Méthodologie de travail collaboratif – Git / GitHub

Projet de référence confirmé par le Prof : **MOSBAM P Cloud**

1. Organisation générale

1.1 Composition de l'équipe

Nous travaillons en binôme :

- **Charles-Henri Moser** : plutôt responsable de la partie backend (API, base de données, logique métier).
- **Mathieu Bamert** : plutôt responsable de la partie frontend (interface, formulaires, affichage).

Ce ne sont pas des rôles stricts : chacun peut aider l'autre si besoin.
Les deux personnes sont responsables de :

- Respecter les règles Git/GitHub définies ci-dessous.
- Mettre à jour les tâches dans GitHub.
- Relire le code de l'autre avant intégration.

1.2 Outils utilisés

Nous utilisons principalement :

- **GitHub**
    - Pour héberger le code du projet.
    - Pour gérer les issues (tâches, bugs, idées).
    - Pour les Pull Requests (revue de code).
- **GitHub Projects (Kanban simple)**
  Tableau avec les colonnes :
    - `À faire`
    - `En cours`
    - `En revue`
    - `Terminé`
- **Discord, Teams (ou WhatsApp)**
    - Messages rapides pour se poser des questions.
    - Appels vocaux si besoin d'expliquer un problème plus compliqué.

1.3 Transparence et suivi des tâches

- Chaque tâche importante (nouvelle fonctionnalité, bug, nettoyage de code) a une issue GitHub.
- L'issue contient :
    - Un titre clair (ex. `Ajouter la création d'un livre`)
    - Une petite description de ce qu'il faut faire.
- Avant de commencer une tâche, on déplace l'issue dans `En cours`.

- Quand la Pull Request est créée, l'issue passe dans `En revue`.
- Quand la PR est mergée, l'issue passe dans `Terminé` et est fermée.

Cela permet à chacun de voir rapidement qui fait quoi et où en est le projet.

---

2. Workflow Git / GitHub

2.1 Organisation des branches

Nous utilisons peu de branches pour rester simple :

- `main`
  - Contient la version stable du projet.
  - Sert de base pour les livraisons et les démos.
  - Pas de commit direct dessus.
- `dev`
  - Branche de travail principale.
  - Toutes les nouvelles fonctionnalités validées sont mergées ici avant `main`.
- **Branches de fonctionnalité :** `feature-nom-tache`
  - Exemple : `feature-ajout-livre`, `feature-login`.
  - Créées à partir de `dev`.
  - Supprimées après merge.
- **Branches de correction :** `fix-nom-bug` **(si besoin)**
  - Exemple : `fix-erreur-500-livres`.
  - Utilisées quand on corrige un bug précis.

Fonctionnement typique :

1. On part de `dev` à jour.
2. On crée une branche `feature-...`.
3. On effectue les commits sur cette branche.
4. On pousse la branche sur GitHub.
5. On ouvre une Pull Request vers `dev`.
6. Après revue et validation, on merge dans `dev`.
7. Quand `dev` est stable, on merge `dev` vers `main` pour une nouvelle version.

2.2 Versionnage

Pour les versions, nous utilisons un schéma simple :

- **MAJEUR.MINEUR.CORRECTION**

Exemples :

- `1.0.0` → première version stable présentée.

- `1.1.0` → ajout de nouvelles fonctionnalités.
- `1.1.1` → petite correction de bug.

Nous notons les versions dans les releases GitHub ou dans le README pour savoir quelle version a été présentée en cours.

2.3 Convention de commits

Nous utilisons une convention simple pour les messages de commit :

- Format recommandé :
  `type: message court`

Types utilisés :

- `feat:` → nouvelle fonctionnalité
  - ex. `feat: ajout de la création de livre`
- `fix:` → correction de bug
  - ex. `fix: correction du tri par date`
- `doc:` → documentation
  - ex. `doc: mise à jour du README`
- `refactor:` → amélioration du code sans changement fonctionnel
  - ex. `refactor: simplification du contrôleur`
- `style:` → changements de mise en forme (indentation, renommage léger, CSS)

Avantage :

- En lisant l'historique, on voit rapidement quels commits ajoutent des fonctionnalités, lesquels corrigent des bugs, etc.

---

3. Processus de revue de code (Pull Requests)

3.1 Déroulement d'une Pull Request

1. Le développeur termine la fonctionnalité sur la branche `feature-...`.
2. Il pousse la branche sur GitHub.
3. Il crée une Pull Request vers la branche `dev` avec :
   - Un titre clair (ex. `Ajout de la création de livre`).
   - Une petite description des changements.
   - Éventuellement : comment tester (ex. « Aller sur /livres, cliquer sur Ajouter, vérifier que… »).
4. Il demande à son binôme de relire la PR.
5. Le binôme :
   - Vérifie que le code est compréhensible.
   - Vérifie que ça compile et que la fonctionnalité fonctionne.
   - Peut faire des commentaires si quelque chose ne va pas.

6. Quand tout est bon :
    - La PR est approuvée.
    - La branche est mergée dans `dev`.
    - La branche `feature-...` peut être supprimée.

3.2 Critères pour accepter une PR

Avant de merger une PR, nous vérifions :

1. **Fonctionnement**
    - La fonctionnalité décrite dans la PR marche réellement.
    - Les principaux cas d'usage ont été testés manuellement.
2. **Code**
    - Pas de gros morceaux de code commentés inutiles.
    - Pas de `console.log` (ou équivalent) laissés par erreur.
    - Le code est lisible, avec un minimum de commentaires si nécessaire.
3. **Impact**
    - La nouvelle fonctionnalité ne casse pas une autre partie visible du projet (test rapide des écrans principaux).
4. **Tâches**
    - L'issue associée est mise à jour et sera déplacée dans `Terminé` après le merge.

---

4. Bonnes pratiques de collaboration

Nous essayons de respecter quelques règles simples :

- Ne jamais travailler directement sur `main`.
- Toujours créer une branche pour une fonctionnalité ou une correction.
- Faire des commits réguliers plutôt qu'un énorme commit à la fin.
- Puller / fetch `dev` régulièrement pour éviter les gros conflits.
- Préférer des petites PR faciles à relire.
- Se parler (Discord / WhatsApp) dès qu'il y a un doute ou un blocage.

---

5. Justification de cette organisation

Avantages :

- Méthode simple, adaptée à un petit binôme.
- Peu de branches à gérer (`main`, `dev`, `feature-...`).
- Les Pull Requests permettent de relire le code et d'éviter de gros bugs.
- Le tableau Kanban et les issues donnent une vue claire de l'avancement du projet.

Limites :

- Nous n’avons pas mis en place de vrais outils de CI/CD automatiques (tests automatiques, etc.).
- Si nous ne mettons pas régulièrement à jour le Kanban ou les issues, l’organisation peut vite devenir moins claire.
- Le versionnage reste simple et manuel (pas de tags systématiques, pas d’automatisation).