# Rapport : Projet CAPI

Aymene Belmeguenai
Eren Yetkin

## 1 Introduction

Le Planning Poker est une méthode utilisée pour estimer la complexité de tâches de développement. La méthode consiste à l'estimation de la difficulté des tâches par les membres de l'équipe, l'estimation se fait en choisissant un nombre d'abord anonymement puis en la dévoilant en même temps que tout le monde, cette méthode permet d'ouvrir la discussion.

Pour la création du planning poker, nous avons utilisé Python, son framework Django et les WebSockets pour une communication en directe entre les participants, permettant d'avoir une interface fluide.

## 2 Choix techniques

### 2.1 Python et Django

Le choix de Python a été assez naturel : nos expériences personnelles nous ont conduit à travailler avec Python sur de nombreux projets, nous sommes tous les deux à l'aise et c'est ce que nous cherchions étant donné le nombre de projets à rendre en cette fin d'année, nous ne voulions pas nous compliquer la tâche à gérer des erreurs sur des langages qui nous sont étrangers.

Nous avons choisi Django comme framework Python pour le backend car il est très complet et nous permet de gérer facilement les utilisateurs, les sessions, les bases de données, les formulaires, les templates, les URLs, etc. Nous avons d'ailleurs utilisé Django Channels pour la partie WebSocket de notre application, expliquée dans le prochain paragraphe.

### 2.2 WebSockets

Il nous fallait faire communiquer les utilisateurs en direct, pour cela les WebSockets étaient essentiels pour ce projet. Cette technologie permet aux

clients et au serveur de maintenir une connexion ouverte, facilitant ainsi l'échange de messages instantanés comme les votes ou les mises à jour d'états des sessions. Grâce à Django Channels, nous avons pu intégrer les WebSockets dans notre application.

### 2.3 Doxygen pour la documentation

Pour garantir une documentation claire et maintenable, nous avons utilisé Doxygen. Cet outil a permis de générer automatiquement des documents à partir des commentaires dans le code. En plus de faciliter la collaboration entre les développeurs, cette documentation constitue une base solide pour les équipes souhaitant reprendre ou améliorer le projet.

Nous avons également rendu cette documentation disponible en ligne afin de faciliter son accès : `https://eren-nere.github.io/ProjetCAPI/index.html`. La génération de cette documentation a été intégrée dans notre pipeline, nous permettant ainsi de toujours disposer d'une version à jour.

## 3 Difficultés rencontrées : WebSockets

L'implémentation des WebSockets avec Django Channels a posé des problèmes, notamment sur la gestion des groupes et l'envoi des événements. La logique d'ajout, de suppression et de synchronisation des joueurs dans les groupes a demandé des ajustements et un débogage approfondi. Un autre défi majeur a été de garantir la stabilité des connexions tout en minimisant les conflits liés aux états des sessions.

## 4 Tests unitaires

L'écriture des tests unitaires pour vérifier la logique de l'application était complexe en raison des interactions en temps réel et de l'état de la base de données. Les tests ont nécessité des bases temporaires et l'utilisation d'outils comme `Mock` pour simuler les WebSockets. Les scénarios impliquant plusieurs utilisateurs simultanés ont également été difficiles à reproduire.

Nos tests unitaires vérifient la cohérence des modèles, s'assurent que les données sont correctement insérées en base de données et simulent un scénario complet de création de `room` puis de vote via WebSockets. Ces tests garantissent que notre logique métier fonctionne comme prévu même dans un contexte temps réel.

# 5 Explication de la mise en place de l'intégration continue

Nous avons mis en place une intégration continue (CI) afin de nous assurer que chaque modification du code est testée et validée automatiquement avant d'être intégrée. Pour cela, nous avons utilisé **GitHub Actions** car gratuite et facile à configurer, qui exécute nos tests à chaque *push* sur le dépôt. Ainsi, en cas de régression, nous sommes immédiatement informés, ce qui nous permet de corriger rapidement les problèmes.

# 6 Choix techniques supplémentaires (Architecture, Langage, Classes)

L'architecture retenue, basée sur Django, repose sur une séparation entre la couche logique (modèles, vues, consumer) et la couche de présentation (templates et interaction WebSocket côté client).

Le modèle de données est conçu pour refléter les concepts métier. Une entité représentant une « room » (salle de planification) stocke le backlog des fonctionnalités à estimer, le mode de décision (unanimité ou majorité absolue) et l'historique des fonctionnalités estimées. Les joueurs, quant à eux, sont gérés comme des entités distinctes, associées à une room, avec un champ de vote permettant d'enregistrer leur estimation. Cette modélisation facilite les opérations de base de données (par exemple, l'enregistrement d'un vote) et contribue à une maintenance aisée : les classes sont cohérentes avec le domaine fonctionnel, ce qui limite le risque de confusion ou de redondance.

## Asynchronisme et gestion des interactions en temps réel

L'utilisation de *consumers* asynchrones, spécifiques à Django Channels, permet de traiter les événements reçus via WebSockets en temps réel. Chaque *consumer* est dédié à une fonctionnalité précise, comme la gestion des votes ou la mise à jour des fonctionnalités du backlog. L'approche asynchrone garantit que les opérations (telles que la récupération d'informations en base de données ou la diffusion des résultats aux joueurs) s'effectuent sans bloquer l'application, améliorant l'expérience des utilisateurs.

### Organisation interne du code

La logique est répartie de façon à renforcer la clarté du code :

— **Couches logiques :** Les modèles définissent le domaine fonctionnel (rooms, joueurs, votes), tandis que les *consumers* gèrent la logique temps réel (réception des votes, révélation des résultats, progression dans le backlog). Les vues classiques (HTTP) fournissent des pages initiales ou des endpoints simples.

— **Paramétrage et configuration :** Les fichiers de configurations (comme `settings.py` et `urls.py`) organisent l'espace de noms, les routes, le chargement des applications et l'intégration de Channels.

— **Évolutivité du modèle :** En définissant les entités (par exemple, la room et son backlog) sous forme d'objets simples possédant leurs attributs et comportements, il est facile d'ajouter de nouvelles fonctionnalités, de nouveaux modes de vote, ou de nouvelles règles d'estimation, sans perturber l'ensemble de l'architecture.

# 7 Interface et interaction utilisateur supplémentaires

Le projet est accessible à l'adresse : `http://server1.aymene.tech:8999/`

Étant donné que le site n'est pas en HTTPS, la fonction de copier un lien via un bouton peut ne pas fonctionner correctement. Pour rejoindre une partie, l'utilisateur doit copier l'URL de la `room` puis ajouter `/join` à la fin.

Par exemple, si l'URL de la room est : `http://server1.aymene.tech:8999/poker/room`

l'utilisateur doit naviguer vers : `http://server1.aymene.tech:8999/poker/room/join`

# 8 Explication du code principal

Le code de l'application repose sur un système de WebSockets pour gérer les différentes interactions en temps réel. Voici les points clés :

## 8.1 Connexion des utilisateurs

Les utilisateurs se connectent à une session identifiée par un nom de salle (`room_name`). Lors de leur connexion, ils rejoignent un groupe associé à cette salle via Django Channels. Le consumer correspondant gère cette logique en

ajoutant les utilisateurs au groupe et en assurant la synchronisation de leur état.

### 8.2 Gestion des votes

Les utilisateurs soumettent leurs votes via des messages envoyés au serveur. Ces votes sont ensuite enregistrés dans la base de données. Une fois que tous les membres ont voté, les résultats sont analysés puis renvoyés à tous les participants. Le serveur peut ainsi révéler les votes et déterminer si une condition (unanimité, majorité) est remplie, ce qui influe sur la progression du backlog.

### 8.3 Révélation des votes

Une fois tous les votes reçus, le serveur calcule s'il existe une unanimité ou une majorité suffisante. En cas de succès, la fonctionnalité actuelle est estimée et l'application passe à la suivante. En cas d'échec, un nouveau vote est nécessaire, prolongeant la discussion et l'estimation collective.

### 8.4 Interface utilisateur

L'interface frontend affiche en temps réel : les votes, les joueurs n'ayant pas encore voté, les fonctionnalités à estimer et leur état d'avancement. Des alertes indiquent aux utilisateurs si le vote est validé ou s'il doit être recommencé, offrant ainsi une expérience transparente et réactive.

## 9 Exemple d'interaction utilisateur

Lorsqu'un utilisateur clique sur le bouton « Révéler les votes », un message de type `reveal` est envoyé au serveur. Le backend traite ce message, vérifie l'unanimité des votes et retourne un message au client avec les résultats. Si tous les utilisateurs ont voté de la même manière, la fonctionnalité est validée et l'on passe à la suivante. Sinon, un nouveau vote est demandé.

## 10 Conclusion

Ce projet a permis d'explorer les avantages de Django et des WebSockets. Des difficultés ont été rencontrées comme expliqué plus haut, mais elles nous ont permis d'enrichir notre compréhension des interactions en temps réel.

Nous avons un sentiment de réussite avec ce projet, que nous avons réalisé selon nos objectifs initiaux.