Cours transverse

Comprendre Git

Le jour où vous arriverez dans une équipe de développement, personne ne vous demandera si vous connaissez Git. On partira du principe que c'est le cas.

On vous donnera l'adresse d'un projet, et on s'attendra à ce que vous sachiez le récupérer, travailler sur votre partie et proposer vos modifications sans perturber vos collègues. Ce cours est là pour vous y préparer.

- ✓ Enregistrer et relire l'histoire d'un projet
- ✓ Résoudre un conflit sans paniquer
- ✓ Protéger le code et les secrets d'un projet
- ✓ Travailler sur une branche sans rien casser
- ✓ Collaborer via GitHub ou GitLab
- ✓ Choisir où héberger ses dépôts

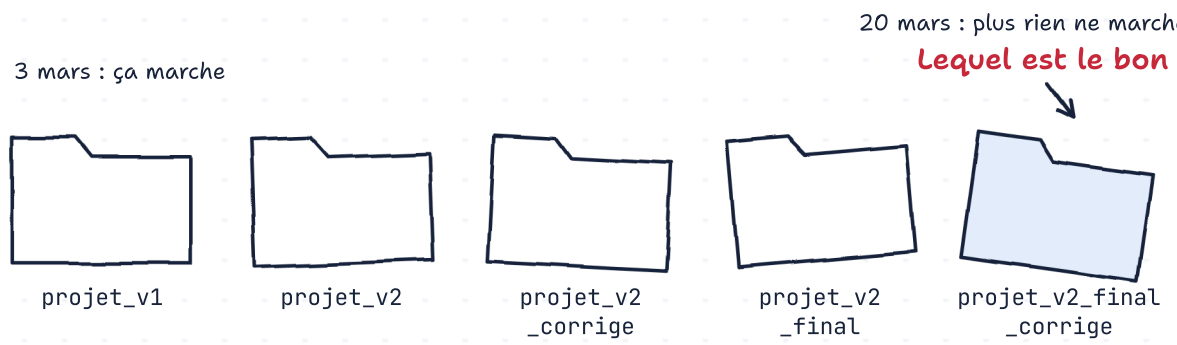
Le plan, dans l'ordre de lecture

01 Pourquoi gérer des versions ?
02 Centralisé ou distribué
03 L'histoire de Git
04 Comment Git voit votre projet
05 Votre premier dépôt
06 Les branches et les fusions
07 Revenir en arrière sans casse
08 Travailler à plusieurs
09 GitHub, GitLab et l'automatisation
10 Confidentialité et sécurité
11 Héberger ses dépôts soi-même
12 L'essentiel à retenir

Chapitre 1

Pourquoi gérer des versions ?

Commençons par une situation que vous avez peut-être déjà vécue, même sans écrire une ligne de code. Vous travaillez sur un document important. Avant une grosse modification, vous le dupliquez par prudence. Puis vous recommencez. Quelques semaines plus tard, votre dossier ressemble à ceci.



La méthode a un mérite : elle part d'un bon réflexe. Elle ne dit pourtant ni ce qui a changé, ni pourquoi, ni qui l'a fait.

Sur un petit projet personnel, on s'en sort. En entreprise, c'est une autre histoire. Une application web représente des centaines de fichiers : code source, feuilles de style, configuration, documentation. Ils changent tous les jours, et plusieurs personnes les modifient en même temps.

Arrivent alors les questions qu'on entend vraiment dans une équipe :

- « Ça marchait hier. Qu'est-ce qui a changé depuis ? »
- « Pourquoi cette ligne a-t-elle été ajoutée, et par qui ? »
- « Peut-on revenir à la version d'avant la mise en ligne ? »
- « Si je modifie ce fichier, est-ce que j'écrase le travail de Sam ? »

Aucune copie de dossier ne répond à ces questions. Il nous faut un outil qui tienne le journal du projet à notre place.

L'idée

Cet outil enregistre l'état de vos fichiers à chaque étape importante, avec la date, le nom de l'auteur et une explication. Vous gardez un seul dossier de travail, bien propre, et toutes les étapes passées restent accessibles.

Vous pouvez alors relire l'histoire du projet, comparer deux étapes, revenir en arrière, ou faire avancer plusieurs travaux en parallèle avant de les réunir.

Ce type d'outil s'appelle un **système de gestion de versions**, ou VCS en anglais (*Version Control System*). Gardez cette image en tête : c'est la mémoire du projet.

Si vous avez déjà ouvert l'historique d'un document Google Docs, vous avez une première idée du résultat. La différence, c'est qu'ici vous décidez vous-même quand enregistrer une étape, et vous expliquez pourquoi.

Point de vigilance

Ces outils brillent sur les fichiers texte, comme le code. Ils savent conserver une image ou une vidéo, mais ne peuvent pas vous montrer ce qui a changé à l'intérieur, et les fichiers lourds alourdissent vite l'historique.

Il existe plusieurs outils de ce genre, et ils ne sont pas tous construits de la même façon. Cette différence de construction explique en grande partie le succès de Git.

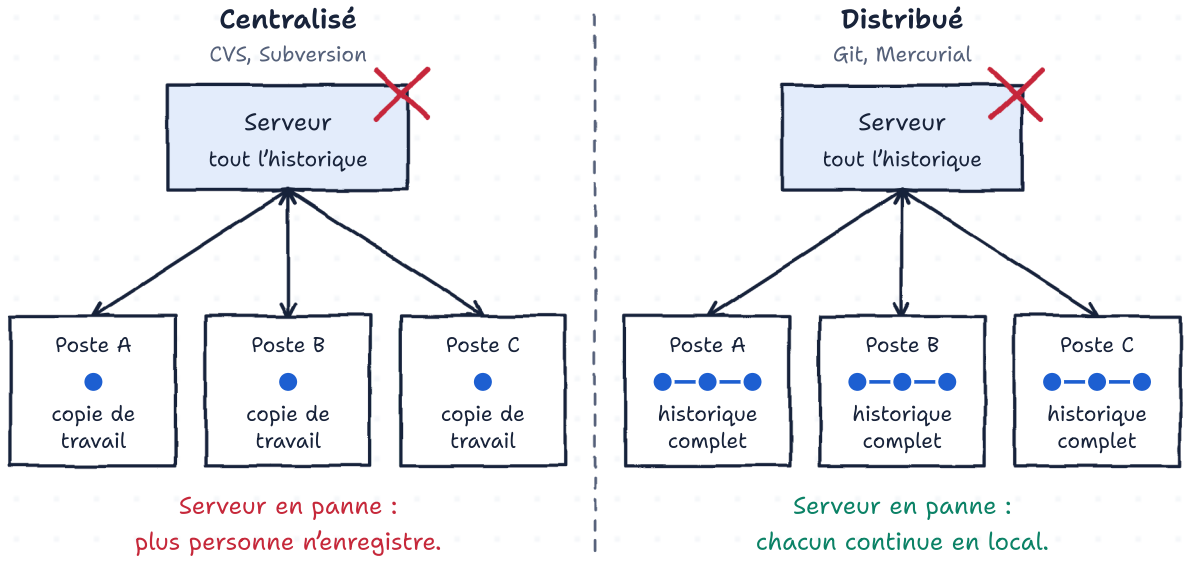
Chapitre 2

Centralisé ou distribué

Prenons l'image d'une bibliothèque d'archives. Il existe deux façons de l'organiser.

Première option, un seul bâtiment garde toutes les archives. Vous empruntez la dernière édition d'un document, mais pour consulter une édition ancienne ou déposer la vôtre, il faut passer sur place. C'est le modèle **centralisé**. Des outils comme CVS, très utilisé dans les années 1990, puis Subversion (SVN), son successeur, fonctionnent ainsi : un serveur central détient l'historique, et votre poste n'a qu'une copie de travail.

Seconde option, chaque lecteur repart avec une copie complète des archives. Il consulte tout chez lui, même dans le train, et les lecteurs se synchronisent quand ils le souhaitent. C'est le modèle **distribué**, celui de Git et de Mercurial. On parle de DVCS (*Distributed Version Control System*).



Même panne, deux conséquences très différentes. Le schéma montre aussi pourquoi Git est rapide : l'historique est déjà sur votre machine.

Au quotidien, voici ce que cela change.

Situation	Subversion	Git
Consulter l'historique	Il faut interroger le serveur	Immédiat, en local
Enregistrer une étape	Passe par le serveur	Se fait en local
Créer une branche	Opération sur le serveur	Instantané, en local
Travailler dans le train	Très limité	Presque tout est possible

Point de vigilance

Distribué ne veut pas dire sans serveur. En équipe, on désigne presque toujours un dépôt de référence sur lequel tout le monde se synchronise. C'est une règle d'organisation : techniquement, tous les dépôts se valent.

Vous croiserez encore Subversion sur certains projets anciens, donc ne soyez pas surpris. Mais si vous démarrez un projet aujourd'hui, il y a de très fortes chances que ce soit avec Git.

Comment un outil a-t-il pu s'imposer à toute une profession ? Pour le comprendre, il faut remonter en 2005 et regarder une communauté soudain privée de son outil de travail.

Chapitre 3

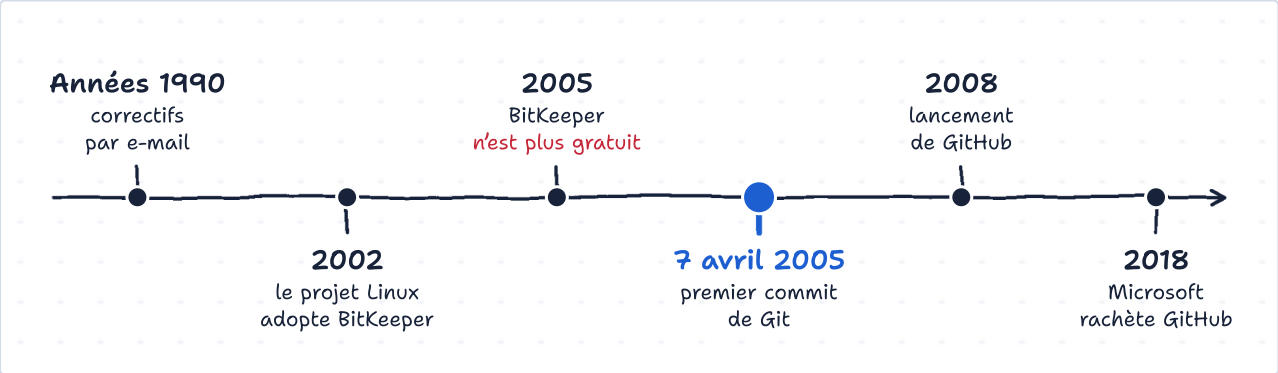
L'histoire de Git

Git est né d'un problème très concret, sur l'un des plus gros projets collaboratifs de la planète : le noyau Linux.

Pendant des années, les contributeurs du monde entier envoient leurs modifications par courriel, sous forme de fichiers de correctifs. Linus Torvalds, le créateur de Linux, les intègre. Il existe bien des outils centralisés, mais il les juge inadaptés. Il considère CVS comme un mauvais modèle, et SVN comme le même modèle mieux habillé.

En 2002, la communauté adopte BitKeeper, un outil distribué et propriétaire que son éditeur laisse utiliser gratuitement sous conditions. Le gain est immédiat. Mais en 2005, à la suite d'une tentative de rétro-ingénierie par un membre de la communauté, les relations se tendent et la gratuité est retirée.

Linus Torvalds se retrouve sans outil. Il regarde les alternatives, dont Monotone, un système distribué aux idées intéressantes, mais trop lent pour un projet de cette taille. Il décide alors d'écrire le sien.



Le 7 avril 2005, il enregistre le premier commit de Git. Il lui faut une dizaine de jours pour obtenir une version utilisable sur le noyau. Quelques mois plus tard, il confie la maintenance à Junio C. Hamano, qui fera mûrir le projet et en assure depuis la continuité.

Un cahier des charges dicté par le terrain

Chaque exigence de départ répond à une difficulté vécue par la communauté Linux.

Ce que vit le projet Linux	Ce que Git doit savoir faire
Des milliers de contributions à intégrer	Fusionner vite et bien
Des développeurs partout, sans chef d'orchestre permanent	Fonctionner de façon distribuée
De nombreux travaux menés en parallèle	Créer des branches sans effort
Un code critique dont dépendent des millions de machines	Détecter toute altération des données

Ce dernier point mérite une explication. Git calcule pour chaque contenu une **empreinte** à partir de ce contenu lui-même. Changez un seul caractère et l'empreinte change. Une corruption ou une modification inattendue se repère donc tout de suite. Git utilise historiquement l'algorithme SHA-1, et une transition vers SHA-256, plus robuste, est en cours.

Pourquoi tout le monde l'a adopté

Trois raisons pratiques expliquent ce succès. D'abord le confort : presque tout se passe en local, c'est donc rapide, et cela fonctionne hors connexion. Ensuite la souplesse des branches, que nous verrons au chapitre 6. Enfin l'universalité : Git est libre, tourne sur Windows, Linux et macOS, et ne dépend d'aucun langage. Un projet PHP, Java, Python ou C# se gère de la même façon.

Il manquait une chose pour que le grand public des développeurs s'y mette : un endroit simple où partager son code. GitHub arrive en 2008 et joue ce rôle. Les éditeurs comme Visual Studio Code ou IntelliJ IDEA intègrent ensuite Git nativement, et la boucle est bouclée.

> **Pour la petite histoire**
> En argot britannique, un *git* est une personne désagréable. Linus Torvalds s'en est amusé : il dit donner à ses projets des noms qui lui ressemblent.

Assez d'histoire. Avant de taper la moindre commande, il nous reste à comprendre comment Git voit votre projet. Trois idées suffisent.

Chapitre 4

Comment Git voit votre projet

Beaucoup de personnes apprennent Git comme une liste de commandes à réciter, et se retrouvent bloquées au premier imprévu. Si vous comprenez les trois idées de ce chapitre, les commandes deviendront logiques.

Première idée : le dépôt

Git doit bien ranger son journal quelque part. Quand vous lui confiez un dossier, il y crée un sous-dossier caché nommé `.git`. Tout s'y trouve : les étapes enregistrées, les lignes de travail, les informations sur chaque auteur.

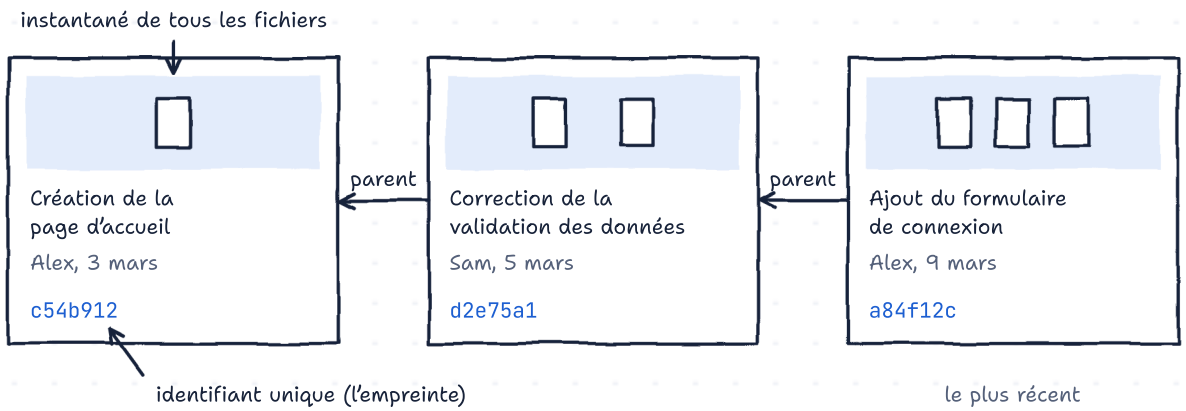
Un dossier suivi de cette façon s'appelle un **dépôt** (*repository*, souvent abrégé en *repo*). Il vit sur votre machine, et vous n'avez besoin d'aucun compte en ligne pour vous en servir.

> **Point de vigilance**
> Supprimer le dossier `.git`, c'est effacer la mémoire du projet. Vos fichiers actuels restent en place, mais l'historique local disparaît avec lui.

Deuxième idée : le commit

Git n'enregistre pas chaque frappe au clavier. C'est vous qui décidez qu'un état du projet mérite d'être retenu. C'est comme prendre une photo : vous figez le projet à un instant donné et vous écrivez une légende au dos.

Cette photo s'appelle un **commit**. Chacun garde aussi le souvenir de la photo précédente, qu'on appelle son parent. C'est ce lien qui permet de remonter le temps.



Chaque flèche pointe vers le commit précédent. Pour lire l'histoire d'un projet, Git part du dernier et remonte de parent en parent.

Pour le plaisir du détail : Git enregistre bien une image complète du projet à chaque commit, et il évite de stocker deux fois ce qui n'a pas changé. C'est ce qui lui permet de rester léger.

Le message, lui, est écrit pour vos collègues et pour vous dans six mois. C'est lui qu'on lira le jour où il faudra comprendre l'origine d'un bug.

À éviter	Bien mieux
`modif`	`Correction de la validation de l'e-mail`
`test2`	`Ajout du formulaire de connexion`
`corrections diverses`	`Suppression du double envoi du formulaire`

Conseil de pro

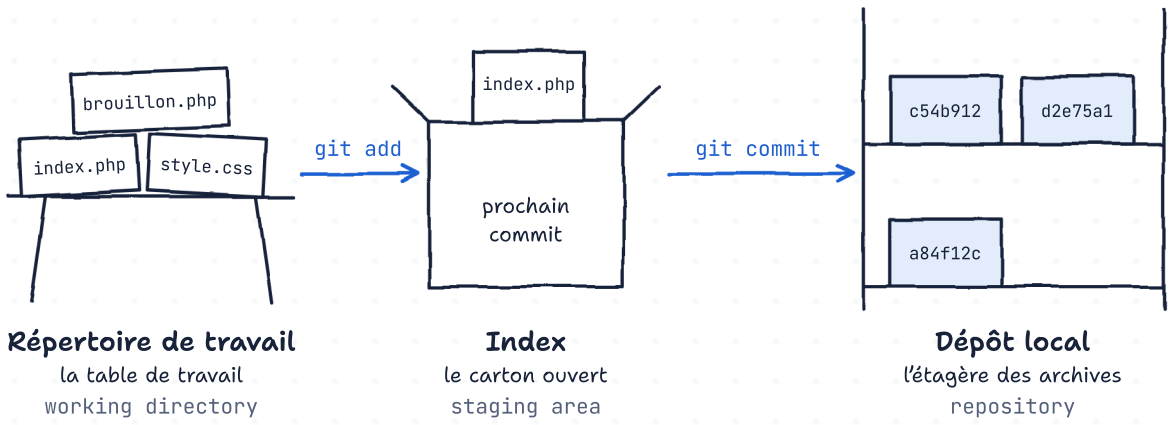
Un commit, une intention. Si votre message a besoin d'un « et », c'est souvent qu'il fallait deux commits. Vous verrez aussi que certaines équipes imposent un format de message : renseignez-vous dès votre arrivée.

Troisième idée : les trois zones

Imaginez une matinée ordinaire. Vous corrigez un bug, puis vous commencez une nouvelle page que vous n'avez pas le temps de finir. Vous voulez enregistrer la

correction, mais surtout pas le brouillon. Il faut donc pouvoir choisir ce qui entre dans le prochain commit.

Pensez à la préparation d'un colis. Vos affaires sont étalées sur la table. Vous mettez dans le carton ce que vous voulez expédier, et seulement cela. Puis vous le fermez, vous l'étiquetez et vous le rangez.



Tant qu'un fichier reste sur la table, il n'entre pas dans le commit. Ici, `brouillon.php` peut attendre tranquillement.

Le **répertoire de travail** contient vos fichiers tels que vous les voyez. L'**index** rassemble ce que vous avez choisi pour le prochain commit. Le **dépôt local** conserve les commits enregistrés.

Vous avez la carte en tête. Passons sur le terrain : nous allons créer un petit site, que nous suivrons jusqu'à la fin du cours.

Chapitre 5

Votre premier dépôt

Installer Git et se présenter

Sur Windows, téléchargez Git depuis git-scm.com. Sur Linux, passez par le gestionnaire de paquets, par exemple `sudo apt install git`. Sur macOS, tapez `git --version` dans le terminal et le système vous proposera de l'installer s'il manque.

Comme chaque commit porte le nom de son auteur, Git a besoin de savoir qui vous êtes. Cette configuration ne se fait qu'une fois par machine.

```
Terminal
$ git config --global user.name "Alex Martin"
$ git config --global user.email "alex@exemple.fr"
$ git config --global init.defaultBranch main
```

La troisième ligne mérite un mot. Par défaut, Git nomme encore `master` la première ligne de travail d'un projet. L'usage s'est fixé sur `main`, et Git 3.0 doit adopter ce nom par défaut. Autant prendre les devants.

Créer le dépôt

Notre fil rouge sera un petit site vitrine. Créez un dossier et confiez-le à Git.

```
Terminal
$ mkdir mon-site
$ cd mon-site
$ git init
Initialized empty Git repository in /home/alex/mon-site/.git/
```

Ajoutez maintenant deux fichiers dans ce dossier, `index.php` et `style.css`, avec le contenu de votre choix. Puis posez à Git la question que vous lui poserez le plus souvent.

```
Terminal
$ git status
On branch main

No commits yet

Untracked files:
    index.php
    style.css
```

`git status` ne modifie rien. La commande vous dit où vous en êtes, et Git annonce ici qu'il voit deux fichiers qu'il ne suit pas encore. Selon la langue de votre système, ces messages peuvent s'afficher en français.

Enregistrer un premier commit

Mettons le premier fichier dans le carton, puis fermons-le.

```
Terminal
$ git add index.php
$ git commit -m "Création de la page d'accueil"
[main (root-commit) d6c8054] Création de la page d'accueil
 1 file changed, 1 insertion(+)

$ git add style.css
$ git commit -m "Ajout de la feuille de style"
```

Pour relire le journal, utilisez `git log`, ou sa version compacte. Les identifiants seront différents chez vous, c'est normal.

```
Terminal
$ git log --oneline
7538dad Ajout de la feuille de style
d6c8054 Création de la page d'accueil
```

Vérifier avant d'enregistrer

Modifiez le titre dans `index.php`, puis demandez à Git de vous montrer la différence.

Terminal

```
$ git diff
--- a/index.php
+++ b/index.php
@@ -1 +1 @@
-<h1>Bienvenue</h1>
+<h1>Bienvenue sur notre site</h1>
```

La ligne en moins correspond à l'ancienne version, la ligne en plus à la nouvelle. `git diff` montre ce qui n'est pas encore dans l'index. Pour relire ce que vous vous apprêtez à enregistrer, utilisez `git diff --staged`.

Vous pouvez aussi ajouter d'un coup toutes les modifications du dossier avec `git add .`. C'est pratique, à condition de savoir ce que vous embarquez.

Le réflexe quotidien

`git status`, `git diff`, `git add`, `git commit`. Dans cet ordre, à chaque fois. Ces quelques secondes de relecture évitent de publier un mot de passe ou du code de test oublié.

Dire à Git ce qu'il doit ignorer

Certains fichiers n'ont rien à faire dans l'historique : les dépendances, qui se réinstallent, et surtout les fichiers de configuration contenant des identifiants. Créez à la racine un fichier `.gitignore` et listez-les.

.gitignore

```
.env
vendor/
node_modules/
```

Pensez à le créer dès le début du projet, avant le premier `git add .`.

Il existe des interfaces graphiques pour tout cela, intégrées à Visual Studio Code ou proposées par des outils comme GitKraken. Elles sont confortables, mais apprenez d'abord la ligne de commande. Les interfaces manipulent les mêmes notions, et c'est au terminal que vous dépannerez une situation délicate.

Pour l'instant, votre histoire est une ligne droite. Mais que faire si vous voulez tester une idée risquée sans toucher à ce qui fonctionne ?

Chapitre 6

Les branches et les fusions

Votre site est en ligne et fonctionne. On vous demande une page de connexion, pour laquelle il faudra trois jours. Le deuxième jour, un bug urgent apparaît en production. Si vous avez tout mélangé dans la même ligne de travail, vous voilà coincé avec du code à moitié terminé.

La solution consiste à ouvrir une ligne de travail parallèle pour la nouveauté. C'est comme une déviation : la route principale reste ouverte, et vous la rejoindrez une fois les travaux terminés. Cette ligne parallèle s'appelle une **branche**. La principale se nomme généralement `main`.

```
Terminal
$ git switch -c feature/connexion
Switched to a new branch 'feature/connexion'
```

L'option `-c` crée la branche et vous place dessus. Créez un fichier `connexion.php`, enregistrez-le dans un commit, puis revenez sur `main` et regardez votre dossier.

```
Terminal
$ git add connexion.php
$ git commit -m "Ajout du formulaire de connexion"
$ git switch main
$ ls
index.php  style.css
```

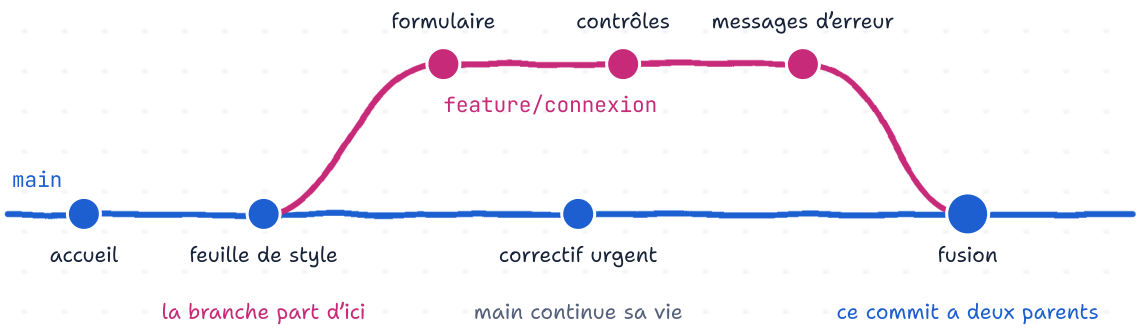
Le fichier a disparu. Rassurez-vous : Git a simplement remis le dossier dans l'état de `main`. Revenez sur votre branche et il sera de nouveau là. C'est souvent à cet instant que les débutants comprennent ce qu'est une branche.

Créer une branche ne coûte presque rien, car Git ne copie aucun fichier. Une branche n'est qu'une petite étiquette posée sur un commit. C'est pour cela qu'on en crée sans hésiter, parfois plusieurs par jour.

Réunir les travaux

La fonctionnalité est prête ? On se place sur la branche qui doit recevoir le travail, puis on fusionne.

```
Terminal
$ git switch main
$ git merge feature/connexion
```



Le correctif urgent a pu partir en production sans attendre la page de connexion. Voilà tout l'intérêt des branches.

Si `main` n'a pas bougé entre-temps, Git se contente d'avancer l'étiquette : c'est ce qu'il appelle un *fast-forward*. Si les deux lignes ont évolué, il crée le commit de fusion que vous voyez sur le croquis.

Quand deux modifications se contredisent

La plupart du temps, Git réunit le travail tout seul, même lorsque deux personnes ont touché le même fichier. Il bloque uniquement quand la même ligne a été modifiée de deux façons différentes. Il ne peut pas deviner laquelle garder : c'est un **conflit**.

Alex a écrit « Bonjour et bienvenue » sur `main`, et Sam « Bienvenue sur notre site » sur sa branche. Au moment de fusionner, Git s'arrête.

Terminal

```
$ git merge feature/titre
Auto-merging index.php
CONFLICT (content): Merge conflict in index.php
Automatic merge failed; fix conflicts and then commit the result.
```

Ouvrez le fichier : Git y a indiqué les deux versions.

index.php

```
<<<<<<< HEAD                    début : la version de votre branche actuelle
<h1>Bonjour et bienvenue</h1>
=======                         la séparation
<h1>Bienvenue sur notre site</h1>
>>>>>>> feature/titre           fin : la version de la branche fusionnée
```

À vous de trancher. Écrivez la version finale, supprimez les trois lignes de repères, puis signalez à Git que c'est réglé. Si vous préférez parler d'abord avec votre collègue, `git merge --abort` annule la fusion et remet tout comme avant.

Terminal

```
$ git add index.php
$ git commit -m "Fusion du nouveau titre"
```

Vous pouvez ensuite admirer le résultat.

Terminal

```
$ git log --oneline --graph --all
*   740d88a Fusion du nouveau titre
|\
| * dc5482f Nouveau titre d'accueil
* | 91df140 Titre plus chaleureux
|/
* c2d3631 Ajout du formulaire de connexion
```

Conseil de pro

Un conflit fait peur la première fois, mais il signifie seulement que Git a besoin de vous pour arbitrer. Pour en avoir moins, gardez des branches courtes et récupérez souvent le travail des autres.

Expérimenter est rassurant, à condition de pouvoir faire demi-tour. Voyons comment corriger une erreur, selon le moment où on s'en aperçoit.

Chapitre 7

Revenir en arrière sans casse

Tout le monde se trompe, et en développement plus souvent qu'ailleurs. Avec Git, la bonne commande dépend d'une seule question : **à quel stade en est votre erreur ?**

Votre situation	La commande	Ce qui se passe
Vous voulez abandonner une modification pas encore enregistrée	`git restore index.php`	Le fichier revient à son dernier état connu
Vous avez fait `git add` par erreur	`git restore --staged index.php`	Le fichier sort du carton, vos modifications restent
Un commit déjà partagé pose problème	`git revert 7538dad`	Un nouveau commit annule les effets de l'ancien

La dernière ligne fonctionne comme en comptabilité : on ne rature pas une écriture passée, on en passe une nouvelle qui la compense. L'historique reste honnête, et vos collègues qui ont déjà récupéré l'ancien commit ne sont pas perturbés.

Point de vigilance

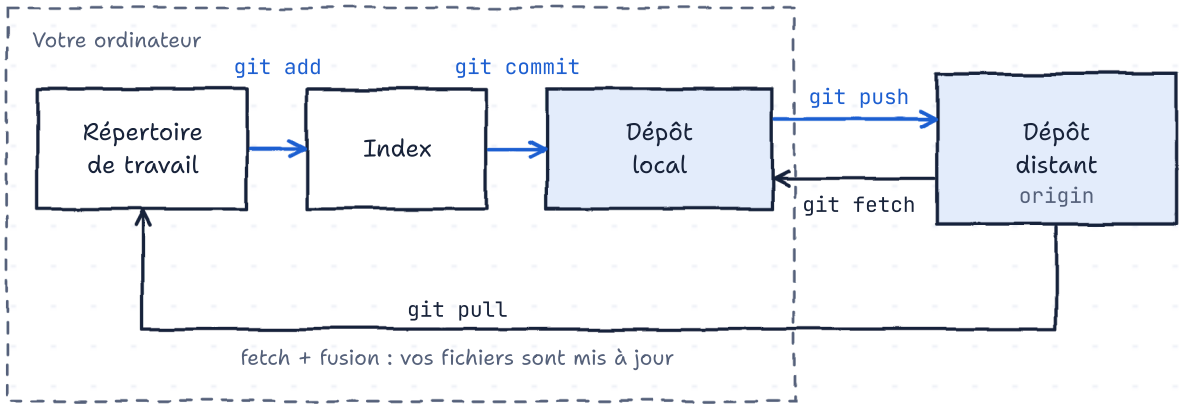
`git restore` efface pour de bon les modifications non enregistrées : Git ne peut pas retrouver ce qu'il n'a jamais reçu. Vous croiserez aussi sur Internet `git reset --hard` et `git push --force`. Ces commandes réécrivent ou effacent de l'historique. Évitez-les sur une branche partagée tant que vous ne maîtrisez pas leurs effets.

Jusqu'ici, tout s'est passé sur votre machine. Il est temps d'en sortir et de retrouver le reste de l'équipe.

Chapitre 8

Travailler à plusieurs

Alex et Sam ont chacun un dépôt complet sur leur poste. Pour échanger leur travail, ils ont besoin d'un point de rendez-vous. Ce sera un dépôt placé sur un serveur, auquel tout le monde a accès. On l'appelle **dépôt distant** (*remote*), et son nom habituel est `origin`.



En bleu, ce que vous envoyez. En noir, ce que vous recevez. Remarquez que `git fetch` s'arrête au dépôt local : vos fichiers ne bougent pas.

Les commandes de synchronisation

Pour récupérer un projet existant, on le clone. Vous obtenez un dépôt complet, déjà relié à son serveur d'origine.

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

Si au contraire votre projet existe déjà en local, comme notre site, il faut le relier au dépôt distant. L'option `-u` mémorise cette association.

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

Une distinction revient souvent en entretien, celle entre `fetch` et `pull`. Imaginez votre boîte aux lettres. `git fetch` consiste à relever le courrier et à le poser sur la table sans l'ouvrir. `git pull` relève le courrier et le classe directement dans vos dossiers.

Terminal

```
$ git fetch origin
$ git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.

$ git pull origin main
```

Point de vigilance

Si un collègue a envoyé son travail juste avant vous, Git refusera votre `git push`. C'est une protection : il vous évite d'écraser ce travail. Faites un `git pull`, réglez un éventuel conflit, puis renvoyez vos commits.

Une journée type

Alex doit créer la page de connexion, Sam la gestion des utilisateurs. Chacun part de la dernière version commune.

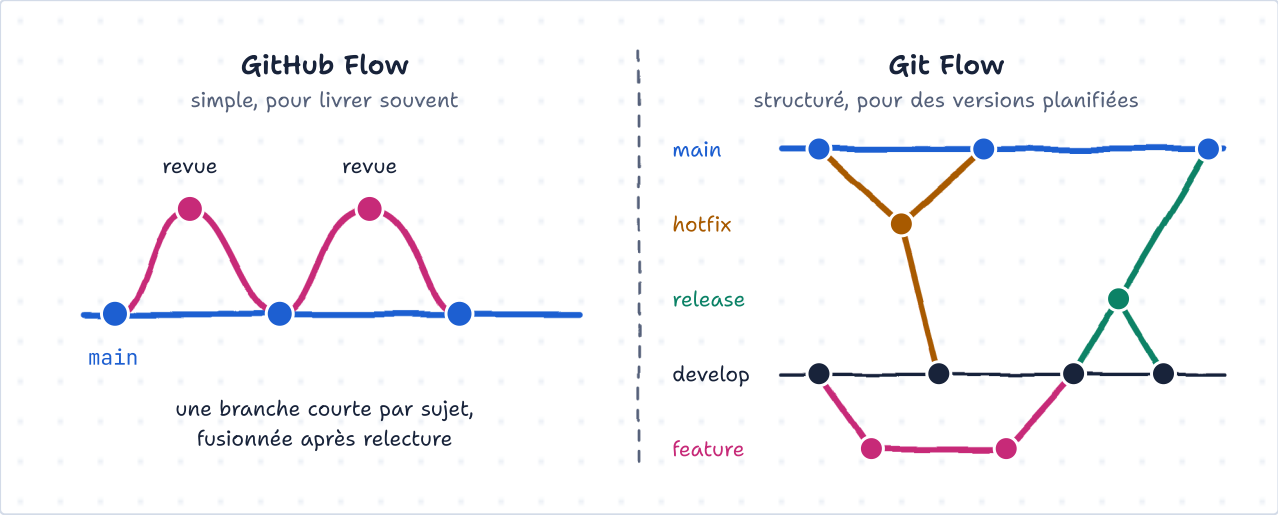
Terminal d'Alex

```
$ git switch main
$ git pull origin main
$ git switch -c feature/connexion
# travail, commits...
$ git push -u origin feature/connexion
```

Sam fait exactement la même chose avec `feature/utilisateurs`. Personne ne touche directement à `main`, qui reste stable et prête à être livrée. Une fois le travail relu, chaque branche y est fusionnée.

S'organiser en équipe

Plus une équipe grandit, plus il faut se mettre d'accord sur la façon d'utiliser les branches. Deux organisations sont particulièrement connues.



Avec **GitHub Flow**, il y a une branche principale stable et une branche temporaire par sujet. C'est adapté à une application web mise en ligne plusieurs fois par semaine. **Git Flow** ajoute une branche d'intégration (`develop`), des branches de préparation de version (`release/*`) et de correction urgente (`hotfix/*`). C'est plus lourd, mais utile lorsqu'on maintient des versions numérotées, comme pour un logiciel installé chez des clients.

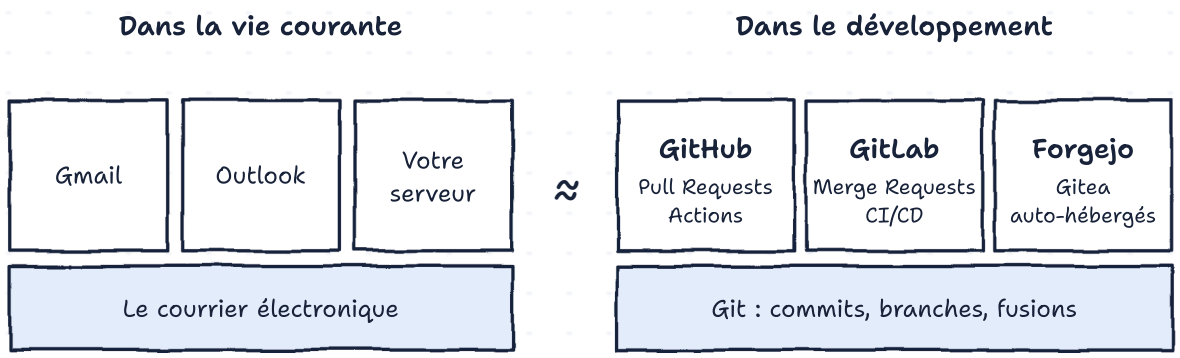
> **Conseil de pro**
>
> En arrivant dans une équipe, ne cherchez pas à savoir quelle méthode est la meilleure. Demandez laquelle est utilisée, et respectez-la. Une convention appliquée par tous vaut mieux que la méthode idéale appliquée à moitié.

Pour que ce rendez-vous existe, il faut un endroit où héberger le dépôt. C'est le métier de GitHub et de GitLab, qui en profitent pour apporter bien d'autres services.

Chapitre 9

GitHub, GitLab et l'automatisation

Commençons par dissiper la confusion la plus répandue chez les débutants : Git et GitHub sont deux choses différentes. Une comparaison avec le courrier électronique suffit à s'en souvenir.



On peut envoyer des e-mails sans Gmail, comme on peut utiliser Git sans GitHub. Les plateformes ajoutent des services par-dessus.

La fonction qui change tout : la demande de fusion

Dans une équipe, on ne fusionne pas son travail dans `main` sans qu'un collègue l'ait relu. Les plateformes organisent ce moment. Alex pousse sa branche et ouvre une demande de fusion. Sam voit les lignes modifiées, pose des questions, suggère des corrections. Les tests se lancent automatiquement. Quand tout est validé, un clic suffit pour fusionner.

GitHub appelle cela une **Pull Request**, GitLab une **Merge Request**. C'est la même idée. Cette relecture, appelée revue de code, est aussi l'un des meilleurs moyens de progresser quand on débute.

Choisir une plateforme

GitHub, qui appartient à Microsoft depuis 2018, est le point de ralliement de l'open source et un lieu naturel pour montrer ses projets à un recruteur. GitLab mise sur une plateforme unique qui couvre tout le cycle de vie d'une application, dans une logique DevOps, c'est-à-dire de rapprochement entre développement et exploitation. Bitbucket, d'Atlassian, reste présent dans certaines entreprises.

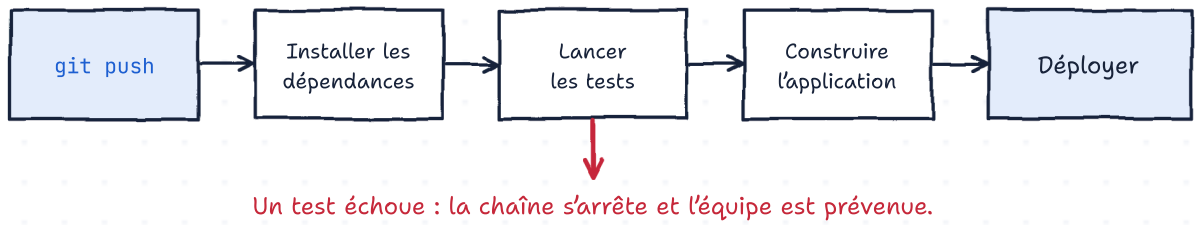
	GitHub	GitLab
Demande de fusion	Pull Request	Merge Request
Automatisation	GitHub Actions	GitLab CI/CD
Point fort	Communauté, écosystème	Plateforme DevOps intégrée
Sur ses propres serveurs	GitHub Enterprise Server	GitLab Self-Managed

Les deux couvrent les besoins de la majorité des équipes. Le choix se fait surtout selon les habitudes de l'équipe, le budget et les contraintes d'hébergement.

Automatiser ce qui est répétitif

Après chaque fusion, il faut installer les dépendances, lancer les tests, construire l'application et la déployer. Fait à la main, c'est long, et un jour quelqu'un oublie les tests un vendredi soir. On confie donc cette chaîne à une machine.

C'est ce qu'on appelle la **CI/CD**, pour intégration continue et déploiement continu (*Continuous Integration / Continuous Deployment*).



La machine n'oublie aucune étape, et un code qui ne passe pas les tests ne part pas en production. La protection vaut ce que valent vos tests.

Sur GitHub, on décrit cette chaîne dans un fichier YAML placé dans le dossier `.github/workflows/`. Ici, l'exemple correspond à une application Laravel.

.github/workflows/deploiement.yml

```yaml
name: Tests et déploiement

# Se déclenche à chaque envoi sur main
on:
  push:
    branches:
      - main

# Le workflow ne reçoit que le droit de lire le code
permissions:
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Récupération du code
        uses: actions/checkout@v7

      - name: Installation des dépendances
        run: composer install --no-interaction

      - name: Tests
        run: php artisan test

      - name: Déploiement
        run: echo "À remplacer par votre commande de déploiement"
```

Le fichier se lit presque comme une phrase : à chaque envoi sur `main`, sur une machine Ubuntu, exécute ces étapes dans l'ordre. Si l'une échoue, les suivantes ne sont pas lancées. La dernière étape est à adapter : commande SSH, publication d'une image Docker, etc.

Confier son code à une plateforme est très pratique. Mais que devient-il une fois en ligne, et comment un service gratuit gagne-t-il sa vie ?

Chapitre 10

Confidentialité et sécurité

Pourquoi est-ce gratuit ?

GitHub fonctionne sur un modèle dit *freemium* : une offre gratuite généreuse pour que tout le monde adopte la plateforme, puis des services payants pour ceux qui en ont besoin. Les entreprises paient pour la gestion des équipes, la sécurité avancée, l'assistant Copilot, ou les environnements de développement en ligne. Les quotas évoluent régulièrement, alors consultez la page des tarifs au moment de faire un choix.

Un dépôt privé est-il vraiment privé ?

Un dépôt privé n'est accessible qu'à vous et aux personnes que vous y invitez, et GitHub s'engage à en préserver la confidentialité. Pour autant, le risque n'est jamais nul, comme pour tout service en ligne. Une faille, une mauvaise configuration ou une application tierce trop gourmande en permissions peuvent exposer votre code. GitHub est aussi une société américaine, soumise à des obligations légales.

Ne traitez donc pas un dépôt privé comme un coffre-fort : votre code reste hébergé chez un prestataire.

Et l'intelligence artificielle ?

La question revient souvent, et il faut distinguer deux situations.

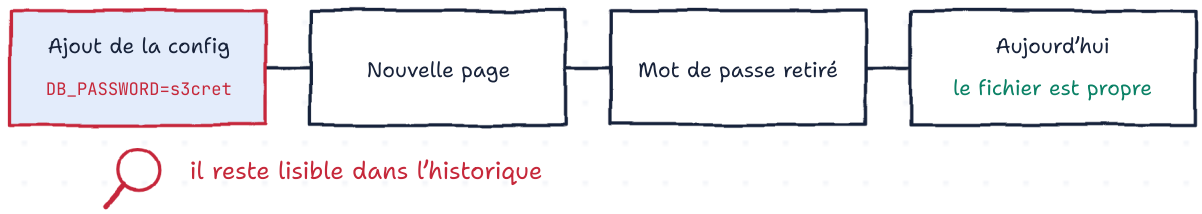
Le code qui dort dans vos dépôts privés n'est pas utilisé pour entraîner les modèles de GitHub. En revanche, depuis le 24 avril 2026, les **échanges avec Copilot** des utilisateurs des offres Free, Pro et Pro+ peuvent servir à cet entraînement, sauf opposition de leur part. Cela comprend vos demandes, les réponses, les extraits de code et leur contexte, y compris lorsque vous travaillez sur un dépôt privé. Les offres Business et Enterprise ne sont pas concernées.

Conseil de pro

Sur le projet d'un client, vérifiez ce point avant d'activer Copilot. L'option se trouve dans vos paramètres Copilot, section *Privacy*. Le détail est dans l'annonce officielle de GitHub.

L'erreur classique : le mot de passe dans l'historique

Vous enregistrez par inadvertance un fichier contenant un mot de passe. Vous vous en rendez compte, vous retirez la ligne, vous faites un nouveau commit. Problème réglé ? Pas du tout, et vous savez maintenant pourquoi.



Le commit de départ existe toujours. Toute personne qui clone le dépôt peut remonter jusqu'à lui.

Le bon réflexe : considérez le mot de passe comme compromis et **changez-le immédiatement**. Nettoyer l'historique est possible, mais c'est long, et rien ne garantit que personne ne l'ait déjà récupéré.

Quelques habitudes vous éviteront ce genre de mésaventure :

- Gardez les fichiers `.env` et les identifiants hors du dépôt, grâce au `.gitignore`.
- Stockez les secrets de vos automatisations dans les GitHub Secrets ou leur équivalent GitLab.
- Limitez les droits des workflows et vérifiez les actions et dépendances tierces.
- Protégez `main` pour imposer une relecture avant toute fusion.
- Sauvegardez les dépôts importants ailleurs que sur la plateforme.

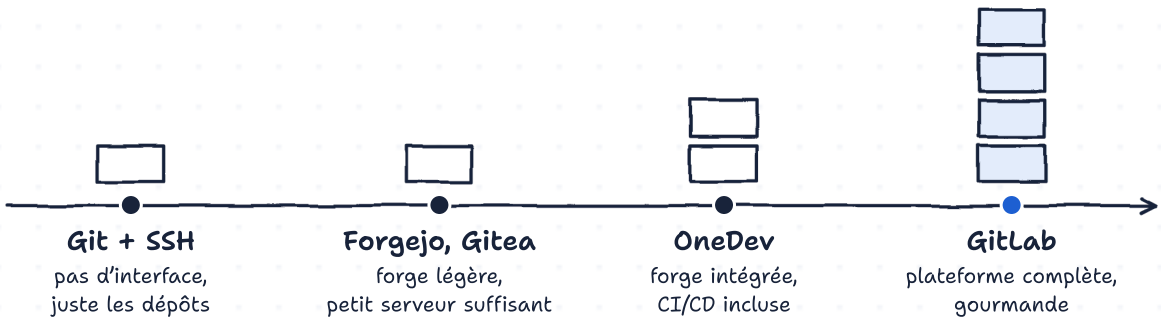
Si ces conditions ne conviennent pas à votre projet, rien ne vous oblige à passer par une grande plateforme. Git étant distribué, n'importe quel serveur peut jouer le rôle de dépôt de référence.

Chapitre 11

Héberger ses dépôts soi-même

Un client du secteur de la santé exige que son code reste sur ses propres serveurs. Une petite structure veut ne dépendre d'aucune plateforme commerciale. Dans ces situations, on héberge ses dépôts soi-même.

Plusieurs solutions existent, de la plus minimaliste à la plus complète. Celles qui ajoutent une interface web, des tickets et des demandes de fusion portent le nom de **forges logicielles**.



Les petits serveurs empilés donnent une idée des ressources à prévoir et de l'administration à assurer.

Git seul, avec SSH

C'est l'option la plus simple. Il suffit d'un serveur Linux accessible en SSH, sur lequel on crée un dépôt dit « nu » (*bare*) : il contient l'historique, sans copie de travail, puisque personne ne développe directement sur le serveur.

Sur le serveur

```
$ sudo adduser git
$ sudo mkdir -p /srv/git/mon-site.git
$ sudo git init --bare -b main /srv/git/mon-site.git
$ sudo chown -R git:git /srv/git/mon-site.git
```

Ajoutez ensuite votre clé SSH publique au compte `git` du serveur, puis reliez votre projet.

Sur votre poste

```
$ git remote add origin git@mon-serveur:/srv/git/mon-site.git
$ git push -u origin main
```

Cela fonctionne très bien pour un développeur seul ou une sauvegarde. Il n'y a en revanche ni interface web, ni tickets, ni demandes de fusion.

Une forge légère : Forgejo ou Gitea

Forgejo et Gitea offrent l'essentiel d'une plateforme : dépôts, utilisateurs, tickets, Pull Requests, wiki, et même un système d'automatisation proche de GitHub Actions. Forgejo est né en 2022 d'un fork de Gitea, pour garder le projet sous gouvernance communautaire. Tous deux s'installent facilement avec Docker et tournent sur un petit serveur, ce qui en fait d'excellents choix pour un indépendant ou une petite équipe.

Les plateformes complètes : OneDev et GitLab

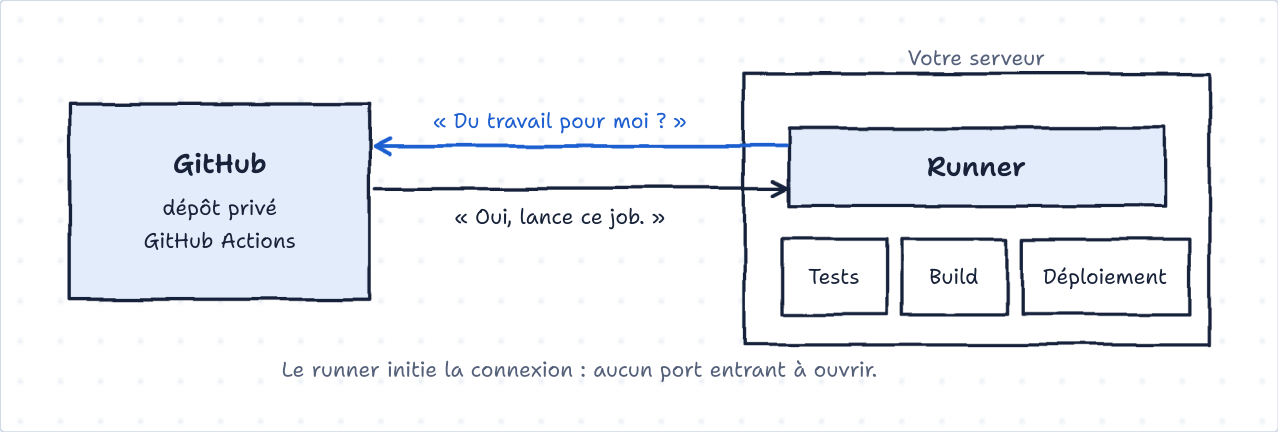
OneDev regroupe hébergement, tickets et pipelines dans un seul outil. GitLab Self-Managed apporte tout l'environnement DevOps de GitLab sur vos serveurs, avec des besoins en mémoire et en administration nettement supérieurs.

Point de vigilance

Héberger soi-même, c'est aussi devenir responsable des mises à jour, des sauvegardes et de la sécurité. Si personne n'a le temps de s'en occuper, un service géré est souvent le choix le plus prudent.

Le compromis : garder GitHub, utiliser ses machines

Il existe une solution intermédiaire : laisser ses dépôts sur GitHub, mais exécuter les tâches automatiques sur son propre serveur, par exemple un VPS. On y installe un petit agent appelé **runner** (*self-hosted runner*) qui exécute les tâches à la place des machines de GitHub.



Vous conservez l'écosystème GitHub tout en profitant de la puissance de votre serveur. Ces minutes ne consomment pas votre quota GitHub Actions. Vérifiez tout de même les tarifs : fin 2025, GitHub a annoncé vouloir les facturer, avant de reporter cette décision.

Point de vigilance

Un runner exécute du code sur votre machine. Un workflow malveillant peut donc compromettre le serveur. GitHub recommande de les réserver aux dépôts privés : ne les branchez jamais sur un dépôt public où n'importe qui peut proposer du code. La documentation officielle détaille les précautions à prendre.

Vous avez fait le tour. Rassemblons maintenant l'essentiel, puis passons à ce qui compte vraiment : la pratique.

Chapitre 12

L’essentiel à retenir

Si vous ne deviez garder qu’une phrase de ce cours :

Git est le système de gestion de versions. GitHub et GitLab sont des services construits autour de lui.

Et un **serveur Git** est simplement une machine qui héberge des dépôts pour que l’équipe s’y synchronise. Cette machine peut être chez GitHub, chez GitLab, sur une forge auto-hébergée, ou sur un simple serveur Linux.

Votre aide-mémoire

Commande	À quoi elle sert
Démarrer	
`git init`	Créer un dépôt dans le dossier courant
`git clone`	Copier un dépôt distant sur son poste
Enregistrer	
`git status`	Savoir où on en est
`git diff`	Voir les modifications
`git add`	Préparer des modifications pour le prochain commit
`git commit`	Enregistrer une étape
`git log`	Relire l'historique
Brancher	
`git branch`	Lister ou créer des branches
`git switch`	Changer de branche
`git merge`	Fusionner une branche dans la branche courante
Partager	
`git fetch`	Récupérer les nouveautés sans toucher à ses fichiers
`git pull`	Récupérer et intégrer les nouveautés
`git push`	Envoyer ses commits sur le dépôt distant
Corriger	
`git restore`	Abandonner une modification non enregistrée
`git revert`	Annuler un commit avec un nouveau commit

Le vocabulaire de l'équipe

Le mot	Ce qu'il veut dire
Repository (dépôt)	Le projet et tout son historique
Commit	Une étape enregistrée du projet
Staging area (index)	La sélection du prochain commit
Branch (branche)	Une ligne de travail indépendante
Merge (fusion)	La réunion de deux lignes de travail
Conflict (conflit)	Deux modifications incompatibles, à arbitrer
Remote (dépôt distant)	Le dépôt de référence, souvent appelé `origin`
Pull Request / Merge Request	Une demande de fusion, relue par l'équipe
CI/CD	L'automatisation des tests et du déploiement

À vous de jouer

Git s'apprend avec les doigts. Prenez une heure, un petit projet sans enjeu, et enchaînez ces étapes.

1. Créez un dépôt local et ajoutez un `.gitignore`.
2. Enregistrez au moins trois commits avec des messages soignés.
3. Créez une branche et ajoutez-y un fichier. Revenez sur `main` pour le voir disparaître.
4. Modifiez la même ligne d'un même fichier sur deux branches.
5. Fusionnez pour provoquer un conflit, puis résolvez-le.
6. Publiez le dépôt sur GitHub ou GitLab.
7. Depuis une nouvelle branche, ouvrez une demande de fusion et validez-la.

Vous saurez que c'est réussi quand `git log --oneline --graph --all` vous racontera une histoire que vous comprenez du premier coup d'œil.

Pour aller plus loin

Une fois ces bases acquises, vous entendrez parler de `git stash` pour mettre un travail de côté, de `git rebase` pour réorganiser une branche, ou des *tags* pour marquer une version livrée. Chacune de ces commandes trouvera naturellement sa place dans ce que vous venez d'apprendre.

- Le livre *Pro Git*, en français : la référence, gratuite.
- La documentation GitHub et celle de GitLab.
- La documentation Forgejo et celle de Gitea.
- Les 20 ans de Git, racontés par Linus Torvalds, en anglais.