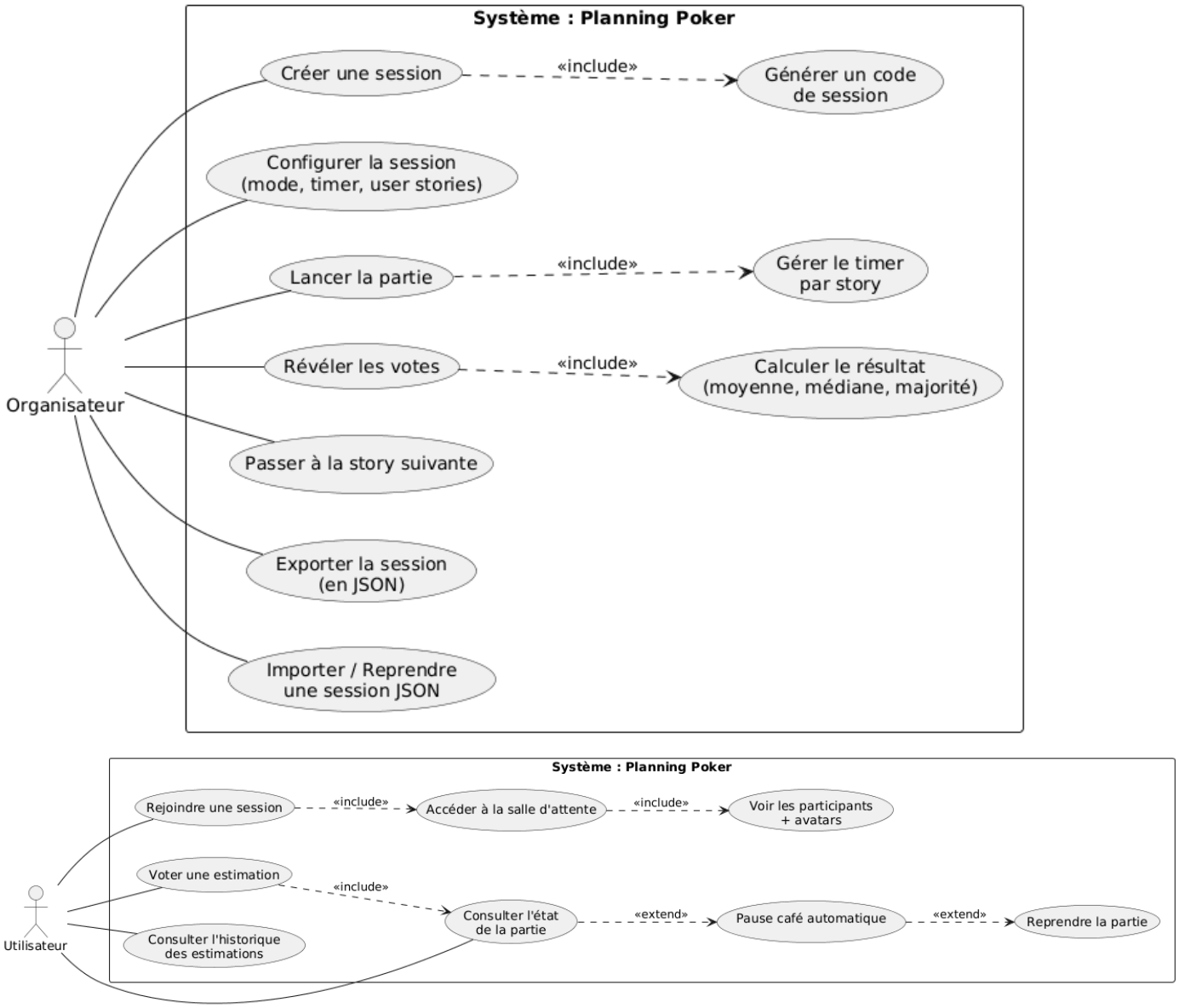
I. Introduction et contexte :

Le projet donné consiste à concevoir et à développer une application de Planning Poker c'est-à-dire une application de vote en collaboration pour estimer des tâches ou des fonctionnalités d'un backlog.

L'objectif est de fournir un cadre clair pour la phase d'estimation. En effet, pour chaque user story (définie par l'organisateur), l'application permet à plusieurs participants de proposer une valeur d'effort (qui correspond à la suite de Fibonacci). Cette estimation individuelle permet ensuite d'aboutir à une estimation partagée fondée sur le mode de jeu choisi. Cette approche vise à améliorer la fiabilité des estimations en s'appuyant sur la confrontation de points de vue, à rendre le processus plus transparent (chacun contribue), et à faciliter la planification en produisant des résultats directement exploitables.

Cas d'utilisations :



Nous réalisons ce projet en binôme en nous appuyant sur des pratiques agiles ainsi que sur les principes d'intégration continue vus en cours. Étant deux, nous avons appliqué le pair programming issu de l'Extreme Programming (XP) : lors des sessions en présentiel, nous travaillons sur une même machine avec une répartition claire des rôles. L'une de nous tient le rôle de driver (implémentation, intégration, corrections), tandis que l'autre joue le rôle de navigateur (relecture immédiate, validation de la logique, identification des cas limites, vérification de la conformité au cahier des charges). Dans ces phases de pair programming, nous avons fait le choix de travailler directement sur la branche main, afin de garder un flux simple, une base de code unique et constamment partagée, et d'éviter des divergences inutiles sur des fonctionnalités développées à deux en temps réel. Nous alternons régulièrement les rôles pour maintenir une compréhension commune et réduire le risque qu'une partie du projet dépende d'une seule personne.

À l'inverse, lorsque nous travaillons à distance, nous passons sur une organisation plus classique : chacune développe sur sa branche, puis nous intégrons via une étape de revue et de fusion. Cette séparation limite les conflits, permet de travailler en parallèle et facilite la traçabilité des changements. Pour structurer cette répartition, nous utilisons ClickUp : nous y définissons les tâches, priorités et sous-tâches, suivons l'avancement, et faisons un point régulier en général en fin de journée afin de synchroniser ce qui a été fait, ce qui reste à faire, et les difficultés rencontrées, et cela dans un esprit proche des méthodes agiles.

Aperçu de l'interface Clickup

Enfin, le projet s'appuie sur Git et sur GitHub Actions pour l'intégration continue. Git nous sert à versionner le code, documenter l'évolution via des commits réguliers et sécuriser les retours en arrière si nécessaire. GitHub Actions automatise une partie des contrôles à chaque mise à jour du dépôt (par exemple lors d'un push ou d'une pull request) : exécution des tests et génération de documentation. Cette automatisation permet de détecter plus tôt les régressions, de stabiliser l'intégration des contributions de chacune, et de maintenir une qualité constante tout au long du développement.

II. Choix techniques et architecture :

1. Langages & frameworks :

Le projet de planning poker repose sur une architecture web qui combine un back-end en Python et un front-end en JavaScript/HTML/CSS.

Python est utilisé pour la partie serveur. Nous l'avons choisi parce que c'est un langage que nous maîtrisons toutes les deux, qu'il gère très bien les données au format JSON, et qu'il s'intègre facilement dans une architecture web grâce à des frameworks légers comme Flask.

Pour le projet nous avons utilisé plusieurs bibliothèques Python :

- Flask pour construire l'application web, définir les routes, gérer les pages HTML, les redirections et la logique de requête/réponse entre client et serveur.
- Json pour charger et sauvegarder les fichiers JSON qui décrivent l'état des parties (backlog, sauvegarde de session).
- Os pour manipuler les chemins et accéder au système de fichiers.

- Random, string et time pour générer des identifiants, créer des valeurs aléatoires et gérer certains aspects temporels liés aux sessions.
- Firebase-admin (avec Firestore) pour connecter l'application à Firebase Firestore et y stocker ou lire des informations de session dans une base de données distante, ce qui permet d'avoir une application collaborative en ligne avec un stockage centralisé.

Le front-end, lui, est réalisé en JavaScript, HTML et CSS. Nous avons choisi ces technologies car elles sont standards pour le développement web, supportées par tous les navigateurs, et particulièrement adaptées à une application interactive multi-utilisateurs. Elles permettent aux joueurs d'accéder au planning poker simplement via un navigateur, sans installation, et de voir les mises à jour de la partie quasiment en temps réel.

On a donc HTML qui structure les différentes pages et écrans, CSS qui gère la mise en forme et JavaScript qui communique avec le back-end Flask.

2. Architectures logicielles :

L'architecture logicielle du projet de planning poker suit une organisation en couches, proche d'un modèle MVC simplifié, qui sépare clairement la partie interface utilisateur, la logique métier et la gestion des données.

La couche de présentation regroupe tout le code exécuté dans le navigateur:

- Le HTML sert à définir la structure des écrans : page d'accueil pour créer ou rejoindre une session, écran principal avec l'affichage de la story en cours, la liste des participants, la zone de vote et les contrôles réservés à l'organisateur.
- Le CSS prend en charge la mise en forme et l'ergonomie de l'interface : apparence des cartes de vote, disposition de la « table » de planning poker, présentation des avatars, lisibilité générale.
- Le JavaScript, enfin, assure le comportement dynamique de l'application. Il envoie les actions de l'utilisateur au serveur (rejoindre une session, voter, passer à la story suivante, mettre en pause la partie, etc.), récupère l'état courant de la session via des appels à l'API exposée par Flask, puis met à jour le DOM pour refléter la réalité du jeu : quels joueurs ont voté, quelle user story est active, quel est le statut de la partie. Dans cette architecture, le front-end se comporte comme la « Vue » : il se concentre sur l'affichage et l'interaction, sans porter la logique métier principale.

La couche métier et contrôleur est implémentée en Python avec Flask. C'est elle qui centralise les règles du planning poker et la coordination de l'application. Elle définit les routes HTTP et les API REST appelées par le front-end pour créer une nouvelle session, rejoindre une session existante, enregistrer un vote, changer la story ou obtenir l'état actuel du

jeu. À chaque requête, cette couche met à jour l'état de la partie, applique les règles nécessaires et renvoie au front-end une représentation structurée de cet état.

Enfin, une troisième couche est dédiée à la gestion des données et à la persistance de l'état des parties. Elle est située côté serveur et pilotée par le code Python. C'est elle qui est responsable de la sauvegarde et du rechargement des informations relatives aux sessions, au backlog, aux participants, aux votes et à l'historique. Cette couche s'appuie à la fois sur des fichiers structurés et sur un stockage distant via Firebase.

3. Modélisation :

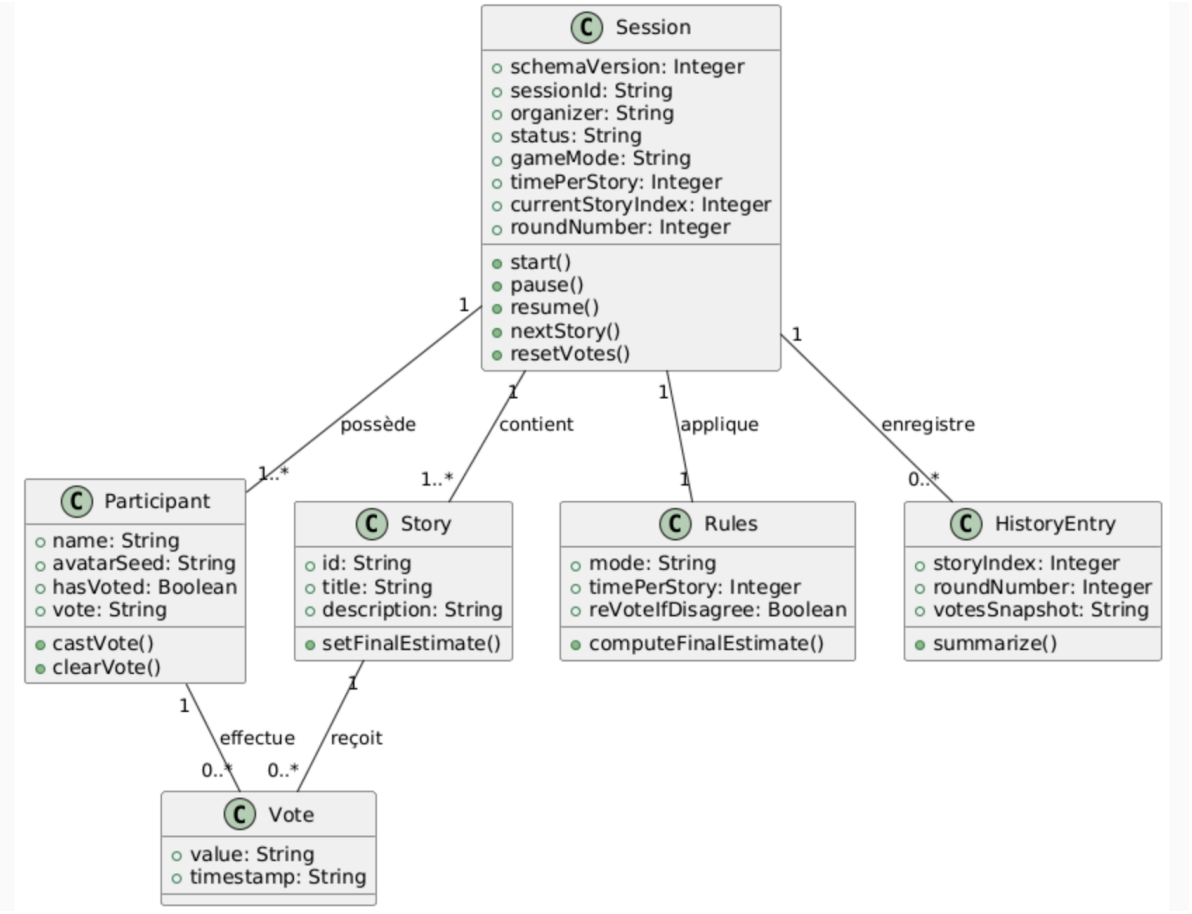


Diagramme de classes

- Session : représente une partie de planning poker. Elle stocke l'identifiant de session, l'organisateur, le statut, le mode de jeu, le temps par story, l'index de la story courante et le numéro de round. Elle possède les participants, les stories, un historique et applique un ensemble de règles.
- Participant : représente un joueur, avec un nom, un avatar, un état de vote et la valeur de son vote courant. Il peut voter et réinitialiser son vote.
- Story : représente une user story du backlog, avec un identifiant, un titre et une description. On peut lui associer une estimation finale.
- Vote : représente un vote individuel, avec une valeur et une date/heure (simplifiée en chaîne de caractères ici).
- Rules : regroupe les paramètres des règles de la partie (mode de jeu, temps par story, gestion des revotes) et le calcul du résultat final des votes.
- HistoryEntry : enregistre un round passé pour une story donnée (indices de story et de round, plus un « snapshot » des votes résumé sous forme de texte).

4. Gestion des données :


```
{
  "schemaVersion": 1,
  "sessionId": "VWKTG9",
  "organizer": "ruru",
  "status": "paused",
  "gameMode": "strict",
  "timePerStory": 5,
  "userStories": [
    "erzfzr"
  ],
  "currentStoryIndex": 0,
  "round_number": 2,
  "history": [],
  "participants": [
    {
      "hasVoted": true,
      "vote": "🎱",
      "name": "YOYO",
      "avatarSeed": "astronaut"
    },
    {
      "hasVoted": true,
      "vote": "🎱",
      "name": "ruru",
      "avatarSeed": "astronaut"
    }
  ]
}
```


Exemple de JSON

La gestion des données dans notre projet repose sur un fichier JSON qui sert à la fois de backlog et de sauvegarde de partie. Il contient tout l'état d'une session de planning poker.

On y trouve d'abord les informations générales de la session comme 'schemaVersion' pour la version du format, 'sessionId' pour identifier la partie, 'organizer' pour le créateur, 'status' pour l'état courant (par exemple en pause ou active), 'gameMode' pour le mode de jeu et 'timePerStory' pour le temps alloué à chaque story.

La partie backlog est représentée par le tableau 'userStories', qui contient les stories à estimer ou déjà estimées, et par 'currentStoryIndex', qui indique quelle story est actuellement en cours. Le champ 'round_number' précise le numéro du round de vote pour cette story , permettant de savoir s' il ya eu un revote. Lorsque nous ajoutons une estimation finale à une user story, nous pouvons la stocker dans une propriété dédiée (par exemple 'estimation'). Le fait de laisser cette estimation à 'null' permet de savoir qu'une user story n'a pas encore été traitée ou validée, ce qui distingue clairement les stories déjà estimées de celles qui restent à faire.

Enfin, la sauvegarde de la partie inclut 'history', prévu pour stocker l'historique des tours, et surtout 'participants', qui est la liste des joueurs. Chaque participant a un 'name', un 'avatarSeed', ainsi que 'hasVoted' et 'vote' pour indiquer s'il a voté et quelle valeur il a choisie.

Ce format JSON est simple, lisible et directement exploitable aussi bien par le back-end Python que par le front-end JavaScript, ce qui facilite la sauvegarde et la reprise d'une partie à tout moment.

III. Mise en place de l'intégration continue :

Notre projet intègre plusieurs mécanismes visant à assurer la qualité logicielle et la maintenabilité du code. En effet, nous avons implémenté des tests automatisés couvrant le backend et le frontend, une chaîne d'intégration continue pour exécuter ces tests à chaque modification, ainsi qu'une génération automatique de documentation technique à partir du code.

1. Structure des fichiers et des scripts :

Comme vu précédemment, le projet est structuré de manière à séparer les différentes responsabilités :

- Le code source principal du backend se trouve dans “app.py” (application Flask).
  Les templates HTML sont dans “templates/”.
- Les fichiers statiques (JS, CSS, images) dans “static/” et “asset/” (les scripts JavaScript du front sont dans “static/scripts/”).

Les tests sont organisés par couche :

- Le dossier “tests/backend/” contient les tests Pytest pour le serveur Flask (fichier test_app.py).
- Le dossier “tests/frontend/” contient les tests Jest pour le front.

Et enfin, le répertoire ".github/workflows/" contient les configurations des pipelines d'intégration continue :

- ci.yml pour les tests
- docs-pages.yml pour la documentation.

Cette organisation en arborescence claire (séparation du code, des tests, des assets, de la doc, des CI scripts) suit les conventions courantes et facilite la navigation dans le projet.

2. Outils utilisés et justifications :

2.1 Pytest (tests backend) :

Pour les tests automatisés du backend Python, le choix s'est porté sur Pytest pour sa simplicité d'utilisation. Il permet d'écrire des tests dans un style très concis et lisible en utilisant la syntaxe Python habituelle. Pytest supporte par ailleurs nativement les fixtures, qui offrent un moyen puissant de factoriser le code de préparation et de nettoyage des tests.

Notre projet exploite justement cette fonctionnalité pour gérer la connexion à la base Firestore de test et purger les données entre chaque test. En somme, Pytest apporte une solution de test complète et efficace pour le backend Python.

2.2 Jest (tests frontend) :

Le choix de Jest pour les tests front-end JavaScript s'explique par la nature du projet et les atouts de cet outil. Jest est un framework de test JavaScript largement adopté et simple d'usage. Jest propose une configuration "zero-config" par défaut, ce qui a permis de l'intégrer rapidement au projet sans avoir à écrire de configuration complexe. Il inclut également un simulateur de DOM (JSDOM) ce qui le rend idéal pour tester du code prévu pour le navigateur sans nécessiter de navigateur réel. Son API intègre directement tout le nécessaire (framework de tests, assertions, mocking) dans un seul outil unifié, ce qui simplifie la mise en place.

Dans notre projet, Jest permet de tester le code JS du planning poker avec des tests unitaires de fonctions, mais aussi des tests d'interaction utilisateur simulée. Jest fournit donc un environnement de test JavaScript complet, fiable et facile à configurer, parfaitement adapté pour garantir la qualité du code front du projet.

2.3 Doxygen (documentation) :

Pour la génération de la documentation technique, l'outil retenu est Doxygen. Doxygen est un générateur de documentation très répandu dans l'industrie logicielle, qui permet de produire une documentation détaillée à partir de commentaires dans le code. L'une des raisons majeures de son choix dans ce projet est sa prise en charge de multiples langages de programmation au sein d'un même outil. En effet, Doxygen supporte entre autres le C++, le

Java, le Python, le JavaScript, etc. Cette capacité était essentielle ici car notre projet est hybride (Python pour le backend, JavaScript pour le frontend) et il était souhaitable d'utiliser un unique outil pour documenter l'ensemble. Le choix de Doxygen se justifie aussi par la possibilité d'intégrer des pages écrites à la main (en Markdown) pour fournir des vues d'ensemble, ce qui a été fait avec les documents d'architecture et de modèle de données.

2.4 GitHub Actions (CI/CD) :

Le projet étant hébergé sur GitHub, nous avons logiquement opté pour GitHub Actions afin d'implémenter l'intégration continue. GitHub Actions est une plateforme CI/CD intégrée à GitHub qui présente plusieurs avantages : d'abord, elle est gratuite pour les dépôts publics. Ensuite, elle offre une intégration transparente avec le dépôt Git, les workflows sont versionnés dans le repo et toute poussée de code peut déclencher des builds automatiquement. Dans ce projet, GitHub Actions a permis de mettre en place rapidement un pipeline multi-jobs (tests Python, tests JS et documentation auto) grâce à sa syntaxe YAML.

3. Tests automatisés et couverture fonctionnelle :

3.1 Tests backend (Pytest et Flask)

Côté serveur (backend), le projet utilise “Pytest” pour définir et exécuter une suite de tests automatisés. Ces tests se trouvent dans le répertoire “tests/backend/” et couvrent l'ensemble des fonctionnalités critiques du backend Flask (app.py).
On y trouve des tests unitaires et d'intégration qui vérifie entre autre :

- La création de session de poker.
- La jonction à une session via un code.
- le vote sur une user story.
- Le reveal des votes par l'organisateur.
- Le fonctionnement du chat intégré.
- L'export et l'import de partie.

Chaque route et logique métier importante fait l'objet d'un test garantissant que le comportement est conforme aux spécifications.

Pour isoler les tests et éviter les effets de bord, le code de test utilise des fixtures Pytest. Par exemple, une fixture marquée “autouse=True” est définie pour nettoyer la base Firestore avant et après chaque test. Elle supprime toutes les collections et documents créés lors d'un test afin que le suivant démarre sur une base de données vierge. Ainsi, on s’assure que les tests sont indépendants les uns des autres (pas de dépendance à l'état laissé par un test précédent).

Les tests backend utilisent également le client de test Flask (app.test_client()) via une fixture “client” pour simuler des requêtes HTTP sur l'application Flask sans lancer de serveur réel. Cela permet de tester les routes (GET/POST sur /create, /join, /vote, etc.) et les réponses JSON de l'API.

L'ensemble de ces tests automatisés assure que chaque composant critique fonctionne conformément aux attentes et qu'aucune régression n'est introduite lors des évolutions du code.

3.2 Tests frontend (Jest, JSDOM) :

Côté interface utilisateur (frontend), le projet comporte également des tests automatisés écrits en JavaScript, utilisant le framework “Jest”. Ces tests sont situés dans “tests/frontend/” et couvrent plusieurs aspects de la logique front-end.

On y trouve :

- Des tests unitaires de fonctions utilitaires (fichier “vote-utils.test.js” pour tester des fonctions comme le calcul de moyenne des estimations ou l'arrondi à la carte Fibonacci la plus proche)
- Et des tests plus orientés DOM (fichier “vote-dom.test.js”) pour vérifier le comportement des interactions utilisateur sur l'interface de vote.

Grâce à Jest, l'environnement de test simule un navigateur via “JSDOM”. Cela permet de créer un faux DOM en mémoire et d'y injecter du HTML afin de tester, par exemple, que cliquer sur une carte de vote sélectionne bien cette carte, remplit un champ caché du formulaire et déclenche l'affichage d'un message de statut.

Les tests front-end vérifient aussi que le code gérant l'écran d'attente (waiting.js) se comporte correctement. On s'assure ainsi que la logique côté client fonctionne comme prévu, indépendamment du navigateur réel.

Certaines adaptations ont été nécessaires pour exécuter les tests frontend dans Node/JSDOM. Par exemple, certains APIs web comme “TextEncoder/TextDecoder” n'étaient pas définies par défaut dans l'environnement de test. Pour pallier ce problème, un fichier de configuration Jest (tests/frontend/jest.setup.js) ajoute explicitement ces objets au scope global de Node avant d'exécuter les tests, afin que le code front les trouve comme dans un vrai navigateur.

Enfin, les tests Jest sont exécutés en mode “serial” (option --runInBand) lors du CI pour éviter tout conflit d'accès concurrent au faux DOM global. L'ensemble de ces tests frontend contribue à garantir qu'aucun changement de code ne brise l'expérience utilisateur ou la logique métier côté client.

4. Documentation technique automatique (Doxygen)

4.1 Structure de la documentation générée :

Nous avons également mis en place d'une documentation technique automatique générée à partir du code et de fichiers Markdown, en utilisant Doxygen. Le dépôt contient un dossier “docs/” avec des documents écrits manuellement (architecture.md détaillant l'architecture de l'application, data_model.md décrivant le schéma de données Firestore), ainsi qu'un fichier de configuration Doxyfile “docs/doxygen/Doxyfile”. Ce dernier paramètre Doxygen pour parcourir le code source et produire un site de documentation HTML.

Pour ce faire, nous avons renseigné le code Python et JavaScript avec des commentaires structurés au format Doxygen afin que les fonctions, classes et modules soient décrits dans la documentation.
Par exemple, en tête du fichier principal “app.py”, on trouve un commentaire multi-ligne indiquant le @brief (résumé) et le @details du module, ainsi que les routes prises en charge, comme illustré ci-dessous :

```
"""
@file app.py
@brief Backend Flask de Planning Poker (sessions, votes, chat, export/import).
@details
- Pages HTML: /, /create, /join, /waiting/<id>, /vote/<id>
- API JSON: /api/game/<id>, /api/participants/<id>, /api/chat/<id>, ...
- Persistance: Firestore (sessions + sous-collections participants/chat)
"""
```

4.2 Configuration de Doxygen :

Le fichier Doxyfile de base (fourni dans docs/doxygen/Doxyfile) a été personnalisé afin de couvrir à la fois le code Python et JavaScript et de présenter la documentation en français et dans un format cohérent.

Principaux points de configuration :

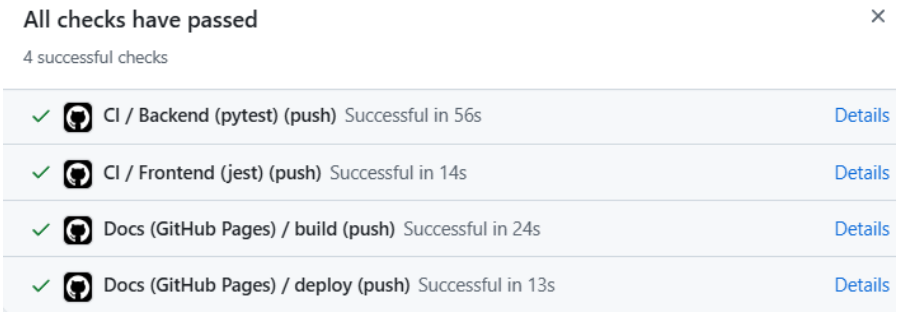
- **Langages et formats supportés :** La directive “EXTENSION_MAPPING” est utilisée pour indiquer à Doxygen de traiter les fichiers “.py” comme du Python et “.js” comme du JavaScript. De plus, “FILE_PATTERNS” inclut *.py, *.js, *.html, *.css et *.md ce qui signifie que Doxygen analysera non seulement le code source, mais aussi les fichiers HTML/CSS et les fichiers Markdown.
- **Langue de sortie :** “OUTPUT_LANGUAGE” est positionné à “French” pour avoir une documentation en français
- **Extraction du code :** Il a été décidé de ne pas documenter les éléments privés ou sans commentaires (paramètres “EXTRACT_ALL = NO”, “EXTRACT_PRIVATE =

NO”). Cela implique que seuls les éléments du code ayant des commentaires Doxygen apparaîtront dans la documentation. C'est un choix courant pour éviter d'exposer des fonctions internes non documentées.

L'implémentation de Doxygen dans notre projet montre une volonté d'avoir une documentation structurée et de la maintenir à jour automatiquement via l'intégration continue.

5. Chaîne d'intégration continue (CI) :

Le projet est équipé d'une chaîne d'intégration continue basée sur GitHub Actions, qui se déclenche à chaque push sur le dépôt ainsi que sur chaque pull request.



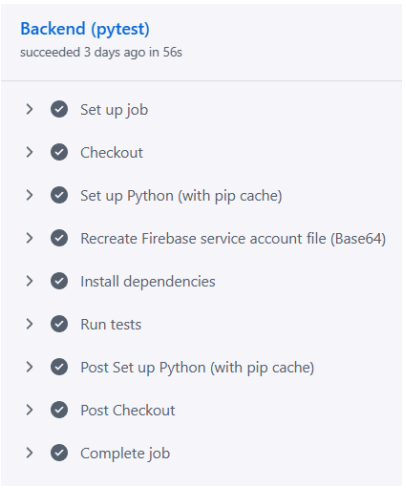
Aperçu des jobs GitHub Actions exécutés avec succès

5.1 Automatisation des tests

Le workflow principal (défini dans “.github/workflows/ci.yml”) exécute automatiquement les tests décrits ci-dessus. Concrètement, le pipeline CI comporte deux jobs parallèles : Backend (pytest) et Frontend (jest). Chaque job tourne sur une machine virtuelle Ubuntu fournie par GitHub Actions, avec un timeout de 10 minutes pour prévenir les blocages.

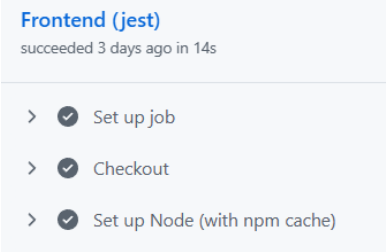
Le job backend :

- Clone le dépôt dans le runner pour rendre le code accessible aux steps suivants
- Installe Python 3.11
- Restaure le cache de dépendances pip
- Installe les packages du fichier requirements.txt
- Il reconstruit le fichier de crédentiels Firebase nécessaire
- Puis lance les tests backend via “pytest -q”.



Étapes du job backend dans le pipeline CI

Le job frontend :

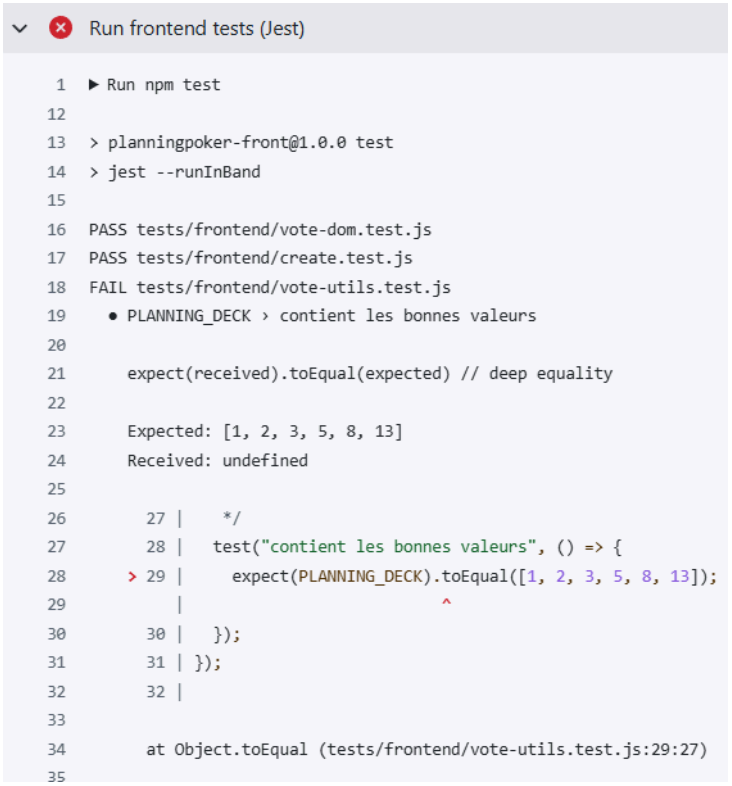


- Clone le dépôt dans le runner pour rendre le code accessible aux steps suivants
- Installe Node.js (version 20) avec cache npm
- Exécute npm ci pour installer les dépendances JavaScript du projet (définies dans package.json)
- Puis lance les tests front-end avec la commande npm test -- --ci (Jest en mode CI).

Étapes du job frontend dans le pipeline CI

Grâce à cette séparation, les tests Python et JS s'exécutent simultanément, ce qui réduit le temps total d'intégration.

Si tous les tests passent, cela confirme que le commit n'a pas introduit de régression. En cas d'échec de test, la plateforme GitHub affiche directement les logs et résultats, nous permettant d'obtenir un feedback rapide et détaillé sur l'erreur.



Exemple d'erreur sur le run des tests front

5.2 Sécurisation des identifiants Firebase dans la CI :

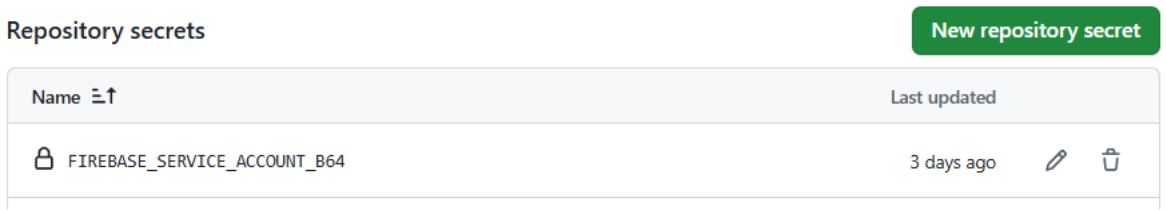
Le projet utilise Firebase côté backend via le SDK "firebase_admin" pour Python, ce qui nécessite un fichier de credentials JSON (donné par Firebase lors de la création d'un compte de service) afin de s'authentifier auprès de l'API Google.

Plusieurs mesures ont été prises pour gérer ce secret de façon sécurisée et pratique dans le CI :

- **Ne pas commiter le secret** : Le fichier JSON de clés privée Firebase n'est pas présent dans le dépôt. Le ".gitignore" contient une règle excluant ce fichier JSON. Ainsi, même si un développeur utilise un fichier localement nommé "pokerplanning-749a9-firebase-adminsdk-fbsvc-10f7d5cc49.json" il ne sera pas versionné par git. On évite tout risque d'exposer la clé sensible sur GitHub.

```
# Credentials
pokerplanning-*-firebase-adminsdk-*.json
*.key
*.pem
*.p12
```

- **Utiliser un secret GitHub pour Firebase** : Afin de pouvoir effectuer les tests backend, la solution adoptée a été de stocker la clé Firebase encodée en base64 dans un Secret GitHub. Ce secret, nommé "FIREBASE_SERVICE_ACCOUNT_B64" est injecté dans le workflow CI. L'étape de reconstitution dans "ci.yml" le décode vers un fichier JSON sur la VM de build et exporte "GOOGLE_APPLICATION_CREDENTIALS" vers ce fichier. Ainsi, lorsque le code Python de tests s'exécute, il trouve la variable d'environnement renseignée, et le SDK Firebase peut initialiser "firebase_admin" avec le bon contexte.



Création de secrets dans GITHUB

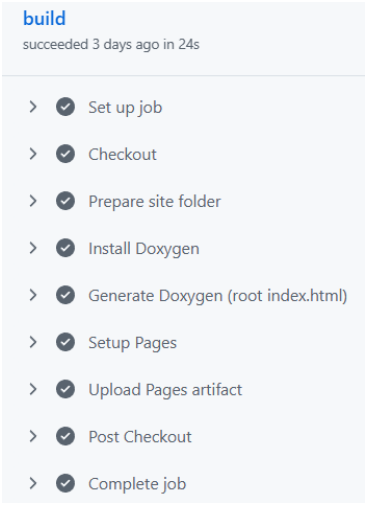
5.3 Génération et publication automatique de la documentation :

Le workflow de génération et de publication de la documentation (défini dans .github/workflows/docs-pages.yml) automatise la création d'un site web de documentation technique à partir du code et des fichiers Markdown du projet. Il se déclenche automatiquement lors d'un push sur la branche main, ou manuellement via l'interface GitHub (événement workflow_dispatch).

Concrètement, le pipeline se compose de deux jobs : build (génération de la documentation avec Doxygen) puis deploy (déploiement sur GitHub Pages). Il s'exécute sur une machine virtuelle Ubuntu standard, fournie par GitHub Actions.

Le job build :

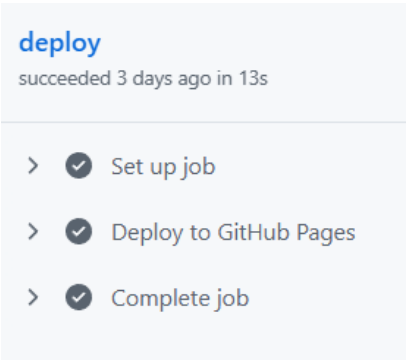
- Installe Doxygen via apt
- Vérifie la présence d'un fichier Doxyfile dans "docs/doxygen/"
- Redéfinit dynamiquement certains paramètres ("OUTPUT_DIRECTORY=site", "HTML_OUTPUT=.") pour que la documentation soit générée dans le répertoire "site/" (compatible GitHub Pages)
- Exécute la commande doxygen avec la configuration ainsi modifiée
- Uploade l'artefact généré (site/) en tant que contenu statique à publier



Étapes des jobs GitHub Actions pour la documentation

Le job deploy :

- Configure l'environnement GitHub Pages (via actions/configure-pages)
- Télécharge l'artefact généré précédemment (documentation HTML)
- Le déploie automatiquement sur la branche gh-pages (utilisée par GitHub Pages pour publier le site)



Déploiement du site sur GitHub Page

Lorsque la génération et la publication se déroulent correctement, la documentation est immédiatement disponible et à jour sur GitHub Pages, sans action manuelle. Cela garantit que les modifications de code ou de structure sont toujours accompagnées d'une documentation synchronisée, renforçant ainsi la maintenabilité et la transparence du projet.

Exemple documentation en ligne :

Aperçu de la documentation générée par Doxygen sur GitHub Pages

6. Difficultés rencontrées et solutions :

Le développement de ces fonctionnalités au sein de notre projet a pu présenter un certain nombre de difficultés techniques. Voici les principaux défis auxquels nous avons dû faire face, ainsi que les solutions apportées pour chacun.

Tests backend avec Firestore

L'utilisation de Firestore dans l'application a complexifié les tests backend. En effet, tester des fonctionnalités dépendant d'une base externe en temps réel pose des problèmes de sécurité des données, notamment au niveau de la clé FireStore. Nous avons résolu ce problème en mettant en place un secret CI sur Github pour permettre aux tests de se connecter au Firestore de test sans divulguer la clé.

Tests frontend et simulation du DOM

Tester le code front-end sans navigateur a également été un défi. Les modules JS du projet interagissent avec le DOM (document HTML) et certaines APIs globales du navigateur. Nous avons ainsi dû utiliser JSDOM (fourni par Jest) pour créer un environnement DOM factice. Même avec cela, tester les interactions utilisateur (clics, soumissions de formulaire) ne fut pas simple. En effet, JSDOM ne gère pas le rendu visuel mais permet de manipuler les éléments et de déclencher des événements. Les tests ont donc dû créer du HTML représentatif puis appeler les fonctions du module “vote.js” qui attachaient des handlers, afin de simuler un clic et vérifier les effets.

Intégration continue

Mettre en place la CI à présenté quelques difficultés, notamment pour reproduire un environnement proche de la prod. Par exemple, il a fallu s'assurer que la configuration de Firebase marche en CI (d'où le travail sur le secret et la reconstruction du fichier JSON).

Aussi, le pipeline devant installer à la fois Python et Node, il a fallu veiller à ce que chaque job installe les bonnes versions et dépendances.

Couverture documentaire et mise à jour des docs

Produire la documentation automatique a également ses défis. L'un d'eux fut de s'assurer que tout le code important est documenté de manière à apparaître dans la doc. Si on oublie de commenter une fonction publique, elle risque de ne pas figurer dans la sortie Doxygen (puisque EXTRACT_ALL est off). Par conséquent, il a fallu parcourir le code et ajouter des commentaires Doxygen partout ce qui est un investissement en temps non négligeable.

Le recours à des pratiques modernes (tests, mocks, CI, documentation continue) a certes introduit de la complexité, mais nous avons su la maîtriser et en tirer profit pour livrer un projet plus fiable, maintenable et en phase avec les standards professionnels actuels.

IV. Manuel utilisateur et fonctionnalités :

1. Utilisation du logiciel :

2.

Solution 1 depuis VSCode ou toute application similaire :

Pour utiliser notre application de planning poker, il faut lancer depuis un terminal :

git clone https://github.com/planningpokerteams/planningpoker.git

cd planningpoker

Ensuite, placer le JSON fourni dans le dossier récupéré pour l'accès à Firebase.

Il faut ensuite installer les bibliothèques Python nécessaires :

pip install -r requirements.txt

(ou, si vous utilisez un environnement virtuel, l'activer avant puis exécuter la même commande).

Une fois les dépendances installées, on peut lancer le serveur Flask :

python app.py

Le serveur démarre sur une adresse de type 'http://127.0.0.1:5000/'. Il suffit alors d'ouvrir cette URL dans un navigateur (Chrome, Firefox, etc.) pour accéder à l'interface web du planning poker.

Solution 2 depuis GitBash :

Cloner le projet depuis GitBash

git clone https://github.com/planningpokerteams/planningpoker.git

cd planningpoker

Ensuite, placer le JSON fourni dans le dossier récupéré pour l'accès à Firebase.

Rendre le script exécutable :

chmod +x run.sh

Lancer le script avec :

./run.sh

Le script se charge d'exécuter les commandes nécessaires (installation des dépendances, lancement du serveur Python/Flask, etc.).

Solution 3 depuis le WEB:

Aller sur : https://planningpoker-q3jm.onrender.com/

Attention ! Le déploiement étant fait avec la version gratuite, il se peut que lorsque vous accédez au lien, le déploiement se relance. Il faudra attendre quelques minutes pour que le site soit prêt !

3. Modes de jeu implémentés :

Notre application propose plusieurs modes de jeu, mais ils ont tous le même premier tour. Celui-ci est toujours en mode strict, c'est-à-dire à l'unanimité. La story courante commence donc par un tour 1 où chacun choisit une carte du deck (1, 2, 3, 5, 8, 13, …), puis l'outil révèle les votes. Si toutes les cartes numériques sont identiques (en ignorant les '?' et les cafés), la story est directement considérée comme estimée, alors, on enregistre cette valeur comme résultat final, et on peut passer à la story suivante. Si les votes sont différents, on ne fixe pas encore d'estimation et l'on passe au tour deux sur la même story. C'est à partir de ce tour deux que les modes de jeu se différencient réellement.

Dans le mode strict, ce comportement d'unanimité reste valable pour tous les tours. Après le premier tour, les joueurs discutent, revotent, et à chaque révélation l'application vérifie de nouveau si toutes les cartes numériques sont identiques. Tant qu'il y a des valeurs différentes, aucune estimation n'est validée, on lance la discussion puis on revote. La story n'est donc validée que lorsque tout le monde se met d'accord sur la même carte.

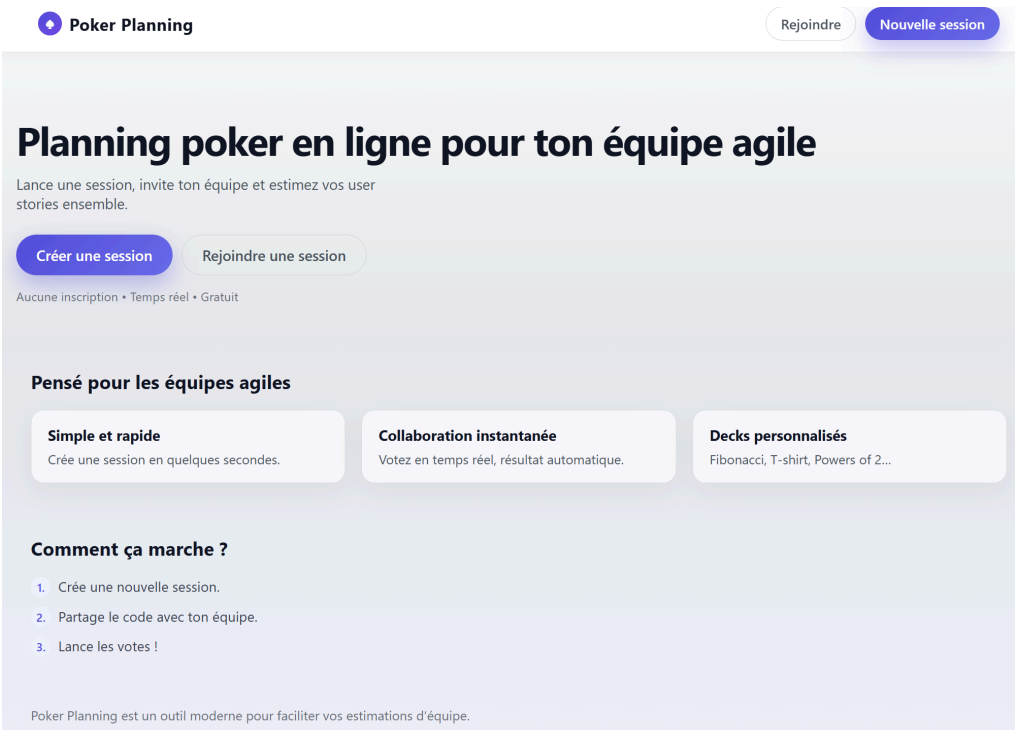
Dans le mode moyenne, le premier tour fonctionne comme en strict, mais à partir du second tour, si les votes restent différents, l'outil calcule la moyenne des votes numériques. Les cartes spéciales ('?' et 'café') sont exclues de ce calcul, on ne garde que les valeurs du deck (1, 2, 3, 5, 8, 13). La fonction de calcul détermine d'abord la moyenne exacte, puis choisit la carte du deck la plus proche de cette moyenne.

Le mode médiane suit la même logique, mais en s'appuyant sur la médiane plutôt que sur la moyenne. À partir du tour 2, si les votes sont encore dispersés, on extrait seulement les votes numériques, on les trie, après on calcule la médiane, puis on sélectionne la carte la plus proche qui sera l'estimation finale.

Les deux autres modes implémentés sont les modes de majorité. En mode 'majorité absolue', on cherche à partir du second tour une valeur qui obtient plus de 50 % des votes numériques. Le script compte le nombre d'occurrences de chaque carte (toujours en écartant '?' et 'café'). Si une valeur dépasse la barre des 50 %, elle est choisie comme estimation finale de la story. Si aucune carte n'atteint cette majorité absolue, la story n'est pas encore fixée et on peut relancer un nouveau tour ou rediscuter. En mode 'majorité relative', les tours suivants sélectionnent comme vote final la valeur la plus votée comme estimation. En cas d'égalité parfaite entre deux ou plusieurs cartes, on laisse la main à l'organisateur.

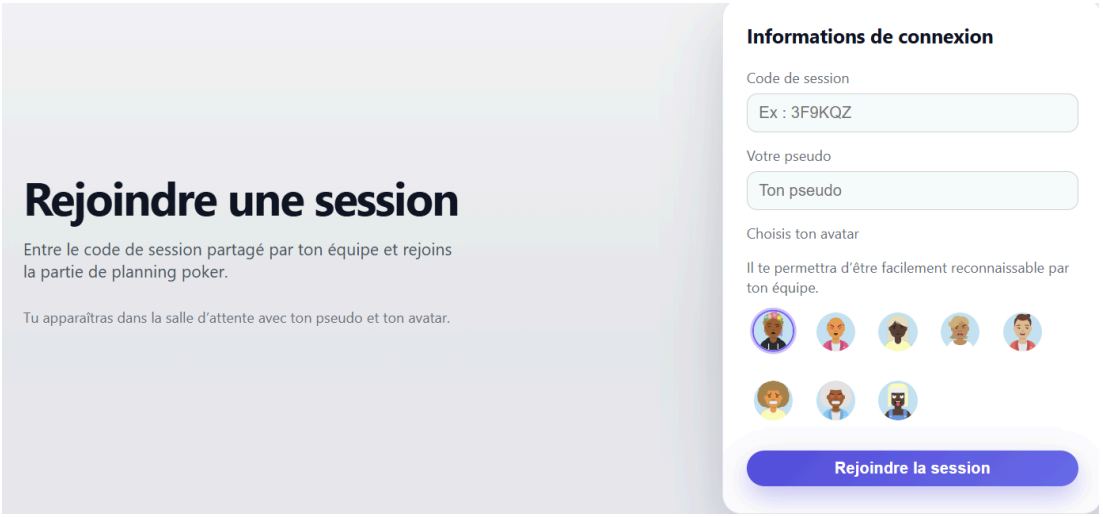
Dans tous ces modes, la gestion des cartes spéciales reste la même. La carte café sert à indiquer qu'un joueur fait une pause ou ne souhaite pas participer à l'estimation sur ce tour. La carte '?'signale au contraire qu'on ne sait pas. Ces deux valeurs sont toujours ignorées dans le calcul de l'estimation. En revanche, si tout le monde joue la carte café, la session passe automatiquement en pause, ce qui invite l'équipe à s'arrêter ou à reprendre plus tard, tout en permettant à l'organisateur de télécharger le JSON de la partie en cours.

4. Présentation de l'interface et flux d'une partie



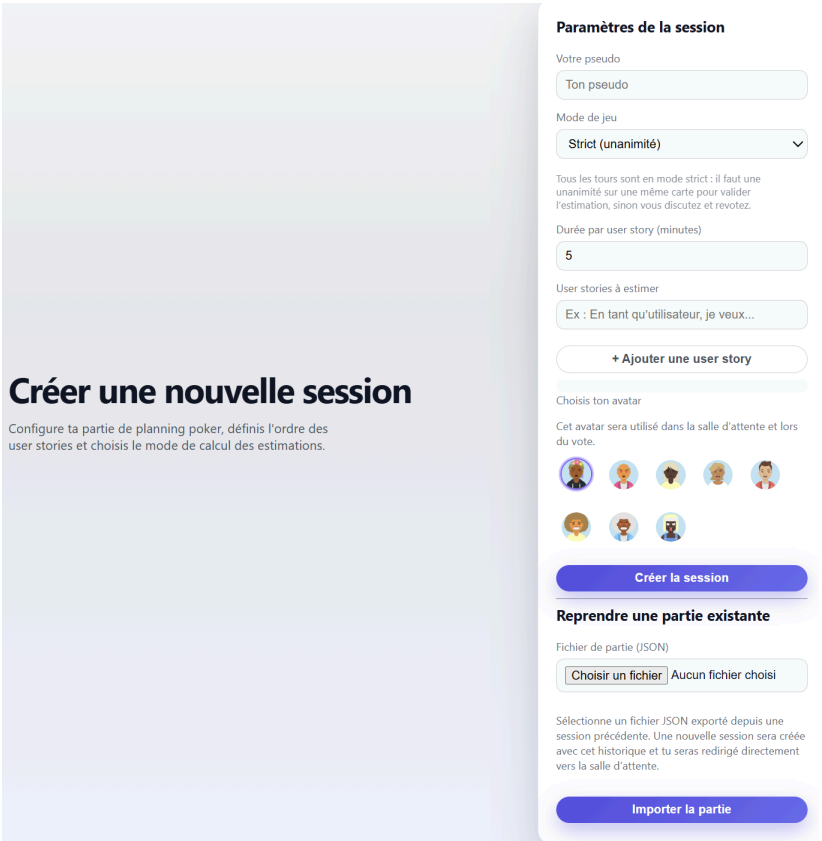
Page d'accueil du site

Sur la page d'accueil, l'utilisateur arrive sur un écran qui présente le principe de l'outil Planning poker. Deux actions principales sont mises en avant au centre de la page : créer une session et rejoindre une session.



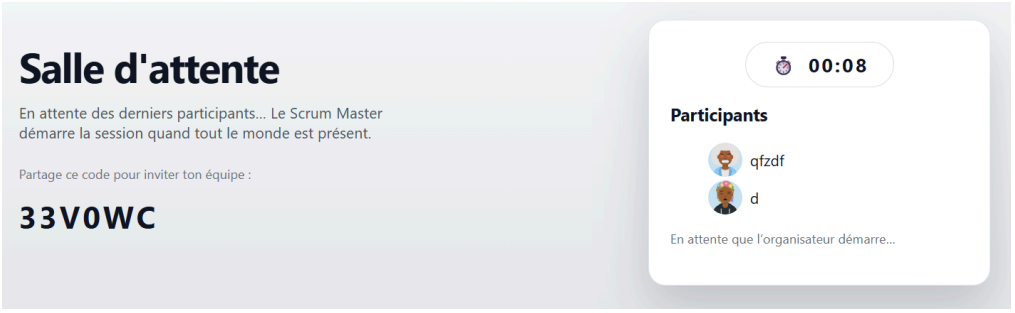
Page pour rejoindre une partie

L'écran de rejoindre une partie comporte un formulaire simple où l'utilisateur renseigne le code de session transmis par l'organisateur, son pseudo et choisit son avatar. Une fois ces informations saisies, le bouton « Rejoindre la session » permet d'entrer dans la partie en cours.



Page de création d'une partie

L’écran de création de partie permet de configurer une nouvelle session. Sur la partie droite, un formulaire que doit remplir l’organisateur avec son pseudo, le mode de jeu choisi, la durée par story et le choix de l’avatar. Il a aussi la possibilité de rentrer à la main ses users stories en validant avec le bouton «Ajouter une User Story» à chaque fois et une fois qu’on a terminé, cliquer sur « Créer la session » pour lancer la partie. Sinon, en bas du formulaire, il peut reprendre une partie existante via l’import d’un fichier JSON en cliquant sur « Choisir une fichier», qui va ouvrir l’explorateur de fichiers pour qu’on sélectionne notre fichier puis, il faudra cliquer sur « Importer la partie » pour lancer le jeu.

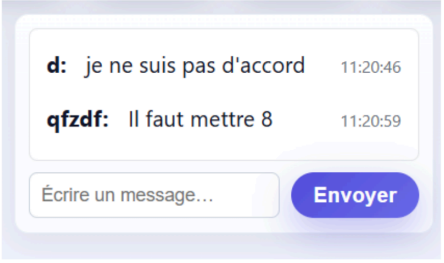


Salle d’attente (côté organisateur)

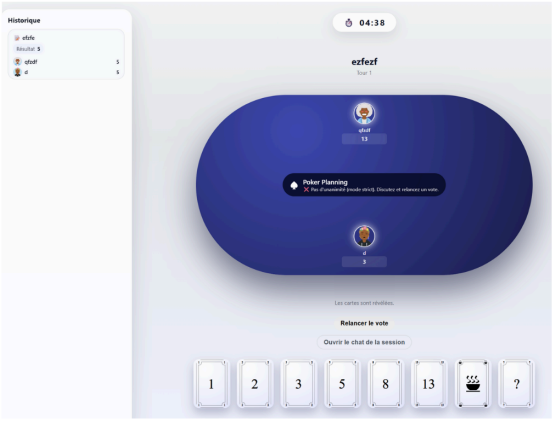
La capture permet de voir la salle d’attente côté organisateur (côté joueur on a la même sans le bouton « Lancer la partie »). L’organisateur peut suivre le temps qu’il a attendu avant de lancer une partie et peut lancer la partie quand tout le monde est là.

Page de vote - Révélation des votes

Page de vote - Pause café



Page de vote - Chat de discussion



Page de vote – Divergence des votes

Les quatre images ci-dessus montrent l'interface de vote et ses différentes options.

En haut à gauche, on voit la table de jeu avec les avatars des participants disposés autour d'une table de jeu. On a aussi le minuteur du temps de jeu autorisé. En bas de l'écran, les cartes de vote. Une fois que tout le monde a voté, un message apparait confirmant que tous les participants ont voté, et un bouton « Révéler les cartes » apparait auprès de l'organisateur.

En haut à droite de l'image illustre le cas où tous les participants ont choisi la carte café. Un message indique que tous les joueurs ont choisi la carte café et que la partie est en pause. L'interface propose alors à l'organisateur deux actions : « Reprendre la partie » pour revenir au jeu ou « Exporter l'état de la partie (JSON) » pour sauvegarder la session courante.

En bas à gauche, on voit le chat de discussion qui apparait chez les joueurs comme chez l'organisateur quand les votes divergent. Celui-ci disparait quand l'organisateur relance le vote (dernière image) pour l'organisateur et lorsqu'une carte est validée pour les joueurs. En bas à droite on voit aussi l'historique qui permet de suivre les votes de chacun et l'estimation retenue pour chaque user story.

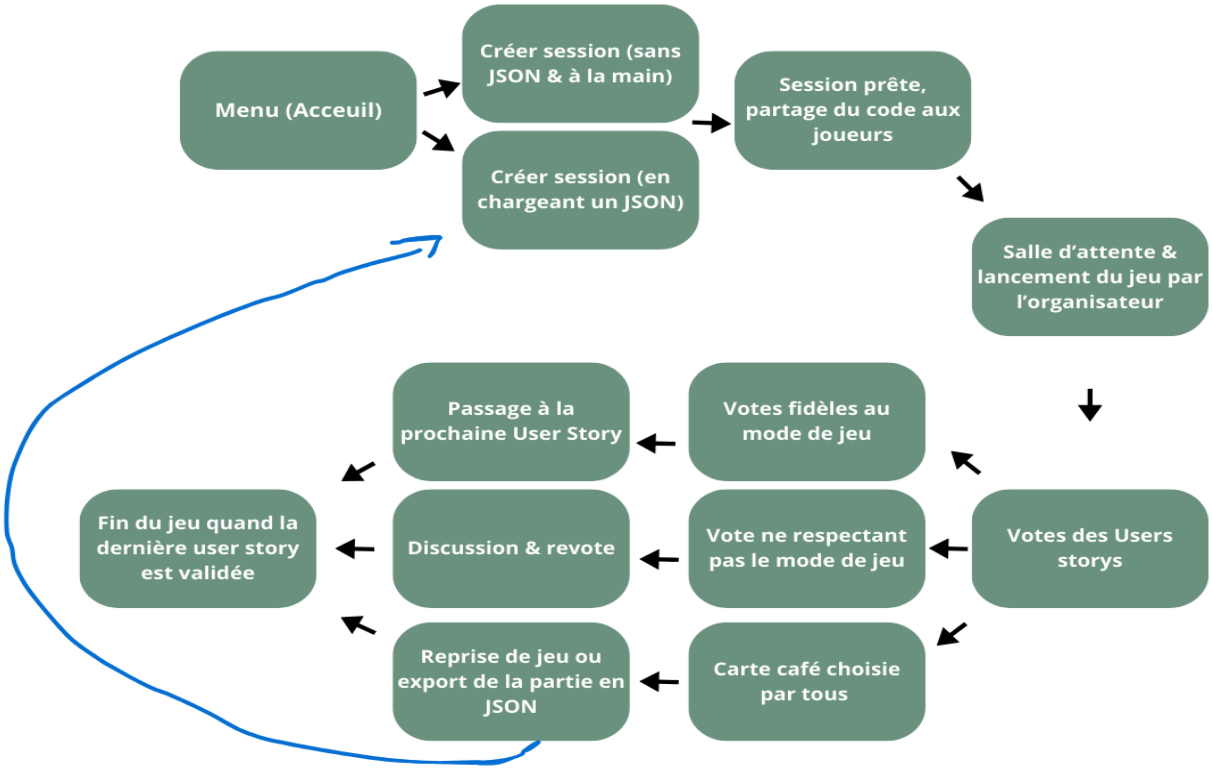


Schéma du flux d'une partie

Le flux d'une partie suit le schéma ci-dessus. Depuis le menu d'accueil, l'organisateur peut soit créer une nouvelle session manuellement, soit créer une session en important un fichier JSON issu d'une partie précédente. Une fois la session prête, il partage le code aux joueurs, qui vont dans la salle d'attente avant le lancement du jeu.

Quand la partie commence, les joueurs procèdent aux votes sur les user stories. Si tous les votes sont cohérents avec le mode de jeu, on passe simplement à la prochaine user story. Si les votes ne respectent pas le mode, alors une phase de discussion et de revote s'ouvre.

Les joueurs peuvent aussi choisir collectivement la carte café, ce qui met la partie en pause. À ce moment-là, l'organisateur peut reprendre le jeu plus tard ou exporter l'état de la partie en JSON pour le sauvegarder. S'il exporte le JSON, il peut reprendre la boucle à la création d'une nouvelle partie à partir du JSON. On répète ce cycle user story/votes/revotes et discussion jusqu'à la validation de la dernière user story qui termine la partie.