Gestion de Projet

Sommaire

1. Méthodologie

1.1. Approche adoptée

Le projet a été géré en mode **itératif par fonctionnalité**, inspiré des méthodes agiles. Chaque fonctionnalité correspond à une **issue GitHub**, une **branche dédiée** et une **pull request** vers la branche d'intégration `develop`.

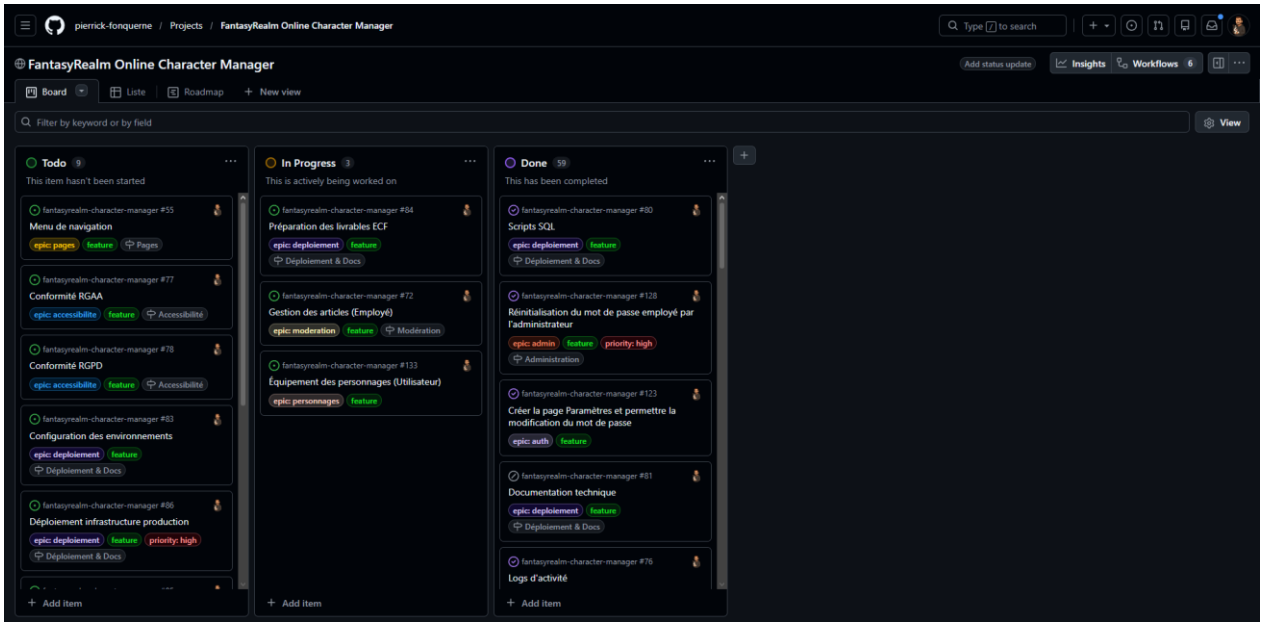
Ce choix s'explique par le contexte du projet : développeur unique, délai fixe, périmètre fonctionnel défini par l'énoncé ECF. L'approche par fonctionnalité permet de livrer des incréments fonctionnels testables à chaque étape.

1.2. Transition Jira → GitHub

Le suivi des tâches a initialement été mis en place sur **Jira** (Atlassian) avec la redaction de "user stories" et de "epics". Cependant, le plan gratuit de Jira ne permet pas de rendre un projet public, ce qui posait un problème pour la transparence du travail vis-à-vis de l'évaluation.

L'ensemble du backlog a donc été **migré vers GitHub Projects**, permettant :

- Un suivi public et vérifiable du travail réalisé
- Une intégration native avec les branches et les pull requests
- Un board Kanban intégré au repository



Toutefois le fonctionnement GitHub Projects avec ses "issues" a demandé une adaptation des tickets, la creation de "tags" et de "milestones" en remplacement des "Epics" de Jira.

2. Organisation du backlog

2.1. Structure des issues

Chaque issue est caractérisée par :

- Un **titre** descriptif de la fonctionnalité ou tâche
- Un ou plusieurs **labels** pour la catégorisation
- Un **numéro** unique servant de référence pour la branche et la PR

2.2. Labels et Epics

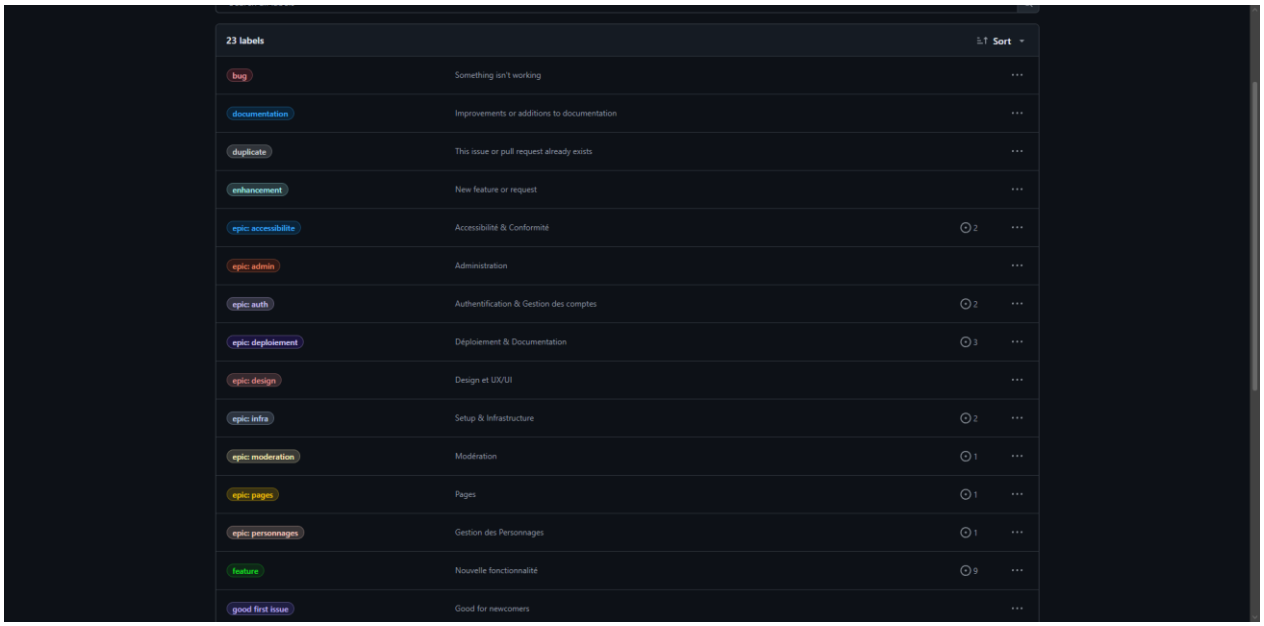
Le backlog est organisé en **9 epics** (domaines fonctionnels) identifiés par des labels de couleur :

Epic	Nombre d'issues	Description
epic: personnages	13	Création, modification, équipement, actions sur personnages
epic: infra	12	Setup projet, Docker, CI/CD, outils
epic: auth	9	Inscription, connexion, gestion de mots de passe, rôles
epic: deploiement	9	Déploiement, documentation, scripts SQL
epic: pages	8	Pages publiques (accueil, galerie, détail, contact, etc.)
epic: design	7	Charte graphique, wireframes, mockups, design system
epic: modération	5	Modération personnages, commentaires, utilisateurs, articles
epic: admin	5	Dashboard admin, gestion employés, logs d'activité
epic: accessibilité	2	Conformité RGAA et RGPD

Des labels complémentaires qualifient la nature des issues :

- **feature** (63 issues) : nouvelles fonctionnalités
- **task** (6 issues) : tâches techniques

- **refactoring** (1 issue) : migration vers DDD
- **priority: high** (3 issues) : priorité élevée



2.3. Chiffres clés

Indicateur	Valeur
Issues créées	71
Pull requests mergées	55
Commits totaux	~180
Durée du projet	~2,5 mois (décembre 2025 – février 2026)

---

3. Workflow Git

3.1. Stratégie de branches (GitFlow)

Le projet suit le modèle **GitFlow** avec 3 niveaux de branches :

```
main          ← Production stable
  └── develop   ← Intégration des fonctionnalités
        └── XX-nom-de-la-fonctionnalite   ← Branche par issue
```

- **main** : code en production, protégée (seule `develop` peut y être mergée)
- **develop** : branche d'intégration, reçoit toutes les PR
- ***-*** : branches éphémères nommées `{numéro}-{slug}` (ex: `73-gestion-des-utilisateurs-employé`)

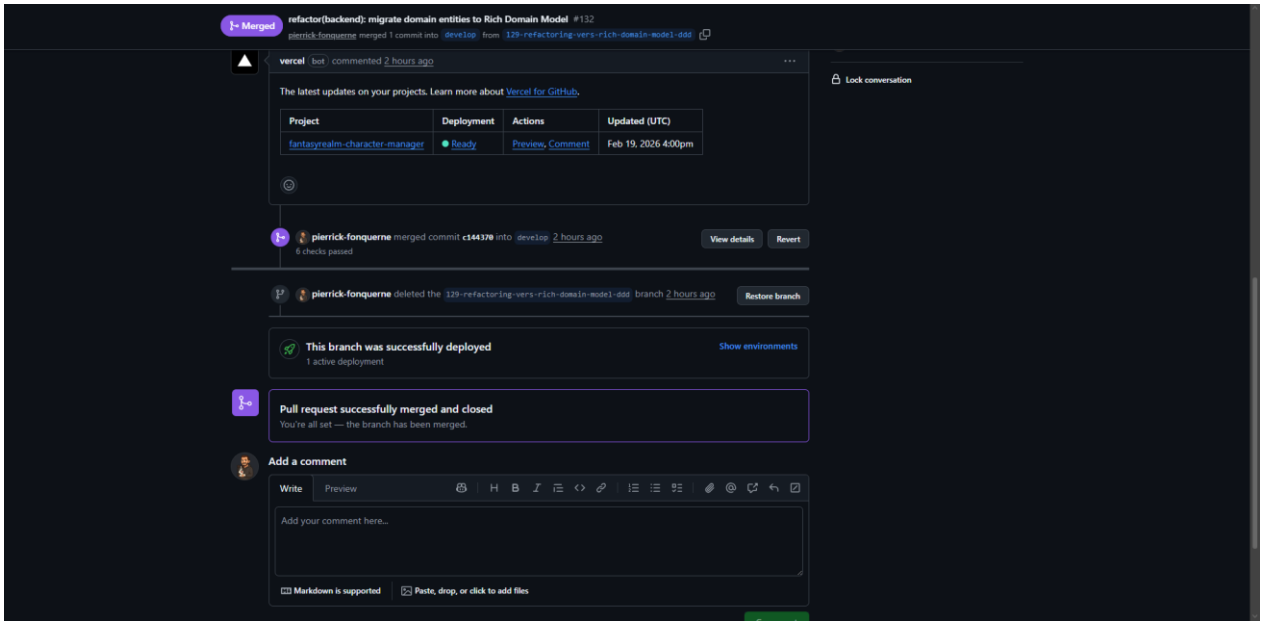
3.2. Convention de nommage

Élément	Convention	Exemple
Branche	`{numéro}-{slug}`	`74-dashboard-administrateur`
Commit	`{type}({scope}): {description}`	`feat(backend): add employee management system`
PR	Titre descriptif, merge vers `develop`	`feat: add admin dashboard with platform stats (#74)`

Types de commits utilisés : `feat`, `fix`, `refactor`, `test`, `ci`, `docs`

3.3. Cycle de vie d'une fonctionnalité

1. **Créer l'issue** sur GitHub avec les labels appropriés
2. **Créer la branche** depuis `develop` : `git checkout -b 74-dashboard-administrateur`
3. **Développer** la fonctionnalité (backend puis frontend)
4. **Commiter** séparément backend et frontend : `feat(backend):` puis `feat(frontend):`
5. **Pousser** et créer une **pull request** vers `develop`
6. **La CI vérifie** automatiquement : build, tests, lint, type-check
7. **Merger** la PR après validation
8. **Fermer l'issue** automatiquement



4. Intégration continue (CI/CD)

4.1. Pipeline CI

Chaque push et pull request déclenche un pipeline **GitHub Actions** (`ci.yml`) qui :

1. **Détecte les changements** (backend et/ou frontend modifiés)
2. **Backend** : restore → build → test (311 tests unitaires + 102 tests d'intégration)
3. **Frontend** : npm ci → lint → type-check → build → test
4. **Vérification finale** : échoue si un des jobs a échoué

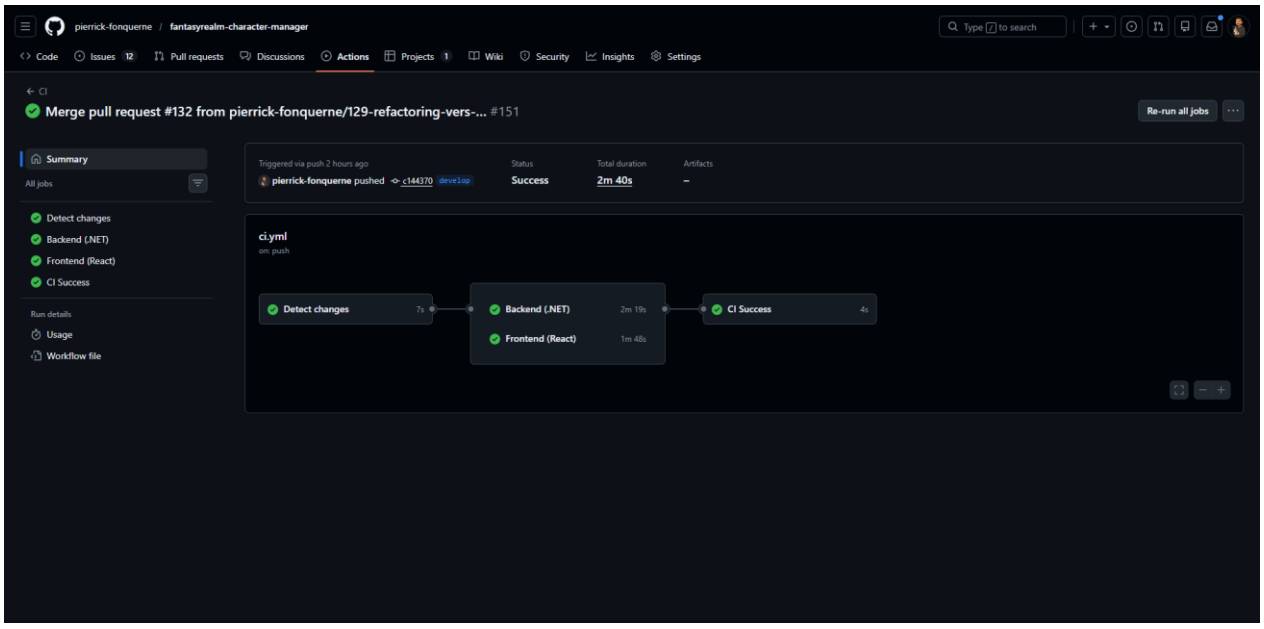
Cette détection des changements évite de rebuilder les deux projets à chaque commit, optimisant le temps de CI.

4.2. Protection des branches

- **main** : seule la branche `develop` peut y être mergée (vérifié par `pr-main.yml`)
- **develop** : les PR doivent passer la CI avant merge

4.3. Déploiement automatique

- **Railway** : déploiement automatique du backend à chaque push sur la branche configurée
- **Vercel** : déploiement automatique du frontend, avec preview deployments pour chaque branche



---

5. Découpage fonctionnel

Le développement a suivi un ordre logique, des fondations vers les fonctionnalités avancées :

Phase 1 — Fondations (décembre 2025)

- Setup monorepo, backend .NET, frontend React
- Configuration des bases de données (PostgreSQL, MongoDB)
- CI/CD GitHub Actions
- Docker Compose pour le développement local
- Service d'email centralisé (Brevo)
- Charte graphique, wireframes, mockups, design system

Phase 2 — Authentification (janvier 2026)

- Inscription utilisateur
- Connexion et gestion JWT
- Mot de passe oublié
- Changement de mot de passe obligatoire
- Gestion des rôles (User, Employee, Admin)

Phase 3 — Fonctionnalités utilisateur (février 2026)

- Dashboard utilisateur
- Création et modification de personnage
- Actions sur personnage (partager, dupliquer, supprimer)
- Galerie publique des personnages
- Système d'avis sur les personnages
- Pages publiques (contact, mentions légales, CGV)

Phase 4 — Modération et administration (février 2026)

- Dashboard employé
- Modération des personnages et commentaires
- Gestion des utilisateurs (suspension, réactivation, suppression)
- Gestion des articles (CRUD employé)
- Dashboard administrateur (statistiques)
- Gestion des employés
- Logs d'activité (MongoDB)
- Refactoring vers Rich Domain Model (DDD)

Phase 5 — Finalisation (février 2026)

- Scripts SQL consolidés
- Documentation technique
- Préparation des livrables ECF

---

6. Suivi et traçabilité

6.1. GitHub Projects

Un board Kanban a été mis en place avec les colonnes :

- **Todo** : issues à venir
- **In Progress** : issue en cours de développement
- **Done** : fonctionnalité livrée

6.2. Traçabilité complète

Chaque fonctionnalité est traçable de bout en bout :

```
Issue #74 "Dashboard administrateur"
  → Branche: 74-dashboard-administrateur
    → Commits: feat(backend):... + feat(frontend):...
      → PR #122 → merge dans develop
        → CI verte → déploiement automatique
```

Cette traçabilité permet de vérifier à tout moment l'historique des décisions et le travail réalisé.

---

7. Difficultés rencontrées et solutions

Difficulté	Solution adoptée
Jira non accessible publiquement (plan gratuit)	Migration complète vers GitHub Issues + Projects
Railway bloque les ports SMTP	Adoption de Brevo API (HTTP) pour les emails transactionnels
Tests d'intégration avec InMemoryDatabase peu fiables	Migration vers Testcontainers (PostgreSQL réel dans Docker)
Modèle anémique limitant la logique métier	Refactoring vers Rich Domain Model (DDD) avec aggregates et factory methods

---

8. Bilan

Le projet a été développé sur **~2,5 mois** avec un découpage en **71 issues** réparties en **9 epics**. L'approche par fonctionnalité avec GitFlow a permis de livrer des incréments testables et de maintenir un historique clair du travail réalisé.

Les outils GitHub (Issues, Projects, Actions) offrent une solution intégrée et publique pour le suivi de projet, la CI/CD et la traçabilité du code.