remettez la version 1 en production et votre client est satisfait. ;)



Retour à une ancienne version

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

L’utilisation de ces logiciels est donc indispensable pour un développeur digne de ce nom !

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

Admettons que Henry modifie le fichier X, mais que vous aussi ayez modifié ce fichier ; pas de panique, avec le contrôle de versions, vous pourrez assembler vos modifications. Ce n’est pas magique ?

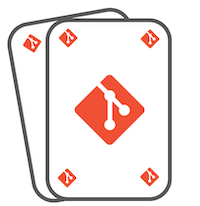
Dans ce cours, nous allons apprendre à utiliser le contrôleur de versions Git.

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

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

La maîtrise de Git est très souvent demandée lors d’un recrutement, c’est pourquoi il est essentiel de le maîtriser, mais pas de panique ! :)

À la fin de ce cours, vous serez des as de Git (ahahaha vous avez compris ? As de pique/As de Git... OK, je l’admets, ce n’était pas drôle... ). :honte:



As de Git

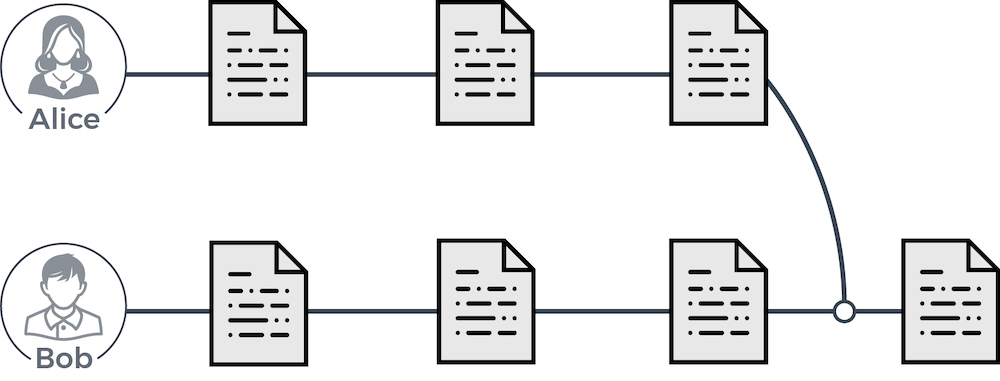
**Pourquoi est-ce utile dans le travail d’équipe ?**

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

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

"En local" signifie sur votre machine, par opposition à "en ligne".

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



Travail mutualisé d'Alice et Bob

**Comment l’utiliser dans des projets open source ?**

Un des aspects passionnants lorsque vous faites du développement, c'est que vous pouvez apporter votre pierre à plein d'édifices en contribuant à des projets open source.

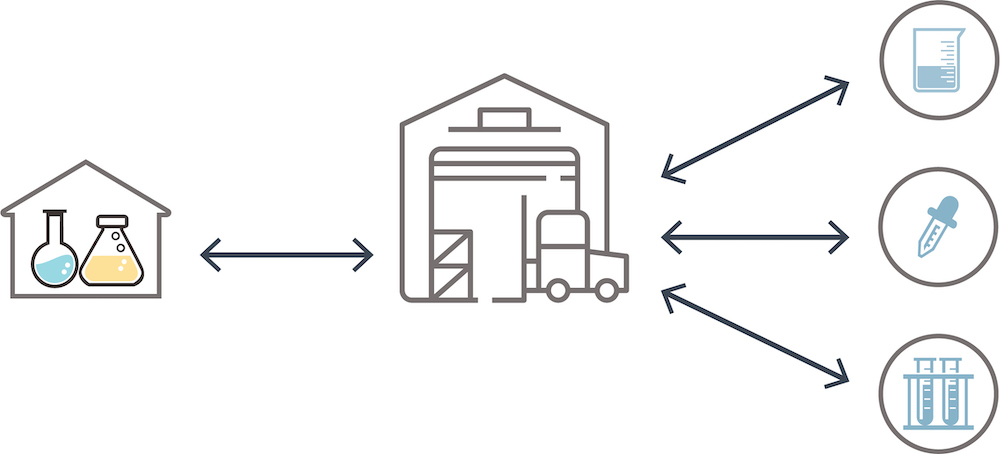
*Open source* signifie que le code source d'un logiciel est public et accessible. Le logiciel en question peut alors être modifié et diffusé par n'importe quel individu.

Travailler sur un projet open source est passionnant et permet de développer rapidement ses compétences. Que ce soient le codage, la conception de l’interface utilisateur, la conception graphique, la rédaction ou l’organisation, si vous cherchez de la pratique, il y a une tâche pour vous sur un projet open source.

Git ou GitHub ? Quelle est la différence ?

Git est l'outil qui nous permet de créer un dépôt local et de gérer les versions de nos fichiers, alors que GitHub est un service en ligne qui va héberger notre dépôt, qui sera du coup distant (puisqu'il ne sera pas sur notre machine).

Prenons un petit exemple. Admettons que nous fassions une préparation pour du parfum. Chez nous, nous allons créer la base du parfum en mélangeant divers ingrédients. Ensuite, nous allons envoyer notre base de parfum à un entrepôt où ils pourront soit le distribuer tel quel, soit le modifier en y ajoutant des ingrédients, des arômes, etc. Eh bien, c'est la même chose que Git et GitHub. Git est la préparation que vous avez réalisée chez vous, et GitHub est l'entrepôt où il peut être modifié par les autres ou distribué. :)



Git et GitHub

Prenons, par exemple, un projet réalisé sous GitHub. La plupart du temps, il faut suivre les étapes suivantes pour collaborer sur un projet open source :

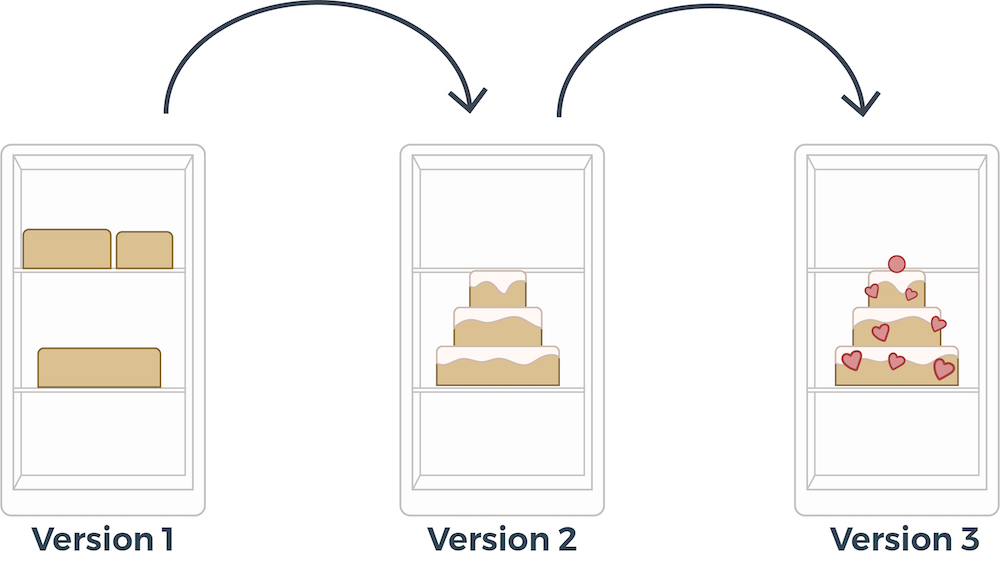
* le premier réflexe est de regarder la documentation. Il y a souvent tout un tas d’informations sur la manière de collaborer au mieux au projet ;
* dans un second temps, vous devrez rapatrier le dépôt distant sur votre dépôt local. Sur votre dépôt local, vous pourrez réaliser vos modifications ;
* une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

*Passons au chapitre suivant pour en savoir plus sur les dépôts distants !*

**Saisissez l'utilité des dépôts distants sur GitHub**

**Faites la différence entre dépôt local et dépôt distant**

Prenons l'image de la réalisation d'un gâteau. On va dire que le réfrigérateur est le dépôt local. C'est l'endroit où l'on va stocker nos préparations au fur et à mesure. Dans un premier temps, on réalise la pâte, on la stocke au réfrigérateur, puis on réalise la crème, on l'assemble avec la pâte et on stocke l'ensemble au réfrigérateur. Finalement, on réalise la décoration du gâteau, on finalise notre gâteau en y ajoutant les décorations  et on le remet au réfrigérateur.



Fonctionnement du dépôt local : rangez votre frigo !

Voilà le fonctionnement de notre **dépôt local** ! On réalise une version, que l'on va petit à petit venir améliorer en stockant toutes ces versions.

On peut aussi prendre l'image d'un livre d'école ! Vous avez sûrement déjà remarqué que sur vos livres, il y a souvent écrit *première édition* ou *cinquième édition*. L'éditeur a réalisé une version et petit à petit, il a corrigé l'orthographe, ou modifié le contenu. Eh bien, les versions dans Git fonctionnent de la même manière. On va réaliser une première version, que l'on va venir améliorer au fil du temps.

Le**dépôt distant**est un peu différent. Il permet de stocker certaines versions qu'on lui aura envoyées, afin de garder un historique délocalisé. Ben oui, imaginez que votre PC rende l’âme demain, vous aurez toujours vos super programmes sur GitHub. Mais en plus de les stocker, vous pouvez aussi les rendre publics, et chacun pourra alors venir y ajouter ses évolutions.

Afin que vous puissiez collaborer sur des projets, il est nécessaire de disposer de dépôts distants. Le dépôt distant est un**historique** de votre projet hébergé sur Internet ou sur un réseau. Vous pouvez avoir plusieurs dépôts distants avec des droits différents (lecture seule, écriture, etc.).

Mais avant tout, avez-vous bien compris ce qu’était réellement un dépôt ?

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

C’est bon, ça devient un peu plus clair ? Super !

Le dépôt distant est un type de dépôt qui devient réellement important (voire **indispensable**) lorsque l’on travaille à plusieurs sur le même projet, puisqu’il permet de centraliser le travail de chaque développeur. Je vous recommande donc fortement d’utiliser GitHub pour vos dépôts distants. :)Plus aucun souci de**pertes de données** ou de travail. Sur GitHub, vous pouvez bien entendu créer des dépôts distants publics, mais aussi privés.

Sur un dépôt public, les personnes pourront collaborer à votre projet alors que sur un dépôt privé, vous seul aurez accès à votre travail !

Je vous rappelle que l’intérêt de Git est le**suivi des modifications**, mais aussi la sauvegarde de vos projets. C’est pourquoi je vous conseille de toujours commencer par copier vos sources sur un dépôt distant, si possible situé à l’extérieur de vos locaux ! Oui oui, exit la paranoïa des vilains méchants du web qui pirateraient votre dépôt, vous avez bien plus de chances de perdre vos données chez vous que sur GitHub. C’est aussi sur le dépôt distant que toutes les modifications de tous les collaborateurs seront fusionnées. Vous allez me dire, mais alors pourquoi des dépôts distants, avec GitHub on a tout ce dont on a besoin, on ne va pas s'embêter à créer une copie locale. Eh bien si ! La majeure partie de votre travail se fera sur votre dépôt local qui est un clone de votre dépôt distant. C’est sur votre dépôt local que vous ferez toutes vos modifications de codes, vos créations de branches (il y en a aussi sur le dépôt distant), et vos **commits** ; et seulement quand vos modifications seront prêtes à être partagées à l’équipe, vous les pousserez sur le dépôt distant.

**Pourquoi utiliser un dépôt ?**

Vous êtes convaincu de la nécessité d’utilisation de dépôts ?

Pas encore ?

Avec les dépôts, vous vous éviterez de longues heures de perte de travail, de recherche des dernières modifications et de galère de travail d’équipe !

Les dépôts sont utiles si :

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

Je vous l’ai dit, qu’il deviendrait vite votre meilleur ami. :-°

**Quels sont les outils existants ?**

Il existe plusieurs outils intéressants (GitHub, GitLab, Bitbucket, SourceForge), et nous allons donc voir les principaux avantages et inconvénients de chacun.

Commençons par [GitHub](https://github.com/) (mon préféré, mais chuttt !! ). **GitHub** est un outil de communication et de collaboration entre plusieurs développeurs (ou tout autre personne qui écrit du texte). C’est une interface web créée pour faciliter l’interaction avec Git.

Bon, c’est vrai, tous ces outils font cela ! :euh:

L’avantage de GitHub, c’est que depuis quelques années GitHub est devenu le **book/portfolio des développeurs**! Dans beaucoup de processus de recrutement, on vous demandera maintenant votre lien GitHub ! Si ça, c’est pas un argument de taille !  Il permet de mettre en avant la qualité de son code, et ainsi montrer ses capacités et sa plus-value lorsque l’on recherche un emploi. GitHub est considéré comme un véritable réseau social et permet de contribuer à des projets open source. GitHub fonctionne par abonnement, mais pas de panique, il y a un abonnement gratuit qui est déjà très bien. :)

Concernant [GitLab](https://about.gitlab.com/), il est la principale alternative à GitHub depuis le rachat de GitHub par Microsoft ! Les anti-Microsoft ont même lancé le hashtag *#MovingToGitLab* ! GitLab fonctionne avec une version gratuite à installer sur son propre serveur ou une version cloud payante.

[BitBucket](https://fr.atlassian.com/software/bitbucket) est la version de Atlassian. Payante, elle plaira néanmoins aux habitués de la gestion de projet sous Atlassian. BitBucket conviendra aussi bien aux étudiants ou petites teams qu’aux grands groupes.

Parlons enfin de SourceForge, le petit dinosaure dans le domaine. SourceForge a été créé 10 ans avant les autres, afin de gérer à la base des projets open source. SourceForge intègre un outil de suivi des bugs et un répertoire de code intégré. Il n’est plus très populaire depuis ces dernières années.

Vous avez fait votre choix ? Nous étudierons dans ce cours la solution GitHub qui est la plus plébiscitée par les développeurs.

*Maintenant, mettons les pieds dans le plat !*:soleil:

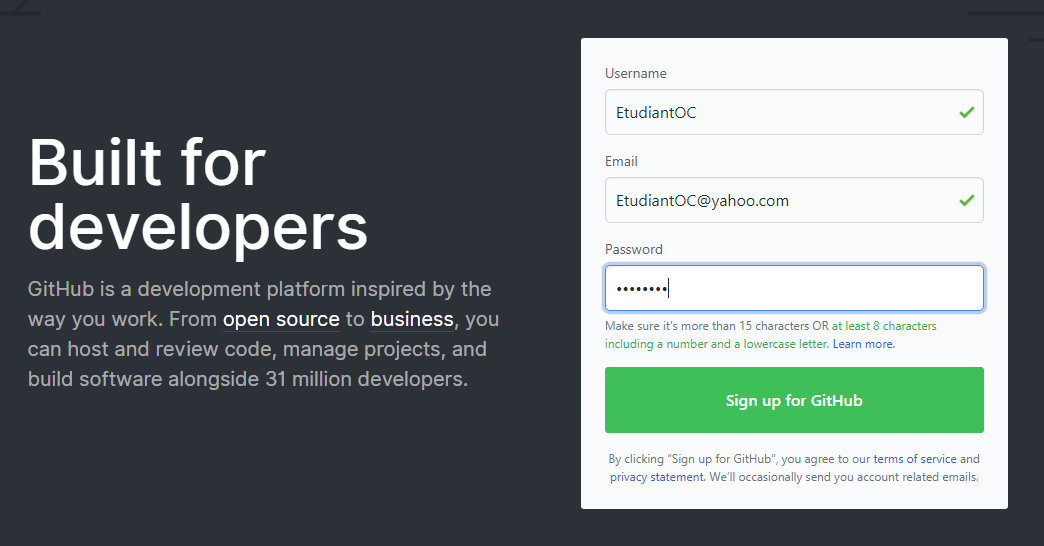


**Démarrez votre projet avec GitHub**

**Créez un compte GitHub**

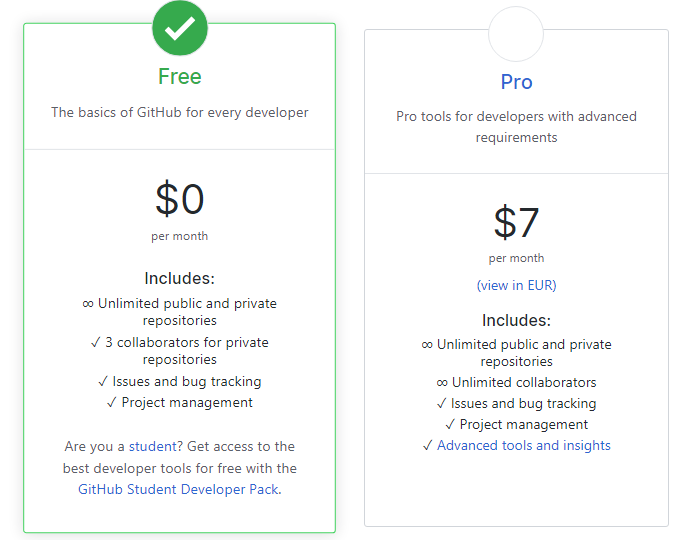
Comme nous l’avons vu dans le chapitre précédent, GitHub est un service en ligne permettant d’héberger ses dépôts distants.

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 email et un mot de passe.



Inscription GitHub

Une fois ces informations remplies, vous devrez choisir votre **abonnement** (gratuit ou pro). La principale différence entre les deux offres est que la première est destinée aux particuliers, ou aux équipes de moins de 3 collaborateurs, alors que la seconde offre est destinée aux plus grandes équipes. Sachez néanmoins que si vous faites des projets open source, il n’y a aucune limitation sur le nombre de collaborateurs. ;)



Plan GitHub

[À vous de créer votre compte GitHub !](https://github.com/)

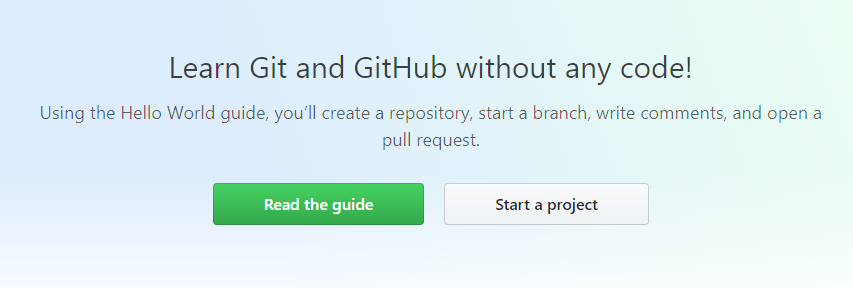
**Faites un petit tour de GitHub**

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

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 référentiels et pages d'équipe, rester à jour sur les activités récentes des organisations et des référentiels auxquels vous êtes abonné et explorer les référentiels recommandés.

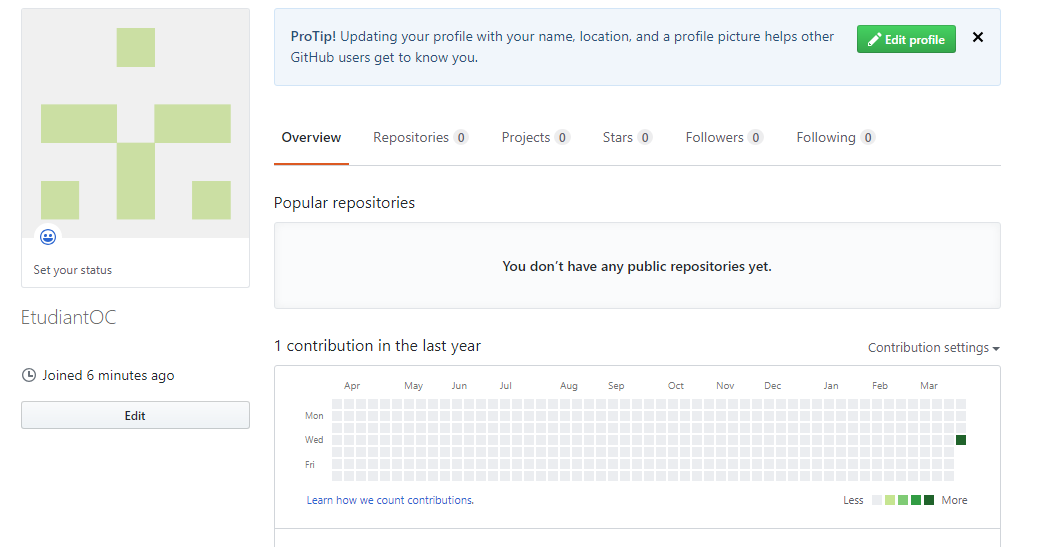
L’interface Repositories est l’emplacement où vous pourrez créer et retrouver vos dépôts existants.

Pour créer un projet, il suffit de cliquer sur “Start a project”.



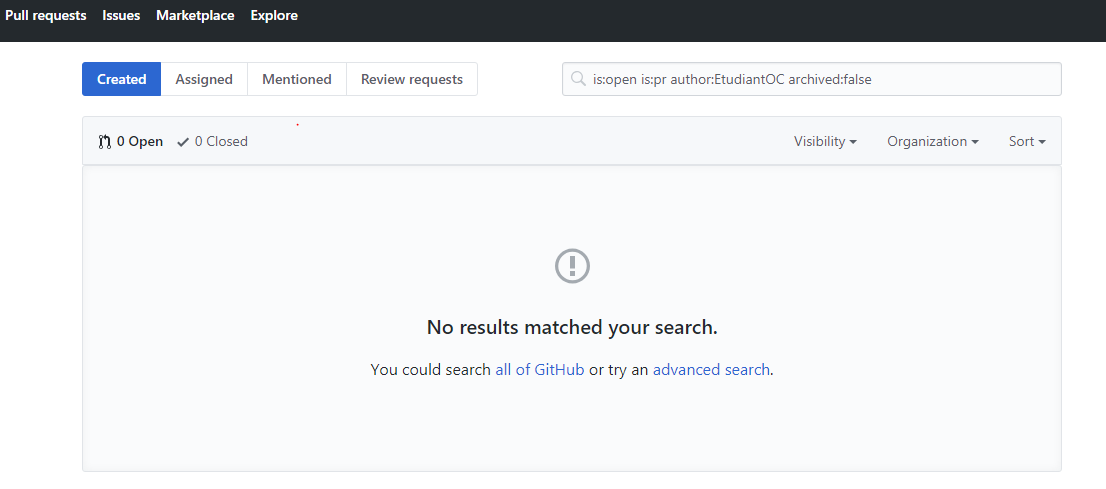
Cliquez sur "Start a project"

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



Profil

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



Onglet "Pull requests"

Dans la section "**Activité récente**" de votre fil d’actualité, vous pouvez rapidement rechercher et suivre les problèmes récemment mis à jour, et extraire les demandes sur lesquelles vous travaillez. Sous "Activité récente", vous pouvez prévisualiser jusqu'à 12 mises à jour récentes effectuées au cours des deux dernières semaines.

Une activité est récente lorsque :

* vous avez ouvert un problème ou une demande d'extraction ;
* quelqu'un a commenté un problème ou tiré une demande que vous avez ouverte ;
* votre problème ou demande d'extraction a été rouvert ;
* votre avis a été demandé sur une demande de tirage ;
* vous avez été affecté à un problème ou à une demande d'extraction ;
* vous avez référencé un problème ou une requête d'extraction via un commit ;
* vous avez commenté un problème ou une demande d'extraction.

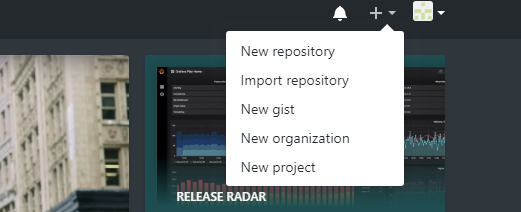
Un des derniers points importants sur GitHub est la fonctionnalité **Explore**.

Via Explore, vous pourrez trouver de nouveaux projets open source intéressants sur lesquels travailler, en parcourant les projets recommandés, en vous connectant à la communauté GitHub et en recherchant des référentiels par sujet ou par libellé.

**Créez votre propre dépôt**

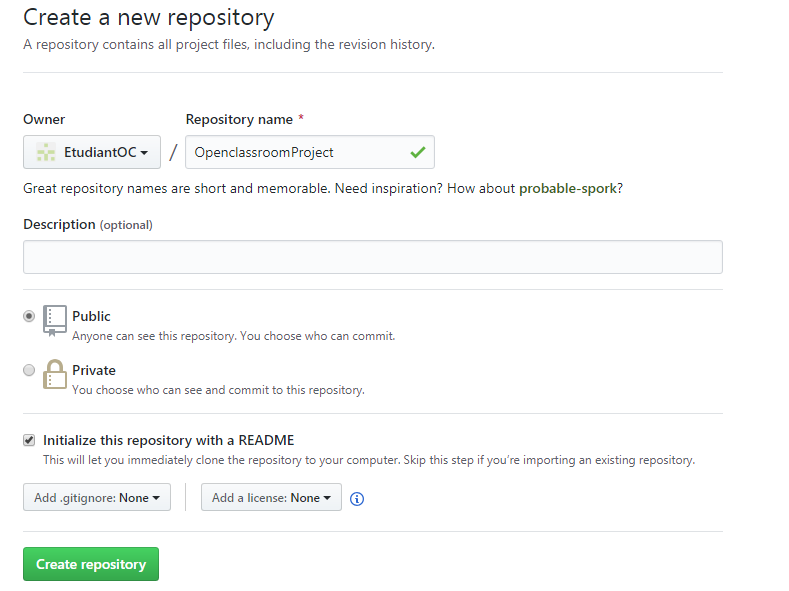
Pour mettre votre projet sur GitHub, vous devez créer un référentiel dans lequel il pourra être installé.

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



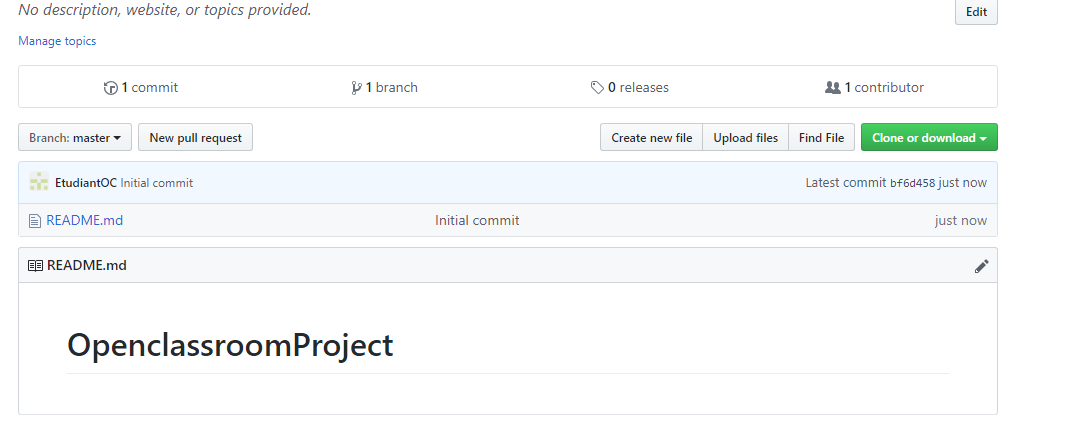
Ajoutez un référentiel

Choisissez un nom simple pour votre dépôt, choisissez si vous souhaitez créer un dépôt public ou privé, initialiser un readme et un gitignore.



Créez un dépôt

Cliquez ensuite sur “Créer un dépôt”.



Votre premier dépôt Git

*Félicitations ! Vous avez créé votre premier dépôt GitHub ! La prochaine étape est d'installer Git sur votre ordinateur. :)*

**Installez Git sur votre ordinateur**

Tadammm ! L’heure est venue d’installer **votre futur meilleur ami**. :)

Je vous invite à suivre ce screencast pour installer Git sur votre ordinateur. Les étapes sont également détaillées sous la vidéo.

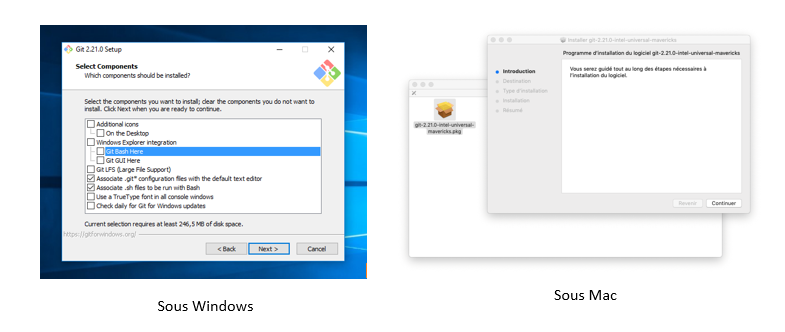
Afin d'installer Git, il faut dans un premier temps le **télécharger**. Vous pourrez trouver [le lien de téléchargement ici](https://git-scm.com/downloads). Choisissez la **version** qui correspond à votre installation et téléchargez-la.

Exécutez le fichier que vous venez de télécharger :



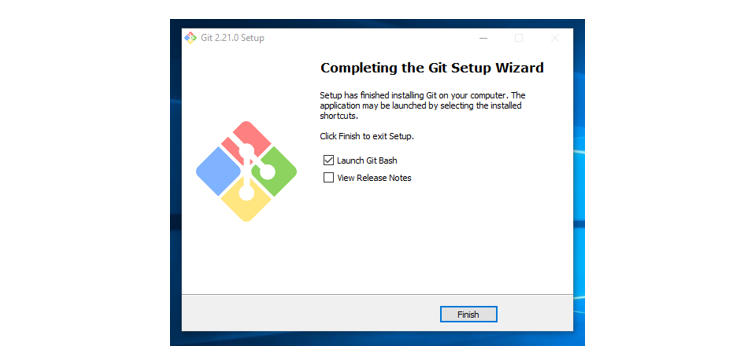
Exécutez le fichier

Appuyez sur Suivant à chaque fenêtre puis sur **Installer**. Lors de l’installation, laissez toutes les options par défaut, elles conviennent bien.



Lancez l'installation

Cochez ensuite Launch Git Bash.



Lancez Git Bash

Git Bash se lance.



Fenêtre Git Bash

**Initialisez Git**

La première chose à faire est de configurer son identité. Pour cela, nous allons entrer dans le monde des lignes de commande !

Nous allons commencer par renseigner votre nom et votre adresse e-mail. C'est une information importante car toutes les validations dans Git utilisent cette information et elle est indélébile dans toutes les validations que vous pourrez réaliser :

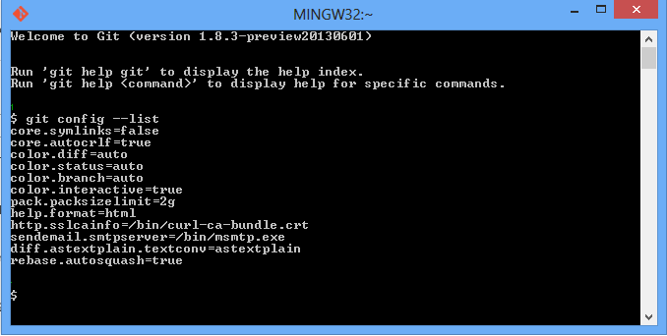
$ git config --global user.name "John Doe"

$ 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 par contre, pour **un projet spécifique**, changer votre nom d’utilisateur, vous devrez repasser cette ligne mais sans le --global.

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



Il est recommandé 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

Par défaut, Git utilisera Vim comme éditeur et Vimdiff comme outil de merge. Vous pouvez les modifier en utilisant :

$ git config --global core.editor notepad++

$ git config --global merge.tool vimdiff

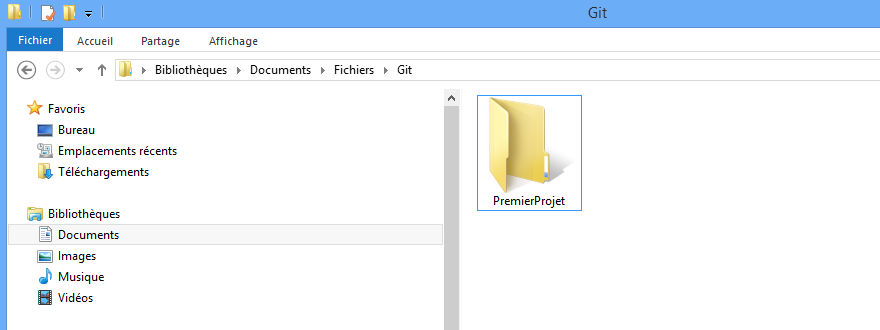
Maintenant que nous avons paramétré les paramètres de base, nous allons **créer notre fameux dépôt local**. Pour ce faire, deux solutions possibles :

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

Dans un logiciel de gestion de versions comme Git, un dépôt représente une copie du projet.

Chaque ordinateur d’un développeur qui travaille sur le projet **possède donc une copie du dépôt**.  
Dans chaque dépôt, on trouve les fichiers du projet ainsi que leur historique.

Nous allons maintenant voir comment créer un dépôt vide. On va créer, dans un premier temps, un dossier sur notre disque (avec le nom de notre projet, c’est mieux ;)).



Créez un dossier

Accédez à votre dossier, et lancez la ligne ci-dessous dans Git Bash en ciblant ce dossier :

johndoe ~

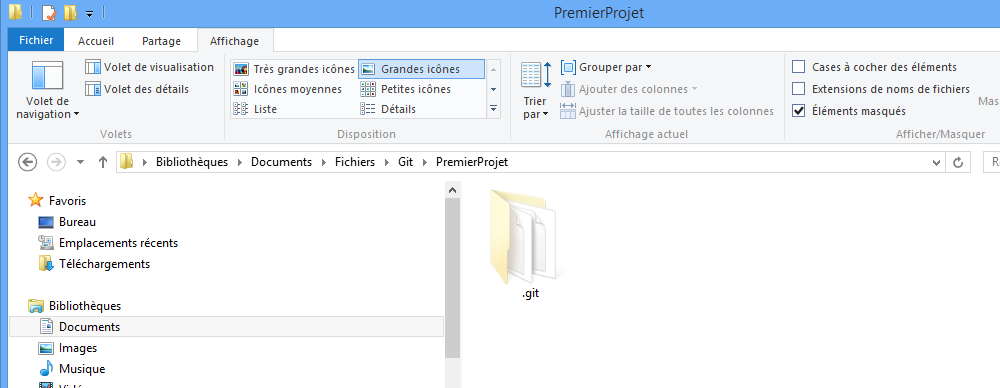
$ cd Documents/Fichiers/Git/PremierProjet

johndoe ~/Documents/Fichiers/Git/PremierProjet

$ git init

Initialized empty Git repository in c:/users/JohnDoe/Documents/Fichiers/Git/PremierProjet/

Votre dossier n’a rien de nouveau, mais c’est normal. :) Vous avez pourtant bien initialisé votre dépôt Git. **Un dossier caché .git a été créé** ! Vous pouvez l'afficher en allant dans Affichage => Éléments masqués.



Affichez le dossier Git caché

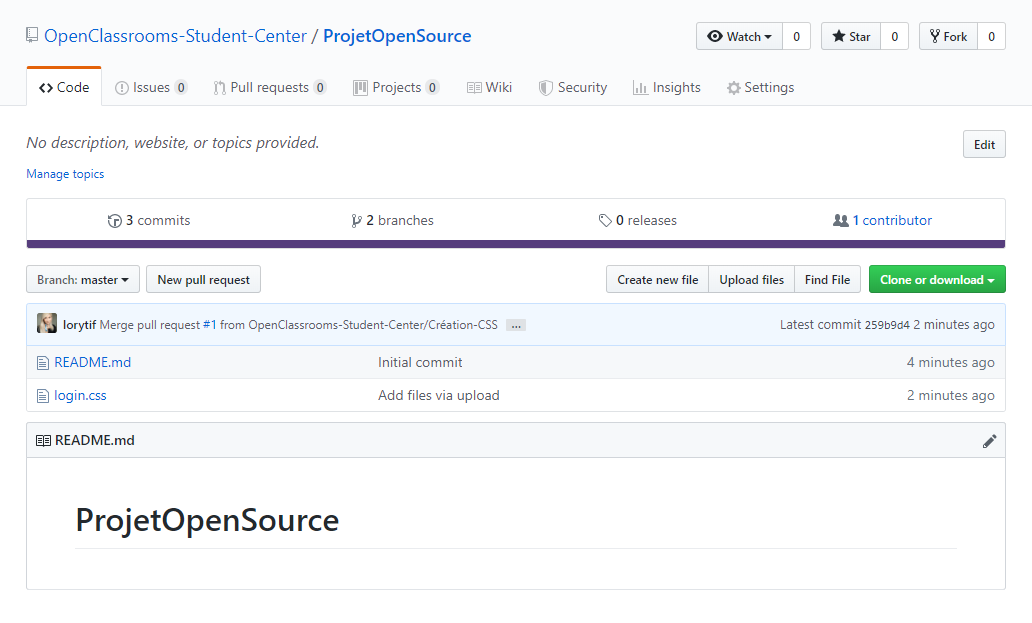
*Nous verrons plus tard comment l’utiliser. Dans le prochain chapitre, vous découvrirez comment contribuer à la communauté grâce aux projets open source !*

## Utilisez les commandes de base de Git !

### Accédez à un dépôt distant OC

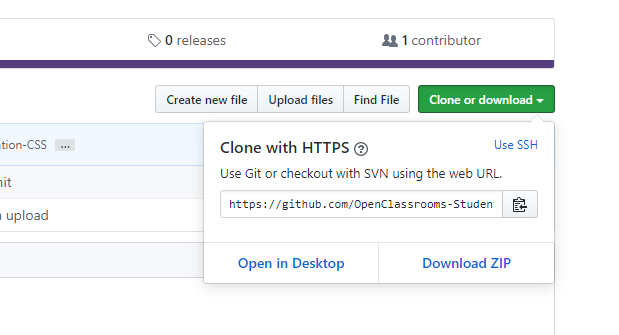
Maintenant que vous êtes un pro du dépôt local et de GitHub, :-° nous allons voir comment **accéder à un dépôt distant** et le cloner en local. Il va falloir tout d’abord récupérer l’URL du dépôt distant, et là cela se passe sur GitHub ! :)

Allez dans GitHub et accédez au dépôt distant de notre projet open source [ici](https://github.com/OpenClassrooms-Student-Center/ProjetOpenSource).



Accéder à un dépôt open source

Cliquez sur le bouton Clone or download.



Clonez le dépôt

Et tadam, vous avez la fameuse URL dont nous allons avoir besoin. Copiez-la dans le presse-papier.

Retournez sur Git Bash, et tapez la commande suivante :

git remote add OC https://github.com/OpenClassrooms-Student-Center/ProjetOpenSource.git

OC représente le nom court que vous utiliserez ensuite pour appeler votre dépôt. Appelez-le comme bon vous semble, mais un nom court et simple est toujours plus facile.

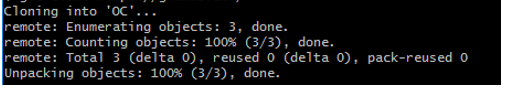
Cette ligne ne permet pas de cloner le dépôt, mais permet de dire au dépôt que l’on pointe vers le dépôt distant.

### Clonez 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/OpenClassrooms-Student-Center/ProjetOpenSource.git

Cela devrait afficher ces petites lignes de commandes !



Résultat de la commande de clonage

C’est bon signe, la magie du clonage a fonctionné ! :magicien:

Vous devriez maintenant avoir un nouveau répertoire (ayant le nom du projet) et dans ce dossier, tous vos fichiers. Le clonage s’effectue très rapidement, puisque les fichiers vont être compressés avant le transfert. On va donc les recevoir à la vitesse de l’éclair ! Enfin... rapidement, quoi. :)

Vous pouvez maintenant vous mettre à vos développements, et cela sans faire de bêtises ! :p

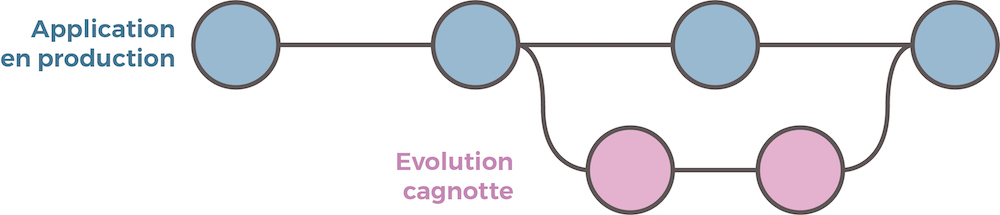
### Appréhendez le système de branches

Le principal atout de Git est **son système de branches** (pas d’arbre, bien entendu). :D C’est sur ces branches que repose toute la magie 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 master**. C’est celle-ci, où au final, vous aurez à la fin toutes vos modifications. Le but est de ne surtout pas réaliser les modifications directement sur cette branche, mais de réaliser les modifications sur d’autres branches, et après tests, les intégrer sur la branche master. Vous avez l’impression que c’est du charabia ? Je vous comprends tout à fait ! :lol:

Nous allons prendre un cas un peu plus concret. Imaginons que vous ayez réalisé une superbe application bancaire pour M. Robert, et que M. Robert ait une superbe idée de cagnotte à ajouter à son application. Panique à bord dans votre tête ! Votre application fonctionne parfaitement, elle est en production, et y toucher serait prendre le risque de faire tout planter. Avec Git et ses fameuses branches, pas de soucis. Vous allez pouvoir créer une branche correspondant à l’évolution cagnotte et cela, sans toucher à votre application en production qui fonctionne parfaitement. Une fois que toutes vos modifications auront été testées, vous pourrez les envoyer en production sans crainte (et dans le pire des cas, revenir en arrière simplement) !



Branche principale et évolution Cagnotte

Avec Git, aucun problème de fusion. :soleil: Vous n’avez pas besoin de connaître tous les bouts de code que vous avez modifiés. Il va, comme un grand, fusionner votre évolution avec votre code principal.

On dit qu’un bon développeur est toujours fainéant ! :-° En fait, un bon développeur trouvera toujours une technique simple pour faire le travail à sa place. Eh bien, Git est l’outil idéal dans ce cas. Pas besoin de s’enquiquiner des heures avec une centaine de copier/coller, avec le risque de tout casser, alors qu’il fait la fusion pour nous et sans erreurs.

Il va créer une branche virtuelle, mémoriser tous vos changements, et seulement quand vous le souhaitez, les ajouter à votre application principale. Il va vérifier s'il n’y a pas de conflits avec d’autres fusions, et le tour est joué !

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

git branch

Dans un premier temps, vous n’aurez que :

git branch

\* master

Et c’est normal. L’étoile signifie que c’est la branche sur laquelle vous vous situez et que c’est sur celle-ci qu'actuellement vous réalisez vos modifications.

Je vous conseille fortement de créer une branche si votre modification va être longue, qu’elle peut avoir des impacts, qu’elle n’est pas simple ou que vous ne voyez pas tout de suite comment faire la modification.

Entre nous, il est souvent préférable de créer une branche pour une modification. La création prend 30 secondes et vous économise beaucoup de temps de galère si vous faites des bêtises sur votre branche master.

Revenons au projet de M. Robert. 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 (elle ne va pas être dupliquée sur le dépôt distant).

Vous pouvez de nouveau taper la ligne de commande :

git branch

\* master

cagnotte

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

Vous aurez donc :

git branch

master

\* cagnotte

La branche va fonctionner comme un dossier virtuel. Avec **Git checkout**, on va être téléporté dans le dossier virtuel Cagnotte. On reste dans le dossier OC physiquement, mais virtuellement nous sommes passés dans un monde parallèle !  Vous pouvez désormais réaliser votre évolution sans toucher à la branche master qui abrite votre code principal qui fonctionne. Vous pouvez rebasculer si besoin à tout moment sur la branche master, sans impacter les modifications de la branche Cagnotte.

### Réalisez un commit

Vous avez réalisé des évolutions sur la branche Cagnotte et il va falloir maintenant demander à Git de les enregistrer.

Un commit est tout simplement un enregistrement de votre travail à un instant T sur la branche courante où vous êtes.

Pour ce faire, nous allons lui envoyer la commande :

git commit -m “Réalisation de la partie cagnotte côté front end”

Vos modifications sont maintenant enregistrées avec comme description “Réalisation de la partie cagnotte côté front end”. La description est très importante pour retrouver le fil de vos commits, et revenir sur un commit en particulier. Ne la négligez pas !

Félicitations ! Vous venez d’être couronné le roi du commit ! :pirate:

### Réalisez votre premier push

La commande **Git push** permet d'envoyer les modifications que l'on a réalisées en local sur le dépôt à distance, alors que la commande **Git pull** permet de récupérer en local le projet distant ! Nous reparlerons de ces deux commandes, plus en détail, un peu plus tard dans le cours !

Cette partie vous a permis de découvrir la gestion de versions pour le développement et de prendre en main Git. Bien sûr, il vous arrivera de faire des erreurs et nous allons voir dans la partie suivante comment les réparer !

Mais avant cela, passons au quiz ! :)

# Démarrez avec Git

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Utiliser les commandes de base de Git

### Question 1

**Que permet un gestionnaire de contrôle de versions ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Conserver un historique

* + 

Revenir en arrière en cas de problème

* + 

Travailler à plusieurs

* + 

Suivre les modifications de chaque fichier

* + 

Mettre en ligne un site web

*Un gestionnaire de contrôle de versions permet de conserver un historique, revenir en arrière en cas de problème, collaborer, et suivre les modifications de chaque fichier.*

### Question 2

**Quelle est la différence entre Git et GitHub ?**

* + 

GitHub est plus performant

* + 

Git crée un dépôt local alors que GitHub crée un dépôt distant

* + 

GitHub permet de faire du développement

*Git crée un dépôt local alors que GitHub crée un dépôt distant.*

### Question 3

**Quelle commande permet d'initialiser son projet dans Git Bash ?**

* + 

Git init

* + 

Git start

* + 

Git go

* + 

Git run

*Git init est la seule commande qui permet d'initialiser son projet Git une fois le dossier créé.*

### Question 4

**Vous avez créé votre dépôt distant sur GitHub, que l'on appellera monDepotDistant, et vous souhaitez maintenant le lier avec votre dépôt  local sur Git.**

**Quelle ligne de commande devez-vous taper ?**

* + 

git remote DEP https://github.com/xxxxxx/monDepotDistant

* + 

git remote DEP add https://github.com/xxxxxx/monDepotDistant

* + 

git remote add DEP https://github.com/xxxxxx/monDepotDistant

* + 

git add DEP https://github.com/xxxxxx/monDepotDistant

*La commande correcte est*git remote add DEP "URL de votre dépôt"*, DEP étant le nom court.*

### Question 5

**Maintenant que votre dépôt distant est lié à votre dépôt local, vous allez devoir dupliquer son contenu en local. Pour ce faire, nous devons utiliser :**

* + 

git clone DEP https://github.com/xxxxxx/monDepotDistant

* + 

git duplicate https://github.com/xxxxxx/monDepotDistant

* + 

git clone https://github.com/xxxxxx/monDepotDistant

* + 

git duplicate DEP

*C'est la commande clone qui permet de dupliquer le contenu distant en local.*

### Question 6

**Nous allons désormais ajouter des branches à notre dépôt. Je souhaite créer trois nouvelles branches : "Branch1", "Branch2" et "Branch3", et basculer sur ma branche "Branch2". Que dois-je faire ?**

* + 

git branch  
git branch  
git branch  
git checkout branch

* + 

git branch Branch1  
git branch Branch2  
git branch Branch3  
git checkout Branch2

* + 

git checkout Branch1  
git checkout Branch2  
git checkout Branch3  
git branch

*Il va falloir créer chacune des branches via la commande Git branch, puis réaliser un Git checkout sur la branche où nous souhaitons basculer.*

### Question 7

**Quelle commande permet de consulter nos branches et quel est le résultat de cette commande ?**

* + 

git branch  
master  
Branch1  
\*Branch2  
Branch3

* + 

git checkout  
master  
Branch1  
\*Branch2  
Branch3

* + 

git branch  
Branch1  
\*Branch2  
Branch3

* + 

git checkout  
Branch1  
\*Branch2  
Branch3

*C'est la commande Git branch qui permet de consulter l'état de notre arbre.*

### Question 8

**Vous devez maintenant réaliser des modifications sur votre branche Branch2, puis les modifier en indiquant le message "Je suis le roi du commit". Comment faites-vous ce commit ?**

* + 

git commit "Je suis le roi du commit"

* + 

git commit -m “Je suis le roi du commit”

* + 

git commit -m -n “Je suis le roi du commit”

* + 

commit

git commit -m*permet de réaliser un commit en indiquant un message de description.*

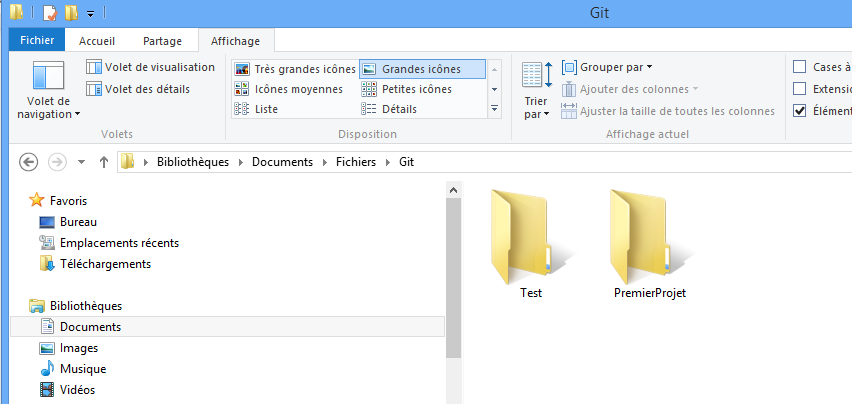
## Corrigez vos erreurs sur votre dépôt local

Quand vous utilisez Git, au début, vous pouvez avoir l’impression que la vie n’est pas un long fleuve tranquille.  :'(

Git est un outil merveilleux, mais on a vite fait de créer une branche alors qu’on ne le souhaitait pas, de modifier la master ou encore d’oublier des fichiers dans ses commits. Mais ne vous inquiétez pas, nous allons voir ensemble que toutes ces petites erreurs ne sont pas difficiles à corriger avec **les bonnes techniques**. Nous allons maintenant créer un petit bac à sable pour nous entraîner !

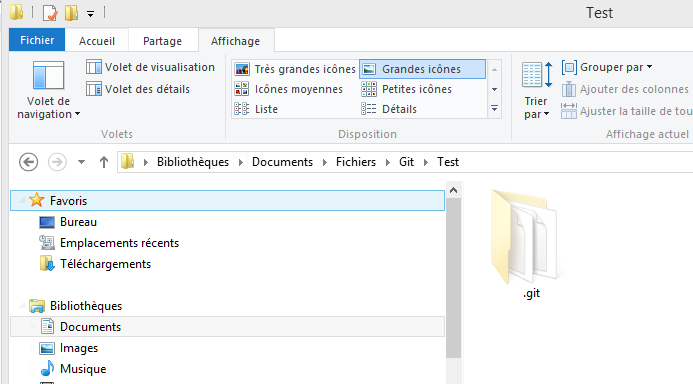
Mettons en œuvre ce que nous avons vu dans la partie précédente.

Créons un dépôt Git que l’on va nommer Test.



Créez un dépôt local "Test"

Initialisez le dépôt avec la commande **git init**.



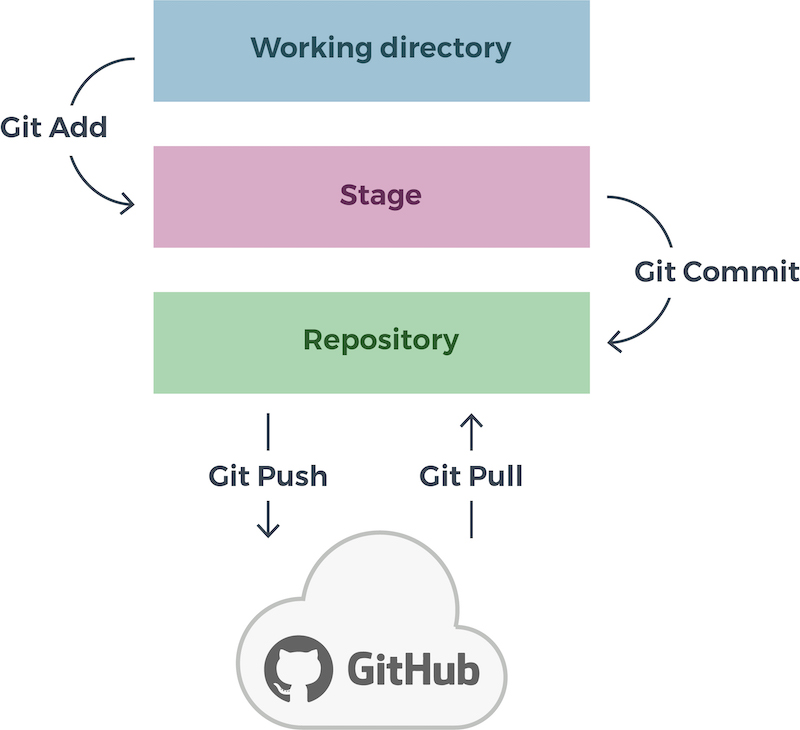
Affichez les dossiers masqués

Votre dépôt est maintenant initialisé, et si vous faites apparaître les dossiers masqués, vous pouvez voir le dossier .git.

Si vous ne le savez pas déjà, un des avantages majeurs de Git réside dans l’aspect local des travaux réalisés : un dépôt Git gère son cycle de vie local indépendamment de la connectivité avec son dépôt distant. On gagne ainsi en performance, mais pas seulement…

Git gère les versions de vos travaux locaux à travers 3 zones locales majeures :

* **le répertoire de travail** (working directory/WD) ;
* **l’index, ou stage**(nous préférerons le second terme) ;
* **le dépôt local**(Git directory/repository).



3 zones locales

L'index, ou stage, désigne tous les fichiers modifiés que vous souhaitez voir apparaître dans votre **prochain commit**. C'est avec la fonction  git add  que l'on ajoute un fichier au stage.

Le dépôt local est **l'historique de l'ensemble de vos actions** (commits, configurations...). L'archivage se fait principalement avec la commande  git commit.  Il est possible d'accéder à cet historique en faisant un  git reflog  qui affichera toutes vos actions et leurs SHA. Le SHA, c'est ce grand code qui vous permettra de revenir à un commit exact. C'est l’identifiant de votre action !

Passons à nos erreurs ! :pirate:

### J’ai créé une branche que je n’aurai pas dû créer

Votre bac à sable est prêt !

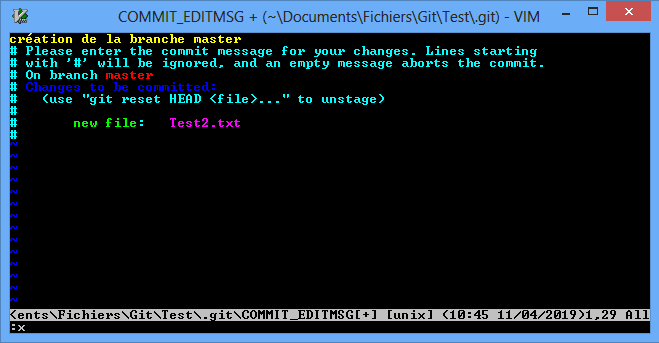
Avant de créer une branche, vous devez créer votre branche principale (master). Pour créer la branche master, vous devez simplement ajouter un fichier et le commiter.

Créez un fichier "PremierFichier.txt" dans votre répertoire Test, et ajoutez-le avec la commande :

git add PremierFichier.txt

git commit

On vous demande alors d'indiquer le message du commit puis de valider. Pour valider le message, une fois que vous l'avez écrit, appuyez sur Echap (votre curseur va basculer sur la dernière ligne) et tapez  :x. Cette commande va sauvegarder et quitter l'éditeur des messages de commit.



Nous allons maintenant créer une branche (autre que la master).

git branch brancheTest

Cette commande, comme nous l'avons vu précédemment, va créer la branche brancheTest.

Nous pouvons d'ailleurs le vérifier avec la commande :



Youppiiii !:ange:

Arf... En fait, non, nous voulions ajouter nos fichiers avant de la créer et nous sommes maintenant bloqués avec cette branche que nous ne voulions pas tout de suite. :euh::euh::euh:

Heureusement, il est très simple sous Git de supprimer une branche que nous venons de créer. Pour cela, il suffit d'exécuter la commande :

git branch -d brancheTest

Tadammm !  :magicien: Notre branche est supprimée et nous pouvons ajouter dans un premier temps nos fichiers avant de créer la branche.



Attention, si toutefois vous avez déjà fait des modifications dans la branche que vous souhaitez supprimer, il faudra soit faire un commit de vos modifications, soit mettre vos modifications de côté, et ça, je vous l'expliquerai un peu plus tard ; soit forcer la suppression en faisant :

git branch -D brancheTest

Forcer la suppression de cette manière entraînera la suppression de tous les fichiers et modifications que nous n'aurez pas commité sur cette branche.

### ****J’ai modifié la branche principale****

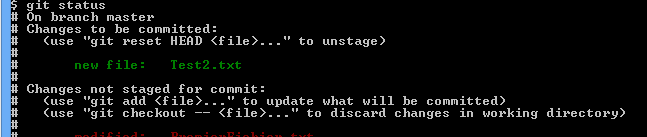
Nous allons maintenant voir le cas où vous avez modifié votre branche master par erreur.  :waw: Nous allons dans un premier temps voir ensemble le cas où vous avez modifié votre branche master mais que vous n'avez pas encore fait le commit, et nous verrons dans un second temps le cas où vous avez commité.

Vous avez modifié votre branche master avant de créer votre branche et vous n'avez pas fait le commit. Ce cas est un peu plus simple. Nous allons faire ce qu'on appelle une remise. La remise va permettre de mettre vos modifications de côté, le temps de créer votre nouvelle branche et ensuite appliquer cette remise sur la nouvelle branche. :-°

Afin de voir comment cela fonctionne, allez sur votre branche master, modifiez des fichiers. Vous pouvez à tout moment voir à quel état sont vos fichiers en faisant :

git status

Vous pouvez voir vos fichiers qui ont été modifiés mais qui n'ont pas encore été commités.



Nous allons donc créer une remise.

git stash

Cela vous donne :



Vous pouvez maintenant vous assurer que votre branche master est de nouveau propre, en faisant un nouveau git status

Vous devriez avoir :



Parfait ! C'est exactement ce que nous voulions. Nous allons maintenant créer notre branche "brancheCommit".

git branch brancheCommit

Nous obtenons donc :



Nous allons basculer sur cette branche.

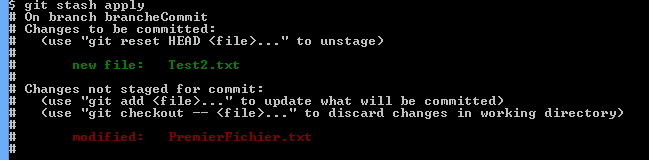
git checkout brancheCommit

Ce qui nous donne :



Et finalement, nou allons pouvoir appliquer la remise, afin de récupérer nos modifications sur notre nouvelle branche.

git stash apply



Cette commande va appliquer la dernière remise qui a été faite. Si pour une raison ou une autre, vous avez créé plusieurs remises, et que la dernière n'est pas celle que vous souhaitiez appliquer, pas de panique, il est possible d'appliquer une autre remise.

Nous allons d'abord regarder la liste de nos remises. Pour ce faire :

git stash list

Cette commande va nous retourner un "tableau" des remises avec des identifiants du style :



Il suffira alors d'appeler la commande git stash  en indiquant l'identifiant.

git stash apply stash@{0}

Et voila, le tour est joué ! :D

Maintenant, admettons que vous ayez réalisé vos modifications et qu'en plus vous ayez fait le commit. Le cas est plus complexe, puisque vous avez enregistré vos modifications sur la branche master, alors que vous ne deviez pas.

Allez-y, modifiez des fichiers, et réalisez le commit. :'(

Nous allons devoir aller analyser vos derniers commits avec la fonction  git log, afin de pouvoir récupérer l'identifiant du commit que l'on appelle couramment le hash. Par défaut, git log  va vous lister par ordre chronologique inversé tous vos commits réalisés.

$ git log

commit ca83a6dff817ec66f443420071545390a954664949

Author: Marc <Marc@gee-mail.com>

Date: Mon Mar 19 21:52:11 2019 -0700

Maintenant que vous disposez de votre identifiant, gardez-le bien de côté. Vérifiez bien que vous êtes sur votre branche master et réalisez la commande suivante :

git reset --hard HEAD^

Cette ligne de commande va permettre de supprimer de la branche master votre dernier commit.  Le Head^ indique que c'est bien le dernier commit que nous voulons supprimer.

Nous allons maintenant créer notre nouvelle branche.

git branch brancheCommit

Nous allons basculer sur cette branche.

git checkout brancheCommit

Maintenant que nous sommes sur la bonne branche, nous allons de nouveau faire un git reset, mais celui-ci va permettre d'appliquer ce commit sur notre nouvelle branche ! Il n'est pas nécessaire d'écrire l'identifiant en entier. Seuls les 8 premiers caractères sont nécessaires.

git reset --hard ca83a6df

Et voila, le tour est joué ! :D

### Je souhaite changer le message de mon commit

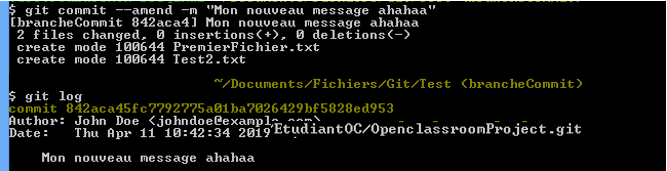
Lorsque l'on travaille sur un projet avec Git, il est très important, lorsque l'on propage les modifications, de bien marquer dans le message descriptif les modifications que l'on a effectuées. Si jamais vous faites une erreur dans l'un de vos messages de commit, il est tout à fait possible de changer le message après coup.

Attention ! Cette commande va fonctionner pour le dernier commit réalisé !

Imaginons que vous veniez de faire un commit et que vous ayez fait une erreur dans votre message. L'exécution de cette commande, lorsqu'aucun élément n'est encore modifié, vous permet de modifier le message du commit précédent sans modifier son instantané. L'option -m permet de transmettre le nouveau message.

git commit --amend -m "Votre nouveau message de commit"

Et on vérifie avec  git log  :



Elle est pas belle la vie ? :p

### J’ai oublié un fichier dans mon dernier commit

Imaginons maintenant (je sais, il faut beaucoup d'imagination dans ce cours) :D que vous ayez fait votre commit mais que vous réalisiez que vous avez oublié un fichier. :honte: Ce n'est pas bien grave ! Nous allons réutiliser la commande git --amend, mais d'une autre manière. La fonction git --amend, si vous avez bien compris, permet de modifier le dernier commit.

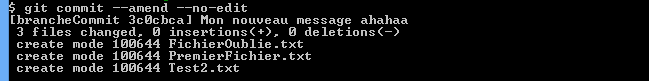
Nous allons donc réutiliser cette fonction, mais sans le -m qui permettait de modifier son message.

Nous allons dans un premier temps ajouter notre fichier, et dans un deuxième temps réaliser le git --amend.

git add FichierOublie.txt

git commit --amend --no-edit

Votre fichier a été ajouté à votre commit et grâce à la commande --no-edit que nous avons ajoutée, nous n'avons pas modifié le message du commit.



Pour résumer, git commit --amend vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements en attente. Vous pouvez ajouter ou supprimer des changements afin de les appliquer avec commit --amend. Si aucun changement n'est en attente, --amend vous permet de modifier le dernier message de log du commit avec -m.

Vous savez maintenant comment corriger les erreurs les plus courantes sur votre dépôt local. Mais comment procéder si vous vous trompez sur votre dépôt distant ? C'est ce que nous allons découvrir dans le prochain chapitre !

## Corrigez vos erreurs sur votre dépôt distant

### Corrigez vos erreurs en local et à distance

La journée avait été difficile et par mégarde **vous avez pushé des fichiers erronés**. Le problème, c'est que maintenant ce n'est plus que sur votre dépôt local, mais à disposition de tout le monde. La première chose à faire est de prévenir vos collaborateurs. Nous avons tout de même la solution à votre problème. ;)

Il est possible d'annuler son commit public avec la commande **Git revert**. L'opération Revert annule un commit en créant un nouveau commit. C'est une méthode sûre pour annuler des changements, car elle ne risque pas de réécrire l'historique du commit.

git revert HEAD^

Nous avons maintenant reverté notre dernier commit public et cela a créé un nouveau commit d'annulation. Cette commande n'a donc **aucun impact sur l'historique** ! Par conséquent, il vaut mieux utiliser  git revert  pour annuler des changements apportés à une branche publique, et git reset  pour faire de même, mais sur une branche privée.

Gardez à l'esprit que Git revert sert à annuler des changements commités, tandis que Git reset HEAD permet d'annuler des changements non commités.

Toutefois, attention, Git revert peut écraser vos fichiers dans votre répertoire de travail, il vous sera donc demandé de commiter vos modifications ou de les remiser.

### L'accès à distance ne fonctionne pas

Git base toute sa gestion d’authentification sur le mécanisme **des clés SSH**. Ce système est d’ailleurs immensément utile de façon générale sous Linux, Unix et OSX, dès qu’il s’agit de s’authentifier sur une machine tierce. Afin d’avoir un maximum de confort lorsqu’on accède à un dépôt nécessitant une identification (lecture de dépôts privés ou écriture dans le cas général), il est donc important de bien maîtriser les clés SSH.

Nous allons maintenant générer notre duo de clés SSH :

Dans Git Bash, exécutez la commande :

$ ssh-keygen -t rsa -b 4096 -C "johndoe@example.com"

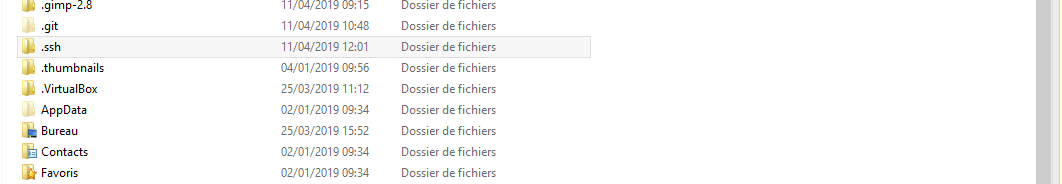
Vous obtenez ceci :



Vous pouvez soit appuyer sur Entrée, soit indiquer un nom de fichier. Un mot de passe vous est ensuite demandé.

Félicitations ! Vous avez obtenu votre clé SSH !

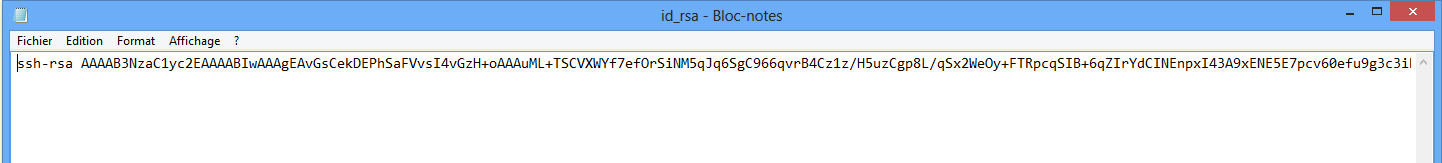
Pour la trouver, il suffit d'aller à l'adresse : C:\Users\VotreNomD'Utilisateur\, et d'afficher les dossiers masqués.



Affichez les dossiers masqués

Dans ce dossier, vous avez donc deux fichiers, votre clé publique et votre clé privée.

La clé id\_rsa.txt est votre **clé privée** alors que la clé id\_rsa.pub est votre**clé publique**. Vous pouvez copier votre clé en l'ouvrant dans un bloc-notes.



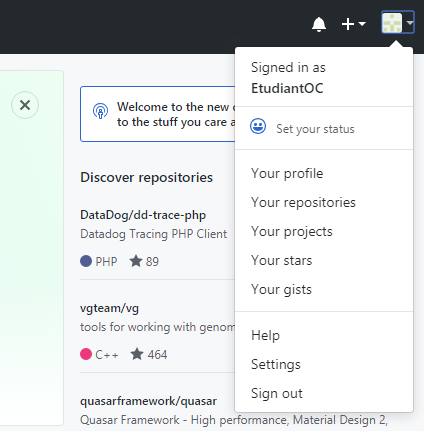
Copiez votre clé

Maintenant que vous disposez de votre clé SSH, allons voir comment l'ajouter pour GitHub !

### Modifiez vos informations d'identification et supprimez la clé

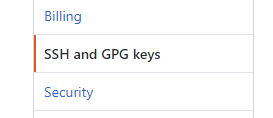
Ajoutons maintenant notre clé à notre compte GitHub.

Connectez-vous à votre espace GitHub puis allez dans l'angle droit de votre compte et cliquez sur Settings.



Espace "Settings" dans votre espace GitHub

Cliquez sur SSH and GPG keys :



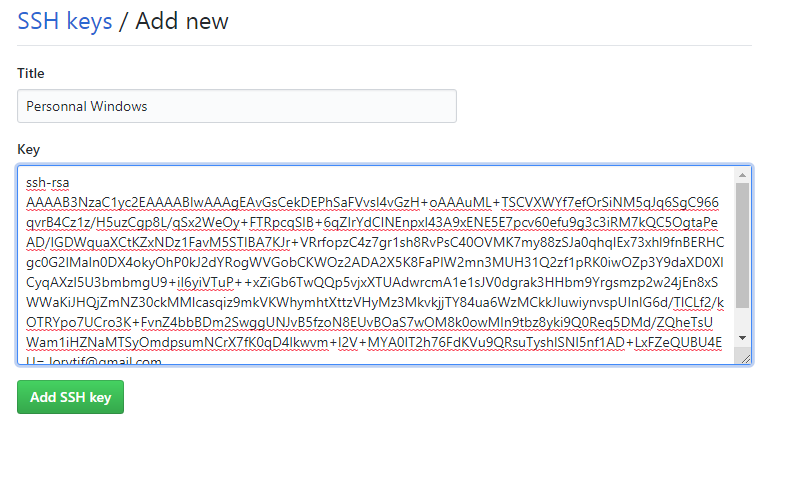
Cliquez sur "SSH and GPG keys"

Puis sur New SSH Key :

Cliquez sur

Cliquez sur "New SSH Key"

Choisissez un titre et collez votre clé SSH :



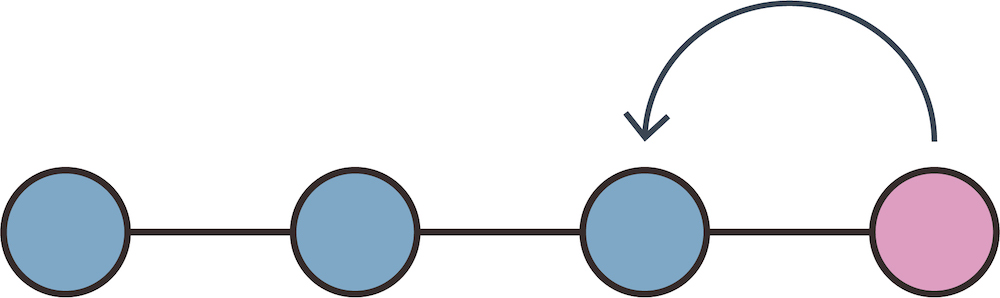
Ajoutez une clé SSH

Vous devrez ensuite confirmer votre mot de passe, et votre clé SSH sera alors ajoutée à votre compte GitHub !

Dans le chapitre suivant, vous allez apprendre à utiliser une commande très utile de Git : Git reset. À tout de suite !

**Utilisez Git reset**

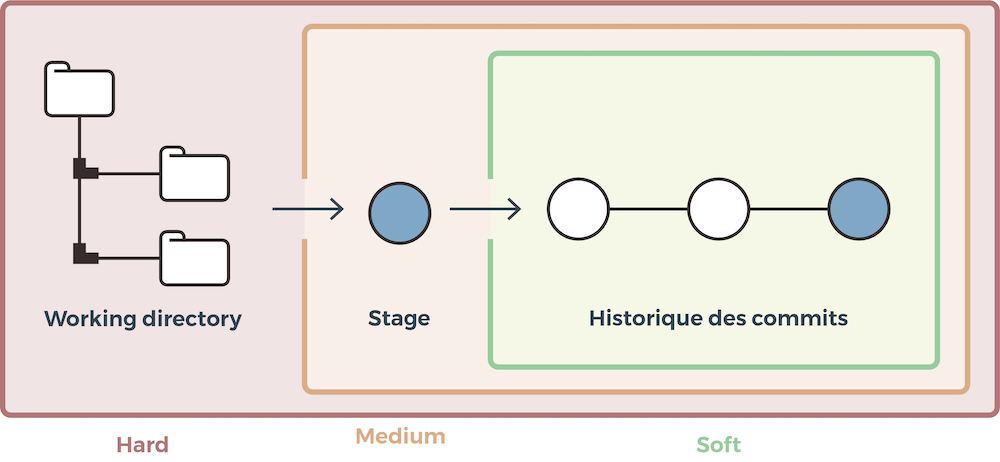
Imaginez que votre client vous demande une nouvelle fonctionnalité ; vous travaillez dessus toute la journée et le lendemain, finalement, il change d'avis. Catastrophe ! :waw: Premièrement, vous avez perdu une journée à développer une fonctionnalité pour rien, mais en plus il faut que vous trouviez le moyen de revenir en arrière ! Heureusement, notre ami Git arrive à notre rescousse avec la commande  git reset  !



Git reset

**Les trois types de réinitialisation de Git**

La commande  git reset  est un outil complexe et polyvalent pour **annuler les changements**. Elle peut être appelée de trois façons différentes, qui correspondent aux arguments de ligne de commande**--soft, --mixed et --hard**.



3 types de réinitialisation

Nous allons commencer par Reset --hard. Attention, si vous voulez exécuter cette commande, vérifiez 5 fois avant de la lancer et soyez sûr de vous à 200 % .

Pour ce faire, nous allons exécuter la commande :

git reset notreCommitCible --hard

Cette commande va permettre de **revenir à n'importe quel commit** mais en oubliant absolument tout ce qu'il s'est passé après ! Quand je dis tout, c'est TOUT ! Que vous ayez fait des modifications après ou d'autres commits, tout sera effacé ! C'est pourquoi il est extrêmement important de revérifier plusieurs fois avant de la lancer, vous pourriez perdre toutes vos modifications si elle est mal faite.

Cette utilisation de  git reset  constitue une manière simple d'annuler des changements qui n'ont pas encore été partagés. Cette commande est incontournable lorsque vous commencez à travailler sur une fonctionnalité, puis que tout à coup, vous vous rendez compte que vous vous êtes trompé et que vous voulez tout recommencer.

Le  git reset --mixed  va permettre de revenir juste après votre dernier commit ou le commit spécifié, sans supprimer vos modifications en cours. Il va par contre créer un HEAD détaché. Il permet aussi, dans le cas de fichiers indexés mais pas encore commités, de désindexer les fichiers.

git reset HEAD~

Si rien n'est spécifié après  git reset, par défaut il exécutera un  git reset --mixed HEAD~.

Nous avons enfin le  git reset --Soft. Le **git reset --Soft** permet juste de se placer sur un commit spécifique afin de voir le code à un instant donné ou créer une branche partant d'un ancien commit. Il ne supprime aucun fichier, aucun commit, et ne crée pas de HEAD détaché.

Le **HEAD**, si vous n'êtes pas sûr d'avoir bien compris, est un pointeur, une référence sur notre position actuelle dans notre répertoire de travail Git. C’est un peu comme notre ombre : elle nous suit où qu’on aille ! Par défaut, HEAD pointe sur la branche courante, master, et peut être déplacé vers une autre branche ou un autre commit.

**Oups, j'ai des conflits !**

Vous avez vu dans le chapitre précédent comment fusionner des branches. Nous avons utilisé un exemple assez simple où tout s'est bien passé. Mais malheureusement, il arrive parfois, même souvent, que cela ne se passe pas aussi bien et que des conflits apparaissent. o_O  Cela arrive souvent lorsque plusieurs personnes travaillent sur un même fichier et modifient les mêmes lignes.

Prenons un exemple simple. Pendant vos développements, Henry et Bob doivent travailler sur la fonction Hello World (oui, je sais, c'est très recherché). :p Henry affiche alors le message "Hello World" alors que Bob affiche le message "Hello World !". Sauf que lors du commit, Git va voir que sur la même ligne on essaie de lui commiter deux choses différentes et il ne va pas pouvoir deviner lequel prendre. Il est fort mais pas devin ! :magicien: Git va donc **afficher un conflit** sur le fichier HelloWorld.php. Ce conflit, vous allez donc devoir le résoudre avant de faire le commit.

Pour cela, rien de plus simple, vous allez devoir ouvrir un éditeur. Si vous utilisez VIM, vous pouvez l'ouvrir avec la ligne de commande :

vim helloworld.php

Sinon, vous ouvrez votre éditeur habituel. Vous allez devoir maintenant régler les conflits en comparant les deux fichiers HelloWorld.php et en choisissant quelles modifications vous gardez en fusionnant les fichiers.

Maintenant que vous avez résolu le conflit, il vous reste à le dire à Git !

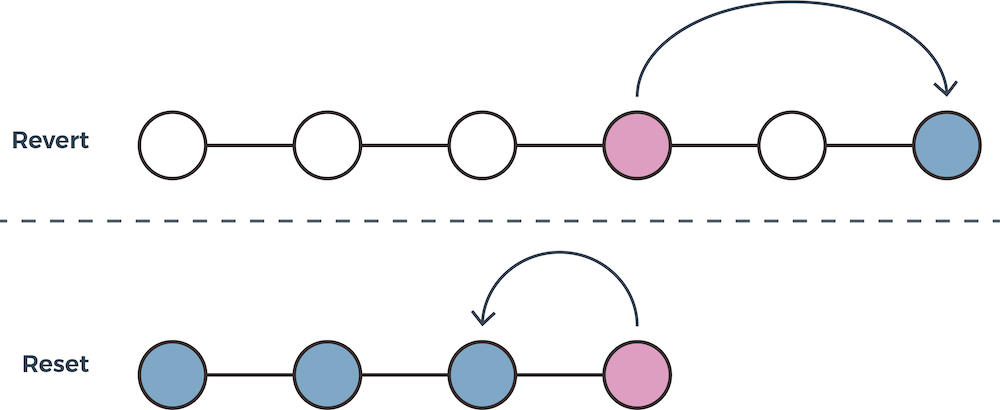
git commit

Git va détecter que vous avez résolu les conflits et va vous proposer un message de commit. Vous pouvez bien entendu le modifier.

**La machine à remonter le temps ! J’ai ajouté le mauvais fichier au commit**

Ralalala ! Décidément vous en faites des bêtises ! :D  Alors comme cela, vous avez fait un commit mais un fichier s'est glissé par "erreur". Ne vous inquiétez pas, avec Git nous avons une super fonction qui va **remonter le temps**. Elle va réaliser une sorte de Undo, mais en faisant un deuxième commit. Elle ne va pas revenir en arrière et supprimer votre commit, mais va inverser vos actions dans le commit et réaliser un second commit. Au lieu de supprimer le commit de l'historique du projet, elle détermine comment **annuler les changements** introduits par le commit et ajoute un nouveau commit avec le contenu ainsi obtenu.  Vous allez donc revenir à l'état précédent mais avec un nouveau commit. Ainsi, Git ne perd pas l'historique, lequel est important pour l'intégrité de votre historique de révision et pour une collaboration fiable.

La différence entre Revert et Reset est que Reset va revenir à l'état précédent sans créer un nouveau commit, alors que Revert va créer un nouveau commit.



Revert et Reset

Essayons cette super commande en faisant un premier commit que nous allons finalement ne plus vouloir. Une fois votre commit fait, écrivez la commande suivante :

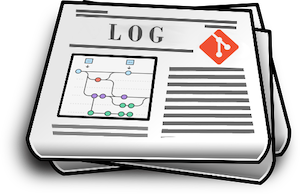
git revert HEAD

Une fois, votre commit "annulé", vous allez pouvoir enlever votre fichier, et réaliser de nouveau votre commit. :)

*Voyons comment corriger un commit raté dans le prochain chapitre !*

**Corrigez un commit raté**

Cela faisait plusieurs semaines que tout allait bien dans le plus beau des mondes et que vous n'aviez pas eu de problèmes au cours de vos développements. Sauf que la vie de développeur n'est jamais Bisounours et votre chef de projet vous dit que l'on va avoir besoin de revenir à une version précédente, et qu'il a besoin de savoir qui a fait quel commit ! :waw:  Heureusement, les techniques de **journalisation** de Git ont exactement été prévues pour répondre à ce genre de situation.



Log

**Un trou de mémoire ? Git reflog !**

L'objectif d'un système de contrôle de versions est d'enregistrer les changements apportés à votre code. Il vous permet de **consulter l'historique de votre projet** pour voir qui a contribué à quoi, de déterminer où des bugs ont été introduits et d'annuler les changements problématiques. Oui, mais disposer de cet historique est inutile si vous ne savez pas comment l'utiliser !  C'est là que la commande git log  entre en scène !

git log

Par défaut,  git log   **énumère en ordre chronologique inversé** les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier. Cette commande affiche chaque commit avec son identifiant SHA, l'auteur du commit, la date et le message du commit.

Git dispose d'un outil encore plus puissant, poussant le journal de logs à l’extrême.

git reflog

git reflog  va loguer les commits, mais aussi toutes les autres actions que vous avez pu faire en local. Git reflog va enregistrer vos commits, vos modifications de messages, vos merges, vos resets, enfin tout, quoi. :-°  Ce qui est très pratique, c'est que comme Git log, Git reflog va afficher un identifiant SHA-1 pour chaque action. Il est donc très facile de revenir à une action donnée **grâce au SHA.** Cette commande, c'est votre joker, elle assure votre survie en cas d'erreur.  Pour revenir à une action donnée, on prend les 8 premiers caractères de son SHA et on fait :

git checkout e789e7c

TADAMMMMMMM ! Vous avez été propulsé dans le temps. :magicien:

**Qui s’est amusé dans mon dépôt ? Git blame**

Maintenant que vous êtes un pro des logs, que vous savez trouver l'auteur des derniers commits, il est parfois utile de connaître à l'intérieur d'un fichier **qui a modifié chaque ligne**. Cela peut permettre de trouver l'auteur d'un bug. :pirate:

La commande  git blame  permet **d’examiner le contenu d’un fichier ligne par ligne** et de déterminer la date à laquelle chaque ligne a été modifiée, et le nom de l’auteur des modifications.:lol:

git blame monFichier.php

git blame  va afficher pour chaque ligne modifiée, son ID, l'auteur, l'horodatage, le numéro de la ligne et le contenu de la ligne. Avec git blame, vous devenez un super espion de votre projet !

**Il me faut ce commit ! Vite Git cherry-pick**

Lorsque vous travaillez avec une équipe de développeurs sur un projet de moyenne à grande taille, la gestion des modifications entre plusieurs branches de Git peut devenir une tâche complexe. Parfois, vous ne voulez pas fusionner une branche entière dans une autre et vous n'avez besoin que de choisir un ou deux commits spécifiques. Ce processus s'appelle**cherry-pick** ! Attention, Git cherry-pick n'est pas très apprécié dans la communauté des développeurs ! En effet, cette commande va **dupliquer des commits existants**. Il sera préférable, si possible, de réaliser un merge.

Admettons que vous travailliez sur votre branche "Mes évolutions", et que vous ayez déjà réalisé plusieurs commits. Votre collègue a besoin de l'une de ces évolutions pour la livrer au client, mais pas des autres. C'est dans ce cas bien précis que nous allons faire appel à  git cherry-pick  ! Cette commande va permettre de sélectionner un ou plusieurs commits grâce à leurs SHA (décidément ils sont partout) et de les migrer sur la branche master, sans pour autant fusionner toute la branche "Mes évolutions".

git cherry-pick d356940 de966d4

Ici, nous prenons les deux commits ayant pour SHA d356940  et de966d4, et nous les ajoutons à la branche master sans pour autant les enlever de votre branche actuelle. Nous les dupliquons !

Vous êtes arrivé à la fin de cette partie et vous savez maintenant corriger toutes vos erreurs sur Git et GitHub ! Pour le vérifier, je vous invite à réaliser le quiz de cette partie, puis on se retrouve dans la partie suivante. À tout de suite !

# Réparez les erreurs les plus courantes

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Corriger les erreurs courantes sur GitHub

### Question 1

**Maintenant que je suis sur ma Branch1, nous allons ajouter un fichier "File2.txt".**

**Finalement, je ne veux plus ajouter mon fichier sur cette branche mais sur une nouvelle branche, "BranchFile". Que dois-je faire ?**

* + 

git status  
git branch BranchFile  
git checkout BranchFile  
git status apply

* + 

git status  
git stash  
git branch BranchFile  
git checkout BranchFile  
git status apply

* + 

git status  
git stash  
git branch BranchFile  
git checkout BranchFile  
git stash apply

*Nous abordons ici la remise. Nous allons devoir remiser les changements faits sur Branch1 pour les appliquer ensuite sur la nouvelle branche, "BranchFile".  git stash  permet de remiser des modifications et  git stash apply  permet d'appliquer une remise sur une branche.*

### Question 2

git commit --amend -m "Test"

**Cette commande va :**

* + 

permettre de revenir au commit précédent

* + 

permettre de modifier le message du commit précédent

* + 

permettre de créer un nouveau commit avec le message "Test"

*git commit --amend -m modifie le message du commit précédent. C'est le -m qui indique que nous allons modifier le message.*

### Question 3

**Je souhaite revenir sur le commit précédent et modifier son contenu en ajoutant un fichier, que dois-je faire ?**

* + 

git commit --amend --no edit  
git add monFichierOublié  
git commit

* + 

git add monFichierOublié  
git commit --amend --no-edit

* + 

git commit --amend  
git add monFichierOublié  
git commit

* + 

git add monFichierOublié  
git commit --amend

*git add monFichierOublié  permet d'ajouter mon fichier.  
git commit --amend --no-edit permet de modifier le commit sans changer le message.*

### Question 4

**Quelle est la différence entre Git revert et Git reset ?**

* + 

Git revert ne va réinitialiser qu'un commit alors que Git reset réinitialise tout.

* + 

Git revert va créer un nouveau commit alors que Git reset, non.

* + 

Git reset va créer un nouveau commit alors que Git revert, non.

*Git reset ne crée jamais un nouveau commit alors que Git revert, oui.*

### Question 5

**Je souhaite savoir qui a touché à une ligne en particulier dans le fichier test.html, quelle commande vais-je devoir exécuter ?**

* + 

git log

* + 

git reflog

* + 

git blame

* + 

git cherry pick

*Si vous ne vous en souvenez plus, allez faire un petit tour sur le cours "Qui s’est amusé dans mon dépôt ? Git blame".*

### Question 6

**Quelles sont les trois zones locales majeures ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Working directory

* + 

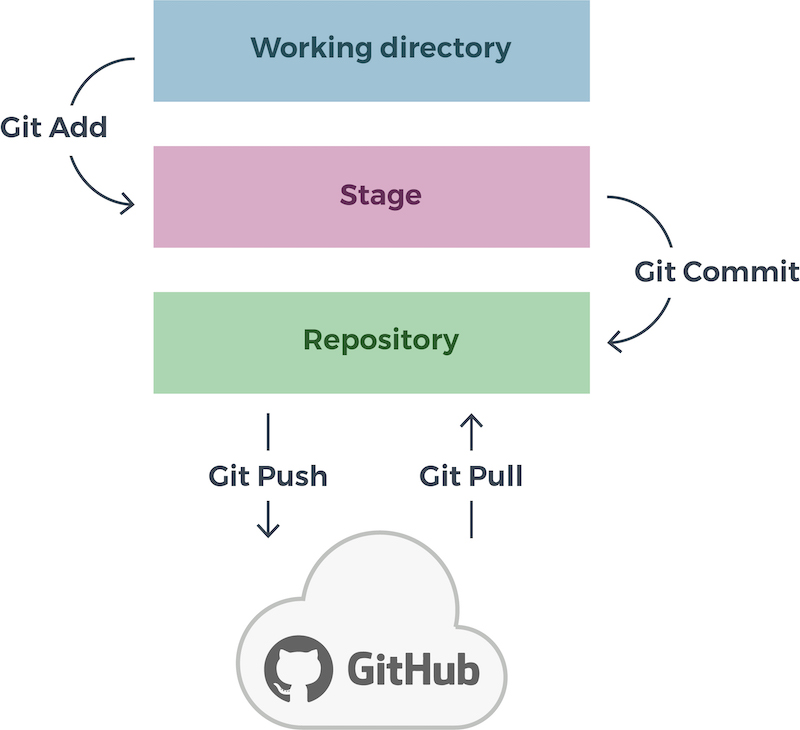
Cloud

* + 

Repository

* + 

Stage

*3 zones locales majeures*

*Git gère les versions de vos travaux locaux à travers 3 zones locales majeures :*

* + ***le répertoire de travail****(working directory/WD) ;*
  + ***l’index ou*stage***(nous préférerons le second terme) ;*
  + ***le dépôt local****(Git directory/repository).*

*Le répertoire de travail désigne****l'arborescence****de vos fichiers, c'est-à-dire tous vos fichiers et répertoires qui sont indépendants de Git. Ils sont même là avant que vous réalisiez votre Git init !*

*L'index ou*stage*désigne tous les fichiers modifiés que vous souhaitez voir apparaître dans votre****prochain commit****. C'est avec la fonction  git add  que l'on ajoute un fichier au stage.*

*Le dépôt local est****l'historique de l'ensemble de vos actions****(commits, configurations...).*

### Question 7

**Qu'est ce qu'un SHA ?**

* + 

Un SHA est un numéro aléatoire permettant de se connecter à GitHub

* + 

Un SHA est un identifiant pour les commits et autres actions gardés en mémoire par Git

* + 

Un SHA est un petit animal qui ronronne

*Le SHA, c'est ce grand code qui vous permettra de revenir à un commit exact. C'est l’identifiant de votre action !*

## Identifiez la structure de fichier de Git

Maintenant que vous savez utiliser les fonctionnalités de base de Git, nous allons voir sa façon de **structurer les dossiers** et son fonctionnement. Il est primordial de connaître le fonctionnement interne d'un programme afin de pouvoir par la suite l'utiliser au mieux !

### Découvrez l’arbre Git et sa structure

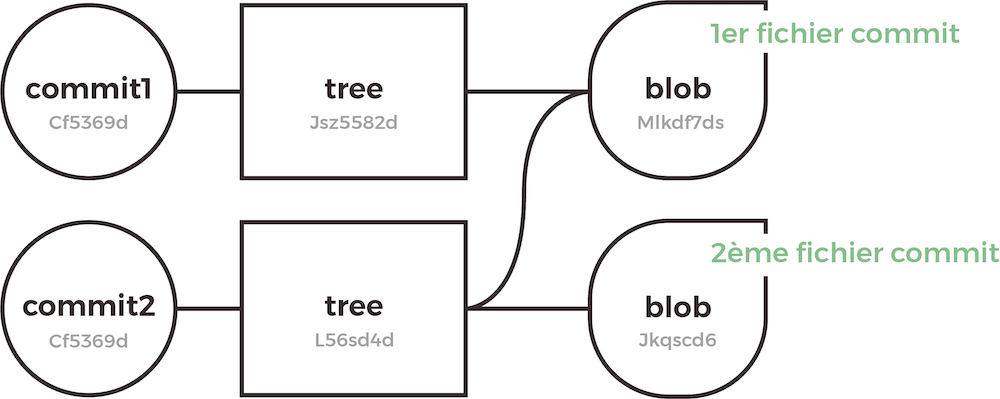
Il existe trois types d'objets et demi. :D Si si !  Et demi. :pirate: En effet, il existe trois types d'objets principaux et un qui est un peu plus secondaire.

Les trois principaux types d'objets sont :

* le "**tree**" ou l'arbre Git qui est une forme de répertoire. Il va référencer une liste de trees et de blobs (sous-répertoires et fichiers) ;
* le "**commit**" qui va pointer vers un arbre spécifique et le marquer, afin de représenter son état à un instant donné ;
* Le "**blob**" qui représente en général un fichier (Binary Large Object).

Git n'utilise donc pas les noms des fichiers et des répertoires pour classer et stocker vos données, mais il utilise leur empreinte ou identifiant SHA-1 !

Il existe un dernier objet qui ne l'est pas vraiment. o_O C'est l'objet Tag. Le tag va représenter un commit d'une version spécifique. Mais ne nous étalons pas dessus. Le plus important est que vous reteniez le tree, le blob et le commit. :)



Tree, Blob et Commit

### Représentation cryptographique d'un commit

Toutes les informations nécessaires pour décrire l’historique d’un projet sont stockées dans des fichiers référencés par un identifiant de 40 caractères qui ressemble à quelque chose comme ça :

8gh96c4636981e4759825791c8ea3bcf5f2278t9

Pour chacun des objets dans Git, vous trouverez cette chaîne de 40 caractères que nous appelons le **hash SHA-1.** Celui-ci représente le contenu de l'objet. Pour deux objets différents, il est donc impossible d'avoir le même nom. Cela a l'avantage que par conséquent, Git peut tout de suite reconnaître deux objets identiques. **Le commit étant un objet**, lui aussi a son empreinte SHA-1. Il est donc tout à fait possible d'appeler n'importe quel commit à n'importe quel moment grâce à cet identifiant unique.

Il est possible sur Internet de trouver des générateurs de hash SHA-1. Vous pouvez donc vous-même tester en mettant le contenu de votre fichier dans l'un de ces générateurs. :)

Exemple de générateur [ici](https://www.sha1.fr/).

Je vous conseille fortement d'aller y jeter un œil afin de comprendre que le générateur vous donnera toujours la même clé pour le même texte. :)

### Comment fonctionne la fusion sous Git ?

Il est très courant sous Git de vouloir fusionner le travail fait sur différentes branches. Pour cela, nous avons la fonction **Merge**. Un  git merge  ne devrait être utilisé que pour la récupération fonctionnelle, intégrale et finale d’une branche dans une autre, afin de préserver un graphe d’historique sémantiquement cohérent et utile, lequel représente une véritable valeur ajoutée. Comme son nom l’indique,  merge  réalise une **fusion**.  git merge  va combiner plusieurs séquences de commits en un historique unifié. Le plus souvent,   git merge  est utilisé pour combiner deux branches.  git merge  va créer un nouveau commit de merge.

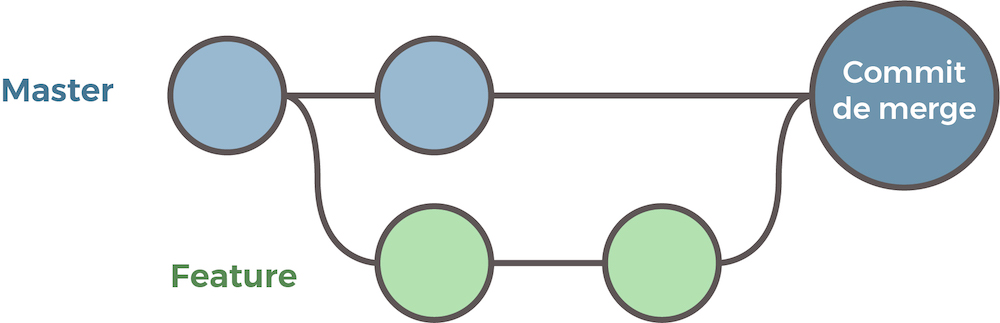
Imaginons que vous ayez votre branche master et une branche "Nouvelle fonctionnalité". Nous souhaitons maintenant faire un merge de cette branche de fonctionnalité dans la branche master. Appeler cette commande permettra de merger la fonctionnalité de branche spécifiée dans la branche courante, disons master.

Il faut toujours préparer le terrain avant de réaliser un merge !

Vous devez toujours vous assurer d'être sur la bonne branche. Pour cela, vous pouvez réaliser un  git status. Si vous n'êtes pas sur la bonne, réalisez un  git checkout, pour changer de branche. Maintenant que le terrain est prêt, vous pouvez réaliser votre merge.

git merge Nouvelle Fonctionnalité

Votre branche Nouvelle fonctionnalité va être fusionnée sur la branche master en créant un nouveau commit.



Fusion

Si les deux branches que vous essayez de fusionner modifient toutes les deux la même partie du même fichier, Git ne peut pas déterminer la version à utiliser. Lorsqu'une telle situation se produit, Git s'arrête avant le commit de merge, afin que vous puissiez résoudre manuellement les conflits.

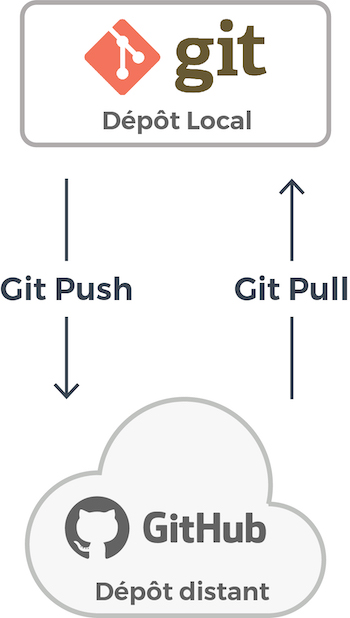
### Les options Git pull/Git push

La commande Git pull permet de télécharger les modifications qui ont eu lieu sur le dépôt distant, dans le but de les rapatrier sur le dépôt local. Git pull est en réalité la fusion de deux commandes Git :  git merge  que nous venons de voir et  git fetch  que nous verrons juste après.  git pull va créer un nouveau commit de fusion comme le fait  git merge. La commande  git pull  exécute d'abord  git fetch  qui télécharge le contenu du référentiel distant spécifié. Ensuite, un git merge  est exécuté pour fusionner les modifications du dépôt distant et créer un nouveau commit de merge en local.

git pull <remote>

À l'inverse, la commande Git push permet d'envoyer des modifications que l'on a réalisées en local sur le dépôt à distance.

git push <remote>



Pull et Push

Nous allons voir maintenant la commande Git fetch d'un peu plus près. Elle aussi permet de récupérer les modifications d'un dépôt distant.

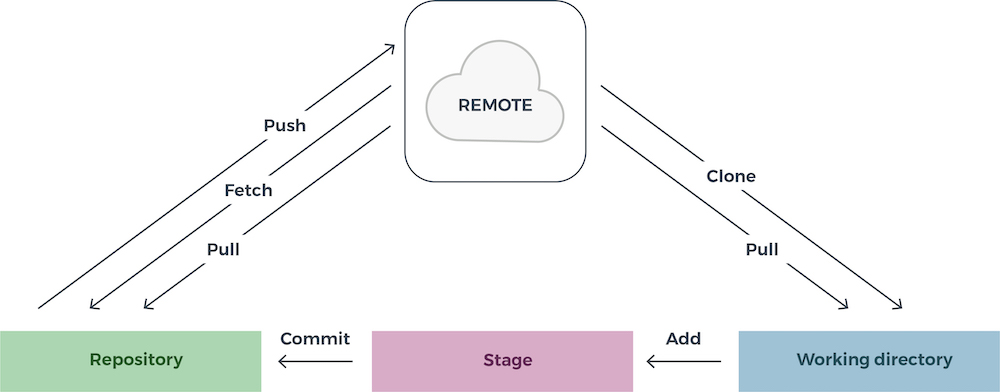
### À quoi sert Git fetch ?

Git fetch, contrairement à Git pull, va aller **chercher les modifications** sur le dépôt distant mais**ne va pas les fusionner avec nos modifications locales**. Git isole le contenu récupéré en tant que contenu local existant, cela n'a absolument aucun effet sur votre travail de développement local. La commande Git fetch va récupérer toutes les données des commits effectués sur la branche courante qui n'existent pas encore dans votre version en local. **Ces données seront stockées dans le répertoire de travail local**, mais ne seront pas fusionnées avec votre branche locale. Si vous souhaitez fusionner ces données pour que votre branche soit à jour, vous devez utiliser ensuite la commande Git merge.

Le choix de la commande à utiliser dépend de la façon dont vous souhaitez travailler.

**La commande Git pull automatise la mise à jour des données**, mais peut entraîner de nombreux conflits si vous avez modifié beaucoup de fichiers. Utiliser la commande Git fetch permet de garder son répertoire de travail à jour et de contrôler le moment où l'on souhaite fusionner les données.

Voici un petit récapitulatif du fonctionnement des fonctions que nous avons vues :



Push et Pull

Maintenant que vous maîtrisez les commandes Pull et Push, passons au rebasage !

**Modifiez vos branches avec Rebase**

**Comment fonctionne Git rebase ?**

**Git rebase** a le même objectif que Git merge. Ces deux commandes permettent de **transférer les changements d'une branche à une autre**. Seule la manière de procéder va différer. Le rebase dispose de puissantes fonctionnalités de réécriture de l'historique. Il existe deux types de rebase : le rebase manuel et le rebase interactif. Au niveau du contenu, le rebase **consiste à changer la base de votre branche d'un commit vers un autre**, donnant l'illusion que vous avez créé votre branche à partir d'un commit différent. Git va prendre vos modifications d'une branche et les transposer sur une autre branche.

Le rebase permet de garder un historique plus clair et plus compréhensible.

Rebaser est une méthode courante pour **intégrer les changements** en amont dans votre répertoire local. L'intégration des changements en amont avec Git merge génère un commit de merge superflu dès que vous voulez voir comment le projet a évolué.

Attention ! Vous ne devez jamais rebaser des commits pushés sur le dépôt public !

Cela remplacerait les anciens commits du dépôt public, et cela  serait comme si votre historique avait brusquement disparu !

La commande  git rebase  standard appliquera les commits à votre branche courante puis à la pointe de la branche transférée.

**Réécrivez l’historique avec l’interactif Rebase de Git**

Exécuter  git rebase  avec l'option -i démarre **une session de rebasage interactive**. Cette fonctionnalité permet de déplacer les commits un à un en ayant la possibilité de les modifier. Vous avez donc la possibilité de supprimer certains commits ou de les modifier. Cette action ouvre un éditeur dans lequel vous pouvez entrer des commandes  pour chaque commit à rebaser.

Voici les commandes possibles :

# Commandes :

# p, pick = utilisez le commit

# r, reword = utilisez le commit, mais éditez le message de commit

# e, edit = utilisez le commit, mais arrêtez-vous pour apporter des changements

# s, squash = utilisez le commit, mais intégrez-le au commit précédent

# f, fixup = commande similaire à "squash", mais qui permet d'annuler le message de log de ce commit

# x, exec = exécutez la commande (le reste de la ligne) à l'aide de Shell

# d, drop = supprimez le commit

Le rebasage interactif vous donne donc **un contrôle complet sur l'historique de votre projet**. Il sert principalement à nettoyer son historique. Il est beaucoup apprécié des développeurs qui aiment nettoyer leurs historiques avant de pousser sur le dépôt distant.

Le rebasage interactif permet ainsi de maintenir la propreté et la cohérence de l'historique d'un projet.

**Modifiez l'ordre des commits**

Décidément, vous voulez en faire des choses. o_O  Modifier l'ordre des commits peut être utile lors d'un nettoyage avant un push sur le serveur distant. Le rebase interactif permet de le faire. Eh oui, on peut vraiment tout faire avec lui. :)

Cela est très simple à mettre en place. Le rebase interactif vous ouvre une fenêtre pour chaque commit à rebaser. Pour chacun de ces commits, vous allez devoir choisir une option dans la liste des commandes que nous avons vues.

# Commandes :

# p, pick = utilisez le commit

# r, reword = utilisez le commit, mais éditez le message de commit

# e, edit = utilisez le commit, mais arrêtez-vous pour apporter des changements

# s, squash = utilisez le commit, mais intégrez-le au commit précédent

# f, fixup = commande similaire à "squash", mais qui permet d'annuler le message de log de ce commit

# x, exec = exécutez la commande (le reste de la ligne) à l'aide de Shell

# d, drop = supprimez le commit

Le petit truc à savoir, c'est que le rebase interactif va créer votre historique dans l'ordre où vous allez agir sur les commits !

Par exemple, vous avez trois commits ; on va faire simple, ils s’appelleront commit1,  commit2 et commit3. C'était difficile, comme noms ! :pImaginons que vous deviez mettre le commit2 avant le commit1 et que vous vouliez supprimer le commit3. On va faire un rebase interactif sur nos trois derniers commits  :

$ git rebase -i HEAD~3

Puis vous utiliserez la commande Pick pour indiquer à Git dans quel ordre vous les voulez, et la commande Drop pour la suppression du commit3.

drop 58gfbg56 commit3

pick 14hg58g1 commit2

pick 25frgf83 commit1

N'oubliez pas que l'on appelle toujours les commits par leurs SHA-1.

**Modifiez les messages de validation**

La commande Git rebase interactif permet aussi de modifier les messages de commit. Imaginons que nous voulions agir sur le dernier commit.

git rebase -i HEAD^

Pour modifier ensuite son message de validation, nous allons utiliser la commande Edit.

edit 54dfiosd

Vous allez devoir sauvegarder et quitter l'éditeur. Vous aurez alors une ligne de commande vous indiquant :

$ git rebase -i HEAD^

Stopped at 54dfiosd... updated the gemspec to hopefully work better

You can amend the commit now, with

git commit --amend

Once you’re satisfied with your changes, run

git rebase --continue

Vous devrez alors faire :

 git commit –amend

Puis modifier votre message de commit et enfin valider en faisant :

$ git rebase --continue

*Le rebase est très pratique pour modifier vos branches, comme nous venons de le voir, mais il est aussi très utile pour les nettoyer ! Voyons cela dans le chapitre suivant !*

**Utilisez des techniques de nettoyage de branche**

**Comment utiliser Rebase pour nettoyer votre branche ?**

Nous avons vu dans le chapitre précédent que Git rebase était très utile pour modifier l'ordre des commits, et modifier les messages de commit. Nous allons voir maintenant que **Git rebase est un véritable couteau suisse de Git.** Il va nous permettre de modifier des commits intermédiaires avant d'envoyer sur le dépôt à distance.

Il est important de toujours nettoyer son historique avant d'envoyer sur le dépôt à distance !

Le rebasage interactif permet ainsi de maintenir la propreté et la cohérence de l'historique d'un projet.

Rappelons les commandes du rebase interactif :

# Commandes :

# p, pick = utilisez le commit

# r, reword = utilisez le commit, mais éditez le message de commit

# e, edit = utilisez le commit, mais arrêtez-vous pour apporter des changements

# s, squash = utilisez le commit, mais intégrez-le au commit précédent

# f, fixup = commande similaire à "squash", mais qui permet d'annuler le message de log de ce commit

# x, exec = exécutez la commande (le reste de la ligne) à l'aide de Shell

# d, drop = supprimez le commit

Sélectionnons les deux derniers commits :

$ git rebase -i HEAD~2

Puis supprimons-les avec la commande Drop :

drop 58gkbg56 commit52

drop 899hbg78 commit53

Vos commits ont été supprimés et votre branche est désormais propre.

**Supprimez les branches non suivies**

Maintenant que vous savez comment nettoyer vos branches à l'aide Git rebase, il est parfois très utile de tout simplement les **supprimer**.

Prenons l'exemple des arbres, il y a des branches qui sont harmonieuses et que les paysagistes vont laisser, et d'autres qui ne sont pas jolies ou qui n'ont pas d'utilité et que l'on va tailler ! :)

Avec Git, on supprime les branches non suivies avec :

git branch -d brancheTest

**Squash ? Débarrassez-vous des commits à tout va !**

Ahah ! Du squash ? Nous ne parlons bien évidemment pas du sport de raquettes ici. :soleil:  Squash est encore une de ces super fonctionnalités du rebase interactif ! Un squash est un regroupement de plusieurs commits. Le but est de les fusionner en un seul pour avoir un historique Git plus propre. Il est notamment conseillé de le faire dans le cadre des envois sur le dépôt à distance, pour simplifier la relecture des modifications effectuées.

 Non seulement Squash va fusionner vos modifications, mais il va aussi fusionner vos messages de commit !

Mais alors, comment l'utiliser ? il s'utilise comme le reste des commandes du rebase interactif. On va aller sélectionner tous les commits nous intéressant, et on va leur appliquer la commande Squash !

git rebase -i HEAD~3

On leur applique la commande Squash :

pick 57dcsd58 Création du formulaire

squash 58gkbg56 Design Formulaire

squash 899hbg78 Correction du formulaire

Concrètement, on dit à GIT de se baser sur le premier commit et on lui applique tous les suivants pour n’en faire qu’un seul.

Lorsque l’on valide le squash, Git va réappliquer les commits dans le même ordre dans lequel ils ont été configurés juste avant. On met alors un message de commit qui concerne le regroupement de nos commits.

**Trouvez l’origine d’un bug avec Git bisect**

On ne va pas se mentir, lorsque l'on développe, on perd beaucoup de temps à déboguer notre application. :euh: C'est toujours long et fastidieux de retrouver ce qui a pu provoquer le bug en question !

Git se propose de réduire ce fameux temps de recherche grâce à Git bisect.

Le principe est plutôt simple. Vous indiquez à Git que **vous cherchez un bug**, **il se déplace de commit en commit,** vous testez la version et vous lui dites si le bug est présent dans le commit courant ou pas.

**Le but est de retrouver le premier commit où le bug est apparu.** Grâce à cela, vous saurez que l'une des modifications faites dans ce commit a causé le bug. ;) Il faut bien entendu avoir réalisé des commits réguliers. :) Si vous n'avez pas fait de commit depuis deux semaines, passez votre chemin ! Il faudra chercher le bug à la main.

D'où l'intérêt de tout petits commits : plus vos commits seront petits, plus l'utilisation de  git bisect  sera utile, puisqu'elle permettra de localiser plus précisément où est situé le bug dans votre code. :D

On commence par la commande start :

git bisect start [bad] [good]

Au lieu de *bad*, vous devrez mettre le hash d'un commit où le bug est présent. À la place de *good*, vous devrez mettre le hash d'un commit où le bug n'était pas présent !

Git va alors naviguer dans chacun des commits entre les deux, dans le but de trouver le premier commit où le bug est apparu.

Git va se déplacer sur chaque commit et vous allez devoir, pour chacun de ces commits, lui indiquer si le commit est **good** ou **bad**.  :)

Si le commit ne présente pas le bug :

git bisect good

Si le commit présente le bug :

git bisect bad

Une fois chaque commit vérifié, Git va vous indiquer le commit qui a provoqué le bug. Il va l'indiquer de cette manière :

fvsd54g5s5d4g5f34g5dfg47df578q9qdff6 is first bad commit

commit fvsd54g5s5d4g5f34g5dfg47df578q9qdff6

Author: Moi <Moi@example.com>

Date: Tue mar 27 16:28:38 2019 -0800

Add fonctionnality AB

Il ne vous reste plus qu'a trouver dans vos modifications, la modification qui a engendré le problème. :)

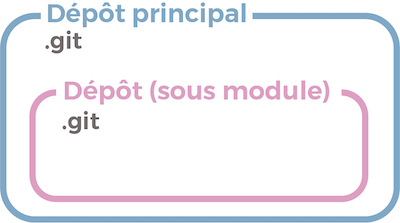
*Passons à l'intégration d'autres dépôts !*

**Intégrez les dépôts d’autres personnes dans le vôtre**

**Définissez les sous-arborescences et les sous-modules**

Les sous-modules et les sous-arborescences ? Mais qu'est-ce que c'est ?

Les sous-modules reposent sur l'imbrication de dépôts : vous avez des dépôts… dans des dépôts.



Sous-modules

Concernant les sous-arborescences, il n'y a pas de dépôts imbriqués : on n'a qu'un dépôt, le conteneur. Les sous-arborescences sont plutôt un concept.

Commençons par les**sous-modules**. N'avez-vous jamais eu envie d'inclure un projet tiers au sein du projet sur lequel vous étiez en train de travailler ? Je ne parle pas d'une simple dépendance tierce, mais d'un module ou d'une bibliothèque que vous développez simultanément.

Admettons que vous ayez à développer deux projets totalement distincts, mais qui vont à un moment devoir se retrouver.

C'est un cas typique d'utilisation des sous-modules Git, car **ils vont vous permettre d'inclure un autre dépôt Git au sein de votre projet actuel**. Il vous sera alors possible de gérer vos commits séparément pour chacun des dépôts.

git submodule add https://github.com/etudiantOC/ProjetSubModule dossier/destination

Cette commande va ajouter à notre dépôt courant le projet ProjetSubModule, comme sous-module dans le dossier Dossier/Destination.

Vous noterez également qu'au travers de cette opération, Git a ajouté un nouveau fichier de configuration nommé .gitmodules contenant la description des sous-modules utilisés par le projet.

Maintenant, voyons la différence avec **les sous-arborescences**.  Si vous avez déjà développé des projets, vous avez sûrement déjà dû vous dire qu'il vous serait utile de garder une fonctionnalité pour la réutiliser dans d'autres projets. Il pourrait être très facile de juste copier-coller les fichiers, et de les remettre plus tard dans les autres projets. Cependant, vous perdriez tout l'historique sur ces fonctionnalités. Heureusement, il existe les sous-arborescences Git !

**Git subtree** va vous permettre de **créer un nouvel arbre de commits** pour un sous-dossier de votre dépôt Git. Autrement dit, Git subtree régénère l’historique d’un dossier.

git subtree push -P monRépertoire git@mon-serveur-git:group/projet.git master

Votre répertoire va être pushé sur votre nouveau dépôt distant sur la branche master. :)

Vous savez maintenant modifier et nettoyer vos branches et vous connaissez les sous-modules. Terminons cette partie par un quiz ! Rendez-vous ensuite dans la partie suivante ! :)

# Gérez plusieurs versions avec Git

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Gérer plusieurs versions sur GitHub

### Question 1

**Quels sont les trois principaux types d'objets de Git ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Blob

* + 

Staging

* + 

Tree

* + 

File

* + 

Commit

*Les trois principaux objets de Git sont le blob, le commit et le tree. Voir le chapitre "Identifier la structure de fichier de Git".*

### Question 2

**Quel est l'identifiant SHA-1 de "Openclassrooms" ?**

* + 

jth54986f26aa05bae5c1f20dce41dd4c6d5d11f

* + 

fab96207f26aa05bae5c1f20dce41dd4c6d5d00f

* + 

rtg56863f26aa05bae5c1f20dce41dd4c6d5d33f

*Allez sur le site*[*https://www.sha1.fr/*](https://www.sha1.fr/)*et tapez O*penclassrooms*, vous obtenez alors son hash Sha-1.*

### Question 3

**Sur notre dossier Git de test, créons une nouvelle fonctionnalité "fctMerge". Maintenant, comment faites-vous pour la fusionner avec la commande  merge  ?**

* + 

git checkout fctMerge  
git status  
git checkout master  
git merge fctMerge

* + 

git merge fctMerge

* + 

git merge master fctMerge

*On bascule dans un premier temps sur notre branche, on vérifie que l'on est bien dessus et on exécute le merge.*

### Question 4

**Quel est le résultat du merge ci-dessus ?**

* + 

Le merge fusionne la branche master et la branche fctMerge mais ne crée pas de nouveau commit.

* + 

Le merge fusionne la branche master et la branche fctMerge et crée un nouveau commit.

*Un merge crée toujours un nouveau commit. On appelle ce commit, le "commit de merge".*

### Question 5

**Quelle est la différence entre  git pull  et  git fetch  ?**

* + 

Git pull récupère les données sur le serveur alors que Git fetch les envoie sur les serveurs.

* + 

Git pull va fusionner les modifications sur la branche locale alors que Git fetch, non.

* + 

Git fetch va fusionner les modifications sur la branche locale alors que Git pull, non.

*Git fetch, contrairement à Git pull, va aller chercher les modifications sur le dépôt distant mais ne va pas les fusionner avec nos modifications locales. Git isole le contenu récupéré en tant que contenu local existant, cela n'a absolument aucun effet sur votre travail de développement local. La commande Git fetch va récupérer toutes les données des commits effectués sur la branche courante qui n'existent pas encore dans votre version en local. Ces données seront stockées dans le répertoire de travail local, mais ne seront pas fusionnées avec votre branche locale.*

### Question 6

**Comment réaliser un rebase interactif ?**

* + 

git Rebase -interactif

* + 

git Rebase -hard

* + 

git Rebase -i

*Voir le cours sur le rebase : "Modifier ses branches avec Rebase"*

### Question 7

**Sur votre dossier Git de tests, nous réalisons trois commits successifs.**

**Git log nous donne :**

24gfbg56 commit3  
35hg58g1 commit2  
76frgf83 commit1

**Nous voulons ensuite changer l'ordre de ces commits en intervertissant le premier et le dernier commits.**

**Que devons-nous faire ?**

* + 

pick 24gfbg56   
pick 35hg58g1   
pick 76frgf83

* + 

pick 76frgf83   
pick 35hg58g1   
pick 24gfbg56

* + 

pick 35hg58g1   
pick 76frgf83   
pick 24gfbg56

*Vous utilisez la commande Pick pour indiquer à Git dans quel ordre vous les voulez.*

## Travaillez en équipe en utilisant un workflow

### Qu'est-ce que le flux de travail ou workflow?

Un **workflow Git** est une recommandation sur la façon d'utiliser Git pour effectuer un travail de manière **cohérente** et **productive**. Plusieurs workflows connus sont couramment utilisés par les développeurs pour des projets personnels, mais aussi par les entreprises. Le plus connu est **GitFlow**, que nous verrons dans le chapitre suivant. :-°

Les workflows Git encouragent les utilisateurs à exploiter Git de manière efficace et cohérente.

Git offre beaucoup de **flexibilité** dans la manière dont les utilisateurs gèrent les changements. Étant donné que Git met l’accent sur la flexibilité, il n’existe pas de processus normalisé pour interagir avec Git.

Lorsque vous travaillez avec une équipe sur un projet géré par Git, il est important de vous assurer que l'équipe est tout à fait d'accord sur la manière dont le flux de modifications sera appliqué. o_O

Pour que l'équipe soit sur la même page, un workflow Git convenu doit être développé ou sélectionné. Il existe plusieurs flux de travail Git connus qui conviendront peut-être à votre équipe. Ici, nous allons discuter de certaines de ces options de workflow.

Ces workflows sont conçus pour servir de références et ne constituent pas des règles concrètes.

Comme nous vous l'avons dit juste au-dessus, il existe plusieurs workflows déjà "standardisés".

 Parmi les plus utilisés, vous trouverez le **workflow de duplication** (fork).

Le workflow de duplication (fork) est fondamentalement différent des autres workflows Git populaires.

Au lieu d'utiliser un dépôt unique côté serveur pour agir en tant que base de code « centrale », ce workflow fournit un dépôt côté serveur à chaque développeur. Par conséquent, chaque contributeur dispose non pas d'un, mais de deux dépôts Git : un privé en local et un public côté serveur.

Vous le rencontrerez souvent dans de gros projets open source ! :D

Un autre workflow populaire est le**workflow de fonctionnalités.** Le principe de base du workflow de branche par fonctionnalité est que chaque fonctionnalité est développée dans une branche prévue à cet effet plutôt que dans la branche master. Grâce à cette encapsulation, plusieurs développeurs peuvent travailler aisément sur une même fonctionnalité sans modifier la base de code principale. Il sera utilisé essentiellement dans le cadre de l'intégration continue.

Et enfin, le workflow le plus connu : GitFlow. :)

GitFlow est parfaitement adapté aux projets avec un cycle de livraison planifié. Maintenant, voyons-le un peu de plus près !

### Appréhendez le fonctionnement de GitFlow

Utiliser Git est une chose, l'utiliser correctement en est une autre !  Beaucoup de personnes ont donc réfléchi à différentes manières d'utiliser au mieux Git. :soleil:

Vincent Driessen a alors créé le GitFlow.

**GitFlow est une méthode**, une architecture Git permettant de**séparer au maximum le travail** et de toucher le moins possible à la branche master. Cette méthode représente donc une architecture en branches. GitFlow est une des architectures les plus connues. GitFlow n'ajoute aucun concept ni aucune commande, il attribue plutôt **des rôles très spécifiques aux différentes branches** et définit comment et quand elles doivent interagir. Pour utiliser GitFlow, il va falloir dans un premier temps l'installer. Une fois GitFlow installé, vous pouvez l'utiliser dans votre projet en exécutant la commande  git flow init .

La commande Git flow init est une extension de la commande Git init par défaut. Elle ne change rien dans votre dépôt et ne fait que créer des branches pour vous.

GitFlow va définir dans un premier temps deux branches distinctes dans lesquelles les développeurs n'auront aucunement le droit de développer. :pirate:

La branche **master** est la branche qui va correspondre à notre environnement de production. Il est donc logique que l'on ne puisse y pousser nos modifications directement.

La branche**Develop** centralise toutes les nouvelles fonctionnalités qui seront livrées dans la prochaine version. Ici, il va falloir se forcer à ne pas y faire de modifications directement. Dans le cadre d'un gros projet, la branche Develop correspond en général à notre environnement de recette. L'environnement de recette est une copie du projet qui est censée partir en production et où les testeurs vont réaliser une batterie de tests afin d'être sûrs de ne pas envoyer de bugs en production.  
La branche master stocke l'historique officiel des versions, et la branche Develop sert de branche d'intégration pour les fonctionnalités.

Il peut être utile de donner un numéro de version à chaque commit sur la branche master.

### Que signifient tous les types de branches et quelles sont leurs utilités ?

GitFlow ne se contente bien sûr pas uniquement de deux branches. À ces deux branches vont venir s'ajouter trois autres types de branches :

* feature ;
* hotfix ;
* release.

Les branches**Feature** permettent de commencer à travailler sur une nouvelle fonctionnalité. La branche Feature est créée à partir de la branche Develop. Vous devrez créer pour chaque nouvelle fonctionnalité une branche Feature ! Lorsque nous avons fini de développer notre nouvelle fonctionnalité, il faudra alors la commiter puis la fusionner sur la branche Develop.

La branche **Hotfix**, quant à elle, va permettre de corriger un bug en production. En ce sens, elle sera créée à partir de la branche master, car c'est la branche master qui correspond à l'environnement de production. Une fois la branche Hotfix terminée, elle est mergée dans la branche Develop et dans la branche master.

La branche Hotfix ne doit être utilisée que pour de minimes corrections !

La branche**Release** est créée à partir de la branche Develop en cas de livraison en production imminente. En effet, dans le cadre d'un projet un peu plus conséquent, il y a souvent plusieurs versions. Une fois que toutes les fonctionnalités d'une version ont été créées, c'est à ce moment que nous devons créer une branche Release. Elle va permettre de réaliser nos tests alors que d'autres développeurs pourront commencer à travailler sur la version suivante. Lorsque la branche Release est terminée, nous devons la merger dans la branche Develop et dans la branche master.

Maintenant que vous savez tout de GitFlow, nous allons voir comment l'implémenter simplement.

Nous allons dans un premier temps, initialiser notre dépôt avec GitFlow.

git flow init

Vous pouvez maintenant faire un  Git branch  et vous verrez que vous avez une branche master et une branche Develop et que vous vous trouvez sur la branche Develop.

Pour commencer à travailler sur notre projet, nous allons devoir créer une feature. Étant au tout début du projet, notre feature va correspondre à la base notre projet. Elle s’appellera donc main.

git flow feature start main

Avec l'exécution de cette commande, Git créera la branche Feature main et nous basculera dessus. Vous pouvez maintenant commencer vos développements.

Vous connaissez maintenant l'essentiel du GitFlow, découvrons à présent les outils pour améliorer Git !

**Améliorez Git avec des outils supplémentaires**

**Exploitez au mieux la marketplace de GitHub**

Maintenant que vous êtes expert Git et GitHub, et que la gestion de dépôt n'a plus de secret pour vous, que diriez-vous de découvrir **quelques outils** qui vous faciliteront la vie ? :magicien:  C'est en 2017 que la marketplace de GitHub a vu le jour !

Au début, elle était divisée en 5 catégories : qualité du code, analyse du code, intégration continue, suivi et gestion de projet.

Maintenant, beaucoup de nouvelles catégories ont vu le jour pour notre plus grand bonheur !

L’idée de la marketplace est de proposer des outils utilisables rapidement par les développeurs sur un unique compte.

**Quelques outils et leurs fonctionnalités**

Nous allons maintenant voir quelques-uns des outils les plus connus et les plus utilisés de la marketplace de GitHub.

Dans le domaine de la **sécurité**, nous pouvons citer WhiteSource Bolt !

**WhiteSource Bolt** for GitHub est une application gratuite, qui analyse en permanence tous vos dépôts, détecte les vulnérabilités des composants open source et apporte des correctifs. Il prend en charge les référentiels privés et publics ! :soleil:

Dans le domaine de la **gestion de projets**, nous pouvons citer ZenHub.

**ZenHub** est le seul outil de gestion de projet qui s'intègre de manière native dans l'interface utilisateur de GitHub. ZenHub est un outil de gestion de projet agile fonctionnant par sprint et générant des rapports assez poussés.

Dans le domaine de**l'intégration continue**, nous pouvons bien sûr citer Travis CI, qui est la référence en la matière.

**Travis CI** permet à votre équipe de tester et déployer vos applications en toute confiance. Très polyvalent, il s'adapte aux petits comme aux grands projets !

Il existe bien sûr de nombreux autres outils sur la marketplace. :)

À vous de trouver ceux qui vous conviennent le mieux !

**Comparez deux fichiers avec une interface graphique**

Vous connaissez maintenant de nombreux outils pour vous faciliter la vie, mais il vous en manque un essentiel ! Le comparateur de code ! o_O

Ce petit outil est indispensable en développement ! Il vous sera très utile pour comparer deux versions de votre fichier.

Dans les plus connus, nous avons actuellement **WinMerge** et **Meld**.

Les deux ont exactement le même but, comparer simplement deux fichiers en indiquant les zones où votre code est différent !

En plus de vous indiquer les différences entre vos deux fichiers, vous allez pouvoir les fusionner de façon intelligente. Pour chaque ligne différente, l'outil de comparaison vous demandera quelle version vous souhaitez conserver. Il est donc indispensable en cas de conflit dans Git.

Par exemple, vous avez travaillé sur une fonctionnalité et votre collègue aussi. Au moment du push, vous avez un conflit sous Git, car les lignes que vous avez modifiées ont aussi été modifiées par votre collègue. Grâce aux outils de comparaison, vous n'écrasez pas le code de l'autre bêtement !

*Vous connaissez maintenant différents outils pour faciliter votre travail. Découvrez GitLab dans le chapitre suivant !*

## Utilisez le GitLab intégration continue (IC)

Git, GitHub, GitLab ? Mais quelle est la différence ?

Développeur aguerri, jeune padawan ou simple curieux du web, vous avez probablement déjà entendu parler de**GitLab**. Facilement reconnaissable à sa petite tête de renard, c’est un des outils incontournables en matière de **gestion de projets web**.

### Qu'est-ce que GitLab ?

GitLab est une plateforme permettant d'**héberger et gérer des projets web**. GitLab est considérée comme la plateforme des développeurs modernes ! Comparé à GitHub, la palette fonctionnelle de GitLab se veut nettement plus large. Au-delà de la compilation et de la gestion de dépôts de code source sur lesquels se concentre le premier, GitLab s'étend au **test logiciel**, au**packaging** d'applications, à**l'intégration** et au **déploiement continus**, à la **configuration**, jusqu'au **monitoring** et à la **sécurité applicative**.

Ce qu’il faut retenir sur GitLab :

* il permet d’héberger les projets web et la gestion de versions des codes sources ;
* il permet la gestion de tout le processus de développement ;
* il permet une collaboration simple ;
* il est open source et collaboratif ;
* c’est gratuit ;
* c’est aussi une solution pour les entreprises.

GitLab est donc une réelle alternative à GitHub ! En effet, GitLab CI est un système d'intégration continue très puissant et gratuit.

### Où l'IC se situe-t-elle dans le flux de travail et que fait-il ?

**L'intégration continue**, ou IC, permet d'intégrer le code de votre équipe dans un référentiel partagé. Les développeurs partagent leur nouveau code dans une demande de fusion (extraction), ce qui déclenche la création, le test et la validation par un**pipeline**, avant la fusion des modifications dans votre référentiel.

**GitLab CI/CD** va vous permettre d’automatiser les builds, les tests, les déploiements, etc.  de vos applications. L’ensemble de vos tâches peut être divisé en étapes et l’ensemble de vos tâches et étapes constituent un pipeline.

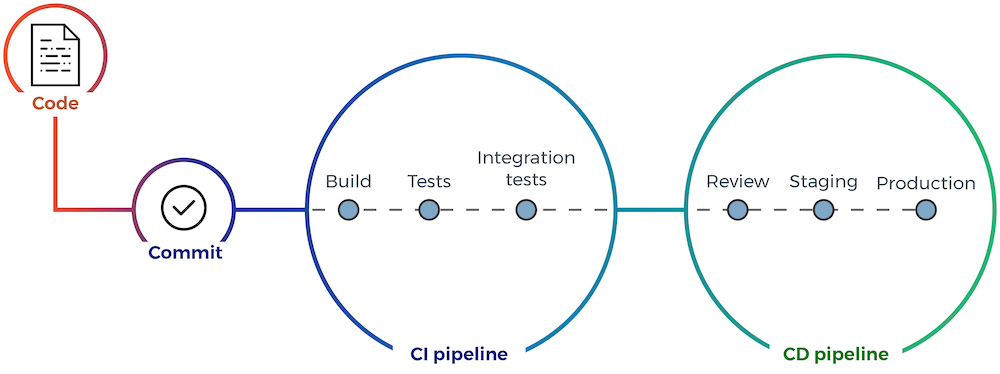
Les outils d'intégration continue - CI, et de déploiement continu - CD, permettent d'automatiser beaucoup de processus qui prennent du temps, et sont sujets à erreur humaine.

Cela permet aux développeurs de se concentrer sur la création de logiciels !

GitLab est un service qui **stocke les dépôts** en utilisant le backbone Git, tout comme GitHub.

Il dispose d'un pipeline de CI-CD qui permet à un développeur de configurer un script automatisé pour les tests en continu, les demandes de pull, les builds, les différents environnements de test et bien plus encore !

**La CI permet une intégration de code plus rapide et moins déstabilisante**, puisque le code est intégré au fur et à mesure, à un rythme plus soutenu qu'avec d'autres approches de développement.



L'intégration continue

### Quels sont les autres outils de CI ?

Jetons un œil aux autres outils de CI.

L'un des principaux concurrents de GitLab est Jenkins.

Jenkins

**Jenkins** est l'un des premiers **serveurs d'intégration continue** open source, et reste l'option la plus couramment utilisée aujourd'hui.

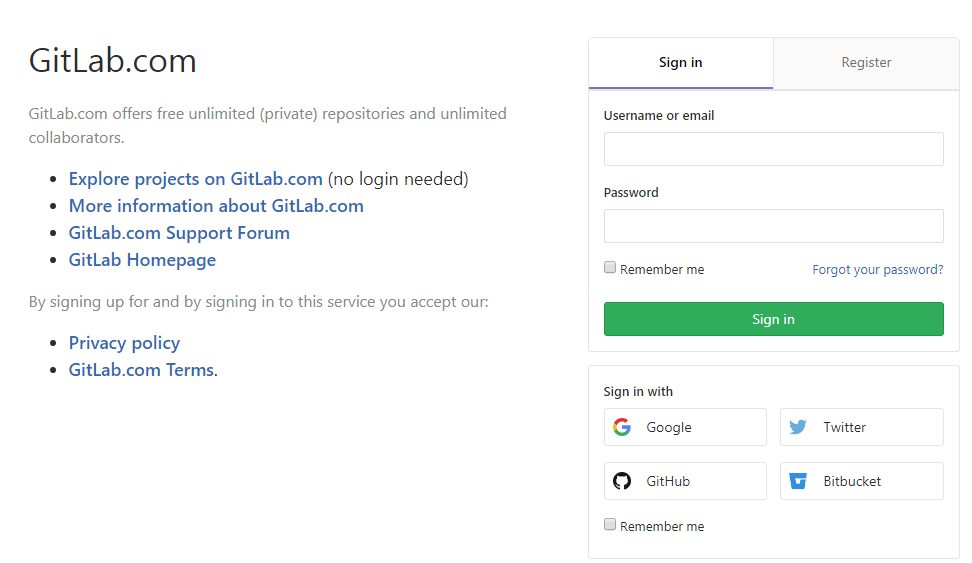
**Avantages :**

* logiciel gratuit ;
* plus de 1 000 plugins sont disponibles ;
* vous pouvez créer un plugin si celui que vous désirez n'existe pas ;
* vous pouvez également partager ce plugin ;
* logiciel facile à installer.

Nous pouvons évoquer aussi Buildbot, Drone, Concourse, mais Jenkins et GitLab restent les maîtres en la matière. :)

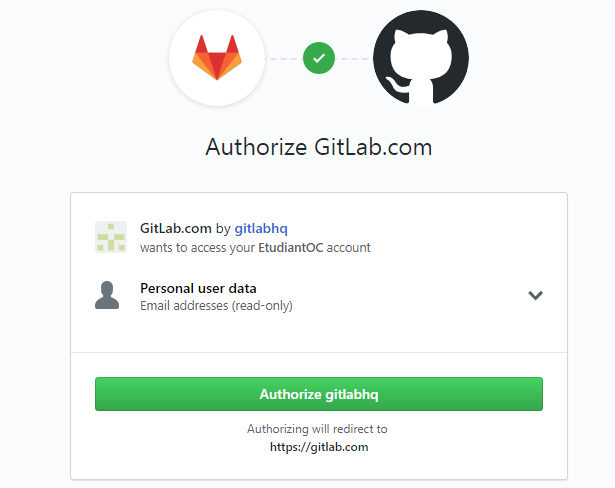
### Fonctionnement de GitLab CI

Afin d'utiliser l'intégration continue de GitLab, il vous faudra créer un compte GitLab [ici](https://gitlab.com/users/sign_in).



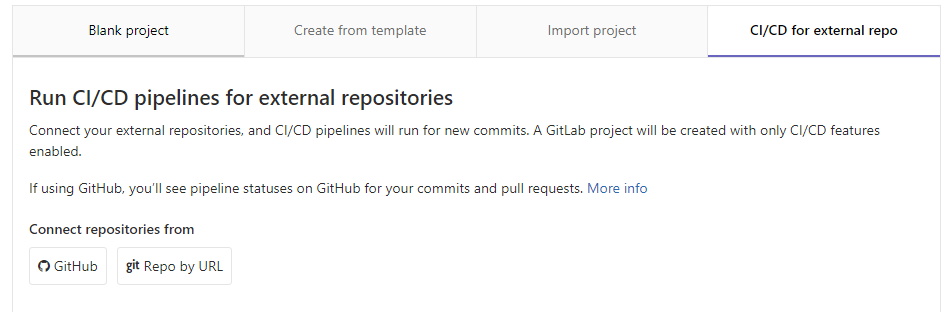
GitLabCI

Pour la création du compte, vous pourrez utiliser GitHub comme connecteur. :)



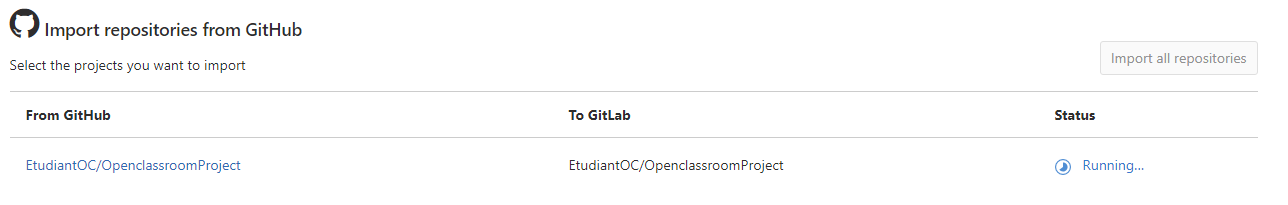
Autorisation

Une fois le compte créé, vous pourrez soit créer un projet dans GitLab, soit uniquement utiliser le pipeline CI/CD sur un dépôt externe Git ou GitHub.



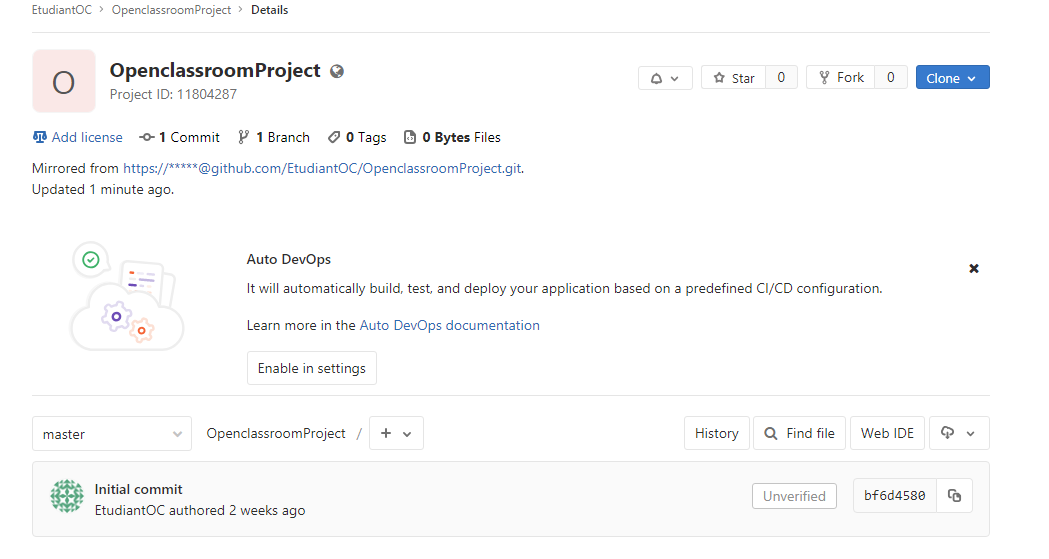
Pipeline

Vous pourrez ensuite importer le dépôt dont vous avez besoin.



Importation

Vous obtiendrez finalement la vue de votre projet.



Projet

Pour configurer l"intégration continue, nous devons, dans un premier temps, créer dans notre dépôt un fichier .gitlab-ci.yml  qui va lister les différentes tâches à effectuer. Lorsque GitLab détecte un fichier   .gitlab-ci.yml, il va automatiquement essayer d'exécuter les étapes avec des runners.

Dans le fichier .yml, nous pourrons mettre les étapes de test, de construction, de déploiement et de notification.

GitLab est un outil merveilleux, n'est-ce pas ? Apprenons à gérer les demandes de pull dans le prochain chapitre !

## Gérez les demandes de pull

### Pourquoi est-il important de gérer les demandes de pull régulièrement pour maintenir un projet actif ?

Il est très important de gérer les demandes de pull request régulièrement, si vous souhaitez garder une communauté active et avoir un projet vivant.

En effet, imaginez un peu un ami qui vous propose toutes les semaines de vous inviter à boire un café, à venir manger chez lui, à aller faire une sortie entre amis. Toutes les semaines vous l'ignorez, soit par manque de temps, soit car vous n'avez pas envie. Petit à petit, il ne vous sollicitera plus et ne s’intéressera plus à vous, même s'il vous apprécie beaucoup !

Eh bien, un projet GitHub c'est presque la même chose, il reste intéressant aux yeux des contributeurs uniquement s'il reste **actif** et que les**demandes de pull sont traitées**.

Imaginez que vous ayez créé une super fonctionnalité pour un projet open source sur GitHub, que vous ayez passé beaucoup de temps à la développer. Vous faites une**pull request** et personne ne la traite. Vous avez donc travaillé pour rien ! Et ce n'est jamais agréable d'être ignoré. ;)

### Étiquetez votre projet avec des badges

Les badges permettent de garder le projet à jour et indiquent une certaine qualité. Les badges peuvent être utilisés pour tout un tas de choses.

Ils peuvent être utiles :

* dans le cas de **l'intégration continue**, par exemple. "Build passing" indique que les tests du projet sont concluants et que le projet est fonctionnel ;



* dans le cadre de la**sécurité**. Ce badge indique qu'aucune vulnérabilité n'a été détectée sur votre projet ;



* pour indiquer que **votre code a été testé**. Ce badge indique le pourcentage de votre code qui fait l'objet de tests. Il est très utile de créer des scénarios de tests, des tests automatiques, des tests de régression pour afficher un 100 % !



* pour indiquer aux contributeurs **la version de votre module**. Cela n'est pas obligatoire, mais peut être apprécié ;



* pour indiquer la **maintenabilité** d'un projet.

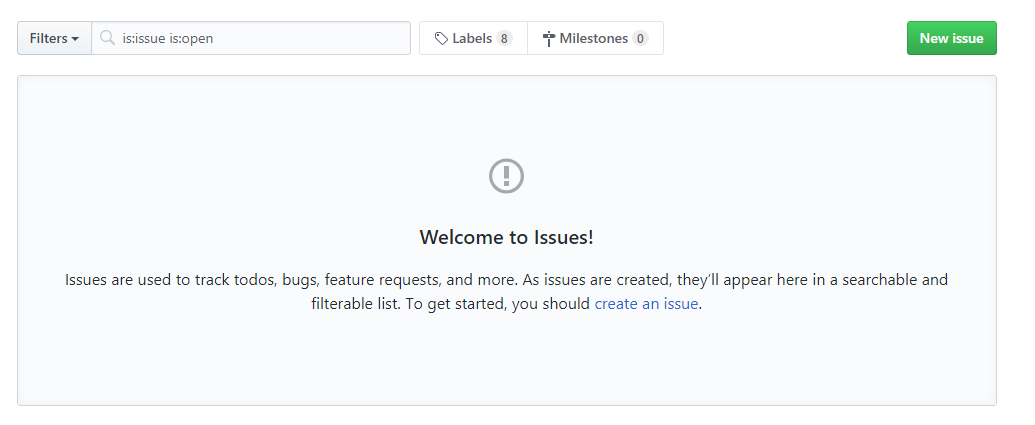


Le score obtenu reflète la qualité du code d'un projet mesuré en fonction de plusieurs facteurs, notamment la **complexité**/**simplicité**, la **lisibilité**, la **maintenabilité**, la **répétition** et le **nombre de lignes par fichier**.

Il y a encore de nombreux autres badges, mais nous vous avons vu les principaux !

### Gérez les corrections de bugs avec la console GitHub

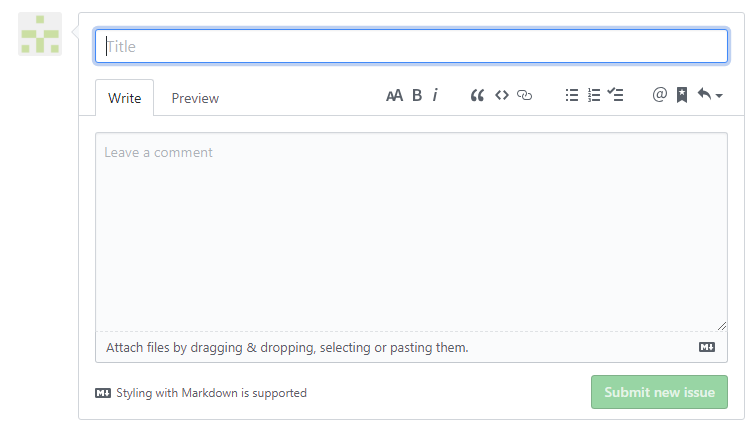
Dans GitHub, il y a un onglet très pratique qui permet de maintenir la gestion des bugs, l'onglet "Issues".



Onglet Issues

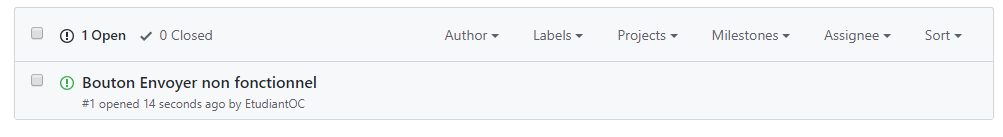
Les "issues" permettent aux utilisateurs et aux contributeurs d'indiquer des bugs afin qu'ils puissent être corrigés par vous, mais aussi par les autres contributeurs.

Afin de créer un nouveau report de bug, nous devons cliquer sur "New issue".



New issue

GitHub vous demande alors d'indiquer un nom pour l'issue et un descriptif. Attention, une issue doit correspondre à un bug unique et non à plusieurs bugs.



Exemple issue

Il est possible ensuite de lui ajouter un badge, pour indiquer si cela est bien un bug, si on ne le reproduit pas, s'il existe déjà une issue pour ce bug, etc.

Il est aussi possible d'assigner le bug à quelqu'un pour lui dire de le résoudre. Par exemple, vous travaillez en équipe. Vous êtes sur la partie frontend et votre collègue sur la partie backend ; si l'envoi de mail ne fonctionne plus, c'est à votre collègue de s'en charger. Vous pouvez donc lui **assigner l'issue**. Si au contraire, un utilisateur rencontre un bug d'affichage, vous pouvez vous assigner le bug.

Une fois le bug résolu, il est très important de **passer l'issue au statut "clos"**. Cela évite que quelqu'un travaille dessus inutilement, et puis c'est toujours mieux d'avoir le minimum de bugs sur votre projet. :)

### Quand devriez-vous fusionner dans la branche principale ?

Vous vous demandez sûrement comment être sûr de ne pas fusionner n'importe quoi sur votre branche principale.

Tout d'abord, vous devez bien entendu regarder ce que l'on vous a transmis.

Ensuite, sur des projets plus conséquents, vous pouvez indiquer un nombre minimum d'approbations (par les autres contributeurs), avant d'avoir la possibilité de fusionner.

 C'est un gage de sécurité et de qualité !

Nous avons désormais fait le tour des commandes de base de Git et du fonctionnement de GitHub. Faisons un petit récapitulatif de nos connaissances

## Récapitulez ce que vous avez appris

Vous êtes arrivé à la fin de ce cours ! Vous devez maintenant être capable de :

* utiliser les commandes de base de Git ;
* corriger les erreurs courantes sur GitHub ;
* gérer plusieurs versions sur GitHub ;
* collaborer grâce à GitHub en utilisant les workflows.

N'oubliez pas de réaliser les exercices à la fin de chaque partie pour valider ces compétences. C'était un plaisir de vous accompagner tout au long de ce cours.

Je vous souhaite une très bonne continuation dans tous vos projets !

# Gérez un travail d’équipe

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Collaborer grâce à GitHub en utilisant les workflows

### Question 1

**Quels sont les types de branches d'un GitFlow ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Master

* + 

Feature

* + 

Hotfix

* + 

Forecast

* + 

Release

* + 

Develop

* + 

Delayed

*La branche****master****est la branche qui va correspondre à notre environnement de production. Il est donc logique que l'on ne puisse y pousser nos modifications directement.*

*La branche****develop****centralise toutes les nouvelles fonctionnalités qui seront livrées dans la prochaine version. Ici, il va falloir se forcer à ne pas y faire de modifications directement. Dans le cadre d'un gros projet, la branche Develop correspond en général à notre environnement de recette.*

*Les branches****feature****permettent de commencer à travailler sur une nouvelle fonctionnalité. La branche Feature est créée à partir de la branche Develop. Vous devrez créer pour chaque nouvelle fonctionnalité une branche Feature ! Lorsque nous avons fini de développer notre nouvelle fonctionnalité, il faudra alors la commiter puis la fusionner sur la branche Develop.*

*La branche****hotfix****, quant à elle, va permettre de corriger un bug en production. En ce sens, elle sera créée à partir de la branche master, car c'est la branche master qui correspond à l'environnement de production. Une fois la branche Hotfix terminée, elle est mergée dans la branche Develop et dans la branche master.*

*La branche****release****est créée à partir de la branche Develop en cas de livraison en production imminente. En effet, dans le cadre d'un projet un peu plus conséquent, il y a souvent plusieurs versions. Une fois que toutes les fonctionnalités d'une version ont été créées, c'est à ce moment que nous devons créer une branche Release. Elle va permettre de réaliser nos tests alors que d'autres développeurs pourront commencer à travailler sur la version suivante. Lorsque la branche Release est terminée, nous devons la merger dans la branche Develop et dans la branche master.*

### Question 2

**Comment initialiser un GitFlow ?**

* + 

git init

* + 

git flow init

* + 

git init  
git flow init

*La commande*git flow init*est une extension de la commande*git init*par défaut. Elle ne change rien dans votre dépôt et ne fait que créer des branches pour vous.*

### Question 3

**Comment créer une nouvelle feature ?**

* + 

git init feature maFonctionnalité

* + 

git flow feature start maFonctionnalité

* + 

git flow feature maFonctionnalité

*Pour commencer à travailler sur un projet ou créer une nouvelle fonctionnalité, nous allons devoir créer une feature. Elle s’appellera donc maFonctionnalité et pour ce faire, nous devrons utiliser la commande*git flow feature start maFonctionnalité.

### Question 4

**Quels sont les avantages de l'intégration continue et du déploiement continu ?**

*Attention, plusieurs réponses sont possibles.*

* + 

L'automatisation des tests, les builds, les déploiements

* + 

Améliorer la collaboration entre les développeurs

* + 

Anticiper les bugs

*Les outils d'intégration continue (CI) et de déploiement continu (CD) permettent d'automatiser beaucoup de processus qui prennent du temps et sont sujets à erreur humaine, comme l'automatisation des tests, les builds et les déploiements. Ils permettent aussi de faciliter la collaboration.*

### Question 5

**Que comprend le CI pipeline ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Les tests

* + 

Le staging

* + 

La production

* + 

Les tests d'intégration

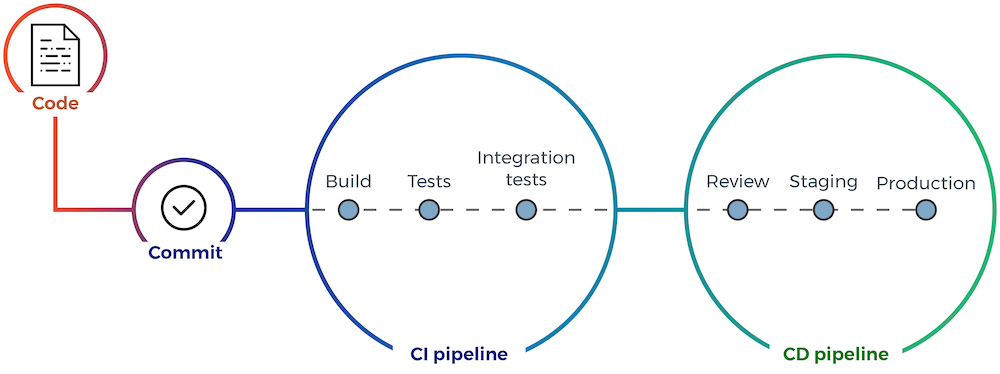
* + 

Le build

* + 

Le review

*Le CI pipeline comprend : build, tests, integration tests.*

*L'intégration continue*

### Question 6

**Que comprend le CD pipeline ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Les tests

* + 

Le staging

* + 

La production

* + 

Les tests d'intégration

* + 

Le build

* + 

Le review

*Le CD pipeline comprend : review, staging, production (cf. schéma ci-dessus).*

### Question 7

**Quelle est l'extension du fichier qui indique à GitLab les différentes tâches à effectuer pour l'intégration continue ?**

* + 

.Xml

* + 

.Yml

* + 

.Txt

* + 

.Xaml

*C'est le fichier  .gitlab-ci.yml, dont l'extension est le  .yml, qui va indiquer les différentes tâches pour le CI.*

### Question 8

**À quoi servent les badges ?**

*Attention, plusieurs réponses sont possibles.*

* + 

À indiquer que votre code est sécurisé

* + 

À indiquer que votre code est testé

* + 

À indiquer la version de votre package

* + 

À indiquer la maintenabilité

* + 

À indiquer que votre code est public

*Les badges permettent de garder le projet à jour et indiquent une certaine qualité. Les badges peuvent être utilisés pour tout un tas de choses (sécurité, tests, version, maintenabilité...).*