19 DÉCEMBRE 2025

UNIVERSITÉ LUMIÈRE LYON 2 PROGRAMMATION DE SPÉCIALITÉ - PYTHON 2025-2026

PLANNING POKER

Enseignant: Valentin Lachand-Pascal

Dépôt GitHub :

https://github.com/MarteOued/planning-poker

Réalisé par : Martine OUEDRAOGO

& NDIAYE Aida

Table des matières

1. Introduction

Imaginez un outil qui permet à toute votre équipe de projet de s'aligner rapidement sur la complexité des tâches. C'est exactement ce que propose notre application Planning Poker : une plateforme web collaborative en temps réel, spécialement conçue pour les équipes agiles.

Grâce à cette solution, les développeurs peuvent estimer ensemble les fonctionnalités du backlog en utilisant la méthode du Planning Poker, mais sans avoir besoin de cartes physiques. Tout se fait en ligne, de façon fluide et interactive.

Problématique

Dans un contexte où le travail à distance est de plus en plus répandu, les équipes rencontrent des difficultés à organiser des sessions d'estimation efficaces et synchronisées. Les outils existants sont parfois peu adaptés, manquent de fluidité en temps réel ou ne proposent pas une gestion structurée des sessions et des votes. Ce projet vise donc à répondre à cette problématique en proposant une solution simple, intuitive et accessible à distance.

Objectifs du projet

Notre application a pour objectif de permettre la création et la gestion de sessions de Planning Poker à distance. Nous avons mis en place une communication en temps réel entre les participants grâce à l'utilisation des WebSockets afin de garantir des échanges fluides et synchronisés.

Pour valider les estimations, nous proposons deux approches :

- le mode Strict, qui nécessite l'unanimité des participants,
- le mode Moyenne, qui calcule une estimation à partir des votes lorsque le consensus n'est pas atteint.

L'application permet aussi d'importer et d'exporter des backlogs, ainsi que de sauvegarder et de reprendre une session en cours. Enfin, nous accordons une attention particulière à la qualité du code, à sa fiabilité et à sa maintenabilité, en nous appuyant sur des tests automatisés et une documentation détaillée.

Auteurs

Ce projet a été mené en binôme par Ouedraogo Martine et Ndiaye Aida. Nous avons adopté une organisation de travail en pair programming, avec une répartition claire de nos responsabilités respectives. Cette collaboration nous a permis d'améliorer la qualité du code, de progresser ensemble et d'avancer régulièrement tout au long du développement.

Martine s'est principalement concentrée sur le développement du backend avec Node.js et Express. Elle a également mis en place la communication en temps réel via Socket.io, conçu et exécuté les tests unitaires et d'intégration, et configuré l'intégration continue avec GitHub Actions.

De son côté, **Aida** a pris en charge le développement de l'interface frontend avec React 18. Elle a conçu une interface utilisateur moderne, intuitive et responsive grâce à Tailwind CSS, et a rédigé la documentation utilisateur et technique du projet.

Les choix techniques et les décisions importantes ont été discutés ensemble, ce qui a permis d'assurer la cohérence de l'application et une bonne coordination entre les différentes parties du système.

2. Justification des choix techniques

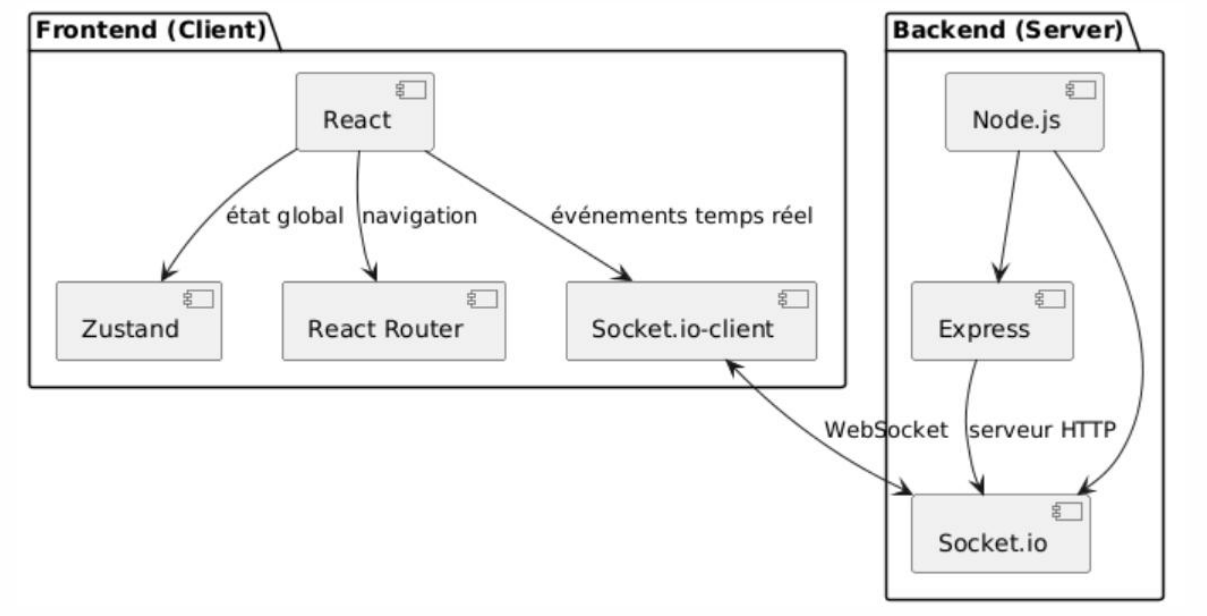
Dans cette partie nous présentions les choix techniques réalisés pour le projet ainsi que l'architecture logicielle adoptée. L'objectif n'est pas seulement de décrire les technologies utilisées, mais d'expliquer en quoi elles sont adaptées aux besoins d'une application collaborative de Planning Poker en temps réel.

2.1 Architecture client-serveur

Nous avons opté pour une architecture client-serveur séparée. Ce choix permet de distinguer les responsabilités entre les différentes couches du système. Le backend est chargé de la logique métier, de la gestion des sessions et de la communication en temps réel, tandis que le frontend se concentre sur l'interface utilisateur et l'expérience interactive.

Cette architecture présente plusieurs avantages. Elle facilite la maintenance du projet, car chaque partie peut évoluer indépendamment. Elle améliore également l'évolutivité, le backend pouvant servir plusieurs types de clients (web, mobile ou autre) sans modification majeure.

Enfin, elle est particulièrement adaptée aux applications collaboratives, car elle centralise la logique métier tout en assurant une synchronisation cohérente entre les utilisateurs.



Communication entre le frontend et le backend

La communication entre le frontend et le backend repose principalement sur une connexion temps réel basée sur :

Connexion WebSocket (Socket.io)

- Le frontend se connecte au backend via socket.io-client.
- Les événements (vote, révélation, timer, etc.) sont envoyés et reçus en temps réel.

Lorsqu'un utilisateur effectue une action (création ou rejoint d'une session, soumission d'un vote, changement d'état de la partie), cette information est immédiatement transmise au serveur. Celui-ci traite l'événement, applique les règles métier, puis diffuse la mise à jour à l'ensemble des participants concernés. Ce mécanisme garantit une synchronisation instantanée des données et une expérience utilisateur fluide, sans rechargement de page.

2.2 Choix des technologies backend

- **Node.js avec Express**

Pourquoi nous avons opter pour Node.js avec Express ? Plusieurs raisons bien concrètes justifient ce choix. D'abord, Node.js se distingue par ses performances, grâce à son architecture

non-bloquante un vrai plus pour les applications en temps réel qui doivent gérer de nombreuses connexions en parallèle. Ensuite, il y a l'écosystème npm, qui regorge de bibliothèques fiables et permet d'aller plus vite dans le développement. Autre avantage : utiliser JavaScript à la fois côté client et serveur simplifie le partage de code et aide toute l'équipe à mieux saisir le projet dans son ensemble. Et pour finir, la communauté autour de Node.js est très dynamique, ce qui signifie qu'on trouve facilement de la documentation et un bon support en cas de besoin.

- **Socket.io**

Pour assurer une communication en temps réel entre les joueurs et le serveur, nous avons opté pour Socket.io. Cette bibliothèque permet d'échanger des données instantanément dans les deux sens, ce qui est indispensable pour synchroniser les votes au sein d'une même session de jeu. Un autre avantage pratique : Socket.io gère automatiquement les reconnexions si la connexion est coupée, évitant ainsi des interruptions frustrantes pour les utilisateurs. Et si jamais WebSocket n'est pas pris en charge par un navigateur ou un réseau, la solution bascule sans accroc sur d'autres méthodes, comme le long-polling. Enfin, grâce à son système de rooms intégré, on peut facilement organiser plusieurs sessions en parallèle, chacune fonctionnant de manière isolée.

- **Jest pour les tests**

Pour les tests, on s'est tourné vers Jest, et ce n'est pas un hasard. C'est un framework complet qui regroupe tests unitaires, tests d'intégration et outils de mocking dans un seul outil. Conçu par Facebook pour JavaScript, il s'intègre parfaitement avec Node.js, ce qui rend le développement vraiment agréable. En plus, il génère automatiquement des rapports de couverture de code détaillés, ce qui aide à garder un œil sur la qualité des tests. Et son système de snapshots est un vrai plus : il détecte tout seul les changements inattendus et simplifie énormément les tests de régression.

2.3 Choix des technologies frontend

Dans le cadre de ce projet, nous avons utilisé plusieurs outils et bibliothèques afin de construire une application moderne, performante et maintenable. Parmi l'ensemble des technologies employées, trois ont joué un rôle central dans le développement du frontend.

React 18

Quand on parle de développement frontend aujourd'hui, React 18 s'impose presque naturellement. Sa logique de composants réutilisables rend la maintenance et l'évolution des interfaces bien plus fluide. Grâce au Virtual DOM, les performances sont optimisées, car on

évite de manipuler le DOM réel à tout bout de champ. Et puis, les Hooks, arrivés avec les versions récentes, ont vraiment simplifié la gestion de l'état et des effets de bord, sans avoir à passer par des classes. Autre atout majeur : son écosystème est ultra-mature, avec une bibliothèque pour à peu près tous les besoins. Et comme la communauté est très active, on bénéficie d'un support solide et d'une documentation vraiment bien faite.

React Router

Pour la navigation de l'application, nous avons opté pour React Router. Cette bibliothèque nous permet de créer une application à page unique, où l'utilisateur navigue sans subir de rechargements de page. C'est vraiment ce qui fait la différence pour une expérience fluide et agréable. D'ailleurs, React Router est aujourd'hui la solution la plus utilisée pour le routing dans l'écosystème React, ce qui nous assure une bonne pérennité et un support solide. Et puis, son API est assez simple à prendre en main, même pour des structures de navigation un peu complexes.

Tailwind CSS

Côté design, nous avons choisi Tailwind CSS. Ce qui est pratique avec cette approche, c'est qu'on peut styliser directement dans le JSX grâce à ses classes utilitaires, sans avoir à jongler entre les fichiers. Ça accélère vraiment le développement. Tailwind nous offre aussi un système de design cohérent, ce qui garantit une harmonie visuelle sur l'ensemble de l'application. Et niveau performance, c'est intéressant : le framework ne génère que le CSS effectivement utilisé, ce qui réduit le poids des fichiers en production. Créer des interfaces responsives devient aussi plus simple grâce aux classes adaptées à chaque breakpoint. Enfin, on peut facilement personnaliser le tout via un fichier de configuration central, pour l'adapter parfaitement aux besoins du projet.

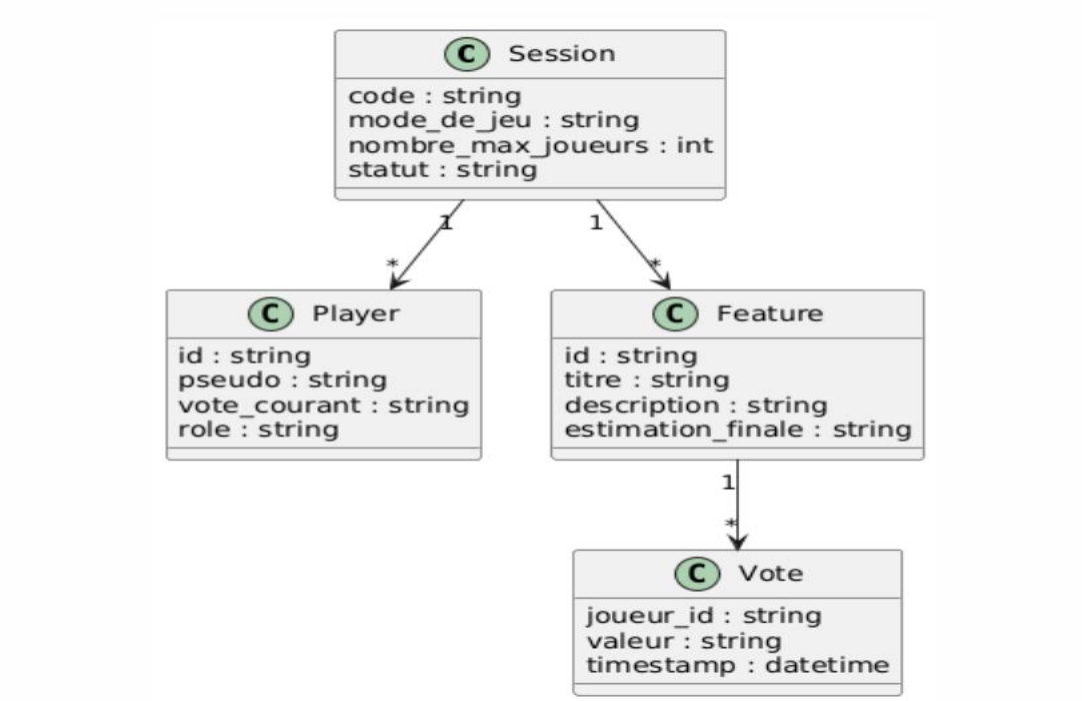
En complément de ces technologies principales, plusieurs outils ont été utilisés pour améliorer l’architecture et l’expérience utilisateur :

- **Vite** : outil de build rapide et moderne, offrant des temps de démarrage et de compilation très courts.
- **Zustand** : bibliothèque légère pour la gestion de l’état global de l’application.
- **Framer Motion** : gestion des animations et transitions pour une interface plus dynamique.

- **Socket.io-client** : communication temps réel avec le backend pour synchroniser les votes et les actions des joueurs.
- **Lucide React** : bibliothèque d'icônes simple et cohérente visuellement.

2.4 Modélisation objet

La modélisation du domaine repose sur plusieurs classes métier principales : Session, Player, Feature et Vote. La classe Session centralise la gestion des participants, des fonctionnalités à estimer et des paramètres de jeu. La classe Player représente un utilisateur participant à une session. La classe Feature correspond à une fonctionnalité du backlog, tandis que la classe Vote stocke une estimation fournie par un joueur.



Cette modélisation orientée objet favorise l'encapsulation des données et de la logique métier, améliore la maintenabilité du code et permet de tester chaque composant de manière indépendante.

3. Architecture et structure du projet

Dans cette section, nous décrivons l'organisation globale du projet ainsi que l'architecture mise en place côté backend et frontend. L'objectif est de montrer comment la structure du code contribue à la lisibilité, la maintenabilité et l'évolutivité de l'application.

3.1 Structure des répertoires

Pour que le projet reste facile à comprendre, à maintenir et à faire évoluer, nous l'avons structuré de façon claire et modulaire.

À la racine, vous trouverez le dossier `.github/workflows`, qui gère l'intégration et le déploiement continus (CI/CD) grâce à GitHub Actions. Concrètement, cela signifie que les tests et les builds se lancent automatiquement à chaque modification du code.

- Le dossier server regroupe l'ensemble du backend développé avec Node.js. Il contient :
  - un sous-dossier src, organisé par responsabilités (modèles métier, logique applicative, gestion des communications temps réel) ;
  - un dossier tests, dédié aux tests unitaires et d'intégration réalisés avec Jest ;
  - un dossier docs, qui accueille la documentation technique générée automatiquement.
- Côté client, c'est le frontend React. Nous y avons rangé les composants réutilisables, les pages principales, la gestion de l'état global et la communication via Socket.io. Un dossier `public` sert à héberger les ressources statiques, comme les images des cartes de vote.

Enfin, le dossier data, placé à la racine, nous permet de gérer les données persistantes : sauvegardes automatiques des sessions et exports des résultats, par exemple. Cette organisation assure une séparation nette entre le frontend, le backend, les outils d'automatisation et les données.

3.2 Architecture backend

Concernant le backend, on a opté pour une architecture de type MVC, mais adaptée pour gérer des interactions en temps réel.

- Les Models, comme Session, Player ou Feature, représentent les entités métier de l'application. Ils définissent la structure des données et les règles qui les régissent.
- La logique métier principale est porté par les Managers. Ils orchestrent les interactions entre les différents modèles, un peu à la manière des contrôleurs dans une architecture MVC plus traditionnelle.
- Enfin, les Socket handlers, basés sur Socket.io, prennent en charge la communication en temps réel. Ils reçoivent les actions des clients et diffusent les mises à jour aux utilisateurs concernés.

3.3 Architecture frontend

Pour faciliter la maintenance et la collaboration, nous avons opté pour une architecture frontend modulaire. L'idée, c'est de rendre le code plus lisible et de pouvoir réutiliser facilement les composants.

Voici comment c'est organisé :

- Le dossier `components` est le cœur de notre bibliothèque réutilisable.
- Dans `ui`, on trouve tous les éléments d'interface de base : boutons, cartes, champs de formulaire.
- Le dossier `game` regroupe, lui, tout ce qui est spécifique à notre application : le chat, le minuteur, les cartes de vote, etc.
- Le dossier `pages` correspond simplement aux différents écrans de l'appli, ceux auxquels on accède via la navigation.
- La gestion de l'état global est centralisée dans `stores`, grâce à la bibliothèque Zustand.
- Les fonctions utilitaires, comme le client pour la communication en temps réel avec Socket.io, sont dans `utils`.
- Enfin, `App.jsx` et `main.jsx` sont les points d'entrée : le premier définit la structure générale, le second lance l'application.

Cette séparation claire entre l'interface, la logique métier et l'état nous aide à garder un code propre et organisé.

3.4 Communication temps réel

Imaginez une conversation fluide où chaque mot est entendu au moment même où il est prononcé. C'est exactement comme ça que fonctionne notre système de communication en temps réel. Lors d'une session de Planning Poker, tous les participants restent parfaitement synchronisés, sans jamais avoir à rafraîchir leur écran.

Dès qu'un utilisateur pose une action, qu'il crée une session, rejoigne une partie, vote, envoie un message ou met la session en pause , cette information est envoyée au serveur en un clin d'œil. Le serveur la traite et la retransmet aussitôt à toutes les personnes concernées.
Concrètement, quand une session est lancée, les joueurs reçoivent instantanément les infos pour se connecter. Si quelqu'un rejoint la partie, les autres en sont informés sans délai. Chaque vote est pris en compte à la seconde, et le passage d'un tour à l'autre se fait de manière parfaitement synchronisée pour tout le monde.

Une fois tous les votes recueillis, le serveur calcule automatiquement le résultat – selon le mode choisi (strict ou moyenne) et l'affiche immédiatement pour l'ensemble des participants. Les fonctionnalités de pause, de reprise et de sauvegarde suivent la même logique, pour garantir une expérience de jeu fluide et sans accroc.

Grâce à cette architecture basée sur les événements, l'expérience collaborative se rapproche d'une interaction en face à face : chaque action est visible par tous en temps réel, ce qui rend les sessions dynamiques, réactives et cohérentes pour chaque joueur.

3.5 Gestion des données

Les données de l'application sont stockées sous forme de fichiers JSON. Les fichiers de backlog contiennent la liste des fonctionnalités à estimer, tandis que les fichiers de sauvegarde enregistrent l'état complet d'une session (participants, votes, mode de calcul). Ce choix permet une solution simple, lisible et facilement exportable, adaptée à un projet collaboratif de taille moyenne.

```
{
  "features": [
    {
      "id": "feat-001",
      "name": "Authentification utilisateur",
      "description": "Permettre aux utilisateurs de se connecter avec email et mot de basse"
    },
    {
      "id": "feat-002",
      "name": "Gestion des profils",
      "description": "Les utilisateurs peuvent modifier leur profil et leurs informations"
    }


  ]
}
```

4 Manuel utilisateur et fonctionnalités

4.1 Installation et lancement

Le projet est disponible sur GitHub à l'adresse suivante :

https://github.com/MarteOued/planning-poker

Prérequis

- Node.js (version 18 ou supérieure)

- npm

Étapes d’installation

1. Cloner le dépôt :

- git clone https://github.com/MarteOued/planning-poker.git
- cd planning-poker
2. Installer les dépendances du backend :
- cd server
- npm install
3. Lancer le serveur backend
- npm run dev
4. Installer les dépendances du frontend :
- cd ../client
- npm install
5. Lancer l’application frontend :
- npm run dev

L’application est alors accessible depuis un navigateur web à l’adresse :
http://localhost:5173

4.2 Modes de jeu

Notre application s'adapte à votre façon de travailler en proposant deux approches pour estimer vos tâches.

Mode Strict

Ici, on ne valide une estimation que lorsque tout le monde est d'accord. Si les votes divergent, un nouveau tour est automatiquement lancé. Ce mode pousse vraiment à la discussion et à trouver un terrain d'entente solide.

Petite précision : quel que soit le mode choisi, le premier tour suit toujours cette règle d'unanimité. C'est un bon moyen de lancer la conversation et de confronter les points de vue dès le départ.

Mode Moyenne

Parfois, trouver un consensus pur et simple prend trop de temps. Dans ce mode, si l'unanimité n'est pas là après le premier tour, on calcule simplement la moyenne des votes. Pour garder

une certaine logique dans les chiffres, ce résultat est ensuite arrondi à la valeur la plus proche dans la suite de Fibonacci. L'idée ? Gagner en fluidité sans perdre la cohérence d'équipe.

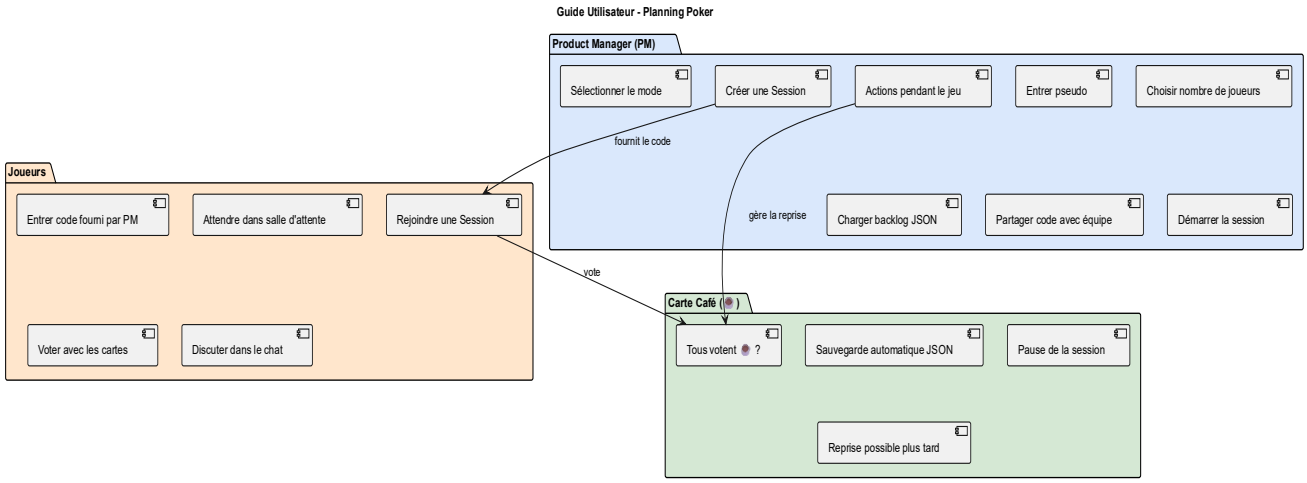
4.3 Interface utilisateur et déroulement d'une session

L'interface utilisateur est volontairement simple et intuitive afin de faciliter la prise en main, même pour des utilisateurs non techniques.

Une session de Planning Poker suit le déroulement suivant :

- Depuis la page d'accueil, un Product Manager crée une nouvelle session.
- Les joueurs rejoignent la session à l'aide du code généré.
- Le Product Manager charge un backlog au format JSON.
- Une fonctionnalité est sélectionnée et un tour de vote est lancé.
- Chaque joueur choisit une carte d'estimation.
- Une fois tous les votes reçus, le résultat est affiché selon le mode de jeu choisi.
- La session peut être mise en pause, reprise ou sauvegardée à tout moment.

Le diagramme ci-dessous représente chaque acteur du Planning Poker dans des boîtes colorées pour plus de clarté. Les actions ou étapes de chaque acteur sont listées à l'intérieur de ces boîtes. Les flèches indiquent les interactions et le flux d'information entre le Product Manager, les joueurs et la carte café.

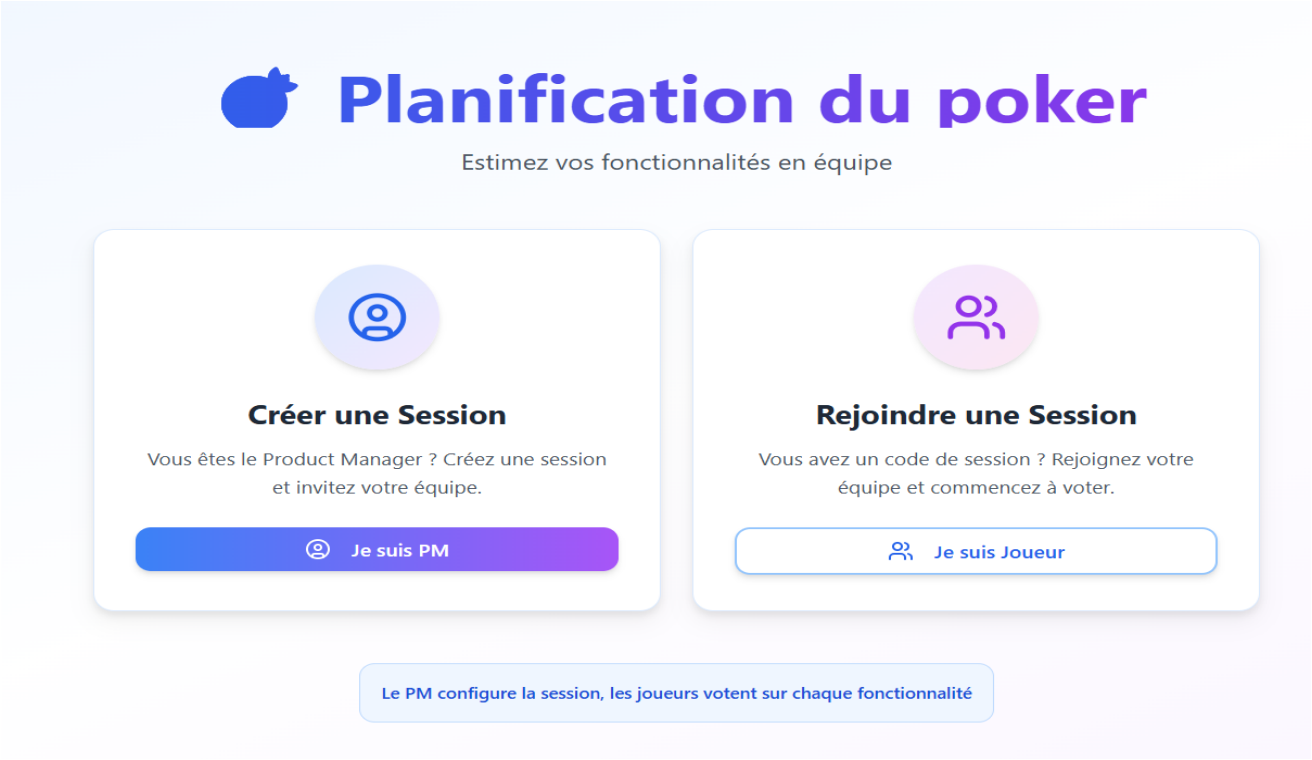


✓ **Résultats**

Des captures d'écran illustrent les principales étapes : page d'accueil, salle d'attente, interface de vote et affichage des résultats.

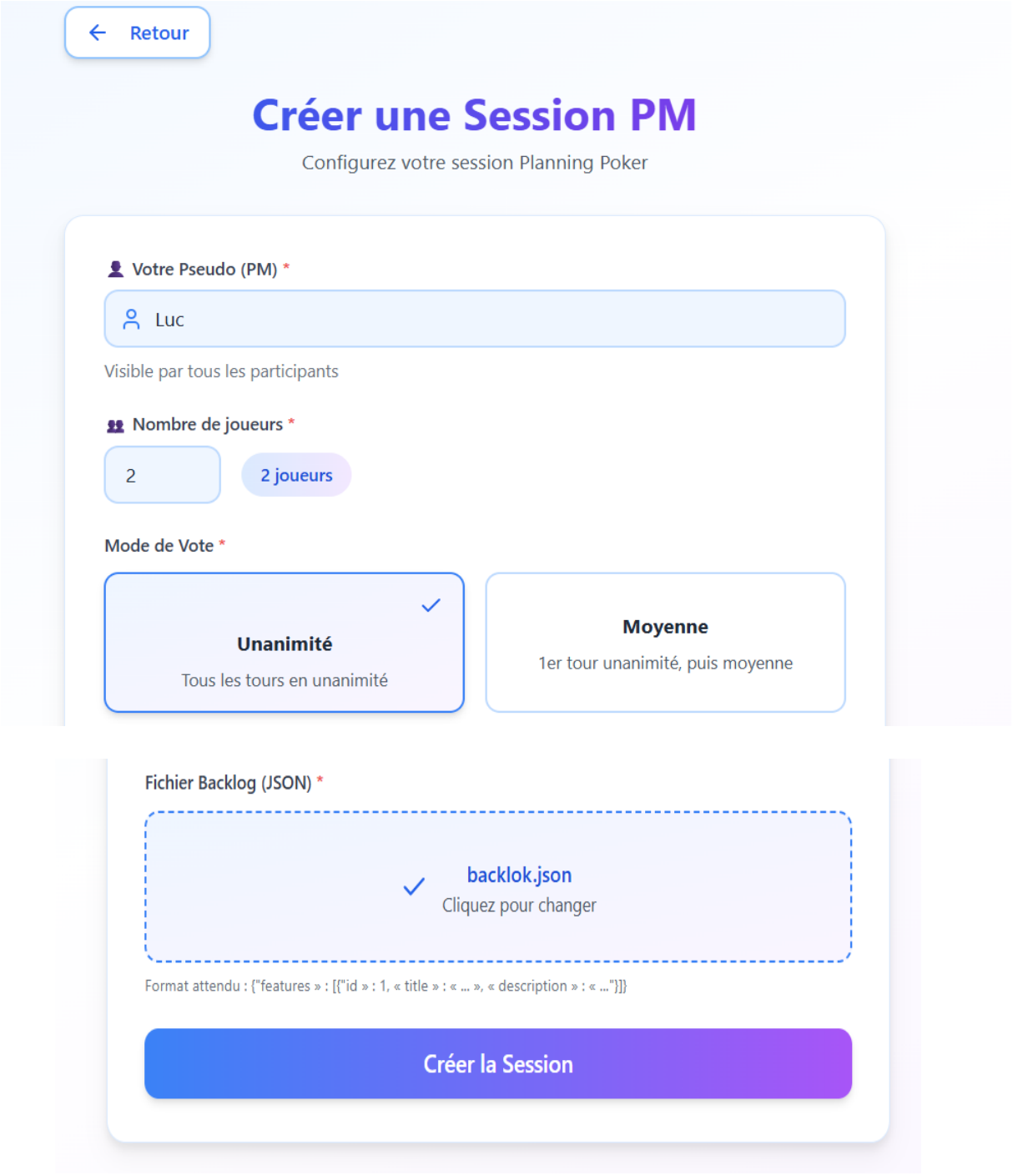
Page d'accueil :

- Permet au PM de créer une session et aux joueurs de rejoindre une session existante.
- Capture : boutons « Je suis PM » et « Je suis Joueur ».



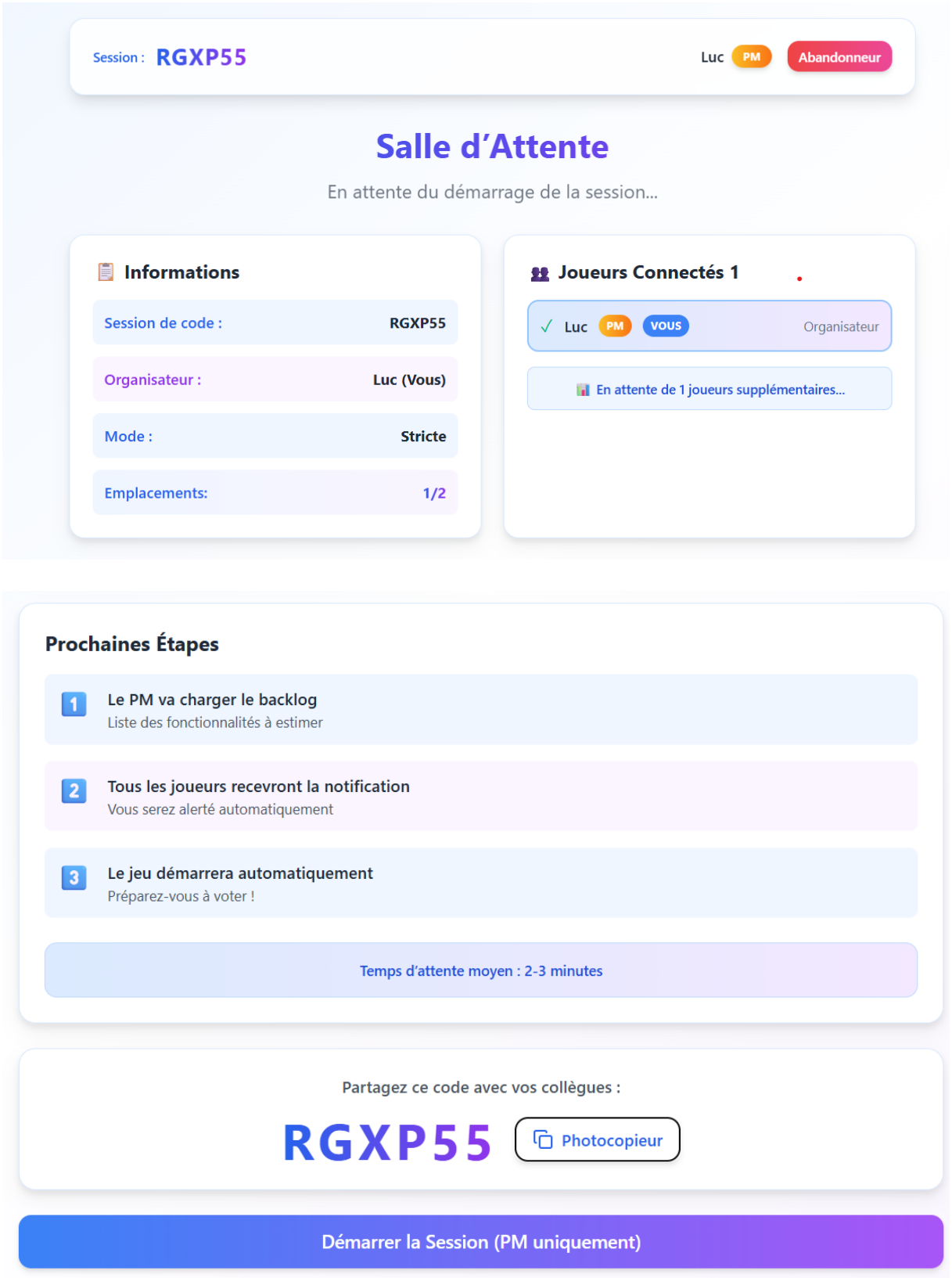
Création de session PM :

- Le PM saisit son pseudo, le nombre de joueurs, le mode de vote, et importe un backlog JSON.
- Capture : formulaire de configuration de session.



Salle d'attente :

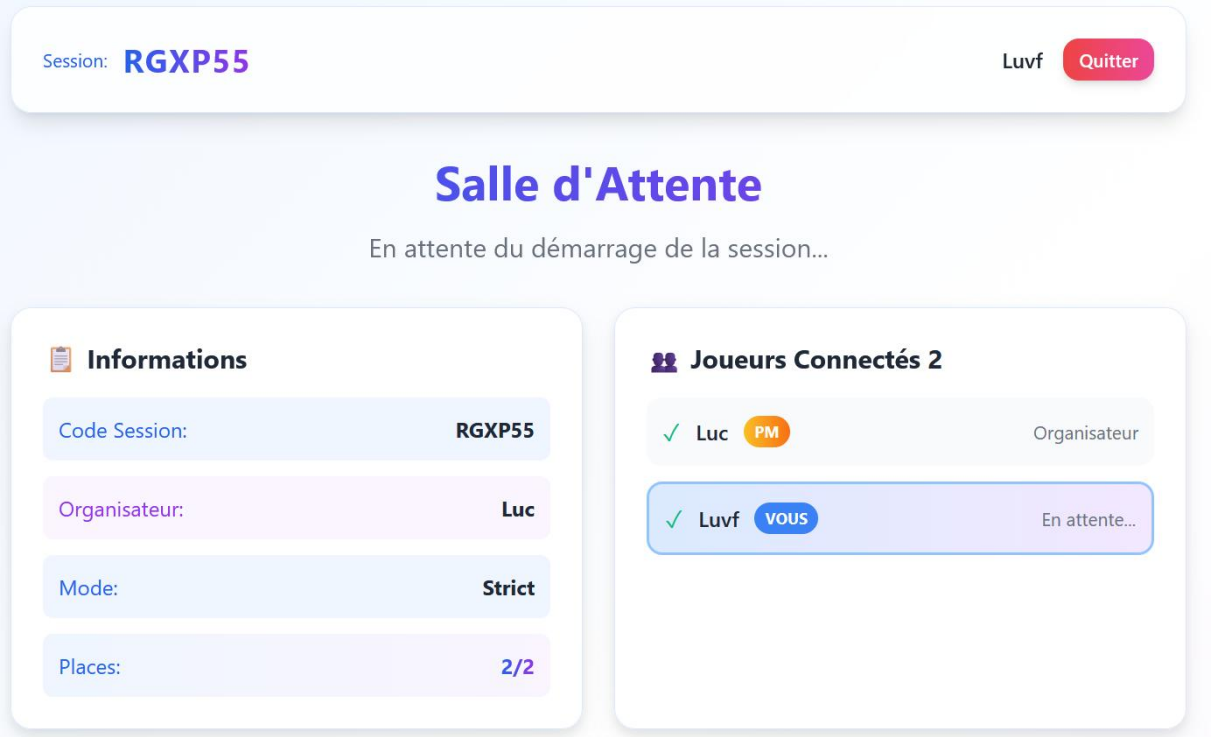
- Affiche le code de session, les joueurs connectés et le mode de vote.
- Capture : tableau de la salle d'attente avec informations PM et joueurs.



Salle d'attente joueur

- Affiche les informations de la session (code, mode, joueurs connectés).

- Accessible par le PM après la création de la session et par les joueurs après saisie du code.



Salle de jeu

- Lien 1 – Accès à la salle de jeu (PM)
  Accessible automatiquement par le PM après le démarrage de la session depuis la salle d'attente.
- Lien 2 – Accès à la salle de jeu (Joueur)
  Accessible pour les joueurs lorsque le PM lance la session.

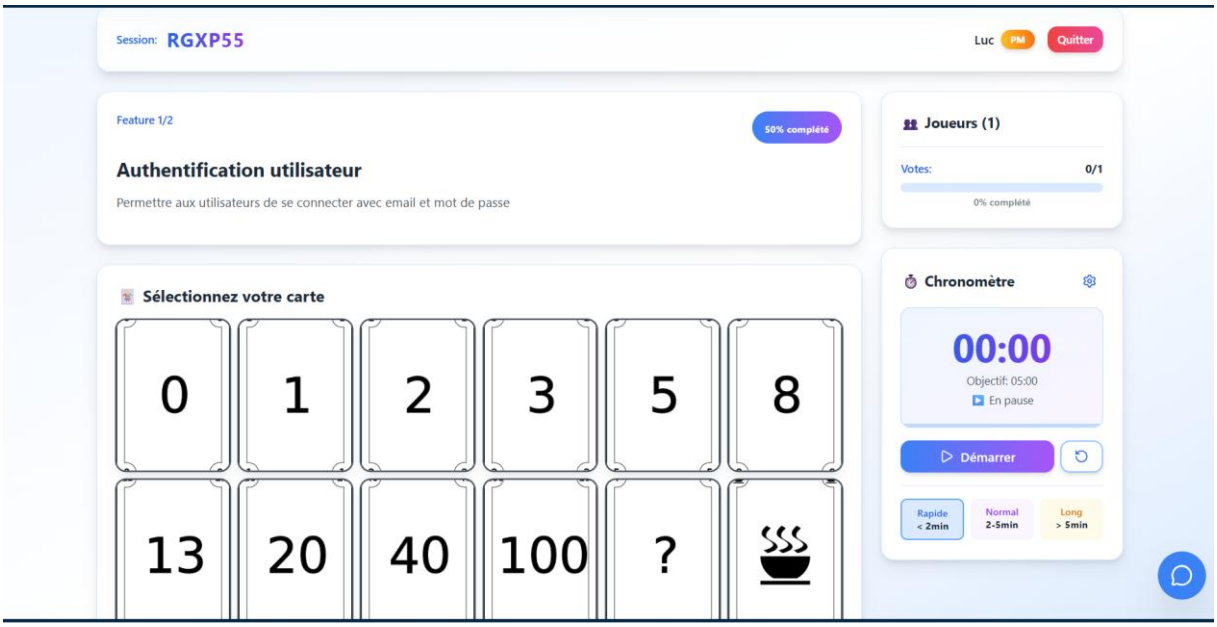
La salle de jeu permet aux joueurs de participer au vote des estimations pour chaque feature du backlog.

Elle comprend :

- la feature en cours avec son titre et sa description
- l'indication de progression (feature X / total)
- les cartes de vote (suite de Fibonacci et options spéciales)
- la liste des joueurs et l'état des votes
- un chronomètre configurable par le PM

- les boutons de contrôle (démarrer, pause, reset)

Chaque joueur sélectionne une carte pour voter. Les votes sont comptabilisés et affichés une fois le vote terminé.



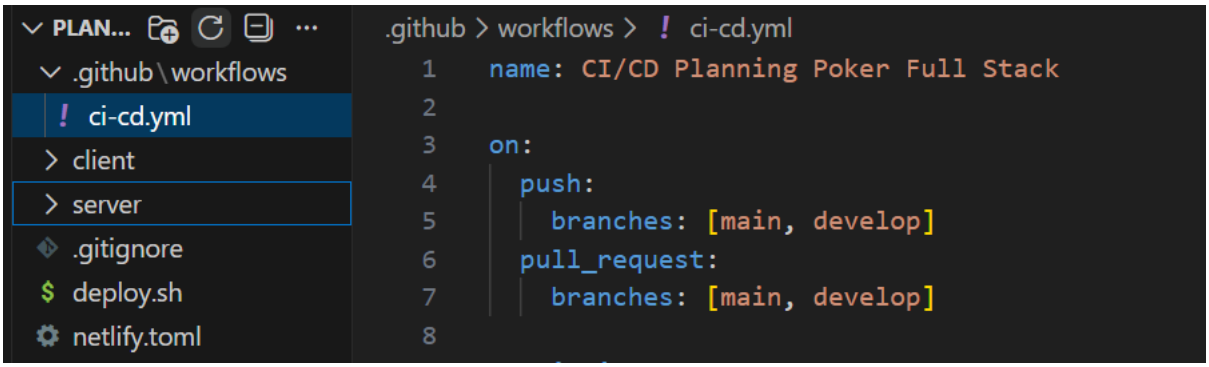
4. Intégration continue

4.1 Configuration GitHub Actions

Nous avons mis en place GitHub Actions afin d'automatiser l'ensemble du processus d'intégration continue (CI) du projet. Cette configuration est définie dans le fichier .github/workflows/ci-cd.yml, situé à la racine du projet.

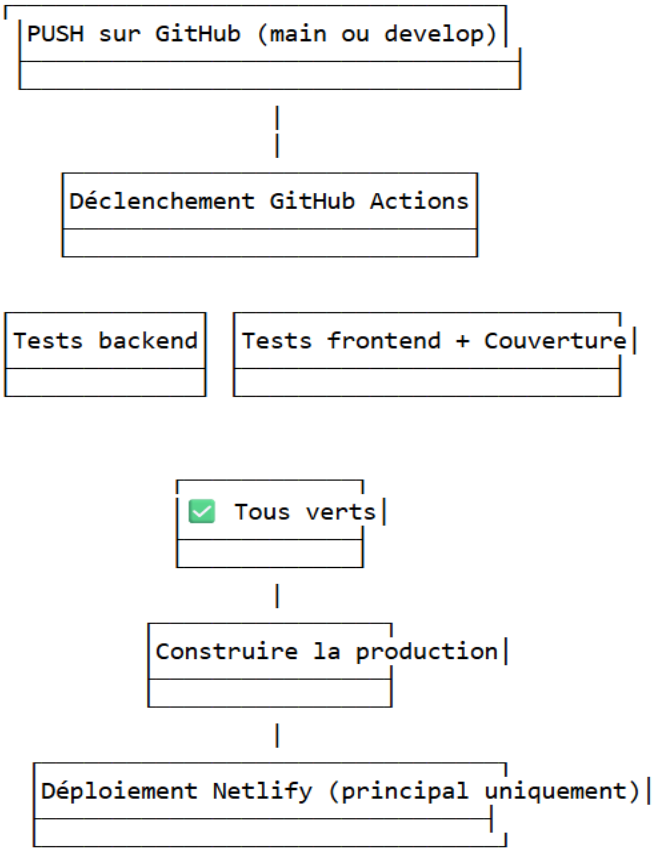
Le pipeline se déclenche automatiquement à chaque push sur les branches main et develop, ainsi que lors de la création d'un pull request vers la branche main. Cette automatisation nous permet de nous assurer que chaque modification du code est correctement vérifiée avant d'être intégrée aux branches principales, ce qui limite les erreurs et les régressions.

4.2 Pipeline CI/CD



Le pipeline d'intégration continue se déroule en plusieurs étapes bien définies :

- Récupération du code source : le code est cloné depuis le dépôt Git grâce à l'action checkout.
- Configuration de l'environnement : nous avons testé avec Node.js, en particulier les versions 18.x et 20.x. Cela nous permet de nous assurer que le code fonctionne bien sur les versions de Node.js encore très répandues en production.
- L'installation des dépendances : on utilise la commande `npm ci`, particulièrement adaptée aux environnements d'intégration continue, car elle garantit une installation reproductible et fiable.

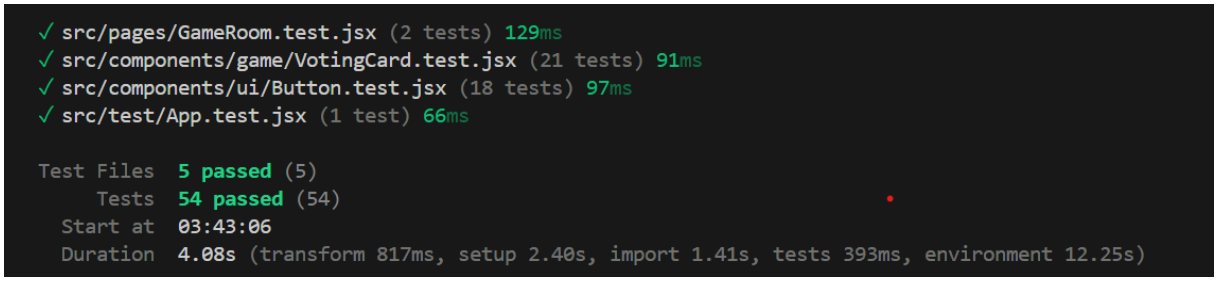


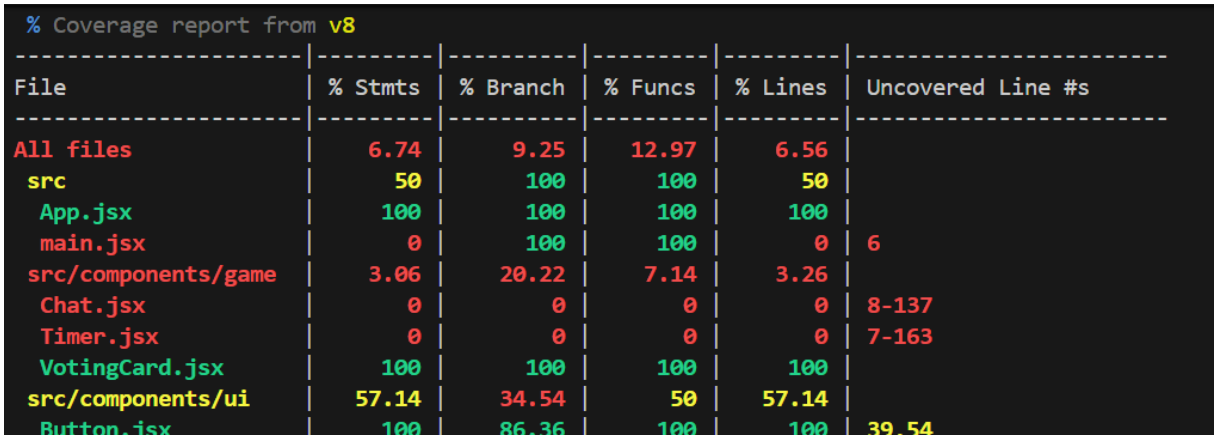
Exécution des tests automatisés

Les tests jouent un rôle central dans notre pipeline.

Pour le backend, le pipeline lance d'abord

- npm test pour lancer l'ensemble des tests unitaires et d'intégration ;
- npm run test:coverage afin de générer un rapport de couverture de code.





Ces tests automatisés valident le bon fonctionnement des composants principaux, des pages et du store de session.

Les 54 tests passent avec succès, ce qui confirme la stabilité globale de l'application. La couverture de code montre que les parties critiques sont bien testées, même si certaines pages ne sont pas encore couvertes, ce qui explique un pourcentage global plus faible.

Voici les différents types de tests que nous avons mis en place :

- Tests unitaires : ils permettent de vérifier le comportement de nos classes métier, comme Session, Player et Feature, de manière isolée.
- Tests d'intégration : ils valident le bon fonctionnement global de composants tels que SessionManager et VoteManager dans des scénarios réalistes.
- Tests des gestionnaires Socket.io : ils assurent que la communication en temps réel entre les participants se déroule sans accroc.

Frontend

Côté frontend, le pipeline exécute aussi des tests automatisés via la commande `npm test`, avec une génération de la couverture de code. Ces tests nous aident à vérifier que les composants React et la logique côté client se comportent comme prévu.

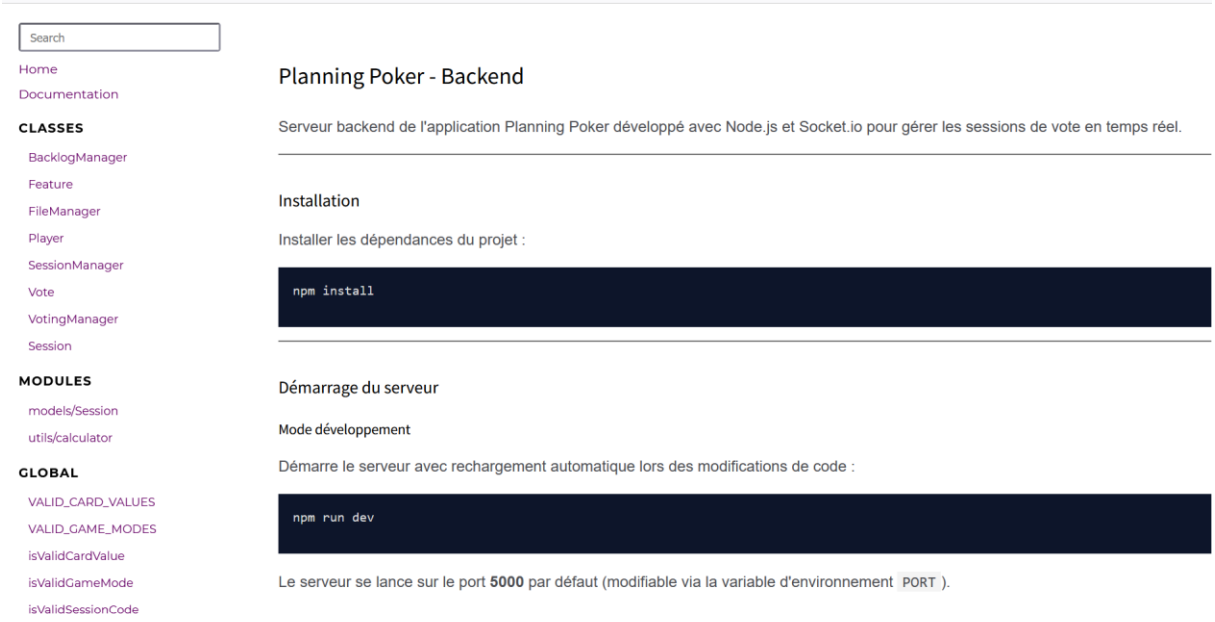
Génération automatique de la documentation

La documentation technique est générée automatiquement via la commande ‘’ npm run docs’’ qui utilise JSDoc. JSDoc analyse tous les commentaires du code source et génère une documentation HTML interactive et professionnelle.

Cette approche présente plusieurs avantages :

- la documentation est toujours à jour et cohérente avec le code ;
- la navigation entre les classes et les méthodes est intuitive ;
- des exemples d’utilisation peuvent être intégrés directement dans le code ;
- la documentation peut être exportée dans différents formats selon les besoins.

La structure de la documentation générée est complète et bien organisée. Elle comprend une page d'accueil présentant une vue d'ensemble du projet, un index listant toutes les classes et modules disponibles, et une documentation détaillée pour chaque classe incluant sa description et son objectif, la liste complète de ses propriétés, la documentation de toutes ses méthodes avec leurs paramètres et valeurs de retour, ainsi que des exemples d'utilisation concrets.



Archivage des artefacts

Enfin, l'archivage des artefacts permet de conserver les rapports de couverture de code et la documentation générée. Ces fichiers peuvent être consultés ultérieurement pour analyser l'évolution de la qualité du code au fil du temps.

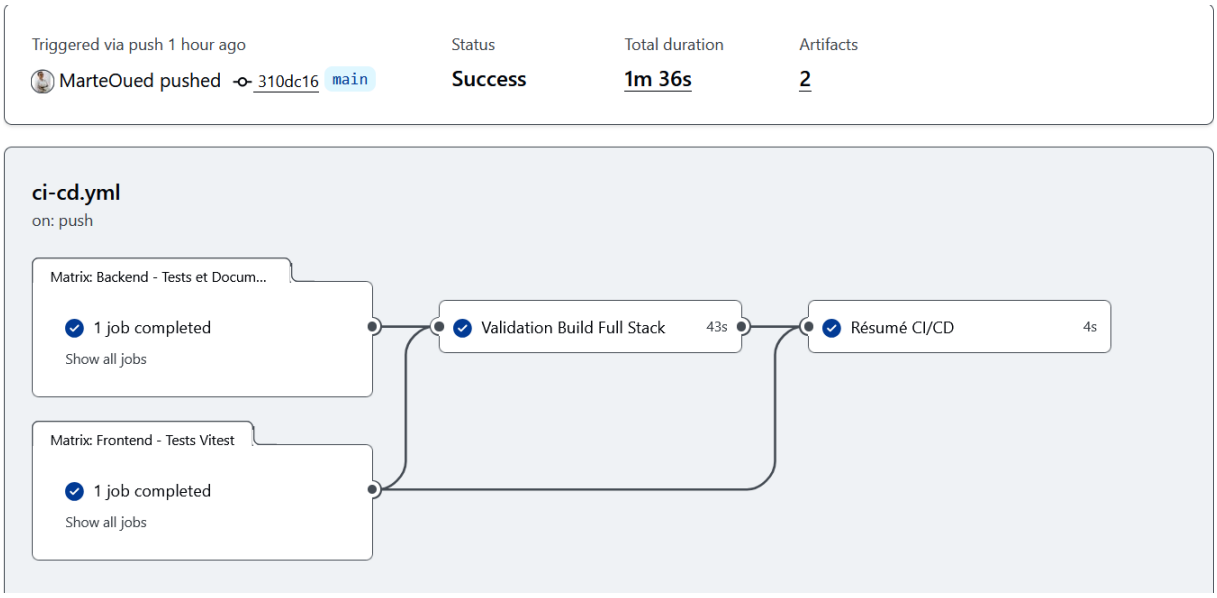
Résultat du pipeline CI/CD

Le pipeline d'intégration continue et de déploiement (CI/CD) est configuré via GitHub Actions. Il valide automatiquement le frontend et le backend à chaque push ou pull request.

L'image ci-dessous illustre l'exécution récente du pipeline :

- Backend – Tests et Documentation : exécute les tests unitaires et génère la documentation côté serveur.
- Frontend – Tests Vitest : exécute les tests unitaires et de composants côté client.
- Validation Build Full Stack : vérifie que l'application complète (frontend + backend) se construit correctement.
- Résumé CI/CD : compile les résultats finaux et fournit un retour global sur l'état du build.

Tous les jobs se sont terminés avec succès, indiquant que le code est correct, que les tests sont passés et que la documentation a été générée automatiquement.



4.3 Bénéfices de l'intégration continue

L'intégration continue apporte de nombreux avantages concrets au projet. La détection précoce des bugs est considérablement améliorée car les tests s'exécutent automatiquement à chaque commit ou pull request, ce qui nous permet d'identifier les problèmes avant même la fusion du code dans la branche principale et réduit drastiquement le coût de correction des bugs.

La qualité du code est garantie par le système puisqu'il devient impossible de merger du code qui fait échouer les tests. La couverture de code est mesurée et suivie automatiquement à chaque exécution, et la documentation technique reste constamment à jour sans intervention manuelle.

La confiance dans le processus de déploiement est renforcée grâce à la validation automatique effectuée avant chaque release. L'historique complet des builds et des tests permet de tracer précisément l'origine de tout problème, et la traçabilité de toutes les modifications est assurée par le système.

Enfin, la collaboration entre développeurs est facilitée car chacun sait immédiatement si son code introduit des problèmes. Les pull requests sont validées automatiquement avant toute review humaine, et le feedback sur la qualité des contributions arrive rapidement, permettant des corrections immédiates.

4.4 Qualité des tests et couverture de code

Les tests unitaires ont pour objectif de couvrir les fonctionnalités critiques de l’application. Pour la classe **Session**, les tests vérifient notamment la génération correcte de codes de session uniques, l’ajout de joueurs avec validation des limites, la validation des votes en mode strict, ainsi que le calcul des moyennes en mode moyenne. Pour le **VoteManager**, les tests s’assurent que tous les votes sont correctement collectés avant de révéler les résultats, que les nouveaux tours sont déclenchés lorsque l’unanimité n’est pas atteinte, et que les résultats sont calculés selon le mode de jeu configuré.

La couverture de code actuelle montre que certaines parties de l’application sont bien testées, tandis que d’autres le sont beaucoup moins. Par exemple :

- Les composants critiques comme **VotingCard.jsx** et **sessionStore.js** sont entièrement couverts (100%).
- Certains composants comme **Button.jsx** et **HomePage.jsx** sont partiellement couverts (entre 33% et 57%).

- Plusieurs composants et pages, notamment **Chat.jsx**, **Timer.jsx**, **Card.jsx**, **Input.jsx**, **JoinSessionPage.jsx**, **PMSetupPage.jsx** et **WaitingRoom.jsx**, présentent une couverture très faible, proche de 0%.
- Le fichier **socket.js**, qui gère la communication temps réel, n'est couvert qu'à hauteur d'environ 28%.

Ces résultats montrent que **l'objectif de 80% de couverture globale n'est pas encore atteint** et qu'il reste des zones critiques à tester, en particulier l'interface utilisateur et les interactions temps réel. Malgré tout, les tests existants garantissent déjà la fiabilité des classes métier et des gestionnaires principaux. L'amélioration de la couverture dans les composants UI et les pages permettra de renforcer la confiance dans le code et la robustesse de l'application.

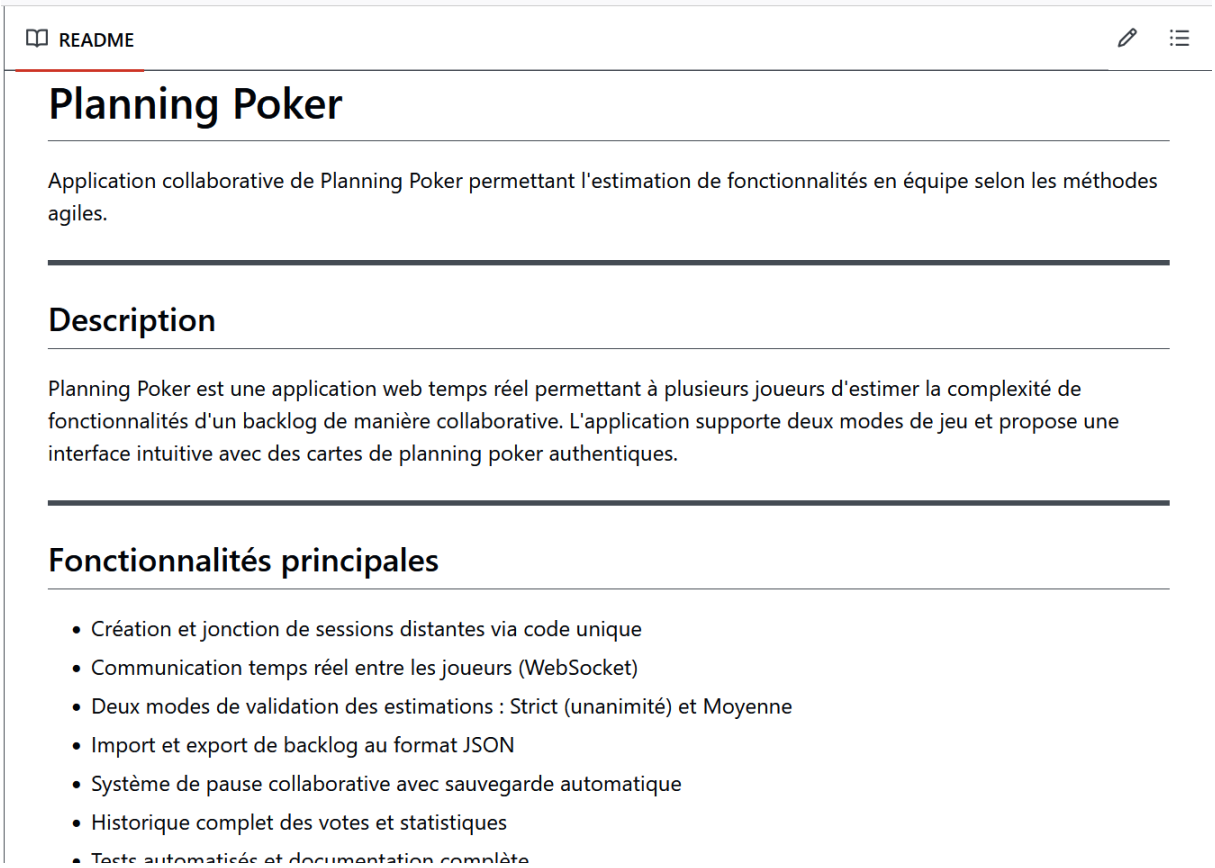
5. Documentation

5.1 Documentation utilisateur

Le fichier README.md constitue la documentation utilisateur principale et offre une vue d'ensemble complète du projet. Il commence par une description détaillée du projet et de toutes ses fonctionnalités. Les instructions d'installation sont particulièrement détaillées avec chaque commande expliquée et les prérequis clairement listés. Un guide d'utilisation pas à pas accompagne l'utilisateur depuis la création d'une session jusqu'à l'export des résultats finaux.

Des exemples concrets de fichiers JSON pour le backlog sont fournis, montrant les deux formats acceptés par l'application. Une section de dépannage aide les utilisateurs à résoudre les problèmes courants comme un port déjà utilisé ou des erreurs de connexion WebSocket. Les informations sur les auteurs, la licence MIT et les liens vers les ressources complémentaires complètent cette documentation.

La clarté et l'exhaustivité de cette documentation sont particulièrement soignées. Les prérequis techniques sont listés de manière claire et vérifiable. Toutes les commandes sont précises et peuvent être copiées-collées directement. Des exemples concrets illustrent chaque fonctionnalité importante. Des liens vers les ressources complémentaires permettent d'approfondir certains aspects techniques.



README

Planning Poker

Application collaborative de Planning Poker permettant l'estimation de fonctionnalités en équipe selon les méthodes agiles.

Description

Planning Poker est une application web temps réel permettant à plusieurs joueurs d'estimer la complexité de fonctionnalités d'un backlog de manière collaborative. L'application supporte deux modes de jeu et propose une interface intuitive avec des cartes de planning poker authentiques.

Fonctionnalités principales

- Création et jonction de sessions distantes via code unique
- Communication temps réel entre les joueurs (WebSocket)
- Deux modes de validation des estimations : Strict (unanimité) et Moyenne
- Import et export de backlog au format JSON
- Système de pause collaborative avec sauvegarde automatique
- Historique complet des votes et statistiques
- Tests automatisés et documentation complète

5.2 Documentation technique

La documentation technique est générée automatiquement grâce à JSDoc, à partir des commentaires présents dans le code source. Elle offre une vue d'ensemble de l'architecture, décrit chaque classe avec ses responsabilités, ses attributs et ses méthodes, et propose même des exemples d'utilisation. Pour faciliter la compréhension, des diagrammes illustrent les interactions entre les différents composants.

Pour y accéder, c'est très simple :

- Rendez-vous sur le dépôt GitHub du projet.
- Cliquez sur l'onglet Branches et sélectionnez la branche gh-pages.
- Vous y trouverez tous les fichiers HTML générés par JSDoc.
- Ouvrez index.html pour accéder à la page d'accueil de la documentation, ou parcourez les fichiers pour consulter les détails de chaque classe et module.

De cette façon, tout le monde peut consulter la documentation directement depuis GitHub, sans avoir besoin de générer les fichiers en local.

Lien direct vers la branche gh-pages : https://marteoued.github.io/planning-poker/

5.3 Commentaires dans le code

Le code source est parsemé de commentaires bien placés, qui jouent un rôle clé. Quand la logique métier devient complexe et que le code seul ne parle pas assez, ces notes viennent éclairer l'intention derrière les lignes. Ils explicitent aussi certains choix d'implémentation qui pourraient surprendre un développeur qui débarquerait sur le projet, et documentent les cas limites pour éviter de casser quelque chose plus tard. Au final, ces annotations rendent la maintenance bien plus fluide : on saisit vite le contexte et les contraintes, sans avoir à tout décortiquer.

Pour ce qui est des bonnes pratiques, on les respecte à la lettre. Les commentaires vont à l'essentiel, sans répéter inutilement ce que le code dit déjà. Chaque paramètre et chaque valeur de retour sont documentés de façon cohérente. Pour les fonctions un peu tordues, on ajoute même des exemples d'utilisation, histoire de montrer comment s'en servir correctement. Et quand une idée d'amélioration traverse l'esprit, on la note sous forme de TODO comme ça, on garde une trace des pistes d'évolution pour plus tard.

6. Conclusion

Le projet Planning Poker a atteint l'ensemble de ses objectifs en proposant une application fonctionnelle et intuitive. La communication temps réel est fiable grâce à l'utilisation de WebSocket via Socket.io. Les deux modes de jeu, Strict et Moyenne, sont implémentés de manière complète et fonctionnelle. Le système d'import et d'export de backlogs fonctionne parfaitement avec validation des données, tout comme la sauvegarde automatique lors des pauses café. Une suite de tests automatisés garantit une bonne couverture du code. L'intégration continue via GitHub Actions est pleinement opérationnelle. Enfin, la documentation est complète, couvrant à la fois les aspects utilisateur et technique du projet.

Annexes

1. Outils et langages utilisés

Langages de programmation

- **JavaScript** : langage principal utilisé pour le développement du backend et du frontend.
- **HTML5** : utilisé pour la structure des pages web.
- **CSS3** : utilisé pour la mise en forme et le style de l'interface utilisateur.

Frameworks et bibliothèques

- **Node.js** : environnement d'exécution JavaScript côté serveur.
- **Express.js** : framework backend pour la création de l'API REST.
- **Socket.io** : bibliothèque permettant la communication temps réel entre le serveur et les clients.
- **React 18** : bibliothèque JavaScript utilisée pour le développement de l'interface utilisateur.
- **Vite** : outil de build et de développement rapide pour le frontend.
- **Tailwind CSS** : framework CSS utilitaire pour la création d'une interface moderne et responsive.

Tests, qualité et documentation

- **Vitest** : framework de tests pour les composants et la logique frontend.
- **Jest / outils de test Node.js** : tests unitaires et d'intégration côté backend.
- **V8 Coverage** : génération des rapports de couverture de code.
- **JSDoc** : génération automatique de la documentation technique à partir des commentaires du code.

Outils DevOps et déploiement

- **Git** : gestion de version du code source.
- **GitHub** : hébergement du dépôt et gestion de la collaboration.
- **GitHub Actions** : mise en place de l'intégration continue (tests, build, documentation).
- **Netlify** : déploiement et hébergement du frontend.

Outils de développement

- **Visual Studio Code** : environnement de développement.
- **npm** : gestionnaire de dépendances JavaScript.
- **Navigateur Web (Chrome / Edge)** : tests et débogage de l'application.

2. Bibliographie et références

1. Node.js Foundation. Node.js Documentation. https://nodejs.org/en/docs/
2. Express.js Team. Express Documentation. https://expressjs.com/
3. Socket.io Team. Socket.io Documentation. https://socket.io/docs/v4/
4. Meta Platforms, Inc. React Documentation. https://react.dev/
5. Vite Team. Vite Documentation. https://vitejs.dev/
6. Tailwind Labs. Tailwind CSS Documentation. https://tailwindcss.com/docs
7. GitHub, Inc. GitHub Actions Documentation. https://docs.github.com/en/actions
8. OpenJS Foundation. JSDoc Documentation. https://jsdoc.app/
9. Schwaber, K., & Sutherland, J. The Scrum Guide. https://scrumguides.org/

Liens du projet : https://github.com/MarteOued/planning-poker

Le repository GitHub du projet est accessible à l'adresse ci-dessus où se trouve l'intégralité du code source. La documentation technique générée est disponible dans le dossier server/docs/ après exécution de la commande de génération. Le système d'intégration continue peut-être consulter dans l'onglet Actions du repository GitHub.

Auteurs :

Ouedraogo Martine s'est occupée du développement backend avec Node.js et Express, de la mise en place de la communication en temps réel via Socket.io, de la conception et de l'exécution des tests, ainsi que de la configuration de l'intégration continue avec GitHub Actions.

Ndiaye Aida a développé l'intégralité du frontend avec React 18, créé une interface utilisateur moderne et responsive grâce à Tailwind CSS, et rédigé la documentation utilisateur et technique du projet.