Application de planning poker

Eoin BRERETON HURLEY
Jules VOLPEI

Enseignant : Lachand-Pascal Valentin
Année scolaire 2025 - 2026

Table des matières

Introduction et contexte

La création de cette application s'inscrit dans l'évaluation de notre cours sur le développement de projet informatique en suivant la méthodologie agile en cours. Le sujet de notre projet est de concevoir une application permettant la réalisation d'un "planning poker" pour une équipe d'informatique via une interface sur leurs ordinateurs plutôt que directement avec de vraies cartes. Cette application permet donc de résoudre le problème de la présence physique d'un collaborateur lors d'une session de planning poker. Le planning poker est une méthode d'estimation de tâches en fonction de l'effort qui sera requis pour le développement de ses fonctionnalités.

Nous avons développé l'application à deux avec Eoin BRERETON HURLEY et Jules VOLPEI en essayant de respecter un maximum l'organisation de travail qui nous était demandée : le pair programming. Malheureusement, avec notre emploi du temps en tant qu'alternant et notre impossibilité de nous voir de manière physique lors de nos périodes d'entreprise, en grande partie dû au fait que Jules a son alternance dans la très charmante, mais lointaine, ville de Lille, nous n'avons pu mettre en place cette pratique que lors des heures de cours dédiées à la conception et à la réalisation du projet.

La suite du document détaillera les aspects techniques du projet ainsi qu'un manuel d'utilisation et d'installation du projet afin de pouvoir correctement effectuer sa session de planning poker.

Choix techniques et architectures

1. Langages et Frameworks

1. Front-End

Pour la partie front de notre application, nous avons décidé d'utiliser le langage **TypeScript** couplé à **React**. Ce choix s'explique en grande partie par la réutilisabilité des composants développés avec React, allégeant le développement pour la partie interface. Couplé à TypeScript, on évite également certains problèmes du langage JavaScript notamment la détection des erreurs ou encore le typage fort des variables. Pour l'initialisation et la création du projet, nous avons choisi **Vite** avant tout par confort mais également car il permet une mise à jour en temps réel des pages de l'application à chaque modification, évitant de relancer l'application pour chaque changement.

Concernant un peu plus la partie design de l'application, nous avons décidé d'utiliser la bibliothèque de composants graphiques **shadcn/ui** qui utilise également le framework CSS **Tailwind CSS**. Cette bibliothèque graphique met à disposition un vaste choix de composants permettant ainsi de ne pas s'attarder sur le développement de fonctionnalités qui peuvent être redondantes (ex : composant de champs de texte, boutons, …), nous permettant donc de nous occuper principalement du comportement et du fonctionnement global de l'application. Tailwind CSS permet quant à lui de directement développer le CSS dans l'HTML du composant plutôt que dans un fichier séparé, bien que la bibliothèque graphique en fournisse un lors de son installation pour instancier les valeurs constantes du style de l'application. Toujours sur la partie graphique, nous avons utilisé **Lucide React**, une bibliothèque d'icônes au format SVG, un format vectoriel adaptatif, permettant l'utilisation simple d'icônes pour la partie graphique de l'application.

Moins lié à la partie graphique, mais toujours avec React, nous avons utilisé les bibliothèques **TanStack Query** et **React Router**. La première est une bibliothèque facilitant les appels d'API et surtout pouvoir refaire les appels à l'aide d'une seule ligne permettant une actualisation dynamique des données. La deuxième est une bibliothèque permettant de gérer la navigation de l'utilisateur dans l'application notamment sans la recharger à chaque changement de pages. Couplé à **Vite**, elles permettent un développement et une utilisation fluide du site web.

2. Back-End

Pour la partie back-end de l'application, notre choix de langage s'est porté sur **Python**, pour le développement du framework web API, et **Alwaysdata**, pour l'hébergement de notre base de données distante. Avec toutes les technologies disponibles avec Python, nous avions la certitude de trouver les outils adaptés. Notre choix de framework s'est finalement porté sur **FastAPI**, un framework facilitant la conception de l'intermédiaire entre notre base de données et l'application. Pour lancer notre serveur, nous avons décidé de coupler FastAPI avec **Uvicorn**, car couplé à ce framework ces deux technologies sont **des plus efficaces** quand on souhaite développer un back end python.

Nous avons décidé d'utiliser l'ORM **SQLModel** qui est une surcouche d'un autre ORM appelé **SQLAlchemy** également utilisé dans le projet. Ils permettent de définir les tables dans le programme en Python et d'interagir avec la base de données via les objets représentant les tables. De plus, nous avons utilisé **Pydantic**, permettant d'assurer la validité des données entrantes et sortantes dans l'application, s'assurant que ce qui est envoyé côté client correspond bien à ce qui est attendu côté serveur en termes de champs et nommage de champs.

D'autres bibliothèques ont été utilisées lors du développement en Python, permettant des opérations spécifiques : **Passlib** pour le hachage du mot de passe utilisateur lors de l'inscription ou **Numpy** pour optimiser des opérations.

3. Génération documentation

Enfin, nous avons décidé d'utiliser deux bibliothèques pour générer automatiquement notre documentation : **TypeDoc**, pour la partie front avec TypeScript, et **Sphinx**, pour la partie back avec Python. Ces deux outils ont l'avantage de générer la documentation automatiquement en fonction de commentaires écrits dans le programme, de plus ils utilisent tous les deux les standards de notations des commentaires de leurs langages respectifs, permettant également une meilleure lisibilité directement dans l'IDE.

2. Architecture Logicielle

L'architecture du projet suit davantage une architecture en couches avec trois couches distinctes :

- Le client (la partie **Front** de l'application).
- Le serveur (la partie **Back** de l'application).
- La base de données (base de données distante).

Cette architecture nous permet de séparer les différentes composantes de l'application en fonction de leurs responsabilités dans son fonctionnement global. La partie associée au côté client de l'application suit la convention de conception de composant en React, séparant chaque composant selon sa fonction dans l'affichage (ex : `components/ui` pour les composants de bases de shadcn/ui ou encore `services/` pour le fichier permettant de faire les requêtes API). La partie associée au côté serveur de l'application suit le même principe avec des fichiers Python dédiés en fonction de la tâche voulue (ex : `models.py` pour les tables de l'ORM SQLAlchemy et SQLModel ou encore `database.py` qui permet d'avoir une connexion unique à la base de données).

Notre application n'utilise pas l'architecture MVC en raison du découpage entre la partie serveur et la partie client dans notre application. Notre serveur nous sert exclusivement d'API REST et ne s'occupe pas de la construction de la page comme le voudrait le principe MVC.

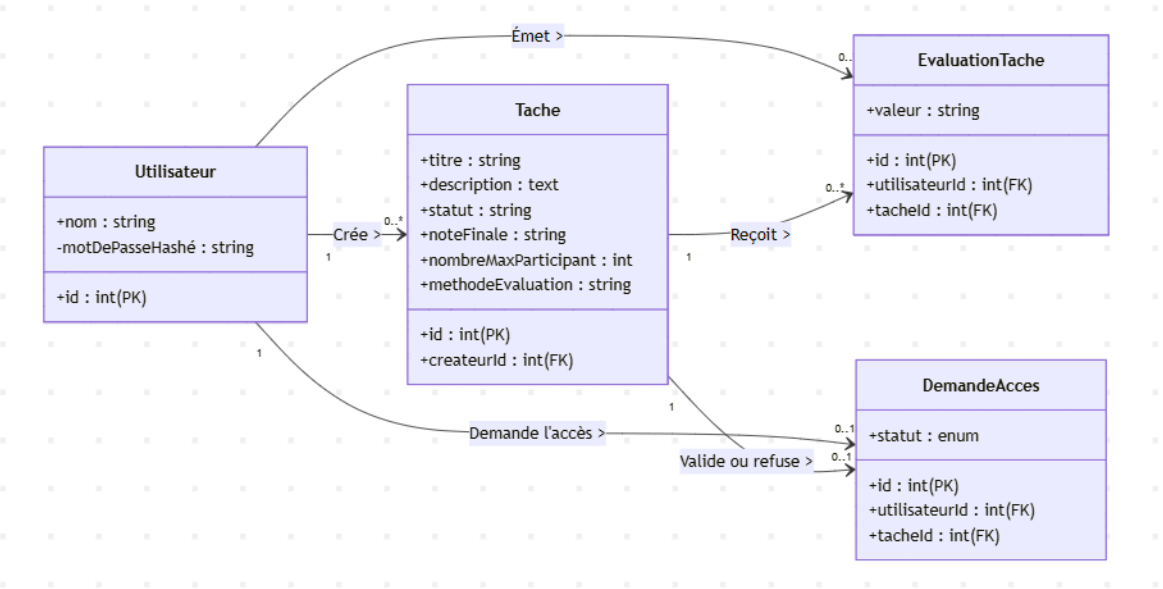
De plus, avec l'utilisation de **React** et **React Router**, notre application, côté client, suit le modèle **Single Page Application**. Cette architecture côté client permet une navigation fluide sans avoir besoin de recharger la page lors de chaque changement de page.

Enfin, notre base de données respecte la troisième forme normale (**3NF**). Elle respecte l'**atomicité** car chaque table contient des valeurs indivisibles, aucune liste, elle respecte donc la première forme normale. Elle respecte ensuite l'**identification unique** car chaque table possède une clé primaire auto-incrémentée et tous les autres attributs dépendent de cette clé, elle respecte donc la deuxième forme normale. Enfin, elle respecte l'**absence de dépendances transitives**, car aucune colonne non-clé ne dépend d'une autre colonne, elle respecte donc la troisième forme normale.

3. Modélisation (Diagrammes)

Voici les diagrammes de classes pertinents pour expliquer comment fonctionne le système de création de tâche, de demande d'accès et d'évaluation.

Ce qui correspond au **Backlog** demandé dans les attendus ici correspondrait à une tâche avec un statut à "archivée".



4. Gestion des données

Nous avons fait le choix d'utiliser une base de données distante, permettant ainsi de simuler le fait que l'application fonctionne en temps réel en grande partie grâce à TanStack et React Query. Les utilisateurs vont avoir des sessions indépendantes avec leurs propres serveurs permettant de requêter la base, que tout le monde a en commun.

La base de données dispose de quatre tables :

Utilisateur

Nom	Signification	Type	Propriété	Exemple
id	Identifiant de l'utilisateur	INTEGER	AUTO_INCREMENT, PK, UNIQUE, NOT NULL	1
nom	Nom de l'utilisateur	VARCHAR(100)	NOT NULL	"CertainementPasJules"
motDePasse	Mot de passe de l'utilisateur pour se connecter	VARCHAR(255)	Hashé avec sha256, NOT NULL	b9e50e0e8b504aa57a1bb6711ee832ee4ce9c641a1618b91833582382c709023

Tâche

Nom	Signification	Type	Propriété	Exemple
id	Identifiant de la tâche	INTEGER	AUTO_INCREMENT, PK, UNIQUE, NOT NULL	1
titre	Titre de la tâche	VARCHAR(255)	NOT NULL	“Écriture de la partie Gestion de donnée du rapport”
description	Description en détails de la tâche	TEXT		“Il faudrait savoir quoi écrire en description”
statut	Statut de la tâche	VARCHAR(50)	DEFAULT=“ouverte”	“archivée”
createurId	Identifiant du créateur de la tâche	INTEGER	FK, NOT NULL	1
nombreMaxParticipant	Nombre maximum de participant à une tâche	INTEGER	NOT NULL	14
methodeEvaluation	Méthode d’évaluation choisie pour calculer la note finale	VARCHAR(50)	NOT NULL	“Unanimité”

DemandeAccessTache

Nom	Signification	Type	Propriété	Exemple
id	Identifiant de la demande d'accès	INTEGER	AUTO_INCREMENT, PK, UNIQUE, NOT NULL	1
utilisateurId	Identifiant de l'utilisateur demandant l'accès à une tâche	INTEGER	NOT NULL, FK	1
tacheId	Identifiant de la tâche que l'utilisateur souhaite accéder	INTEGER	NOT NULL, FK	2
statut	Statut de la demande	ENUM("enAttente", "acceptee", "refusee")	DEFAULT="enAttente"	"acceptee"

EvaluationTache

Nom	Signification	Type	Propriété	Exemple
id	Identifiant de l'estimation d'une tâche	INTEGER	AUTO_INCREMENT, PK, UNIQUE, NOT NULL	1
utilisateurId	Identifiant de l'utilisateur estimant la tâche	INTEGER	NOT NULL, FK	1
tacheId	Identifiant de la tâche évaluée par l'utilisateur	INTEGER	NOT NULL, FK	2
valeur	Valeur de l'estimation	VARCHAR(50)	NOT NULL	"café"

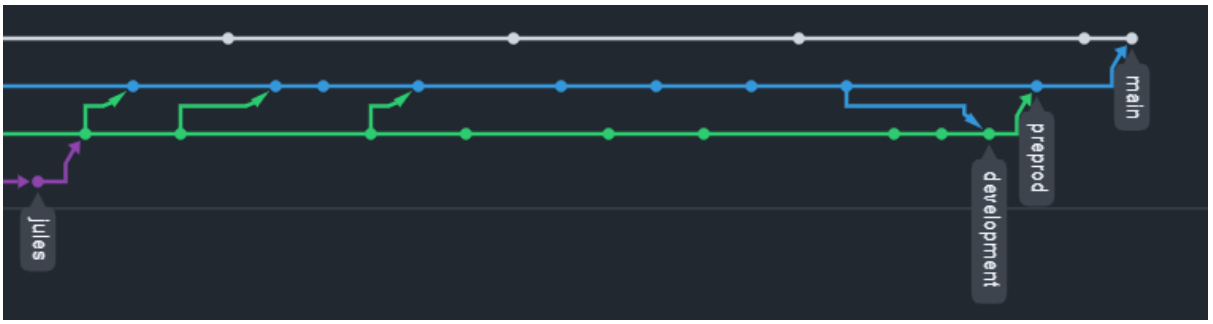
De plus, avec une fonctionnalité qui sera développé plus tard, l'utilisateur peut choisir de copier dans le presse-papier ou télécharger au format JSON le résultat de l'estimation d'une tâche dont voici un exemple :


```json
{
  "titre": "Tâche pour copier l'estimation au format JSON",
  "description": "Juste un test rien de plus",
  "createurId": 11,
  "methodeEvaluation": "Unanimité",
  "id": 27,
  "statut": "archivee",
  "nombreMaxParticipant": 1,
  "votes": [
    {
      "votant": "Test",
      "valeur": "5"
    }
  ],
  "noteFinale": {
    "etat": "Reussite",
    "message": "5"
  },
  "access": "acceptee"
}
```

Mise en place de l'Intégration Continue

1. Branches et Gitflow

Nous avons choisi d'héberger notre projet sur GitHub et de découper notre processus de développement à l'aide de plusieurs branches ayant chacune leurs propres fonctions :



Notre approche est similaire à une architecture de branche en **GitFlow** avec :

- La branche **main**, représentant la branche de production, le résultat final actuel.
- La branche **preprod**, permettant de simuler l'environnement de production tout en passant une série de validation avant de pouvoir être merge sur la branche de production (tests unitaires et génération de documentation).
- La branche **development** permettant de fusionner toutes les fonctionnalités développées sur les branches personnelles.
- Les branches personnelles (**jules** et **eoin**) permettant de développer plusieurs fonctionnalités indépendamment avant de fusionner ces branches sur la branche **development** pour mettre le travail en commun.

2. Workflow, test unitaires et documentations

Le projet possède un fichier **GitHub workflow** permettant d'effectuer plusieurs opérations à chaque push/merge sur la branche **preprod**. Notre workflow est constitué de trois jobs :

- Un job de **test unitaire Python**, qui va dans un premier temps checkout dans le répertoire avant d'installer les dépendances nécessaires (**Pytest**) grâce à `pip install` avant d'exécuter les tests unitaires liés aux fonctions de hachage de mot de passe. Nous n'avons implémenté que des tests unitaires pour cette partie-là du programme et non sur des tests de requêtes à notre API car, pour des raisons de bonnes pratiques, le dépôt est dépourvu du fichier `.env` contenant les identifiants nécessaires pour se connecter à la base de données.
- Un job de **génération de la documentation Python avec Sphinx**, qui va également checkout dans le répertoire, installer les dépendances nécessaires avant d'exécuter la commande nécessaire pour générer la documentation. Cette documentation générée sera ensuite exportée dans une archive téléchargeable dans la page des actions du dépôt.
- Un job de **génération de la documentation TypeScript avec TypeDoc** qui va aussi checkout dans le répertoire, installer les dépendances avec `npm`, avant de générer à son tour la documentation à l'aide de la commande à exécuter. Cette documentation générée sera également téléchargeable en tant qu'archive aux côtés de la documentation Python.

Lien pour télécharger les dernières archives générées en cliquant sur les jobs en gras.

Manuel d'utilisation et fonctionnalités

1. Installation et lancement

L'installation du projet va demander d'exécuter les commandes dans l'ordre suivant :

Cloner le dépôt :
- `git clone` https://github.com/JulesVolpei/PlanningPokerApp

Création du fichier .env :
- `cd .\PlanningPokerApp\planningpoker\back\`
- `touch .env` Ou simplement créer un fichier s'appelant ".env"
- Mettre dans le nouveau fichier :

DATABASEURL=mysql://435429:Admin13!@mysql-planningpoker.alwaysdata.net/planningpoker_db

Installation des dépendances :
- `cd ../..` (toujours dans le répertoire `/back`)
- `pip install -r requirements.tx`
- `cd .\planningpoker\`
- `npm install`

Lancement :
- Ouvrir un second terminal à la racine du projet
- Dans le premier terminal (celui dans `.\PlanningPokerApp\planningpoker\`) : `npm run dev`
- Dans le second terminal : `cd .\PlanningPokerApp\planningpoker\back\`
- Dans le second terminal : `uvicorn main:app --host 127.0.0.1 --port 800`

Une fois le serveur lancé dans le second terminal, rendez-vous dans le premier terminal pour cliquer sur : ➜ **`Local:`** http://localhost:5173/

2. Modes de jeu

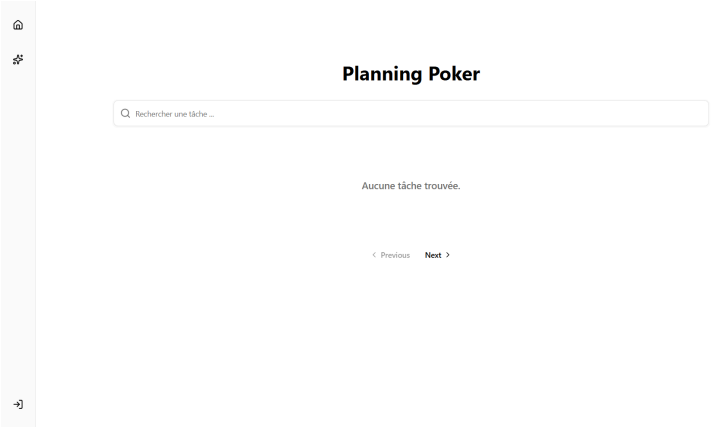
Différents modes d'évaluation ont été implémenté dans notre application :

La méthode d'évaluation : "Unanimité" est la méthode d'évaluation donnée par défaut lors de la création d'une tâche. Chaque mode d'évaluation a son calcul spécial dans la fonction `calculNoteFinale()`.

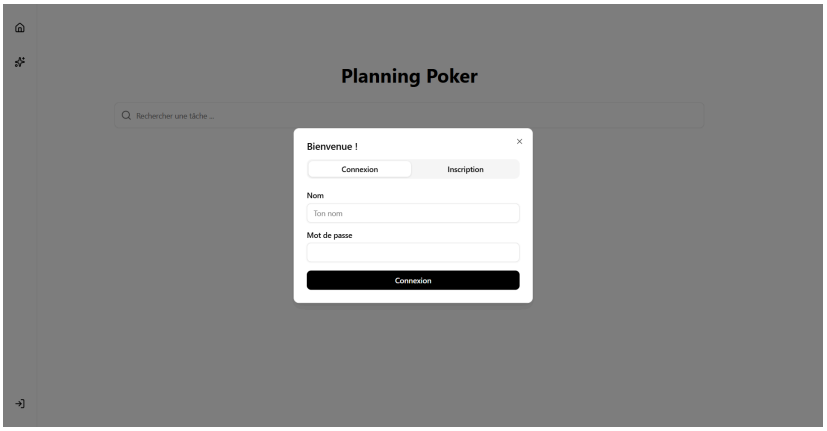
Si un utilisateur décide de voter : "?", son vote ne sera pas pris en compte lors du calcul de la note finale. Cependant, si un autre utilisateur a voté "café" alors le vote pour le calcul de la note finale devra être relancé.

3. Interface

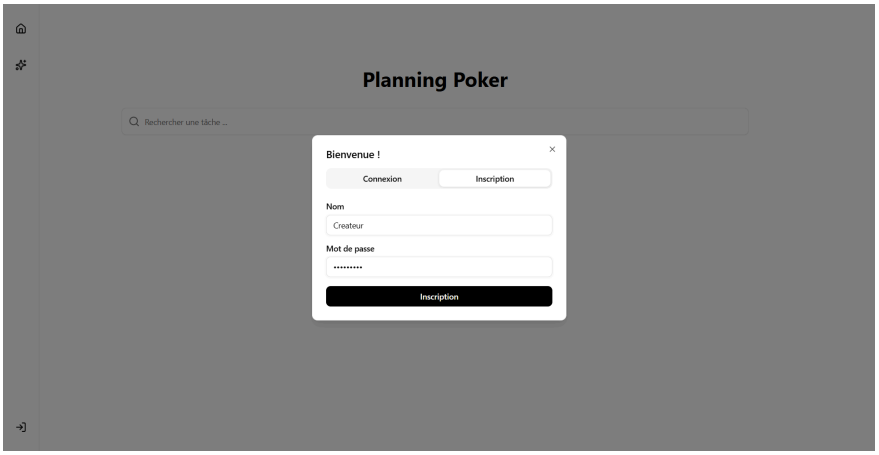
Nous allons maintenant présenter l'interface utilisateur dans le cas de l'évaluation d'une tâche par une équipe :

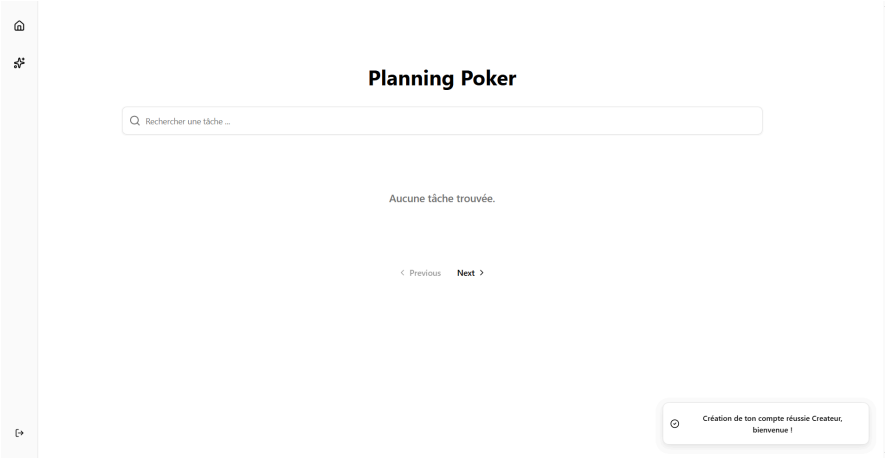


Un utilisateur appelé "Créateur" souhaite créer une tâche à évaluer pour lui et son ami "AmiCréateur". Il va commencer par se créer un compte en cliquant sur l'icône en bas à gauche de la sidebar de l'application.

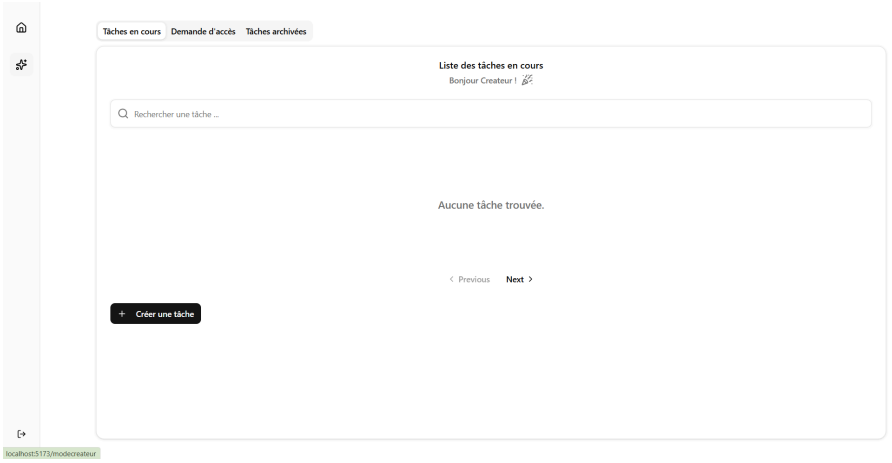


Un formulaire de connexion et d'inscription s'ouvre alors, il va ensuite cliquer sur l'option : "Inscription" pour rentrer ses identifiants et s'inscrire.

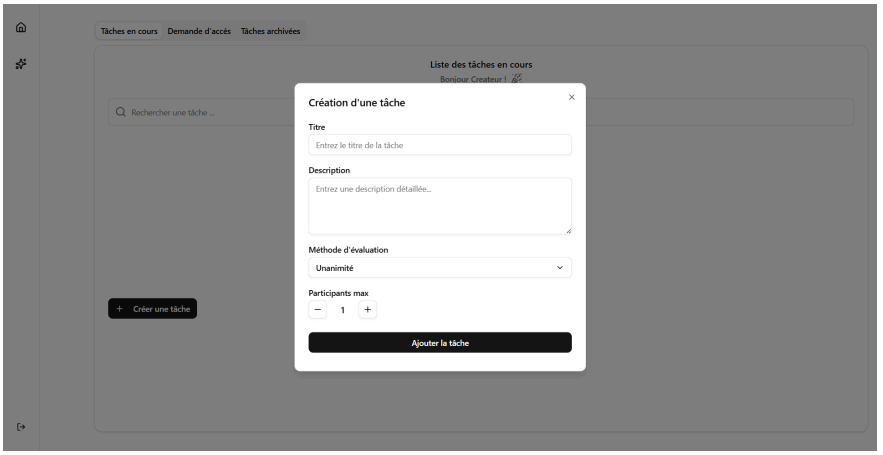




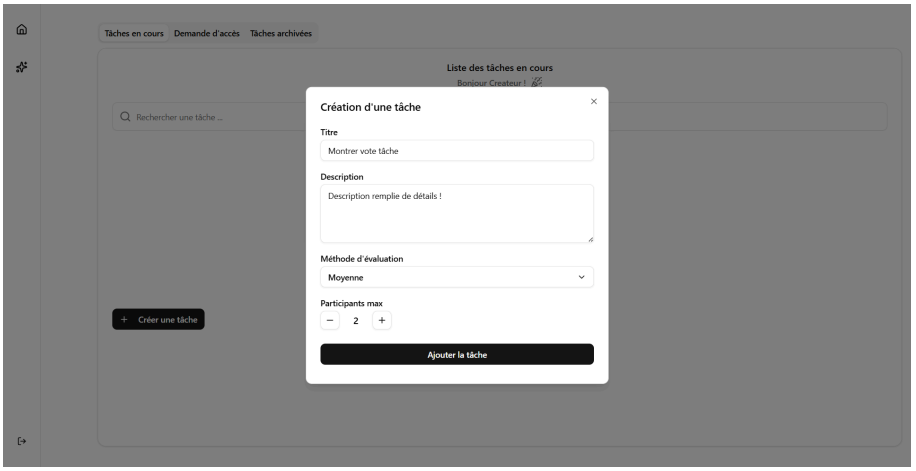
Une fois inscrit, Créateur sera notifié de son inscription. Il pourra ensuite se rendre dans le "mode créateur" en cliquant sur les petites icônes d'étoiles sur la sidebar. S'il n'était pas connecté, cela aurait simplement ouvert le formulaire de connexion.



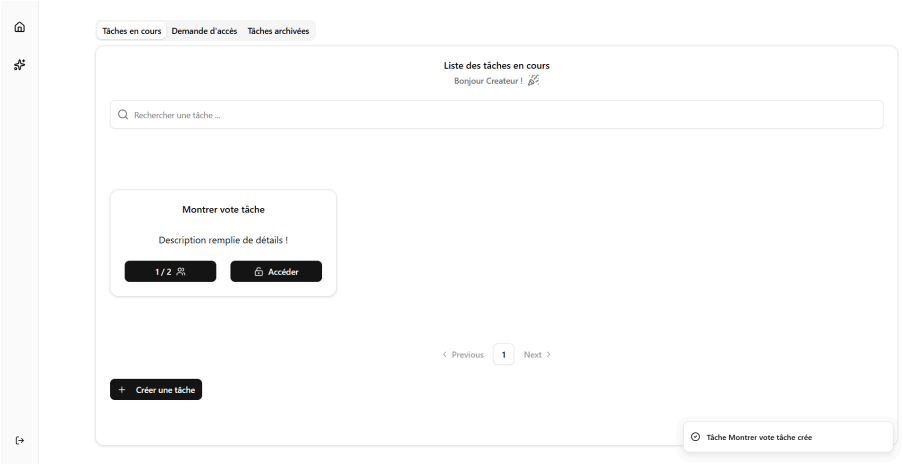
Il peut ensuite cliquer sur le bouton “+ Créer une tâche” pour ouvrir le formulaire de création de tâche.



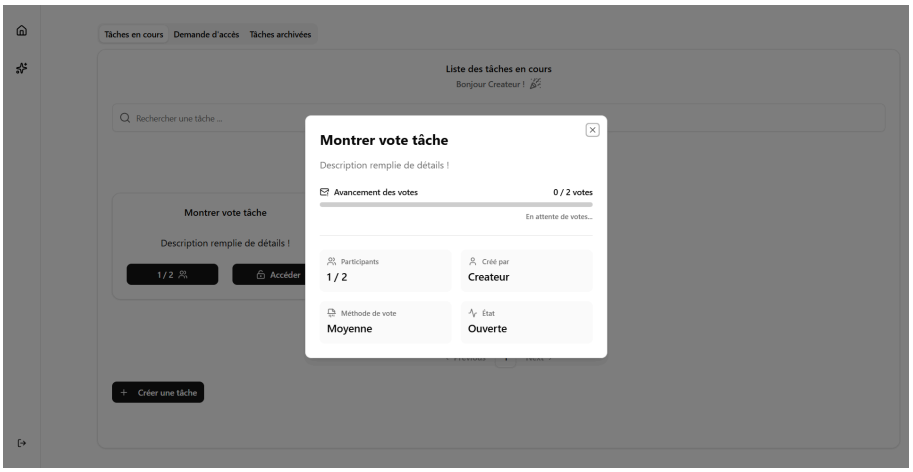
Il peut ensuite entrer les informations qu’il souhaite : le titre, la description, la méthode d’évaluation ainsi que le nombre de participants (entre 1 et 99 personnes).



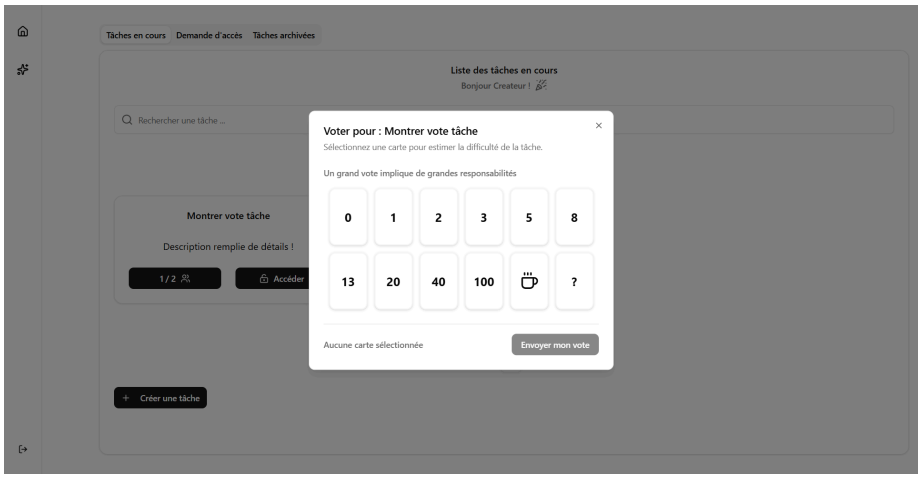
Une fois les informations entrées, il peut appuyer sur “Ajouter la tâche” pour venir ajouter la tâche à la liste des tâches en cours dans le mode créateur.



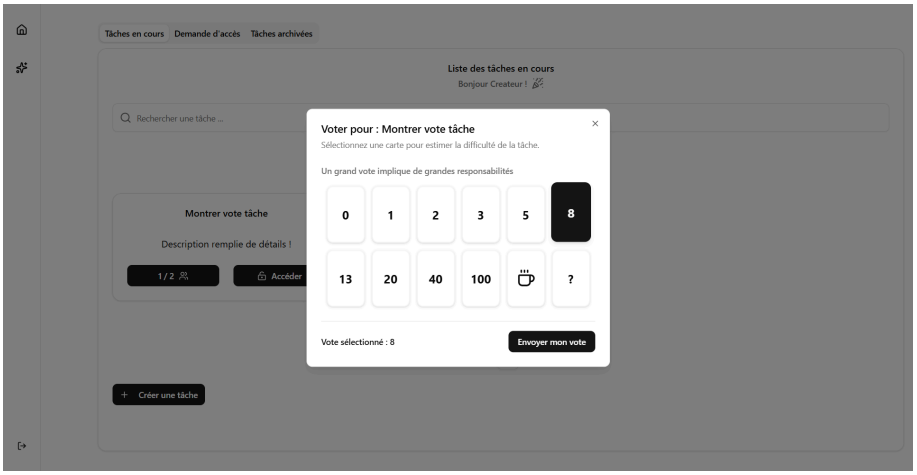
L’utilisateur sera notifié de la validation de la création de sa tâche dans la base de données et cette dernière va s’afficher dans le menu des tâches en cours car elle est toujours en cours.



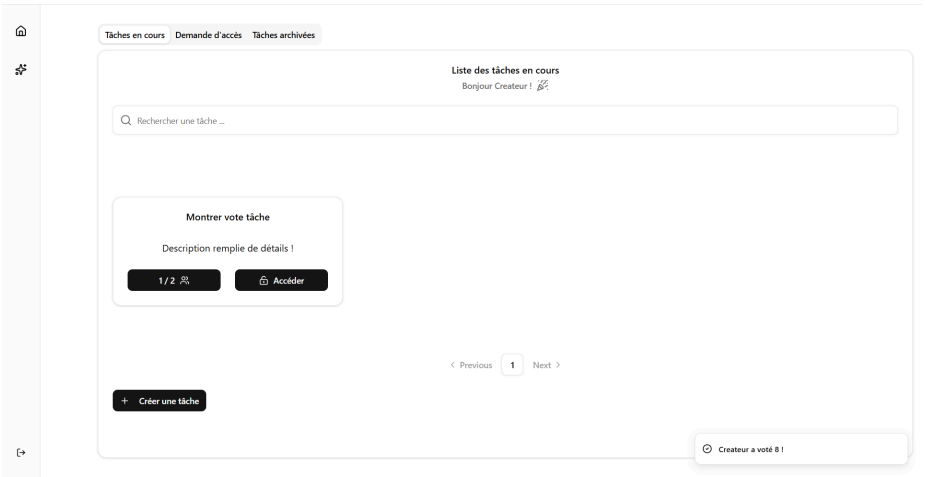
Cliquer sur la carte de la tâche va ouvrir un pop-up avec différentes informations sur la tâche, notamment une barre de progression des votes.
L’utilisateur peut ensuite cliquer sur le bouton “Accéder” pour accéder au pop-up de vote.

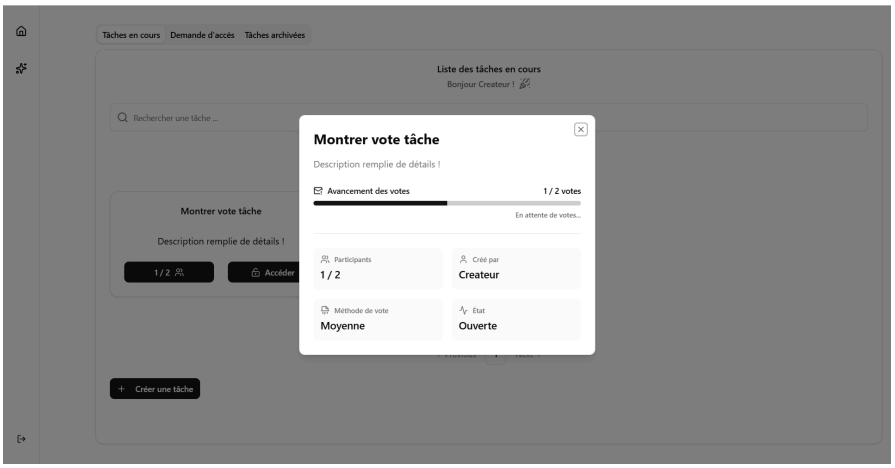


Il peut ensuite enregistrer son vote en cliquant sur la carte de son choix.



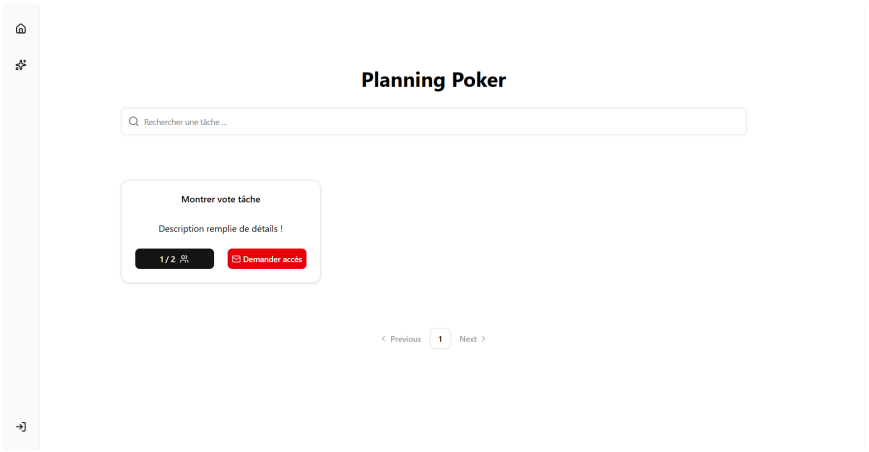
Une fois son vote confirmé en appuyant sur “Envoyer le vote”, l’utilisateur sera notifié de l’ajout de son vote en base de données.



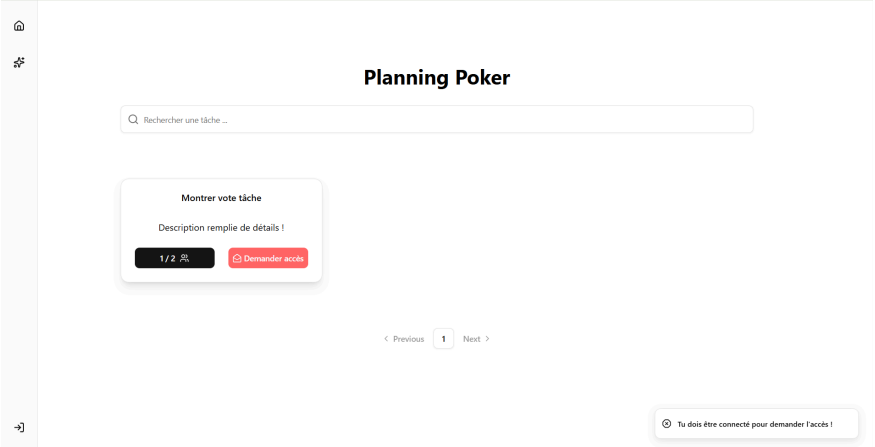


La barre de progression est alors automatiquement modifiée.

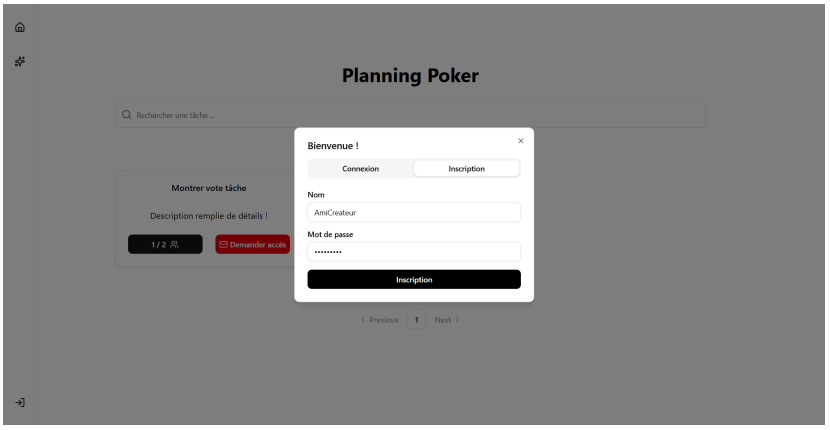
L'utilisateur "Créateur" peut alors se déconnecter en cliquant sur l'icône en bas à gauche de la sidebar. Une fois de retour sur la page d'accueil, on remarque que notre tâche a été ajoutée à l'ensemble des tâches disponibles sur le menu principal.



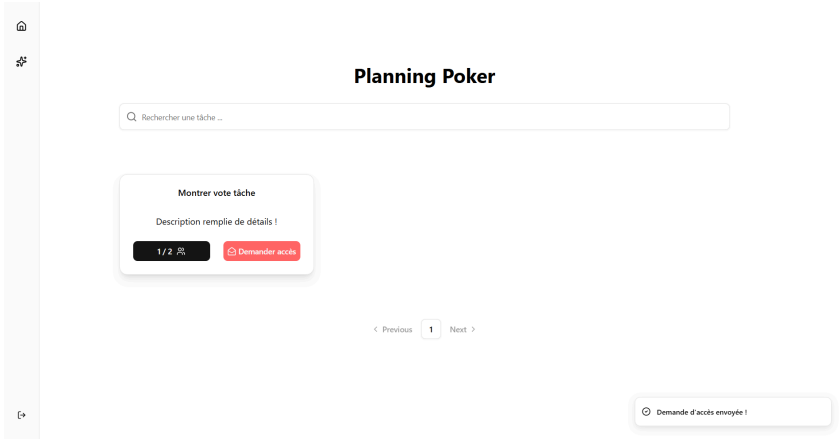
Cliquer sur le bouton "Demander accès" notifie l'utilisateur non connecté qu'il faut qu'il se connecte avant de pouvoir demander l'accès à la tâche.

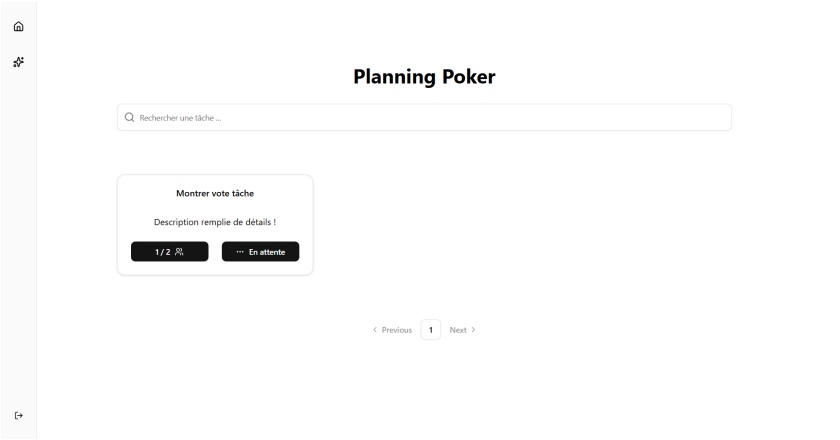


"AmiCréateur" se crée donc un compte à son tour pour pouvoir demander l'accès à la session de planning poker.

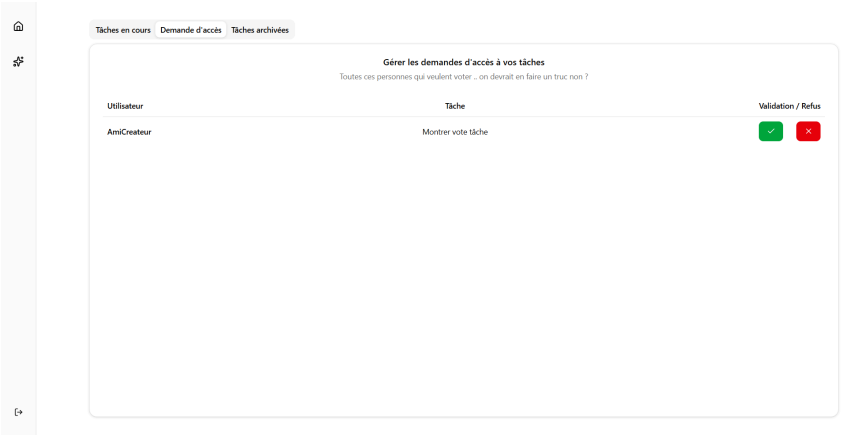


Une fois connecté, ce dernier peut désormais cliquer sur le bouton pour demander l'accès. L'application le notifie de l'ajout de sa demande d'accès dans la base de données.

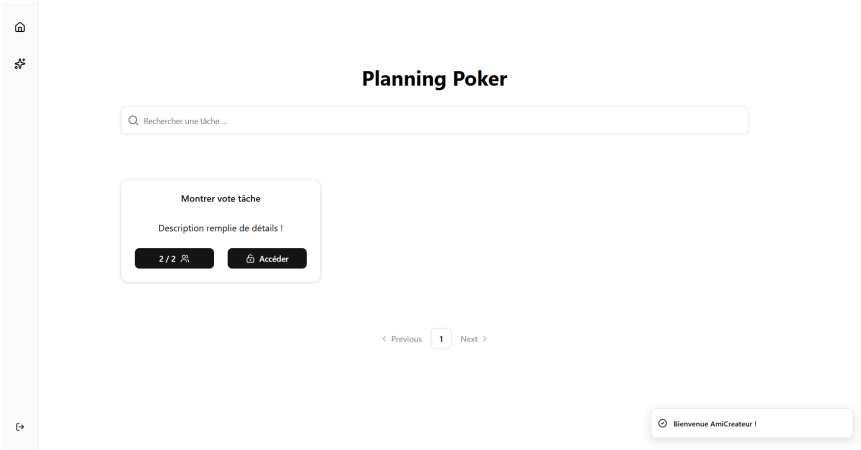


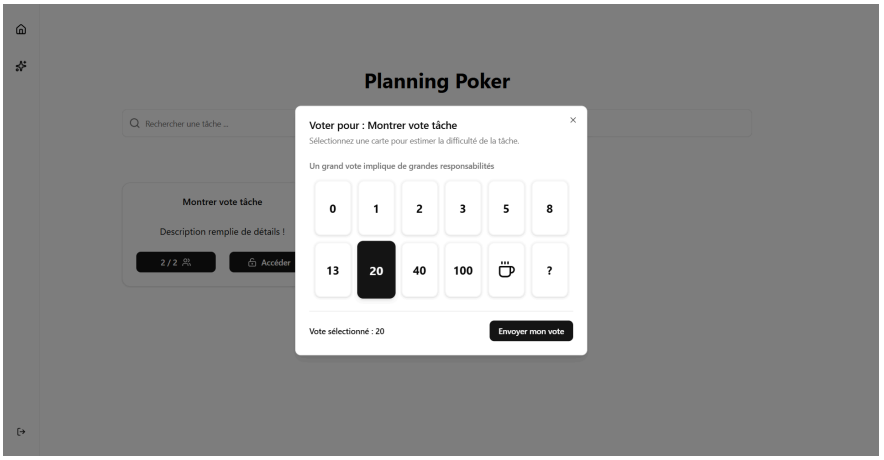


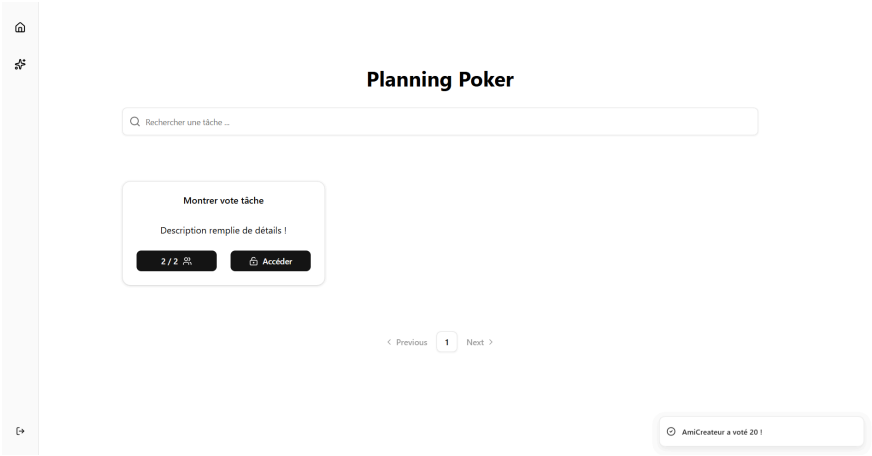
Pendant ce temps, chez "Créateur", si ce dernier va dans l'onglet "Demande d'accès" du menu créateur, il reçoit la demande d'accès de son ami.



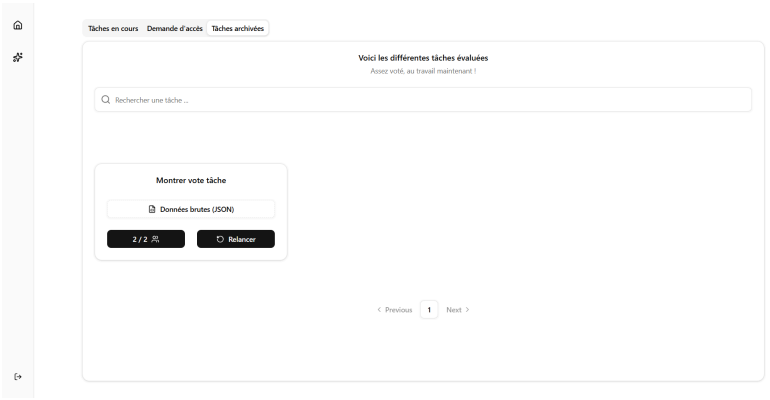
Une fois fait, le bouton d'accès pour la tâche change pour "AmiCréateur" et ce dernier peut lui aussi accéder au vote d'estimation de la tâche. Cette tâche n'apparaîtra plus que pour eux deux car le nombre de participants à la session est atteint.

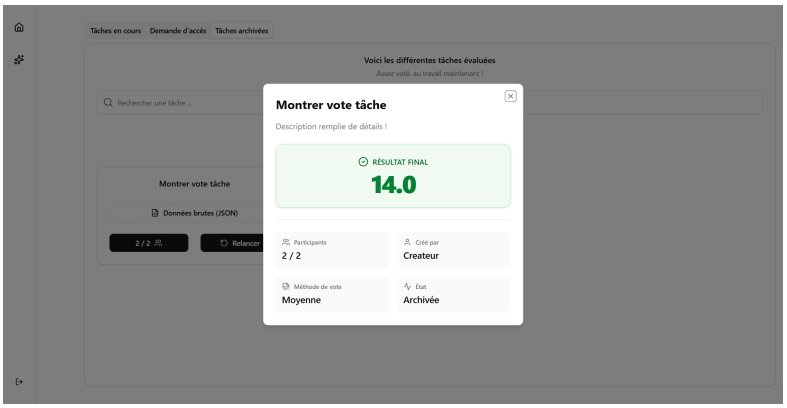




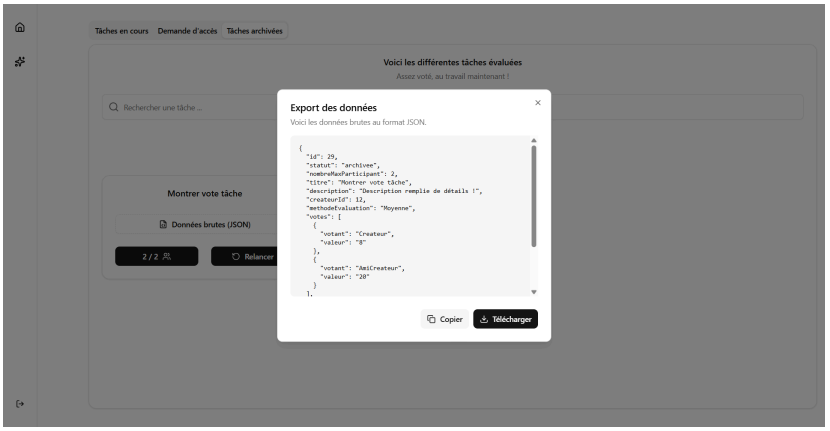


Comme tous les utilisateurs ont voté pour la tâche, cette dernière va disparaître et “Créateur” peut désormais se rendre dans l’onglet “Tâches archivées” du mode créateur pour voir le résultat en cliquant sur la carte de la tâche.





Le créateur de la tâche peut aussi accéder aux données brutes au format JSON pour les copier dans le presse-papier ou directement les télécharger dans un fichier JSON.



Il peut également décider d'appuyer sur "Relancer" pour relancer le vote de l'estimation de la tâche. La tâche apparaîtra alors de nouveau pour les utilisateurs y ayant accès et ils pourront voter de nouveau.