⎇ GIT

GUIDE COMPLET · NIVEAU DÉBUTANT

Apprendre Git

Le système de contrôle de version expliqué pas à pas, des concepts de base jusqu'à votre premier dépôt sur GitHub.

Aucune connaissance préalable requise.

Comprendre · Installer · Versionner · Collaborer

Table des matières

CHAPITRE 1

Introduction : qu'est-ce que Git ?

Avant d'écrire la moindre commande, prenons le temps de comprendre *pourquoi* Git existe et quel problème concret il résout. C'est la clé pour que tout le reste devienne logique.

Le problème : gérer les versions d'un travail

Imaginez que vous écrivez un mémoire. Pour ne rien perdre, vous créez sans doute des fichiers comme :

```
memoire.docx
memoire_v2.docx
memoire_v2_final.docx
memoire_v2_final_VRAIMENT_final.docx
memoire_corrections_prof.docx
```

Cette méthode fonctionne mal : on ne sait plus quelle version est la bonne, on ne sait pas **ce qui a changé** entre deux fichiers, et collaborer à plusieurs devient un cauchemar. **Git résout exactement ce problème**, mais de façon propre et puissante.

> Définition
> **Git** est un **système de contrôle de version** (en anglais *VCS, Version Control System*). C'est un logiciel qui enregistre l'historique des modifications de vos fichiers, vous permet de revenir en arrière à tout moment, et facilite le travail en équipe.

L'analogie de la sauvegarde de jeu vidéo

Le meilleur moyen de comprendre Git est de penser aux **points de sauvegarde** d'un jeu vidéo. À chaque moment important, vous sauvegardez votre partie. Si quelque chose tourne mal, vous rechargez la dernière sauvegarde. Mieux encore : vous pouvez explorer un chemin risqué, et si ça échoue, revenir au point de sauvegarde sans rien perdre.

Dans Git, chaque sauvegarde s'appelle un **commit**. Git garde la liste complète de tous vos commits : c'est votre **historique**. Vous pouvez consulter n'importe quelle version passée, comparer deux versions, ou repartir d'un point précis.

Git n'est pas GitHub !

C'est la confusion la plus fréquente chez les débutants, alors clarifions-la tout de suite.

Git	GitHub
Un **logiciel** installé sur votre ordinateur.	Un **site web** (un service en ligne).
Gère l'historique de vos fichiers **en local**.	Héberge vos dépôts Git **dans le cloud**.
Fonctionne sans connexion Internet.	Sert à **partager** et collaborer en ligne.
Créé par Linus Torvalds en 2005.	Une plateforme parmi d'autres : GitLab, Bitbucket…

> À retenir
> Git est l'**outil** ; GitHub est un **endroit où ranger** ce que l'outil produit. Une bonne image : Git est comme la prise de photos, GitHub est comme l'album partagé en ligne. On peut très bien utiliser Git sans GitHub.

Pourquoi apprendre Git ?

- **Ne plus jamais perdre son travail** : tout l'historique est conservé.
- **Expérimenter sans risque** : grâce aux branches, vous testez des idées sans casser ce qui fonctionne.
- **Collaborer à plusieurs** : plusieurs personnes peuvent travailler sur le même projet sans s'écraser mutuellement.
- **Comprendre ce qui a changé** : Git montre précisément quelles lignes ont été ajoutées, modifiées ou supprimées.
- **C'est un standard professionnel** : Git est utilisé partout dans le développement logiciel, mais aussi pour des documents, de la configuration, de la rédaction technique…

CHAPITRE 2

Installation et configuration

Mettons en place votre environnement. Cette étape ne se fait qu'une seule fois.

Étape 1 — Installer Git

Sur Windows

Rendez-vous sur `git-scm.com`, téléchargez l'installateur et lancez-le. Acceptez les options par défaut (elles conviennent parfaitement à un débutant). L'installation inclut un outil appelé **Git Bash**, un terminal qui vous servira à taper vos commandes.

Sur macOS

Le plus simple est d'ouvrir le Terminal et de taper `git --version`. Si Git n'est pas présent, macOS vous proposera de l'installer automatiquement. Sinon, on peut passer par le gestionnaire Homebrew avec `brew install git`.

Sur Linux (Debian/Ubuntu)

```
$ sudo apt update
$ sudo apt install git
```

Étape 2 — Vérifier l'installation

Ouvrez un terminal (Git Bash sur Windows, Terminal sur Mac/Linux) et tapez :

```
$ git --version
git version 2.49.0
```

Si un numéro de version s'affiche, félicitations : Git est installé !

> Le terminal, pas de panique
> Le terminal (ou « ligne de commande ») est une fenêtre où l'on tape des instructions en texte plutôt que de cliquer. Cela impressionne au début, mais vous n'aurez besoin que d'une poignée de commandes. Le symbole `$` au début des exemples représente l'invite du terminal : **ne le tapez pas**, il indique simplement « ici vous pouvez écrire ».

Étape 3 — Se présenter à Git

Git attache votre nom et votre adresse e-mail à chaque commit, pour savoir qui a fait quoi. Configurez-les une bonne fois pour toutes :

```
$ git config --global user.name "Votre Nom"
$ git config --global user.email "vous@exemple.com"
```

L'option `--global` signifie « pour tous mes projets sur cet ordinateur ». Vous pouvez vérifier votre configuration avec :

```
$ git config --list
```

Étape 4 — Choisir le nom de la branche par défaut

Historiquement, la branche principale d'un projet s'appelait `master`. La convention moderne, adoptée par GitHub, GitLab et l'ensemble de l'industrie, est désormais `main`. Configurez-le ainsi :

```
$ git config --global init.defaultBranch main
```

> Bon à savoir
> Vous croiserez encore `master` dans d'anciens projets ou tutoriels : c'est exactement la même chose qu'une branche, juste un nom différent. La future version majeure de Git utilisera `main` par défaut, mais le configurer dès maintenant vous évite toute surprise.

CHAPITRE 3

Les concepts fondamentaux

Ce chapitre est le plus important du cours. Si vous comprenez bien ces quelques concepts, toutes les commandes deviendront évidentes. Prenez votre temps ici.

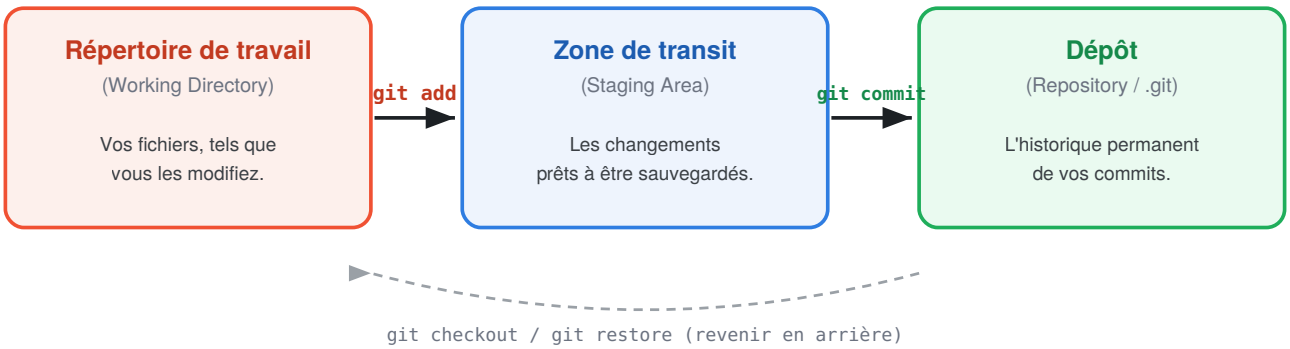
Le dépôt (repository)

> Définition
> Un **dépôt** (ou *repository*, souvent abrégé « repo ») est simplement un dossier de projet que Git surveille. Concrètement, c'est un dossier normal dans lequel Git a créé un sous-dossier caché nommé `.git`, où il stocke tout l'historique.

Tant que vous ne supprimez pas ce dossier `.git`, votre historique complet est préservé. Transformer un dossier ordinaire en dépôt Git se fait avec une seule commande (`git init`), que nous verrons au chapitre suivant.

Les trois zones de Git

C'est **le** concept central. Dans un projet Git, vos fichiers transitent par trois zones distinctes. Comprendre ce voyage rend tout limpide.



Le voyage d'une modification à travers les trois zones de Git.

1. **Le répertoire de travail** : c'est votre dossier de projet visible, là où vous créez et modifiez réellement vos fichiers.
2. **La zone de transit** (aussi appelée *index* ou *staging area*) : une zone d'attente où vous placez les changements que vous souhaitez inclure dans votre prochaine sauvegarde. C'est comme préparer les articles que vous allez mettre dans un colis avant de l'expédier.
3. **Le dépôt** : quand vous validez (*commit*), les changements de la zone de transit sont enregistrés de façon permanente dans l'historique.

Pourquoi une zone de transit ?
Elle vous laisse **choisir précisément** ce qui entre dans chaque commit. Vous avez modifié 5 fichiers mais 3 seulement concernent une même tâche ? Vous ne mettez en transit que ces 3-là, et créez un commit propre et cohérent. C'est ce qui rend l'historique Git si lisible.

Le commit : une photo instantanée

Définition
Un **commit** est un instantané (une « photo ») de l'état de votre projet à un moment donné, accompagné d'un message décrivant ce qui a changé. Chaque commit possède un identifiant unique (un long code comme `a3f9c2e…`) et connaît le commit qui le précède, ce qui forme une chaîne : votre historique.

Le cycle de vie d'un fichier

Du point de vue de Git, un fichier passe par différents états :

État	Signification
Non suivi (untracked)	Le fichier existe dans votre dossier mais Git ne le surveille pas encore. Il vient typiquement d'être créé.
Suivi & modifié (modified)	Git connaît ce fichier et a détecté qu'il a changé depuis le dernier commit.
En transit (staged)	Les modifications ont été ajoutées à la zone de transit, prêtes pour le prochain commit.
Validé (committed)	Les modifications sont enregistrées en sécurité dans l'historique du dépôt.

La commande `git status`, que nous découvrons dans le prochain chapitre, vous indique en permanence dans quel état se trouve chaque fichier. Ce sera votre meilleure amie.

CHAPITRE 4

Premiers pas : créer et gérer un dépôt

Place à la pratique ! Nous allons créer un projet, faire nos premiers commits, et consulter l'historique. Ouvrez votre terminal et suivez les étapes.

1 Créer un dépôt avec `git init`

Créons un dossier de projet et transformons-le en dépôt Git :

```
$ mkdir mon-projet      # crée un dossier
$ cd mon-projet         # entre dedans
$ git init              # initialise Git ici
Initialized empty Git repository in /…/mon-projet/.git/
```

Ce message confirme que le dossier caché `.git` a été créé. Votre dépôt est prêt.

2 Vérifier l'état avec `git status`

Créons un premier fichier, puis demandons à Git où en sont les choses :

```
$ echo "Bonjour Git" > lisez-moi.txt
$ git status
Sur la branche main
Aucun commit

Fichiers non suivis :
  (utilisez "git add <fichier>…" pour inclure dans ce qui sera validé)
        lisez-moi.txt
```

Git voit le fichier mais le considère **non suivi**. Il nous indique même la commande à utiliser ensuite : `git add`.

> Conseil de débutant
> Tapez `git status` très souvent : avant et après chaque action. Git est verbeux et vous guide : ses messages indiquent presque toujours la prochaine commande utile. Lisez-les !

3 Mettre en transit avec `git add`

Plaçons le fichier dans la zone de transit :

```
$ git add lisez-moi.txt
```

Pour ajouter *tous* les fichiers modifiés d'un coup, on utilise un point (qui signifie « le dossier courant ») :

```
$ git add .                # ajoute tout
```

4 Valider avec `git commit`

Enregistrons l'instantané dans l'historique, avec un message descriptif (`-m`) :

```
$ git commit -m "Ajout du fichier lisez-moi"
[main (commit racine) a1b2c3d] Ajout du fichier lisez-moi
 1 file changed, 1 insertion(+)
```

Bravo, vous venez de créer votre premier commit ! Votre travail est désormais en sécurité dans l'historique.

⎇ Le cycle de base à mémoriser

Tout le travail quotidien avec Git tourne autour de ces trois commandes, dans cet ordre :

`git add` → **`git commit`** → (et plus tard) **`git push`**

Je modifie mes fichiers, je sélectionne ce que je veux sauvegarder (**`add`**), je crée une photo de cet état (**`commit`**). Répétez ce cycle aussi souvent que nécessaire.

5 Consulter l'historique avec `git log`

```
$ git log
commit a1b2c3d4e5f6… (HEAD -> main)
Author: Votre Nom <vous@exemple.com>
Date:   Thu May 28 14:22:01 2026

    Ajout du fichier lisez-moi
```

Pour un affichage plus compact, une ligne par commit :

```
$ git log --oneline
a1b2c3d Ajout du fichier lisez-moi
```

> Le pointeur HEAD
> Dans la sortie ci-dessus, **`HEAD`** est simplement un pointeur qui indique « où vous vous trouvez actuellement » dans l'historique — en général, le dernier commit de votre branche active.

Bien rédiger ses messages de commit

Un bon message explique **ce que fait** le commit, à l'impératif présent. Cela rend l'historique lisible des mois plus tard.

✓ À faire	✗ À éviter
« Corrige le calcul du total du panier »	« modifs »
« Ajoute la page de contact »	« truc »
« Met à jour la documentation d'installation »	« asdfgh »

CHAPITRE 5

Voir et annuler ses changements

Git brille par sa capacité à montrer précisément ce qui a changé, et à revenir en arrière en toute sécurité. Voici les outils essentiels.

Voir les différences avec `git diff`

Après avoir modifié un fichier (mais avant de l'ajouter), `git diff` vous montre exactement les lignes changées :

```
$ git diff
--- a/lisez-moi.txt
+++ b/lisez-moi.txt
@@ -1 +1,2 @@
 Bonjour Git
+Ligne ajoutée aujourd'hui
```

Les lignes précédées d'un `+` sont ajoutées, celles précédées d'un `-` sont supprimées. Pour voir ce qui est déjà en transit : `git diff --staged`.

Annuler des modifications avec `git restore`

Vous avez modifié un fichier et souhaitez revenir à la dernière version validée (abandonner vos changements non sauvegardés) ?

```
$ git restore lisez-moi.txt
```

> Attention
> Cette commande **supprime définitivement** les modifications non validées du fichier concerné. Utilisez-la seulement si vous êtes sûr de vouloir jeter ces changements.

Vous avez ajouté un fichier en transit par erreur ? Retirez-le du transit (sans perdre vos modifications) avec :

```
$ git restore --staged lisez-moi.txt
```

> Ancien et nouveau vocabulaire
> Les commandes `git restore` et `git switch` sont les versions modernes (depuis Git 2.23) qui remplacent l'unique `git checkout`, longtemps surchargé et déroutant. Vous verrez encore `git checkout` partout ; il fonctionne toujours, mais `restore` et `switch` sont plus clairs pour débuter.

Ignorer des fichiers avec `.gitignore`

Certains fichiers ne doivent **jamais** entrer dans Git : fichiers temporaires, mots de passe, dossiers de dépendances volumineux, fichiers générés automatiquement. On les liste dans un fichier nommé `.gitignore` à la racine du projet :

```
# Exemple de fichier .gitignore
# Tout ce qui est listé ici sera ignoré par Git

node_modules/        # dossier de dépendances
*.log                # tous les fichiers .log
.env                 # fichier de secrets/mots de passe
.DS_Store            # fichier système macOS
/build/              # dossier de fichiers générés
```

> Réflexe de sécurité
> Ne mettez **jamais** de mots de passe, clés d'API ou informations sensibles dans Git. Une fois validés et envoyés en ligne, ils restent dans l'historique même après suppression. Ajoutez ces fichiers à `.gitignore` dès le départ.

CHAPITRE 6

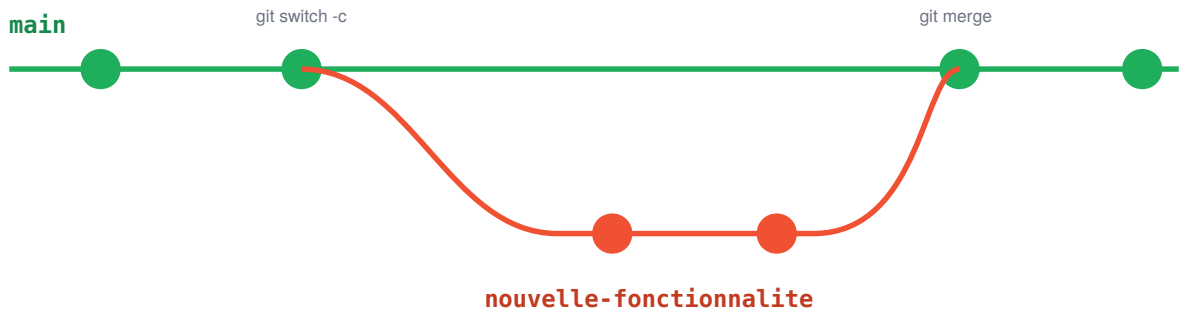
Les branches

Les branches sont la fonctionnalité la plus puissante de Git. Elles vous permettent de développer des idées en parallèle sans jamais mettre en danger votre travail principal.

Qu'est-ce qu'une branche ?

Reprenons l'analogie du jeu vidéo. Une **branche**, c'est comme créer une ligne de temps parallèle : vous partez d'un point de sauvegarde, vous explorez une nouvelle idée (une fonctionnalité, une correction…), et si elle est concluante, vous la **fusionnez** avec la ligne principale. Sinon, vous l'abandonnez sans aucune conséquence.

> Définition
> Une **branche** est une ligne de développement indépendante. La branche principale s'appelle `main` par convention. Techniquement, une branche n'est qu'un pointeur léger vers un commit : c'est pour cela que créer une branche dans Git est instantané.



On crée une branche depuis `main`, on y fait des commits, puis on la fusionne dans `main`.

Lister, créer et basculer entre branches

```
$ git branch                      # liste les branches
* main

$ git switch -c nouvelle-fonctionnalite  # crée ET bascule
Basculé sur la nouvelle branche 'nouvelle-fonctionnalite'

$ git switch main                 # revenir sur main
```

L'option `-c` signifie *create* (créer). L'astérisque `*` dans `git branch` indique la branche sur laquelle vous vous trouvez actuellement.

> Équivalent avec checkout
> Dans d'anciens tutoriels, vous verrez `git checkout -b nom` pour créer et basculer, et `git checkout nom` pour basculer. C'est exactement équivalent à `git switch`.

Fusionner une branche avec `git merge`

Une fois le travail terminé sur votre branche, vous la fusionnez dans `main`. La règle : on se place **sur la branche qui doit recevoir** les changements, puis on fusionne l'autre.

```
$ git switch main                    # on se place sur main
$ git merge nouvelle-fonctionnalite   # on intègre la branche
Updating a1b2c3d..f4e5d6c
Fast-forward
 fichier.txt | 10 ++++++++++
```

Une fois fusionnée et devenue inutile, on peut supprimer la branche :

```
$ git branch -d nouvelle-fonctionnalite
```

Les conflits de fusion

Un **conflit** survient quand deux branches ont modifié *les mêmes lignes* d'un même fichier. Git ne peut pas deviner quelle version garder ; il vous demande de trancher. Dans le fichier concerné, Git insère des marqueurs :

```
<<<<<<< HEAD
Texte de la branche main
=======
Texte de l'autre branche
>>>>>>> nouvelle-fonctionnalite
```

Pour résoudre : ouvrez le fichier, choisissez la bonne version (en supprimant les marqueurs `<<<`, `===`, `>>>` et le texte non désiré), puis validez le résultat :

```
$ git add fichier.txt      # marque le conflit comme résolu
$ git commit               # finalise la fusion
```

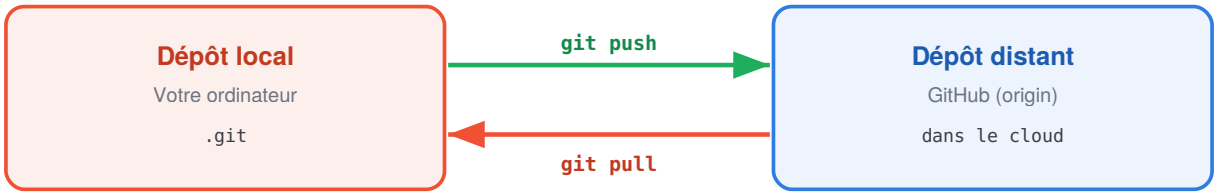
> Pas de panique face à un conflit
> Les conflits font partie du travail normal avec Git, ce n'est pas une erreur de votre part. Travailler par petites étapes et fusionner souvent réduit grandement leur fréquence et leur taille.

CHAPITRE 7

Travailler avec un dépôt distant (GitHub)

Jusqu'ici, tout s'est passé sur votre ordinateur. Connectons maintenant votre dépôt à un service en ligne comme GitHub, pour sauvegarder votre travail dans le cloud et collaborer.

> Définition
> Un **dépôt distant** (*remote*) est une copie de votre dépôt hébergée ailleurs, généralement en ligne. Par convention, le dépôt distant principal s'appelle `origin`. Vous synchronisez votre dépôt local et le distant en **envoyant** (push) et en **récupérant** (pull) des commits.



`push` envoie vos commits vers le distant ; `pull` récupère ceux des autres.

Cas 1 : récupérer un projet existant avec `git clone`

Pour copier un dépôt existant (le vôtre ou celui d'autrui) sur votre machine :

```
$ git clone https://github.com/utilisateur/projet.git
```

Cette commande télécharge tout le projet, son historique complet, et configure automatiquement le distant `origin`. Vous n'avez rien d'autre à faire pour commencer.

Cas 2 : publier un projet local existant

Vous avez créé un dépôt en local (chapitre 4) et voulez l'envoyer sur GitHub ? D'abord, créez un dépôt vide sur GitHub via le site. Puis, dans votre terminal :

```
# 1. relier le dépôt local au dépôt distant
$ git remote add origin https://github.com/vous/projet.git

# 2. envoyer la branche main pour la première fois
$ git push -u origin main
```

L'option `-u` (pour *upstream*) lie votre branche locale `main` à celle du distant. Grâce à elle, les fois suivantes, un simple `git push` suffira.

Le travail quotidien : push et pull

```
$ git push     # envoie vos nouveaux commits sur GitHub
$ git pull     # récupère les commits des autres et les intègre
```

> Bon réflexe en équipe
> Faites un `git pull` **avant de commencer à travailler** chaque jour, pour partir de la version la plus à jour. Cela évite bien des conflits.

Et `git fetch` dans tout ça ?

`git fetch` récupère les nouveautés du distant **sans** les fusionner dans votre travail. C'est utile pour « regarder ce qui a changé » avant de décider. En réalité, `git pull` = `git fetch` + `git merge` automatique.

> Authentification
> Lors de votre premier `push`, GitHub vous demandera de vous identifier. Aujourd'hui, on n'utilise plus le mot de passe du compte mais un **jeton d'accès personnel** (Personal Access Token) ou une **clé SSH**. GitHub vous guide pour en créer un dans les paramètres de votre compte ; c'est une configuration ponctuelle.

CHAPITRE 8

Cas pratique : un projet de A à Z

Rassemblons tout ce que vous avez appris dans un scénario complet et réaliste. Suivez ce déroulé comme un mini-projet guidé.

Le scénario

Vous démarrez un petit site web. Vous allez le versionner, le publier sur GitHub, ajouter une fonctionnalité sur une branche dédiée, puis la fusionner.

1. Initialiser le projet

```
$ mkdir mon-site && cd mon-site
$ git init
$ echo "<h1>Mon site</h1>" > index.html
```

2. Premier commit

```
$ git add .
$ git commit -m "Crée la page d'accueil"
```

3. Publier sur GitHub

```
$ git remote add origin https://github.com/vous/mon-site.git
$ git push -u origin main
```

4. Développer une fonctionnalité sur une branche

```
$ git switch -c page-contact
$ echo "<h1>Contact</h1>" > contact.html
$ git add contact.html
$ git commit -m "Ajoute la page de contact"
```

5. Fusionner dans main et publier

```
$ git switch main
$ git merge page-contact
$ git branch -d page-contact
$ git push
```

⎇ Votre boucle quotidienne, en une image

`git pull` (récupérer) → travailler → `git add` → `git commit -m "…"` → `git push` (publier)

Pour une nouvelle tâche un peu conséquente, ajoutez au début : `git switch -c ma-tache`, et à la fin, fusionnez dans `main`.

CHAPITRE 9

Aide-mémoire (toutes les commandes)

Gardez cette page sous la main. Elle résume l'essentiel de ce que vous utiliserez au quotidien.

Configuration (une seule fois)

Commande	Action
`git config --global user.name "…"`	Définir son nom
`git config --global user.email "…"`	Définir son e-mail
`git --version`	Vérifier l'installation

Créer & sauvegarder

Commande	Action
`git init`	Transformer un dossier en dépôt Git
`git status`	Voir l'état des fichiers (à utiliser souvent !)
`git add <fichier>`	Mettre un fichier en transit
`git add .`	Mettre tous les changements en transit
`git commit -m "message"`	Créer un commit (une sauvegarde)
`git log --oneline`	Voir l'historique de façon compacte

Inspecter & annuler

Commande	Action
`git diff`	Voir les modifications non encore ajoutées
`git diff --staged`	Voir ce qui est en transit
`git restore <fichier>`	Annuler les modifications d'un fichier
`git restore --staged <fichier>`	Retirer un fichier du transit

Branches

Commande	Action
`git branch`	Lister les branches
`git switch -c <nom>`	Créer une branche et y basculer
`git switch <nom>`	Basculer sur une branche existante
`git merge <nom>`	Fusionner une branche dans la branche actuelle
`git branch -d <nom>`	Supprimer une branche

Dépôt distant (GitHub)

Commande	Action
`git clone <url>`	Copier un dépôt distant en local
`git remote add origin <url>`	Relier un dépôt local à un distant
`git push`	Envoyer ses commits vers le distant
`git push -u origin main`	Premier envoi (lie la branche)
`git pull`	Récupérer et intégrer les commits distants
`git fetch`	Récupérer sans fusionner

CHAPITRE 10

Bonnes pratiques & erreurs courantes

Voici les habitudes qui distinguent rapidement un usage maladroit d'un usage serein de Git.

Les bonnes habitudes à prendre

- **Commitez souvent, par petites touches.** Un commit = une idée, une tâche cohérente. C'est plus facile à comprendre et à annuler si besoin.
- **Écrivez des messages clairs et descriptifs.** Votre « vous » du futur vous remerciera.
- **Utilisez `git status` et `git diff` avant chaque commit** pour vérifier ce que vous allez réellement enregistrer.
- **Créez une branche pour chaque nouvelle fonctionnalité** plutôt que de tout faire sur `main`.
- **Faites un `git pull` régulièrement** quand vous travaillez à plusieurs.
- **Configurez un `.gitignore` dès le début** du projet.

Les pièges classiques du débutant

L'erreur	Comment l'éviter
Committer des mots de passe ou clés secrètes	Toujours les placer dans `.gitignore` en amont
Messages de commit vagues (« maj », « fix »)	Décrire l'action concrète réalisée
Oublier `git add` avant `git commit`	Le commit ne contient que ce qui est en transit : vérifier avec `git status`
Tout faire sur la branche `main`	Isoler chaque travail sur sa propre branche
Committer des fichiers énormes/inutiles	Filtrer avec `.gitignore`
Paniquer face à un message d'erreur	Lire le message : Git suggère presque toujours la solution

> La règle d'or
> Tant que vous avez **committé** votre travail, il est presque impossible de le perdre : Git conserve tout. La plupart des « catastrophes » de débutant concernent du travail *non encore committé*. Donc, dans le doute : committez.

ANNEXE

Glossaire & pour aller plus loin

Glossaire des termes clés

Terme	Définition
Dépôt (repository)	Un projet suivi par Git, contenant le dossier `.git`.
Commit	Un instantané sauvegardé de votre projet à un instant donné.
Branche	Une ligne de développement indépendante.
main	Le nom moderne de la branche principale (anciennement `master`).
Zone de transit	Espace d'attente des changements avant un commit (staging area).
HEAD	Pointeur indiquant votre position actuelle dans l'historique.
Remote / origin	Un dépôt distant ; `origin` en est le nom par défaut.
Push / Pull	Envoyer / récupérer des commits entre local et distant.
Clone	Copier un dépôt distant complet en local.
Merge	Fusionner les changements d'une branche dans une autre.
Conflit	Situation où Git ne peut pas fusionner automatiquement deux versions.
Fork	Copie personnelle d'un dépôt d'autrui sur GitHub (pour contribuer).

Pour continuer votre apprentissage

- **La documentation officielle** : `git-scm.com` propose le livre « Pro Git » gratuitement, en français.
- **L'aide intégrée** : tapez `git help <commande>` (ex : `git help commit`) pour la documentation d'une commande.
- **Pratiquez** : créez un dépôt « bac à sable » et expérimentez sans crainte. C'est la meilleure façon d'apprendre.
- **Les prochaines étapes** : une fois à l'aise, explorez les *pull requests* sur GitHub, `git stash` (mettre du travail de côté temporairement) et `git rebase` (réorganiser l'historique).

⎇ Le mot de la fin

Git s'apprend surtout par la pratique. Vous n'avez pas besoin de tout maîtriser : les commandes des chapitres 4 et 7 couvrent déjà 90 % de l'usage quotidien. Lancez-vous, committez souvent, et n'ayez pas peur de l'erreur : votre historique est votre filet de sécurité.