WORKBOOK ÉTUDIANT

Missions guidées · tuto pas-à-pas · cheat sheets

Git

Votre carnet de bord pratique sur 7 séances

Formateur	Formation Insim – M. Slimani
Projet fil rouge	Osdigitalis - Formation - Git
Format	7 séances · ateliers en autonomie guidée
Version	v1.0 — 07/06/2026

Comment utiliser ce workbook

Ce carnet contient les **missions pratiques** à réaliser en séance, organisées par séance. Chaque mission donne un contexte, un objectif et une marche à suivre pas-à-pas. Les **cheat sheets** en fin de document sont vos antisèches à garder sous la main.

> ✓ **Réflexe à prendre**
>
> Après chaque commande, tapez `git status`. C'est la boussole : elle vous dit toujours où vous en êtes dans les 3 états de Git.

Séance 1 — Fondamentaux & premier dépôt

Mission 1.1 — Mon premier dépôt

> ℹ️ **Contexte**
> Vous démarrez le projet osd-blog. Rien n'existe encore.

Objectif : Créer un dépôt Git et réaliser votre premier commit.

Marche à suivre :

1. Configurez votre identité : `git config --global user.name` et `user.email`.
2. Créez un dossier `osd-blog`, placez-vous dedans et lancez `git init`.
3. Vérifiez la présence du dossier `.git` avec `ls -a`.
4. Créez un fichier `README.md` avec un titre.
5. Ajoutez-le à l'index (`git add`) puis validez (`git commit -m ...`).

> ✓ **Checkpoint**
> Un `git log` affiche votre commit ; `git status` indique un dépôt propre.

Mission 1.2 — Observer les 3 états

> ℹ️ **Contexte**
> Vous voulez comprendre le voyage d'un fichier dans Git.

Objectif : Identifier untracked → staged → committed.

Marche à suivre :

6. Créez un fichier `notes.txt` et lancez `git status` : notéz son état.
7. Faites `git add notes.txt`, relancez `git status` : qu'a-t-il changé ?
8. Modifiez `notes.txt` SANS refaire add : observez les deux sections de `git status`.
9. Commitez, puis vérifiez un dépôt propre.

> ✓ **Checkpoint**
> Vous savez nommer l'état d'un fichier à chaque étape.

Séance 2 — Cycle de travail de base

Mission 2.1 — Commits ciblés

> ℹ️ **Contexte**
> Vous avez modifié plusieurs fichiers mais voulez des commits propres et thématiques.

Objectif : Stager sélectivement et rédiger de bons messages.

Marche à suivre :

10. Créez `src/app.py` et modifiez `README.md` dans la même session.
11. Ajoutez puis commitez UNIQUEMENT `src/app.py` (`feat: ...`).
12. Ajoutez puis commitez `README.md` séparément (`docs: ...`).
13. Comparez `git diff` et `git diff --staged`.

> ✓ **Checkpoint**
> Deux commits distincts, chacun avec un message préfixé et explicite.

Mission 2.2 — Lire l'histoire

> ℹ️ **Contexte**
> Le projet a maintenant un historique.

Objectif : Naviguer dans l'historique et l'ignorer ce qui doit l'être.

Marche à suivre :

14. Essayez `git log`, `git log --oneline`, puis `--oneline --graph --all`.
15. Affichez le détail du dernier commit avec `git show HEAD`.
16. Créez un `.gitignore` (ex. `*.log`, `.env`, `__pycache__/`).
17. Vérifiez qu'un fichier `.env` n'apparaît plus dans `git status`.

> ✓ **Checkpoint**
> Le `.gitignore` est commité et masque bien les fichiers ciblés.

Séance 3 — Branches, fusion & sauvetage

Mission 3.1 — Travailler en branche

> ℹ️ **Contexte**
> Vous développez une fonctionnalité sans risquer la branche principale.

Objectif : Créer une branche, la fusionner.
Marche à suivre :

18. Créez et basculez sur `feature/login` (`git switch -c`).
19. Ajoutez `src/login.py`, commitez.
20. Revenez sur `main`, fusionnez `feature/login`.
21. Visualisez avec `git log --oneline --graph --all`.

> ✓ **Checkpoint**
> La fonctionnalité est sur main, le graphe montre la fusion.

Mission 3.2 — Résoudre un conflit

> ℹ️ **Contexte**
> Deux branches ont modifié la même ligne. Git ne peut pas choisir à votre place.

Objectif : Provoquer puis résoudre un conflit de fusion.
Marche à suivre :

22. Sur `feature/titre`, changez la 1ère ligne de `README.md`, commitez.
23. Sur `main`, changez la MÊME ligne différemment, commitez.
24. Fusionnez `feature/titre` dans `main` : un conflit apparaît.
25. Éditez le fichier, supprimez les marqueurs `<<<<<<<` `=======` `>>>>>>>`.
26. Faites `git add` puis `git commit` pour finaliser la fusion.

> ✓ **Checkpoint**
> Le `README.md` est cohérent et `git status` est propre.

Mission 3.3 — Le filet de sécurité

> ℹ️ **Contexte**
> Vous avez fait une fausse manip et cru tout perdre.

Objectif : Utiliser reset et reflog pour revenir en arrière.

Marche à suivre :

27. Faites un commit « de trop », puis `git reset --soft HEAD~1` : observez l'index.
28. Refaites le commit, puis testez `git reset --hard HEAD~1`.
29. Lancez `git reflog` et retrouvez le commit « perdu ».
30. Récupérez-le avec `git reset --hard HEAD@{1}`.

> ✓ **Checkpoint**
> Vous savez qu'un `--hard` est rattrapable via reflog.

Séance 4 — Réécriture & investigation

Mission 4.1 — Nettoyer avant de partager

> ℹ️ **Contexte**
> Votre branche contient 3 commits « wip » peu glorieux.

Objectif : Fusionner ces commits en un seul propre (squash).

Marche à suivre :

31. Sur une branche, créez 3 commits successifs « wip 1/2/3 ».
32. Lancez `git rebase -i HEAD~3`.
33. Passez les 2 derniers en `squash`, gardez le 1er en `pick`.
34. Rédigez un message final clair (`feat: ...`).

> ✓ **Checkpoint**
> Un `git log --oneline` ne montre plus qu'un commit propre.

Mission 4.2 — Chasser le bug avec bisect

> ℹ️ **Contexte**
> Un test échoue mais vous ignorez quel commit l'a cassé.

Objectif : Localiser le commit fautif par dichotomie.

Marche à suivre :

35. Lancez `git bisect start`.
36. Marquez la version actuelle : `git bisect bad`.
37. Marquez un ancien commit sûr : `git bisect good <hash>`.
38. Testez le commit proposé, répondez `good` ou `bad`, répétez.
39. Terminez par `git bisect reset`.

> ✓ **Checkpoint**
> Git désigne le premier commit « bad » : vous l'avez identifié.

Mission 4.3 — cherry-pick

> ℹ️ **Contexte**
> Un commit utile vit sur une autre branche, mais vous ne voulez que lui.

Objectif : Transplanter un commit précis sur main.

Marche à suivre :

40. Repérez le hash du commit voulu (`git log --oneline`).
41. Basculez sur `main`.
42. Appliquez `git cherry-pick <hash>`.

> ✓ **Checkpoint**
> Le commit apparaît sur main sans le reste de la branche source.

Séance 5 — GitLab : dépôt distant

Mission 5.1 — Pousser sur GitLab

> ℹ️ **Contexte**
> Votre travail est local ; il faut le partager sur la forge.

Objectif : Lier un dépôt distant et synchroniser.

Marche à suivre :

43. Créez un projet vide `osd-blog` sur GitLab.com.
44. Ajoutez le distant : `git remote add origin <url>`.
45. Vérifiez avec `git remote -v`.
46. Poussez : `git push -u origin main`.
47. Faites une modif via l'interface GitLab puis `git pull` en local.

> ✓ **Checkpoint**
> Le code est visible sur GitLab et le `pull` ramène la modif distante.

Séance 6 — Administration & collaboration

Mission 6.1 — Merge request & review

> ℹ️ **Contexte**
> En équipe, le code passe par une revue avant d'entrer dans main.

Objectif : Ouvrir une MR et la faire relire.

Marche à suivre :

48. Créez `feature/footer`, commitez une petite modif, poussez-la.
49. Ouvrez une **Merge Request** vers main sur GitLab.
50. Assignez un binôme comme reviewer ; il laisse un commentaire.
51. Prenez en compte la remarque, poussez à nouveau, puis mergez.

> ✓ **Checkpoint**
> La MR est mergée, la discussion de review est visible.

Mission 6.2 — Pipeline CI minimal

> ℹ️ **Contexte**
> Vous voulez automatiser les tests à chaque push.

Objectif : Créer un premier pipeline GitLab CI.

Marche à suivre :

52. Ajoutez un fichier `.gitlab-ci.yml` avec un job `test`.
53. Commitez et poussez.
54. Observez le pipeline se déclencher dans l'onglet **CI/CD**.

> ✓ **Checkpoint**
> Le pipeline passe au vert (ou échoue de manière lisible).

Séance 7 — Gitflow & synthèse

Mission 7.1 — Mini cycle Gitflow

> ℹ️ **Contexte**
> L'équipe adopte un workflow structuré pour préparer une release.

Objectif : Dérouler un cycle feature → develop → release → main.

Marche à suivre :

55. Créez `develop` depuis `main`.
56. Créez `feature/recherche` depuis develop, commitez, fusionnez dans develop (`--no-ff`).
57. Créez `release/1.0` depuis develop.
58. Fusionnez release dans main et taggez `v1.0` ; reportez dans develop.

> ✓ **Checkpoint**
> main porte le tag v1.0 ; le graphe reflète le modèle Gitflow.

Cheat sheet — commandes essentielles

Quotidien

Commande	Rôle
git status	État du dépôt (la boussole)
git add <f>	Stager un fichier
git commit -m "msg"	Valider l'index
git log --oneline --graph --all	Voir l'historique
git diff / --staged	Voir les changements

Branches & fusion

Commande	Rôle
git switch -c <b>	Créer + basculer
git switch <b>	Changer de branche
git merge <b>	Fusionner dans la branche courante
git branch -d <b>	Supprimer une branche fusionnée

Sauvetage & réécriture

Commande	Rôle
git reset --soft/--mixed/--hard	Reculer HEAD (3 niveaux)
git reflog	Historique de HEAD (rattrapage)
git stash / pop	Mettre de côté / réappliquer
git rebase -i HEAD~n	Réécrire les n derniers commits
git cherry-pick <hash>	Appliquer un commit précis
git revert <hash>	Annuler par un commit inverse

Investigation

Commande	Rôle
git blame <f>	Auteur ligne par ligne
git bisect start/good/bad/reset	Trouver le commit fautif

Distant

Commande	Rôle
git remote add origin <url>	Lier un distant
git push -u origin <b>	Pousser + suivre
git fetch / pull	Récupérer / récupérer+fusionner
git clone <url>	Cloner un dépôt

> ⚠ **Règle d'or**
> On ne réécrit JAMAIS (rebase, reset --hard, push --force) un historique déjà partagé avec l'équipe.