ASCALEA
STRUCTURE BALANCE IMPACT

Git & GitHub

De zéro à la collaboration maîtrisée

Guide pratique pour comprendre le versionnement, gérer ses projets et collaborer efficacement

Git & GitHub — De zéro à la collaboration maîtrisée

Guide pratique pour comprendre le versionnement, gérer ses projets et collaborer efficacement

ASCALEA — MYGASPOZ

Table des matières

Ce cours propose une première approche structurée de Git et de GitHub pour des personnes qui n'ont pas encore d'expérience avec le suivi de versions ou le travail collaboratif sur des fichiers. Il privilégie une progression graduelle : comprendre les idées avant de mémoriser les commandes, puis appliquer ces idées dans un projet simple.

Les termes et les commandes de Git sont majoritairement en anglais. Le cours les emploie donc tels qu'ils apparaissent dans les outils, tout en les expliquant en français. L'objectif n'est pas de connaître toutes les commandes disponibles, mais d'acquérir un mode de travail fiable : savoir où se trouvent ses modifications, les enregistrer avec intention, les retrouver et les partager.

Objectif du cours

À l'issue du parcours, le lecteur doit pouvoir :

- distinguer Git de GitHub et comprendre le rôle respectif de chacun ;
- installer et configurer Git sur son ordinateur ;
- créer un dépôt local pour suivre un projet ;
- examiner l'état des fichiers avant de les enregistrer ;
- sélectionner les modifications à inclure dans un enregistrement ;
- créer des enregistrements d'historique clairs ;
- relier un projet local à GitHub afin de le publier ou de le partager ;
- adopter des bases de sécurité pour son compte GitHub.

Le fil conducteur est simple : un projet existe d'abord dans un dossier sur l'ordinateur. Git permet d'en suivre l'évolution. GitHub permet, entre autres usages, d'héberger et de partager des dépôts en ligne.

> Idée directrice : Git n’est pas un bouton « sauvegarder ». Il permet de constituer un historique explicite et lisible des changements d’un projet.

Public visé et niveau de départ

Ce chapitre s’adresse aux débutants. Aucune connaissance préalable de Git, de GitHub, du développement logiciel ou de la ligne de commande n’est supposée.

Il est toutefois utile de savoir :

- créer et retrouver un dossier sur son ordinateur ;
- créer, modifier et enregistrer un fichier ;
- utiliser un navigateur web ;
- copier et coller du texte ;
- suivre attentivement des instructions écrites.

Les exemples peuvent concerner du code, mais Git ne se limite pas au code. Le mécanisme de suivi de versions peut s’appliquer à différents types de fichiers. Pour apprendre efficacement, il est néanmoins préférable de commencer avec de petits fichiers texte : leurs changements sont plus faciles à observer et à comprendre.

Vocabulaire essentiel

Les mots suivants apparaîtront régulièrement. Ils sont introduits ici pour donner des repères ; leurs usages seront détaillés au fil du cours.

Terme	Sens dans ce cours
Dépôt (repository ou repo)	Espace de travail suivi par Git, qui contient les fichiers du projet et leur historique.
Répertoire de travail	Dossier et fichiers que l'on modifie directement sur son ordinateur.
Historique	Suite des enregistrements qui décrivent l'évolution d'un dépôt.
Modification	Écart entre l'état actuel d'un fichier et son dernier état enregistré dans l'historique.
Zone d'index (staging area)	Zone intermédiaire dans laquelle on prépare précisément les modifications à enregistrer.
Commit	Enregistrement identifié dans l'historique de Git, créé à partir des modifications préparées.
Branche	Ligne de développement indépendante dans un même dépôt.
Dépôt distant (remote repository)	Version d'un dépôt accessible par un service distant, tel que GitHub.
Cloner (clone)	Créer une copie locale d'un dépôt existant.
Synchroniser	Échanger des modifications entre un dépôt local et un dépôt distant.

La documentation Git distingue notamment le répertoire de travail, la zone d'index et le dépôt local. La commande git add sert à ajouter le contenu de fichiers à la zone d'index, tandis que git status permet d'observer l'état des fichiers et des zones de travail [GIT-ADD] [GIT-STATUS].

Ne cherchez pas à retenir tous ces termes immédiatement. Au début, il suffit de retenir le parcours fondamental suivant : modifier, préparer, enregistrer.

Git : l'outil de suivi de versions

Git est un système de contrôle de versions distribué. Il sert à suivre les changements apportés à des fichiers et à conserver un historique de ces changements [GITHUB-ABOUT-GIT-FR] [GIT-OVERVIEW].

Dans la pratique, Git s'exécute sur l'ordinateur du lecteur. Il permet donc de travailler sur un dépôt local, y compris avant toute publication sur Internet. Chaque dépôt dispose de son propre historique local.

Un usage typique suit cette séquence :

1. créer ou modifier un ou plusieurs fichiers dans un dossier ;
2. demander à Git de montrer l'état du dépôt ;
3. préparer les modifications choisies ;
4. créer un commit qui documente cet état du projet.

Cette séparation est importante. Modifier un fichier ne l'ajoute pas automatiquement à l'historique. Préparer une modification ne crée pas encore un commit. Git invite ainsi à vérifier ce qui sera effectivement enregistré.

GitHub : une plateforme autour des dépôts Git

GitHub est une plateforme qui permet notamment d'héberger des dépôts Git et de collaborer autour de ceux-ci. GitHub s'appuie sur Git, mais Git et GitHub ne sont pas le même outil [GITHUB-ABOUT-GIT-FR].

Un compte GitHub permet d'utiliser les fonctionnalités proposées par la plateforme. La création d'un compte est documentée par GitHub [GITHUB-ACCOUNT-CREATION]. Lorsque le compte est utilisé, la protection de ses accès est une responsabilité essentielle : GitHub documente la configuration de l'authentification à deux facteurs ainsi que les méthodes de récupération associées [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY].

Une distinction à retenir

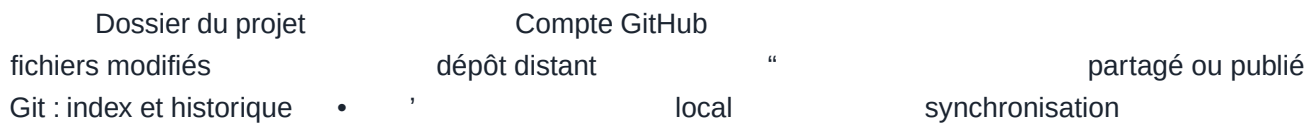
Élément	Git	GitHub
Nature	Logiciel de contrôle de versions	Plateforme de collaboration et d'hébergement de dépôts Git
Lieu principal d'utilisation	Sur l'ordinateur du lecteur	Dans un navigateur web et via des connexions avec les outils locaux
Rôle	Enregistrer et consulter l'historique d'un projet	Partager, héberger et organiser le travail autour de dépôts
Nécessite un compte GitHub ?	Non, pour travailler sur un dépôt local	Oui, pour utiliser un compte sur la plateforme

Il est donc possible d'apprendre les bases de Git sans publier de projet sur GitHub. Inversement, créer un dépôt sur GitHub ne remplace pas l'apprentissage du fonctionnement de Git : les deux compétences sont complémentaires.

Schéma mental : local, Git et GitHub

Le modèle ci-dessous résume les trois espaces à ne pas confondre.

```text Ordinateur du lecteur En ligne
```



Le travail quotidien commence généralement à gauche : les fichiers sont modifiés localement, puis Git enregistre un historique local. La synchronisation avec GitHub intervient lorsqu'un dépôt distant est configuré et que l'on souhaite échanger des changements.

Outils et prérequis techniques

Le cours repose sur deux outils principaux :

1. Git, installé sur l'ordinateur ;
2. un navigateur web pour accéder à GitHub et, si nécessaire, un compte GitHub.

Le site officiel de Git propose des instructions d'installation pour Windows, macOS et Linux [GIT-INSTALL] [GIT-INSTALL-WINDOWS] [GIT-INSTALL-MACOS] [GIT-INSTALL-LINUX]. Après l'installation, une configuration initiale permet notamment de définir l'identité associée aux enregistrements Git ; la documentation officielle présente cette étape [GIT-FIRST-CONFIG] [GIT-CONFIG].

Le cours utilisera principalement le terminal, c'est-à-dire l'interface textuelle dans laquelle les commandes Git sont saisies. Cette approche rend visibles les mécanismes fondamentaux et s'applique quel que soit l'environnement de développement choisi par la suite.

Une application graphique peut aussi faciliter certaines opérations. GitHub Desktop est une option proposée par GitHub ; les systèmes d'exploitation pris en charge sont indiqués dans sa documentation officielle [GITHUB-DESKTOP-SUPPORTED-OS]. Elle peut compléter l'apprentissage, mais ne remplace pas la compréhension des notions de dépôt, d'état, de zone d'index et de commit.

> Préparation recommandée : installez Git avant d'aborder les chapitres pratiques. Créez également un compte GitHub si vous souhaitez publier ou partager vos dépôts pendant le parcours.

Vue d'ensemble du parcours d'apprentissage

Le cours est organisé selon une progression qui va du local vers le partage en ligne.

Étape	Question à laquelle elle répond	Compétence visée
1. Comprendre	Pourquoi suivre les versions d'un projet ?	Situer Git et GitHub dans un flux de travail.
2. Installer et configurer	Mon environnement est-il prêt ?	Disposer de Git et d'une identité de base configurée.
3. Créer un dépôt	Comment démarrer le suivi d'un dossier ?	Initialiser un dépôt local.
4. Observer et préparer	Qu'est-ce qui a changé et que vais-je enregistrer ?	Lire l'état d'un dépôt et sélectionner des modifications.
5. Enregistrer	Comment créer un historique utile ?	Réaliser des commits cohérents et compréhensibles.
6. Publier et synchroniser	Comment relier mon travail local à GitHub ?	Utiliser un dépôt distant sans confondre local et distant.
7. Collaborer progressivement	Comment travailler avec d'autres personnes sans perdre le fil ?	Employer les mécanismes de partage et de branches de façon raisonnée.

Les premières étapes constituent le socle du cours. Avant de penser à la collaboration, il faut être capable de répondre clairement à trois questions : dans quel dossier suis-je ? qu'ai-je modifié ? qu'est-ce que je veux enregistrer maintenant ?

Repères avant de continuer

Pour démarrer dans de bonnes conditions, retenez les points suivants :

- Git est l'outil qui suit l'historique des modifications dans un dépôt ;
- GitHub est une plateforme qui peut héberger et partager des dépôts Git ;
- un dépôt local peut exister sans GitHub ;
- un changement de fichier n'entre dans l'historique que lorsque l'on le prépare puis que l'on crée un commit ;
- les commandes Git seront en anglais, mais leur logique sera expliquée pas à pas en français ;
- la prochaine étape consiste à préparer l'environnement de travail, puis à créer un premier dépôt local.

CHAPITRE 2

Avant de créer un premier projet, il est utile de mettre en place un environnement simple, personnel et vérifiable. L'objectif n'est pas d'installer tous les outils possibles : il s'agit de disposer des éléments indispensables pour travailler localement avec Git et, lorsque nécessaire, accéder à GitHub.

À la fin de ce chapitre, vous devez pouvoir :

- ouvrir un terminal ;
- exécuter la commande git sur votre ordinateur ;
- identifier la version installée ;
- associer un nom et une adresse e-mail à vos futurs validations ;
- accéder à un compte GitHub protégé ;
- savoir distinguer ce qui se passe sur votre ordinateur de ce qui se passe sur GitHub.

> Important — Git et GitHub sont deux éléments distincts. Git est le logiciel de gestion de versions installé sur votre ordinateur. GitHub est un service en ligne qui peut héberger des dépôts Git et faciliter la collaboration. Git peut être utilisé sans GitHub ; GitHub s'appuie sur Git pour les opérations de versionnage. [GITHUB-ABOUT-GIT-FR]

Les composants de l'environnement de travail

Un environnement minimal pour débuter comporte quatre composants.

Composant	Rôle	Indispensable au départ ?
Un ordinateur et un système d'exploitation	Exécuter Git et stocker les fichiers du projet	Oui
Un terminal	Saisir les commandes Git	Oui
Git	Suivre les versions de fichiers dans un dépôt local	Oui
Un compte GitHub	Héberger ou partager des dépôts en ligne	Recommandé

Vous utiliserez aussi un éditeur de texte ou un environnement de développement pour créer et modifier vos fichiers. Cet outil n'est pas Git : Git observe les changements apportés aux fichiers, mais ne les écrit pas à votre place.

Vocabulaire essentiel

Terminal

Le terminal est une application dans laquelle vous donnez des instructions sous forme de texte. Sous Windows, il peut s'agir notamment de Git Bash, de Windows Terminal, de l'Invite de commandes ou de PowerShell. Sous macOS et Linux, l'application s'appelle fréquemment Terminal. Les commandes présentées dans ce cours sont saisies dans cet environnement.

Dépôt

Un dépôt est un dossier de projet dont Git suit l'historique. Il contient les fichiers du projet ainsi que des informations internes utilisées par Git. La création d'un dépôt local sera étudiée dans le chapitre consacré au démarrage d'un projet ; la commande Git prévue à cet effet est git init. [GIT-INIT]

Identité de configuration

L'identité de configuration correspond principalement au nom et à l'adresse e-mail que Git enregistrera avec vos validations. Une validation est un enregistrement structuré d'un état du projet dans l'historique Git. Cette identité est configurée dans Git ; elle ne doit pas être confondue avec le nom d'utilisateur affiché sur GitHub.

Validation

Dans Git, une validation — souvent appelée commit — est un point d'historique associé à des changements. Les fichiers sont d'abord sélectionnés pour l'index, puis les changements préparés sont enregistrés dans une validation. [GIT-ADD] [GIT-RECORDING-CHANGES]

Représentation de l’environnement

``text Vous éditeur de texte / environnement de développement crée et modifie les fichiers du projet terminal exécute les commandes Git Git installé sur votre ordinateur suit l'historique dans le dépôt local navigateur web donne accès à votre compte et à vos dépôts GitHub ``

Le dépôt local reste sur votre ordinateur. GitHub intervient lorsqu’un dépôt est publié sur le service en ligne ou lorsque vous récupérez un dépôt existant. Garder cette séparation à l’esprit évite une confusion courante : créer ou modifier un fichier localement ne le publie pas automatiquement sur GitHub.

Installer Git

Git doit être installé sur l’ordinateur avant que la commande git puisse être utilisée dans un terminal. Le site officiel de Git fournit des parcours d’installation distincts pour Windows, macOS et Linux. [GIT-INSTALL] [GIT-INSTALL-WINDOWS] [GIT-INSTALL-MACOS] [GIT-INSTALL-LINUX]

Choisir le parcours adapté à votre système

Système d’exploitation	Point de départ recommandé
Windows	Page officielle d’installation pour Windows [GIT-INSTALL-WINDOWS]
macOS	Page officielle d’installation pour macOS [GIT-INSTALL-MACOS]
Linux	Page officielle d’installation pour Linux [GIT-INSTALL-LINUX]

Utilisez de préférence les liens de téléchargement ou les instructions proposés par la documentation officielle. Cela limite le risque d’installer un programme non désiré ou une version provenant d’une source non fiable.

Installation sous Windows

Sur Windows, suivez le parcours indiqué par la page officielle Git pour Windows. Une fois l’installation terminée, ouvrez un terminal disponible sur votre système, en particulier celui installé avec Git si cette option vous est proposée, puis vérifiez l’installation avec la commande indiquée plus loin dans ce chapitre. [GIT-INSTALL-WINDOWS]

Lorsqu’un assistant d’installation propose de nombreux choix techniques, un débutant peut conserver les réglages proposés par défaut, sauf si son organisation lui a donné des consignes précises. L’objectif immédiat est que la commande git soit reconnue dans un terminal.

Installation sous macOS

La documentation officielle Git présente les méthodes d'installation disponibles pour macOS. Suivez l'une de ces méthodes, puis fermez et rouvrez le terminal avant de vérifier que Git est accessible. [GIT-INSTALL-MACOS]

Installation sous Linux

Les distributions Linux proposent généralement un gestionnaire de paquets. La page officielle Git consacrée à Linux indique les commandes et méthodes correspondant à plusieurs distributions. Utilisez la méthode adaptée à votre distribution, puis vérifiez l'installation dans un nouveau terminal. [GIT-INSTALL-LINUX]

Vérifier immédiatement l'installation

Dans un terminal, saisissez :

``bash git --version ``

L'option --version fait partie des options générales de la commande Git. [GIT-OVERVIEW]

Si Git est correctement installé et accessible, le terminal affiche une ligne comportant une version de Git. Le numéro exact varie selon votre système et le moment de l'installation : il n'est pas nécessaire qu'il soit identique à celui d'un exemple trouvé ailleurs.

Si le terminal indique que git est introuvable ou qu'il ne reconnaît pas la commande :

1. fermez complètement le terminal ;
2. ouvrez-en un nouveau ;
3. exécutez de nouveau git --version ;
4. si le problème persiste, reprenez la procédure officielle correspondant à votre système.

Ne poursuivez pas la configuration tant que cette commande ne fonctionne pas. Tous les réglages qui suivent dépendent d'une installation opérationnelle.

Créer un compte GitHub ou accéder à un compte existant

GitHub est utile pour conserver une copie en ligne d'un projet, partager un dépôt ou collaborer avec d'autres personnes. La création d'un compte se fait depuis le parcours officiel de GitHub ; celui-ci demande notamment de fournir une adresse e-mail, de choisir un mot de passe et de choisir un nom d'utilisateur. GitHub prévoit également une étape de vérification de l'adresse e-mail. [GITHUB-ACCOUNT-CREATION]

Si vous créez votre premier compte

Suivez les instructions affichées par GitHub pour :

1. fournir une adresse e-mail à laquelle vous avez réellement accès ;
2. choisir un mot de passe robuste et personnel ;
3. sélectionner votre nom d'utilisateur ;

4. terminer la vérification de l’adresse e-mail demandée par GitHub. [GITHUB-ACCOUNT-CREATION]

Votre nom d’utilisateur GitHub sert à vous identifier sur la plateforme. Prenez le temps de le choisir : il peut apparaître dans les adresses de vos dépôts et dans vos interactions publiques sur GitHub.

Si vous possédez déjà un compte

Connectez-vous à votre compte dans un navigateur et vérifiez que vous avez accès à vos paramètres. Si l’adresse e-mail associée au compte n’est plus accessible, ou si vous ne pouvez plus vous connecter, résolvez ce point avant d’utiliser le compte pour des projets importants.

Protéger le compte avec l’authentification à deux facteurs

GitHub permet de configurer l’authentification à deux facteurs (2FA), c’est-à-dire une protection qui demande un second facteur en plus du mot de passe lors de la connexion. La documentation officielle décrit les méthodes proposées ainsi que le parcours de configuration. [GITHUB-2FA-CONFIG]

Activez l’authentification à deux facteurs dès que possible pour un compte que vous comptez utiliser durablement. Après l’activation, configurez aussi des méthodes de récupération : elles servent à retrouver l’accès au compte si votre méthode principale devient indisponible. [GITHUB-2FA-RECOVERY]

> Pratique de sécurité Conservez les informations de récupération dans un endroit sûr, distinct de votre ordinateur lorsque cela est possible. Un compte GitHub perdu peut empêcher l’accès aux dépôts auxquels vous collaborez.

Configurer l’identité de Git

Après l’installation, Git doit connaître le nom et l’adresse e-mail à inscrire dans vos validations. La configuration initiale recommandée par la documentation Git consiste à définir user.name et user.email. [GIT-FIRST-CONFIG]

Dans le terminal, exécutez les commandes suivantes en remplaçant les valeurs entre guillemets par les vôtres :

``bash git config --global user.name "Votre Nom" git config --global user.email "vous@example.com" ``

L’option --global enregistre ces paramètres pour votre utilisateur sur cet ordinateur. La documentation Git distingue notamment les niveaux de configuration système, global et local ; une configuration locale s’applique à un dépôt particulier et peut remplacer une valeur plus générale. [GIT-CONFIG] [GIT-FIRST-CONFIG]

Choisir le bon nom et la bonne adresse e-mail

Le nom peut être votre nom réel, un nom professionnel ou un pseudonyme cohérent avec l’usage prévu. L’essentiel est de choisir une identité que vous acceptez de voir associée à vos validations.

Pour l’adresse e-mail, utilisez une adresse que vous contrôlez et qui correspond à la manière dont vous souhaitez être identifié dans l’historique Git. Si vous travaillez dans une organisation, vérifiez ses règles avant d’utiliser une adresse professionnelle ou de publier un dépôt.

> À retenir user.name et user.email sont des paramètres de Git. Le nom d'utilisateur GitHub est un identifiant de plateforme. Ils peuvent se ressembler, mais ils ne sont ni le même réglage ni nécessairement la même valeur.

Vérifier votre identité configurée

Pour afficher le nom configuré globalement :

``bash git config --global user.name ``

Pour afficher l'adresse e-mail configurée globalement :

``bash git config --global user.email ``

La commande git config sert à obtenir et à modifier des options de configuration. [GIT-CONFIG]

Vous pouvez également afficher l'ensemble de votre configuration avec :

``bash git config --list ``

Avant de partager une capture d'écran ou le résultat de cette commande, relisez-le : il peut contenir des informations personnelles, notamment votre adresse e-mail.

Corriger une valeur

Si vous avez saisi une valeur erronée, exécutez simplement à nouveau la commande correspondante avec la bonne valeur :

``bash git config --global user.name "Nom corrigé" git config --global user.email "adresse-corrigee@exam-ple.com" ``

Git met à jour la valeur du paramètre ciblé. [GIT-CONFIG]

Réglages initiaux utiles

Les deux réglages d'identité sont prioritaires. Quelques autres options peuvent améliorer le confort de travail, mais elles restent secondaires pour un premier usage.

Définir l'éditeur de texte utilisé par Git

Certaines opérations Git peuvent ouvrir un éditeur de texte, par exemple pour vous demander d'écrire un message. L'option core.editor permet de définir l'éditeur que Git utilisera. [GIT-FIRST-CONFIG] [GIT-CON-FIG]

La commande générale suit cette structure :

``bash git config --global core.editor "commande-de-votre-editeur" ``

Ne copiez pas une valeur trouvée pour un autre ordinateur sans la comprendre : la commande à indiquer dépend de l'éditeur installé et de votre système. Si vous ne savez pas quelle valeur choisir, laissez ce réglage de côté pour le moment ; Git utilisera alors son comportement par défaut.

Définir le nom initial des nouvelles branches

Git permet de définir, via init.defaultBranch, le nom de branche initial utilisé lors de la création de nouveaux dépôts. [GIT-FIRST-CONFIG] [GIT-CONFIG]

La structure de la commande est :

``bash git config --global init.defaultBranch "nom-de-branche" ``

Ce réglage concerne les futurs dépôts initialisés sur votre ordinateur ; il ne renomme pas automatiquement les branches de dépôts existants. Pour un débutant, le plus important est de comprendre qu'une branche possède un nom et que ce nom peut être défini par la configuration ou par les règles d'un projet.

Comprendre les niveaux de configuration

Niveau	Portée	Usage typique
Système	Tous les utilisateurs de l'ordinateur	Réglages administrés sur une machine partagée
Global	Votre utilisateur sur cet ordinateur	Nom, e-mail et préférences personnelles
Local	Un dépôt précis	Exception propre à un projet

Git lit ces niveaux de configuration selon un ordre de priorité ; un réglage plus spécifique peut remplacer un réglage plus général. [GIT-CONFIG] [GIT-FIRST-CONFIG]

Pour débuter, utilisez --global pour votre identité personnelle. Vous rencontrerez la configuration locale plus tard, lorsqu'un projet exigera une identité ou un comportement différent.

Faut-il installer GitHub Desktop ?

GitHub Desktop est une application graphique qui permet d'utiliser certains flux Git sans saisir toutes les commandes dans un terminal. Son utilisation est facultative : Git et le terminal restent des bases utiles à comprendre, y compris si vous utilisez ensuite une interface graphique.

GitHub Desktop n'est pris en charge que sur certains systèmes d'exploitation ; consultez la page officielle de compatibilité avant de décider de l'installer. [GITHUB-DESKTOP-SUPPORTED-OS]

Pour ce cours, le terminal est retenu comme point de référence, car il rend les commandes et leur effet explicites. Vous pouvez toutefois utiliser GitHub Desktop en complément, à condition de ne pas confondre l'application avec GitHub, le service en ligne.

Vérification complète : la checklist de départ

Avant de passer au premier dépôt, vérifiez les éléments suivants.

Vérification	Comment la réaliser	Résultat attendu
Git est installé	git --version	Une version de Git s'affiche [GIT-OVERVIEW]
Votre nom Git est défini	git config --global user.name	Le nom choisi s'affiche [GIT-CONFIG]
Votre e-mail Git est défini	git config --global user.email	L'adresse choisie s'affiche [GIT-CONFIG]
Vous pouvez ouvrir un terminal	Ouvrez-en un nouveau	Vous pouvez saisir une commande
Vous pouvez accéder à GitHub	Connectez-vous dans un navigateur	Votre compte est accessible
Votre compte est protégé	Consultez les paramètres de sécurité GitHub	L'authentification à deux facteurs et les solutions de récupération sont configurées, si possible [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY]

La commande suivante offre une vérification rapide des deux paramètres principaux :

``bash git config --global --list ``

Vous devriez y retrouver des lignes correspondant à user.name et user.email. La liste peut contenir d'autres paramètres : ce n'est pas un problème. [GIT-CONFIG]

Bonnes pratiques de configuration personnelle

Garder une identité cohérente

Utilisez un nom et une adresse e-mail stables pour les projets personnels. Cela rend l'historique plus lisible. Si un projet professionnel impose une autre identité, configurez-la localement dans ce dépôt plutôt que de modifier sans cesse votre configuration globale.

Ne pas partager de données sensibles

Un dépôt, un historique Git ou une capture de terminal peuvent contenir des informations que vous ne souhaitiez pas rendre publiques. Relisez toujours ce que vous publiez ou partagez. En particulier, ne placez pas de mots de passe, de codes de récupération ou de secrets dans les fichiers d'un projet.

Distinguer les comptes et les machines

La configuration --global est liée à votre compte utilisateur sur une machine donnée. Lorsque vous utilisez un autre ordinateur, vous devez vérifier ou refaire la configuration de Git sur cet autre environnement. [GIT-FIRST-CONFIG]

Préférer une progression simple

Au début, évitez de modifier de nombreux réglages Git à la fois. Une installation fonctionnelle, une identité bien définie et un compte GitHub sécurisé suffisent pour commencer. Vous ajouterez des outils et des préférences lorsque vous rencontrerez un besoin concret.

> Environnement prêt Votre environnement est prêt lorsque git --version fonctionne, que votre identité Git est configurée et que vous pouvez vous connecter à votre compte GitHub. Vous pouvez alors créer un dossier de projet et commencer à utiliser Git pour en suivre l'historique.

CHAPITRE 3

Git est un système de contrôle de version : il permet d'enregistrer l'évolution d'un projet au fil du temps. Plutôt que de remplacer silencieusement un fichier par une nouvelle version, Git permet de conserver des points d'historique identifiables. Un projet peut ainsi être relu, comparé et repris à partir de versions précédemment enregistrées. [GITHUB-ABOUT-GIT-FR]

Pour utiliser Git avec confiance, il est essentiel de distinguer trois idées : le projet, le dépôt et les états successifs des fichiers. Ces notions expliquent ce que Git enregistre, ce qu'il n'enregistre pas encore, et pourquoi une commande telle que git status est si utile.

> Idée centrale : modifier un fichier ne l'ajoute pas automatiquement à l'historique. Une modification passe par plusieurs espaces avant de devenir une version enregistrée.

Notion de projet et de dépôt

Un projet est l'ensemble des fichiers qui servent à produire quelque chose : par exemple un site web, un programme, un document, une présentation ou une collection de notes. Un projet peut comprendre des fichiers de code, des images, une documentation et des fichiers de configuration.

Un dépôt — repository en anglais, souvent abrégé en repo — est le projet placé sous le suivi de Git. Il contient les fichiers du projet et les données que Git utilise pour conserver leur historique. Git peut créer un dépôt dans un répertoire existant avec la commande git init. [GIT-INIT]

Lorsqu'un répertoire devient un dépôt Git, Git y crée un répertoire administratif nommé .git. Ce répertoire contient les informations nécessaires à la gestion du dépôt et de son historique. [GIT-INIT]

Il faut donc distinguer :

Élément	Rôle	Exemple
Projet	Travail que l'on souhaite réaliser ou conserver	Un site de recettes
Répertoire de travail	Dossier visible dans lequel les fichiers sont modifiés	site-recettes/
Dépôt Git	Projet dont Git suit les versions	Le répertoire site-recettes/ après git init
Répertoire .git	Données internes utilisées par Git pour gérer le dépôt	site-recettes/.git/

Le dépôt est d'abord local : il se trouve sur l'ordinateur de la personne qui travaille. GitHub peut ensuite héberger une copie du dépôt en ligne, mais Git et GitHub ne désignent pas la même chose. Git est l'outil de gestion de versions ; GitHub est une plateforme qui permet notamment d'héberger des dépôts Git. [GITHUB-ABOUT-GIT-FR]

> Prudence : le dossier .git est indispensable au dépôt local. Le supprimer revient à supprimer les informations de suivi et l'historique Git présents dans ce dossier.

Historique et versions

L'historique d'un dépôt est une suite de versions enregistrées. Dans Git, une version enregistrée est généralement appelée un commit, terme que l'on peut traduire par validation ou enregistrement de validation. La documentation GitHub emploie également le terme commit dans son glossaire français. [GITHUB-GLOSSARY-FR]

Une validation correspond à un instant choisi dans l'évolution du projet. Elle enregistre les modifications qui ont été explicitement préparées pour faire partie de cette version. Elle ne doit donc pas être confondue avec une simple sauvegarde automatique de tous les fichiers modifiés.

Prenons un fichier recette.txt :

1. une première version contient le titre et la liste des ingrédients ;
2. une validation enregistre cet état initial ;
3. une étape de préparation est ajoutée ;
4. une deuxième validation enregistre cette modification ;
5. une correction est apportée au temps de cuisson ;
6. une troisième validation enregistre cette correction.

L'historique peut se représenter ainsi :

``text Version 1 Version 2 Version 3 [recette initiale] ' [ajout d'une étape] ' [correction de cuisson] commit commit commit ``

Chaque validation permet de documenter une étape cohérente. Dans un projet réel, l'historique aide notamment à répondre à des questions simples :

- Quelle modification a été enregistrée ?
- À quel moment une fonctionnalité ou une correction a-t-elle été ajoutée ?
- Quels fichiers font partie d'une version donnée ?
- Quelle était la dernière version enregistrée avant une modification en cours ?

Git enregistre les modifications dans le dépôt à travers une zone de préparation, appelée index ou staging area en anglais. Cette zone permet de choisir précisément les modifications qui feront partie de la prochaine validation. [GIT-RECORDING-CHANGES]

Fichiers suivis et non suivis

Git ne traite pas tous les fichiers de la même manière. La première distinction importante est celle entre un fichier suivi et un fichier non suivi.

Un fichier suivi — tracked file — est un fichier que Git connaît déjà. Il peut provenir d'une validation précédente ou avoir été ajouté à la zone de préparation afin d'être inclus dans une prochaine validation. Les fichiers suivis peuvent être non modifiés, modifiés ou préparés pour la prochaine validation. [GIT-RECORDING-CHANGES]

Un fichier non suivi — untracked file — existe dans le répertoire de travail, mais Git ne l'a pas encore ajouté à son suivi. Il n'appartient donc pas encore à l'historique du dépôt. [GIT-RECORDING-CHANGES]

Situation	Terme anglais	Ce que cela signifie
Git connaît déjà le fichier	tracked	Le fichier fait partie du suivi Git
Le fichier existe mais Git ne le suit pas encore	untracked	Il n'est pas encore prévu pour une validation
Un fichier suivi a été changé	modified	Son contenu diffère de la dernière version enregistrée
Des changements sont prêts pour la prochaine validation	staged	Ils ont été ajoutés à l'index ou à la zone de préparation

La commande git status affiche l'état des fichiers du répertoire de travail et de l'index. Elle indique notamment les fichiers non suivis, les changements préparés pour la prochaine validation et les changements non préparés. [GIT-STATUS]

Supposons qu'un projet contienne déjà un fichier suivi nommé README.md. Si l'on crée ensuite un fichier notes.txt, Git peut afficher une situation comparable à celle-ci :

```text
Changes not staged for commit:
  modified: README.md

Untracked files:
  notes.txt
```

Cette lecture signifie :

- README.md est déjà connu de Git, mais il a été modifié depuis la dernière validation ;
- notes.txt existe dans le dossier, mais Git ne le suit pas encore ;
- aucune de ces modifications n'est encore préparée pour la prochaine validation.

Les espaces de travail à distinguer

Pour comprendre le cycle d'une modification, il faut séparer trois espaces :

1. le répertoire de travail — working tree ou working directory ;
2. la zone de préparation — staging area ou index ;
3. le dépôt local — local repository.

Le répertoire de travail est l'ensemble des fichiers visibles et modifiables dans le dossier du projet. C'est là que l'on écrit, corrige, crée ou supprime des fichiers.

La zone de préparation est une sélection de modifications destinées à la prochaine validation. La commande git add ajoute le contenu de fichiers à l'index. [GIT-ADD]

Le dépôt local contient les validations déjà créées. Lorsqu'une validation est réalisée, les modifications préparées deviennent une nouvelle étape de l'historique du dépôt. La documentation Git décrit ce processus comme l'enregistrement des changements dans le dépôt. [GIT-RECORDING-CHANGES]

Espace	Question à se poser	Action associée
Répertoire de travail	Qu'ai-je modifié sur mon ordinateur ?	Modifier ou créer des fichiers
Zone de préparation	Que veux-je inclure dans la prochaine validation ?	git add
Dépôt local	Quelles versions sont déjà enregistrées ?	Créer une validation

Le parcours habituel d'un fichier peut être représenté de la façon suivante :

``text Fichier créé ou modifié ¼ Répertoire de travail git add ¼ Zone de préparation (index / staging area) création d'une validation ¼ Historique du dépôt local ``

Une même modification ne passe pas obligatoirement immédiatement dans l'historique. Elle peut rester dans le répertoire de travail pendant que l'on continue à écrire. Elle peut aussi être préparée, puis nécessiter une vérification avant la validation. Cette séparation donne à la personne qui travaille un contrôle précis sur le contenu de chaque version enregistrée.

États des modifications

Les états les plus utiles pour débuter peuvent être lus comme un cycle.

1. Non suivi : untracked

Un nouveau fichier vient d'être créé, mais Git ne le connaît pas encore. Il est présent dans le répertoire de travail, sans appartenir à l'historique.

``text notes.txt : non suivi ``

Pour demander à Git de préparer ce fichier, on utilise git add :

``bash git add notes.txt ``

La commande git add ajoute le contenu du fichier à l'index. [GIT-ADD]

2. Préparé : staged

Après git add, la version actuelle du fichier est dans la zone de préparation. Elle est prête à être incluse dans la prochaine validation.

``text notes.txt : préparé pour la prochaine validation ``

Préparer un fichier ne crée pas encore une version dans l'historique. Cela sélectionne seulement son contenu pour la prochaine validation.

3. Validé : committed

Une fois la validation créée, le contenu préparé est enregistré dans l'historique local. Le fichier est alors suivi par Git et, tant qu'il n'est pas modifié, son état est propre — clean dans certains messages Git.

``text notes.txt : enregistré dans l'historique ``

4. Modifié : modified

Si l'on modifie ensuite notes.txt, Git détecte une différence entre le fichier présent dans le répertoire de travail et sa dernière version enregistrée.

``text notes.txt : suivi, mais modifié ``

Il faut alors préparer à nouveau la modification avec git add si l'on souhaite qu'elle fasse partie de la prochaine validation. La commande git add sert donc aussi bien à ajouter un nouveau fichier qu'à préparer une nouvelle version d'un fichier déjà suivi. [GIT-ADD]

> À retenir : git add ne signifie pas seulement « ajouter un fichier ». Dans le flux de travail Git, cette commande place dans la zone de préparation l'état actuel du contenu sélectionné.

Le rôle des validations

Une validation ne doit pas être envisagée comme une formalité technique. Elle constitue une unité d'historique. Elle permet de dire : « à ce point précis, le projet comprend ces modifications ».

Une validation utile rassemble idéalement des changements qui répondent à une même intention. Par exemple :

- ajouter une page de présentation ;
- corriger une faute dans la documentation ;
- mettre à jour une liste de dépendances ;
- ajouter une image à une page existante.

À l'inverse, mélanger dans une seule validation une correction de texte, une suppression accidentelle et une fonctionnalité sans rapport rend l'historique plus difficile à comprendre.

La zone de préparation permet justement de choisir ce qui entre dans une validation. Si deux fichiers sont modifiés mais qu'un seul est prêt, il est possible de préparer uniquement celui qui doit faire partie de la prochaine version. Git distingue explicitement les changements préparés de ceux qui ne le sont pas. [GIT-STATUS]

Voici une situation fréquente :

``text README.md modifié et préparé brouillon.txt modifié mais non préparé ``

La prochaine validation contiendra la version préparée de README.md, mais pas la modification non préparée de brouillon.txt. Cette distinction est l'un des mécanismes fondamentaux de Git.

Lecture d'un état de dépôt

Avant de préparer des fichiers ou de créer une validation, la commande git status donne une vue d'ensemble du dépôt. La documentation officielle indique que cette commande affiche les chemins présentant des différences entre le fichier d'index, la validation courante et le répertoire de travail. [GIT-STATUS]

Pour un débutant, il est plus simple de la lire en trois questions :

1. Quels fichiers sont déjà préparés pour la prochaine validation ?
2. Quels fichiers suivis ont été modifiés mais ne sont pas encore préparés ?
3. Quels nouveaux fichiers Git ne suit-il pas encore ?

Exemple :

```text On branch main

Changes to be committed: new file: contact.html modified: index.html

Changes not staged for commit: modified: style.css

Untracked files: logo-test.png ```

Interprétation :
```

Partie affichée	Interprétation	Conséquence pour la prochaine validation
Changes to be committed	Les fichiers sont dans la zone de préparation	Ils seront inclus si une validation est créée maintenant
Changes not staged for commit	Git suit ces fichiers, mais leurs modifications ne sont pas préparées	Elles ne seront pas incluses tant qu'elles ne sont pas ajoutées à l'index
Untracked files	Git ne suit pas encore ces fichiers	Ils ne feront pas partie de l'historique sans préparation explicite

Le nom de branche affiché en tête, ici main, indique le contexte de travail courant. La notion de branche sera étudiée plus loin ; à ce stade, il suffit de comprendre que git status décrit l'état du dépôt dans le contexte actuel.

La commande est particulièrement utile dans les situations suivantes :

- juste après avoir modifié des fichiers ;
- avant d'utiliser git add ;
- après avoir utilisé git add, pour vérifier ce qui est préparé ;
- avant de créer une validation ;
- lorsqu'un message Git semble inattendu.

> Bon réflexe : lorsqu'il existe un doute sur ce que Git va enregistrer, exécuter git status avant toute validation.

Vocabulaire bilingue essentiel

Les messages de Git sont souvent en anglais, même si l'on suit un cours en français. Connaître quelques termes évite de confondre les étapes du cycle de travail.

Français	Anglais courant dans Git	Sens pratique
dépôt	repository	Projet géré par Git, avec son historique
dépôt local	local repository	Dépôt présent sur l'ordinateur
répertoire de travail	working tree / working directory	Fichiers visibles et modifiables du projet
zone de préparation	staging area	Zone qui sélectionne le contenu de la prochaine validation
index	index	Nom technique de la zone de préparation
fichier suivi	tracked file	Fichier déjà connu de Git
fichier non suivi	untracked file	Fichier que Git ne connaît pas encore
fichier modifié	modified file	Fichier suivi dont le contenu a changé
préparer	stage	Placer des modifications dans la zone de préparation
validation	commit	Enregistrement d'une version dans l'historique
historique	history	Ensemble des validations du dépôt
état	status	Vue de la situation des fichiers dans Git

Les termes staging area et index désignent la même zone dans la documentation Git. De même, le mot commit reste très courant, y compris dans des explications en français. [GIT-RECORDING-CHANGES]

Modèle mental à conserver

Un dépôt Git n'est pas seulement un dossier contenant des fichiers. C'est un projet auquel Git associe un historique de validations. Entre une modification et son inscription dans cet historique, Git distingue volontairement le répertoire de travail, la zone de préparation et le dépôt local.

Le flux à retenir est donc le suivant :

``text Je modifie un fichier “ Je vérifie son état avec git status “ Je sélectionne son contenu avec git add “ Je crée une validation pour l'inscrire dans l'historique local ``

Cette progression explique pourquoi un fichier peut être visible dans le dossier du projet sans être suivi, pourquoi une modification peut ne pas être incluse dans la prochaine validation, et pourquoi l'historique Git reste structuré en étapes choisies plutôt qu'en simples copies successives de fichiers.

CHAPITRE 4

Ce chapitre met en pratique le cycle de travail fondamental avec Git : créer un espace de projet, demander à Git d'en suivre l'historique, examiner ce qui a changé, sélectionner ce qui doit être enregistré, puis créer une validation.

Une validation — souvent appelée commit — est un enregistrement identifiable d'un état préparé du projet. Elle ne conserve pas automatiquement tout ce qui se trouve dans le dossier : Git enregistre ce qui a été explicitement préparé pour cette validation.

> Vocabulaire essentiel - Dépôt local : dossier de projet dont Git gère l'historique sur votre ordinateur. - Fichier non suivi : fichier présent dans le dossier, mais pas encore pris en charge par Git. - Zone de préparation : sélection du contenu qui sera inclus dans la prochaine validation ; elle est aussi appelée index ou staging area. - Validation : enregistrement d'un état préparé dans l'historique.

Le flux de travail à retenir

Le cycle le plus fréquent peut se résumer ainsi :

1. vous modifiez ou créez des fichiers dans le dossier du projet ;
2. vous observez leur état avec git status ;
3. vous préparez les changements choisis avec git add ;
4. vous créez une validation avec git commit ;
5. vous consultez l'historique avec git log.

Git distingue donc volontairement la modification d'un fichier et son enregistrement dans l'historique. Cette séparation permet de construire une validation cohérente, même lorsque plusieurs travaux sont en cours. La documentation Git présente git status, git add et git commit comme les commandes centrales de l'enregistrement des modifications. [GIT-RECORDING-CHANGES]

Créer un dossier de projet

Commencez par créer un dossier dédié à un petit projet. Dans cet exemple, il s'agit d'un dossier nommé notes-git.

Vous pouvez le créer avec votre explorateur de fichiers ou votre environnement de développement. Ouvrez ensuite un terminal dans ce dossier ou déplacez-vous dans celui-ci :

``bash mkdir notes-git cd notes-git ``

La commande mkdir crée le dossier et cd place le terminal dans ce dossier. Selon votre système et le terminal employé, la création peut aussi être faite graphiquement ; l'important est que les commandes Git suivantes soient exécutées à l'intérieur du dossier du projet.

À ce stade, le dossier existe, mais Git ne suit encore aucun fichier et aucun historique n'y est associé.

Initialiser un dépôt local

Pour transformer le dossier courant en dépôt Git, exécutez :

``bash git init ``

La commande git init initialise un dépôt Git vide dans le dossier courant. Elle met en place les éléments nécessaires à la gestion de l'historique local. [GIT-INIT]

Un message ressemblant à celui-ci peut s'afficher :

``text Initialized empty Git repository in .../notes-git/.git/ ``

Le chemin exact varie selon votre ordinateur. Le dossier caché .git contient les données et la configuration internes du dépôt. Ne le supprimez pas et ne le modifiez pas manuellement pendant votre apprentissage : Git le gère par ses commandes.

Vous pouvez vérifier que vous vous trouvez bien dans un dépôt en exécutant :

``bash git status ``

Git affiche alors l'état courant du dépôt. La commande git status indique notamment les fichiers non suivis, les changements préparés pour la prochaine validation et les changements non préparés. [GIT-STATUS]

Dans un dépôt nouvellement initialisé et vide, le résultat indique qu'il n'y a encore rien à valider.

> Point de repère git init ne crée pas un projet à votre place : il active le suivi Git dans un dossier existant. Vous créez ensuite les fichiers de votre projet comme vous le feriez habituellement.

Créer un premier fichier et consulter l'état du dépôt

Créez maintenant un fichier nommé README.md dans le dossier notes-git. Ouvrez-le dans un éditeur de texte et ajoutez par exemple :

```markdown
# Mes notes sur Git

Ce dépôt contient mes premiers essais avec Git. ```

Enregistrez le fichier, puis revenez au terminal :

``bash git status ``

Git devrait signaler que README.md est un fichier non suivi. Une sortie typique ressemble à ceci :

```text On branch ...

No commits yet

Untracked files: (use "git add <file>..." to include in what will be committed) README.md

nothing added to commit but untracked files present ```

Cette sortie est indicative : le nom de la branche et certains textes peuvent différer selon la version et la configuration de Git. Le point important est la rubrique Untracked files : Git voit le fichier dans le dossier, mais ne l'inclura dans aucune validation tant que vous ne l'aurez pas ajouté.

### Lire git status comme un tableau de bord

Après chaque étape importante, git status répond à trois questions pratiques :

| Question | Ce que recherche Git | Action possible |
|---|---|---|
| Quels fichiers Git ne suit-il pas encore ? | Les fichiers non suivis | Les préparer avec git add si leur suivi est souhaité |
| Qu'est-ce qui sera inclus dans la prochaine validation ? | Les changements préparés | Vérifier la sélection avant git commit |
| Qu'est-ce qui a changé sans être préparé ? | Les modifications de fichiers déjà suivis | Les préparer, les compléter ou les laisser de côté |

La documentation officielle décrit git status comme une commande affichant l'état de l'arbre de travail et de la zone de préparation. [GIT-STATUS]

## Ajouter un fichier au suivi et le préparer

Pour sélectionner README.md en vue de la prochaine validation, utilisez :

``bash git add README.md ``

La commande git add ajoute le contenu indiqué à la zone de préparation. Elle permet notamment d'ajouter un nouveau fichier au suivi et de préparer des modifications pour la prochaine validation. [GIT-ADD]

Vérifiez immédiatement le résultat :

``bash git status ``

La sortie devrait désormais contenir une section de ce type :

``text Changes to be committed: (use "git restore --staged <file>..." to unstage) new file:   README.md ``

Le fichier est maintenant préparé. Cela ne signifie pas encore qu'il est enregistré dans l'historique : il est seulement sélectionné pour la prochaine validation.

### Préparer un fichier précis ou plusieurs fichiers

Pour préparer un seul fichier, indiquez son nom :

``bash git add README.md ``

Pour préparer plusieurs fichiers précis, indiquez-les à la suite :

``bash git add README.md chapitre-1.md ``

La documentation de git add précise que la commande accepte des chemins de fichiers et met à jour le contenu préparé à partir de l'arbre de travail. [GIT-ADD]

Pour débuter, privilégiez les noms de fichiers explicites. Vous verrez ainsi exactement ce que vous sélectionnez. Une commande courte ne doit pas remplacer la vérification : consultez git status avant toute validation.

## Comprendre la différence entre modifier, préparer et valider

La distinction entre ces trois états est au cœur de Git.

Supposons que README.md a été préparé avec git add README.md, puis que vous modifiez à nouveau le fichier avant de créer la validation. Git peut alors signaler deux états pour le même fichier :

- une version préparée, qui sera incluse dans la prochaine validation ;
- une modification plus récente, encore non préparée, qui ne sera pas incluse.

Pour inclure cette dernière modification, vous devez exécuter à nouveau :

``bash git add README.md ``

La zone de préparation correspond donc au contenu que vous avez choisi de valider, et non à une promesse générale de tout prendre dans le dossier. Git permet ainsi de préparer les changements de manière sélective avant de les enregistrer. [GIT-RECORDING-CHANGES]

> Réflexe de débutant utile  Adoptez la séquence suivante : modifier ’ git status ’ git add ’ git status ’ git commit.  Le second git status confirme ce qui sera effectivement enregistré.

### Flux visuel : du fichier à l’historique

``text Fichier créé ou modifié  ¼ Dossier de travail  git add README.md ¼ Zone de préparation  git commit -m "..." ¼ Historique local du dépôt ``

Le dossier de travail contient ce que vous êtes en train d’écrire. La zone de préparation contient ce que vous avez choisi pour la prochaine validation. L’historique local contient les validations déjà créées.

## Créer une première validation

Lorsque git status indique que README.md est prêt à être validé, créez votre première validation :

``bash git commit -m "Ajoute un fichier de présentation" ``

L’option -m fournit le message de validation directement dans la commande. Git crée alors une validation à partir du contenu présent dans la zone de préparation. Les fichiers modifiés mais non préparés n’y sont pas inclus. [GIT-RECORDING-CHANGES]

Une sortie similaire à celle-ci peut apparaître :

``text [... (root-commit) ...] Ajoute un fichier de présentation 1 file changed, 3 insertions(+) create mode ... README.md ``

Les identifiants et les nombres affichés dépendent de votre dépôt. Retenez surtout que Git confirme la création de la validation et résume les fichiers concernés.

Après la validation, exécutez :

``bash git status ``

Si aucun nouveau changement n’a été fait, Git indique que le dossier de travail ne contient rien à valider. Vous venez alors de passer d’un fichier non suivi à un premier point enregistré dans l’historique du dépôt.

### Si Git vous demande votre identité

Une validation est associée à un nom et à une adresse électronique. Si Git refuse la validation en demandant de configurer ces informations, définissez-les conformément à la configuration de votre environnement :

``bash git config --global user.name "Votre Nom" git config --global user.email "vous@example.com" ``

Git permet de définir des variables de configuration avec git config, et la configuration initiale documentée par Git inclut le nom et l’adresse électronique utilisés pour les validations. [GIT-CONFIG] [GIT-FIRST-CONFIG]

Ensuite, relancez la commande git commit.

## Enregistrer une deuxième étape de travail

Une seule validation ne constitue pas un historique très utile. L’intérêt apparaît lorsque vous enregistrez des étapes successives et compréhensibles.

Modifiez README.md en ajoutant une ligne :

``markdown J’apprends à créer des validations locales. ``

Puis suivez le même cycle :

``bash git status git add README.md git status git commit -m "Précise l’objectif des notes" ``

Vous obtenez alors un historique composé de deux validations :

``text [Validation la plus récente] Précise l’objectif des notes ¼ [Première validation] Ajoute un fichier de présentation ``

Chaque validation représente une étape du projet. L’objectif n’est pas de valider mécaniquement à intervalles réguliers, mais d’enregistrer une modification suffisamment cohérente pour être comprise et retrouvée plus tard.

## Lire l’historique des validations

Pour afficher l’historique local, utilisez :

``bash git log ``

Git affiche les validations, de la plus récente à la plus ancienne. Le journal contient notamment les identifiants de validation, les informations d’auteur, les dates et les messages de validation. La consultation de l’historique fait partie du flux de travail présenté dans la documentation Git sur l’enregistrement des changements. [GIT-RECORDING-CHANGES]

Une version plus compacte est souvent pratique :

``bash git log --oneline ``

Elle affiche généralement une validation par ligne, avec un identifiant abrégé et son message :

``text abc1234 Précise l’objectif des notes def5678 Ajoute un fichier de présentation ``

Les identifiants affichés sur votre machine seront différents. Ils servent à distinguer de façon précise les validations dans l’historique.

> À retenir  git status décrit le présent : ce qui est modifié, préparé ou non suivi.  git log décrit le passé : les validations déjà enregistrées.

## Rédiger des messages de validation utiles

Git accepte le message que vous lui donnez ; il n’impose pas une formulation universelle. Les règles suivantes sont donc une convention pédagogique destinée à rendre l’historique lisible, pas une exigence technique de Git.

### Une règle simple : décrire l’action réalisée

Rédigez un message court qui exprime ce que la validation apporte. En français, une formulation au présent de l’impératif est pratique :

| Moins utile | Plus utile |
|---|---|
| modif | Ajoute un fichier de présentation |
| test | Corrige le titre du document |
| mise à jour | Précise l’objectif des notes |
| fichiers | Ajoute les notes du premier chapitre |

Un bon message aide une personne — y compris vous-même dans quelques semaines — à comprendre l’intention de la validation sans devoir ouvrir immédiatement tous les fichiers concernés.

### Faire correspondre le message et le contenu

Avant de valider, vérifiez que le message correspond réellement à ce qui a été préparé :

``bash git status ``

Par exemple, si votre message est Corrige une faute dans le titre, la zone de préparation ne devrait idéalement contenir que la correction associée, et non plusieurs changements sans rapport. Cette cohérence rend l’historique plus facile à lire et à utiliser.

### Éviter les validations trop larges au début

Pour apprendre, préférez des étapes simples et séparées :

- ajouter un fichier de présentation ;
- corriger un titre ;
- compléter une section ;
- ajouter une information précise.

Cette approche vous donne un historique explicite et facilite la vérification de ce qui a été enregistré. La possibilité de sélectionner les changements avant la validation est précisément assurée par l'utilisation de git add puis de git commit. [GIT-ADD] [GIT-RECORDING-CHANGES]

## Procédure de référence pour chaque nouvelle modification

Lorsque vous travaillez seul sur un dépôt local, utilisez cette procédure :

```bash

# 1. Vérifier ce qui a changé

git status

# 2. Préparer le ou les fichiers voulus

git add nom-du-fichier

# 3. Vérifier précisément la sélection

git status

# 4. Enregistrer l'étape

git commit -m "Décrit clairement la modification"

# 5. Contrôler le résultat

git status git log --oneline ```
```

Cette séquence ne nécessite aucun compte GitHub et n'envoie rien sur Internet : vous travaillez entièrement dans votre dépôt local. Git est conçu pour enregistrer l'historique des changements dans un dépôt, tandis que les services d'hébergement distant seront abordés ultérieurement. [GITHUB-ABOUT-GIT-FR]

Synthèse

Vous savez maintenant créer et alimenter un historique local :

- git init initialise un dépôt dans un dossier ;
- git status montre l'état des fichiers et de la zone de préparation ;

- git add sélectionne le contenu destiné à la prochaine validation ;
- git commit -m "..." crée une validation à partir de cette sélection ;
- git log permet de relire les validations déjà créées.

La compétence décisive est de ne pas confondre trois moments : modifier, préparer, puis valider. En vérifiant régulièrement l’état du dépôt, vous gardez la maîtrise de ce qui entre dans l’historique.

CHAPITRE 5

Dans un dépôt Git, une erreur n’est pas forcément grave : un fichier modifié par mégarde, un élément ajouté à la zone de préparation ou une validation réalisée trop tôt peuvent généralement être traités. La première règle est toutefois de déterminer précisément l’état du changement avant d’agir.

Git distingue notamment :

- le contenu actuellement présent dans le dossier de travail ;
- les changements placés dans la zone de préparation, c’est-à-dire ceux qui seront inclus dans la prochaine validation ;
- la dernière validation enregistrée dans l’historique.

Une même commande de correction n’agit pas sur ces trois emplacements. C’est pourquoi git status doit être le point de départ habituel : il affiche les fichiers modifiés, ajoutés, supprimés ou non suivis, ainsi que des indications adaptées à leur état. [GIT-STATUS]

> Réflexe de sécurité : avant une commande susceptible d’écarter du contenu, inspectez les changements et, si nécessaire, copiez provisoirement le texte important dans un fichier séparé.

Inspection des changements

Placez-vous dans le dossier du dépôt, puis exécutez :

``bash git status ``

Cette commande répond notamment aux questions suivantes :

- Quels fichiers ont été modifiés ?
- Quels changements sont déjà dans la zone de préparation ?

- Quels fichiers ne sont pas encore suivis par Git ?
- Le dossier de travail est-il propre ?

Git propose également un format court :

``bash git status --short ``

Ce format condense l'état de chaque fichier. Il est utile lorsque la liste est longue, mais le format normal est souvent plus explicite pour une personne qui débute. La documentation de git status décrit ces deux formes d'affichage. [GIT-STATUS]

Lire l'état avant de corriger

Considérez ces situations courantes :

Ce que vous observez	Ce que cela signifie	Première action conseillée
Un fichier est modifié, mais pas préparé	Le contenu a changé dans le dossier de travail	Comparer le changement avant toute correction
Un fichier est préparé	Il fera partie de la prochaine validation si vous validez maintenant	Vérifier précisément son contenu préparé
Un fichier est non suivi	Git ne l'inclut pas encore dans l'historique	Décider s'il doit être ajouté ou ignoré
Un fichier apparaît comme supprimé	Sa suppression sera potentiellement enregistrée	Vérifier que cette suppression est intentionnelle

La commande git status n'enregistre ni ne supprime de contenu : elle sert à observer. Cette séparation entre observation et action réduit fortement les corrections accidentelles. [GIT-STATUS]

Comparaison de versions

Voir qu'un fichier est modifié ne suffit pas toujours. Il faut souvent savoir quelles lignes ont changé. La commande git diff affiche les différences entre les états concernés. La documentation de Git présente cette commande comme l'outil de comparaison des changements dans le dossier de travail et dans la zone de préparation. [GIT-RECORDING-CHANGES]

Comparer le dossier de travail avec la zone de préparation

``bash git diff ``

Cette commande montre les modifications présentes dans le dossier de travail mais qui ne sont pas encore préparées.

Exemple de lecture :

``diff -Title: Draft +Title: Final draft ``

La ligne précédée de - correspond au contenu retiré ; celle précédée de + correspond au contenu ajouté. Ces signes décrivent une différence : ils ne font pas partie du fichier lui-même.

Pour limiter l'affichage à un fichier :

``bash git diff -- README.md ``

Comparer la zone de préparation avec la dernière validation

``bash git diff --staged ``

Cette comparaison répond à une question importante avant une validation : qu'est-ce qui sera réellement enregistré si je lance git commit maintenant ? Les changements préparés et les changements non préparés sont distincts ; il est donc utile de les contrôler séparément. [GIT-RECORDING-CHANGES]

Comparer deux validations

Lorsque vous connaissez les identifiants des validations à comparer, vous pouvez écrire :

``bash git diff <validation-1> <validation-2> ``

Vous pouvez aussi comparer un fichier entre ces deux points de l'historique :

``bash git diff <validation-1> <validation-2> -- README.md ``

Ici, remplacez <validation-1> et <validation-2> par les identifiants de validation concernés. Une validation est aussi appelée commit dans la terminologie Git.

> À retenir : git diff n'enregistre aucune modification. C'est une commande d'inspection, particulièrement utile avant une action irréversible ou difficile à annuler.

Fichiers à ignorer

Un dépôt contient parfois des fichiers utiles localement mais qui ne doivent pas être ajoutés à l'historique : fichiers temporaires, sorties de compilation, journaux d'exécution, paramètres propres à une machine ou informations sensibles.

Git utilise notamment un fichier nommé .gitignore pour définir des motifs de fichiers à ignorer. Les règles d'ignorance permettent d'éviter que des fichiers non suivis apparaissent comme candidats à l'ajout. [GIT-RECORDING-CHANGES]

Créer un fichier .gitignore

À la racine du dépôt, créez un fichier .gitignore :

```gitignore
# Fichiers temporaires
*.tmp
```

Journaux

*.log

Dossier généré localement

build/

Configuration locale

.env ```

Dans cet exemple :

- *.tmp ignore les fichiers dont le nom se termine par .tmp ;
- *.log ignore les fichiers dont le nom se termine par .log ;
- build/ ignore le dossier build ;
- .env ignore un fichier portant exactement ce nom.

Les commentaires commencent par #. Les règles suivent un ordre : une règle plus basse peut préciser ou neutraliser une règle précédente selon la syntaxe des motifs. La documentation Git détaille les motifs, les exceptions et les emplacements possibles des règles d’ignorance. [GIT-RECORDING-CHANGES]

> Encadré — Ne publiez pas par défaut Évitez d’ajouter sans vérification des fichiers contenant des mots de passe, des jetons d’accès, des clés privées ou une configuration réservée à votre machine. Un fichier .gitignore aide à prévenir un ajout involontaire, mais il ne remplace pas une vérification de git status avant chaque validation.

Limite importante : un fichier déjà suivi n’est pas ignoré automatiquement

Les règles de .gitignore s’appliquent principalement aux fichiers que Git ne suit pas encore. Si un fichier a déjà été ajouté et validé, l’ajouter ensuite à .gitignore ne le retire pas de l’historique ni de l’index de suivi. Les mécanismes d’ignorance ne constituent donc pas une procédure de nettoyage d’un fichier déjà enregistré. [GIT-RECORDING-CHANGES]

Avant de partager un dépôt, vérifiez notamment si un fichier sensible a déjà été validé. Dans ce cas, ne vous contentez pas de l’ajouter à .gitignore : la suite des actions dépend du contexte de partage et nécessite une procédure adaptée.

Renommage et suppression de fichiers suivis

Git détecte l’état des fichiers suivis dans le dépôt. Après avoir supprimé ou renommé un fichier dans votre explorateur de fichiers ou dans votre éditeur, git status vous permet de constater le changement avant de le préparer. [GIT-STATUS]

Supprimer un fichier suivi

Pour supprimer un fichier du dossier de travail et préparer cette suppression :

``bash git rm ancien-fichier.txt ``

Après cette commande, contrôlez l'état :

``bash git status ``

La suppression apparaît alors parmi les changements prêts à être validés. Git documente git rm dans le flux d'enregistrement des changements. [GIT-RECORDING-CHANGES]

Renommer un fichier suivi

Pour renommer un fichier à l'aide de Git :

``bash git mv ancien-nom.txt nouveau-nom.txt ``

Puis vérifiez :

``bash git status ``

Cette commande effectue le renommage puis prépare le changement correspondant. [GIT-RECORDING-CHANGES]

Vous pouvez aussi renommer ou supprimer un fichier avec votre système d'exploitation, puis utiliser git status et git add pour préparer le résultat. La commande git add sert à ajouter le contenu de fichiers à la zone de préparation. [GIT-ADD]

Correction de changements non enregistrés

Avant de modifier une validation existante, identifiez l'emplacement de l'erreur. Dans beaucoup de cas, le changement n'est pas encore validé : il se trouve soit dans le dossier de travail, soit dans la zone de préparation.

Cas 1 — Conserver le changement, mais ne pas le préparer tout de suite

Si un fichier a été préparé trop tôt, vous pouvez le retirer de la zone de préparation sans vouloir supprimer votre travail local. Les outils de Git permettent de séparer le contenu préparé du contenu présent dans le dossier de travail ; la documentation de Git décrit cette distinction dans le cycle d'enregistrement des changements. [GIT-RECORDING-CHANGES]

Pour une personne débutante, la priorité est de contrôler les deux vues suivantes :

``bash git diff git diff --staged ``

Si le changement important apparaît dans la seconde vue, il est préparé. Si vous n'êtes pas certain de la commande de retrait adaptée à votre version de Git, ne lancez pas d'action destructive : conservez le fichier, consultez l'aide intégrée et vérifiez le résultat avec git status.

``bash git help restore ``

Cas 2 — Abandonner un changement non préparé

Si vous voulez réellement retrouver la version du fichier telle qu'elle était avant votre modification locale, commencez par afficher la différence :

``bash git diff -- README.md ``

Assurez-vous que vous n'avez besoin d'aucune des lignes affichées. Les commandes de restauration peuvent remplacer le contenu local d'un fichier par une version antérieure ; une telle opération doit donc être précédée d'une inspection attentive. Les mécanismes de restauration et d'annulation font partie du travail de gestion des changements décrit par Git. [GIT-RECORDING-CHANGES]

> Règle pratique : si vous hésitez entre « je veux enlever ce fichier de la prochaine validation » et « je veux détruire mes modifications locales », arrêtez-vous. La première intention concerne la zone de préparation ; la seconde concerne le contenu du fichier dans le dossier de travail. Ce ne sont pas les mêmes opérations.

Cas 3 — Ajouter seulement une partie d'un fichier

Un même fichier peut contenir plusieurs modifications, dont certaines ne doivent pas encore être validées. Git permet une préparation interactive par portions de changement avec :

``bash git add -p ``

Cette forme de git add propose de traiter des groupes de lignes séparément. Elle peut être utile lorsque vous voulez créer une validation claire et limitée à une intention. [GIT-ADD]

Pour débuter, utilisez cette commande avec prudence : lisez chaque proposition, puis vérifiez le résultat avec git diff --staged avant de valider.

Correction de validations : scénarios à distinguer

Une validation est déjà inscrite dans l'historique local. La façon de la corriger dépend surtout de deux critères :

1. s'agit-il de la dernière validation ou d'une validation plus ancienne ?
2. cette validation a-t-elle déjà été partagée avec d'autres personnes ?

La référence HEAD désigne la validation actuellement active dans votre historique local. Les opérations qui modifient la dernière validation ou réorganisent des validations précédentes doivent être distinguées des opérations qui ajoutent simplement une nouvelle correction. Git documente notamment la modification de la dernière validation dans son guide sur l'enregistrement des changements. [GIT-RECORDING-CHANGES]

Matrice de décision

Situation	Intention	Approche prudente
Erreur non validée	Relire ou conserver le travail	Utiliser git status et git diff
Fichier préparé par erreur	Le conserver localement, sans l'inclure dans la prochaine validation	Retirer le fichier de la zone de préparation, puis vérifier l'état
Dernière validation locale incomplète	Ajouter ou corriger son contenu avant partage	Examiner la possibilité de modifier la dernière validation
Validation déjà partagée	Corriger sans surprendre les autres personnes	Préférer une nouvelle validation de correction et suivre les règles de l'équipe
Historique partagé à réécrire	Modifier ou supprimer des validations existantes	Ne pas agir sans accord explicite et sans procédure vérifiée

Corriger la dernière validation locale

Lorsque la dernière validation n'a pas encore été partagée, Git permet de la modifier avec l'option --amend de git commit. Cette option remplace la dernière validation par une nouvelle version de celle-ci. [GIT-RECORDING-CHANGES]

Flux général :

```bash
# Préparer les fichiers corrigés
git add README.md
# Modifier la dernière validation
git commit --amend ```

Cette commande ouvre normalement l'éditeur de message de validation afin de permettre la conservation ou la modification du message. Vérifiez auparavant le contenu préparé :

``bash git diff --staged ``

L'option --amend est adaptée à une dernière validation locale que vous souhaitez compléter ou corriger immédiatement. Elle ne doit pas être traitée comme une simple correction anodine si la validation a déjà été partagée : elle crée une nouvelle version de la dernière validation. [GIT-RECORDING-CHANGES]
```

Corriger une validation déjà partagée

Dès qu'une validation a été communiquée à d'autres personnes, elle peut faire partie de leur historique local ou de leur travail en cours. Dans ce contexte, modifier l'historique existant peut compliquer la synchronisation.

L'approche la plus lisible consiste souvent à ajouter une nouvelle validation qui corrige explicitement le problème, plutôt qu'à remplacer silencieusement une validation déjà connue. La commande précise à employer et la procédure de partage doivent être choisies selon les pratiques du projet ; elles ne sont pas détaillées par les sources enregistrées pour ce chapitre.

Précautions avant toute opération de réécriture d'historique

Une réécriture d'historique consiste à modifier, remplacer, déplacer ou supprimer des validations déjà présentes dans la ligne d'historique. Modifier la dernière validation avec git commit --amend est déjà un exemple de remplacement de validation. [GIT-RECORDING-CHANGES]

Avant une telle opération, vérifiez systématiquement :

1. Le périmètre : quel fichier ou quelle validation souhaitez-vous réellement corriger ?
2. L'état local : que montrent git status, git diff et git diff --staged ?
3. Le partage : la validation concernée a-t-elle déjà été communiquée à d'autres personnes ?
4. La sauvegarde du contenu utile : avez-vous conservé ailleurs les modifications que vous ne voulez pas perdre ?
5. La règle de l'équipe : le projet autorise-t-il cette modification de l'historique ?

> Principe de prudence : plus un changement est ancien et plus il est partagé, moins il faut chercher à le faire disparaître de l'historique. Une correction ajoutée de manière explicite est souvent plus compréhensible pour les autres contributeurs qu'un historique réécrit.

Repères opérationnels

Pour gérer sereinement les fichiers d'un dépôt local, retenez cette séquence :

```bash
# 1. Observer
git status

# 2. Lire les changements non préparés
git diff

# 3. Lire les changements préparés
git diff --staged
```

4. Préparer volontairement les fichiers souhaités

git add <fichier>

5. Contrôler une dernière fois

git status git diff --staged ```

Cette routine réduit les validations involontaires. Les fichiers à ne pas suivre doivent être définis dans .git-ignore avant leur ajout, tandis que les corrections de validations doivent être choisies selon leur caractère local ou déjà partagé. [GIT-ADD] [GIT-RECORDING-CHANGES] [GIT-STATUS]

CHAPITRE 6

Une branche permet d'ouvrir une ligne de travail séparée dans un même dépôt. Elle sert à développer une évolution, corriger un problème ou essayer une idée sans modifier immédiatement la ligne de développement de référence. GitHub définit une branche comme une version parallèle d'un dépôt, distincte de la branche principale, qui permet de travailler sans affecter cette dernière. [GITHUB-GLOSSARY-FR]

Dans ce chapitre, les commandes sont exécutées dans le dossier d'un dépôt Git existant.

> Idée essentielle : une branche n'est pas une copie indépendante de tout le projet. C'est un repère qui désigne une suite de validations (commits) dans l'historique. Lorsque vous changez de branche, Git met votre dossier de travail dans l'état correspondant à cette branche.

Pourquoi créer une branche ?

Travailler directement sur la branche principale convient à une expérimentation très courte, mais devient vite risqué dès qu'une modification mérite d'être isolée. Une branche de travail donne un espace dédié pour :

- ajouter une fonctionnalité ;
- corriger un défaut ;
- mettre à jour une partie de la documentation ;
- tenter une solution avant de décider de la conserver ;
- avancer sur plusieurs sujets sans mélanger leurs modifications.

L'intérêt ne se limite pas au travail à plusieurs. Même seul, vous pouvez préserver une branche principale stable pendant qu'une évolution reste en cours.

Notion de branche

Supposons qu’un projet possède une branche principale appelée main. Elle contient trois validations :

``text A --- B --- C  main ``

Vous créez une branche nommée ajout-bienvenue à partir de C. Au moment de sa création, les deux noms désignent la même validation :

``text A --- B --- C  main, ajout-bienvenue ``

Vous vous placez ensuite sur ajout-bienvenue et créez deux validations. La branche de travail avance, tandis que main reste au même point :

``text A --- B --- C  main \ D --- E  ajout-bienvenue ``

Cette divergence est normale : elle représente deux lignes d’historique qui ont évolué depuis une base commune.

Branche actuelle et référence HEAD

Git conserve une indication de la branche actuellement utilisée. Cette indication est souvent représentée par le nom HEAD dans les schémas et certains messages affichés par Git.

Lorsque vous êtes sur ajout-bienvenue :

- les fichiers visibles dans votre dossier de travail correspondent à cette branche ;
- une nouvelle validation est ajoutée à cette branche ;
- la branche main ne reçoit pas cette validation.

Avant de changer de branche, il est prudent de vérifier l’état du dépôt :

``bash git status ``

La commande affiche notamment la branche courante ainsi que les modifications qui ne sont pas encore enregistrées dans une validation. [GIT-STATUS]

> Réflexe à adopter : avant toute opération de branchement ou d’intégration, lancez git status. Un dossier de travail propre rend les résultats plus prévisibles et les erreurs plus faciles à comprendre.

Branche principale et branches de travail

La branche principale est généralement celle qui représente l’état de référence du projet. Son nom est souvent main, mais un dépôt peut utiliser un autre nom. Ne supposez donc pas automatiquement que main existe : consultez les branches du dépôt.

``bash git branch ``

Cette commande affiche les branches locales. L’astérisque identifie la branche active :

```text
  • main
```

Une branche de travail porte un nom qui exprime clairement son objectif. Dans ce cours, les exemples utilisent des noms descriptifs avec des mots séparés par des tirets :

``text ajout-bienvenue correction-typo-accueil mise-a-jour-documentation ``

Ces exemples sont une convention pédagogique, et non une règle imposée par Git. L'essentiel est d'adopter des noms compréhensibles pour les personnes qui liront l'historique.

Une évolution par branche

Une branche est plus lisible lorsqu'elle correspond à un objectif limité. Par exemple :

Situation	Choix conseillé
Ajouter un message de bienvenue	Créer ajout-bienvenue
Corriger une faute dans un fichier déjà publié	Créer correction-typo-accueil
Modifier simultanément le message, la navigation et les images	Découper en plusieurs branches si les sujets sont indépendants
Faire une expérience provisoire	Créer une branche dédiée, puis l'intégrer ou la supprimer selon le résultat

Évitez de rassembler dans une même branche une correction urgente, une refonte importante et une modification de documentation sans lien. Si un problème apparaît, il devient plus difficile de savoir quelle modification est responsable.

Création et navigation entre branches

Créer une branche

Pour créer une branche sans vous y déplacer immédiatement :

``bash git branch ajout-bienvenue ``

Pour constater qu'elle existe :

``bash git branch ``

Exemple de résultat :

```text ajout-bienvenue
  • main
```

```

À ce stade, vous êtes toujours sur main : l’astérisque reste devant main.

### Changer de branche

La commande suivante permet de basculer vers une branche existante :

``bash git switch ajout-bienvenue ``

Vérifiez ensuite votre position :

``bash git status ``

Vous devez voir que la branche actuelle est ajout-bienvenue. La documentation Git présente Git comme un système de gestion de versions dont les commandes permettent de travailler avec l’historique et les différentes lignes de développement d’un dépôt. [GIT-OVERVIEW]

### Créer une branche et s’y placer

Pour créer la branche et y basculer dans la même opération :

``bash git switch -c ajout-bienvenue ``

La séquence de travail devient alors :

```bash

# Partir de la branche de référence

git switch main

# Vérifier l’état courant

git status

# Créer une branche de travail et s’y placer

git switch -c ajout-bienvenue ```

Ensuite, modifiez les fichiers concernés, préparez les changements et créez des validations sur cette branche. Les commandes d’ajout et de validation suivent les mêmes principes que dans le reste du dépôt : git add prépare le contenu destiné à la prochaine validation. [GIT-ADD]

### Tableau des opérations courantes

| Intention | Commande | Effet attendu |
|---|---|---|
| Voir les branches locales | git branch | Affiche les noms des branches et marque la branche active |
| Créer une branche | git branch nom-branche | Crée une branche sans changer de branche active |
| Aller sur une branche existante | git switch nom-branche | Change la branche active et l'état des fichiers suivis |
| Créer et utiliser une branche | git switch -c nom-branche | Crée la branche, puis la rend active |
| Vérifier la situation | git status | Indique notamment la branche courante et l'état des fichiers [GIT-STATUS] |

> Attention : le changement de branche peut être refusé si vos modifications locales risquent d'être écrasées par le contenu de la branche cible. Dans ce cas, ne forcez pas l'opération sans comprendre le message. Enregistrez d'abord votre travail dans une validation, ou revenez à un état de travail cohérent.

## Historique divergent

Une branche de travail devient utile lorsque son historique progresse différemment de celui de la branche principale.

Imaginons le point de départ suivant :

``text A --- B --- C  main ``

Vous créez ajout-bienvenue, puis vous y effectuez une validation D :

``text A --- B --- C  main \ D  ajout-bienvenue ``

Pendant ce temps, une validation E peut être ajoutée sur main :

``text A --- B --- C --- E  main \ D  ajout-bienvenue ``

Les deux branches ont maintenant divergé. Cela ne signifie pas qu'il y a une erreur : elles conservent simplement des évolutions distinctes depuis leur dernier point commun, ici C.

Le rôle de l'intégration est de réunir, lorsque cela est souhaité, les changements de la branche de travail dans la branche qui doit les recevoir.

## Intégration de changements

Intégrer des changements signifie incorporer l'historique d'une branche dans une autre. Dans ce chapitre, l'objectif est d'intégrer ajout-bienvenue dans main.

La règle à retenir est simple : placez-vous d'abord sur la branche qui doit recevoir les changements, puis lancez l'intégration.

```bash

# Se placer sur la branche destinataire

git switch main

# Vérifier que le dépôt est dans un état compréhensible

git status

# Intégrer les changements de la branche de travail

git merge ajout-bienvenue ```

Après cette opération, les changements de ajout-bienvenue font partie de l'historique de main.

### Deux résultats possibles à connaître

Selon la forme de l'historique, Git peut intégrer les changements de différentes manières.

#### Avance rapide

Si main n'a pas évolué depuis la création de la branche de travail, Git peut avancer directement le repère de main jusqu'à la dernière validation de la branche de travail :

```text Avant : A --- B --- C  main \ D --- E  ajout-bienvenue

Après : A --- B --- C --- D --- E  main, ajout-bienvenue ```

L'historique reste linéaire.

#### Validation de fusion

Si les deux branches ont reçu des validations depuis leur point commun, Git peut créer une validation qui réunit les deux lignes :

```text Avant : A --- B --- C --- E  main \ D --- F  ajout-bienvenue

Après : A --- B --- C --- E ------- M  main \  / D --- F ---------  ajout-bienvenue ```

M représente une validation de fusion. Elle relie les deux historiques.

Le vocabulaire GitHub distingue notamment une branche, une fusion et un conflit de fusion dans son glossaire. [GITHUB-GLOSSARY-FR]
```

Vérifier après l’intégration

Après une intégration réussie, vérifiez au minimum :

``bash git status ``

Puis ouvrez ou exécutez le projet selon sa nature : relisez le fichier modifié, lancez l’application ou utilisez les vérifications prévues par le projet. Git peut confirmer que l’intégration technique s’est terminée, mais il ne peut pas décider à votre place si le résultat répond au besoin fonctionnel.

Lorsque la branche de travail n’est plus nécessaire, vous pouvez la supprimer localement :

``bash git branch -d ajout-bienvenue ``

Ne supprimez une branche qu’après avoir confirmé que son contenu utile a bien été intégré, ou qu’il ne doit pas être conservé.

Introduction aux conflits

Un conflit survient lorsque Git ne peut pas déterminer automatiquement quelle version conserver lors d'une intégration. GitHub décrit le conflit de fusion comme une situation dans laquelle des modifications concurrentes doivent être résolues avant que la fusion puisse être terminée. [GITHUB-GLOSSARY-FR]

Un cas typique se produit lorsque deux branches modifient la même zone d'un même fichier de manière incompatible.

```text Version commune : Bonjour

Dans main : Bonjour à toutes et à tous

Dans ajout-bienvenue : Bienvenue dans le projet ```

Si Git ne peut pas combiner ces deux modifications sans ambiguïté, il interrompt l'intégration et signale le conflit. Ce n'est pas un échec irréversible : c'est une demande de décision humaine.

> Un conflit n'indique pas que Git est défaillant. Il indique que plusieurs versions possibles existent et que Git ne peut pas choisir laquelle exprime l'intention du projet.

## Les marqueurs de conflit

Dans un fichier en conflit, Git insère des marqueurs semblables à ceux-ci :

```text <<<<<<< HEAD Bonjour à toutes et à tous ======= Bienvenue dans le projet

> >>>>>> ajout-bienvenue

```

Dans cet exemple :

- la partie située entre <<<<<<< HEAD et ======= correspond au contenu de la branche qui reçoit l'intégration ;
- la partie située entre ======= et >>>>>>> ajout-bienvenue correspond au contenu apporté par l'autre branche ;
- les lignes de marqueurs ne doivent pas rester dans le fichier final.

Ne choisissez pas automatiquement un bloc parce qu'il est affiché en premier. Lisez les deux versions, comprenez leur objectif, puis écrivez le contenu final voulu.

Principes de résolution et de vérification

La résolution d'un conflit suit une séquence méthodique.
```

1. Identifier les fichiers concernés

Après l'arrêt de l'intégration, exécutez :

``bash git status ``

La commande indique les fichiers qui nécessitent une intervention. [GIT-STATUS]

2. Ouvrir chaque fichier en conflit

Recherchez les marqueurs :

```text <<<<<<< =======

> >>>>>>

```

Lisez le contexte autour du conflit. Une ligne isolée peut sembler facile à choisir, mais sa signification dépend souvent des lignes voisines.

3. Produire le contenu final

Modifiez le fichier de façon à conserver :

- soit la version de la branche destinataire ;
- soit la version de la branche intégrée ;
- soit une combinaison cohérente des deux ;
- soit une nouvelle formulation plus adaptée.

Supprimez tous les marqueurs de conflit.

Par exemple, le résultat final peut être :

``text Bienvenue à toutes et à tous dans le projet ``

4. Marquer le conflit comme résolu

Une fois le fichier corrigé, préparez-le :

``bash git add nom-du-fichier ``

La commande git add ajoute le contenu sélectionné à l'index en vue de la prochaine validation. [GIT-ADD]

Répétez l'opération pour chaque fichier concerné, puis contrôlez l'état :

``bash git status ``

5. Terminer l’intégration

Lorsque tous les conflits sont résolus et préparés, terminez l’intégration avec :

``bash git commit ``

Git ouvre généralement un message de validation proposé pour la fusion. Relisez-le avant de confirmer.

6. Vérifier réellement le résultat

Une résolution est terminée lorsque le dépôt est cohérent et que le projet fonctionne comme attendu. Vérifiez notamment :

- qu’aucun marqueur <<<<<<<, ======= ou >>>>>>> ne subsiste dans les fichiers modifiés ;
- que git status ne signale plus de conflit ;
- que le contenu final correspond bien à l’intention des deux branches ;
- que les contrôles disponibles pour le projet passent correctement.

Abandonner une intégration en cours

Si vous constatez que vous avez lancé l’intégration au mauvais moment ou que vous ne comprenez pas encore le conflit, n’effacez pas des fichiers au hasard. Git fournit une commande pour abandonner une fusion en cours et revenir à l’état antérieur à la tentative :

``bash git merge --abort ``

Utilisez-la comme une décision consciente : elle annule la tentative d’intégration en cours, mais elle ne remplace pas la compréhension du travail qui doit ensuite être repris.

Méthode de travail recommandée

Pour une évolution simple, appliquez ce déroulé :

1. Vérifier l’état du dépôt avec git status.
2. Se placer sur la branche de référence, par exemple main.
3. Créer une branche descriptive avec git switch -c nom-branche.
4. Réaliser l’évolution et créer des validations cohérentes sur cette branche.
5. Revenir sur la branche destinataire avec git switch main.
6. Vérifier l’état avec git status.
7. Intégrer la branche avec git merge nom-branche.
8. Résoudre les conflits éventuels, puis vérifier le résultat.
9. Supprimer la branche locale devenue inutile seulement après validation de l’intégration.

Cette discipline maintient l’historique lisible et limite le risque de mélanger des changements sans rapport. Les branches deviennent alors un outil de sécurité, d’organisation et de collaboration, avant même l’utilisation d’une plateforme distante telle que GitHub.

CHAPITRE 7

GitHub : de l’historique local à un espace partagé

Jusqu’ici, le dépôt Git a vécu sur votre ordinateur : il contient vos fichiers, vos validations et l’historique de votre projet. GitHub ajoute un espace en ligne dans lequel un dépôt peut être hébergé, consulté et partagé.

Git et GitHub ne sont pas la même chose. Git est le système qui enregistre l’historique des modifications d’un projet. GitHub est une plateforme qui permet notamment d’héberger des dépôts Git et d’organiser la collaboration autour de ces dépôts. [GITHUB-ABOUT-GIT-FR]

Pour utiliser GitHub, vous créez un compte sur la plateforme. La création de ce compte relève de la gestion de compte GitHub ; elle est distincte de l’installation et de la configuration de Git sur votre ordinateur. [GITHUB-ACCOUNT-CREATION] [GITHUB-SET-UP-GIT]

> Idée essentielle — Le dépôt local reste votre copie de travail sur votre ordinateur. Le dépôt distant est une copie du projet accessible via un service tel que GitHub. Les deux dépôts peuvent être reliés afin d’échanger l’historique et les modifications.

Le rôle d’un dépôt distant

Un dépôt distant est une version du dépôt, située hors de votre ordinateur et accessible par un réseau. Dans un projet individuel, il peut servir à publier ou à sauvegarder le travail. Dans un projet collectif, il devient un point de référence commun : les personnes autorisées peuvent y consulter le projet et y contribuer selon les droits qui leur sont attribués. Le vocabulaire GitHub distingue notamment le dépôt, le dépôt distant, le clonage et la duplication (fork). [GITHUB-GLOSSARY-FR]

Il est utile de représenter les deux espaces séparément :

Espace	Où se trouve-t-il ?	Rôle principal
Dépôt local	Sur votre ordinateur	Travailler sur les fichiers et construire l'historique local
Dépôt distant	Sur GitHub	Partager, publier ou centraliser une copie du projet

La présence d'un dépôt distant ne remplace pas le dépôt local. Git continue de fonctionner localement : vous pouvez consulter l'état d'un dépôt et enregistrer des modifications sur votre machine. [GIT-STATUS] [GIT-RECORDING-CHANGES]

En revanche, dès qu'un projet est destiné à être visible ou partagé, le dépôt distant devient le lieu de référence que les autres personnes pourront ouvrir dans GitHub.

Lire l'interface d'un dépôt GitHub

Après avoir ouvert un dépôt sur GitHub, l'objectif n'est pas de mémoriser toute l'interface immédiatement. Commencez par reconnaître les zones qui répondent aux questions de base : quel projet est affiché, quels fichiers contient-il, quel historique possède-t-il et comment les personnes peuvent-elles participer ?

Une page de dépôt présente généralement les éléments suivants :

Élément à identifier	À quoi sert-il pour débuter ?
Nom du propriétaire et nom du dépôt	Identifier précisément le projet consulté
Description du projet	Comprendre rapidement l'objectif annoncé
Liste des fichiers	Parcourir le contenu actuellement visible du dépôt
Historique des validations	Voir que le projet évolue par étapes enregistrées
Branches	Repérer les lignes de développement disponibles
Fichier de présentation	Lire les premières informations destinées aux visiteurs
Informations de contribution ou de discussion	Comprendre comment la collaboration est organisée, lorsque ces fonctions sont utilisées

Le glossaire GitHub recense ces notions, notamment les dépôts, les branches, les validations (commits), les problèmes signalés (issues) et les demandes de fusion (pull requests). [GITHUB-GLOSSARY-FR]

Au début, ne confondez pas :

- le contenu du dépôt : fichiers et dossiers du projet ;
- l'historique : la suite des validations qui ont conduit à cet état ;

- les branches : des lignes de travail distinctes ;
- les outils de collaboration : mécanismes utilisés pour discuter, signaler un travail ou proposer une intégration.

Les branches ont déjà été abordées du point de vue local. Sur GitHub, elles deviennent aussi visibles aux autres membres du projet, une fois que le dépôt local et le dépôt distant sont effectivement reliés et synchronisés.

Créer un dépôt distant : commencer par une décision de publication

Créer un dépôt sur GitHub consiste d'abord à définir ce que vous souhaitez rendre disponible. Avant toute publication, préparez une réponse claire aux questions suivantes :

1. Quel projet vais-je publier ?
2. Le dépôt contient-il des informations qui ne doivent pas être partagées ?
3. À qui le projet doit-il être accessible ?
4. Est-ce un nouveau projet en ligne ou la publication d'un dépôt local existant ?
5. Quel nom permettra d'identifier le projet sans ambiguïté ?

Ces décisions précèdent les manipulations techniques. Un dépôt est un conteneur d'historique : publier un projet ne revient donc pas seulement à montrer son dernier fichier, mais potentiellement à exposer les fichiers et validations présents dans l'historique que vous transférez.

Le cas le plus simple : un nouveau projet

Pour un nouveau projet, le dépôt distant peut être créé avant le dépôt local. Vous disposez alors d'un emplacement en ligne identifié, auquel vous associerez ensuite une copie de travail locale.

Le cas le plus fréquent après les premiers chapitres : publier un dépôt local existant

Dans ce cours, vous possédez déjà un dépôt local avec des validations. Le scénario visé est donc généralement le suivant :

1. créer un dépôt distant sur GitHub ;
2. relever son adresse de connexion ;
3. associer cette adresse au dépôt local ;
4. transférer l'historique local vers le dépôt distant ;
5. vérifier dans l'interface GitHub que les fichiers et validations attendus sont visibles.

> Ne créez pas deux histoires par inadvertance. Si votre projet local contient déjà des validations, traitez-le comme la source de l'historique que vous souhaitez publier. Avant d'initialiser du contenu en ligne, vérifiez avec soin la procédure proposée par GitHub pour ce scénario précis.

Visibilité : une décision à prendre avec prudence

Les réglages de visibilité déterminent qui peut accéder à un dépôt. GitHub emploie notamment les notions de dépôts publics et privés dans son vocabulaire. [GITHUB-GLOSSARY-FR]

Pour une personne débutante, la bonne approche consiste à raisonner à partir de l'audience :

Situation envisagée	Question de prudence
Projet d'apprentissage à montrer	Le contenu peut-il être consulté par l'audience que vous visez ?
Projet réalisé avec d'autres personnes	Toutes les personnes concernées ont-elles convenu du partage ?
Projet professionnel ou personnel sensible	Le dépôt contient-il des données, identifiants, documents ou historiques qui ne doivent pas être publiés ?
Code récupéré d'un autre projet	Avez-vous le droit de le redistribuer ou de le rendre visible ?

N'utilisez jamais un dépôt comme un lieu de stockage pour des mots de passe, clés d'accès ou autres informations secrètes. Si vous découvrez qu'un élément sensible a été ajouté, n'interprétez pas sa suppression dans le dernier fichier comme une garantie : le contenu peut avoir été enregistré dans une validation antérieure. Dans ce cas, interrompez la publication et appliquez la procédure appropriée à votre contexte.

La sécurité du compte GitHub mérite également une attention distincte de la visibilité du dépôt. GitHub documente l'authentification à deux facteurs et les méthodes de récupération associées. [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY]

Relier le dépôt local au dépôt distant

Relier les deux dépôts signifie enregistrer, dans la configuration locale, l'adresse du dépôt distant et un nom local qui permet de le désigner. Ce nom est un alias : il facilite les échanges entre votre dépôt local et un dépôt hébergé. La notion de dépôt distant fait partie du vocabulaire GitHub. [GITHUB-GLOSSARY-FR]

Le modèle mental à retenir est le suivant :

1. votre dossier de projet contient un dépôt Git local ;
2. GitHub héberge un dépôt distant ;
3. votre configuration Git locale connaît l'adresse de ce dépôt distant ;
4. des opérations de synchronisation permettent ensuite d'envoyer ou de récupérer des validations.

La configuration Git peut être définie à différents niveaux, et GitHub recommande de configurer Git avant de l'utiliser avec GitHub. [GIT-CONFIG] [GITHUB-SET-UP-GIT]

Vocabulaire de la synchronisation

Terme	Sens pratique
Dépôt local	Dépôt dans votre dossier de travail
Dépôt distant	Dépôt accessible sur un serveur ou une plateforme telle que GitHub
Adresse du dépôt	Adresse utilisée pour identifier le dépôt distant lors de la connexion
Envoyer des validations	Transférer des validations locales vers le dépôt distant
Récupérer des validations	Ramener dans le dépôt local des modifications présentes sur le dépôt distant
Synchroniser	Mettre en cohérence, selon l'opération effectuée, les deux historiques

La connexion entre un dépôt local existant et un dépôt GitHub dépend de choix techniques qui doivent être vérifiés dans la documentation officielle actuelle : méthode d'authentification, adresse de connexion choisie et état initial du dépôt distant. N'effectuez pas ces opérations en recopiant une commande isolée sans comprendre quel dépôt local et quel dépôt distant elle cible.

Contrôles avant la première synchronisation

Avant de relier ou de publier un dépôt, vérifiez les points suivants dans votre terminal :

- vous êtes placé dans le bon dossier de projet ;
- ce dossier est bien un dépôt Git ;
- l'état de travail est compris : fichiers modifiés, fichiers non suivis et validations à enregistrer ;
- vos nom et adresse électronique Git sont correctement configurés pour les validations à venir ;
- vous connaissez précisément l'adresse du dépôt GitHub ciblé.

Les commandes git status et la configuration Git aident à contrôler respectivement l'état du dépôt et les paramètres de configuration. [GIT-STATUS] [GIT-CONFIG] [GIT-FIRST-CONFIG]

> Réflexe de sécurité — Avant toute commande qui transfère des données, relisez l'adresse du dépôt distant et confirmez que le nom du projet affiché dans GitHub est bien celui que vous voulez utiliser.

Vérifier le résultat dans GitHub

Une fois le dépôt local relié et les validations transférées selon la procédure officielle applicable, ouvrez le dépôt dans GitHub. Votre vérification doit porter sur le contenu, mais aussi sur l'historique.

Utilisez cette liste de contrôle :

- le nom et le propriétaire du dépôt sont ceux attendus ;
- la visibilité correspond à votre décision de publication ;
- les fichiers visibles correspondent au projet voulu ;
- aucun fichier sensible n'apparaît ;
- les validations attendues apparaissent dans l'historique ;
- la branche attendue est visible ;
- le fichier de présentation du projet est lisible et utile.

Cette vérification complète la commande locale : git status renseigne votre copie de travail, tandis que l'interface GitHub vous permet d'inspecter ce qui est effectivement présent dans le dépôt distant. [GIT-STATUS]

Les fichiers qui expliquent le projet

Un dépôt utile n'est pas seulement une collection de fichiers. Une personne qui le découvre doit pouvoir comprendre ce qu'il contient et comment l'utiliser. GitHub désigne notamment le fichier README comme un élément important du vocabulaire des dépôts. [GITHUB-GLOSSARY-FR]

Le rôle du fichier README

Le fichier README est le point de départ naturel pour présenter le projet. Pour un projet débutant, il peut répondre sobrement à ces questions :

- Quel est le nom du projet ?
- Quel problème ou quel objectif poursuit-il ?
- Que contient le dépôt ?
- Comment consulter, lancer ou utiliser le projet, si cela s'applique ?
- Quel est son état : exercice, démonstration, projet en cours ou projet terminé ?

Un premier README peut rester très court. Sa valeur vient de sa clarté, non de sa longueur.

```markdown
# Nom du projet

Courte description de l'objectif du projet.
```

Contenu

- description du dossier ou fichier principal

État

Projet d'apprentissage en cours. ```

Cet exemple est un modèle de rédaction : adaptez-le au contenu réel de votre dépôt et n’annoncez pas de fonctionnalités qui n’existent pas.

Autres informations de présentation

Selon le projet, vous pourrez ensuite ajouter des informations sur les règles de contribution, la licence ou la manière de signaler un problème. GitHub emploie des termes spécifiques pour ces éléments et pour les mécanismes de collaboration associés. [GITHUB-GLOSSARY-FR]

Pour l’instant, retenez une règle simple : un dépôt destiné à être partagé doit pouvoir être compris sans explication orale de son auteur.

À retenir

- Git fonctionne dans votre dépôt local ; GitHub peut héberger un dépôt distant associé à ce travail. [GITHUB-ABOUT-GIT-FR]
- Un dépôt distant constitue un point de partage, de publication ou de collaboration, sans supprimer votre copie locale. [GITHUB-GLOSSARY-FR]
- Avant de publier, décidez explicitement du contenu, de l’audience et de la visibilité du dépôt.
- La liaison entre un dépôt local et GitHub repose sur l’identification exacte du dépôt distant et sur une procédure de synchronisation adaptée à l’état initial des deux dépôts.
- Dans GitHub, vérifiez à la fois les fichiers visibles, l’historique des validations, les branches et les informations de présentation.
- Un fichier README clair transforme un dépôt brut en projet compréhensible. [GITHUB-GLOSSARY-FR]

Le chapitre suivant pourra s’appuyer sur cette relation entre dépôt local et dépôt distant pour détailler les opérations de synchronisation et les pratiques de collaboration, à partir de procédures officiellement vérifiées.

CHAPITRE 8

Après avoir créé un dépôt distant sur GitHub, le projet existe à deux endroits :

- le dépôt local, sur votre ordinateur ;
- le dépôt distant, hébergé sur GitHub.

La synchronisation consiste à faire circuler les validations — appelées commits dans Git — entre ces deux emplacements. Git permet de conserver un historique local, tandis que GitHub peut servir de dépôt distant partagé. [GITHUB-ABOUT-GIT-FR]

> Idée essentielle — Enregistrer un commit ne le publie pas automatiquement sur GitHub. Inversement, une modification visible sur GitHub n'apparaît pas automatiquement dans votre dossier local.

Dans les exemples, la branche principale est nommée main et le dépôt distant est nommé origin. Ces noms sont fréquents, mais votre projet peut utiliser d'autres noms. Vérifiez toujours votre branche courante avant d'exécuter une commande.

Notion de synchronisation

Une branche locale peut être associée à une branche distante. Cette association permet à Git de comparer les deux historiques et, selon la situation, d'indiquer si votre branche locale est :

- à jour : les deux historiques pointent vers la même validation ;
- en avance : vous possédez des commits locaux qui ne sont pas encore publiés ;
- en retard : le dépôt distant contient des commits absents de votre copie locale ;
- divergente : chaque côté contient des commits que l'autre n'a pas.

La commande suivante est le point de départ de toute synchronisation :

``bash git status ``

git status renseigne sur l'état de la copie de travail et peut aussi indiquer la relation entre la branche locale et sa branche de suivi distante. [GIT-STATUS]

Un résultat peut notamment signaler que votre branche est en avance sur origin/main, ou qu'elle est en retard. Lisez ce message avant de choisir une action.

Les deux directions

Direction	Action habituelle	Commande courante	Effet recherché
Local vers GitHub	Publier	git push	Envoyer des commits locaux vers le dépôt distant
GitHub vers local	Récupérer sans intégrer	git fetch	Mettre à jour les références distantes connues localement
GitHub vers local	Récupérer et intégrer	git pull	Récupérer puis intégrer les changements distants

Les commandes de synchronisation font partie des commandes Git documentées dans la référence officielle de Git. [GIT-OVERVIEW]

Publication de changements locaux

Publier signifie envoyer vers GitHub les commits déjà créés dans le dépôt local. Les fichiers modifiés mais non enregistrés dans un commit ne sont pas publiés par git push.

Vérifier avant de publier

Avant une publication, vérifiez l'état du projet :

``bash git status ``

Assurez-vous en particulier que :

1. vous êtes sur la bonne branche ;
2. les modifications que vous vouliez enregistrer ont bien été validées ;
3. vous n'avez pas de modification involontaire dans votre copie de travail ;
4. votre branche n'est pas en retard sur le dépôt distant.

Si le statut indique que votre branche est en avance sur sa branche distante, vous pouvez publier les commits locaux.

Envoyer les commits

Pour publier la branche locale main vers la branche main du dépôt distant origin :

``bash git push origin main ``

Lorsque la branche locale suit déjà une branche distante, la forme courte est souvent suffisante :

``bash git push ``

Git utilise des dépôts distants et des références de branche pour échanger l'historique entre dépôts. [GIT-OVERVIEW]

Après une publication réussie :

- les nouveaux commits deviennent visibles sur GitHub ;
- la branche distante est mise à jour ;
- git status devrait normalement indiquer que la branche locale est à jour avec sa branche distante.

Première publication d'une branche

Pour une branche créée uniquement en local, il peut être nécessaire d'établir son lien avec une branche distante lors du premier envoi :

``bash git push -u origin nom-de-ma-branche ``

L'option -u établit le suivi de la branche distante pour les commandes ultérieures. Ensuite, git push et git pull peuvent être utilisés sans préciser systématiquement le dépôt et la branche, tant que vous travaillez sur cette branche suivie.

> Prudence — N'utilisez pas de commande qui force une publication tant que vous ne comprenez pas précisément son effet. Une publication forcée peut réécrire l'historique distant et rendre le travail d'autres personnes plus difficile à retrouver.

Récupération de changements distants

D'autres personnes, ou vous-même depuis une autre machine, peuvent publier des commits sur GitHub. Avant de commencer une nouvelle tâche ou avant de publier votre propre travail, il est utile de vérifier si le dépôt distant a évolué.

Deux approches sont à distinguer :

- observer d'abord, avec git fetch ;
- récupérer et intégrer, avec git pull.

Observer les changements avec git fetch

La commande suivante contacte le dépôt distant et actualise localement les informations connues sur ses branches :

``bash git fetch origin ``

Après cette commande, vos fichiers de travail ne sont pas automatiquement modifiés. Vous pouvez donc examiner la situation avant de modifier votre branche locale.

Exécutez ensuite :

``bash git status ``

Si votre branche est en retard, cela signifie que la branche distante contient des commits que votre branche locale ne contient pas encore.

Cette méthode est particulièrement adaptée lorsque vous avez déjà des modifications locales, lorsque vous ne savez pas qui a publié récemment, ou lorsque vous souhaitez éviter une intégration automatique.

Récupérer et intégrer avec git pull

Lorsque votre copie de travail est propre et que vous souhaitez intégrer les changements de la branche distante suivie, utilisez :

``bash git pull ``

Vous pouvez aussi indiquer explicitement le dépôt et la branche :

``bash git pull origin main ``

git pull récupère les changements distants puis tente de les intégrer dans votre branche locale. La façon précise dont Git intègre les historiques dépend de la configuration et de la situation de la branche ; c'est pourquoi un débutant doit lire attentivement le résultat de la commande et contrôler l'état final avec git status. Les options et commandes Git sont décrites dans la documentation de référence officielle. [GIT-OVERVIEW] [GIT-CONFIG]

> Réflexe sûr — Si vous avez un doute, commencez par git fetch origin, puis consultez git status. Vous séparerez ainsi l'observation de l'intégration.

Mise à jour du dépôt local : une séquence simple

Pour un projet partagé, une séquence prudente consiste à récupérer les changements distants avant de produire ou de publier les vôtres.

Au début d’une session de travail

1. Placez-vous dans le dossier du dépôt.
2. Vérifiez l’état local :

``bash git status ``

1. Si vous n’avez pas de travail local non terminé, récupérez les mises à jour :

``bash git pull ``

1. Vérifiez de nouveau l’état :

``bash git status ``

Avant de publier votre travail

1. Vérifiez que vos modifications souhaitées ont été enregistrées dans des commits.
2. Consultez l’état :

``bash git status ``

1. Actualisez votre connaissance du dépôt distant :

``bash git fetch origin ``

1. Si des changements distants doivent être intégrés, faites-le avant de publier.
2. Vérifiez une nouvelle fois l’état du projet.
3. Publiez :

``bash git push ``

Cette séquence réduit le risque de découvrir trop tard que quelqu’un a déjà modifié la même branche.

Divergences entre historique local et distant

Une divergence apparaît lorsque votre branche locale et la branche distante possèdent chacune des commits que l’autre ne possède pas.

Par exemple :

- vous créez deux commits sur votre ordinateur sans les publier ;
- pendant ce temps, une autre personne publie un commit sur GitHub ;
- votre branche locale est alors en avance par rapport au distant pour vos commits, mais aussi en retard par rapport au commit publié par l’autre personne.

Dans cette situation, Git peut refuser une publication directe afin d’éviter d’écraser des changements distants. Ce refus n’est pas une panne : il vous signale qu’une mise à jour et une intégration sont nécessaires avant de publier.

Que faire face à un refus de publication ?

Suivez une démarche progressive :

1. Ne relancez pas immédiatement la publication avec une option de forçage.
2. Consultez l’état :

``bash git status ``

1. Récupérez les informations distantes :

``bash git fetch origin ``

1. Intégrez les changements nécessaires dans votre branche locale, avec la méthode retenue pour le projet.
2. Si Git signale un conflit, résolvez-le avant toute nouvelle publication.
3. Vérifiez l’état final.
4. Publiez seulement lorsque l’historique local intègre correctement le travail distant :

``bash git push ``

Comprendre les conflits pendant l’intégration

Un conflit peut survenir lorsque Git ne peut pas déterminer seul quelle version conserver dans une même zone d’un fichier. Git marque alors les fichiers concernés et attend une décision humaine.

La règle importante est la suivante : ne considérez jamais une synchronisation comme terminée tant que git status signale des fichiers non résolus ou une opération en cours. La commande git status est précisément conçue pour afficher l’état de la copie de travail et les prochaines actions pertinentes. [GIT-STATUS]

Précautions avant de publier ou d’intégrer des changements

La synchronisation est plus sûre lorsqu’elle repose sur quelques contrôles simples et répétés.

Avant un git push

- Vérifiez la branche courante avec git status.
- Vérifiez que les changements pertinents sont dans des commits.
- Récupérez les informations distantes avec git fetch origin si vous avez travaillé un certain temps sans synchroniser.
- Intégrez les changements distants nécessaires avant la publication.
- Lisez entièrement le message de Git en cas de refus.
- Évitez les options de forçage, surtout sur une branche partagée.

Avant un git pull

- Lancez git status.
- Si vous avez des modifications non validées, ne les mélangez pas précipitamment avec des changements distants.
- En cas de doute, utilisez d'abord git fetch origin afin d'observer la situation.
- Vérifiez que vous récupérez la bonne branche.
- Préparez-vous à résoudre un conflit si les changements portent sur des zones proches des mêmes fichiers.

Cas des modifications non enregistrées dans un commit

Une copie de travail contenant des modifications non validées demande une attention particulière. Selon les fichiers concernés, Git peut refuser l'intégration afin d'éviter d'écraser ce travail local. Ce comportement protecteur doit être pris au sérieux : examinez les fichiers, enregistrez les changements qui doivent l'être, puis reprenez la synchronisation.

Les commandes de préparation et d'enregistrement des changements sont décrites dans la documentation Git consacrée à l'ajout et à l'enregistrement des modifications. [GIT-ADD] [GIT-RECORDING-CHANGES]

Vérifications après synchronisation

Une commande terminée sans message d'erreur ne remplace pas une vérification. Après un git pull, un git push ou une résolution de conflit, contrôlez le résultat.

Contrôle minimal

``bash git status ``

Cherchez notamment les indications suivantes :

- aucune opération de fusion ou de résolution ne reste à terminer ;
- aucun fichier conflictuel n'est signalé ;
- la branche locale est à jour avec sa branche distante, lorsque c'est le résultat attendu ;
- les modifications encore présentes dans la copie de travail sont intentionnelles.

Contrôle sur GitHub après une publication

Après git push, ouvrez le dépôt sur GitHub et vérifiez :

- la branche affichée ;
- la présence du dernier commit attendu ;
- les fichiers modifiés ;
- l'absence de publication sur une mauvaise branche.

GitHub fournit une interface pour consulter le contenu et l'historique d'un dépôt, en complément du travail effectué avec Git en local. [GITHUB-ABOUT-GIT-FR]

Tableau de décision rapide

Situation observée	Action recommandée	Vérification finale
La branche est à jour	Commencer ou poursuivre le travail	git status
La branche est en avance	Publier avec git push après un dernier contrôle	Vérifier le commit sur GitHub puis git status
La branche est en retard	Récupérer et intégrer les changements distants	git status et examen des fichiers modifiés
La branche diverge	Récupérer, intégrer, résoudre les éventuels conflits, puis publier	Aucun conflit en cours ; branche synchronisée
git push est refusé	Ne pas forcer ; récupérer et comprendre la divergence	Relancer git status après intégration
Des changements locaux ne sont pas validés	Examiner et enregistrer le travail utile avant intégration	Copie de travail comprise et intentionnelle

> À retenir — La synchronisation n'est pas une simple routine mécanique. Avant d'envoyer ou de récupérer des changements, vérifiez la branche, l'état local et l'état distant. Après l'opération, vérifiez à nouveau. Cette habitude protège votre travail et celui des autres contributeurs.

CHAPITRE 9

Après avoir appris à synchroniser un dépôt local avec GitHub, l'étape suivante consiste à travailler à plusieurs sans modifier le même espace de travail de façon désordonnée. La collaboration repose sur une idée simple : une personne prépare une contribution dans une branche, la présente aux autres, échange sur son contenu, puis cette contribution est intégrée lorsqu'elle est prête.

Git enregistre l'historique des fichiers, tandis que GitHub fournit un espace partagé autour du dépôt et de ses échanges. GitHub emploie notamment les notions de dépôt (repository), branche (branch), demande de modification (pull request), problème ou tâche (issue) et relecture (review) dans son glossaire. [GITHUB-GLOSSARY-FR]

> Vocabulaire. Dans ce chapitre, une « demande de modification » désigne une pull request. Selon l'interface, votre équipe ou vos réglages linguistiques, vous pourrez aussi rencontrer l'expression anglaise pull request.

Principes de collaboration sur un dépôt

Un dépôt partagé n'est pas seulement un dossier de fichiers : c'est un historique commun et un lieu de coordination. Pour que cet historique reste compréhensible, chaque contribution doit répondre à trois questions :

1. Quel besoin est traité ?
2. Quels changements sont proposés ?
3. Comment les autres personnes peuvent-elles vérifier ces changements ?

La branche principale du dépôt représente généralement une version de référence du projet. Plutôt que de modifier directement cette branche, chaque contribution est préparée dans une branche dédiée. Une branche est une ligne de développement distincte au sein du dépôt. [GITHUB-GLOSSARY-FR]

Cette séparation apporte plusieurs avantages pratiques :

- le travail en cours reste isolé des changements déjà validés ;
- plusieurs sujets peuvent avancer en parallèle ;
- la modification peut être relue avant son intégration ;
- les discussions liées à une modification restent associées à celle-ci ;
- l'équipe peut décider explicitement quand une contribution devient partie du travail commun.

Une bonne contribution est volontairement limitée. Une branche qui mélange la correction d'une erreur, la modification de la mise en page et l'ajout d'une fonctionnalité difficile à relier à ces sujets sera plus complexe à comprendre et à relire. À l'inverse, une contribution centrée sur un objectif identifiable facilite le dialogue et le suivi.

Les rôles possibles dans une collaboration

Les rôles ci-dessous décrivent des responsabilités ; une même personne peut en exercer plusieurs, surtout dans une petite équipe.

Rôle	Responsabilité principale	Attitude attendue
Contributeur ou contributrice	Préparer et présenter une modification	Expliquer le besoin, limiter le périmètre, répondre aux retours
Relecteur ou relectrice	Examiner la proposition	Vérifier, questionner, signaler les points à clarifier
Responsable de l'intégration	Décider de l'intégration dans la branche de référence	S'assurer que la contribution est prête selon les règles de l'équipe
Responsable du dépôt	Organiser les accès, les règles et la structure du projet	Rendre le cadre de travail explicite et accessible

Ces rôles ne doivent pas devenir une barrière inutile. Leur utilité est de rendre visible qui prépare, qui vérifie et qui prend la décision finale.

Contribuer depuis une branche

Le déroulement conceptuel d'une contribution peut être présenté ainsi :

1. partir d'une version récente de la branche de référence ;
2. créer ou sélectionner une branche consacrée au sujet traité ;
3. réaliser les modifications nécessaires ;
4. enregistrer localement des validations (commits) cohérentes ;
5. envoyer la branche vers le dépôt distant ;
6. ouvrir une demande de modification vers la branche de référence ;

7. échanger pendant la relecture ;
8. compléter ou corriger la contribution si nécessaire ;
9. intégrer la contribution après validation ;
10. supprimer ou archiver la branche selon les règles du projet.

Un commit est une validation enregistrée dans l'historique Git. GitHub décrit Git comme un système de contrôle de version distribué qui permet de suivre les modifications et de coordonner le travail entre les personnes. [GITHUB-ABOUT-GIT-FR]

Nommer une branche

Adoptez un nom qui indique l'intention du travail, sans chercher à reproduire une phrase complète. Par exemple :

- ajout-page-contact
- correction-lien-accueil
- mise-a-jour-guide-installation

Le nom exact importe moins que la régularité. Si le projet relie ses tâches à des identifiants, l'équipe peut aussi décider de les inclure dans le nom de la branche.

Préparer des validations lisibles

Une demande de modification contient souvent plusieurs validations. Chacune gagne à être compréhensible isolément : elle devrait correspondre à une étape logique du travail et porter un message qui décrit clairement ce qui a été enregistré.

Avant de présenter une contribution, vérifiez notamment :

- que les fichiers attendus sont bien inclus ;
- qu'aucun fichier temporaire ou personnel n'a été ajouté par erreur ;
- que les modifications correspondent au sujet annoncé ;
- que l'état local est compris avant l'envoi.

La commande git status affiche l'état des fichiers dans le répertoire de travail et dans la zone d'indexation, ce qui aide à contrôler ce qui sera inclus dans une prochaine validation. [GIT-STATUS]

Demandes de modification : objectif et cycle de vie

Une demande de modification est un espace de proposition et de discussion autour de changements apportés par une branche. GitHub définit une pull request comme une proposition de modifications à intégrer dans un projet. [GITHUB-GLOSSARY-FR]

Elle ne signifie donc pas seulement « demander d'ajouter du code ». Elle crée un point de rencontre entre :

- le contexte : le besoin ou le problème traité ;
- les modifications : les différences entre la branche proposée et la branche cible ;
- la conversation : questions, commentaires et décisions ;

- la décision : poursuivre, demander des ajustements, intégrer ou abandonner la proposition.

Cycle de vie indicatif

``text Besoin identifié “ Branche de contribution “ Modifications et validations locales “ Branche envoyée sur GitHub “ Demande de modification ouverte “ Relecture et échanges “ Ajustements éventuels “ Décision d’intégration ou de clôture “ Branche de référence mise à jour ``

Une demande de modification peut évoluer après son ouverture. Les nouvelles validations ajoutées à la branche de contribution enrichissent la proposition. Cette possibilité doit servir à répondre aux retours et non à élargir sans limite le sujet initial.

Rédiger une demande compréhensible

Une demande claire réduit le temps nécessaire à la relecture. Utilisez un titre bref qui exprime l’action ou le résultat, puis complétez la description avec les éléments utiles.

Élément	Question à laquelle il répond
Titre	Que propose cette contribution ?
Contexte	Pourquoi ce changement est-il nécessaire ?
Contenu	Qu’est-ce qui a été modifié ?
Vérification	Comment le relecteur peut-il contrôler le résultat ?
Limites	Qu’est-ce qui n’est pas traité dans cette contribution ?
Liens	Quelle discussion ou quelle tâche est concernée ?

Un modèle de description, à adapter au dépôt, peut prendre cette forme :

```markdown
## Objectif

Décrire le besoin traité.

## Modifications réalisées

- Décrire les changements principaux.

## Vérification effectuée

- Indiquer ce qui a été contrôlé.
```

Hors périmètre

- Préciser ce qui reste à faire ou n'est pas inclus.

```

L’objectif n’est pas de remplir mécaniquement un formulaire. Il est de donner à la personne qui relit assez de contexte pour comprendre la proposition sans devoir reconstituer seule l’intention du contributeur.

### Relecture et commentaires

La relecture est une activité de collaboration, non un jugement sur la personne qui a proposé le changement. Elle vise à améliorer le résultat, à détecter les incompréhensions et à partager la connaissance du projet.

GitHub associe la notion de review à l’examen de modifications proposées dans une demande de modification. [GITHUB-GLOSSARY-FR]

#### Ce qu’un relecteur peut examiner

Selon la nature du dépôt, la relecture peut porter sur :

- l’adéquation entre le besoin annoncé et la modification ;
- la lisibilité des fichiers modifiés ;
- la cohérence avec les conventions existantes ;
- les conséquences possibles sur les autres fichiers ;
- les éléments à vérifier avant l’intégration ;
- l’absence de changements non liés au sujet.

La profondeur de la relecture doit être proportionnée au risque et à la taille de la contribution. Une correction typographique dans un document ne demande pas la même attention qu’une modification qui affecte le fonctionnement central d’un projet.

#### Formuler un commentaire utile

Un commentaire de qualité est précis, actionnable et respectueux. Évitez les formulations vagues telles que « ce n’est pas bien » ou « à refaire ». Préférez une structure simple :

1. localiser le point concerné ;
2. décrire l’observation ;
3. expliquer la conséquence ou la question ;
4. proposer une piste, lorsque cela aide.

Exemples de formulations :

- « Cette phrase peut être comprise de deux façons. Pourrait-on préciser le comportement attendu ? »
- « Je ne retrouve pas la vérification annoncée dans la description. Peux-tu indiquer comment tu l’as effectuée ? »
```

- « Cette modification semble traiter un second sujet. Serait-il possible de le séparer dans une autre contribution ? »
- « Proposition : utiliser le même vocabulaire que dans le fichier voisin pour conserver une terminologie homogène. »

Il est également utile de distinguer les niveaux de priorité convenus par l'équipe :

Indication possible	Signification proposée
Bloquant	Doit être résolu avant l'intégration
Important	Devrait être résolu ou explicitement discuté avant l'intégration
Suggestion	Amélioration facultative, sans bloquer la contribution
Question	Demande de clarification, sans présumer qu'un changement est requis

Ces libellés constituent une convention d'équipe. Ils doivent être définis explicitement si vous choisissez de les utiliser.

Répondre à une relecture

La personne qui contribue n'a pas à accepter automatiquement chaque suggestion. En revanche, elle doit traiter les retours : apporter une correction, demander une précision, expliquer une décision ou convenir d'un suivi ultérieur.

Une réponse utile relie clairement le commentaire à l'action réalisée. Par exemple :

- « Modifié dans la dernière validation : le terme est maintenant harmonisé. »
- « Je comprends la remarque. Je propose de traiter ce point dans une contribution distincte, car il dépasse l'objectif de celle-ci. »
- « Peux-tu préciser le cas d'usage qui motive cette demande ? Je souhaite vérifier que nous parlons du même comportement. »

Intégrer une contribution

Intégrer une contribution revient à incorporer les modifications d'une branche dans une autre, en général dans la branche de référence. GitHub utilise le terme merge pour désigner la combinaison de changements issus de branches différentes. [GITHUB-GLOSSARY-FR]

L'intégration ne devrait pas être une étape automatique déclenchée uniquement parce qu'une demande existe. Avant de décider, la personne responsable vérifie que les règles choisies par le projet sont respectées.

Une liste de contrôle d'équipe peut inclure les questions suivantes :

- le besoin est-il compris et la contribution reste-t-elle dans son périmètre ?

- les commentaires importants ont-ils reçu une réponse ?
- les vérifications prévues ont-elles été réalisées ?
- la branche cible est-elle la bonne ?
- la description permet-elle de comprendre la décision plus tard ?
- reste-t-il un travail à documenter dans une tâche distincte ?

Après l'intégration, la branche de contribution n'est en principe plus l'espace de travail actif pour ce sujet. La suppression de branches terminées peut améliorer la lisibilité du dépôt, à condition que cette pratique corresponde aux règles retenues par l'équipe.

> Attention. Une intégration peut créer un conflit lorsque des changements incompatibles concernent une même zone de travail. La manière de résoudre ces conflits dépend de la situation et des règles du dépôt. Ne validez pas une résolution que vous ne comprenez pas : demandez une relecture complémentaire si nécessaire.

Suivi des discussions et des tâches

Les discussions qui concernent un changement précis ont intérêt à rester dans la demande de modification correspondante. Les travaux à planifier, les défauts à corriger et les idées à examiner peuvent être suivis séparément dans des tâches ou problèmes (issues). GitHub décrit une issue comme un moyen de suivre des idées, des améliorations, des tâches ou des défauts dans un dépôt. [GITHUB-GLOSSARY-FR]

Cette séparation évite deux difficultés fréquentes :

- une demande de modification devient une longue liste de sujets non liés ;
- une décision importante se perd dans une conversation externe au dépôt.

Choisir le bon espace d'échange

Situation	Espace à privilégier
Question directement liée à une ligne ou à un fichier modifié	Commentaire de relecture
Clarification sur l'objectif ou le périmètre d'une proposition	Discussion de la demande de modification
Travail futur, erreur à reproduire ou amélioration à planifier	Tâche ou problème dédié
Décision durable concernant les règles du projet	Document de règles du dépôt ou espace explicitement choisi par l'équipe
Échange urgent ou sensible	Canal défini par l'équipe, puis synthèse utile dans le dépôt si nécessaire

Quand une discussion mène à une décision, formulez-la explicitement. Une phrase telle que « décision : cette contribution couvre uniquement la version française ; les autres traductions seront suivies séparément » permet de rendre le résultat de l'échange retrouvable.

Règles de communication et de coordination à définir

GitHub fournit des mécanismes de dépôt, de branches, de demandes de modification, de commentaires et de tâches. Les règles concrètes de collaboration, elles, appartiennent à chaque projet. Elles doivent être adaptées à la taille de l'équipe, au niveau d'expérience des personnes, au type de contenu et au niveau de contrôle nécessaire.

Pour débuter, formalisez des règles simples, courtes et visibles dans le dépôt. Le tableau suivant est un point de départ à décider collectivement, et non une règle imposée par GitHub.

Sujet	Décision à formaliser
Branche de référence	Quelle branche représente la version partagée de référence ?
Travail direct	Dans quels cas, s'il y en a, une modification directe est-elle acceptable ?
Branches	Quelle convention de nommage est utilisée ?
Taille des contributions	Quel niveau de découpage rend une demande facile à relire ?
Description	Quelles informations minimales doivent figurer dans chaque demande ?
Relecture	Combien de relectures sont souhaitées et pour quels types de changement ?
Vérifications	Quels contrôles doivent être indiqués avant l'intégration ?
Décision	Qui peut intégrer, fermer ou demander des ajustements ?
Délais	Comment signaler une contribution qui nécessite une attention rapide ?
Communication	Quels canaux utilisent l'équipe et où consigne-t-elle les décisions ?
Respect	Quelles règles encadrent le ton des commentaires et la gestion des désaccords ?

Une règle essentielle : commenter le travail, pas la personne

La collaboration devient plus sûre et plus efficace lorsque les remarques portent sur un fichier, une décision, un effet observé ou un besoin utilisateur. Préférez : « cette formulation prête à confusion » à « tu as mal formulé cette phrase ».

De même, une demande d'explication n'est pas nécessairement une critique. Elle peut révéler qu'un contexte important manque dans la description ou dans le projet. Répondre avec précision améliore à la fois la contribution présente et la compréhension collective.

Retenir l'essentiel

Une collaboration saine sur GitHub suit une chaîne lisible :

- un besoin est identifié ;
- une branche dédiée porte une contribution limitée ;
- une demande de modification présente le contexte et les changements ;
- les personnes concernées relisent et commentent avec précision ;
- les retours sont traités ou explicitement discutés ;
- une personne autorisée décide de l'intégration ;
- les tâches restantes sont suivies séparément.

La technique ne suffit pas à elle seule. La qualité de la collaboration dépend aussi de conventions explicites, de descriptions claires, de commentaires respectueux et de décisions traçables. Ces pratiques rendent le dépôt plus accueillant pour les débutants comme pour les personnes qui rejoindront le projet plus tard.

CHAPITRE 10

Un dépôt partagé ne sert pas uniquement à stocker des fichiers. Il doit permettre à une personne qui arrive pour la première fois de comprendre rapidement ce que contient le projet, à quoi il sert, comment le démarrer et quelles précautions respecter avant de le modifier ou de le diffuser.

L'objectif n'est pas de produire une documentation parfaite dès le premier jour. Il s'agit de donner des repères utiles, de ranger les informations à un endroit prévisible et de distinguer clairement ce qui peut être public de ce qui doit rester protégé.

> Principe directeur : une personne qui découvre le dépôt devrait pouvoir répondre en quelques minutes à quatre questions : Quel est le but du projet ? Comment l'utiliser ? Où se trouvent les éléments importants ? Comment contribuer sans prendre de risque ?

Page d'accueil d'un projet

La page d'accueil d'un dépôt doit jouer le rôle d'une porte d'entrée. Elle présente le projet avant même que le lecteur explore les dossiers ou ouvre les fichiers de code.

Un fichier README.md est couramment utilisé comme document d'accueil. L'extension .md correspond au format Markdown, un format de texte léger permettant notamment d'écrire des titres, des listes, des liens et des blocs de code.

Pour un projet débutant, une page d'accueil peut rester courte tout en étant très utile. Elle peut comporter les rubriques suivantes :

Rubrique	Question à laquelle elle répond	Exemple de contenu
Titre	Quel est le nom du projet ?	Calculatrice Python
Description	À quoi sert-il ?	« Une petite application de calcul en ligne de commande. »
État du projet	Est-il terminé, en cours ou expérimental ?	« Projet d'apprentissage en cours de développement. »
Démarrage rapide	Comment l'essayer ?	Les étapes essentielles pour ouvrir ou exécuter le projet
Organisation	Où sont les éléments principaux ?	Une courte présentation des dossiers
Contribution	Comment proposer une amélioration ?	Un lien vers les règles de contribution
Licence	Dans quel cadre le contenu est-il partagé ?	Une indication vers le fichier de licence, après vérification

Voici un exemple de structure de README.md :

```md
# Calculatrice Python

Une application simple permettant d'effectuer des calculs dans le terminal.

## Démarrage

1. Téléchargez ou clonez le projet.
2. Ouvrez un terminal dans le dossier du projet.
3. Lancez le programme selon les instructions du projet.

## Organisation

• src/ : fichiers du programme
• docs/ : documentation complémentaire
• tests/ : fichiers de test

## Contribution

Consultez CONTRIBUTING.md avant de proposer une modification.
```

Licence

Consultez le fichier LICENSE pour connaître les conditions applicables. ```

Cet exemple est un modèle rédactionnel : les commandes exactes de démarrage et les noms de dossiers doivent correspondre au projet réel. Il vaut mieux supprimer une instruction incertaine que publier une procédure qui ne fonctionne pas.

Documentation de démarrage

La documentation de démarrage vise une personne qui ne connaît pas encore le projet. Elle doit éviter les implicites tels que « installez les dépendances habituelles » ou « lancez le script normalement ». Ce qui paraît évident à l'auteur ne l'est pas forcément à un nouvel arrivant.

Une procédure de démarrage claire décrit, dans l'ordre :

1. Les prérequis : logiciels, comptes, outils ou accès nécessaires.
2. L'obtention du projet : comment récupérer une copie de travail.
3. La préparation : éventuelle configuration locale à réaliser.
4. Le lancement : action précise permettant de vérifier que le projet fonctionne.
5. Le résultat attendu : ce que la personne doit voir ou obtenir.
6. Le point de dépannage : où chercher si l'étape échoue.

Le niveau de détail doit être adapté au public. Pour un projet destiné à des débutants, il est préférable d'écrire les commandes sur des lignes séparées et d'expliquer brièvement leur objectif.

```md

## Vérifier l'installation

Exécutez la commande suivante :

``bash python main.py ``

Si tout est correctement configuré, le programme affiche un message d'accueil. ```

Lorsque le projet comporte plusieurs façons de l'utiliser, commencez par le chemin le plus simple. Les variantes avancées peuvent être placées dans une section distincte, par exemple ## Utilisation avancée ou dans un document situé dans le dossier docs/.

> Conseil de lisibilité : testez les instructions en suivant le document dans l'ordre, comme si vous découvriez le projet. Une étape manquante apparaît souvent immédiatement lors de cette relecture.
```

Organiser les fichiers du projet

Une arborescence cohérente aide à comprendre le dépôt sans lire tous les fichiers. L'idée n'est pas d'imposer une structure universelle : un site web, un programme en Python et un document de recherche n'ont pas les mêmes besoins. En revanche, les fichiers remplissant un rôle semblable gagnent à être regroupés.

Une organisation simple peut ressembler à ceci :

``text mon-projet/ README.md CONTRIBUTING.md LICENSE src/ docs/ tests/ exemples/ ``

Élément	Rôle possible
README.md	Présenter le projet et indiquer comment commencer.
src/	Regrouper les fichiers principaux du projet.
docs/	Accueillir une documentation plus détaillée.
tests/	Regrouper les vérifications ou tests du projet.
exemples/	Fournir des exemples réutilisables et non sensibles.
CONTRIBUTING.md	Expliquer comment participer au projet.
LICENSE	Contenir le texte de licence retenu, après validation de son adéquation.

Quelques règles simples rendent cette structure plus facile à parcourir :

- choisir des noms de fichiers et de dossiers explicites ;
- éviter de mélanger documentation, fichiers temporaires et fichiers principaux dans le même emplacement ;
- conserver une convention de nommage régulière ;
- déplacer les documents longs dans un dossier dédié plutôt que de rendre le fichier d'accueil difficile à lire ;
- signaler dans le document d'accueil l'emplacement des ressources importantes.

Il n'est pas nécessaire de créer tous ces dossiers dès le début. Un petit projet peut ne contenir qu'un fichier principal et un README.md. La structure peut évoluer lorsque le projet s'agrandit.

Informations de contribution

Un projet partageable indique comment les autres personnes peuvent participer. Même pour un petit dépôt, quelques règles réduisent les malentendus et évitent que les propositions arrivent sous une forme difficile à examiner.

Un fichier CONTRIBUTING.md peut préciser :

- le type de contributions recherchées : corrections, documentation, exemples ou nouvelles fonctionnalités ;
- la manière de signaler un problème ;
- les étapes à effectuer avant de proposer une modification ;
- les conventions de nommage ou de formatage utilisées par le projet ;
- le niveau de détail attendu dans une proposition ;
- les comportements attendus dans les échanges.

Un modèle minimal peut être le suivant :

```md
# Contribuer au projet

## Avant de commencer

- Lisez le README du projet.
- Vérifiez qu'une demande similaire n'a pas déjà été signalée.
- Décrivez clairement le problème ou l'amélioration proposée.

## Proposer une modification

- Travaillez dans une branche dédiée.
- Expliquez ce qui a été modifié et pourquoi.
- Indiquez comment vous avez vérifié votre modification.

## Communication

Restez clair, respectueux et factuel dans les échanges.
```

Ces règles doivent être réalistes. Une procédure très longue, appliquée de façon irrégulière, crée davantage de confusion qu'une consigne courte et suivie. Les pratiques de collaboration vues précédemment — branches dédiées, revue des changements et descriptions explicites — peuvent être rappelées sans recopier toute leur explication.

Informations importantes à rendre visibles

Certaines informations méritent d'être placées en évidence dans le document d'accueil ou dans un fichier clairement nommé. Leur présence évite aux utilisateurs de devoir les deviner.

Information	Emplacement suggéré	Formulation utile
Projet expérimental	Début du README.md	« Ce projet est un prototype d'apprentissage. »
Limites connues	Section dédiée	« Cette version ne traite pas encore… »
Instructions de sécurité	Début de la procédure concernée	« N'utilisez pas de données réelles dans cet exemple. »
Canal de contribution	CONTRIBUTING.md	« Décrivez le contexte et les étapes de reproduction. »
Documentation détaillée	README.md et docs/	« Consultez docs/installation.md pour le guide complet. »

Le lecteur ne doit pas avoir à supposer qu'un projet est prêt pour un usage réel, compatible avec tous les environnements ou adapté à des données sensibles. Si une limite est connue, la signaler clairement est une forme de qualité documentaire.

Informations de licence : un périmètre à clarifier

La licence est un sujet à traiter avec prudence. Un fichier nommé LICENSE ne doit pas être ajouté simplement parce qu'il est fréquent dans d'autres dépôts. Le choix d'un texte de licence peut dépendre notamment de la nature du contenu, des droits détenus par les auteurs, des éléments tiers inclus dans le projet, du cadre professionnel ou scolaire et des règles de l'organisation concernée.

Avant de publier une information de licence, il est utile de vérifier :

- qui est propriétaire du code, des documents et des autres contenus du dépôt ;
- si le dépôt contient du contenu provenant de tiers ;
- si des conditions déjà applicables limitent la diffusion ou la réutilisation ;
- qui est autorisé à choisir ou valider les conditions de partage ;
- si un avis juridique ou organisationnel est nécessaire.

> Prudence : ce chapitre ne fournit pas de conseil juridique et ne recommande aucune licence particulière. En cas de doute, ne publiez pas d'affirmation sur les droits d'utilisation avant validation par la personne ou l'entité compétente.

Si le périmètre n'est pas encore clarifié, il est préférable de le signaler comme un point à traiter en interne plutôt que de copier un fichier de licence depuis un autre projet sans vérifier qu'il convient.

Gérer les données sensibles et les secrets

Un dépôt visible par d'autres personnes ne doit pas contenir d'informations qui n'ont pas été vérifiées comme partageables. Le risque ne concerne pas uniquement les mots de passe : de nombreuses données peuvent être inadaptées à la publication selon le contexte.

Avant toute diffusion, examinez notamment la présence éventuelle de :

- mots de passe, jetons d'accès ou clés privées ;
- identifiants de connexion ;
- informations personnelles ;
- adresses électroniques non destinées à être publiées ;
- fichiers de configuration contenant des valeurs réelles ;
- données clients, exports de bases de données ou journaux d'activité ;
- documents internes, contrats, captures d'écran ou informations commerciales ;
- clés d'interface de programmation applicative, c'est-à-dire des clés utilisées par une Application Programming Interface (API).

Une séparation explicite est utile :

``text À partager dans le dépôt À vérifier ou à conserver hors du dépôt ---------- ---------- Code source du projet Mots de passe et jetons d'accès Documentation générale Données personnelles Exemples fictifs Données réelles de clients ou d'utilisateurs Fichiers de configuration d'exemple Configuration locale contenant des secrets Instructions d'installation Documents internes ou confidentiels ``

Pour documenter une configuration sans exposer de valeur réelle, un projet peut fournir un exemple fictif et clairement identifié. Par exemple :

``text SERVICE_URL=https://example.invalid SERVICE_TOKEN=REMPLACER_PAR_VOTRE_VALEUR ``

Cet exemple ne doit contenir ni adresse réellement utilisée, ni identifiant réel, ni secret. Le document de démarrage peut ensuite expliquer que chaque personne renseigne ses propres valeurs dans son environnement local, selon les règles applicables au projet.

Ne supposez pas qu'une vérification automatique, un réglage d'accès ou une suppression ultérieure suffira à réparer la publication d'une information sensible. La mesure la plus fiable présentée ici est préventive : relire les fichiers qui vont être partagés et demander une validation lorsque le contenu ou son niveau de confidentialité est incertain.

Liste de vérification avant publication

Avant de rendre un dépôt partageable, passez en revue les éléments suivants.

Compréhension du projet

- [ ] Le titre et la description expliquent clairement l'objectif du projet.

- [ ] Le document d’accueil indique l’état du projet et ses principales limites.
- [ ] Une personne débutante peut identifier les fichiers et dossiers importants.
- [ ] Les étapes de démarrage sont complètes, dans le bon ordre et testées.

Organisation et contribution

- [ ] Les fichiers sont rangés selon leur rôle.
- [ ] Les exemples fournis sont compréhensibles et ne contiennent pas de données réelles.
- [ ] Les règles de contribution sont accessibles et adaptées à la taille du projet.
- [ ] Les informations importantes sont placées dans un fichier ou une section facile à trouver.

Droits et confidentialité

- [ ] Le statut de la licence a été vérifié auprès de la personne ou de l’entité compétente.
- [ ] Les contenus provenant de tiers ont été identifiés et examinés.
- [ ] Aucun mot de passe, jeton, clé privée ou identifiant réel n’est présent.
- [ ] Aucune donnée personnelle, confidentielle ou interne n’est publiée sans autorisation explicite.
- [ ] Les fichiers de configuration partagés ne contiennent que des valeurs d’exemple non sensibles.

Retenir l’essentiel

Un dépôt lisible et partageable repose sur des choix simples : une présentation courte, une procédure de démarrage vérifiable, des fichiers rangés de manière cohérente, des règles de contribution visibles et une vérification attentive de ce qui est publié.

La documentation n’est pas un élément secondaire ajouté à la fin du projet. Elle fait partie de l’expérience de collaboration. Elle aide les utilisateurs à démarrer, les contributeurs à participer et les responsables du projet à communiquer ses limites avec clarté.

Surtout, la lisibilité ne doit jamais se faire au détriment de la confidentialité ou des droits applicables. En cas de doute sur un fichier, une donnée ou une information de licence, interrompez la publication et obtenez une validation adaptée au contexte.

CHAPITRE 11

Ce chapitre clôt le parcours d'initiation. L'objectif n'est pas de mémoriser toutes les commandes de Git, mais d'adopter un enchaînement de travail fiable : observer l'état du projet, préparer précisément ce qui doit être enregistré, créer un historique compréhensible, puis synchroniser ce travail avec GitHub.

Git est un système de gestion de versions distribué : un dépôt local contient l'historique du projet, tandis que GitHub fournit un environnement en ligne pour héberger des dépôts et faciliter la collaboration. [GITHUB-ABOUT-GIT-FR]

Récapitulatif des notions fondamentales

Les notions suivantes constituent le socle à conserver avant d'aborder des workflows plus complexes.

Notion	Rôle dans le travail quotidien	Réflexe utile
Dépôt (repository)	Dossier de projet suivi par Git, avec son historique	Vérifier que l'on travaille dans le bon dépôt
Répertoire de travail (working tree)	Fichiers présents et modifiables sur l'ordinateur	Examiner les changements avant de les enregistrer
Zone d'index (staging area)	Sélection des modifications destinées au prochain commit	Ajouter uniquement les fichiers ou portions de fichiers voulus
Commit	Instantané enregistré dans l'historique local	Décrire clairement l'intention de la modification
Branche (branch)	Ligne de développement distincte	Isoler une fonctionnalité, une correction ou un essai
Dépôt distant (remote repository)	Copie du dépôt accessible par le réseau	Synchroniser avant et après un travail partagé
origin	Nom couramment attribué au dépôt distant principal	Vérifier sa destination avant un envoi
push	Envoi de commits locaux vers un dépôt distant	Partager les commits terminés
pull	Récupération et intégration de changements distants	Se mettre à jour avant de poursuivre une tâche collaborative
Pull request	Proposition de modification sur GitHub, examinable avant intégration	Expliquer le but, le périmètre et les vérifications réalisées

Pour observer l'état du répertoire de travail, de la zone d'index et de la branche courante, git status est la commande de diagnostic de base. La documentation officielle indique qu'elle affiche l'état du répertoire de travail et de la zone d'index. [GIT-STATUS]

Pour préparer le contenu d'un commit, Git utilise git add, qui ajoute le contenu de fichiers à l'index. [GIT-ADD] L'enregistrement des changements dans l'historique est ensuite traité comme une étape distincte. [GIT-RECORDING-CHANGES]

> Fil conducteur à retenir : modifier ' vérifier ' sélectionner ' enregistrer ' synchroniser ' faire relire lorsque le contexte l'exige.

Habitudes de travail à retenir

Les pratiques ci-dessous sont des habitudes pédagogiques adaptées à un usage débutant. Elles visent surtout à réduire les erreurs, à rendre l'historique lisible et à faciliter la collaboration.

Commencer par regarder avant d'agir

Avant d'ajouter, de valider ou de synchroniser des fichiers, utilisez :

``bash git status ``

Cette étape permet de répondre à des questions simples mais essentielles : quels fichiers ont changé ? Sont-ils déjà préparés ? Quel est le contexte de la branche actuelle ? [GIT-STATUS]

Ne considérez pas cette commande comme une formalité : elle constitue un point de contrôle avant toute action qui modifie l'historique ou partage du travail.

Faire des commits petits et cohérents

Un commit utile porte sur une intention identifiable. Par exemple :

- ajouter une page de présentation ;
- corriger une erreur d'affichage ;
- mettre à jour les instructions d'installation ;
- renommer un fichier devenu ambigu.

Évitez de réunir dans un même commit une correction, une nouvelle fonctionnalité et une réorganisation générale des fichiers. Un historique découpé par intentions est plus facile à lire, à examiner et à corriger.

Le message de commit doit permettre à une autre personne — ou à vous-même plus tard — de comprendre ce qui a été fait sans ouvrir immédiatement tous les fichiers. Préférez une formulation directe, par exemple :

``text Add installation instructions Fix broken navigation link Update project description ``

Préparer explicitement ce qui sera enregistré

git add ne signifie pas simplement « sauvegarder ». Cette commande prépare le contenu du prochain commit dans l'index. [GIT-ADD] Prenez donc l'habitude de vérifier les fichiers ajoutés avant de créer le commit.

Une séquence simple est :

``bash git status git add nom-du-fichier git status git commit -m "Describe the change" ``

L'intérêt de cette seconde vérification est de distinguer ce qui reste uniquement dans le répertoire de travail de ce qui fera réellement partie du commit suivant.

Synchroniser à des moments choisis

Dans un projet partagé, évitez de travailler longtemps sans regarder les évolutions du dépôt distant. Avant de commencer une nouvelle portion de travail, récupérez les changements attendus par votre méthode d'équipe. Avant de demander une relecture ou de terminer une session, envoyez vos commits lorsque cela est approprié.

Cette routine réduit le risque de découvrir tardivement que le projet a évolué en parallèle. Elle ne dispense toutefois pas de lire les changements reçus ni de résoudre soigneusement les conflits éventuels.

Utiliser les branches comme espaces de travail

Créez une branche lorsqu'une modification mérite d'être isolée du travail principal : ajout de fonctionnalité, correction ciblée, expérimentation ou amélioration de documentation. Une branche donne un cadre clair à la modification et facilite une pull request lisible.

Avant de créer une pull request, vérifiez notamment :

- que la branche contient le périmètre prévu ;
- que les fichiers inutiles n'ont pas été ajoutés ;
- que les commits ont des messages compréhensibles ;
- que la description explique le changement et, si nécessaire, la manière de le vérifier.

Protéger le compte utilisé pour collaborer

GitHub propose l'authentification à deux facteurs (two-factor authentication, 2FA) comme mécanisme de protection du compte. La documentation officielle explique la configuration de cette fonctionnalité ainsi que la mise en place de méthodes de récupération. [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY]

Conservez les méthodes de récupération dans un emplacement sûr et distinct de votre session habituelle. En cas de doute sur une demande d'accès, un jeton ou une autorisation, interrompez l'action et vérifiez ce qui est demandé avant de poursuivre.

Erreurs fréquentes et signaux d'alerte

Les difficultés rencontrées par les débutants ne révèlent pas un manque d'aptitude : elles signalent généralement qu'une vérification manque dans le processus. Le tableau suivant associe des situations courantes à une réaction prudente.

Situation ou signal	Risque courant	Réaction immédiate
git status affiche des fichiers inattendus	Ajout involontaire de fichiers temporaires, personnels ou non pertinents	Ne pas faire de commit ; identifier l'origine des fichiers et ne préparer que le contenu utile
Un commit contient trop de sujets	Historique difficile à relire ou à annuler partiellement	S'arrêter avant le prochain envoi ; revoir le périmètre des modifications
Une commande est copiée sans être comprise	Modification involontaire de fichiers, de branches ou d'historique	Lire la commande, vérifier le dépôt et demander une explication si nécessaire
Un refus survient lors d'un push	Le dépôt distant contient peut-être des changements absents localement	Ne pas forcer l'envoi ; récupérer le contexte distant et comprendre l'écart
Un conflit apparaît	Deux évolutions concernent une zone incompatible	Lire les fichiers concernés, choisir ou recomposer le résultat, puis vérifier le projet
Une pull request devient très volumineuse	Relecture difficile, problèmes mélangés, risque d'oublis	Réduire le périmètre ou découper le travail en étapes cohérentes
Un secret ou une information sensible semble présent dans un fichier	Exposition potentielle dans l'historique ou sur le dépôt distant	Ne pas publier ; alerter immédiatement la personne ou l'équipe responsable selon le contexte
Le compte GitHub est inaccessible	Blocage de la collaboration ou risque de perte d'accès	Utiliser les méthodes de récupération préalablement configurées [GITHUB-2FA-RECOVERY]

Confondre Git et GitHub

Git et GitHub sont complémentaires, mais ne sont pas la même chose. Git gère l'historique et les opérations de versionnement dans le dépôt local ; GitHub est une plateforme qui s'appuie sur Git pour héberger et collaborer autour de dépôts. [GITHUB-ABOUT-GIT-FR]

Cette distinction aide à diagnostiquer un problème :

- un commit peut exister localement sans avoir été envoyé sur GitHub ;
- un dépôt GitHub peut contenir des changements qui ne sont pas encore présents sur l'ordinateur ;
- une difficulté d'authentification GitHub n'implique pas nécessairement que Git local est mal configuré.

Croire que chaque fichier modifié sera automatiquement inclus

Git distingue les modifications observées dans le répertoire de travail et les contenus ajoutés à l'index. git status permet de voir cet état, tandis que git add prépare le contenu à enregistrer. [GIT-STATUS] [GIT-ADD]

Avant tout commit, vérifiez donc ce qui est réellement préparé. Cette habitude évite autant l'oubli d'un fichier important que l'ajout d'un fichier non souhaité.

Réagir à l'erreur en multipliant les commandes

Face à un message incompris, la meilleure première réponse est souvent d'arrêter la chaîne d'actions. Revenir à git status, relire le dernier message affiché et identifier la branche ainsi que le dépôt concernés donne une base plus sûre que des essais successifs.

> Principe de prudence : si l'effet d'une commande sur l'historique ou sur le dépôt distant n'est pas clair, ne l'exécutez pas immédiatement.

Limites du périmètre du cours

Ce parcours a établi une base opérationnelle pour travailler avec un dépôt local, le synchroniser avec GitHub, employer des branches et participer à une collaboration simple. Il ne prétend pas couvrir tous les usages de Git ni tous les réglages d'une organisation.

Les sujets suivants dépassent le niveau d'initiation traité ici :

Sujet	Pourquoi il dépasse ce parcours
Réécriture avancée de l'historique	Ces opérations peuvent modifier ou remplacer des commits existants et demandent une compréhension solide des conséquences en travail partagé
Stratégies de branches d'équipe	Le choix entre branches de fonctionnalités, branches de publication ou autres conventions dépend du projet et de l'organisation
Résolution complexe de conflits	Les conflits simples peuvent être abordés progressivement ; les conflits étendus exigent de comprendre le code ou les documents concernés
Gestion des sous-modules et dépôts imbriqués	Ces mécanismes ajoutent des relations entre dépôts qui ne sont pas nécessaires à une première prise en main
Automatisation et intégration continue	Les tests, validations et déploiements automatisés nécessitent des outils, des droits et une configuration propres au projet
Administration d'organisation GitHub	Les rôles, permissions, politiques et paramètres d'organisation relèvent d'un cadre collectif ou administratif
Gestion professionnelle des secrets	La protection, la rotation et l'usage de données sensibles requièrent des procédures adaptées au contexte de travail
Commandes Git avancées	Certaines commandes puissantes sont utiles, mais elles ne doivent pas être employées par imitation sans comprendre leur effet sur l'historique

Ne déduisez pas de cette liste que ces sujets sont interdits aux débutants. Ils constituent plutôt la suite logique de l'apprentissage, à aborder un par un, dans un dépôt d'essai ou avec l'accompagnement de personnes expérimentées.

Sujets à approfondir

Après avoir acquis une routine fiable, choisissez une direction d'approfondissement selon votre besoin.

Pour devenir plus autonome avec Git

1. Approfondissez la configuration personnelle de Git : identité, préférences et paramètres locaux ou globaux. Git fournit git config pour obtenir et définir des variables de configuration ; le livre Pro Git présente également la configuration initiale. [GIT-CONFIG] [GIT-FIRST-CONFIG]
2. Relisez la documentation des commandes déjà utilisées, en particulier git status et git add, afin de mieux comprendre les options proposées. [GIT-STATUS] [GIT-ADD]

3. Créez un dépôt de test avec git init pour expérimenter sans risque pour un projet réel. La documentation officielle décrit git init comme la commande qui crée un dépôt Git vide ou réinitialise un dépôt existant. [GIT-INIT]

4. Entraînez-vous à lire un historique et à identifier la branche sur laquelle vous travaillez avant chaque opération importante.

Pour mieux collaborer avec GitHub

1. Explorez le vocabulaire officiel de GitHub afin de consolider les notions de dépôt, branche, fork, pull request et autres termes rencontrés dans l'interface. [GITHUB-GLOSSARY-FR]

2. Améliorez progressivement la qualité de vos pull requests : objectif explicite, périmètre limité, contexte suffisant et réponse attentive aux commentaires.

3. Renseignez-vous sur les conventions de votre équipe : nommage des branches, format des messages de commit, règles de revue et procédure de fusion.

4. Configurez et maintenez les mécanismes de protection du compte, y compris l'authentification à deux facteurs et les options de récupération. [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY]

Parcours visuel des prochaines étapes

``text Fondations acquises “ Routine locale fiable (status ’ add ’ commit) “ Synchronisation réfléchie (travail local ” dépôt distant) “ Collaboration lisible (branches ’ pull requests ’ revue) “ Approfondissement ciblé (conflits, historique, automatisation, sécurité) ``

L'ordre compte : une automatisation ou une stratégie complexe est plus utile lorsque les gestes fondamentaux sont déjà réguliers et compréhensibles.

Glossaire final

Terme	Définition courte
Authentification à deux facteurs (2FA)	Mécanisme de connexion qui ajoute une seconde étape de vérification à l'accès au compte GitHub. [GITHUB-2FA-CONFIG]
Branche	Ligne de développement permettant d'isoler un ensemble de modifications.
Clone	Copie locale d'un dépôt existant, généralement obtenu depuis un dépôt distant.
Commit	Enregistrement d'un ensemble préparé de changements dans l'historique Git.
Conflit	Situation dans laquelle Git ne peut pas déterminer seul comment combiner des modifications concurrentes.
Dépôt	Projet suivi par Git, avec ses fichiers et son historique.
Dépôt distant	Dépôt accessible par le réseau et associé à un dépôt local.
Fork	Copie d'un dépôt dans un espace GitHub distinct, souvent utilisée pour contribuer à un projet que l'on ne gère pas directement. Voir le glossaire GitHub pour la terminologie complète. [GITHUB-GLOSSARY-FR]
Git	Système de gestion de versions distribué. [GITHUB-ABOUT-GIT-FR]
GitHub	Plateforme permettant notamment d'héberger des dépôts Git et de collaborer autour de ceux-ci. [GITHUB-ABOUT-GIT-FR]
Historique	Suite des commits qui retrace l'évolution d'un dépôt.
Index / zone d'index	Zone dans laquelle Git prépare le contenu du prochain commit. [GIT-ADD]
origin	Nom fréquemment utilisé pour désigner le dépôt distant principal.
Pull request	Proposition de modifications sur GitHub, destinée à être examinée puis éventuellement intégrée.
pull	Opération de récupération et d'intégration de changements provenant d'un dépôt distant.
push	Opération d'envoi de commits locaux vers un dépôt distant.
Répertoire de travail	Ensemble des fichiers actuellement présents dans le dossier du projet et manipulés localement.
remote	Référence locale vers un dépôt accessible à distance.
staging area	Expression anglaise désignant la zone d'index.
git status	Commande qui affiche l'état du répertoire de travail et de la zone d'index. [GIT-STATUS]

Ressources officielles à sélectionner et vérifier

Les ressources suivantes proviennent de la documentation officielle de Git ou de GitHub. Elles constituent des points de départ adaptés pour vérifier une commande, clarifier un terme ou poursuivre l'apprentissage.

Besoin	Ressource officielle
Comprendre Git et sa relation avec GitHub	« À propos de Git » dans la documentation GitHub. [GITHUB-ABOUT-GIT-FR]
Retrouver un terme GitHub	Glossaire GitHub en français. [GITHUB-GLOSSARY-FR]
Vérifier l'état du projet avant une action	Documentation de git status. [GIT-STATUS]
Comprendre la préparation des changements	Documentation de git add. [GIT-ADD]
Revoir l'enregistrement des changements	Chapitre « Recording Changes to the Repository » du livre Pro Git. [GIT-RECORDING-CHANGES]
Consulter la référence générale de Git	Documentation de la commande git. [GIT-OVERVIEW]
Configurer Git	Documentation de git config et chapitre de configuration initiale de Pro Git. [GIT-CONFIG] [GIT-FIRST-CONFIG]
Créer un dépôt d'entraînement	Documentation de git init. [GIT-INIT]
Renforcer l'accès au compte GitHub	Documentation de configuration de la 2FA et des méthodes de récupération. [GITHUB-2FA-CONFIG] [GITHUB-2FA-RECOVERY]

Pour chaque nouvelle commande rencontrée, privilégiez la documentation officielle correspondante avant de suivre une instruction trouvée isolément. Vérifiez le nom de la commande, son effet attendu, le dépôt concerné et les conséquences possibles sur le travail partagé.

> Conclusion : la compétence essentielle n'est pas d'exécuter vite des commandes Git. C'est de savoir quel état se trouve le projet, quelle modification sera enregistrée, où elle sera partagée et comment revenir à une situation compréhensible lorsque quelque chose d'inattendu se produit.

[GIT-ADD] Git - git-add Documentation — Documentation officielle Git — https://git-scm.com/docs/git-add

[GIT-CONFIG] Git - git-config Documentation — Documentation officielle Git — https://git-scm.com/docs/git-config

[GIT-FIRST-CONFIG] Git - First-Time Git Setup — Livre Pro Git publié sur le site officiel Git — https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

[GIT-INIT] Git - git-init Documentation — Documentation officielle Git — https://git-scm.com/docs/git-init

[GIT-INSTALL] Git - Install — Site officiel Git — https://git-scm.com/install/

[GIT-INSTALL-LINUX] Git - Install for Linux — Site officiel Git — https://git-scm.com/install/linux

[GIT-INSTALL-MACOS] Git - Install for macOS — Site officiel Git — https://git-scm.com/install/mac

[GIT-INSTALL-WINDOWS] Git - Install for Windows — Site officiel Git — https://git-scm.com/install/windows

[GIT-OVERVIEW] Git - git Documentation — Documentation officielle Git — https://git-scm.com/docs/git

[GIT-RECORDING-CHANGES] Git - Recording Changes to the Repository — Livre Pro Git publié sur le site officiel Git — https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

[GIT-STATUS] Git - git-status Documentation — Documentation officielle Git — https://git-scm.com/docs/git-status

[GITHUB-2FA-CONFIG] Configuring two-factor authentication - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication

[GITHUB-2FA-RECOVERY] Configuring two-factor authentication recovery methods - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication-recovery-methods

[GITHUB-ABOUT-GIT-EN] About Git - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/get-started/using-git/about-git

[GITHUB-ABOUT-GIT-FR] À propos de Git - Documentation GitHub — Documentation officielle GitHub — https://docs.github.com/fr/get-started/using-git/about-git

[GITHUB-ACCOUNT-CREATION] Creating an account on GitHub - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/account-and-profile/how-tos/account-management/creating-an-account-on-github

[GITHUB-DESKTOP-SUPPORTED-OS] Supported operating systems for GitHub Desktop - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/desktop/overview/supported-operating-systems-for-github-desktop

[GITHUB-GLOSSARY-EN] GitHub glossary - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/get-started/learning-about-github/github-glossary

[GITHUB-GLOSSARY-FR] glossaire GitHub - Documentation GitHub — Documentation officielle GitHub — https://docs.github.com/fr/get-started/learning-about-github/github-glossary?showiframe=true

[GITHUB-SET-UP-GIT] Set up Git - GitHub Docs — Documentation officielle GitHub — https://docs.github.com/en/get-started/git-basics/set-up-git

De la première commande à la collaboration

Ce guide conduit pas à pas du niveau débutant vers une utilisation autonome et structurée de Git et GitHub. Chaque notion est expliquée avant d'être mise en pratique sur un projet simple.

- Comprendre ce qu'est le versionnement, sans jargon inutile.
- Installer et configurer Git, puis créer son premier dépôt local.
- Enregistrer, retrouver et corriger ses modifications en confiance.
- Travailler avec les branches et les fusionner sans crainte.
- Publier sur GitHub, collaborer et organiser un projet lisible.

Public visé : toute personne qui débute avec Git et GitHub, sans expérience préalable du suivi de versions.