Rapport Planning Poker

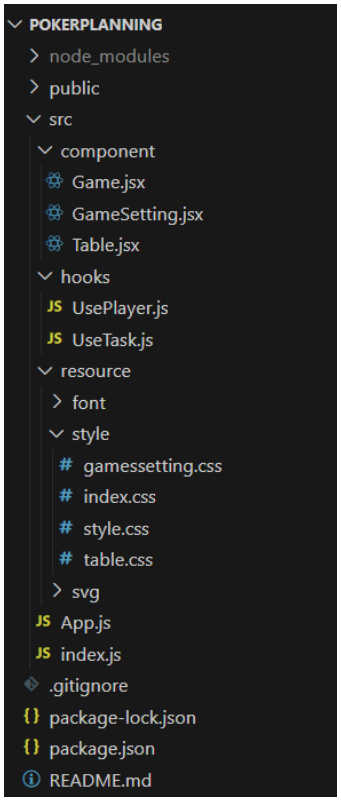
Nous avons décidé de coder notre Planning Poker en React.
Il s’agit d’une bibliothèque JavaScript open-source principalement utilisée pour des applications web, React est utile pour ce genre de projet en raison de plusieurs aspects la caractérisant :

- L’utilisation du JSX (JavaScript XML) pour la création d'éléments d'interface utilisateur. Le JSX ressemble à du HTML, mais il est transformé en JavaScript, offrant une syntaxe plus expressive pour décrire la structure du rendu.
- Grâce à la gestion efficace du DOM virtuel, React permet des mises à jour réactives de l'interface utilisateur en réponse aux changements d'état.
- La syntaxe déclarative de React et l'utilisation du JSX simplifient le développement et la compréhension du code.
- React est pris en charge par une vaste communauté et dispose d'un écosystème riche en documentations et de forum permettant de répondre à nos questions.

DESIGN PATTERNS :

HOOKS :
Les hooks, tels que useState et useEffect, sont utilisés pour gérer l'état local et les effets de bord dans les composants React. Ils peuvent être considérés comme des design pattern car en organisant la logique de l'état et des effets de manière modulaire, ils favorisent ainsi des pratiques de conception qui améliorent la lisibilité, la réutilisabilité et la maintenabilité du code.
Les hooks et le design pattern Observer partagent des similitudes dans leur objectif fondamental de simplifier la gestion de l'état. Les deux concepts mettent en œuvre des mécanismes pour gérer les dépendances, offrir une abstraction autour de la gestion de l'état, promouvoir la modularité du code, et utilisent des notifications pour informer les parties concernées des changements d'état. Bien que les détails d'implémentation diffèrent, ces similitudes résident dans leur volonté commune de fournir des solutions élégantes pour gérer dynamiquement l'évolution de l'état dans une application.

MVC :
Bien que le code ne suit pas strictement le modèle MVC, les concepts de base de séparation des préoccupations sont présents. Par exemple, le composant GameSetting gère la configuration initiale, agissant comme un contrôleur qui interagit avec les modèles (phook, thook, mhook).



Nous pouvons voir sur la capture d'écran l'arborescence de notre projet.
Il comporte plusieurs sous-dossiers ayant chacun leur importance.

Nous allons définir les différents composants (component):

Game
On le divise en étapes (config, play, end) pour gérer différents états du jeu.
On utilise des composants pour la configuration initiale, l'affichage du jeu et la résolution des tâches.
Il permet l'affichage du tapis et des cartes des joueurs.
La gestion de la sélection de cartes par les joueurs.
Résolution des tâches en fonction du mode choisi.
Il est le point central du jeu, ce composant coordonne les interactions entre les joueurs, les tâches et les modes.

Table
On l'utilise afin de gérer l'affichage d'une table avec des fonctionnalités d'ajout, de suppression et de modification des entrées.

La classe Table prend des propriétés (props) telles que hook, limit, et clazz pour personnaliser son comportement et son esthétique en fonction du contexte. Employé dans GameSetting pour gérer la configuration des joueurs et des tâches.

GameSetting
Elle sert à gérer la configuration initiale du jeu en permettant aux utilisateurs d'ajouter des joueurs, des tâches et de sélectionner un mode de jeu.
La classe GameSetting utilise les composants Table pour les joueurs et les tâches, intégrant également des éléments pour la sélection de mode et le démarrage du jeu.

Voyons les hooks :

```
import { useState } from "react";

const usePlayer = (initialState = []) => {
    const [players, setPlayers] = useState(initialState);

    const add = (player) => {
        setPlayers((prev) => [...prev, player]);
    };

    const remove = (idx) => {
        setPlayers((prev) => prev.filter((_, i) => i !== idx));
    };

    const set = (idx, player) => {
        setPlayers((prev) => {
            prev[idx] = { ...prev[idx], ...player };
            return [...prev];
        });
    };

    return [players, add, remove, set];
};

export default usePlayer;
```

‘usePlayer’ Hook :

Ce hook utilise le state local avec ‘useState’ pour stocker un tableau de joueurs.

La fonction ‘add’ permet d'ajouter un nouveau joueur au tableau.

La fonction ‘remove’ supprime un joueur en fonction de son indice.

La fonction ‘set’ met à jour les données d'un joueur spécifique.

```
import { useState } from "react";

const _initialState = [
    { text: "Créer la structure de base de la base de données.", card:
    // ... (autres tâches)
];

const useTask = (initialState = _initialState) => {
    const [tasks, setTasks] = useState(initialState);

    const add = (task) => {
        setTasks((prev) => [...prev, task]);
    };

    const remove = (idx) => {
        setTasks((prev) => prev.filter((_, i) => i !== idx));
    };

    const set = (idx, task) => {
        setTasks((prev) => {
            prev[idx] = { ...prev[idx], ...task };
            return [...prev];
        });
    };

    const clear = () => {
        setTasks([]);
    };

    return [tasks, add, remove, set, clear];
};

export default useTask;
```

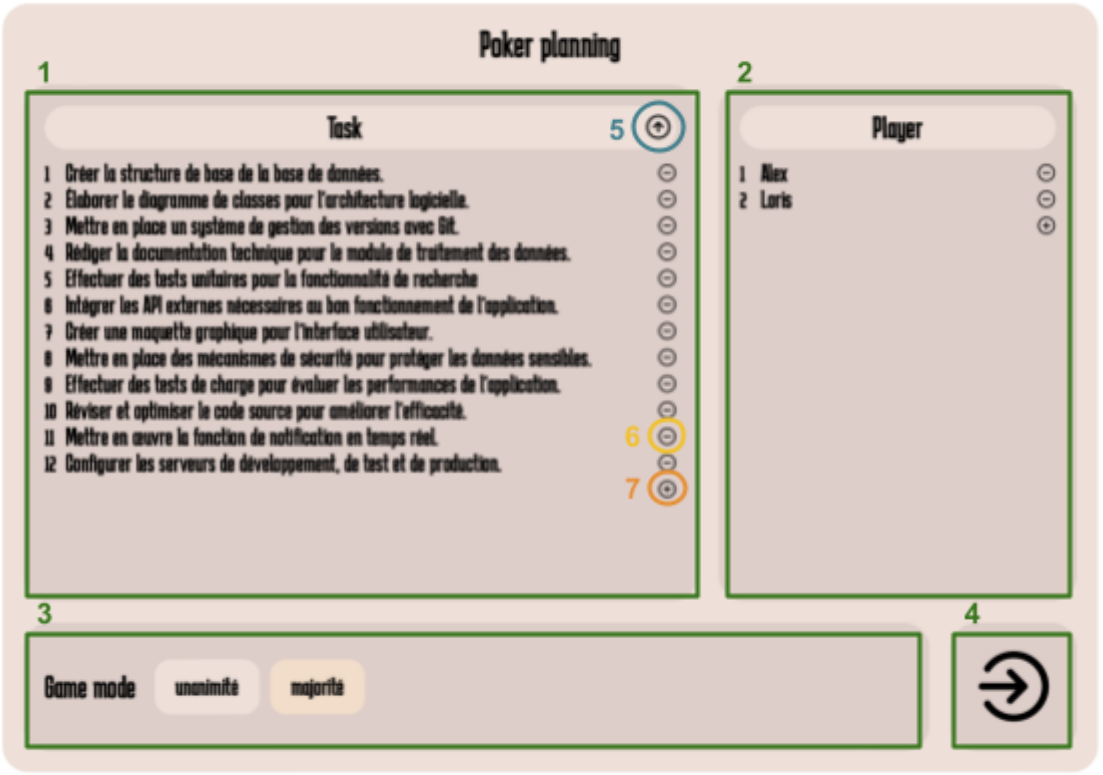
‘UseTask’ Hook :

Ce hook est similaire à ‘usePlayer’ mais est spécifiquement conçu pour gérer des tâches.

Il utilise également ‘useState’ pour stocker un tableau de tâches.

Les fonctions ‘add’, ‘remove’, ‘set’, et ‘clear’ effectuent respectivement l'ajout, la suppression, la mise à jour et la suppression de toutes les tâches.

Page de configuration



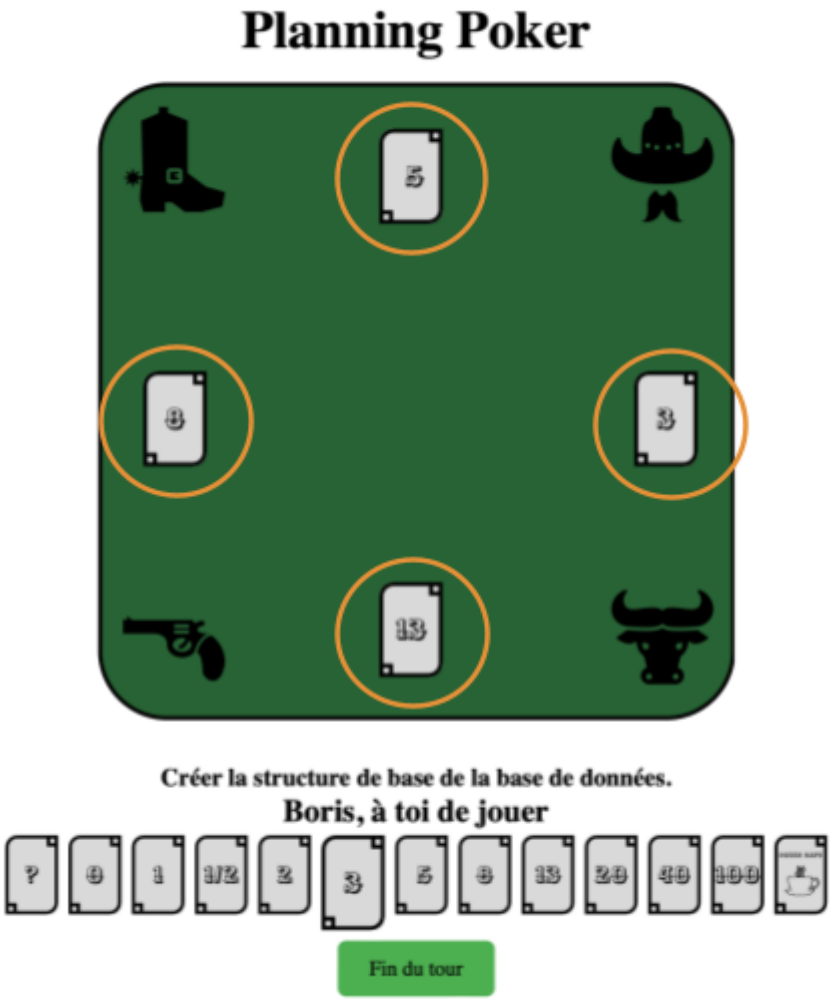
1 - Tableau de tache
2 - Tableau de joueurs
3 - Selection du mode de jeu
4 - Bouton de validation de la configuration

5 - Bouton permettant d'ajouter des taches à partir d'un fichier json*
6 - Bouton permettant de retirer une tache/joueur
7 - Bouton permettant d'ajouter un(e) tache/joueur

* Le fichier json doit être de la forme suivante :

```
[
      {"text": "text de la tache 1"},
      {"text": "text de la tache 2"}
]
```

Page de jeu



En orange sont représentées les cartes jouées par les joueurs (joueur 1 en haut, puis dans le sens anti horaire).

Au cours du développement de notre projet, nous avons adopté une intégration continue basée sur l'utilisation de GitHub Desktop. Cette plateforme nous a permis de travailler efficacement en équipe tout en préservant une gestion claire des versions de notre code.

Chacun d'entre nous travaillait sur une partie spécifique du projet, ce qui facilitait la répartition des tâches et permettait d'avancer simultanément sur plusieurs aspects. Nous avons adopté une pratique régulière de commits et de pushs. À travers ces commits, nous pouvions suivre les modifications apportées, résoudre les conflits éventuels et garantir la stabilité du projet.

Pour assurer une coordination efficace, nous avons planifié plusieurs réunions de suivi toutes les deux semaines (durant les cours) pour mettre en commun nos codes, vérifier l’avancement du projet dans la bonne direction et savoir vers où aller ensuite. Ces petits points sont devenus plus fréquents vers la fin du projet.