

Contents

TP 1 : projet Git collaboratif	1
Ce dont vous avez besoin	1
Partie 1 (jour 1). Le cycle complet et le conflit (1h)	2
Étape 1 : le mainteneur crée le dépôt (10 min)	2
Étape 2 : le mainteneur protège main (5 min)	3
Étape 3 : chacun clone le dépôt et crée sa branche (5 min)	3
Étape 4 : créer sa carte et l'ajouter à l'accueil (10 min)	4
Étape 5 : pousser et ouvrir la Pull Request (5 min)	5
Étape 6 : relire la PR d'un autre membre (10 min)	5
Étape 7 : fusionner... et résoudre le conflit (15 min)	6
Je valide avec le formateur (fin du jour 1)	7
Partie 2 (jour 2). Rebase, tag et release (40 min)	7
Étape 8 : une petite modification sur une nouvelle branche (10 min)	7
Étape 9 : rebaser sa branche sur main avant la fusion (10 min)	8
Étape 10 : taguer la version v1.0.0 (10 min)	8
Étape 11 : publier la release GitHub (10 min)	9
Je valide avec le formateur (jour 2)	10
Variante solo (si vous êtes seul)	10
Si ça ne marche pas (aide-mémoire)	11

TP 1 : projet Git collaboratif

Durée : 1h le jour 1 (partie 1) + 40 min le jour 2 (partie 2) | **Format** : groupes de 3 ou 4 (variante solo en fin d'énoncé) **Objectif** : contribuer à un dépôt partagé avec le cycle complet *branche* → *commit* → *push* → *Pull Request* → *revue* → *merge*, résoudre un conflit provoqué volontairement, puis (jour 2) rebaser sa branche, taguer une version et publier une release GitHub.

Ce dont vous avez besoin

- Git configuré, une clé SSH enregistrée sur GitHub (ssh -T git@github.com répond Hi <pseudo>!) : voir [../ressources/setup-poste.md](https://dwdm.fr/ressources/setup-poste.md).
- VS Code. gh (GitHub en ligne de commande) est pratique mais facultatif : chaque étape donne aussi le chemin dans l'interface web.
- Le site de départ : le dossier [starter/](https://dwdm.fr/ressources/cheatsheet-git.md) (un petit site statique « Promo DWWM »).
- L'antisèche : [../ressources/cheatsheet-git.md](https://dwdm.fr/ressources/cheatsheet-git.md).

Vocabulaire

Terme	Signification
Dépôt (<i>repository</i> , « repo »)	Le projet versionné par Git, ici hébergé sur GitHub
Branche	Une ligne de travail parallèle ; techniquement, un pointeur vers un commit

Terme	Signification
PR (<i>Pull Request</i>)	Demande de fusion d'une branche dans main, avec discussion et revue de code
Revue de code	Relecture du code d'un autre avant la fusion
Conflit	Deux branches ont modifié les mêmes lignes : Git ne sait pas laquelle garder et vous demande de choisir
Mainteneur	La personne qui crée le dépôt et l'administre (invitations, protection de main)

Répartissez les rôles maintenant : une personne du groupe est le **mainteneur** ; les autres sont **contributeurs**. Tout le monde, mainteneur compris, ajoutera sa carte.

Dans les commandes, remplacez :

- <mainteneur> par le pseudo GitHub du mainteneur ;
- <prenom> par **votre** prénom, en minuscules, sans accent (alice, jean-baptiste).

Partie 1 (jour 1). Le cycle complet et le conflit (1h)

Étape 1 : le mainteneur crée le dépôt (10 min)

Mainteneur uniquement. Depuis le dossier de la formation :

```
cp -r tp/tp1-git-collaboratif/starter ~/site-promo
cd ~/site-promo
git init -b main
git add .
git commit -m "chore: initialise le site Promo DWWM"
```

Puis créez le dépôt **public** sur GitHub et poussez :

```
gh repo create site-promo --public --source=. --push
```

Sans gh : sur GitHub, + > *New repository*, nom site-promo, **Public**, ne cochez **rien** (ni README, ni .gitignore), *Create repository*. Puis :

```
git remote add origin git@github.com:<mainteneur>/site-promo.git
git push -u origin main
```

Invitez les autres membres : **Settings** > **Collaborators** > **Add people** > pseudo GitHub de chaque membre.

Contributeurs : acceptez l'invitation (reçue par e-mail, ou sur <https://github.com/<mainteneur>/site-promo/invitations>).

Résultat attendu :

```
$ git log --oneline
3f2a1c9 (HEAD -> main, origin/main) chore: initialise le site Promo DWWM
```

et sur GitHub, la page du dépôt affiche le README.md du site ; chaque contributeur voit le dépôt dans sa liste.

Si ça ne marche pas - cp: cannot stat 'tp/tp1-git-collaboratif/starter' : vous n'êtes pas dans le dossier de la formation (pwd, puis cd). - Permission denied (publickey) au push : votre clé SSH n'est pas sur GitHub ; refaites la section 2 de setup-poste.md. - gh: command not found : utilisez la méthode « sans gh » ci-dessus. - Le dossier .github/ n'apparaît pas avec ls : c'est normal, il est caché. Utilisez ls -a.

Pourquoi ? Le dépôt est **public** car la protection de branche (étape 2) n'est disponible gratuitement que sur les dépôts publics. Le fichier .github/pull_request_template.md pré-remplira chaque PR.

Étape 2 : le mainteneur protège main (5 min)

Mainteneur uniquement. Settings > Branches > Add classic branch protection rule (selon l'interface : *Add branch protection rule*) :

- *Branch name pattern* : main
- ☒ *Require a pull request before merging*
 - ☒ *Require approvals* : **1**
- ☒ *Do not allow bypassing the above settings*
- **Create** (GitHub peut redemander votre mot de passe).

Vérifiez que la protection fonctionne : un push direct sur main doit être **refusé**.

```
git commit --allow-empty -m "test: push direct"
git push
```

Résultat attendu :

```
remote: error: GH006: Protected branch update failed for refs/heads/main.
! [remote rejected] main -> main (protected branch hook declined)
```

Annulez ce commit de test, qui n'existe que chez vous :

```
git reset --hard HEAD~1
```

Si ça ne marche pas - Le push **passe** : la case *Do not allow bypassing* n'est pas cochée (en tant qu'administrateur, vous contourniez la règle) ou le motif n'est pas exactement main. Corrigez la règle, puis annulez le commit poussé avec une PR (git revert) : on ne réécrit pas l'historique de main. - Pas de menu *Branches* : vous n'êtes pas sur **votre** dépôt, ou il est privé.

Pourquoi ? main doit toujours contenir un site qui fonctionne (GitHub Flow). La protection rend la règle **automatique** : plus besoin de « faire attention ».

Étape 3 : chacun clone le dépôt et crée sa branche (5 min)

Tout le monde, mainteneur compris (le mainteneur travaille dans ~/site-promo, déjà cloné) :

```
cd ~
git clone git@github.com:<mainteneur>/site-promo.git
```

```
cd site-promo
git switch -c feature/carte-<prenom>
```

Résultat attendu :

```
$ git branch
* feature/carte-alice
main
```

Si ça ne marche pas - Repository not found : l'invitation n'est pas encore acceptée, ou le pseudo est mal écrit. - fatal: destination path 'site-promo' already exists : vous avez déjà un dossier de ce nom ; clonez ailleurs (git clone ... site-promo-groupe).

Pourquoi ? Une branche par personne et par sujet : chacun travaille sans gêner les autres, et main reste stable. Le nom suit la convention de [starter/CONTRIBUTING.md](#).

Étape 4 : créer sa carte et l'ajouter à l'accueil (10 min)

```
cp cartes/exemple.html cartes/<prenom>.html
code .
```

1. Dans cartes/<prenom>.html, remplacez : le <title>, l'initiale de l'avatar, le nom, la ligne de rôle, les trois rubriques et les étiquettes. Restez bienveillant : le dépôt est public.
2. Dans index.html, ajoutez **une** ligne dans la liste, **juste au-dessus du commentaire** :

```
<ul class="promo">
  <li><a href="cartes/exemple.html">Camille (exemple)</a></li>
  <li><a href="cartes/<prenom>.html">Votre prénom</a></li>
  <!-- chacun ajoute sa ligne ICI, juste au-dessus de ce commentaire -->
</ul>
```

3. Ouvrez index.html dans le navigateur : votre carte apparaît, le lien fonctionne, le lien « Retour » aussi.

Puis committez :

```
git status
git add cartes/<prenom>.html index.html
git commit -m "feat: ajoute la carte de <Prénom>"
```

Résultat attendu :

```
$ git log --oneline
8c1d2e4 (HEAD -> feature/carte-alice) feat: ajoute la carte d'Alice
3f2a1c9 (origin/main, origin/HEAD, main) chore: initialise le site Promo DWWW
```

Si ça ne marche pas - La carte s'affiche sans mise en forme : le lien vers la feuille de style doit rester ../style.css (la carte est dans un sous-dossier). - Le lien de l'accueil mène à une page « introuvable » : vérifiez l'orthographe **et la**

casse du nom de fichier (`alice.html` et `Alice.html` sont deux fichiers différents sous Linux).

Pourquoi ? Tout le monde modifie **la même zone** de `index.html` : c'est voulu, c'est ce qui provoquera le conflit de l'étape 7.

Étape 5 : pousser et ouvrir la Pull Request (5 min)

```
git push -u origin feature/carte-<prenom>
```

Sur GitHub, un bandeau jaune propose **Compare & pull request** : cliquez dessus.

1. Vérifiez la flèche : *base: main* ← *compare: feature/carte-<prenom>*.
2. Complétez la description pré-remplie (cochez les cases de la checklist).
3. Colonne de droite, **Reviewers** : choisissez une personne du groupe (organisez-vous pour que chacun relise une PR).
4. **Create pull request**.

Avec gh : `gh pr create --fill --reviewer <pseudo-du-relecteur>`.

Résultat attendu : la PR est ouverte, l'onglet *Conversation* indique *Review required* et le bouton de fusion est grisé (*Merging is blocked*).

Si ça ne marche pas - Pas de bandeau : onglet **Pull requests** > **New pull request**, puis choisissez votre branche dans *compare*. - Votre camarade n'apparaît pas dans *Reviewers* : il n'a pas encore accepté l'invitation de l'étape 1.

Pourquoi ? La PR est le lieu de la discussion : la description explique **pourquoi**, l'onglet *Files changed* montre **quoi**, et la protection de `main` impose qu'une deuxième personne regarde.

Étape 6 : relire la PR d'un autre membre (10 min)

Ouvrez la PR que vous devez relire, onglet **Files changed**. Vérifiez (voir la section 6 de `CONTRIBUTING.md`) :

- le lien ajouté dans `index.html` pointe vers le bon fichier ;
- le nom de la branche et le message de commit respectent la convention ;
- aucun fichier inattendu (copie de sauvegarde, `.DS_Store`).

Faites **au moins un commentaire sur une ligne** (survolez la ligne, cliquez sur le + bleu), même positif. Puis **Review changes** :

- *Request changes* s'il faut corriger quelque chose : l'auteur corrige sur **sa branche**, committe, pousse ; la PR se met à jour toute seule ;
- *Approve* si tout est bon.

Résultat attendu : chaque PR a au moins un commentaire et une approbation ; le bouton **Merge pull request** devient vert.

Si ça ne marche pas - Vous ne pouvez pas approuver : on ne peut pas approuver **sa propre** PR. Échangez avec un autre membre. - L'auteur a poussé une correction mais la PR ne change pas : il a commité sur `main` au lieu de sa branche (`git branch` pour vérifier).

Pourquoi ? La revue attrape les erreurs avant main, et elle diffuse la connaissance : au moins deux personnes connaissent chaque changement. Un commentaire précis et bienveillant fait progresser ; « c'est nul » ne sert à rien.

Étape 7 : fusionner... et résoudre le conflit (15 min)

1. La **première** PR approuvée est fusionnée par son auteur : **Merge pull request > Confirm merge > Delete branch.**
2. Rechargez les autres PR : elles affichent maintenant *This branch has conflicts that must be resolved*. C'est **normal et voulu** : tout le monde a modifié la même ligne de index.html.

Chaque auteur dont la PR est en conflit le résout **en local** :

```
git switch main
git pull
git switch feature/carte-<prenom>
git merge main
```

Résultat attendu :

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

Ouvrez index.html dans VS Code. Vous voyez :

```
<<<<<<< HEAD
    <li><a href="cartes/bob.html">Bob</a></li>
=====
    <li><a href="cartes/alice.html">Alice</a></li>
>>>>>>> main
```

- entre <<<<<<< HEAD et ===== : **votre** version (votre branche) ;
- entre ===== et >>>>>>> main : la version de main (la carte déjà fusionnée).

Ici, il faut garder **les deux lignes**. Cliquez sur *Accept Both Changes* (au-dessus du bloc dans VS Code), ou supprimez les trois lignes de marqueurs à la main. Résultat :

```
<li><a href="cartes/bob.html">Bob</a></li>
<li><a href="cartes/alice.html">Alice</a></li>
```

Vérifiez qu'il ne reste aucun marqueur, ouvrez la page dans le navigateur, puis terminez la fusion :

```
grep -n '<<<<<<<\|=====\|>>>>>>>' index.html    # ne doit RIEN afficher
git add index.html
git commit --no-edit
git push
```

Résultat attendu : sur GitHub, la PR indique *This branch has no conflicts with the base branch*. Fusionnez-la. Le membre suivant recommence la résolution (un nouveau conflit apparaît à chaque fusion : c'est normal).

Si ça ne marche pas - `git merge main` répond Already up to date : vous n'avez pas fait `git pull` sur main avant. - error: Merging is not possible because you have unmerged files : vous avez oublié `git add index.html`. - Vous êtes perdu au milieu de la fusion : `git merge --abort` remet tout comme avant `git merge main`, puis recommencez. - Un éditeur en ligne de commande s'ouvre (Vim) : tapez `:wq` puis Entrée. La prochaine fois, utilisez `--no-edit`. - Après fusion, la PR est « approuvée » mais vous avez changé du code : demandez quand même un coup d'œil au relecteur.

Pourquoi ? Un conflit n'est pas une erreur : c'est Git qui refuse de choisir à votre place. On le résout **sur sa branche**, jamais directement sur main, et on vérifie le résultat avant de pousser.

Je valide avec le formateur (fin du jour 1)

- ☐ Chaque membre a une PR **relue** (au moins un commentaire) et **fusionnée**.
- ☐ Au moins un conflit a été résolu en local, et je sais expliquer les trois marqueurs.
- ☐ main ne contient aucun marqueur de conflit et toutes les cartes s'affichent.
- ☐ Un push direct sur main est refusé.

Questions de réflexion (une fois que tout fonctionne) :

1. Pourquoi le conflit s'est-il produit dans `index.html` et jamais dans `cartes/<prenom>.html` ?
2. Comment auriez-vous pu organiser le fichier pour éviter ce conflit ? Est-ce souhaitable d'éviter **tous** les conflits ?
3. Lancez `git log --oneline --graph --all` sur main. Combien de commits de fusion voyez-vous ? Est-ce facile à lire ? (Réponse demain avec le rebase.)

Partie 2 (jour 2). Rebase, tag et release (40 min)

Étape 8 : une petite modification sur une nouvelle branche (10 min)

Tout le monde. Repartez de la dernière version de main :

```
cd ~/site-promo
git switch main
git pull
git switch -c feature/style-<prenom>
```

Faites un petit changement **dans votre carte seulement** (`cartes/<prenom>.html`) : une étiquette de plus, une rubrique « Mon projet préféré »... Puis :

```
git add cartes/<prenom>.html
git commit -m "feat: complète la carte de <Prénom>"
git push -u origin feature/style-<prenom>
```

Ouvrez la PR (étape 5) et demandez une revue. **Ne fusionnez pas encore.**

Résultat attendu : une PR ouverte par membre, sans conflit (chacun ne touche que son propre fichier).

Pourquoi ? Pendant que vous travaillez, les autres fusionnent : votre branche part d'une version de main qui vieillit. Le rebase va la « rajeunir ».

Étape 9 : rebaser sa branche sur main avant la fusion (10 min)

Attendez qu'une autre PR de l'étape 8 soit fusionnée (une seule personne fusionne la sienne en premier, avec **Rebase and merge** dans la liste déroulante du bouton vert). Ensuite, chacun des autres membres :

```
git fetch origin
git log --oneline --graph --all -n 8      # AVANT : votre branche part d'un ancien commit
git rebase origin/main
git log --oneline --graph --all -n 8      # APRÈS : votre commit est posé au-dessus du dernier
```

Résultat attendu :

```
$ git rebase origin/main
Successfully rebased and updated refs/heads/feature/style-alice.
$ git log --oneline --graph -n 3
* 5e6f7a8 (HEAD -> feature/style-alice) feat: complète la carte d'Alice
* 9b0c1d2 (origin/main) feat: complète la carte de Bob
* 4d5e6f7 Merge pull request #3 from .../feature/carte-chloe
```

Votre branche a déjà été poussée à l'étape 8 et le rebase a **réécrit** ses commits (nouveaux identifiants). Il faut donc forcer le push, prudemment :

```
git push --force-with-lease
```

Fusionnez ensuite la PR après approbation, en choisissant **Rebase and merge** dans la liste déroulante du bouton vert : l'historique de main reste une ligne droite.

Si ça ne marche pas - CONFLICT pendant le rebase : résolvez comme à l'étape 7, puis `git add <fichier>` et `git rebase --continue` (et **pas** `git commit`). Pour tout annuler : `git rebase --abort`. - ! [rejected] ... (non-fast-forward) sur un `git push` simple : c'est attendu après un rebase, utilisez `git push --force-with-lease`. - stale info avec `--force-with-lease` : quelqu'un d'autre a poussé sur **votre** branche. Faites `git fetch`, regardez ce qui a changé, et parlez-en avec la personne avant de forcer. - Le bouton ne propose pas *Rebase and merge* : le mainteneur l'active dans **Settings > General > Pull Requests > Allow rebase merging**.

Pourquoi ? Le rebase rejoue vos commits au-dessus de main : l'historique est linéaire et la PR contient exactement ce qui sera fusionné. **Règle d'or** : on ne rebase jamais une branche **partagée** (comme main), seulement sa propre branche. `--force-with-lease` refuse d'écraser un travail que vous n'avez pas encore vu, contrairement à `--force`.

Étape 10 : taguer la version v1.0.0 (10 min)

Quand toutes les PR de l'étape 9 sont fusionnées, **le mainteneur** :

```
git switch main
git pull
```

```
git log --oneline --graph -n 10
git tag -a v1.0.0 -m "v1.0.0 : première version du site de la promo"
git push origin v1.0.0
```

Résultat attendu :

```
$ git show v1.0.0 --no-patch
tag v1.0.0
Tagger: Prénom Nom <vous@exemple.fr>
Date:    ...

v1.0.0 : première version du site de la promo

commit 9b0c1d2... (HEAD -> main, tag: v1.0.0, origin/main)
```

Sur GitHub, le menu des branches (onglet *Tags*) affiche v1.0.0. Les autres membres récupèrent le tag avec `git fetch --tags`.

Si ça ne marche pas - fatal: tag 'v1.0.0' already exists : un tag de ce nom existe déjà chez vous. `git tag -d v1.0.0` (local seulement), puis recommencez.
- Le tag n'apparaît pas sur GitHub : un simple `git push` n'envoie **pas** les tags ; il faut `git push origin v1.0.0`.

Pourquoi ? Un tag nomme un commit précis, pour toujours. **SemVer** (*Semantic Versioning*) : MAJEUR.MINEUR.CORRECTIF. On augmente CORRECTIF pour une correction, MINEUR pour un ajout compatible, MAJEUR pour un changement qui casse l'existant. Un tag **annoté** (-a) garde l'auteur, la date et un message, contrairement à un tag léger.

Étape 11 : publier la release GitHub (10 min)

Mainteneur : onglet **Releases** (colonne de droite de la page du dépôt) > **Draft a new release** :

1. *Choose a tag* : v1.0.0.
2. *Release title* : v1.0.0 : première version.
3. Bouton **Generate release notes** : GitHub liste automatiquement les PR fusionnées depuis le début.
4. Relisez, complétez une phrase d'introduction, puis **Publish release**.

Avec gh : `gh release create v1.0.0 --title "v1.0.0 : première version" --generate-notes`.

Résultat attendu : la page *Releases* affiche v1.0.0 (*Latest*), avec la liste des PR et de leurs auteurs, et une archive .zip du code téléchargeable.

Pourquoi ? La release est la version « présentable » d'un tag : des notes lisibles par des humains (une sorte de CHANGELOG, le journal des changements) et une archive. Des messages de commit et des titres de PR propres donnent des notes propres : la qualité se prépare dès le premier commit.

Je valide avec le formateur (jour 2)

- ☐ `git log --oneline --graph` sur main montre un historique lisible ; les branches `feature/style-*` ont été rebasées.
- ☐ J'ai utilisé `git push --force-with-lease` (et je sais pourquoi pas `--force`).
- ☐ Le tag annoté `v1.0.0` est visible sur GitHub.
- ☐ La release `v1.0.0` est publiée avec ses notes générées.

Questions de réflexion :

1. Comparez le graphe de main avant et après la partie 2. Qu'apporte le rebase ? Que perd-on ?
 2. Pourquoi est-il dangereux de rebaser main ? Que verraient vos camarades au `git pull` suivant ?
 3. Vous ajoutez une nouvelle section « Nos projets » au site. Quel est le prochain numéro de version ? Et pour une faute d'orthographe corrigée ?
-

Variante solo (si vous êtes seul)

Vous jouez tous les rôles. Le conflit est simulé avec **deux branches** qui modifient la même ligne.

1. **Étape 1** : créez le dépôt comme le mainteneur (`~/site-promo`), sans inviter personne.
2. **Étape 2** : protégez main, mais **décochez** *Require approvals* (on ne peut pas approuver sa propre PR). Gardez *Require a pull request before merging* et *Do not allow bypassing*.
3. Créez deux cartes sur **deux branches partant toutes deux de main** :

```
git switch main
git switch -c feature/carte-<prenom>
# carte cartes/<prenom>.html + une ligne dans index.html, juste au-dessus du comm
git add . && git commit -m "feat: ajoute la carte de <Prénom>"
git push -u origin feature/carte-<prenom>

git switch main
git switch -c feature/carte-mentor
# carte cartes/mentor.html (votre formateur ou un personnage de votre choix)
# + une ligne dans index.html, AU MÊME ENDROIT
git add . && git commit -m "feat: ajoute la carte du mentor"
git push -u origin feature/carte-mentor
```

4. Ouvrez les **deux** PR. Faites une **auto-revue** : dans *Files changed*, laissez au moins un commentaire sur une ligne (*Review changes* > *Comment*).
5. Fusionnez la PR `feature/carte-<prenom>`. La seconde affiche alors un conflit : résolvez-le exactement comme à l'étape 7 (sur la branche `feature/carte-mentor`), poussez, fusionnez.
6. **Jour 2** : faites les étapes 8 à 11 avec deux branches `feature/style-<prenom>` et `feature/style-mentor` : fusionnez la première, rebasez la seconde sur `origin/main`, `git push --force-with-lease`, fusionnez, puis taguez et publiez la release.

La checklist de validation est la même ; pour la revue, le formateur regarde vos commentaires d'auto-revue.

Si ça ne marche pas (aide-mémoire)

Symptôme	Cause probable	Que faire
GH006: Protected branch update failed Merging is blocked sur la PR CONFLICT (content) Your branch is ahead of 'origin/main' sur main	Vous poussez sur main Pas encore d'approbation, ou conflit Même ligne modifiée des deux côtés Vous avez commité sur main par erreur	Normal : travaillez sur une branche et ouvrez une PR Demander une revue ; résoudre le conflit en local Étape 7 : choisir, supprimer les marqueurs, add, commit git switch -c feature/<sujet> (emporte le commit), puis git switch main && git reset --hard origin/main
rejected (non-fast-forward) Tout semble cassé	La branche distante a des commits que vous n'avez pas, ou vous avez rebasé	Sans rebase : git pull ; après un rebase : git push --force-with-lease git status d'abord : il dit presque toujours quoi faire. Puis appelez le formateur

Exercices complémentaires à faire seul : [../..exercices/jour1/](#) (commandes Git en solo, kata conflit) et [../..exercices/jour2/](#) (rebase interactif).