Planning Poker

Auteurs : Nicolas MENY & David TRUONG
Dépôt GitHub : https://github.com/0TRDavid/planning_poker

Choix techniques

Architecture : Client-Serveur

Nous avons opté pour une architecture Frontend x Backend qui sont deux projets distincts communiquant via HTTP :

- **Séparation claire des responsabilités :** Le Frontend ne s'occupe que de l'affichage et de l'interaction, tandis que le Backend gère uniquement la logique et les données. Cela rend le code plus propre et plus facile à maintenir.
- **Travail en parallèle :** Cette séparation permet de développer simultanément les deux parties du projet sans nous bloquer mutuellement, en suivant simplement le plan de l'API.

Backend : Django & Django REST Framework

Nous avons choisi **Python** avec **Django** pour la rapidité de développement qu'il offre.

Comparaison rapide des options Backend :

Comparaison Rapide

Framework	Complexité	Performance	Idéal pour	Courbe apprentissage
FastAPI	Moyenne	⚡⚡⚡ Excellente	APIs modernes	🟢 Facile
Django REST	Élevée	⚡⚡ Bonne	Gros projets	🟡 Moyen
Flask	Basse	⚡⚡ Bonne	Prototypes	🟢 Très facile
Starlette	Très basse	⚡⚡⚡ Excellente	APIs légères	🟡 Moyen

- **Simplicité et rapidité (Python) :** Python est un langage moderne, la syntaxe claire permet de se concentrer sur la logique métier plutôt que sur le code technique. C'est un atout majeur pour développer rapidement des fonctionnalités complexes.
- **Robustesse et familiarité (Django vs FastAPI) :** Nous avons privilégié Django par rapport à des frameworks plus légers comme FastAPI. Ayant déjà une expérience avec cette bibliothèque, nous avons pu être productifs immédiatement.
- **Gestion flexible des données :** L'ORM de Django nous a permis de manipuler la base de données sans écrire de SQL. De plus, l'utilisation du type `JSONField` pour stocker les User Stories nous offre une grande souplesse : nous pouvons modifier la structure d'une tâche (ajouter une description, un lien, etc.) sans devoir modifier l'architecture de la base de données.

- **Facilité de création d'API :** L'extension **Django REST Framework (DRF)** nous a permis de transformer automatiquement nos modèles en API JSON standardisée, facilitant grandement la connexion avec le Frontend React.

Frontend : React

Pour l'interface utilisateur, nous avons opté pour une stack technique moderne et efficace : **React x Vite** .

- **Expertise et Rapidité :** Nous avons choisi React car nous avions déjà une expérience sur cette technologie. Cela nous a permis de démarrer le développement immédiatement sans phase d'apprentissage.
- **Architecture par Composants :** Cette approche nous a permis de découper l'interface en petites briques autonomes (ex: `VotingDeck`, `PlayersGrid`). Une fois créés, ces composants sont réutilisés partout dans l'application, ce qui évite la duplication de code et facilite la maintenance.
- **Performance (Vite) :** Nous avons utilisé **Vite** (le standard actuel) plutôt que des outils plus anciens. Sa vitesse de compilation instantanée améliore grandement le confort de travail cad que chaque modifications est appliqué immédiatement, ce qui évite de raffraichir la page ou relancer le serveur.
- **Design Professionnel (Material UI) :** Pour ne pas perdre de temps, nous avons utilisé la bibliothèque **Material UI (MUI)** . Cela nous a permis d'obtenir rapidement une interface esthétique, ergonomique et responsive, tout en nous concentrant sur la logique du jeu.

Communication Temps Réel (Polling)

Pour la mise à jour des votes des autres joueurs :

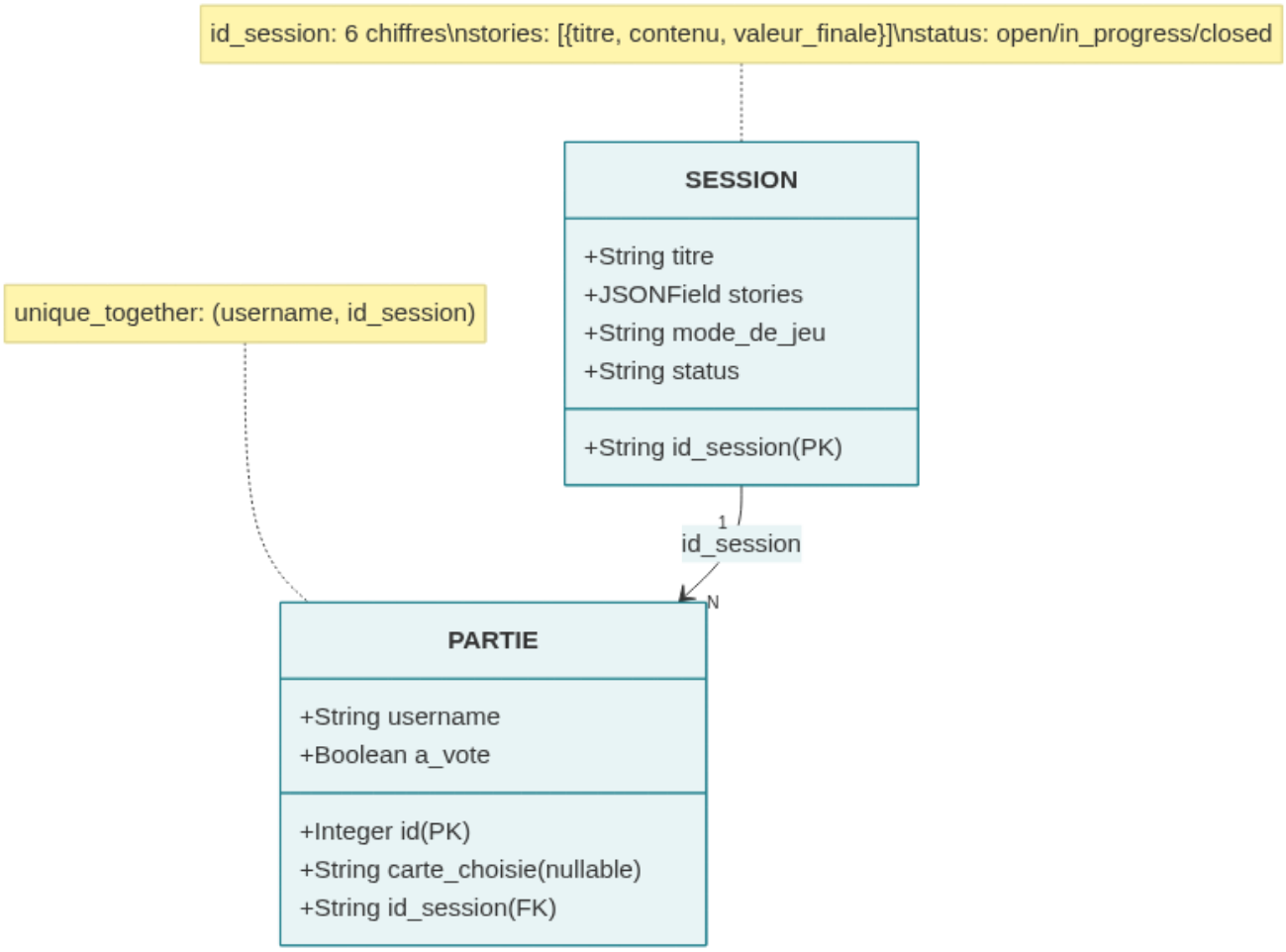
- **Choix :** Nous avons implémenté un système de **Polling** (le client (Front) interroge le serveur (Back) toutes les 2 secondes).
- **Justification :** Bien que les WebSockets soient plus performants, le polling est beaucoup plus simple à mettre en œuvre et à tester pour une application de cette échelle.

2 - Modélisation

2 tables :

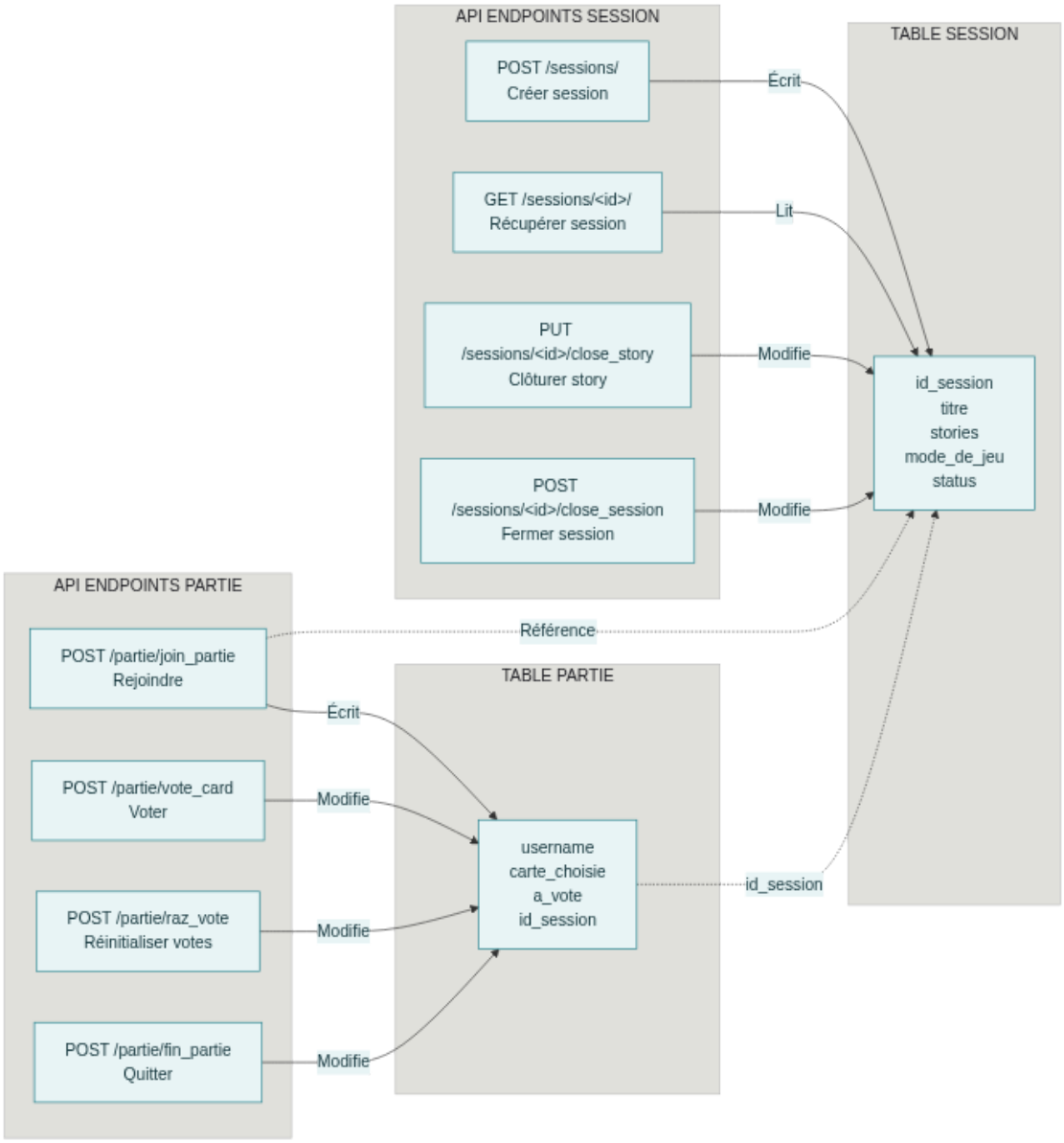
- planning_poker_session : enregistre toutes les parties, les états des parties et les résultats
- planning_poker_partie : enregistre temporairement toutes les votes des personnes liée à la partie

Diagramme uml des tables :



Les différentes API permettent au front d'interagir avec les tables et le back gère les vérifications de données, le calcul des stories et le statut des session en fonction du déroulé de la partie.

Diagramme des APIs :



3 - Intégration continue (CI/CD)

Pour garantir la qualité du code à chaque modification, nous avons mis en place un pipeline d'intégration continue via **GitHub Actions** .

Le fichier de configuration `.github/workflows/ci_pipeline.yml` exécute trois tâches à chaque `push` sur la branche principale (`main`).

Job 1 : Tests Backend

Ce job assure que la logique métier Python reste fonctionnelle.

1. Installation de l'environnement (Python 3.12).
2. Installation des dépendances (`pip install -r requirements.txt`).

3. Exécution des tests unitaires avec **Pytest** .
    - Vérification des modèles (IDs uniques, format JSON).
    - Vérification de la création de sessions
    - Vérification de la participation à une partie
    - Vérification des algorithmes de vote (Strict, Médiane, Moyenne) via des scénarios de test complets.

Job 2 : Build Frontend

Ce job vérifie que le code React est sain.

1. Installation de l'environnement (Node.js 22).
2. Installation propre des paquets (`npm ci`).
3. Analyse statique du code (`npm run lint`) pour détecter les erreurs de syntaxe ou de style.

Job 3 : Documentation Automatique

Ce job ne se lance que si les tests Backend et le Build Frontend ont réussi.

1. Installation de l'outil **Doxygen** et de **Graphviz**.
2. Génération de la documentation technique à partir des commentaires du code (`doxygen Doxyfile`).
3. Déploiement automatique du site : Doc Backend

Test & Qualité

La fiabilité de l'application repose sur une suite de tests automatisés.

Backend (Python/Django)

Les tests backend sont réalisés avec **Pytest**, privilégié pour sa syntaxe et ses fonctionnalités. Nous utilisons également **Factory Boy** pour générer des données de test réalistes.

Architecture des tests

Le dossier `backend/tests/` est structuré pour isoler chaque couche de l'application :

- **Tests Unitaires (Models & Logic) :**
    - Vérification des algorithmes de calcul des votes (Moyenne, Médiane, Majorité).
    - Validation des règles métier (ex: unicité des IDs de session, format du JSON des stories).
    - Traitement des cas limites (égalité des votes, valeurs non-numériques).
- **Tests d'Intégration (API & Views) :**
    - Simulation de requêtes HTTP via `APIClient` de DRF.
    - Vérification des codes de retour HTTP (201 Created, 404 Not Found, etc.).
    - Scénarios complets : Création de session -> Ajout de joueurs -> Vote -> Révélation.

Commandes

```
cd backend

# Lancer tous les tests
pytest
```

Documentation du code

La documentation est écrite directement dans les fichiers sources, ce qui garantit qu'elle reste à jour avec le code.

- **Outil :** Doxygen (configuré pour parser le Python).
- **Format :** Les commentaires utilisent le format `"""! ... """` avec des balises `@brief`, `@param`, `@return`.
- **Couverture :** Tous les modèles (`models.py`), vues (`views.py`) et sérialiseurs sont documentés.
- **Visualisation :** Grâce à Graphviz, Doxygen génère automatiquement les graphes d'héritage et de dépendance des classes Python.