Les bases de Git : dépôts, index, branches et fusions

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-18

Table des matières

1 Objectifs pédagogiques 1
2 Pourquoi un gestionnaire de versions distribué 2
  2.1 Centralisé ou distribué : ce que le modèle change au mécanisme 2
  2.2 Trois questions que Git doit résoudre 2
3 Genèse et positionnement de Git 3
4 Premier contact dans l'environnement du cours 3
  4.1 Identité de l'auteur des validations .. 3
  4.2 Créer le premier dépôt 4
5 Dépôt local, dépôt distant, dépôt nu 4
6 Indexer et commiter : trois zones et un objet 5
  6.1 Le triptyque arbre de travail, index, HEAD 5
  6.2 Observer l'index, du fichier au commit 5
  6.3 Le commit est un objet nommé 6
  6.4 Le réflexe de synchronisation 7
7 Consulter l'histoire 7
  7.1 Les tags 7
8 Exclure des fichiers du suivi 8
9 Comparer les états 9
10 Branches et fusions 9
  10.1 Une branche est un fichier de référence qu'on lit 9
  10.2 Créer et changer de branche 10
  10.3 Le message de commit : une convention nommée 11
  10.4 Fusion : fast-forward ou commit explicite 11
11 Conflits de fusion 12
12 Travailler à plusieurs : dépôts distants . 14
  12.1 Le vocabulaire des échanges 14
  12.2 Démonstration : le push refusé ... 14
  12.3 Le dépôt d'exemple du cours 16
  12.4 Trois utilitaires hors chemin 19
13 Ce que vous retiendrez 19

1 Objectifs pédagogiques

À l'issue de cette leçon, l'étudiant est capable de :

- expliquer ce qu'un gestionnaire de versions distribué résout que le courrier électronique, les services de synchronisation ou l'envoi de correctifs ne résolvent pas ;
- distinguer dépôt local, dépôt distant et dépôt nu, et créer les trois ;
- décrire un commit comme un objet nommé identifiant un instantané du projet, et lire les objets qui le composent ;
- indexer et valider des modifications, consulter l'historique et comparer les états des trois zones ;
- créer des branches, fusionner en *fast-forward* ou avec un commit de fusion explicite, et résoudre un conflit dont les marqueurs ont été observés dans la sortie d'une commande ;

- synchroniser un dépôt local avec un dépôt distant (clone, push, fetch, pull) et expliquer un refus de *push non fast-forward*.

Pré-requis : usage du terminal et du système de fichiers. Git est disponible dans l'image du cours ; son installation sur une machine personnelle est hors périmètre de cette leçon (voir l'annexe de prérequis de la séquence d'environnement).

2 Pourquoi un gestionnaire de versions distribué

Avant les outils de gestion de versions, un projet se diffusait par trois moyens, chacun porteur d'un mode de défaillance précis.

- **S'envoyer le code par courrier électronique** : chaque envoi multiplie les copies sans moyen de savoir laquelle est la plus récente ; deux destinataires ayant modifié leur copie en parallèle produisent deux versions divergentes, et la perte de l'une des deux est silencieuse.
- **Synchroniser des fichiers via un service en ligne** (Dropbox, Google Drive, OneDrive) : l'ensemble est partagé, mais l'outil n'enregistre ni l'intention des modifications, ni un historique des états du projet ; deux écritures simultanées sur un même fichier produisent une copie de type « conflicted copy », c'est-à-dire un doublé à réconcilier à la main, et non une fusion explicite.
- **S'envoyer des correctifs** (*patches*) : `diff` produit la description textuelle des écarts entre deux versions, et `patch` rejoue ces écarts sur la copie du destinataire. La technique suppose que l'émetteur possède les deux versions et que l'historique vive dans une boîte de réception.

Git répond à ces trois défaillances par une hypothèse inverse : **chaque participant conserve une copie complète de l'historique du projet**, et non seulement son dernier état.

2.1 Centralisé ou distribué : ce que le modèle change au mécanisme

Dans un système centralisé (Subversion, par exemple), l'historique réside sur un serveur unique : sans réseau, pas de validation possible, pas de consultation des états anciens. Dans un système distribué, l'opération de clonage transfère tout l'historique ; indexer, commiter, créer des branches et consulter s'exécutent ensuite localement, sans appel réseau. La synchronisation (`push`, `pull`) devient une étape explicite, différée, entre égaux.

Un critère pratique sépare les deux familles : une session de travail sans réseau. Si toute opération de gestion de versions exige une connexion, le système est centralisé. Git se range du côté des systèmes où les opérations courantes restent possibles hors ligne — critère que la suite de la leçon vérifie en manipulant des dépôts purement locaux.

2.2 Trois questions que Git doit résoudre

- Comment revenir à un état antérieur du projet et le reconstituer à l'identique ?
- Comment travailler à plusieurs sur un même ensemble de fichiers sans s'écraser mutuellement ?
- Comment développer une variante expérimentale, puis l'intégrer ou l'abandonner sans laisser de trace dans la ligne principale ?

Les sections qui suivent répondent à ces questions en étudiant les objets que Git manipule, plutôt qu'en décrivant son interface.

3 Genèse et positionnement de Git

Les outils de gestion de versions se sont succédé en résolvant la limite du précédent. La page de manuel `gitglossary(7)`, glossaire officiel du projet Git, distingue les familles suivantes :

1. **Contrôle de révision par fichier** (SCCS, RCS) : chaque fichier possède son propre historique ; la notion d'état cohérent pour l'ensemble du projet n'existe pas.
2. **Contrôle de révision de projet, centralisé** (CVS, Subversion) : l'arborescence entière est validée d'un seul mouvement, mais l'historique reste détenu par un serveur unique.
3. **Contrôle de révision distribué** (Git, Mercurial) : chaque copie de travail est aussi une copie complète de l'historique, échangeable de pair à pair.

Git appartient à la troisième famille. Sa création en 2005 par Linus Torvalds, pour le développement du noyau Linux, est documentée par l'ouvrage de référence *Pro Git* (2e édition, chapitre 1) ; le traitement développé de cette genèse est hors galerie (destination A1).

Les commandes présentées dans cette leçon sont définies par les pages `git(1)` et `gitglossary(7)` installées avec Git ; `man gitglossary` les consulte hors connexion — ce qui n'a rien d'étonnant de la part d'un outil conçu pour le travail local.

4 Premier contact dans l'environnement du cours

L'environnement du cours fournit Git ; rien n'est à installer. Le premier geste utile est donc un geste de configuration, puis la création d'un dépôt.

4.1 Identité de l'auteur des validations

Chaque commit enregistre un auteur et un commiteur. L'identité est posée une fois pour toutes ; la commande `git config --global` écrit dans un fichier, `~/.gitconfig`, que nous lisons immédiatement après l'avoir modifiée : la configuration n'est pas un état magique, c'est du contenu de fichier. On fixe aussi le nom de la branche initiale à `main`, pour que les manipulations ultérieures (`cat .git/refs/heads/main`) soient déterministes quelle que soit la configuration par défaut de la machine.

```
git config --global user.name "Atelier Git"
git config --global user.email "atelier.git@example.invalid"
git config --global init.defaultBranch main
cat ~/.gitconfig
```

```
[filter "lfs"]
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
    process = git-lfs filter-process
    required = true
[user]
    name = Atelier Git
    email = atelier.git@example.invalid
[init]
    defaultBranch = main
```

L'attribut `--global` fixe la portée : tous les dépôts de l'utilisateur. Les portées `--local` (dépôt courant) et `--system` existent ; en cas de désaccord, la valeur du dépôt l'emporte sur la valeur globale, qui l'emporte sur la valeur système.

4.2 Créer le premier dépôt

Les exercices de la leçon s'exécutent dans `/tmp/git-tp`. La première cellule de solution définit `PROJECT_DIR` et repart d'un répertoire vide ; toutes les cellules ultérieures supposent les variables `PROJECT_DIR` et `SERVEUR` définies ci-dessous : en cas de réexécution isolée, réexécuter d'abord la cellule qui les définit.

> ⚠ Exercice : initialiser un dépôt local
>
> Créez un répertoire de travail temporaire, initialisez-y un dépôt Git et observez ce que la commande a créé.

Solution

```
PROJECT_DIR=/tmp/git-tp
```

```
cd ${PROJECT_DIR}
git init
```

```
Initialized empty Git repository in /
tmp/git-tp/.git/
```

```
ls -a ${PROJECT_DIR}
ls ${PROJECT_DIR}/.git
```

```
.  ..  .git
HEAD  config  description  hooks  info
objects  refs
```

`git init` ne modifie aucun fichier de travail : il crée le répertoire caché `.git`, qui contient toutes les données du dépôt. Supprimer `.git` détruit le dépôt sans toucher aux fichiers ; supprimer les fichiers en conservant `.git` détruit la copie de travail sans toucher à l'historique. Les deux opérations ne sont pas symétriques, et la distinction entre le dépôt et sa copie de travail s'enracine là.

5 Dépôt local, dépôt distant, dépôt nu

Trois mots désignent trois rôles que Git distingue techniquement.

- **Dépôt local** : le dépôt dans lequel vous travaillez, ici sous `${PROJECT_DIR}`.
- **Dépôt distant** : un dépôt accessible par URL (réseau ou chemin local), jouant le rôle de point d'échange. Une forge logicielle (GitHub, GitLab) en héberge ; le terme « distant » désigne le rôle, pas la situation géographique.
- **Dépôt nu** (*bare*) : un dépôt **sans arbre de travail**.

Le dépôt nu est la forme canonique d'un dépôt distant : il reçoit des `push` et sert des `clone`, et ne s'édite jamais. Nous en créons un dès maintenant, **hors** du répertoire de travail de l'atelier, sous `/tmp/git-tp-serveur.git` ; il servira à la démonstration de collaboration de l'avant-dernière section.

```
SERVEUR=/tmp/git-tp-serveur.git
rm -rf ${SERVEUR}
git init --bare ${SERVEUR}
ls ${SERVEUR}
```

```
Initialized empty Git repository in /
tmp/git-tp-serveur.git/
HEAD  config  description  hooks  info
objects  refs
```

Le listing montre l'intérieur d'un `.git` érigé en répertoire racine (`HEAD`, `objects`, `refs`…) et **aucun fichier de travail** : c'est la signature d'un dépôt nu.

6 Indexer et commiter : trois zones et un objet

6.1 Le triptyque arbre de travail, index, HEAD

ℹ Définition — les trois zones de Git

Git déplace l'état du projet entre trois zones :

- l'**arbre de travail** (*working tree*) : les fichiers sur le disque, là où l'on édite ;
- l'**index** (ou *staging area*) : la préparation du prochain commit — la liste des contenus que Git s'apprête à enregistrer ;
- **HEAD** : la référence qui désigne le commit courant ; sur une branche, HEAD pointe vers la branche, qui pointe vers un commit.

Indexer (`git add`) copie le contenu de l'arbre de travail dans l'index. Valider (`git commit`) enregistre le contenu de l'index comme objet de l'historique.

```
arbre de travail  --add-->     index
--commit-->     commit  (désigné  par
HEAD)
 fichiers sur disque         contenu
prêt à    objet enregistré
                                être
validé       dans .git
```

`git status` répond exactement à la question de ce modèle : pour chaque fichier, lesquelles de ces trois zones divergent entre elles.

6.2 Observer l'index, du fichier au commit

⚠ Exercice : créer un README et observer les trois zones

Créez un fichier README, consultez l'état avant puis après l'indexation, et validez. Chaque sortie de `git status` se lit comme une affirmation sur les trois zones.

Solution

```
cd ${PROJECT_DIR}
echo "# Atelier Git" > README.md
git status
```

```
On branch main

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 (use "git add" to track)
```

```
git add README.md
git status
```

```
On branch main

No commits yet

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

```
git commit -m "docs: ajouter un README
d'atelier"
git status
```

```
[main (root-commit) 40fa857] docs:
ajouter un README d'atelier
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
On branch main
nothing to commit, working tree clean
```

La première sortie place `README.md` dans l'arbre de travail sans que rien d'autre n'existe (fichier non suivi) ; la seconde le montre dans l'index, en attente ; la troisième constate que les trois zones sont d'accord. Les variantes d'indexation : `git add <fichier>` (un fichier), `git add -u` (modifications et suppressions de fichiers déjà suivis, jamais les nouveautés), `git add .` (tout, sauf l'ignoré).

6.3 Le commit est un objet nommé

Un commit n'est pas une « sauvegarde » : Git emmagasine des **objets**, désignés par l'empreinte (*hash*) de leur propre contenu. Trois sortes d'objets suffisent à comprendre le mécanisme (définis dans `gitglossary(7)`) :

- un **blob** : le contenu d'un fichier, sans nom ni histoire ;
- un **arbre** (*tree*) : une liste de noms, de modes et de références vers des blobs ou d'autres arbres — un répertoire ;
- un **commit** : un arbre racine, la référence à son ou ses parents, un auteur, une date et un message.

Les commandes ci-dessous lisent ces objets dans le dépôt créé à l'instant :

```
git cat-file -t HEAD
git cat-file -p HEAD
```

```
commit
tree 84f5282931e173f518e325710f76dc67e2e22bdc
author          Atelier          Git
<atelier.git@example.invalid>
1792302521 +0000
committer        Atelier        Git
<atelier.git@example.invalid>
1792302521 +0000

docs: ajouter un README d'atelier
```

```
git cat-file -p "HEAD^{tree}"
```

```
100644 blob f9fcc874958f980719d4de1d5a3bc5f1aaf72415
README.md
```

La première cellule affiche le type de l'objet désigné par HEAD (`commit`), puis son contenu : l'arbre racine référencé, l'auteur, le message. La seconde déroule cet arbre : `README.md` y apparaît avec son blob.

Deux fichiers au contenu identique reçoivent la même empreinte, avant même d'être suivis :

```
printf   "contenu   identique\n"   >
jumeau-a.md
printf   "contenu   identique\n"   >
jumeau-b.md
git hash-object jumeau-a.md
git hash-object jumeau-b.md
rm jumeau-a.md jumeau-b.md
```

```
3d0c4c59c5928059c7cc1f20cc2ab63ead11dce5
3d0c4c59c5928059c7cc1f20cc2ab63ead11dce5
```

L'empreinte est calculée par SHA-1 par défaut (40 caractères hexadécimaux). Deux conséquences structurelles : un contenu identique est stocké une seule fois, et modifier un objet change son identifiant — l'historique est donc inviolable sans le trahir. Le support d'un for-

mat de dépôt en SHA-256 est traité en section des branches, avec sa source.

6.4 Le réflexe de synchronisation

Un historique distribué peut diverger : deux clones d'un même distant peuvent chacun contenir des commits que l'autre ignore. D'où le réflexe professionnel : **récupérer et intégrer les modifications du dépôt distant (`git pull`) avant de commencer une séance de travail**, et avant de pousser. Une branche locale qui n'a pas intégré l'avance du distant s'expose à un refus explicite, dit **non fast-forward** : Git rejette un `push` qui ferait disparaître des commits présents sur le distant. La section consacrée au travail à plusieurs produit ce refus et en observe la sortie.

7 Consulter l'histoire

Deux niveaux de lecture de l'historique suffisent en pratique :

- `git log` : l'histoire complète — auteur, date, message de chaque commit ;
- `git log --oneline --graph --decorate --all` : une vue compacte et graphique de toutes les branches.

```
git log --oneline --graph --decorate
--all
```

```
* 40fa857 (HEAD -> main) docs: ajouter
un README d'atelier
```

Pour un usage quotidien, l'alias `lg` ci-dessous produit une version colorée et condensée. Un alias de shell ne survit pas à la session ; pour le rendre permanent, on l'inscrit dans la section `[alias]` de `~/.gitconfig` — le fichier lu en section « Premier contact ».

```
alias      lg="git      log      --
graph       --abbrev-commit       --
decorate    --format=format:'%C(bold
blue)%h%C(reset)  -  %C(bold  green)
(%ar)%C(reset)
%C(white)%s%C(reset)  %C(dim  white)-
%an%C(reset)%C(auto)%d%C(reset)'  --
all"
lg
```

```
* 40fa857 - (1 second ago) docs:
ajouter un README d'atelier - Atelier
Git (HEAD -> main)
```

7.1 Les tags

> **Définition — tag léger, tag annoté**
>
> - **Tag léger** (`git tag v0.1`) : une simple référence pointant vers un commit, sans information supplémentaire — une étiquette mobile.
> - **Tag annoté** (`git tag -a v0.1 -m "message"`) : un véritable objet d'étiquette, qui enregistre auteur, date et message, et peut porter une signature.
>
> Pour marquer une version, préférer le tag annoté : lui seul documente qui a posé l'étiquette et quand.

```
git tag -a v0.1 -m "Premier jalon de
l'atelier"
git cat-file -t v0.1
git show v0.1 --no-patch
```

```
tag
tag v0.1
Tagger:          Atelier          Git
<atelier.git@example.invalid>
Date:   Sun Oct 18 05:48:42 2026 +0000
```

```
Premier jalon de l'atelier

commit 40fa85737d9ad0da5d78eef5b630cacbe3
(HEAD -> main, tag: v0.1)
Author:          Atelier          Git
<atelier.git@example.invalid>
Date:   Sun Oct 18 05:48:41 2026 +0000

    docs: ajouter un README d'atelier
```

La commande `cat-file -t` sur le tag annoté répond `tag` et non `commit` : l'étiquette est un objet à part entière, ce qui distingue le tag annoté du tag léger — vérification directe du modèle d'objets posé en section précédente.

8 Exclure des fichiers du suivi

Tout ce que contient l'arbre de travail n'a pas vocation à entrer dans l'historique : artefacts de compilation, caches, préférences locales. Le fichier `.gitignore` les exclut par motifs :

- `*.tmp` : tout fichier se terminant par `.tmp` ;
- `build/` : le répertoire `build` et tout son contenu ;
- `!important.tmp` : négation, qui réintègre ce qu'une règle précédente excluait.

Deux propriétés des motifs valent d'être énoncées, car elles surprennent. Un fichier ignoré n'est proposé ni par `git status` ni par `git add .`. Mais l'ignorance ne s'applique qu'aux fichiers **non suivis** : un fichier déjà passé dans l'index reste suivi même si une règle le couvre ; il faut le retirer explicitement (`git rm --cached <fichier>`) pour que la règle prenne effet.

Vérification : `git status --ignored`

Le drapeau `--ignored` lève le voile : les fichiers exclus apparaissent dans une section « Ignored files » du statut. C'est le moyen fiable de vérifier qu'une règle exclut bien ce que l'on croit — et rien d'autre.

Exercice : ignorer un fichier de cache

Créez un fichier `cache.tmp`, constatez que Git le voit, excluez-le par une règle, puis vérifiez l'exclusion avec le drapeau de vérification.

Solution

```
cd ${PROJECT_DIR}
touch cache.tmp
git status --short
```

```
?? cache.tmp
```

```
printf "*.tmp\n" > .gitignore
git status --short
git status --ignored --short
```

```
?? .gitignore
?? .gitignore
!! cache.tmp
```

```
git add .gitignore
git commit -m "build: exclure les
fichiers temporaires du suivi"
```

```
[main d1fbe9d] build: exclure les
fichiers temporaires du suivi
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Remarquons le dernier `git status --short` : `.gitignore` y figure comme nouveauté, `cache.tmp` a disparu de la proposition d'indexation — la règle s'applique avant tout stage, ce qui est le comportement voulu.

9 Comparer les états

La commande `git diff` répond à une question du modèle à trois zones : **qu'est-ce qui diffère entre deux des trois zones ?** Selon les arguments, la paire comparée change :

- `git diff` : arbre de travail contre index — ce que `git add` ne capturerait pas encore ;
- `git diff --staged` : index contre HEAD — ce que le prochain commit contiendra ;
- `git diff <commit>` : arbre de travail contre un commit donné de l'historique.

```
printf "## Démarrage\n\nAprès le clone, lire ce fichier puis exécuter git status.\n" >> README.md
git diff
```

```
diff --git a/README.md b/README.md
index f9fcc87..cf176a2 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,4 @@
 # Atelier Git
+## Démarrage
+
+Après le clone, lire ce fichier puis exécuter git status.
```

```
git add README.md
git diff --staged
git commit -m "docs: ajouter une section démo du README"
git diff
```

```
diff --git a/README.md b/README.md
index f9fcc87..cf176a2 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,4 @@
 # Atelier Git
+## Démarrage
+
+Après le clone, lire ce fichier puis exécuter git status.
[main b98df9f] docs: ajouter une section démo du README
 1 file changed, 3 insertions(+)
```

Les sorties de `git diff` utilisent un format à deux colonnes : lignes précédées de `-` (présentes dans la source de comparaison, absentes de la cible), de `+` (le contraire). C'est le format du correctif évoqué en première section — Git en connaît le lecteur comme l'auteur, ce qui explique qu'on puisse appliquer un `git diff` produit par un autre dépôt.

10 Branches et fusions

10.1 Une branche est un fichier de référence qu'on lit

Une branche est un pointeur mobile vers un commit. Le modèle reste une affirmation tant qu'on n'a pas ouvert le dépôt ; or une référence (*ref*) est un fichier Git, contenant une empreinte.

```
cat .git/HEAD
cat .git/refs/heads/main
wc -c .git/refs/heads/main
```

```
ref: refs/heads/main
b98df9fd095e3b233ec9afe691116e1ac612d549
41 .git/refs/heads/main
```

Le fichier de référence contient l'empreinte du commit visé — 40 caractères hexadécimaux et un saut de ligne, soit **41 octets** sous SHA-1, comme le confirme le comptage ci-dessus. `HEAD`, lui, est une référence à une référence : son contenu est la ligne `ref: refs/heads/main`. Un HEAD détaché (`git switch --detach <commit>`) contient directement une empreinte : la branche cesse d'être un intermédiaire.

Le projet Git prend en charge un format de dépôt expérimental reposant sur SHA-256 depuis la version 2.29 (notes de version officielles du projet Git) ; l'empreinte y compte 64 caractères et le fichier de référence atteindrait 65 octets — taille à vérifier par mesure dans un dépôt configuré en `--object-format=sha256`, plutôt qu'à affirmer.

10.2 Créer et changer de branche

- `git branch <nom>` : crée une référence sans s'y déplacer.
- `git switch <nom>` : change de branche.
- `git switch -c <nom>` : crée puis s'y déplace.
- `git restore <fichier>` : restaure un fichier depuis l'index, ou depuis un commit précis (`git restore --source=<commit> <fichier>`).

La commande historique `git checkout` cumulait ces rôles dans une seule interface ambiguë ; elle reste répandue dans la documentation et dans les réflexes, mais n'est pas enseignée ici : `switch` et `restore` la décomposent en gestes explicites.

> ⚠ Exercice : une branche pour une variante
>
> Créez `feature/app`, ajoutez-y un fichier de notes et validez. Vérifiez sur le graphe que `main` n'a pas bougé pendant ce temps.

Solution

```
git switch -c feature/app
echo "Notes de conception de la variante." > notes.md
git add notes.md
git commit -m "docs: ajouter les notes de conception"
git log --oneline --graph --decorate --all
```

```
Switched to a new branch 'feature/app'
[feature/app c54ed2c] docs: ajouter les notes de conception
 1 file changed, 1 insertion(+)
 create mode 100644 notes.md
* c54ed2c (HEAD -> feature/app) docs: ajouter les notes de conception
* b98df9f (main) docs: ajouter une section démo du README
* d1fbe9d build: exclure les fichiers temporaires du suivi
* 40fa857 (tag: v0.1) docs: ajouter un README d'atelier
```

Le graphe montre la séparation : `feature/app` a avancé d'un commit, `main` est restée en place. La branche n'a rien copié — elle n'est qu'une seconde référence ; l'opération coûte un fichier de 41 octets.

10.3 Le message de commit : une convention nommée

> **Convention — Conventional Commits**
>
> Un message de commit se lit aussi par des outils. La spécification *Conventional Commits* (v1.0.0, conventionalcommits.org) fixe la forme `type(portée): description`, où `type` décrit la nature du changement (`feat`, `fix`, `docs`, `refactor`, `test`, `chore`) et `portée` la zone du projet touchée. Des outils dérivent de ces préfixes des journaux de versions et des vérifications automatiques.

Compléments indépendants de toute spécification :

- le sujet tient en une ligne courte, à l'impératif (« Ajouter… », pas « Ajout de… ») ;
- le corps, séparé par une ligne vide, explique le *pourquoi* et l'impact, vers 72 caractères par ligne ;
- commits atomiques : un changement conceptuel par commit ;
- références aux discussions du projet dans le corps (`Fixes #42`).

```
docs(readme): compléter la section démarrage

Le README se limitait au titre. La section démarrage énonce les
deux premiers gestes après le clone, pour qu'un nouvel arrivant
de l'atelier puisse travailler sans consigne orale.
```

10.4 Fusion : fast-forward ou commit explicite

`git merge <branche>` intègre une branche dans la branche courante. Deux issues se distinguent :

- **Fast-forward** : si la branche cible n'a pas bougé depuis la création de la source, aucun contenu à réconcilier n'existe — Git déplace simplement la référence cible vers le commit source. Aucun nouvel objet n'est créé.
- **Commit de fusion** (`git merge --no-ff`) : Git crée un commit à **deux parents**, qui enregistre l'opération elle-même et laisse visible, dans le graphe, que le travail s'est déroulé à l'écart.

```
git switch main
git merge --no-ff feature/app -m "merge: intégrer la branche feature/app"
git log --oneline --graph --decorate --all
```

```
Switched to branch 'main'
Merge made by the 'ort' strategy.
 notes.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 notes.md
*   1cda2e1 (HEAD -> main) merge: intégrer la branche feature/app
|\
| * c54ed2c (feature/app) docs: ajouter les notes de conception
|/
* b98df9f docs: ajouter une section démo du README
* d1fbe9d build: exclure les fichiers temporaires du suivi
* 40fa857 (tag: v0.1) docs: ajouter un README d'atelier
```

```
git cat-file -p HEAD
```

```
tree 14da5c35fee822b31cf05b62dd3cdcd3472f
parent b98df9fd095e3b233ec9afe691116e1ac
parent c54ed2c99e14fa7247b0be5e3e818d6a09
author          Atelier          Git
<atelier.git@example.invalid>
1792302524 +0000
committer        Atelier        Git
<atelier.git@example.invalid>
1792302524 +0000

merge: intégrer la branche feature/
app
```

La dernière cellule vérifie l'affirmation sur le modèle d'objets : le commit de fusion contient deux lignes `parent`, là où tous les commits précédents n'en contenaient qu'une. C'est le critère qui distingue une fusion explicite d'un fast-forward, indépendamment du dessin du graphe.

11 Conflits de fusion

Un conflit surgit quand deux branches ont modifié la **même région** du **même fichier** : Git sait détecter le cas, mais ne peut pas choisir à votre place. La situation se construit en quelques commandes — deux versions divergentes de la ligne de titre d'un fichier.

```
echo "Titre du projet" > titre.md
git add titre.md
git commit -m "docs: ajouter le
fichier titre.md"
git switch -c variante-titre
echo "Titre révisé (variante)" >
titre.md
git commit -am "docs: réviser le titre
dans la variante"
git switch main
echo "Titre révisé (principal)" >
titre.md
git commit -am "docs: réviser le titre
sur main"
```

```
[main a4c3eee] docs: ajouter le
fichier titre.md
 1 file changed, 1 insertion(+)
 create mode 100644 titre.md
Switched to a new branch 'variante-
titre'
[variante-titre decfbc2] docs:
réviser le titre dans la variante
 1 file changed, 1 insertion(+), 1
deletion(-)
Switched to branch 'main'
[main 941a58f] docs: réviser le titre
sur main
 1 file changed, 1 insertion(+), 1
deletion(-)
```

La fusion qui suit échoue volontairement : la cellule est marquée `error: true` pour que le document affiche la sortie de l'échec et poursuive le rendu.

```
git merge variante-titre
```

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

```
: 1
```

Git refuse de trancher et le dit. Reste à observer **où** il bloque, dans le statut puis dans le fichier lui-même :

```
git status
```

```
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   titre.md

no changes added to commit (use "git add" and/or "git commit -a")
```

```
cat titre.md
```

```
<<<<<<< HEAD
Titre révisé (principal)
=======
Titre révisé (variante)
>>>>>>> variante-titre
```

Le fichier contient désormais des **marqueurs de conflit** : `<<<<<<<` ouvre la version du commit courant (la branche cible), `=======` sépare les deux candidats, `>>>>>>>` ferme la version entrante. Ces lignes ne sont du contenu pour personne : une annotation ajoutée par Git dans le fichier, qui doit disparaître entièrement après arbitrage — un fichier validé avec ses marqueurs devient un texte aberrant pour le compilateur ou pour le lecteur du document.

Résoudre se décompose en trois gestes : écrire la version arbitrée, l'indexer (le `add` est la déclaration « conflit traité »), valider la fusion.

```
printf "Titre révisé, en combinant les deux contributions.\n" > titre.md
git add titre.md
git commit -m "merge: fusion de la variante de titre, conflit résolu manuellement"
git log --oneline --graph --decorate --all
```

```
[main cee44ce] merge: fusion de la variante de titre, conflit résolu manuellement
*    cee44ce (HEAD -> main) merge: fusion de la variante de titre, conflit résolu manuellement
|\
| * decfbc2 (variante-titre) docs: réviser le titre dans la variante
* | 941a58f docs: réviser le titre sur main
|/
* a4c3eee docs: ajouter le fichier titre.md
*  1cda2e1 merge: intégrer la branche feature/app
|\
|  *  c54ed2c  (feature/app)  docs: ajouter les notes de conception
|/
* b98df9f docs: ajouter une section démo du README
* d1fbe9d build: exclure les fichiers temporaires du suivi
* 40fa857 (tag: v0.1) docs: ajouter un README d'atelier
```

Remarque : la résolution peut s'opérer dans un éditeur de fusion graphique (Meld, KDiff3) ; le geste est identique — choisir parmi les candidats et indexer le résultat.

```
git branch -d variante-titre
```

```
Deleted branch variante-titre (was decfbc2).
```

La suppression est propre : la variante est entièrement fusionnée, `git branch -d` l'accepte ; sans fusion complète, il refuserait — un garde-fou, pas une formalité.

12 Travailler à plusieurs : dépôts distants

12.1 Le vocabulaire des échanges

- `git remote add <nom> <url>` / `git remote remove <nom>` / `git remote -v` : déclarer, retirer, lister les dépôts distants connus (par convention, le principal porte le nom `origin`).
- `git clone <url>` : créer une copie locale avec tout l'historique, en déclarant et configurant le distant `origin`.
- `git fetch` : récupérer les références distantes **sans toucher** à l'arbre de travail.
- `git pull` : `fetch` suivi d'une intégration — fusion par défaut, rebassage avec `--rebase`.
- `git push` : envoyer vers le distant les commits locaux.

La distinction `fetch` / `pull` se lit dans le modèle à trois zones : `fetch` ne met à jour que la vue locale du dépôt distant (les références `origin/…` stockées dans `.git`) ; rien ne change dans l'arbre de travail tant qu'on ne fusionne pas. `pull` enchaîne les deux gestes : c'est commode quand on anticipe peu, mais l'opération déclenche une fusion implicite, qui peut produire un commit de fusion non voulu. C'est pourquoi la suite du cours intègre presque toujours `fetch` puis inspecte avant de fusionner, et n'emploie `pull --rebase` que sur des commits non partagés.

12.2 Démonstration : le push refusé

Tout s'exécute ici contre le dépôt nu créé plus haut, qui joue le rôle de serveur ; deux clones, donc deux auteurs distincts. La démonstration produit le refus annoncé en section des indexations.

```
rm -rf /tmp/git-tp-a /tmp/git-tp-b
git clone ${SERVEUR} /tmp/git-tp-a
git clone ${SERVEUR} /tmp/git-tp-b
```

```
Cloning into '/tmp/git-tp-a'...
warning: You appear to have cloned an
empty repository.
done.
Cloning into '/tmp/git-tp-b'...
warning: You appear to have cloned an
empty repository.
done.
```

Premier auteur : un contenu initial, poussé sur le serveur vide.

```
cd /tmp/git-tp-a
echo "Contenu initial" > contenu.txt
git add contenu.txt
git commit -m "docs: premier contenu
partagé"
git push -u origin HEAD
```

```
[main (root-commit) 50a30c6] docs:
premier contenu partagé
 1 file changed, 1 insertion(+)
 create mode 100644 contenu.txt
Enumerating objects: 3, done.
Counting objects:  33% (1/3)
Counting objects:  66% (2/3)
Counting objects: 100% (3/3)
Counting objects: 100% (3/3), done.
Writing objects:  33% (1/3)
Writing objects:  66% (2/3)
Writing objects: 100% (3/3)
Writing objects: 100% (3/3), 253
bytes | 253.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta
0), pack-reused 0 (from 0)
```

```
To /tmp/git-tp-serveur.git
 * [new branch]      HEAD -> main
branch 'main' set up to track 'origin/
main'.
```

L'argument `HEAD` désigne la branche courante quel qu'en soit le nom (`main` ou `master` selon la configuration) : la démonstration ne dépend pas d'une convention de nommage.

Second auteur, parti de l'état **antérieur** à cette poussée :

```
cd /tmp/git-tp-b
echo "Contribution du second auteur"
> contribution-b.txt
git add contribution-b.txt
git commit -m "docs: contribution du
second clone"
```

```
[main (root-commit) 843a0c1] docs:
contribution du second clone
 1 file changed, 1 insertion(+)
  create mode 100644 contribution-
b.txt
```

Le clone `b` ignore le travail de `a` : il n'a rien fetché. Pousser revient donc à proposer un historique qui ferait disparaître le commit de `a` présent sur le serveur. Git refuse ; la cellule est marquée `error: true` pour afficher le refus :

```
cd /tmp/git-tp-b
git push -u origin HEAD
```

```
To /tmp/git-tp-serveur.git
 ! [rejected]          HEAD -> main
(fetch first)
error: failed to push some refs to '/
tmp/git-tp-serveur.git'
hint: Updates were rejected because
the remote contains work that you do
not
hint: have locally. This is usually
caused by another repository pushing
to
hint: the same ref. If you want to
integrate the remote changes, use
hint: 'git pull' before pushing
again.
hint: See the 'Note about fast-
forwards' in 'git push --help' for
details.
```

```
: 1
```

La sortie contient l'expression **non-fast-forward** : le commit à pousser n'a pas pour ancêtre le commit actuel du distant, et Git protège l'historique reçu. Le diagnostic se lit après un fetch :

```
git fetch origin
git log --oneline --decorate --all
git status --short --branch
```

```
remote: Enumerating objects: 3,
done.
remote: Counting objects: 33%
(1/3)
remote: Counting objects: 66%
(2/3)
remote: Counting objects: 100%
(3/3)
remote: Counting objects: 100% (3/3),
done.
remote: Total 3 (delta 0), reused
0 (delta 0), pack-reused 0 (from
0)
Unpacking objects: 33% (1/3)
Unpacking objects: 66% (2/3)
Unpacking objects: 100% (3/3)
Unpacking objects: 100% (3/3), 233
bytes | 233.00 KiB/s, done.
```

```
From /tmp/git-tp-serveur
 * [new branch]       main       ->
origin/main
843a0c1   (HEAD   ->   main)   docs:
contribution du second clone
50a30c6  (origin/main,  origin/HEAD)
docs: premier contenu partagé
##   main...origin/main   [ahead   1,
behind 1]
```

On y voit la divergence : `origin/main` porte le commit de `a`, la branche locale porte celui de `b`, et aucune ne contient l'autre. Intégrer puis pousser résout la situation :

```
git pull --rebase origin $(git branch
--show-current)
git log --oneline --graph --decorate
git push -u origin HEAD
git log --oneline --decorate --all
```

```
From /tmp/git-tp-serveur
 * branch             main       -
> FETCH_HEAD
Rebasing (1/1)


Successfully  rebased  and  updated
refs/heads/main.
*  a1bcf98  (HEAD  ->  main)  docs:
contribution du second clone
* 50a30c6 (origin/main, origin/HEAD)
docs: premier contenu partagé
Enumerating objects: 4, done.
Counting objects:  25% (1/4)
Counting objects:  50% (2/4)
Counting objects:  75% (3/4)
Counting objects: 100% (4/4)
Counting objects: 100% (4/4), done.
Delta  compression  using  up  to  32
threads
Compressing objects:  50% (1/2)
Compressing objects: 100% (2/2)
Compressing  objects:  100%  (2/2),
done.
Writing objects:  33% (1/3)
Writing objects:  66% (2/3)
Writing objects: 100% (3/3)
Writing  objects:  100%  (3/3),  333
bytes | 333.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta
0), pack-reused 0 (from 0)
To /tmp/git-tp-serveur.git
   50a30c6..a1bcf98  HEAD -> main
branch 'main' set up to track 'origin/
main'.
a1bcf98 (HEAD -> main, origin/main,
origin/HEAD)  docs:  contribution  du
second clone
50a30c6 docs: premier contenu partagé
```

Le rebassage a rejoué le commit local **au-dessus** de celui récupéré : l'historique reste linéaire. Attention à la portée de l'opération : **`--rebase` réécrit les commits locaux (de nouvelles empreintes pour un même contenu) ; ne jamais l'appliquer à des commits déjà poussés et travaillés par d'autres**, sous peine de reproduire la divergence du côté des partenaires.

12.3 Le dépôt d'exemple du cours

Le dépôt `HelloGit` du cours sert de premier clone réel. La boucle de retry ci-dessous est une précaution d'infrastructure (l'accès réseau des exécuteurs de rendu est instable), pas une opération Git enseignée.

```
# Retry x3 : l'egress des runners
CI est instable (Connection reset by
peer).
for i in 1 2 3; do
  git clone https://github.com/ebpro/
HelloGit.git && break
  echo "tentative $i échouée (réseau),
```

```
retry..."; rm -rf HelloGit; sleep 3
done
```

```
Cloning into 'HelloGit'...
remote:  Enumerating  objects:  39,
done.
remote:  Counting  objects:      2%
(1/39)
remote:  Counting  objects:      5%
(2/39)
remote:  Counting  objects:      7%
(3/39)
remote:  Counting  objects:     10%
(4/39)
remote:  Counting  objects:     12%
(5/39)
remote:  Counting  objects:     15%
(6/39)
remote:  Counting  objects:     17%
(7/39)
remote:  Counting  objects:     20%
(8/39)
remote:  Counting  objects:     23%
(9/39)
remote:  Counting  objects:     25%
(10/39)
remote:  Counting  objects:     28%
(11/39)
remote:  Counting  objects:     30%
(12/39)
remote:  Counting  objects:     33%
(13/39)
remote:  Counting  objects:     35%
(14/39)
remote:  Counting  objects:     38%
(15/39)
remote:  Counting  objects:     41%
(16/39)
remote:  Counting  objects:     43%
(17/39)
remote:  Counting  objects:     46%
(18/39)
remote:  Counting  objects:     48%
(19/39)
remote:  Counting  objects:     51%
(20/39)
remote:  Counting  objects:     53%
(21/39)
remote:  Counting  objects:     56%
(22/39)
remote:  Counting  objects:     58%
(23/39)
remote:  Counting  objects:     61%
(24/39)
remote:  Counting  objects:     64%
(25/39)
remote:  Counting  objects:     66%
(26/39)
remote:  Counting  objects:     69%
(27/39)
remote:  Counting  objects:     71%
(28/39)
remote:  Counting  objects:     74%
(29/39)
remote:  Counting  objects:     76%
(30/39)
remote:  Counting  objects:     79%
(31/39)
remote:  Counting  objects:     82%
(32/39)
remote:  Counting  objects:     84%
(33/39)
remote:  Counting  objects:     87%
(34/39)
remote:  Counting  objects:     89%
(35/39)
remote:  Counting  objects:     92%
(36/39)
remote:  Counting  objects:     94%
(37/39)
remote:  Counting  objects:     97%
(38/39)
remote:  Counting  objects:    100%
(39/39)
remote:  Counting  objects:    100%
```

```
(39/39), done.
remote:  Compressing  objects:     4%
(1/21)
remote:  Compressing  objects:     9%
(2/21)
remote:  Compressing  objects:    14%
(3/21)
remote:  Compressing  objects:    19%
(4/21)
remote:  Compressing  objects:    23%
(5/21)
remote:  Compressing  objects:    28%
(6/21)
remote:  Compressing  objects:    33%
(7/21)
remote:  Compressing  objects:    38%
(8/21)
remote:  Compressing  objects:    42%
(9/21)
remote:  Compressing  objects:    47%
(10/21)
remote:  Compressing  objects:    52%
(11/21)
remote:  Compressing  objects:    57%
(12/21)
remote:  Compressing  objects:    61%
(13/21)
remote:  Compressing  objects:    66%
(14/21)
remote:  Compressing  objects:    71%
(15/21)
remote:  Compressing  objects:    76%
(16/21)
remote:  Compressing  objects:    80%
(17/21)
remote:  Compressing  objects:    85%
(18/21)
remote:  Compressing  objects:    90%
(19/21)
remote:  Compressing  objects:    95%
(20/21)
remote:  Compressing  objects:  100%
(21/21)
```

```
remote:  Compressing  objects:  100%
(21/21), done.
Receiving objects:   2% (1/39)
Receiving objects:   5% (2/39)
Receiving objects:   7% (3/39)
Receiving objects:  10% (4/39)
Receiving objects:  12% (5/39)
Receiving objects:  15% (6/39)
Receiving objects:  17% (7/39)
Receiving objects:  20% (8/39)
Receiving objects:  23% (9/39)
Receiving objects:  25% (10/39)
remote: Total 39 (delta 3), reused
38 (delta 2), pack-reused 0 (from
0)
Receiving objects:  28% (11/39)
Receiving objects:  30% (12/39)
Receiving objects:  33% (13/39)
Receiving objects:  35% (14/39)
Receiving objects:  38% (15/39)
Receiving objects:  41% (16/39)
Receiving objects:  43% (17/39)
Receiving objects:  46% (18/39)
Receiving objects:  48% (19/39)
Receiving objects:  51% (20/39)
Receiving objects:  53% (21/39)
Receiving objects:  56% (22/39)
Receiving objects:  58% (23/39)
Receiving objects:  61% (24/39)
Receiving objects:  64% (25/39)
Receiving objects:  66% (26/39)
Receiving objects:  69% (27/39)
Receiving objects:  71% (28/39)
Receiving objects:  74% (29/39)
Receiving objects:  76% (30/39)
Receiving objects:  79% (31/39)
Receiving objects:  82% (32/39)
Receiving objects:  84% (33/39)
Receiving objects:  87% (34/39)
Receiving objects:  89% (35/39)
Receiving objects:  92% (36/39)
Receiving objects:  94% (37/39)
Receiving objects:  97% (38/39)
```

```
Receiving objects: 100% (39/39)
Receiving objects: 100% (39/39),
11.18 KiB | 673.00 KiB/s, done.
Resolving deltas:  0% (0/3)
Resolving deltas: 33% (1/3)
Resolving deltas: 66% (2/3)
Resolving deltas: 100% (3/3)
Resolving deltas: 100% (3/3), done.
```

⚠ Exercice : pousser vers une forge

Créez un dépôt vide sur GitHub (ou une autre forge), déclarez-le comme `origin` de votre `${PROJECT_DIR}` et poussez-y votre branche. L'opération est identique à la démonstration ci-dessus, l'URL de la forge remplaçant celle du serveur nu.

Solution

```
# Cellule non exécutée par le rendu : elle dépend de votre compte sur la forge.
git remote add origin git@github.com:VOTRE_UTILISATEUR/VOTRE_REPO.git
git push -u origin HEAD
```

12.4 Trois utilitaires hors chemin

Trois commandes se rencontrent dans la pratique ; elles sont nommées ici en ensemble fermé et développées ailleurs dans le module :

- `git stash` : mettre de côté les modifications non indexées pour les retrouver plus tard ;
- `git cherry-pick <commit>` : rejouer un commit désigné sur la branche courante ;
- `git rebase <base>` : rejouer vos commits sur une autre base — mécanisme et usage en leçon 02.

⚠ Commandes destructrices

- `git reset --hard <commit>` : déplace HEAD **et** réécrit l'arbre de travail — les modifications non committées sont perdues.
- `git clean -fd` : supprime fichiers et répertoires non suivis.

Le même réflexe de sûreté s'applique aux deux : simuler d'abord. `git clean -nd` liste ce qui serait supprimé sans rien supprimer. Pour `reset --hard`, les références déplacées restent consultables dans le journal local (`git reflog`) tant que le ramasse-miettes de Git n'a pas collecté les objets — récupération possible, jamais garantie.

13 Ce que vous retiendrez

Chacune des propositions suivantes se décide par une expérience menée dans cette leçon :

1. Git est distribué : après un clone, indexer, commiter, brancher et consulter s'exécutent sans appel réseau. *(Sections 1 et 4.)*
2. Le dépôt est le répertoire `.git` ; la copie de travail est distincte, et les deux suppressions ne sont pas symétriques. *(Section « Premier contact ».)*
3. Un dépôt nu n'a pas d'arbre de travail ; c'est ce qui le rend propre au rôle de serveur. *(Section « Dépôt nu ».)*
4. Un commit enregistre l'index, non l'arbre de travail ; entre les deux, `git diff --staged` montre l'écart. *(Sections « Indexer » et « Comparer ».)*
5. Un commit est un objet qui nomme un arbre et ses parents ; `git cat-file -p HEAD` se lit, et un commit de fusion contient deux lignes `parent`. *(Sections « Indexer » et « Fusions ».)*

6. Une branche est un fichier contenant une empreinte : 41 octets sous SHA-1, mesurés par `wc -c`. *(Section « Branches ».)*
7. Un conflit se matérialise par les marqueurs `<<<<<<<` dans le fichier ; il se résout par édition, puis `git add`, puis `git commit`. *(Section « Conflits ».)*
8. Un push est refusé en *non fast-forward* quand le distant porte des commits ignorés ; `git fetch` puis `git log --decorate --all` rendent la divergence visible. *(Section « Démonstration ».)*