

[Théo Tachdjian](#)

Dossier de projet



Titre Professionnel de Concepteur
Développeur d'Applications

Sommaire

I. Résumé du projet.....

II. Cahier des charges.....

1. Description du cahier des charges du projet.....
2. Fonctionnalités de l'application.....
3. Mécaniques de jeu.....
4. Évolution du cahier des charges.....

III. Conception.....

1. Architecture du projet.....
2. Technologies utilisées.....
 - A. Backend.....
 - Recherche de frameworks et bibliothèques.....
 - L'API.....
 - L'ORM.....
 - Le système de base de données.....
 - Le service de Jeu.....
 - B. Frontend.....
3. Structure de la base de données.....
 - A. Modèle conceptuel de données.....
 - B. Modèle logique de données.....
 - C. Modèle physique de données.....

4. Maquettage.....

5. Pages du site.....

6. Routes de l'API.....

- A. Authentification.....
- B. Comptes.....
- C. Gestion d'amis.....
- D. Récupération de compte.....
- E. Salles de jeu.....
- F. Service de Jeu.....
- G. Server-Sent-Events.....

7. Définition des cartes.....

- A. Variantes de cartes.....
- B. Assets.....

8. Le Jeu.....

- Actions et réponses du serveur.....

9. Conteneurisation.....

10. Déploiement.....

IV. Organisation du projet.....

1. Repository et Versionning.....
2. Organisation des tâches.....
3. Outils de développement.....
4. Canaux d'échanges et partage de ressources.....
5. Directives pour l'écriture du code.....

V. Développement de l'application.....

1. Définition de l'API.....
 - A. Fonctionnement de l'API.....
 - B. Définition des routes.....
 - Macros.....

Fonctions asynchrones.....	
Extraction des données.....	
Données d'application.....	
Connexion à la base de données.....	
Configuration.....	
C. Sécurité.....	
Hachage des mots de passe.....	
Middleware d'authentification.....	
JWT.....	
Garde CORS.....	
D. Documentation de L'API.....	
2. La base de données.....	
A. Création automatique des tables.....	
B. Modèles des tables en Rust.....	
C. Requêtes avec Diesel.....	
3. Composants du site.....	
A. L'App State.....	
B. Garde de connexion.....	
C. Système de cache des entités.....	
D. Composants réutilisables.....	
E. Les vues et panels.....	
Les vues.....	
Les panels.....	
4. Salles de jeu.....	
A. Créer une salle de jeu.....	
B. Rejoindre une salle.....	
C. Liste des pages.....	
D. Mises à jour dynamiques.....	
Envoi des events.....	
Écoute des events.....	
5. Gestion des amis.....	
A. Relation entre comptes.....	
B. Envoi d'une demande d'ami.....	
6. Service de Jeu.....	
A. Définition des cartes.....	
Le trait Card.....	
Utilisation des traits en Rust.....	
Fonction pour jouer une carte.....	
Utilisation des cartes dans le Frontend.....	
B. Chargement des cartes.....	
C. Plan de communication.....	
D. Procédure d'un tour de jeu.....	
E. Chat textuel de jeu.....	
F. Mises à jour de la scène.....	
7. Media Queries.....	
8. Récupération de compte.....	
A. Processus de récupération.....	
B. Envoi d'email.....	
9. Déploiement du projet.....	
Annexes.....	
Schémas de conception du projet.....	
Schémas d'organisation du projet.....	

Documentations relatives à l'API.....	
Définitions des modèles de tables en Rust.....	
Développement du site web.....	
Actions et réponses du serveur de jeu.....	

I. Résumé du projet

Le projet a été réalisé en centre par moi-même et deux autres étudiants. Il se présente comme étant une adaptation d'un jeu de cartes en jeu vidéo en 3D, jouable en ligne depuis un navigateur.

Le projet se nomme **Randomi GO**. Il tire son nom du jeu sur lequel il est basé.

Le jeu de cartes Randomi est un jeu qui a été conçu et commercialisé en 2020 par l'un des membres de mon équipe.

Le projet a été réalisé en l'espace d'environ 9 semaines, en centre, hors semaines passées en entreprise en apprentissage.

II. Cahier des charges

1. Description du cahier des charges du projet

L'objectif du projet est de créer un portage du jeu de cartes Randomi en tant qu'application web.

Les points principaux de ce projet seront:

- De pouvoir jouer une partie en ligne avec d'autres joueurs (2 à 6)
- De respecter au mieux les règles et mécaniques du jeu original
- De permettre l'accessibilité au jeu à un plus grand nombre d'utilisateurs par le biais du navigateur
- De pouvoir y jouer sur les appareils type Mobile et Desktop

Le projet sera composé de 3 parties:

- Un site web
- Une API couplée à une base de données SQL pour la gestion des utilisateurs et des interactions avec le Front.
- Un service de jeu pour la gestion des actions d'une partie et de la communication entre les joueurs.

Le site web sera la partie Frontend de l'application.

L'API, le service de jeu et la base de données feront partie du même cluster et composeront la partie Backend de l'application.

2. Fonctionnalités de l'application

En dehors du fait de pouvoir jouer une partie, les utilisateurs pourront:

- S'enregistrer et s'identifier via des pages dédiées
- Créer et rejoindre des salles de jeu
- S'ajouter en amis et rejoindre les salles de jeu de leurs amis directement depuis un panel dédié
- Demander une récupération de compte par l'envoi d'un mail (changement de mot de passe)

Afin de proposer une expérience de jeu plus attractive, le jeu devra être présenté en 3D grâce à WebGL, supporté par la majorité des navigateurs.

3. Mécaniques de jeu

Le jeu devra supporter les mécaniques suivantes:

- Gain / Retrait de points de vie
- Pioche de cartes
- Calculs selon le score des lancers du dé
- Calculs selon le nombre de cartes dans la main ou la défausse du joueur ou de l'adversaire
- Choix des cibles
- Échange et vol de cartes
- Gestion des bonus (augmentation d'attaque pendant un tour, etc.)

Ces mécaniques font écho à celles du jeu d'origine, il sera donc important de les respecter lors de la conception du jeu.

4. Évolution du cahier des charges

Durant le développement du projet, nous avons dû réduire le nombre de fonctionnalités que nous voulions implémenter à la base. Cela est en partie dû à l'apprentissage de Rust, qui a une courbe d'apprentissage plus exigeante que d'autres langages de programmation.

Par exemple, nous avions initialement prévu d'utiliser *three-nebula* pour ajouter des effets de particules dans la scène, de pouvoir disputer des parties hors connexion contre une IA, ou encore d'avoir un magasin d'éléments cosmétiques payant.

Des restes de certaines de ces fonctionnalités sont encore présentes dans le projet: les données des actions retournées par le service de jeu ont un champ "*effect_id*", prévu pour être utilisé pour la création de particules.

III. Conception

1. Architecture du projet

Voici un schéma décrivant l'architecture globale du projet et les interactions entre les différents composants:

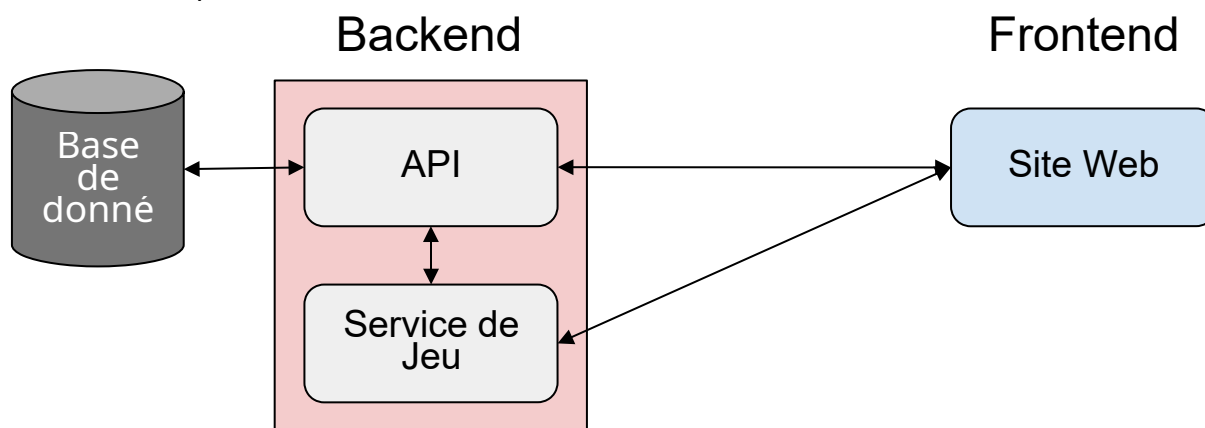


Schéma de l'architecture du

L'API aura un accès exclusif à la base de données: elle ne sera accessible que depuis un réseau privé où l'API et la base de données seront connectées.

2. Technologies utilisées

A. Backend

Pour ce projet, nous avons décidé de nous donner un challenge supplémentaire en utilisant le langage Rust pour la réalisation du Backend, qu'aucun d'entre nous n'avait utilisé jusqu'à présent.

Le langage Rust est connu pour être assez strict sur la manière dont on peut utiliser les données et les variables que l'on déclare dans le code, mais il est également connu pour sa capacité à fournir un moyen d'écrire du code fiable et sécurisé.

Cela est justifié par ses 3 concepts clés:

- Toutes les variables sont immuables par défaut; elles ne sont pas modifiables une fois déclarées. Pour déclarer une variable comme étant mutable, il faut ajouter un mot clé lors de sa déclaration.
- L'appartenance (ou ownership): chaque valeur a un propriétaire unique. Il faut soit copier la valeur, soit utiliser une référence à celle-ci pour pouvoir l'utiliser à plusieurs

endroits.

Si le propriétaire de la valeur n'est plus accessible, elle est supprimée.

- L'emprunt de variables (ou borrowing): permet de référencer une variable sans en prendre possession. On ne peut faire qu'un seul emprunt de référence à une variable à la fois.

Ces 3 concepts forcent les développeurs à spécifier comment une fonction est capable de modifier les variables qu'elle définit et qui lui sont fournies, réduisant grandement le risque de modification des données par inadvertance.

Le compilateur de Rust joue également un rôle essentiel car il applique ces concepts lors de l'analyse du code durant la compilation, réduisant ainsi les possibilités d'erreurs en runtime, et guide les développeurs vers une méthode de conception de leur code plus sûre.

Nous avons décidé d'apprendre ce langage pour non seulement monter en compétence, mais aussi pour découvrir et appliquer au mieux les concepts de sécurité que propose Rust.

Recherche de frameworks et bibliothèques

N'ayant jamais utilisé ce langage, nous n'avions aucune connaissance des frameworks et bibliothèques à utiliser pour la réalisation des composants du Backend.

Nous avons donc commencé nos recherches sur les bibliothèques que nous pourrions utiliser pour créer l'API et le service de jeu.

Nos critères de recherche étaient les suivants:

- Trouver un framework pour la création d'une API REST
- Préférer les frameworks offrant la possibilité de gérer des WebSockets pour la communication en jeu entre les clients et le serveur
- Trouver une bibliothèque pour faire de l'Object-Relational-Mapping (ORM) afin de pouvoir générer des requêtes SQL en utilisant des objets et des fonctions plutôt que d'écrire nous même les requêtes à exécuter
- Trouver une bibliothèque pour la génération de la documentation de l'API (si le framework ne nous fournit pas cette option)

L'API

Notre choix s'est porté sur le framework Actix Web, qui respectait la quasi-totalité des critères de recherches, si ce n'est qu'il ne fournissait pas de moyen de documenter l'API de lui-même.

Nous avons utilisé la bibliothèque Utoipa pour documenter l'API, car elle pouvait être intégrée à notre framework sans problèmes et fournissait une méthode simple et facilement configurable grâce à l'utilisation de macros.

L'ORM

Nous avons choisi la bibliothèque Diesel comme ORM. Un des aspects de Diesel que l'on a trouvé intéressant et qui nous a aidé à faire ce choix est la génération automatique de "schémas" des tables de la base de données en Rust, à partir de scripts de création de tables que Diesel appelle "migrations".

Ces schéma peuvent ensuite être appliqués à des structures Rust que nous pouvons créer nous mêmes pour définir les éléments à requêter à la base de données lors de l'utilisation de requêtes SQL comme *SELECT FROM*.

Diesel dispose d'un outil en ligne de commande pour la définition et la gestion des scripts de migration. C'est également depuis cet outil que l'on peut générer les schémas des tables.

Le système de base de données

Notre choix s'est porté sur les deux grands pionniers des systèmes de base de données relationnelle: PostgreSQL et MySQL, que nous avons déjà eu l'occasion d'utiliser durant notre formation.

Nous avons choisi d'utiliser PostgreSQL comme système de base de données relationnelle. L'un des atouts de PostgreSQL comparé à MySQL est sa plus grande collection de type de données.

La bibliothèque Diesel n'a pas joué un rôle important dans ce choix, car elle semblait supporter aussi bien PostgreSQL que MySQL à très peu de détails près. Il semblait de toute manière assez rapide d'adapter le code sans avoir à repasser sur la majorité des fonctions.

Le service de Jeu

Le service de Jeu fera partie de l'API. Les clients pourront s'y connecter via une requête à une route qui connectera le client à un WebSocket.

Actix Web nous fournit déjà la possibilité de créer et gérer des WebSockets, donc nous n'avons pas besoin de bibliothèques à rajouter pour ça.

Nous avons choisi d'utiliser les WebSockets pour la communication car ils offrent un canal de communication bidirectionnel, ce qui permet au serveur de pouvoir envoyer des données aux clients sans attendre leur demande et inversement.

La communication par WebSocket est par ailleurs supportée sur la grande majorité des navigateurs de nos jours.

B. Frontend

Le site sera décomposé en 2 parties: la partie navigation et la partie gameplay.

La partie navigation regroupe toutes les actions n'étant pas liées au jeu en lui-même (inscription, connexion, gestion des amis, etc.).

Elle ne sera basée sur aucun framework et sera faite “from scratch”, en utilisant JavaScript avec les bibliothèques et API standards que fournissent les navigateurs.

La partie gameplay sera développée en utilisant le framework Three.js. C’est un framework permettant de faire des rendus 3D dans un navigateur à l’aide de WebGL, qui est supporté par la majorité des navigateurs.

Il a une communauté active et est un framework assez mature, il y a donc beaucoup de chance que nous puissions trouver des réponses à des problèmes que nous rencontrerons lors du développement assez rapidement.

Il existe d’autres frameworks permettant de faire des rendus 3D dans un navigateur, comme Babylon.js ou même Unity. Ces frameworks se rapprochaient plus d’un moteur de jeu complet qu’un simple moyen d’afficher des éléments en 3D.

Cependant, en prenant en compte le défi que représente l’apprentissage du langage Rust, nous avons préféré rester sur Three.js pour sa courbe d’apprentissage plus généreuse que Babylon.js ou Unity.

Nous utiliserons l’outil Vite pour tester localement le site.

Vite est un outil s’exécutant avec Node.js, proposant un serveur de développement local pour tester notre site sur notre machine, et d’un système de build pour packager notre site et ainsi pouvoir le déployer sur un serveur HTTP, comme nginx par exemple.

Vite propose également un système de Hot Reloading, qui permet de mettre à jour automatiquement les pages et les modules lorsqu’ils sont modifiés.

Cet outil est l’un des premiers outils mis en avant dans la documentation de Three.js. Nous l’avons choisi car les fonctionnalités qu’il proposait étaient attractives et compatibles avec notre projet.

3. Structure de la base de données

La base de données sera utilisée pour stocker les comptes utilisateurs, leur statistiques, les relations entre les comptes (amis), et les tokens de réinitialisation de mot de passe:

- Les comptes utilisateurs sont définis par un nom d’utilisateur, un email, un mot de passe, et un champ indiquant si le compte est suspendu.
- Une relation entre les comptes (amis) est définie par une référence aux deux comptes et un statut (demande d’ami envoyée, acceptée, ou rejetée).
- Les tokens de réinitialisation de mot de passe sont définis par une suite de caractères, le compte auquel il est associé, sa date d’expiration et s’il a été utilisé ou non.
- Les statistiques d’un compte sont définies par le compte auquel ils sont associés, le niveau et l’expérience du compte, sa date de création, sa dernière date de connexion, son nombre de parties jouées et gagnées.

Nous avons utilisé la méthode Merise pour concevoir la base de données, en procédant en trois étapes:

A. Modèle conceptuel de données

Nous avons tout d'abord créé le modèle conceptuel des données en définissant les entités:

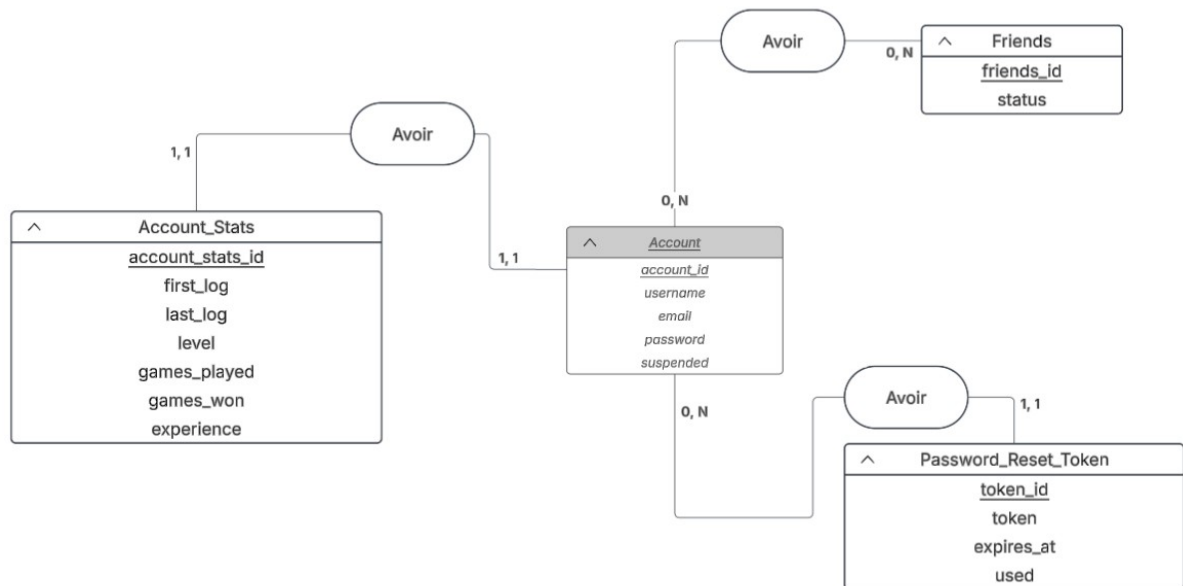


Schéma modèle conceptuel de données

B. Modèle logique de données

Nous avons ensuite créé le modèle logique de données en définissant une clé primaire pour chaque entité, permettant l'identification individuelle des lignes dans la base de données. Les attributs des entités ont été définis, de même pour la création des cardinalités de chaque entité:

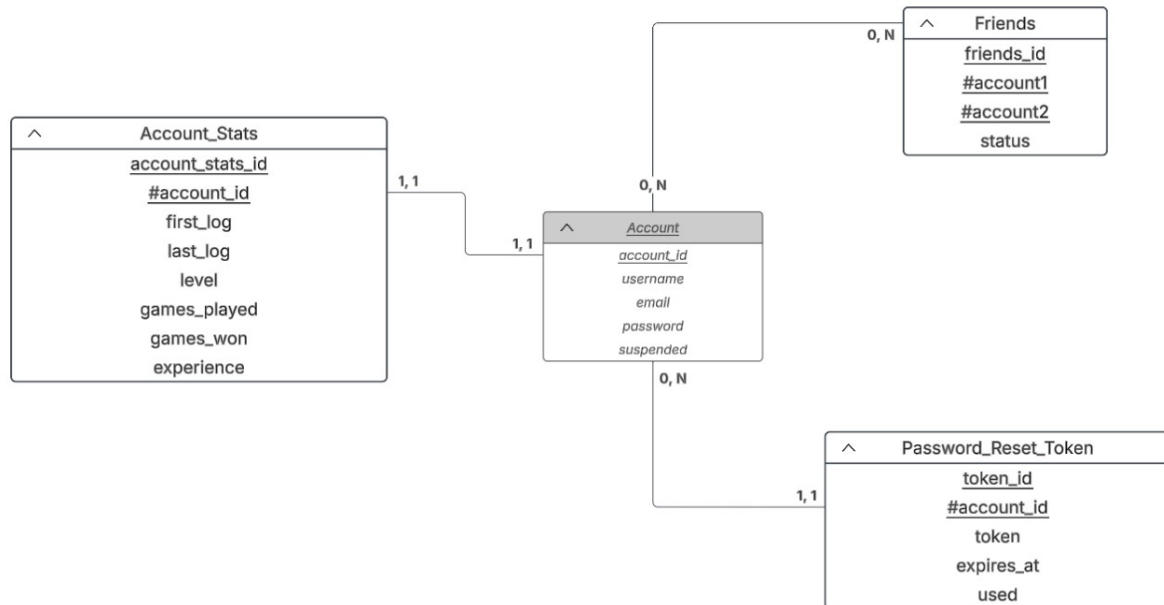


Schéma modèle logique de données

On remarque une relation réflexive entre la table *Account* et *Friends*.

Dans notre cas, il suffit de définir une contrainte la table *Friends* indiquant qu'on compte ne peut pas être ami avec lui-même en utilisant la règle *CHECK* (*account1* <> *account2*) lors de la création de la table.

C. Modèle physique de données

Nous avons typé les attributs de nos entités et défini les clés étrangères afin de garantir l'intégrité des références entre les tables:

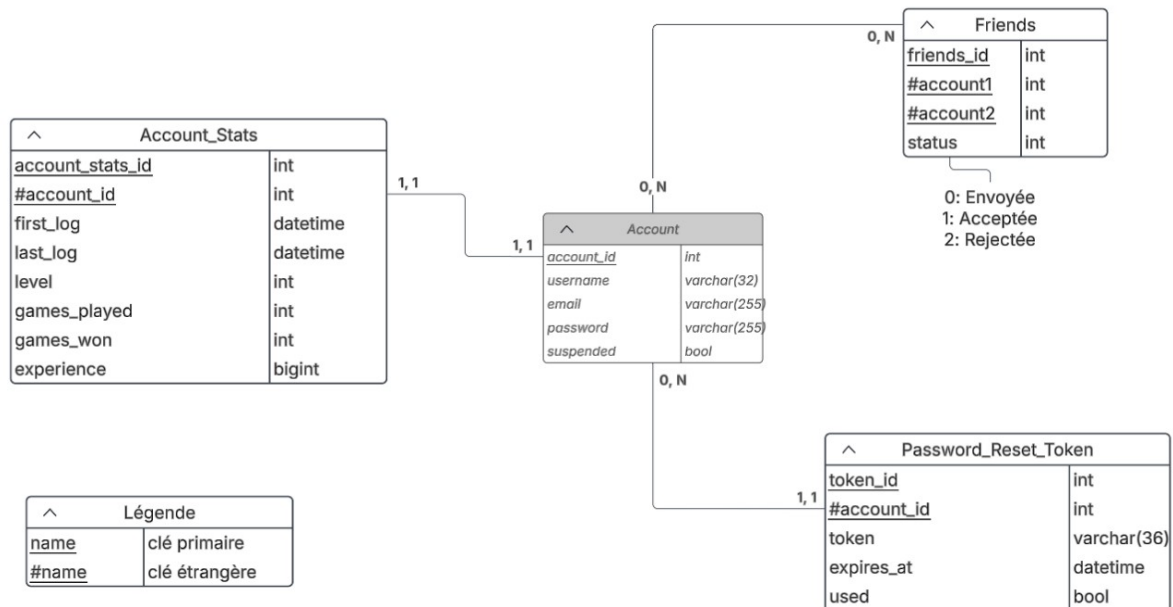


Schéma modèle physique de

Les tables *Account_Stats* et *Password_Reset* ont chacune une clé étrangère faisant référence à la clé primaire du compte qui leur est associé.

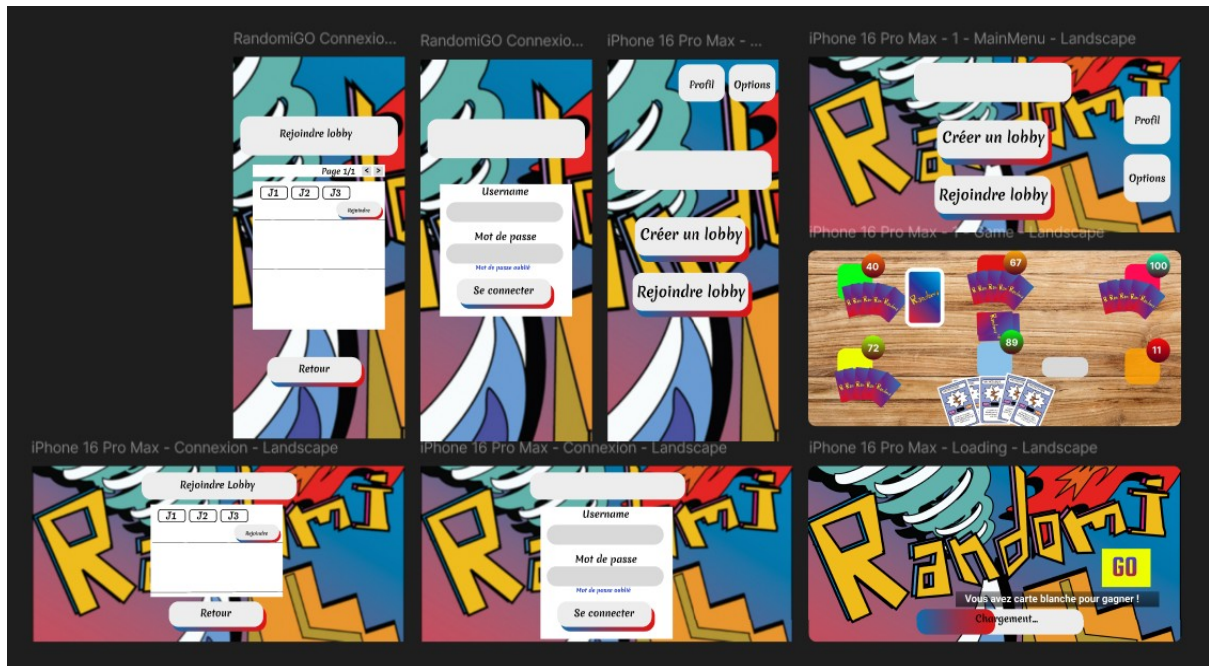
La table *Friends* a deux clés étrangères faisant référence aux deux comptes associés à la ligne.

Les champs définis dans la table *Account_Stats* ne sont actuellement pas tous utilisés en raison d'un manque de temps pour la réalisation de certaines fonctionnalités.

Hormis pour la clé primaire et étrangère, seuls les champs *first_log*, *games_played* et *games_won* sont utilisés pour l'instant.

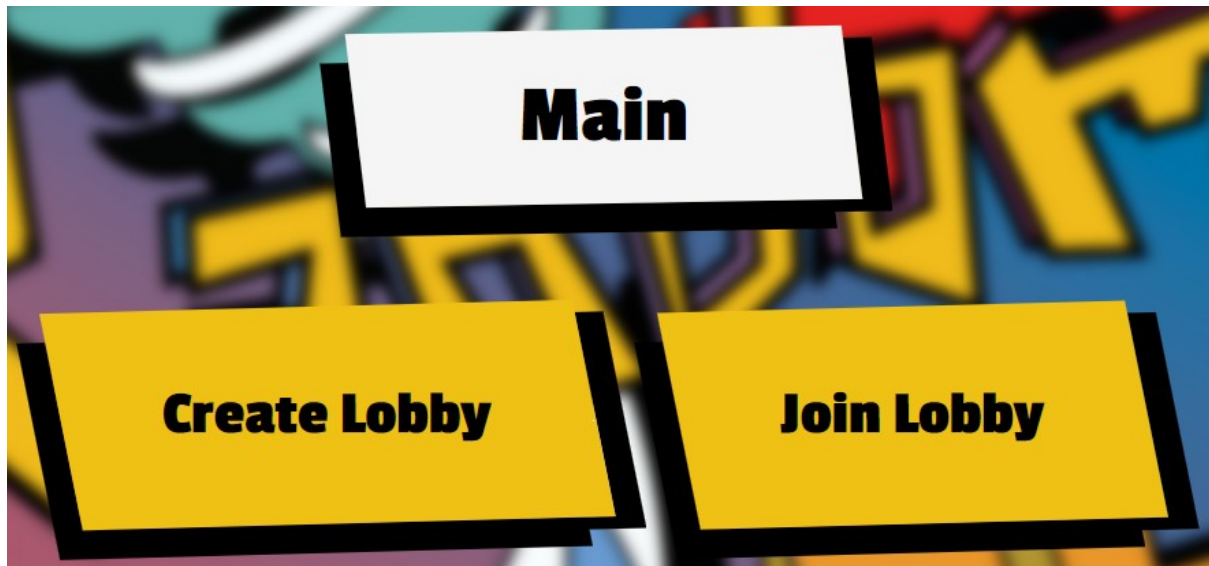
4. Maquettage

Le maquettage de l'interface du site a été réalisé avec l'outil Figma:



Maquette des pages de l'application

Le style des éléments a évolué et été amélioré depuis la conception de la maquette, notamment le style des boutons qui utilisent un design plus dynamique et des couleurs différentes:



Exemple d'évolution du style par rapport à la

5. Pages du site

Les pages prévues sont les suivantes:

- index.html → page d'accueil, où l'utilisateur pourra voir son compte, rejoindre une salle de jeu et gérer ses amis.
- login.html → page de connexion
- register.html → page d'inscription
- ingame.html → page pour la partie gameplay, où le framework Three.js entrera en jeu
- forgot-password.html → page de demande de récupération de compte
- reset-password.html → page de changement de mot de passe lors de la récupération de compte

6. Routes de l'API

Les routes à implémenter dans l'API sont décrites ci-dessous.

Sauf indication contraire, toutes les routes requièrent l'utilisation d'un token pour s'authentifier.

A. Authentification

Deux routes seront définies pour se connecter à l'API et créer un compte respectivement:

- **POST /login**
- **POST /register**

Pour créer un compte, l'utilisateur devra renseigner un nom d'utilisateur, un email et un mot de passe.

Pour se connecter, il devra renseigner son nom d'utilisateur et son mot de passe.

Ces routes seront évidemment publiques et ne nécessitent pas d'être authentifié lors de l'envoi de la requête.

B. Comptes

Le client pourra récupérer les informations et statistiques de son ou d'un autre compte via les routes suivantes:

- **GET /account/profile**
- **GET /account/stats**
- **GET /account/profile/{account_id}**
- **GET /account/stats/{account_id}**

Les profils retournés contiendront à minima l'id et le nom d'utilisateur du compte.

C. Gestion d'amis

Voici les routes à implémenter pour la gestion d'amis et leur fonctions:

- **GET /account/friends** : Récupérer la liste d'amis de l'utilisateur
- **GET /account/requests** : Récupérer les demandes d'amis de l'utilisateur (envoyées et reçues)
- **POST /account/requests** : Envoyer une demande d'ami à un autre utilisateur. L'utilisateur devra renseigner le nom d'utilisateur de l'utilisateur cible dans le corps de la requête.
- **PATCH /account/requests/{username}** : Accepter ou décliner une demande d'ami d'un autre utilisateur
- **DELETE /account/friends/{username}** : Enlever un ami de sa liste d'amis

D. Récupération de compte

Un utilisateur pourra faire une demande de récupération de compte et changer son mot de passe via les routes suivantes:

- **POST /account/request-password-reset** : L'utilisateur devra renseigner l'email associé au compte pour générer un token de récupération. Un lien pour réinitialiser le mot de passe sera ensuite envoyé à l'email associé.
- **POST /account/reset-password** : L'utilisateur devra fournir le token de récupération ainsi qu'un nouveau mot de passe.

Ces routes seront publiques et ne nécessitent pas d'être authentifié lors de l'envoi de la requête.

E. Salles de jeu

Les routes suivantes seront utilisées pour :

- **POST /lobby/create** : Créer une salle de jeu.
- **GET /lobby/current** : Récupérer les informations de la salle de jeu actuelle.
- **GET /lobby/list/{page}** : Lister une page des salles de jeu disponibles.
- **GET /lobby/find/{lobby_id}** : Récupérer les informations d'une salle de jeu spécifique.
- **POST /lobby/join** : Rejoindre une salle de jeu. Son ID devra être spécifié dans le corps de la requête.
- **PATCH /lobby/current/ready** : Change l'état de l'utilisateur dans la salle de jeu (Prêt ou non). La partie est lancée quand tous les joueurs sont prêts et que le nombre minimum de joueurs est atteint.
- **POST /lobby/current/leave** : Quitte la salle de jeu. Action impossible si la partie est en cours.

Initialement, nous avons pensé stocker les salles de jeu dans la base de données. Nous nous sommes rendu compte que ce n'était pas forcément la meilleure solution, car cela pouvait poser problème si nous décidons plus tard d'implémenter des nouveaux modes de jeu, pouvoir personnaliser les règles de la partie, etc.

Cela induirait de devoir modifier la structure des tables pour permettre l'ajout de ces nouvelles features pour permettre la coexistence des différentes règles à paramétrer pour les différents modes, ce qui n'est pas une option idéale.

Nous avons donc décidé de stocker les salles de jeu dans une base de données NoSQL embarquée dans notre API.

De cette façon, nous aurons plus de liberté sur la manière dont nous stockerons nos salles de jeu.

Après quelques recherches, nous avons choisi PoloDB comme système de base de données NoSQL embarqué, car elle s'intègre parfaitement avec la bibliothèque Serde que nous comptons déjà utiliser pour la manipulation de structures JSON.

F. Service de Jeu

Les routes suivantes seront utilisées pour :

- **GET /game/current** : Récupérer les informations de la session de jeu en cours.
- **GET /game/ws** : Se connecter au WebSocket du service de jeu.

G. Server-Sent-Events

En parcourant le dépôt officiel des exemples d'applications réalisées avec Actix, nous avons aperçu un exemple portant sur les Server-Sent Events.

Les Server-Sent Events (SSE) sont une technologie web qui permet à un serveur d'envoyer des données via une connexion HTTP, et ce de manière unidirectionnelle.

Nous avons décidé d'utiliser cette technologie pour mettre à jour dynamiquement la page d'accueil en fonction des actions des autres utilisateurs.

Ce système sera utilisé pour mettre à jour le statut des joueurs de la salle de jeu, pour indiquer qu'une partie a commencé, ou qu'une demande d'ami a été reçue.

Pour l'utiliser, le client pourra se connecter via la route:

- **GET /events**

7. Définition des cartes

A. Variantes de cartes

Les différentes cartes du jeu seraient définies dans un fichier JSON. Ce fichier contiendra une liste d'informations nécessaires pour définir les différentes cartes dans Rust.

L'idée est de définir un trait Card définissant les fonctions communes à toutes les cartes, comme récupérer ses attributs et la procédure à suivre lorsqu'on joue la carte.

En Rust, un trait est comparable aux Interfaces dans le langage Java: il sert à définir un ensemble de fonctions qui pourront être implémentées dans d'autres structures.

De cette manière, nous pouvons implémenter des variantes de cartes avec des logiques différentes pour couvrir un maximum de mécaniques possibles sans changer la procédure pour jouer une carte dans le service de jeu lui-même.

B. Assets

Nous avons récupéré les images des cartes déjà conçues par un membre de l'équipe: nous n'avons donc pas à partir de zéro pour la réalisation des cartes.

8. Le Jeu

La communication entre les clients et le service de jeu se fera à l'aide d'un WebSocket pour permettre l'envoi bidirectionnel de données.

Le service de jeu s'occupera entièrement de la logique des cartes et de la gestion des tours. Le site présentera les actions réalisées au joueurs et pourra envoyer une commande pour jouer une carte dans la main du joueur.

Actions et réponses du serveur

Nous avons défini les structures des données à envoyer et recevoir du serveur pour gérer du mieux possible les différentes mécaniques des cartes, ainsi que le plan de communication pour les différentes actions et réponses (voir schémas complets en annexe):

PlayCard JSON	
action_type	string "PlayCard"
card_index	int
targets	list of player ids (one entry if single target). Only opponents should be present (ex. heal does not require any targets)

**Action du client
pour jouer une carte**

Card Played JSON	
type	string "PlayCard"
player_id	id of the player who played the card
card_id	id
hand_index	index of card in player's hand
actions	list of actions to perform on players

**Réponse du serveur
lorsqu'une carte est
jouée**

9. Conteneurisation

Afin de garantir l'intégrité de l'environnement d'exécution des différents composants du projet, nous avons utilisé Docker pour générer des images de nos composants afin de pouvoir les utiliser dans des conteneurs séparés, capables de communiquer entre eux sur leur réseau privé afin de ne pas les exposer directement sur le réseau.

Nous utilisons l'outil Docker Compose pour orchestrer la génération des images de nos composants, le lancement et la communication des conteneurs.

Les composants sont définis dans un fichier appelé *docker-compose.yml* qui sert à décrire les services (nos composants), la procédure à suivre pour générer leur images, le réseau sur lequel les services communiqueront entre eux, et les variables d'environnement communes à utiliser pour les différents services (ports, mots de passe, clé secrète, etc.).

Nous aurons 3 composants définis dans ce fichier:

- La base de données
- L'API
- Le site

La base de données et l'API pourront communiquer entre eux sur un réseau privé, afin d'éviter de l'exposer sur le réseau.

L'API et le site seront exposés sur le réseau.

Les variables d'environnement seront définies dans un fichier nommé `.env` à la racine du code source du projet.

10. Déploiement

Notre attention pour déployer notre application s'est tournée vers DigitalOcean.

Effectivement, par rapport à d'autres moyens de déploiement, DigitalOcean nous a prouvé que c'était une plateforme de déploiement stable et sécurisée, notamment grâce à l'utilisation de secrets directement configurables dans l'interface de l'application.

De plus, la prise en main, qui peut dans un premier temps sembler difficile, est très bien amenée grâce à une documentation claire et précise des services.

Pour finir, DigitalOcean nous offre un moyen efficace d'envoyer nos images Docker sur une registry Docker privé, nous permettant de déployer facilement nos conteneurs.

IV. Organisation du projet

1. Repository et Versionning

Le code source du projet sera mis en ligne sur GitHub en utilisant l'outil de versionning Git afin de pouvoir mettre en commun nos travaux sur les différentes features de notre projet.

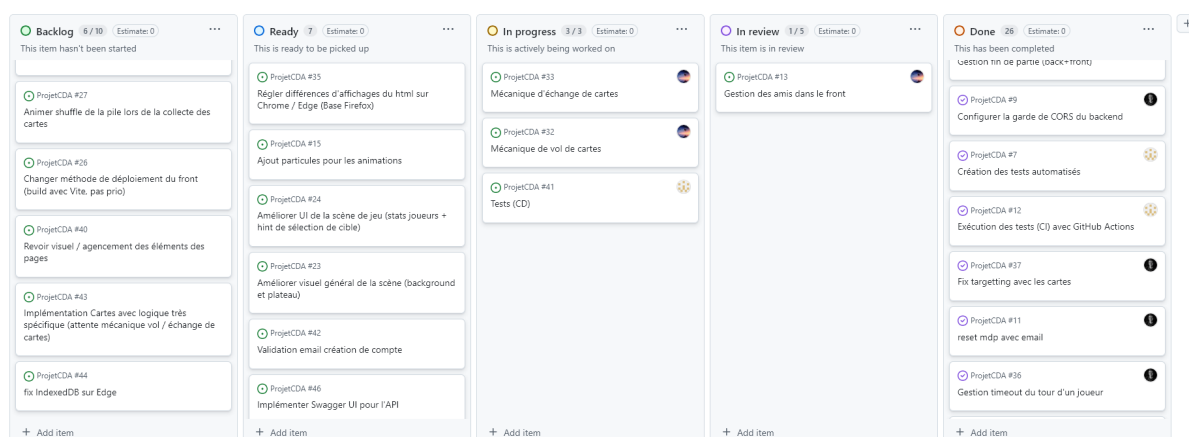
2. Organisation des tâches

Nous avons utilisé GitHub Projects pour l'organisation et l'assignation des tâches à réaliser. GitHub Projects propose un système de gestion de tickets à la Kanban où l'on peut définir des tâches à réaliser, assigner des personnes aux tâches, les classer en fonction de l'avancement des tâches, et leur donner une deadline.

Cette approche s'inscrit pleinement dans la méthode Agile, en favorisant la transparence, l'adaptabilité et la collaboration. Elle permet de structurer le travail de manière itérative et incrémentale, tout en rendant visibles les priorités et les blocages potentiels.

Nous avons classé les tâches dans 5 catégories:

- les tâches envisagées mais pas prêtes à être développées (dépendent de l'avancement des autres tâches)
- les tâches prêtes à être développées
- les tâches en cours de développement
- les tâches terminées en attente de validation
- les tâches complétées



**Gestion de tickets
depuis GitHub
Projects**

3. Outils de développement

Utilisation de Visual Studio Code comme IDE. C'était un choix évident dans la mesure où Visual Studio Code permet l'édition d'une large collection de types de fichier grâce aux extensions qu'il est possible d'y ajouter, et que notre projet utilise plusieurs langages de programmation.

Nous utiliserons Postman pour tester nos différentes routes, en attendant l'implémentation de Swagger UI pour la documentation de l'API.

Pour visualiser le contenu de notre base de données, nous utiliserons l'outil DBeaver. Cet outil nous permettra de visualiser la structure et le contenu des différentes tables de notre base de données via une interface graphique.

4. Canaux d'échanges et partage de ressources

Les documents et autres éléments annexes au projet seront stockés dans Google Drive dans un dossier partagé avec les membres de l'équipe.

Ce dossier partagé inclura des documents tels que les images des cartes, les définitions des cartes dans un tableau Excel, les diagrammes et schémas, des liens vers des ressources utiles (documentations), etc.

Nous pourrions également utiliser les plateformes de discussion Discord ou Google Chat pour le partage rapide d'image de fragments de codes, de liens, etc.

5. Directives pour l'écriture du code

Nous nous sommes mis d'accord avant de commencer à produire du code de respecter les règles suivantes:

- De nommer nos fonctions, variables et fichiers en anglais
- D'écrire nos commentaires en anglais
- De respecter au mieux les conventions de nommage des langages que nous allons utiliser
- De créer une branche de développement pour chaque feature
- S'assurer qu'il n'y ait aucun conflits lors de la fusion d'une branche d'une feature avec la branche principale

V. Développement de l'application

1. Définition de l'API

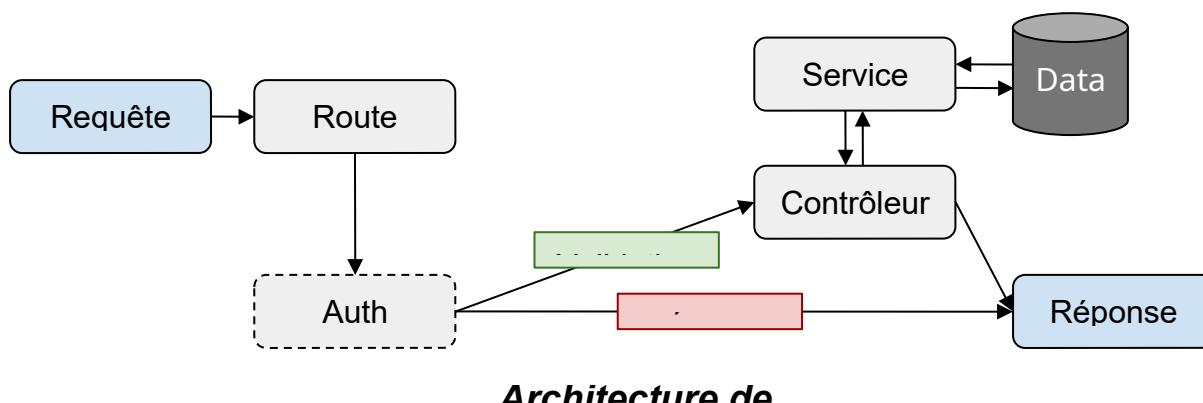
A. Fonctionnement de l'API

Lorsque le client envoie une requête, le routeur analyse l'URL et choisit, en fonction de la méthode et de la route, quel contrôleur appeler.

Avant d'appeler directement le contrôleur, le routeur passe d'abord par un middleware d'authentification qui, en fonction des données de la requête et de la route, décide d'appeler ou non le contrôleur. Une erreur 401 "Unauthorized" est retournée au client lorsque le contrôleur n'est pas appelé.

Si le contrôleur est appelé, il va ensuite faire appel au service qui va s'occuper de communiquer avec la base de données en fonction des données de la requête. En fonction des actions réalisées, le contrôleur retourne une réponse au client au format JSON, avec un code de statut.

Le format de la réponse peut être au format texte si la réponse est une erreur.



En fonction du type d'action à réaliser, l'API peut retourner différents statuts au client pour l'informer de l'état de la requête.

Pour citer les principaux, l'API peut retourner les statuts suivant:

- **200: OK**
Indique que la requête a réussi
- **201: CREATED**
Indique que la requête a réussi et une ressource a été créé
- **400: BAD REQUEST**
Indique que le serveur ne peut pas comprendre la requête, généralement à cause d'éléments manquants dans le corps de la requête ou d'en-têtes non supportés

- **401: UNAUTHORIZED**
Indique que la requête n'a pas été effectuée car le client n'a pas pu être identifié
- **404: NOT FOUND**
Indique que le serveur n'a pas trouvé la ressource demandée
- **500: INTERNAL SERVER ERROR**
Indique que le serveur a rencontré un problème

B. Définition des routes

Macros

Le framework Actix Web nous donne un moyen simple de définir nos routes grâce à l'utilisation de macros.

En Rust, les macros sont des directives qu'il est possible d'ajouter à une fonction ou une structure. Elles généreront du code automatiquement lors de la compilation en fonction de l'objet associé et des paramètres définis.

C'est un moyen élégant d'ajouter des fonctionnalités à un objet:

```
// Génère le code nécessaire pour définir
// la fonction comme contrôleur avec:
// - La méthode (ici GET)
// - La route (/account/profile)
#[get("/account/profile")]
24 implementations
async fn get_my_account(req: HttpRequest, pool: web::Data<DbPool>)
```

Définition d'un contrôleur

Fonctions asynchrones

Les contrôleurs sont définis comme des fonctions asynchrones, avec le mot clé *async*. En déclarant les fonctions des contrôleurs comme étant asynchrones, nous permettons au serveur de gérer de nombreuses connexions simultanées sans bloquer le thread principal, ce qui est crucial pour la performance et la scalabilité de l'application.

Extraction des données

Le framework Actix Web permet de spécifier quels éléments nos contrôleurs ont besoin pour fonctionner en utilisant des *extracteurs*.

Les extracteurs sont définis dans les arguments de la fonction du contrôleur. Ils permettent d'extraire automatiquement des informations de la requête, comme son contenu, les

données du corps de la requête, ou les éléments variables d'une route (ex. /account/profile/{**account_id**}).

```
#[get("/account/profile/{account_id}")]
24 implementations
async fn get_other_account(
    // récupération de la connexion
    // à la base de données
    pool: web::Data<DbPool>,
    // récupération de l'id
    // dans l'url de la requête
    path: web::Path<i32,>
) -> actix_web::Result<impl Responder> {
    let (account_id: i32,) = path.into_inner();
```

***Exemple d'utilisation des extracteurs:
éléments variables de l'URL***

```
#[post("/login")]
24 implementations
async fn login(
    json: web::Json<AccountLogin>,
    pool: web::Data<
        backend::database::actions
    ) -> actix_web::Resu
    let account: Fil
        // Obtaining
        // So, it sh
    pub struct AccountLogin {
        pub username: String,
        pub password: String,
    }
```

***Exemple d'utilisation des
extracteurs: corps de la requête***

Cela nous donne un moyen simple et efficace de récupérer les données de la requête et de les utiliser.

En cas d'échec d'extraction des données, le contrôleur retourne automatiquement une erreur 400: BAD REQUEST.

Données d'application

Dans Actix Web, il est possible de partager des données et de les rendre utilisables par les contrôleurs et services de l'application en définissant des *données d'application*.

Dans l'exemple précédent (voir *Exemple d'utilisation des extracteurs*), le contrôleur récupère la connexion à la base de données via l'extracteur `pool: web::Data<DbPool>`.

`DbPool` est le type d'un objet qui a été ajouté en tant que donnée d'application. L'extracteur `web::Data<T>` permet de récupérer un clone de cet objet et de l'utiliser.

```
App::new()
    .wrap(Logger::default())
    .app_data(web::Data::new(pool.clone()))
    .app_data(web::Data::new(backend_db.clone()))
    .app_data(web::Data::new(server_handlers.clone()))
    .app_data(web::Data::from(Arc::clone(&broadcaster)))
    .app_data(web::Data::new(mailer.clone()))
```

Définition des données d'application

Un objet ne peut être utilisé comme une donnée d'application que s'il implémente le trait `Clone` (qui définit la fonction `clone()`).

Connexion à la base de données

L'API est capable de communiquer avec la base de données en utilisant la bibliothèque Diesel. Elle fournit des structures permettant de se connecter à différents systèmes de base de données.

L'adresse de la base de données est définie dans une variable d'environnement appelée `DATABASE_URL`, et est donc configurable dans le fichier `docker-compose.yml` ou `.env`:

```
let database_url: String = std::env::var("DATABASE_URL").expect("DATABASE_URL env var not set !");
let manager: ConnectionManager<PgConnection> = r2d2::ConnectionManager::<PgConnection>::new(database_url.clone())
let pool: DbPool = r2d2::Pool::builder()
    .build(manager)
    .expect(format!("Unable to connect to database with URL \"{}\" !", database_url).as_str());

println!("Connected to database!");
```

Connexion à la base de données depuis l'API

L'objet permettant de se connecter à la base de données est utilisé comme donnée d'application, comme vu dans l'exemple précédent (figure *Définition des données d'application*).

Configuration

Une fois déclarés, nos contrôleurs doivent ensuite être configurés dans notre application. Pour chaque module où nos contrôleurs sont définis, une fonction de collecte des contrôleurs est exposée et peut être appelée par l'application lors de sa création:

```
pub fn configure_routes(cfg: &mut web::ServiceConfig) {  
    cfg.service(factory: get_my_account) &mut ServiceConfig  
        .service(factory: get_other_account);  
}
```

Enregistrement des contrôleurs du

```
App::new()  
    // [...]  
  
    // auth  
    .configure(routes::auth::configure_routes)  
    // accounts  
    .configure(routes::account::configure_routes)  
    // stats  
    .configure(routes::stats::configure_routes)
```

Appel des fonctions de

C. Sécurité

Hachage des mots de passe

Lors de la création d'un compte, le mot de passe envoyé par le client est hashé avant d'être stocké en base en utilisant la bibliothèque bcrypt, implémentant l'algorithme du même nom. De cette manière les mots de passe des utilisateurs ne sont jamais stockés en clair en base.

Bcrypt est un algorithme de hachage conçu pour stocker les mots de passe de manière sécurisée. Il se repose sur une méthode de chiffrement intitulée Blowfish pour protéger les mots de passe en les transformant de manière irréversible en combinant un sel (suite de caractères) à une clé avant le hachage. Cette méthode implique la définition d'un facteur de coût. Plus le facteur de coût est élevé, plus le hachage prend du temps à être produit. En revanche, cela rend les tentatives d'attaque plus lentes.

Le facteur de coût qu'il est conseillé d'utiliser pour avoir un bon compromis entre temps de génération et sécurité est 12. Nous avons donc utilisé cette valeur comme facteur de coût dans notre application.

Middleware d'authentification

Comme évoqué précédemment, l'API utilise un middleware d'authentification pour permettre l'accès aux routes, seulement si le client est authentifié à l'aide d'un token associé à un compte de la base de données, ou que la route est considérée comme publique.

Le middleware possède 2 mécanismes pour extraire ce token de la requête:

- Il regarde dans l'en-tête "Authorization" si la donnée est de la forme "Bearer <token>"
- Il regarde dans les données des cookies de la requête si une clé "token" est présente

Le premier mécanisme est surtout présent dans un souci de praticité pour tester les requêtes. L'utilisation des données des cookies est la méthode privilégiée lors des requêtes depuis le site.

Lorsque l'authentification réussie, l'ID du compte utilisé pour s'authentifier est placé dans les données de la requête et pourra ainsi être utilisé par le contrôleur:

```
#[get("/account/profile")]
24 implementations
async fn get_my_account(req: HttpRequest, pool: web::Data<DbPool>) -> a
    // get account id based on JWT (put in extensions by JwtMiddleware)
    let account_id: i32 = req.extensions().get::<i32>()
        .unwrap()
        .clone();
    Récupération de l'ID du
    let account: FilteredAccount = web::block(move || {
```

JWT

Dans le but d'effectuer une authentification sécurisée sur mon projet ainsi que stateless, j'utilise les JSON Web Tokens.

Un JWT est un jeton qui permet l'échange des informations sur l'utilisateur de manière sécurisée. C'est une méthode de communication entre deux parties.

Ce jeton est composé de trois parties:

- Un header qui identifie quel algorithme a été utilisé pour générer la signature
- Un payload au format JSON qui contient les informations de l'utilisateur, sous la forme d'une chaîne de caractères hashé en base 64. Par mesure de sécurité, le payload ne contient que l'ID du compte à utiliser pour s'authentifier et n'inclut aucune donnée sensible.
- Une signature permettant au serveur de vérifier l'authenticité du token.

Un token est généré et retourné au client en cas d'authentification réussie avec la route /login, avec une date d'expiration:

```
#[derive(Debug, Deserialize, Serialize)]
3 implementations
pub struct Claims {
    sub: String,
    exp: usize,
    user_id: i32,
}

pub fn create_jwt(user_id: i32) -> String {
    let claims: Claims = Claims {
        sub: user_id.to_string(),
        exp: (Utc::now() + Duration::days(1)).timestamp() as usize,
        user_id,
    };

    let secret: String = std::env::var(key: "BACKEND_SECRET_KEY").unwrap();
    let header: Header = jsonwebtoken::Header::new(Algorithm::HS256);

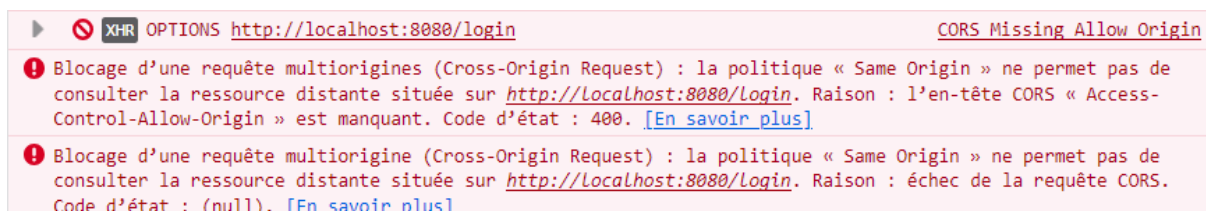
    encode(&header, &claims, key: &EncodingKey::from_secret(secret.as_ref())).unwrap()
}
```

Génération du

Garde CORS

Durant le début du développement, il nous était impossible d'envoyer des requêtes à notre API depuis notre site. Le plus étrange était qu'il était toujours possible d'en envoyer avec Postman.

En examinant la console, nous avons remarqué que le navigateur la requête avait été bloquée à cause de l'exécution d'une "requête multiorigine":



Erreur blocage d'une requête

Après quelques recherches, nous avons identifié la source du problème, elle venait d'un protocole de sécurité mis en place sur les navigateurs appelé le CORS.

Le CORS est un mécanisme qui consiste à ajouter des en-têtes HTTP afin de permettre à un agent utilisateur d'accéder à des ressources d'un serveur situé sur une autre origine que le site courant. (source: <https://developer.mozilla.org/fr/docs/Web/HTTP/Guides/CORS>) Cela permet entre autres de bloquer l'accès à des ressources si l'émetteur de la demande n'est pas reconnu.

Nous avons donc configuré une garde CORS avec Actix, nous permettant entre autres l'envoi de requêtes depuis notre site:

```

HttpServer::new(move || {
    let cors: Cors = Cors::default()
        .allowed_origin(website_url.as_str())
        .allow_any_header()
        .allowed_methods(vec!["GET", "POST", "PATCH", "DELETE", "OPTIONS"])
        .supports_credentials()
        .max_age(3600);

```

Configuration de la garde

D. Documentation de L'API

La documentation de l'API est générée automatiquement à la compilation grâce à la bibliothèque Rust Utoipa. Elle est basée sur Swagger UI.

Swagger UI est un outil maintenu par OpenAPI Initiative permettant d'exposer une documentation d'une API REST à l'aide d'une page web, en listant et groupant les différentes routes définies, les formats attendus du corps des requêtes, mais aussi de permettre l'envoi de requêtes directement depuis l'interface de la page.

Utoipa propose la génération automatique de la documentation en utilisant des macros. Nous n'avons qu'à en ajouter sur nos contrôleurs pour définir leur documentation:

```


#[utoipa::path(
    post,
    path = "/login",
    request_body = AccountLogin,
    responses(
        (status = 200, description = "Login successful, returns JWT token as string", body = String),
        (status = 401, description = "Unauthorized or suspended account"),
        (status = 404, description = "Account not found")
    ),
    tag = "Auth"
)]
#[post("/login")]
24 implementations
async fn login(pool: web::Data<DbPool>, json: web::Json<AccountLogin>) -> actix_web::Result<impl Responder> {
    let account: FilteredAccount = web::block(move || {

```


Macros pour la génération de la documentation de l'API


Exemple d'une route de l'API documentée

randomi-go-api 0.1.0 OAS 3.1
/api-docs/openapi.json
The Randomi GO API

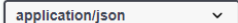
Authorize 

Auth Authentication endpoints

POST /login 

Parameters 

No parameters

Request body *required* 

Example Value | Schema

```
{  "password": "string",  "username": "string"}
```

Responses

Code	Description	Links
200	Login successful, returns JWT token as string	No links

Ce système était très simple à implémenter et bien documenté et nous a permis de nous affranchir de l'utilisation de Postman lors de son ajout tout en ayant une documentation accessible à tout moment.

2. La base de données

A. Création automatique des tables

Lors de la création du service de la base de données, il faut que les tables soient automatiquement créées avant que le service soit utilisé par l'API.

Pour cela, nous avons écrit un script bash pour pouvoir exécuter les scripts de migrations directement lors de la création de l'image:


```
#!/usr/bin/env bash
set -e

echo "Running diesel migration scripts..."

find /docker-entrypoint-initdb.d/migrations/**/up.sql -exec \
    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" -f {} \;
```

Script bash d'exécution des

Ce script va chercher tout les fichiers correspondant au schéma “*/docker-entrypoint-initdb.d/migrations/**/up.sql*” avec *find* et exécuter la commande définie à la dernière ligne du script. Cette commande sert à exécuter un script sur la base de données *POSTGRES_DB* en tant que *POSTGRES_USER*. (définies par *docker-compose.yml*)

Cela à impliqué de créer un Dockerfile spécialement pour la base de données pour pouvoir copier nos scripts de migration et exécuter le script pour les appliquer à la création de l'image de la base de données:

```
1 FROM postgres:17
2
3 # copy diesel migrations to /docker-entrypoint-initdb.d and the init script
4 # to run it when creating the db
5 COPY ./migrations /docker-entrypoint-initdb.d/migrations
6 COPY ./scripts/run-diesel-migrations.sh /docker-entrypoint-initdb.d/run-diesel-migrations.sh
7
8 CMD [ "postgres" ]
```

Dockerfile de la base de données

Le dossier */docker-entrypoint-initdb.d* est documenté dans la description de l'image Docker postgres comme étant un dossier pouvant contenir des scripts à exécuter lors de la création initiale de la base de données.

Le script sera donc exécuté seulement lorsque le service postgres devra créer la base de données.

B. Modèles des tables en Rust

Les modèles des tables de la base sont définis à partir des schémas générés automatiquement par Diesel (voir annexe):

```
#[derive(Queryable, Selectable, Insertable, Serialize)]
#[diesel(table_name = super::schema::accounts)]
#[diesel(check_for_backend(diesel::pg::Pg))]
6 implementations
pub struct Account {
    pub id: i32,
    pub username: String,
    pub email: String,
    pub password: String,
    pub premium: bool,
    pub suspended: bool,
}
```

Modèle de la table Accounts en Rust (complet)

Grâce aux macros, nous pouvons définir cette structure comme étant basée sur un schéma, et qu'il est possible de l'utiliser dans les requêtes (traits Queryable, Selectable et Insertable). Le trait Serialize permet de convertir le modèle en chaîne de caractères au format JSON afin de pouvoir l'envoyer dans la réponse de la requête.

De cette manière, nous pouvons également définir seulement les champs à récupérer et retourner au client en définissant un modèle en omettant certains attributs:

```
// Account struct with sensible fields hidden from the user
#[derive(Queryable, Selectable, Insertable, Serialize)]
#[diesel(table_name = super::schema::accounts)]
#[diesel(check_for_backend(diesel::pg::Pg))]
7 implementations
pub struct FilteredAccount {
    pub id: i32,
    pub username: String,
    pub suspended: bool,
}
```

Modèle de la table Accounts (filtré)

Cela permet de filtrer automatiquement les informations sensibles lors des requêtes.

C. Requêtes avec Diesel

Diesel permet d'effectuer des requêtes directement depuis le code grâce aux schémas de la base de données générés automatiquement et des modèles que nous avons définis.

Par exemple, le code suivant:

```
pub fn get_account_by_id(conn: &mut PgConnection, account_id: i32) -> diesel::QueryResult<FilteredAccount> {
    use super::schema::accounts::dsl::*;

    let account: FilteredAccount = accounts.select(selection: FilteredAccount::as_select()) SelectStatement<...>
        .filter(id.eq(&account_id)) SelectStatement<FromClause<...>, ..., ..., ...>
        .first::<FilteredAccount>(&conn)?;

    Ok(account)
}
```

est équivalent à la requête SQL: *SELECT id, username, suspended FROM accounts WHERE id = <account_id> LIMIT 1*

Diesel supporte également les transactions, assurant l'intégrité des données en cas d'erreurs:

```
let new_account: NewAccount = NewAccount {
    username: username.to_string(),
    email: email.to_string(),
    password: hashed_password,
};

conn.transaction(|conn: &mut PgConnection| {
    insert_into(target: accounts) IncompleteInsertStatement<...>
        .values(records: &new_account) InsertStatement<table, ValuesClause...>
        .execute(&conn)?;

    // get newly inserted account
    let account: FilteredAccount = accounts.select(selection: FilteredAccount::as_select())
        .order_by(expr: id.desc()) SelectStatement<FromClause<...>, ..., ..., .....>
        .first(&conn)?;

    // create stats entry
    insert_into(target: account_stats) IncompleteInsertStatement<...>
        .values(records: NewEmptyStats {account_id: account.id} ) InsertStatement<table, Val...>
        .execute(&conn)?;

    Ok(account)
})
```

Exemple d'utilisation des transactions avec Diesel

3. Composants du site

A. L'App State

Durant le développement, nous avons remarqué qu'il serait plus judicieux de garder certaines informations relatives à l'état du compte (id et nom, etc.) dans un endroit auquel tous les modules pourraient accéder.

Nous avons donc créé un objet pour stocker les informations tels que l'id et le nom de l'utilisateur, ainsi que la salle de jeu actuelle. Cet objet est visible et utilisable par tous les modules.

```
1  import { AccountDTO, LobbyDTO } from "../api/dto";
2
3  /** Store the account (and more) to be used by other modules */
4  export const APP_STATE = {
5      /** @type {AccountDTO | null} */
6      account: null,
7      /** @type {LobbyDTO | null} */
8      lobby: null
9  };
10
```

Objet d'information sur l'état du compte

B. Garde de connexion

Sur les pages nécessitant que l'utilisateur soit authentifié, c'est-à-dire la page d'accueil et du jeu, le module de la page fait appel à la fonction utilitaire appelée *login_guard()*.

```

/** @type {AccountDTO | null} */
export async function login_guard() {
  try {
    const account = await get_my_account();
    if (account == null)
      throw new Error("You are not logged in !")

    APP_STATE.account = account;
    return account;
  } catch (error) {
    console.log("Error: " + error.message);
    window.location.href = "/login.html";
    APP_STATE.account = null;
    return null;
  }
}

```

Fonction de garde de connexion des

Cette fonction essaye de récupérer les informations du compte actuel depuis l'API en utilisant le token stocké dans les cookies de la page.

Si elle échoue, l'utilisateur est redirigé automatiquement vers la page de connexion. Si elle réussit, elle met à jour les informations du compte utilisateur dans l'état de l'application.

C. Système de cache des entités

Afin de limiter l'envoi de requêtes vers l'API, nous avons utilisé l'API IndexedDB disponible dans le navigateur.

L'API IndexedDB permet de stocker des données dans le navigateur de l'utilisateur. Son fonctionnement est très similaire à celui d'une base de données NoSQL: elle permet de stocker des documents JSON et d'interagir avec ces derniers.

Nous avons utilisé ce système pour stocker les données retournées lors de l'envoi des requêtes pour les informations des comptes (noms, statistiques).

Avant de faire une requête vers l'API, on cherche d'abord si une entrée est disponible pour l'élément que nous souhaitons récupérer. Si tel est le cas, on retourne cette entrée au lieu d'envoyer une requête.

Les entrées que nous insérons dans cette base de données sont accompagnées d'un timestamp, nous permettant de définir quand les entrées doivent expirer et qu'il faut en demander de nouvelles à l'API.

```
const store = DB
  .transaction(ACCOUNT_NAMES_STORE, "readwrite")
  .objectStore(ACCOUNT_NAMES_STORE);

const record = { account: accountDTO, date: Date.now() };

const request = store.put(record);
```

***Mise en cache des données d'un
compte sur le navigateur***

D. Composants réutilisables

Comme nous n'avons pas utilisé de frameworks pour le site, nous avons dû nous tourner vers l'utilisation des *custom elements* pour créer des composants réutilisables.

L'idée est de créer une classe qui hérite des propriétés de la classe *HTMLElement* pour définir des éléments autonomes utilisables dans le HTML. Cette fonctionnalité est supportée par la majorité des navigateurs, ormi Safari.

Cela nous permet par exemple d'effectuer des requêtes vers l'API pour définir dynamiquement le contenu de l'élément.

```

export class ProfileStats extends HTMLElement {
  /** @type {AccountStatsDTO | null} */
  accountStatsDTO;

  constructor(accountStatsDTO) {
    super();

    this.innerHTML = WAITING_FOR_STATS_HTML;

    this.update(accountStatsDTO);
  }

  update(accountStatsDTO) {
    if (accountStatsDTO) {
      this.accountStatsDTO = accountStatsDTO;

      this.innerHTML = `
        <h3>Stats</h3>
        <p>Level: ${this.accountStatsDTO.level+1}</p>
        <p>Games played: ${this.accountStatsDTO.games_played}</p>
        <p>Games won: ${this.accountStatsDTO.games_won}</p>
      `;
    }
  }
}

```

Classe de l'élément affichant les statistiques d'un compte

Les composants des pages que nous avons définis comme *custom element* sont:

- Le panel du profil de l'utilisateur : cela nous permet de définir le contenu de l'élément en fonction des informations de notre compte
- Le panel d'amis : cela nous permet de mettre à jour et lier des actions dynamiquement sur les éléments composant le panel
- La liste des salles de jeu : cela nous permet de désactiver les interactions des boutons de la liste durant l'attente de la réponse de l'API quand on rejoint une salle de jeu
- L'élément affichant les informations de la salle de jeu actuelle : permet de définir les méthodes pour mettre à jour l'élément directement dans la classe de l'élément (affichage des utilisateurs prêts, mise à jour de la liste d'utilisateurs)

```

customElements.define("profile-stats", ProfileStats);
customElements.define("profile-settings", ProfileSettings);
customElements.define("profile-info", ProfileInfo);
customElements.define("profile-panel", ProfilePanel);
customElements.define("other-profile-panel", OtherProfilePanel);

<!-- Friend panel -->
<div id="friend-panel-backdrop" class="panel-backdrop"></div>
<friend-panel id="friend-panel"></friend-panel>

<!-- Profile panel -->
<div id="profile-panel-backdrop" class="panel-backdrop"></div>
<profile-panel id="profile-panel"></profile-panel>

<!-- Other Profile panel -->
<div id="other-profile-panel-backdrop" class="panel-backdrop"></div>
<other-profile-panel id="other-profile-panel"></other-profile-panel>

```

Utilisation d'éléments customs dans le HTML

E. Les vues et panels

Les vues

Afin de pouvoir afficher séparément les différentes parties de la page d'accueil et éviter de surcharger l'interface de l'utilisateur, nous avons découpé les différentes parties en "vues". Chaque vue aura une fonction propre et une seule sera affichée à la fois.

La page d'accueil dispose de quatre vues:

- Une vue principale servant de point de chute et permettant de naviguer vers les autres vues.
- Une vue pour lister les salles de jeu joignables.
- Une vue pour créer une salle de jeu.
- Une vue pour afficher la salle de jeu actuelle. Cette vue n'est accessible que si l'utilisateur est dans une salle de jeu.

Pour faciliter le changement de vue, nous avons créé une classe appelée *ViewMgr*, permettant d'animer la disparition et l'apparition des vues:


```
// Simple class to show / hide HTML elements (considered as "views") of the page.
export class ViewMgr {
  /** @type {Array<HTMLElement>} */
  primaryViews = [];
  currentPrimaryView = -1;

  constructor(primaryViews=collectViews()) { ...
  }

  setPrimaryView(view) {
    if (typeof(view) == "number") {
      if (view === this.currentPrimaryView)
        return;
      this.#changePrimaryView(view);
    } else if (typeof(view) == "string") {
      const viewElement = document.getElementById(view);

      const index = this.primaryViews.indexOf(viewElement);

      if (index !== -1)
        this.#changePrimaryView(index);
    } else {
      if (view === this.primaryViews[this.currentPrimaryView])
        return;

      const index = this.primaryViews.indexOf(view);

      if (index !== -1)
        this.#changePrimaryView(index);
    }
  }
}
```

Définition de la classe ViewMgr

Cette classe collecte automatiquement les vues du document HTML. Les vues sont considérées comme les éléments ayant comme nom de classe "view".

```
/**
 * Function to collect views of the document and put them in an array.
 * Any element with the class "view" is collected.
 */
export function collectViews() {
  const documentViews = document.getElementsByClassName("view");
  const views = new Array(documentViews.length);

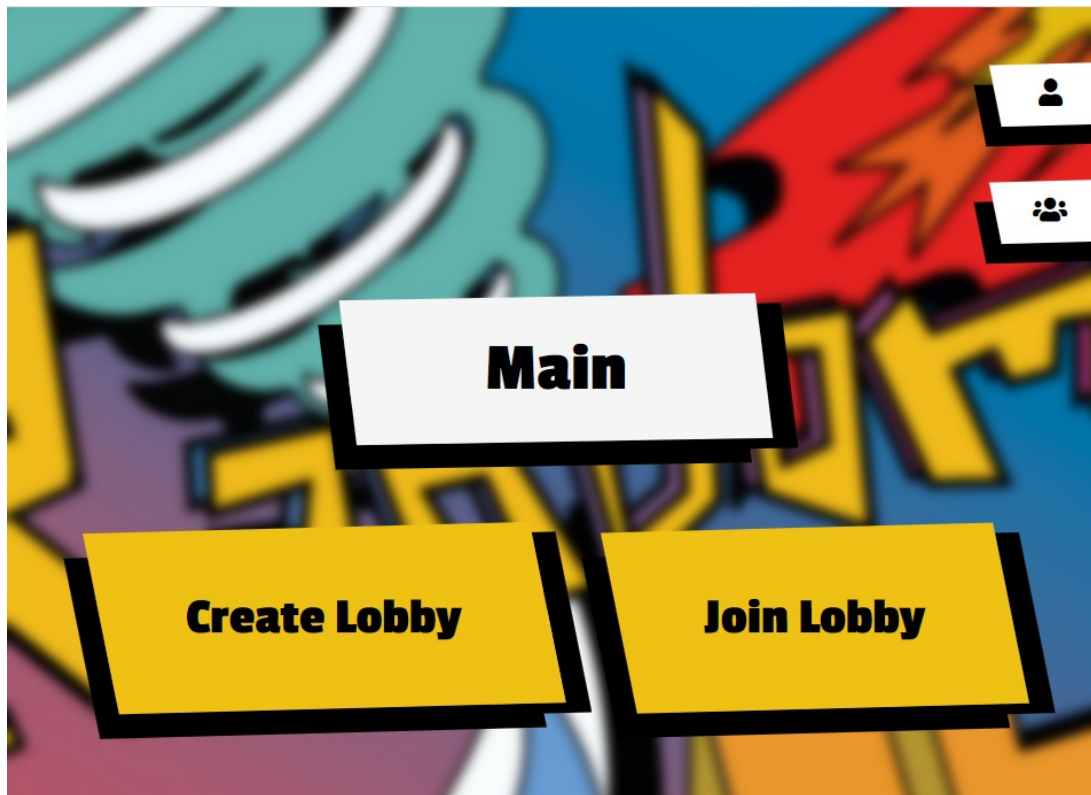
  for (let i = 0; i < documentViews.length; i++)
    views[i] = documentViews[i];

  return views;
}
```

Fonction de collecte des vues

Les panels

Certaines parties de l'application ne nécessitent pas d'utiliser la totalité de l'espace de l'interface. Ces parties sont affichées dans des panels affichables à tout moment à l'aide des boutons d'actions disponible sur le côté de l'interface.



Vue du menu principal

Trois éléments sont affichés dans un panel:

- La vue permettant la gestion des amis
- La vue permettant d'afficher les informations et les paramètres de son compte
- La vue permettant d'afficher les informations d'un compte

4. Salles de jeu

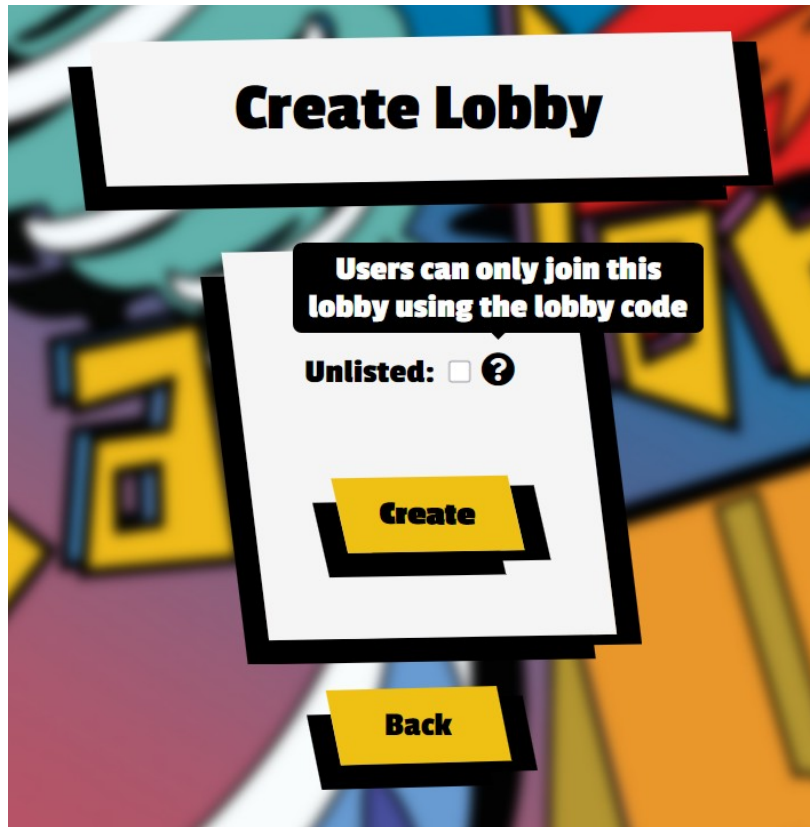
Les salles de jeu permettent aux utilisateurs de se rejoindre pour lancer une partie. Pour lancer une partie, chaque utilisateur doit indiquer s'il est prêt à lancer la partie en envoyant une requête pour indiquer son état.

Une partie commence lorsque tous les joueurs sont prêts. Un service de jeu est alors créé pour cette salle et les utilisateurs peuvent disputer la partie depuis la page *ingame.html*.

A. Créer une salle de jeu

L'utilisateur peut créer une salle de jeu à partir du menu principal.

Pour le moment, il est seulement possible de définir si la salle sera visible dans la liste de salles disponibles.



Vue de création d'une salle de jeu

Nous pourrions envisager d'ajouter de nouveaux modes de jeu ou de pouvoir personnaliser les règles d'une partie à l'avenir.

Lorsqu'une salle est créée, un objet est ajouté à la base de données embarquée dans l'API:

```

let lobbies: Collection<Lobby> = backend_db.lobbies_collection();

if get_lobby_id_for_user(account_id, &lobbies).is_some() {
    return Err(ErrorConflict("User is already in a lobby !"));
}

let lobby_id: String = generate_lobby_id(&lobbies)
    .map_err(error::InternalServerError)?;

let mut lobby: Lobby = Lobby::new(lobby_id.clone(), json.unlisted);
lobby.users.insert(account_id);

lobbies.insert_one(&lobby);

```

Insertion d'une salle de jeu dans la base de données embarqué

B. Rejoindre une salle

Initialement, nous avons pensé à pouvoir créer des salles de jeu avec ou sans mot de passe. Cela aurait permis de créer des parties privées joignables seulement par les utilisateurs disposants du mot de passe.

Cependant, cette méthode présentait un défaut majeur: même en connaissant le mot de passe, il fallait chercher la salle dans la liste des salles joignables. De plus, la principale utilité de ce système est de pouvoir jouer entre amis, or si l'on peut rejoindre les salles de ses amis, ce système perd tout son sens...

C'est un cas que nous n'avions pas anticipé lors de la conception du projet.

Nous avons finalement décidé que pour pouvoir rejoindre une salle de jeu, il fallait simplement connaître son ID. Et qu'il serait possible de choisir de ne pas afficher sa salle de jeu dans la liste grâce à un paramètre.

L'ID serait également affiché dans la vue de la salle actuelle.

La taille de l'ID d'une salle est passée de 32 caractères à 7 caractères, le rendant plus facile à partager et à retenir.

Un utilisateur peut rejoindre une salle de jeu de trois manières:

- En sélectionnant une salle de jeu depuis la liste des salles disponibles
- En entrant directement son ID
- Depuis le panel des amis

Rejoindre une salle de jeu se fait en envoyant une requête POST à la route /lobby/join:

POST /lobby/join

Parameters

No parameters

Request body required

application/json

```
{
  "lobby_id": "ABCDEFGH"
}
```

Requête vers /lobby/join depuis Swagger UI

Il y a 4 code de retours possibles pour cette route:

- La salle a été rejoint (200)
- La salle n'a pas été trouvé (404)
- L'utilisateur est déjà dans une salle, ou la partie a déjà commencé (409)
- La salle est pleine (400)

C. Liste des pages

Afin de ne pas avoir à récupérer l'intégralité des salles de jeu depuis la base de données embarquée, nous affichons la liste des salles par pages de 20 entrées. Cela se fait aisément en spécifiant lors de la requête à la base le nombre d'éléments à récupérer, et à partir de quel index:

```
pub fn list_lobby_page_list_from_db(backend_db: &web::Data<BackendDb>, page: usize) -> Result<LobbyPageList, ()> {
    let lobbies: Collection<Lobby> = backend_db.lobbies_collection();

    // filter out unlisted lobbies
    let entries: ClientCursor<Lobby> = lobbies.find(filter: doc! { "unlisted": false }) Find<'_, '_>, Lo...
        .skip(page*LOBBY_PAGE_SIZE) Find<'_, '_>, Lobby>
        .limit(LOBBY_PAGE_SIZE) Find<'_, '_>, Lobby>
        .run()?;

    let page_count: usize = (lobbies.len() as f64 / LOBBY_PAGE_SIZE as f64).ceil() as usize;

    Ok(LobbyPageList { entries, page, page_count })
}
```

Récupération de la liste des salles par pages

D. Mises à jour dynamiques

Les Server-Sent-Events ont été utilisés pour mettre à jour dynamiquement la liste et l'état des utilisateurs d'une salle, et également pour notifier le client qu'une partie a commencé et qu'il peut s'y connecter.

Envoi des events

Nous avons un service appelé Broadcaster dans notre API. Ce service gère les connexions établies lors de la connexion au SSE via la route /events et permet l'envoi d'events aux clients.

Chaque connexion est identifiée grâce à l'ID du compte utilisé pour s'authentifier, ainsi nous pouvons cibler les clients nécessitant l'envoi d'un event par leur ID.

Les events sont envoyés au clients au format JSON. Le format est défini en fonction des structures Rust correspondantes:

```
#[derive(Serialize, Debug)]
#[serde(tag = "type")]
3 implementations
pub enum SseMessage {
    FriendRequest { request_id: i32, user: i32, status: i32 },
    FriendshipDeleted { id: i32 },
    LobbyUserListChange { users: HashSet<i32> },
    LobbyUserReadyChange { user: i32, ready: bool },
    GameStarted { game_id: GameId }
}

impl SseMessage {
    pub async fn send(&self, tx: &mpsc::Sender<sse::Event>) -> Result<(), mpsc::error::SendError<sse::Event>> {
        let data: Event = sse::Data::new_json(data: &self).unwrap().into();
        log::info!("Sending SseMessage: {:?}", data);
        tx.send(data).await
    }
}
```

Définition des structures des Events du SSE

L'envoi des events de mise à jour de la liste des utilisateur est géré par une fonction qui envoi l'événement seulement aux utilisateurs de la salle:

```
pub async fn notify_lobby_user_list_update(&self, lobby: &Lobby, skip_id: i32) {
    let clients: Vec<SseClient> = self.inner.lock().clients.clone();

    let msg: SseMessage = SseMessage::LobbyUserListChange { users: lobby.users.clone() };

    let send_futures: impl Iterator<Item = impl Future<Output =... = clients
        .iter() Iter<'_, SseClient>
        .filter(|client: &&SseClient|
            client.account_id != skip_id
            && lobby.users.contains(&client.account_id)) impl Iterator<Item = &SseClient>
        .map(|client: &SseClient| msg.send(&client.tx));

    // try to send to all clients
    let _ = future::join_all(iter: send_futures).await;
}
```

Envoi de la mise à jour de la liste des utilisateurs via les SSE

Écoute des events

Le client écoute les events en se connectant à la route `/events` en utilisant la classe `EventSource`. Cette classe fait partie de l'API standard des navigateurs et est spécialement conçue pour ce cas d'utilisation.

Lors de la réception d'un event, l'instance de cette classe émet un événement contenant les données envoyées par l'API.

Nous comparons le type d'event reçu grâce au champ "type", défini par le nom de la variante de l'event dans Rust, et réalisons les mises à jour sur les éléments de la page:

```

events = new EventSource(api_url("/events"), { withCredentials: true });

// listen to onopen and onerror events [...]

events.onmessage = (ev) => {
    let json_data = ev.data;

    // parsing data to JSON [...]

    switch(json_data["type"]) {
        case "FriendRequest":
            handle_friend_request_update(json_data["request_id"], json_data["user"], json_data["status"]);
            break;
        case "FriendshipDeleted":
            handle_friendship_deleted(json_data["id"]);
            break;
        case "LobbyUserListChange":
            handle_lobby_user_list_change(json_data["users"]);
            break;
        case "LobbyUserReadyChange":
            handle_lobby_user_ready_change(json_data["user"], json_data["ready"]);
            break;
        case "GameStarted":
            window.location.href = "/ingame.html";
            break;
        default:
            console.error(`SSE: Unrecognized message type: ${json_data["type"]}`);
            break;
    }
}

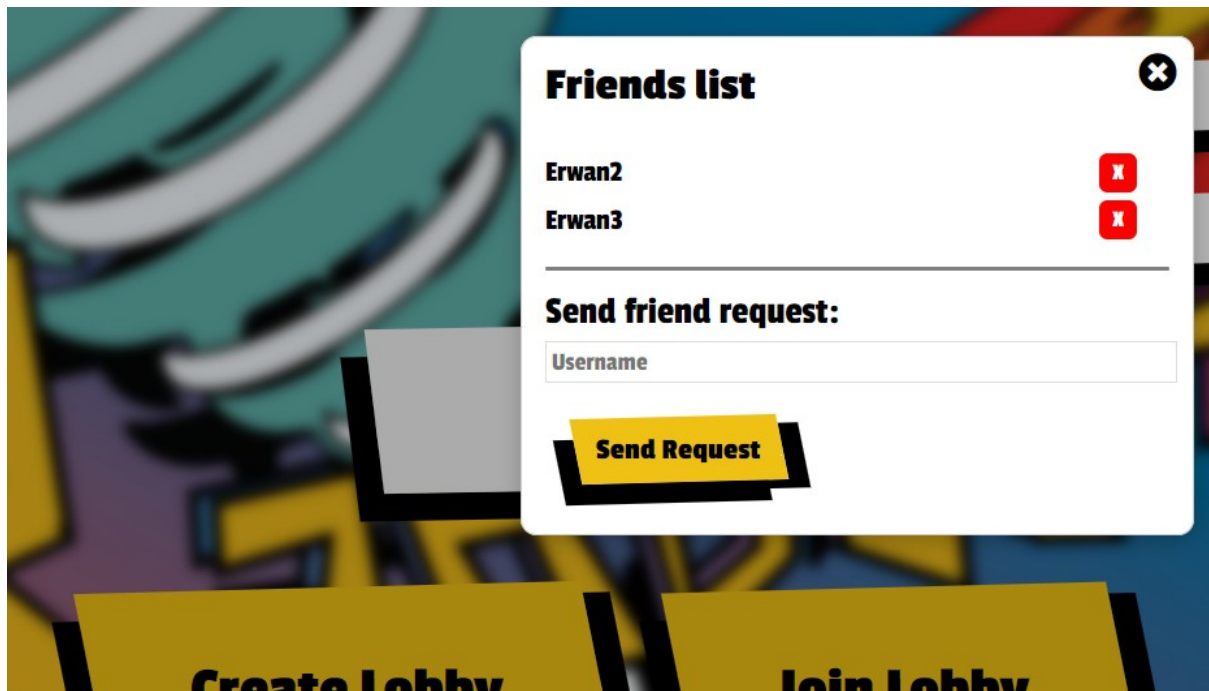
```

Connexion et mises à jour avec les SSE

5. Gestion des amis

L'utilisateur est capable, depuis un panel sur la page d'accueil:

- d'afficher sa liste d'amis
- d'afficher sa liste de demandes d'ami (émises et reçues)
- de supprimer l'un de ses amis
- de répondre à une demande d'ami (accepter ou refuser)
- de rejoindre la salle de jeu de ses amis



Vue du panel de gestion des amis

A. Relation entre comptes

Dans la base de données, être ami avec un compte signifie avoir une “relation” avec un compte. Avoir une relation avec un autre compte n’est pas toujours synonyme d’être ami avec ce compte. Une relation a un statut indiqué par un entier, avec 3 cas possibles:

- 0: désigne une demande d’ami en attente d’acceptation
- 1: désigne une demande d’ami acceptée
- 2: désigne une demande d’ami rejetée

Pour envoyer une demande d’ami à l’utilisateur, l’API regarde d’abord si une relation n’existe pas déjà avant d’en créer une nouvelle.

Cela veut dire que si une demande a déjà été envoyée, acceptée ou refusée, il n’est pas possible d’envoyer de demande à l’utilisateur.

Dans le cas où une demande a déjà été refusée, le receveur de la demande doit supprimer la relation avant que l’émetteur puisse en envoyer une nouvelle.

B. Envoi d’une demande d’ami

Envoyer une demande d’ami se fait en renseignant le nom d’utilisateur de la personne que l’on veut ajouter.

Cela est possible car la table des comptes a une contrainte UNIQUE sur la colonne name, ce qui signifie qu’il n’est pas possible d’insérer une nouvelle ligne dans cette table si la contrainte n’est pas respectée, nous permettant de garantir des noms uniques pour tous les comptes des utilisateurs.

Lors de la création, de la modification du statut, ou de la suppression d'une relation, l'API essaye d'envoyer un event via les SSE à l'utilisateur s'il est connecté et écoute les events de l'API.

Cela permet au panel de se mettre à jour en temps réel en comparant l'ID des requêtes déjà récupérées et celles modifiées.

```
async handleFriendRequestUpdate(request_id, user_id, status) {
  switch (status) {
    // waiting
    case 0: {
      // "fake" req data
      const req = { "id": request_id, "account1": user_id, "account2": APP_STATE.account.id, "status": status };
      const entry = await this.createFriendRequestEntry(req);
      this.friendRequestsDiv.prepend(entry);
    }
    break;
    // accepted
    case 1: {
      const requestElement = document.getElementById(`friend-request-entry-${request_id}`);
      if (requestElement) {
        let username = "Unknown";

```

***Exemple logique de mise à jour
des entrées du panel de gestion***

6. Service de Jeu

A. Définition des cartes

Le trait Card

Comme évoqué dans la partie Conception, les différentes variantes de cartes sont définies dans des structures Rust implémentant un trait commun appelé Card.

Le trait Card définit les fonctions suivantes:

```

pub trait Card: Sync + Send + Debug + CardClone {

    // common play impl
    /// Returns a PlayInfo struct describing the actions made when playing the card + a set of indices of used buffs
    fn play(&self, player_index: usize, target_indices: Vec<usize>, game: &mut Game) -> Result<(PlayInfo, HashSet<usize>), String> {...}
    // basic attack impl
    fn handle_attack(&self, info: &mut PlayInfo, game: &mut Game, player_index: usize, target_indices: &Vec<usize>, dice_roll: u8, dice_roll2: u8) -> Result<(), String> {...}
    // basic heal impl, heal current player
    fn handle_heal(&self, info: &mut PlayInfo, game: &mut Game, player_index: usize, _target_indices: &Vec<usize>, dice_roll: u8, dice_roll2: u8) -> Result<(), String> {...}
    // basic draw impl, draw cards for current player
    fn handle_draw(&self, info: &mut PlayInfo, game: &mut Game, player_index: usize, _target_indices: &Vec<usize>, dice_roll: u8, dice_roll2: u8) -> Result<(), String> {...}
    fn get_id(&self) -> CardId;
    fn get_name(&self) -> String { String::from("???") }
    fn get_attack(&self) -> u32 { 1 }
    fn get_attack_modifier(&self) -> Option<Box<dyn Modifier>> { None }
    fn get_heal(&self) -> u32 { 0 }
    fn get_heal_modifier(&self) -> Option<Box<dyn Modifier>> { None }
    fn get_draw(&self) -> u32 { 0 }
    fn get_draw_modifier(&self) -> Option<Box<dyn Modifier>> { None }
    fn get_description(&self) -> String { String::from("N/A") }
    fn get_kind(&self) -> Kind { Kind::Weapon }
    fn get_element(&self) -> Element { Element::Fire }
    fn get_stars(&self) -> Stars { Stars::One }
    fn get_target_type(&self) -> TargetType { TargetType::Single }
    /// Buffs are granted after the card is played
    fn get_buffs(&self) -> Vec<Box<dyn Buff>> { Vec::with_capacity(0) }
    fn get_damage_effect(&self) -> EffectId { ... }
    fn get_heal_effect(&self) -> EffectId { EffectId::from("heal_regular") }
    // basic validate_targets impl (only check if target count is equal to targets len)
    fn validate_targets(&self, targets: &Vec<&Player>) -> Result<(), String> {...}
} trait Card

```

Fonctions communes à toutes les

Définir un trait nous permet d'avoir un modèle à suivre pour pouvoir chacun de notre côté développer les différentes variantes des cartes.

Par exemple, nous avons défini une variante de carte utilisant la logique de base définie dans le trait, une carte pour réaliser plusieurs actions simples en un même tour (ex. 2 séquences d'attaque), et enfin une carte avec une logique trop spécifique pour pouvoir utiliser les méthodes du trait.

Utilisation des traits en Rust

Les traits permettent de définir des fonctions et des comportements communs pour plusieurs structures, et dans notre cas, permettent de définir une interface d'utilisation pour nos cartes.

Cependant, un trait ne définit pas d'objet concret en soi. Cet aspect nous a posé problème sur la manière dont on peut créer et utiliser les structures implémentant notre trait, notamment lorsque l'on veut utiliser les vecteurs d'éléments.

Le principal problème est qu'un trait n'a pas de taille défini en mémoire. Or pour pouvoir utiliser un vecteur d'éléments, le compilateur doit pouvoir déterminer à l'avance la taille des éléments pour générer l'implémentation correcte du vecteur.

Après réflexion et quelques recherches sur des forums comme StackOverflow et sur la documentation de Rust, nous avons trouvé comment palier à ce problème: il nous suffit de stocker dans le vecteur un objet qui référence notre objet implémentant le trait, qui lui a une taille connu au moment de la compilation.

Nous avons donc utilisé la structure *Box<T>*, qui permet de référencer un objet de type *T* en mémoire. Ici le type *T* serait “*dyn Card*”, désignant n’importe quelle structure implémentant le trait *Card*.

Fonction pour jouer une carte

Pour informer le client des actions réalisées par une carte lorsqu’elle est jouée, nous retournons une structure appelée *PlayInfo* contenant la liste des actions réalisées et sur quels joueurs (attaque, soins, pioche des cartes, lancé de dé, etc.)

Ces actions sont définies dans la structure *PlayAction*:

Structures des actions réalisables par les cartes

```
#[derive(Debug, Clone, Serialize, Deserialize)]
5 implementations
pub struct PlayAction {
    pub dice_roll : u8,
    pub player_dice_id: PlayerId,
    pub targets: Vec<ActionTarget>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
4 implementations
pub struct ActionTarget {
    pub player_id : PlayerId,
    pub action: ActionType,
    pub effect: String
}

#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(tag="type")]
5 implementations
pub enum ActionType {
    Attack{amount: u32},
    Heal{amount: u32},
    Draw{cards: Vec<CardId>},
    Discard{cards: Vec<usize>},
    Steal{cards: Vec<usize>},
}
```

Ces structures sont marquées comme sérialisables et peuvent être converties en objet JSON lors de l’envoi au clients.

Utilisation des cartes dans le Frontend

Le fichier JSON des définitions des cartes est récupéré avant chaque partie en envoyant une requête GET à la route */cards*.

Il sera utilisé pour afficher les noms et descriptions des cartes, ainsi que pour déterminer la manière dont une carte peut être utilisée (cibles).

B. Chargement des cartes

Les cartes sont chargées depuis un fichier JSON à partir de structures servant de modèles pour la désérialisation des objets du JSON:

```
#[derive(Debug, Deserialize, Serialize)]
#[serde(tag = "type")]
3 implementations
enum CardVariant {
    BasicCard(BasicCardData),
    MultiHitCard(MultiHitCardData),
    TargetBothCard(BasicCardData), // same fields as BasicCard
    MultiActionCard(MultiActionCardData),
    PlayersRollsDiceCard(PlayersRollsDiceCardData),
    PearthCard,
}

#[derive(Debug, Deserialize, Serialize)]
3 implementations
struct BasicCardData {
    #[serde(default)]
    attack: u32,
    #[serde(default)]
    heal: u32,
    #[serde(default)]
    draw: u32,
    #[serde(default)]
    attack_modifier: Option<ModifierInfo>,
    #[serde(default)]
    heal_modifier: Option<ModifierInfo>,
    #[serde(default)]
    draw_modifier: Option<ModifierInfo>,
    #[serde(default)]
    targets: TargetType,
}

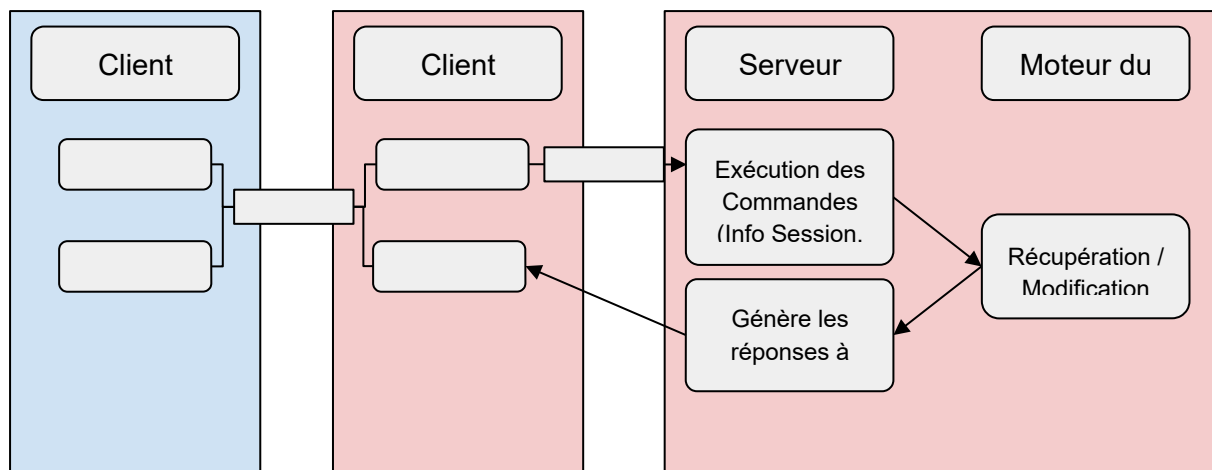
/// Common card data
#[derive(Debug, Deserialize, Serialize)]
4 implementations
pub struct CardInfo {
    #[serde(default)]
    id: CardId,
    name: String,
    element: Element,
    stars: Stars,
    kind: Kind,
    #[serde(default)]
    desc: String,
    #[serde(default)]
    buffs: Vec<BuffVariant>,
    #[serde(flatten)]
    variant: CardVariant
}
```

**Structures de
sérialisation des
cartes définies
dans le JSON**

Chaque structure de carte a un modèle définissant la variante qu'il est possible d'utiliser lors désérialisation du JSON. Ces variantes implémentent une fonction permettant de créer la carte une fois les données récupérées.

C. Plan de communication

Voici le plan de communication entre les clients et le serveur:



Plan de communication entre les clients et le service de jeu

Les requêtes des clients sont d'abord traitées par un handler, qui va demander au serveur de réaliser des actions sur le jeu ou de récupérer l'état de la partie via des commandes (définies en tant que variantes d'un enum Rust). Seul le serveur est capable de modifier directement l'état du jeu.

Passer par un handler permet d'avoir un premier filtre sur les requêtes qui ne sont pas valides ou non reconnues.

Cela nous laisse de la liberté quant à la méthode de communication entre le client et le serveur: nous pouvons par exemple décider plus tard de changer de protocole de communication sans toucher au serveur directement, et ne changer que l'implémentation du handler.

Cela permet également d'externaliser la gestion des sessions connectées: la gestion des sessions est faite exclusivement par le handler.

Lors de la connexion au service, une session est créée pour gérer les échanges. Cette session a un ID assigné correspondant à l'ID du compte de l'utilisateur et permet de l'identifier pour l'envoi des données du jeu.

D. Procédure d'un tour de jeu

Lors du tour d'un joueur, il ne peut réaliser qu'une action: jouer une carte.

Pour jouer une carte, il faut sélectionner la carte désirée et la cible. La sélection de la cible peut varier en fonction du type de la carte et du type de cible:



Ciblage des joueurs dans la scène

Les joueurs disposent d'un temps limité pour jouer une carte (~90 secondes). Passé ce délai, le serveur avance l'état du jeu et saute le tour du joueur.

Si le serveur ne reçoit pas de commande du handler pendant une période prolongée et qu'aucun client n'est connecté au WebSocket, le service est stoppé.

E. Chat textuel de jeu

Les joueurs sont capables d'envoyer des messages aux autres joueurs de la partie.

Cette fonctionnalité est un reste des exemples officiels fournis par Actix sur la communication avec les WebSockets.

Cette fonctionnalité n'était pas prévue à la base mais ne nous coûtait que peu de temps de développement à maintenir: nous l'avons donc gardée.

F. Mises à jour de la scène

La scène est mise à jour en fonction des réponses envoyées par le serveur. Des animations sont jouées sur la scène pour refléter les actions du serveur (carte jouée, défaussée, piochée, etc.).

Les animations sont essentiellement jouées en utilisant la bibliothèque GSAP, en faisant varier les attributs de position et de rotation des objets 3D dans la scène dans le temps grâce à l'interpolation:

```

/** transition smoothly to next position */
goto(x, y, z, duration=0.4) {
  const tl = gsap.timeline();
  tl.to(this.position, { x, y, z, duration });
}

flipCard(instant=false) {
  const tl = gsap.timeline();
  tl.to(this.rotation, { y: (this.flipped ? 0 : -Math.PI), duration: (instant ? 0.0 : 0.15) });
  this.flipped = !this.flipped;
}

```

Exemples d'utilisation de GSAP pour les animations

Afin d'éviter de réaliser plusieurs actions en même temps lors de la réception des réponses du serveur, nous avons implémenté un système de pile d'événements.

L'idée est de créer des classes d'événements définissant les actions à réaliser et les animations à jouer sur la scène.

Ces événements envoient un signal permettant d'exécuter le suivant dans la pile une fois complété.

Nous avons par exemple des events pour la pioche, la défausse d'une carte, l'affichage du résultat du dé, les dégâts et soins, etc.

Nous avons créé un système très simple permettant d'exécuter des objets d'événements. Chaque événement hérite de la classe *GameEvent*:

<pre> export class GameEvent { /** * @type {EventMgr null} * set by EventMgr when pushed to the queue */ mgr = null; timeout = 250; // in ms #started = 0; constructor() {} // reimplement logic here async run() {} // read-only get started() { return this.#started; } /** * called by EventMgr * do not reimplement this function, reimplement run() instead */ execute() { // if timeout < 0, it's up to run() to call notifyMgr() when done if (this.timeout >= 0) { setTimeout(() => { this.onTimeout(); }, this.timeout); this.#started = Date.now(); this.run(); } // tells the EventMgr to execute next event in queue onTimeout() { this.mgr.executeNext(); } } } </pre>	<pre> export class PutCardInPile extends GameEvent { constructor(player, card) { super(); this.timeout = 400; this.player = player; this.card = card; } async run() { if (this.card != null) { this.card.active = false; // transfer card to discard pile this.player.discard_cards.push(this.card); // remove card from hand if it's not fake const idx = this.player.cards.indexOf(this.card); if (idx != -1) { this.player.cards.splice(idx, 1); } this.player.updateHandCardPositions(); this.player.emitCardCountChange(); this.player.updateDiscardCardPositions(); this.player.emitDiscardCountChange(); } } } </pre>
---	---

Définition de la classe

Utilisation de la classe

La pile d'événements est gérée par la classe *EventMgr*. C'est cette classe qui s'occupe d'exécuter les événements et d'appeler les suivants.

Ces événements, mis bout à bout, permettent d'animer la séquence complète d'un tour de jeu.

7. Media Queries

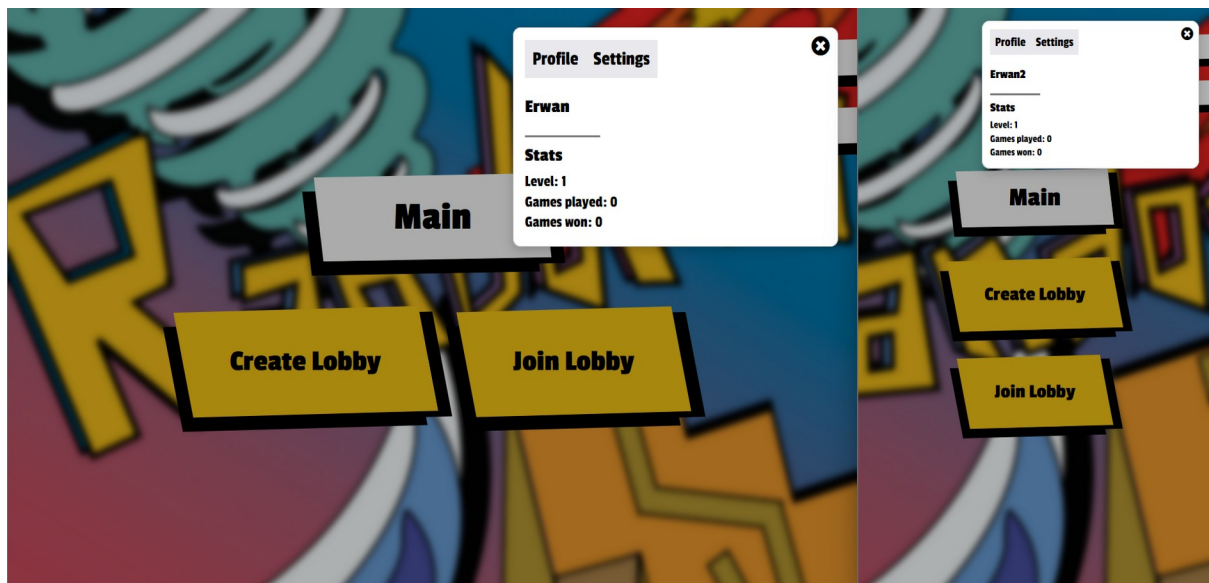
Pour certains éléments du site, il a fallu utiliser les Media Queries pour assurer le bon fonctionnement de l'interface.

Les Media Queries sont des directives utilisables dans le CSS des pages qui permettent d'ajouter des règles en fonction de l'appareil qui est utilisé pour afficher la page (notamment les dimensions de l'écran et la méthode de pointage des éléments).

Par exemple, en fonction des dimensions de l'écran, nous réduisons la taille de la police du corps de la page. Ce changement est répercuté sur tous nos éléments et mettent à jour leur taille:

```
body {  
  font-size: x-large;  
}  
  
/* [...] */  
  
/* MOBILE VIEW */  
@media all and ((max-width: 480px) or (max-height: 480px)) {  
  body {  
    font-size: medium;  
  }  
}  
  
/* [...] */  
}
```

Définition de la taille de la police du corps de la page en fonction des dimensions de l'écran



Comparaison de la taille et de l'agencement des éléments en fonction des dimensions de

Cette utilisation est possible car nos éléments définissent leur taille d'après une valeur définie en *em*, et non en *px*, ce qui permet l'ajustement automatique en fonction de la taille de la police.

Un autre cas d'utilisation courant a été de gérer l'affichage d'information lors du survol d'un élément avec la souris. Cette méthode de fonctionnement n'est tout simplement pas applicable quand il faut utiliser un écran tactile.

Nous avons par exemple fait en sorte de n'afficher le bouton pour rejoindre une salle seulement si le curseur de la souris:

```
.lobby-entry button { visibility: hidden; transition-duration: 0s; /* prevent disappear being animated */ }
.lobby-entry:hover button { visibility: visible; }
```

Affichage d'un bouton seulement si son parent est pointé avec la souris

Afin de préserver cette fonctionnalité, nous avons ajouté une règle qui permet de remplacer l'état de visibilité du bouton dans la règle *.lobby-entry button*:

```
/* TOUCHSCREEN */
@media (pointer: coarse) {
  .lobby-entry button { visibility: visible; } /* always visible */
}
```

Règle pour afficher un élément en fonction de la méthode de pointage de

Le filtre *pointer: coarse* (coarse → grossier en français) permet de sélectionner seulement les appareils avec une méthode de pointage imprécise.

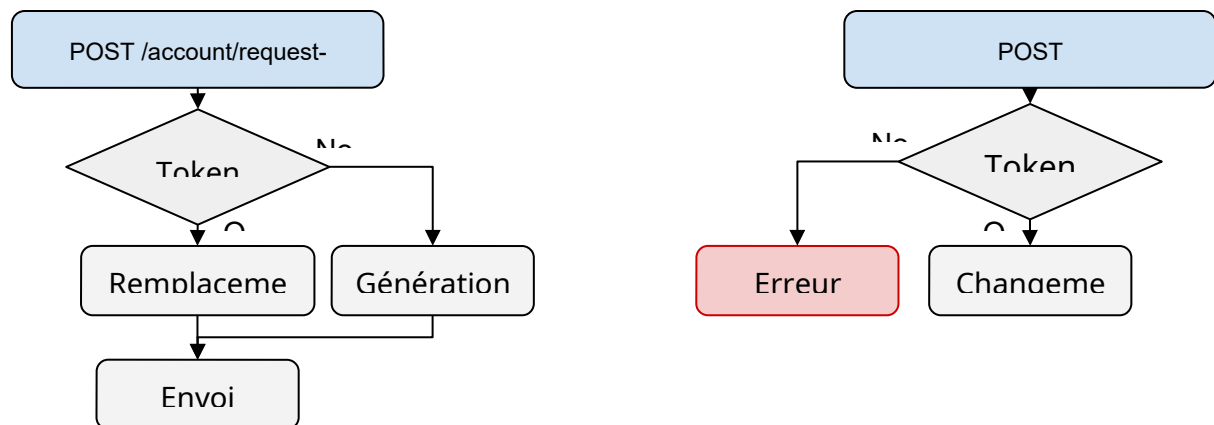
8. Récupération de compte

A. Processus de récupération

La récupération d'un compte se fait en deux étapes: la demande de récupération et la définition d'un nouveau mot de passe.

Afin de réaliser une demande de récupération, l'utilisateur doit renseigner l'email utilisé pour créer son compte. Un message est alors envoyé à l'adresse renseignée si l'email est utilisé par un compte.

Le message contiendra un lien pour redéfinir le mot de passe du compte. Ce lien contiendra un token qui sera utilisé pour valider l'opération. Le token sera invalidé s'il est utilisé un certain temps après avoir été créé.



***Schéma étapes de
récupération de compte***

B. Envoi d'email

Pour envoyer un message par mail, nous avons utilisé la bibliothèque Rust lettre. Elle permet d'envoyer des courriers électroniques en se connectant à un serveur SMTP.

Pour ce projet, nous avons décidé d'utiliser une de nos adresses électroniques et d'utiliser le serveur SMTP de Gmail pour réaliser cette partie.

Nous avons donc créé un module permettant d'envoyer un email à un destinataire, basé sur les structures de la bibliothèque lettre (voir annexe).

Les informations nécessaires à l'authentification auprès du serveur sont définies dans des variables d'environnement.

9. Déploiement du projet

Le projet est automatiquement testé et déployé avec à la définition de workflows, grâce à l'utilisation de GitHub Actions.

Nous avons un workflow pour l'exécution des tests sur GitHub dans un conteneur basé sur ubuntu (intégration continue), et un autre pour le déploiement du projet sur la plateforme DigitalOcean.

Ces workflow sont déclenchés automatiquement lorsqu'on pousse de nouvelles fonctionnalités vers le dépôt du projet (branche de développement principale uniquement). Si un test échoue, les fonctionnalités poussées sont rejetées et le dépôt du projet n'est pas mis à jour.

Workflow de test

D'abord, chaque fonction critique du backend Rust fait l'objet de tests unitaires, lancés via la commande `cargo test`.

Ces tests vérifient par exemple que l'effet « soin de 5 points » restaure correctement la vie d'un joueur, que la pioche distribue toujours un nombre de cartes conforme à la main maximale, ou que la génération d'un code de salon à sept caractères respecte le format attendu.

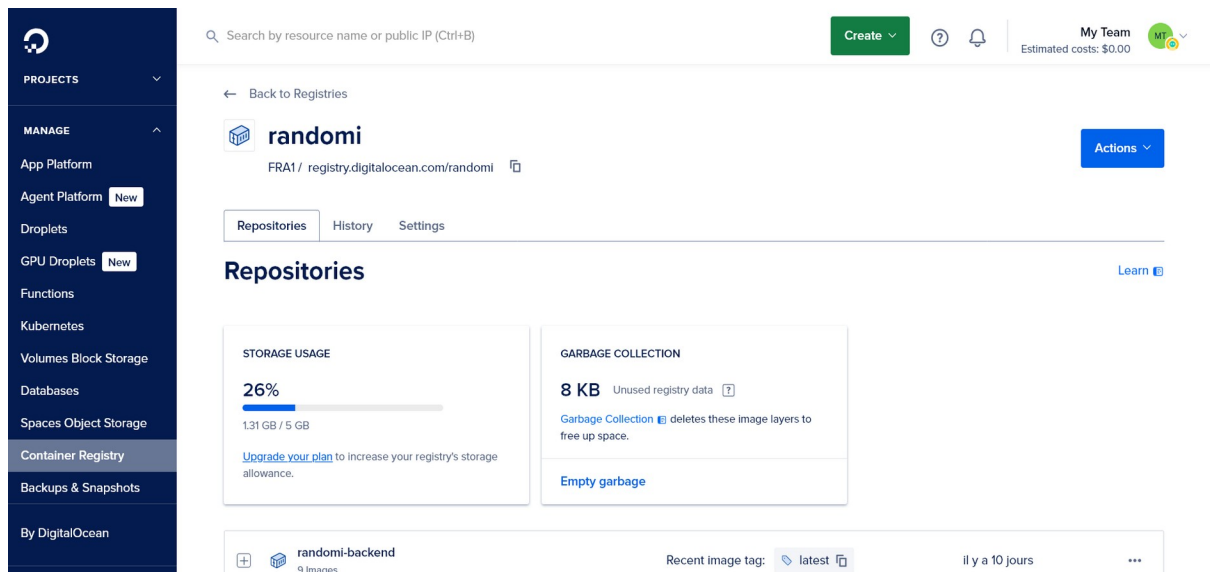
En parallèle, nous avons développé des tests d'intégration qui exercent l'API REST complète : à partir d'un jeu de données initial, un script Postman (exporté en collection JSON) simule la création d'un compte, l'envoi d'une requête de mot de passe oublié, la création d'un salon puis l'invitation d'un second utilisateur.

Ces scénarios valident à la fois la persistance dans PostgreSQL, la génération et l'envoi d'emails, ainsi que la cohérence des réponses HTTP (statuts, corps JSON).

Côté Frontend, nous avons réalisé des tests principalement sur le comportement des fonctions lors de la réception de données depuis l'API, et sur quelques fonctions utilitaires comme la fonction `strjoin()` qui est une implémentation de la méthode `str.join()` de Python en JavaScript faite par nous-mêmes.

Workflow de déploiement

Le workflow de déploiement nous permet de générer et pousser nos images Docker vers la registry privée de notre espace sur DigitalOcean.



Interface du registre des images Docker sur DigitalOcean

Cette action est réalisée en plusieurs étapes:

- L'authentification sur la plateforme DigitalOcean
- La génération des images
- L'envoi des images vers le registre privé sur DigitalOcean
- L'envoi d'une requête vers le service pour déployer nos images sur la plateforme.

Documentation des tests

Pour que ce pipeline soit reproductible et compréhensible par toute l'équipe, nous avons documenté chaque étape dans le dépôt Git avec:

- un fichier *DEPLOYMENT.md* décrivant la configuration requise pour mettre en place le workflow de déploiement
- un fichier *doc/ci-cd.md* détaillant le contenu des workflows des GitHub Actions, les secrets nécessaires (tokens Docker Hub, clés SSH) et les artefacts générés (rapports de test, images Docker)

Cette documentation, jointe au dossier de projet, assure que tout nouveau membre ou intervenant peut comprendre, reproduire et faire évoluer le processus de mise en production sans risque d'erreur manuelle.

Annexes

I. Schémas de conception des éléments du projet

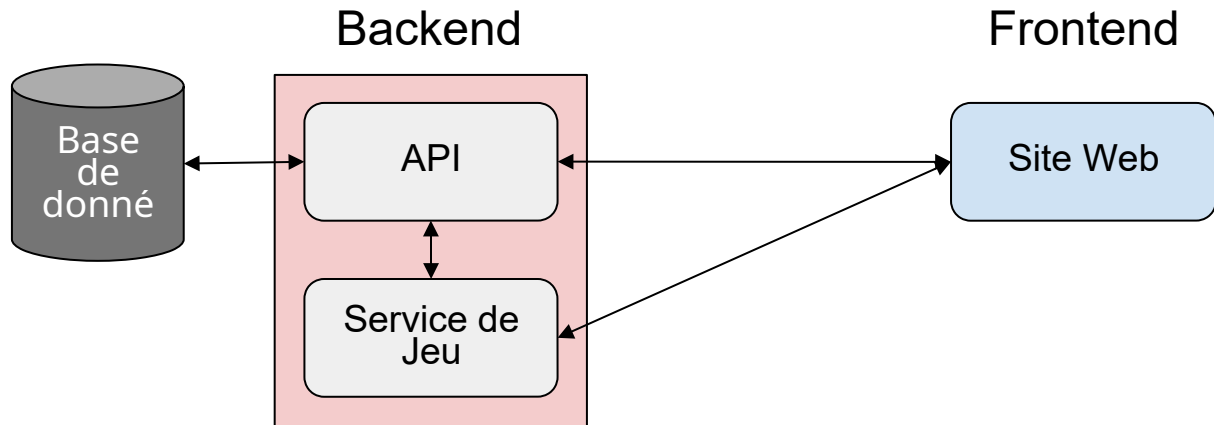


Schéma de l'architecture du

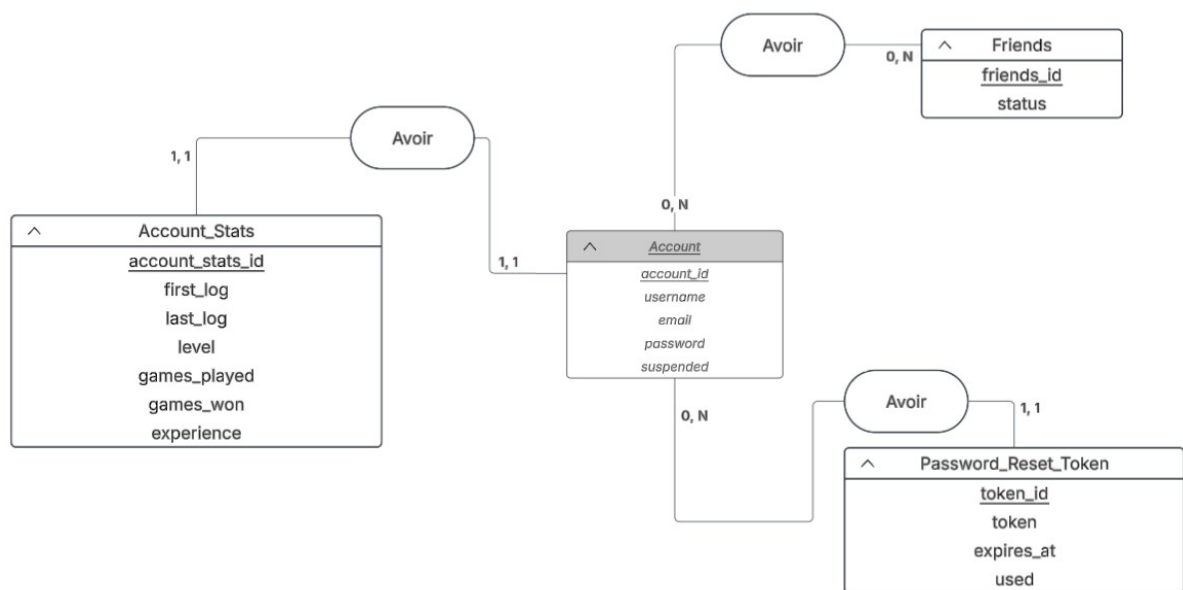


Schéma modèle conceptuel de données

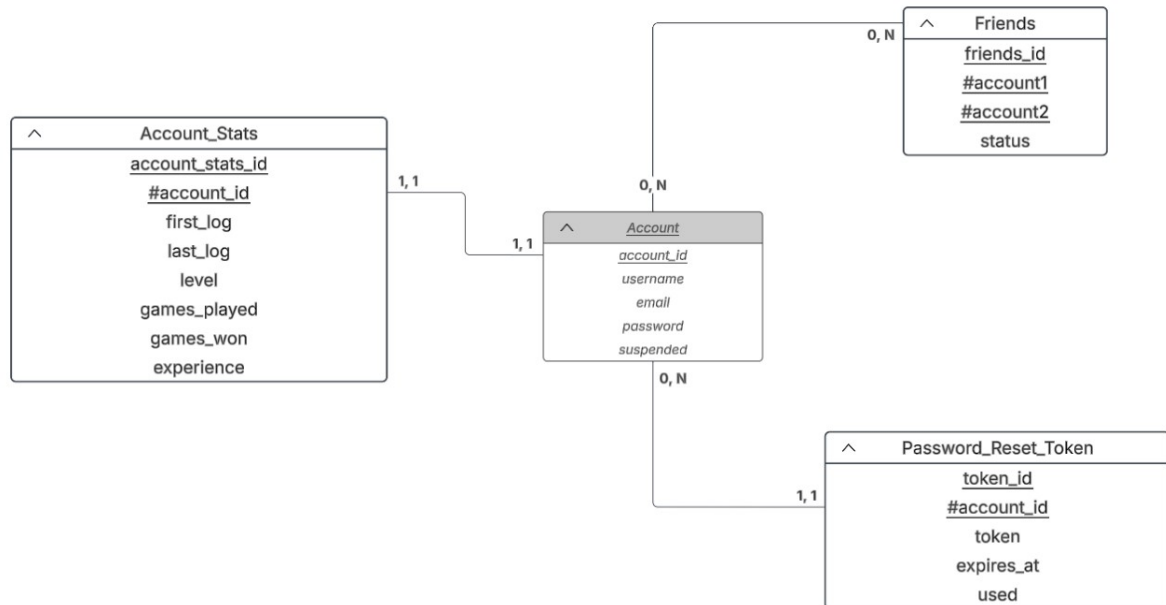
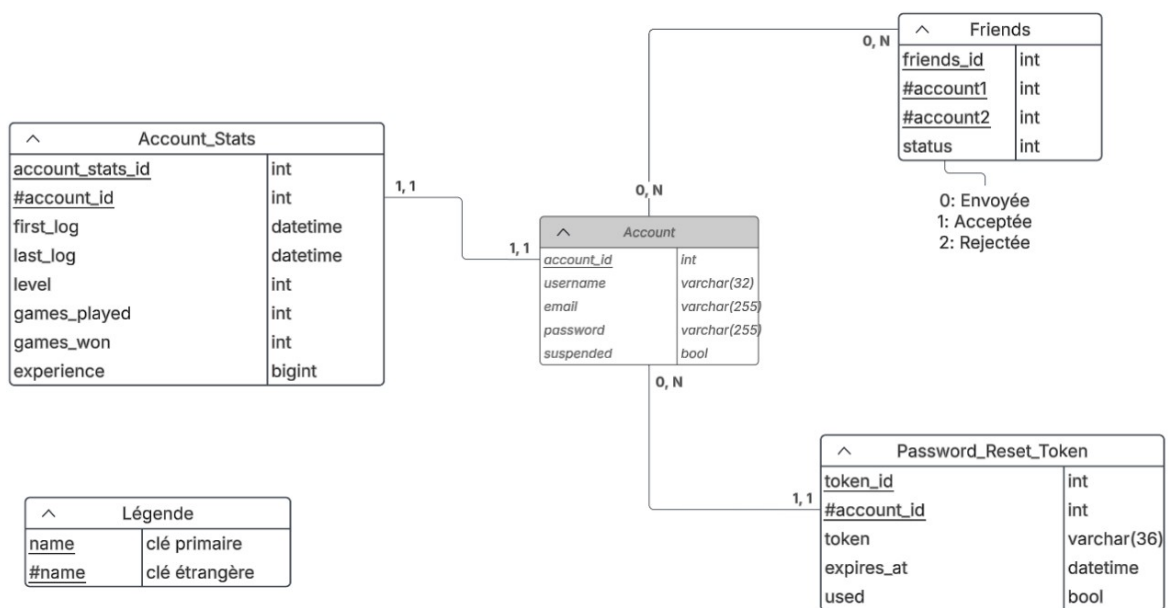
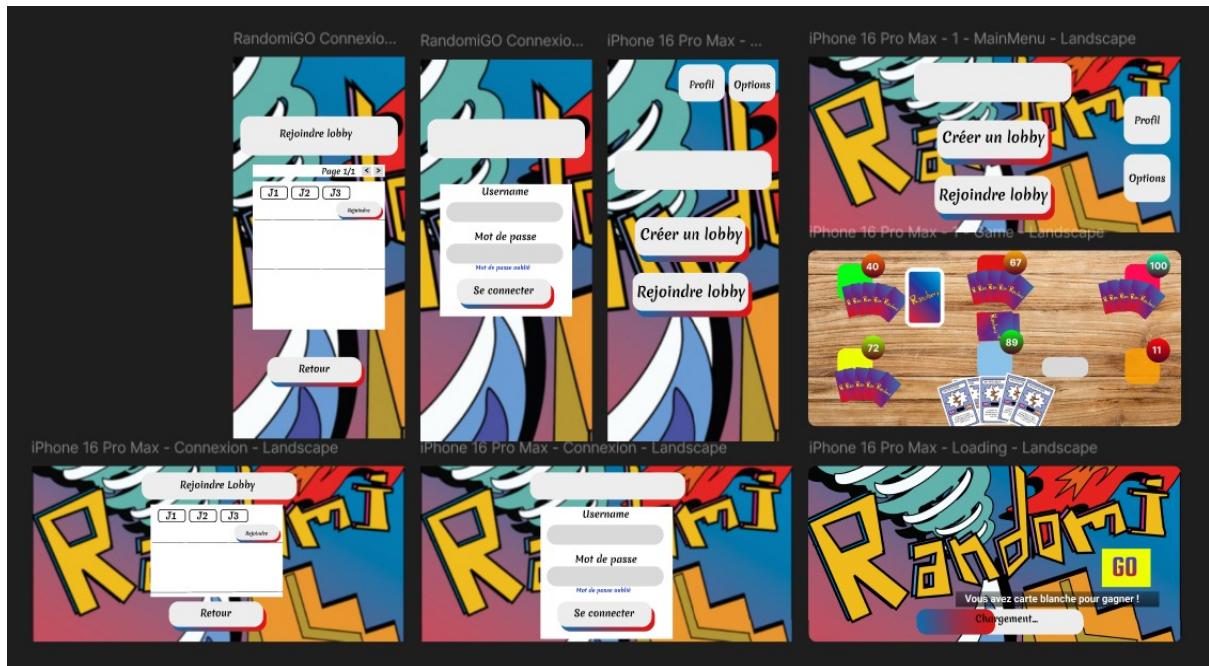
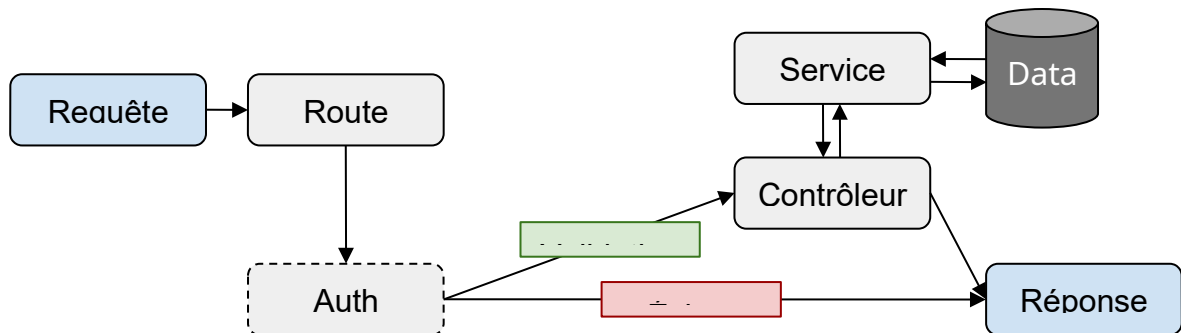


Schéma modèle logique de données





Maquette des pages de l'application



Architecture de

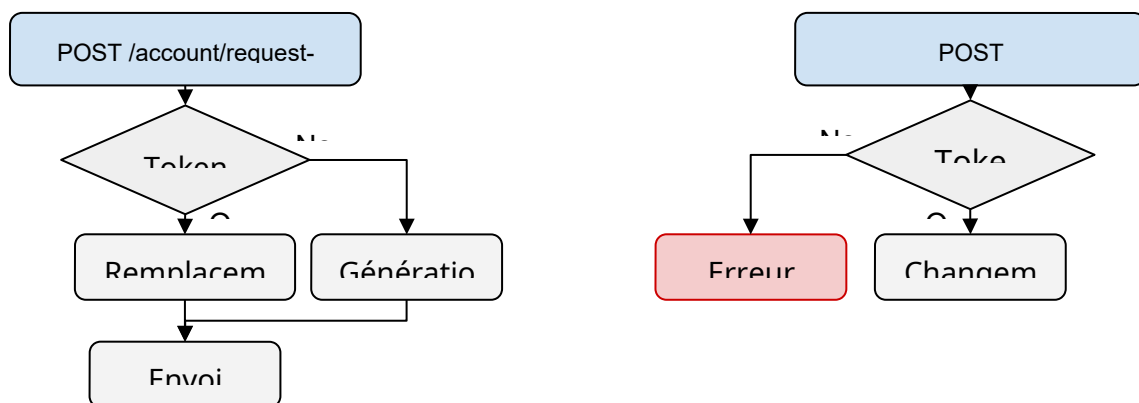
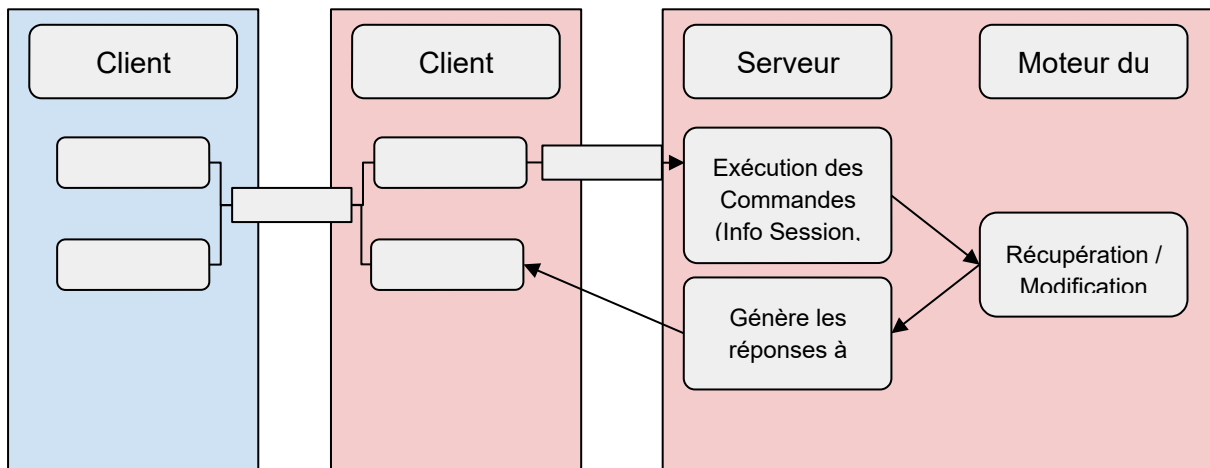


Schéma étapes de récupération de compte

II. Communication entre le client et le service de jeu



Plan de communication entre les clients et le service de jeu

PlayCard JSON	
action_type	string "PlayCard"
card_index	int
targets	list of player ids (one entry if single target). Only opponents should be present (ex. heal does not require any targets)

**Action du client
pour jouer une carte**

Card Played JSON	
type	string "PlayCard"
player_id	id of the player who played the card
card_id	id
hand_index	index of card in player's hand
actions	list of actions to perform on players

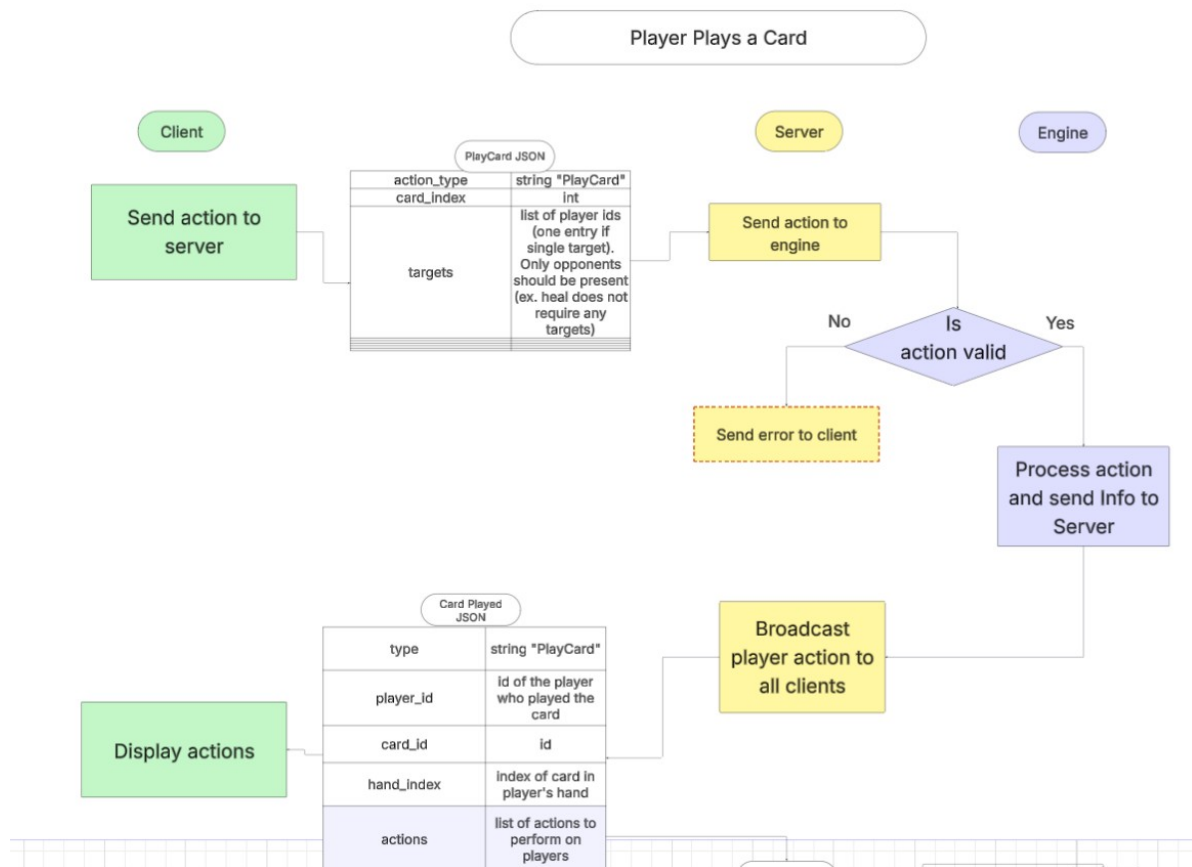
**Réponse du serveur
lorsqu'une carte est
jouée**

Structures des actions réalisables par les cartes

```
#[derive(Debug, Clone, Serialize, Deserialize)]
5 implementations
pub struct PlayAction {
    pub dice_roll : u8,
    pub player_dice_id: PlayerId,
    pub targets: Vec<ActionTarget>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
4 implementations
pub struct ActionTarget {
    pub player_id : PlayerId,
    pub action: ActionType,
    pub effect: String
}

#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(tag="type")]
5 implementations
pub enum ActionType {
    Attack{amount: u32},
    Heal{amount: u32},
    Draw{cards: Vec<CardId>},
    Discard{cards: Vec<usize>},
    Steal{cards: Vec<usize>},
}
```



Plan d'action lorsque le joueur joue une carte

III. Éléments de conception de l'API

```
// Génère le code nécessaire pour définir
// la fonction comme contrôleur avec:
// - La méthode (ici GET)
// - La route (/account/profile)
#[get("/account/profile")]
24 implementations
async fn get_my_account(req: HttpRequest, pool: web::Data<DbPool>)
```

Définition d'un contrôleur

Définition des données

d'application

```
#[post("/login")]
24 implementations
async fn login(
    json: web::Json<AccountLogin>,
    pool: web::Data<
) -> actix_web::Resu
    let account: Fil
        // Obtaining
        // So, it sh
    }
```

Exemple d'utilisation des
extracteurs: corps de la requête

```
backend::database::actions
```

```
pub struct AccountLogin {
```

```
    pub username: String,
```

```
    pub password: String,
```

```
}
```

```
App::new()
    .wrap(Logger::default())
    .app_data(web::Data::new(pool.clone()))
    .app_data(web::Data::new(backend_db.clone()))
    .app_data(web::Data::new(server_handlers.clone()))
    .app_data(web::Data::from(Arc::clone(&broadcaster)))
    .app_data(web::Data::new(mailer.clone()))
```

```
let database_url: String = std::env::var("DATABASE_URL").expect("DATABASE_URL env var not set !");
let manager: ConnectionManager<PgConnection> = r2d2::ConnectionManager::new(database_url.clone())
let pool: DbPool = r2d2::Pool::builder()
    .build(manager)
    .expect(format!("Unable to connect to database with URL \"{}\" !", database_url).as_str());
println!("Connected to database!");
```

```
let database_url: String = std::env::var("DATABASE_URL").expect("DATABASE_URL env var not set !");
let manager: ConnectionManager<PgConnection> = r2d2::ConnectionManager::<PgConnection>::new(database_url.clone())
let pool: DbPool = r2d2::Pool::builder()
    .build(manager)
    .expect(format!("Unable to connect to database with URL \"{}\" !", database_url).as_str());

println!("Connected to database!");
```

Connexion à la base de données Enregistrement des contrôleurs du

```
pub fn configure_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(factory: get_my_account) &mut ServiceConfig
        .service(factory: get_other_account);
}
```

```
App::new()
    // [...]
    // auth Génération du
    // Appel des fonctions de
    .configure(routes::auth::configure_routes)
    // accounts
    .configure(routes::account::configure_routes)
    // stats
    .configure(routes::stats::configure_routes)
```

```
#[derive(Debug, Deserialize, Serialize)]
3 implementations
pub struct Claims {
    sub: String,
    exp: usize,
    user_id: i32,
}

pub fn create_jwt(user_id: i32) -> String {
    let claims: Claims = Claims {
        sub: user_id.to_string(),
        exp: (Utc::now() + Duration::days(1)).timestamp() as usize,
        user_id,
    };

    let secret: String = std::env::var(key: "BACKEND_SECRET_KEY").unwrap();
    let header: Header = jsonwebtoken::Header::new(Algorithm::HS256);

    encode(&header, &claims, key: &EncodingKey::from_secret(secret.as_ref())).unwrap()
}
```

```

HttpServer::new(move || {
    let cors = Cors::default()
        .allowed_origin(website_url.as_str())
        .allow_any_header()
        .allowed_methods(vec!["GET", "POST", "PATCH", "DELETE", "OPTIONS"])
        .supports_credentials()
        .max_age(3600);

```

Configuration de la garde

```

#[utoipa::path(
    post,
    path = "/login",
    request_body = AccountLogin,
    responses(
        (status = 200, description = "Login successful, returns JWT token as string", body = String),
        (status = 401, description = "Unauthorized or suspended account"),
        (status = 404, description = "Account not found")
    ),
    tag = "Auth"
)]
#[post("/login")]
24 implementations
async fn login(pool: web::Data<DbPool>, json: web::Json<AccountLogin>) -> actix_web::Result<impl Responder> {
    let account: FilteredAccount = web::block(move || {

```

Macros pour la génération de la documentation de l'API

Requête vers /lobby/join depuis Swagger UI

POST

/lobby/join

Parameters

No parameters

Request body required

application/json

```
{
  "lobby_id": "ABCDEFGH"
}
```

Cancel

Reset

```

pub fn list_lobby_page_list_from_db(backend_db: &web::Data<BackendDb>, page: usize) -> Result<LobbyPageList, ()> {
    let lobbies: Collection<Lobby> = backend_db.lobbies_collection();

    // filter out unlisted lobbies
    let entries: ClientCursor<Lobby> = lobbies.find(filter: doc! { "unlisted": false }) Find<'_', '_>, Lo...
        .skip(page*LOBBY_PAGE_SIZE) Find<'_', '_>, Lobby>
        .limit(LOBBY_PAGE_SIZE) Find<'_', '_>, Lobby>
        .run()?;

    let page_count: usize = (lobbies.len() as f64 / LOBBY_PAGE_SIZE as f64).ceil() as usize;

    Ok(LobbyPageList { entries, page, page_count })
}

```

```
pub fn list_lobby_page_list_from_db(backend_db: &web::Data<BackendDb>, page: usize) -> Result<LobbyPageList, ()> {
    let lobbies: Collection<Lobby> = backend_db.lobbies_collection();

    // filter out unlisted lobbies
    let entries: ClientCursor<Lobby> = lobbies.find(filter: doc! { "unlisted": false }) Find<'_, '_>, Lo...
        .skip(page*LOBBY_PAGE_SIZE) Find<'_, '_>, Lobby>
        .limit(LOBBY_PAGE_SIZE) Find<'_, '_>, Lobby>
        .run()?;

    let page_count: usize = (lobbies.len() as f64 / LOBBY_PAGE_SIZE as f64).ceil() as usize;

    Ok(LobbyPageList { entries, page, page_count })
}
```

Récupération de la liste des salles par pages

```
#[derive(Serialize, Debug)]
#[serde(tag = "type")]
3 implementations
pub enum SseMessage {
    FriendshipRequest { request_id: i32, user: i32, status: i32 },
    FriendshipDeleted { id: i32 },
    LobbyUserListChange { users: HashSet<i32> },
    LobbyUserReadyChange { user: i32, ready: bool },
    GameStarted { game_id: GameId }
}

impl SseMessage {
    pub async fn send(&self, tx: &mpsc::Sender<sse::Event>) -> Result<(), mpsc::error::SendError<sse::Event>> {
        let data: Event = sse::Data::new_json(data: &self).unwrap().into();
        log::info!("Sending SseMessage: {:?}", data);
        tx.send(data).await
    }
}
```

Définition des structures des Events du SSE

```

events = new EventSource(api_url("/events"), { withCredentials: true });

// listen to onopen and onerror events [...]

events.onmessage = (ev) => {
  let json_data = ev.data;

  // parsing data to JSON [...]

  switch(json_data["type"]) {
    case "FriendRequest":
      handle_friend_request_update(json_data["request_id"], json_data["user"], json_data["status"]);
      break;
    case "FriendshipDeleted":
      handle_friendship_deleted(json_data["id"]);
      break;
    case "LobbyUserListChange":
      handle_lobby_user_list_change(json_data["users"]);
      break;
    case "LobbyUserReadyChange":
      handle_lobby_user_ready_change(json_data["user"], json_data["ready"]);
      break;
    case "GameStarted":
      window.location.href = "/ingame.html";
      break;
    default:
      console.error(`SSE: Unrecognized message type: ${json_data["type"]}`);
      break;
  }
}

```

*Exemple logique de mise à jour
des entrées du panel de gestion*

Connexion et mises à jour avec les SSE

```

async handleFriendRequestUpdate(request_id, user_id, status) {
  switch (status) {
    // waiting
    case 0: {
      // "fake" req data
      const req = { "id": request_id, "account1": user_id, "account2": APP_STATE.account.id, "status": status };
      const entry = await this.createFriendRequestEntry(req);
      this.friendRequestsDiv.prepend(entry);
    }
    break;
    // accepted
    case 1: {
      const requestElement = document.getElementById(`friend-request-entry-${request_id}`);
      if (requestElement) {
        let username = "Unknown";

```


IV. Définitions des modèles de tables en Rust

```
#[derive(Queryable, Selectable, Insertable, Serialize)]
#[diesel(table_name = super::schema::accounts)]
#[diesel(check_for_backend(diesel::pg::Pg))]
6 implementations
pub struct Account {
    pub id: i32,
    pub username: String,
    pub email: String,
    pub password: String,
    pub premium: bool,
    pub suspended: bool,
}
```

Modèle de la table Accounts (filtré)

Modèle de la table Accounts en Rust (complet)

```
// Account struct with sensible fields hidden from the user
#[derive(Queryable, Selectable, Insertable, Serialize)]
#[diesel(table_name = super::schema::accounts)]
#[diesel(check_for_backend(diesel::pg::Pg))]
7 implementations
pub struct FilteredAccount {
    pub id: i32,
    pub username: String,
    pub suspended: bool,
}
```

```
pub fn get_account_by_id(conn: &mut PgConnection, account_id: i32) -> diesel::QueryResult<FilteredAccount> {
    use super::schema::accounts::dsl::*;

    let account: FilteredAccount = accounts.select(selection: FilteredAccount::as_select()) SelectStatement<...
        .filter(id.eq(&account_id)) SelectStatement<FromClause<...>, ...>, ...>
        .first::<FilteredAccount>(conn)?;

    Ok(account)
}
```

```

let new_account: NewAccount = NewAccount {
  username: username.to_string(),
  email: email.to_string(),
  password: hashed_password,
};

conn.transaction(|conn: &mut PgConnection| {
  insert_into(target: accounts) IncompleteInsertStatement<...>
    .values(records: &new_account) InsertStatement<table, ValuesClause...
    .execute(conn)?;

  // get newly inserted account
  let account: FilteredAccount = accounts.select(selection: FilteredAccount::as_select())
    .order_by(expr: id.desc()) SelectStatement<FromClause<...>, ..., ..., .....
    .first(conn)?;

  // create stats entry
  insert_into(target: account_stats) IncompleteInsertStatement<...>
    .values(records: NewEmptyStats {account_id: account.id} ) InsertStatement<table, Val
    .execute(conn)?;

  Ok(account)
})

```

***Exemple d'utilisation des
transactions avec Diesel***

V. Développement du site web

```
/** @type {AccountDTO | null} */
export async function login_guard() {
  try {
    const account = await get_my_account();
    if (account == null)
      throw new Error("You are not logged in !")
      Mise en cache des données d'un
      compte sur le navigateur
    APP_STATE.account = account;
    return account;
  } catch (error) {
    console.log("Error: " + error.message);
    window.location.href = "/login.html";
    APP_STATE.account = null;
    return null;
  }
}
```

Fonction de garde de connexion des

```
const store = DB
  .transaction(ACCOUNT_NAMES_STORE, "readwrite")
  .objectStore(ACCOUNT_NAMES_STORE);

const record = { account: accountDTO, date: Date.now() };

const request = store.put(record);
```

```

customElements.define("profile-stats", ProfileStats);
customElements.define("profile-settings", ProfileSettings);
customElements.define("profile-info", ProfileInfo);
customElements.define("profile-panel", ProfilePanel);
customElements.define("other-profile-panel", OtherProfilePanel);

```

Cible l'adversaire (attaque ou vol

```

<div id="friend-panel-backdrop" class="panel-backdrop"></div>
<friend-panel id="friend-panel"></friend-panel>

```

<!-- Profile panel -->

```

<div id="profile-panel-backdrop" class="panel-backdrop"></div>
<profile-panel id="profile-panel"></profile-panel>

```

<!-- Other Profile panel -->

```

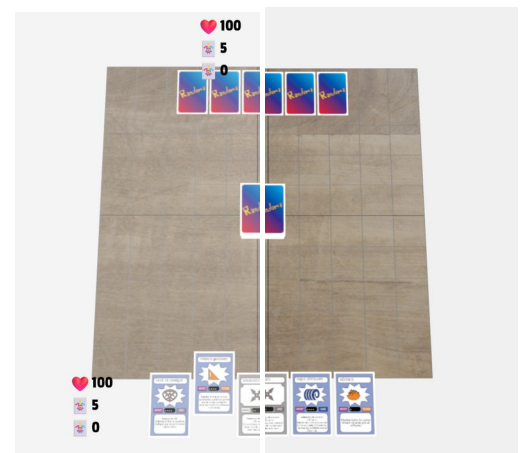
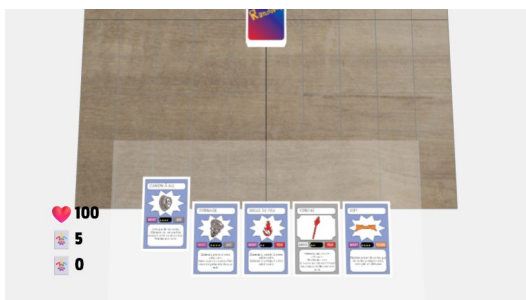
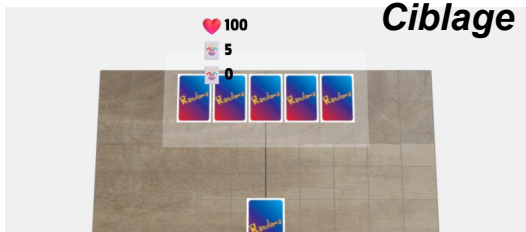
<div id="other-profile-panel-backdrop" class="panel-backdrop"></div>
<other-profile-panel id="other-profile-panel"></other-profile-panel>

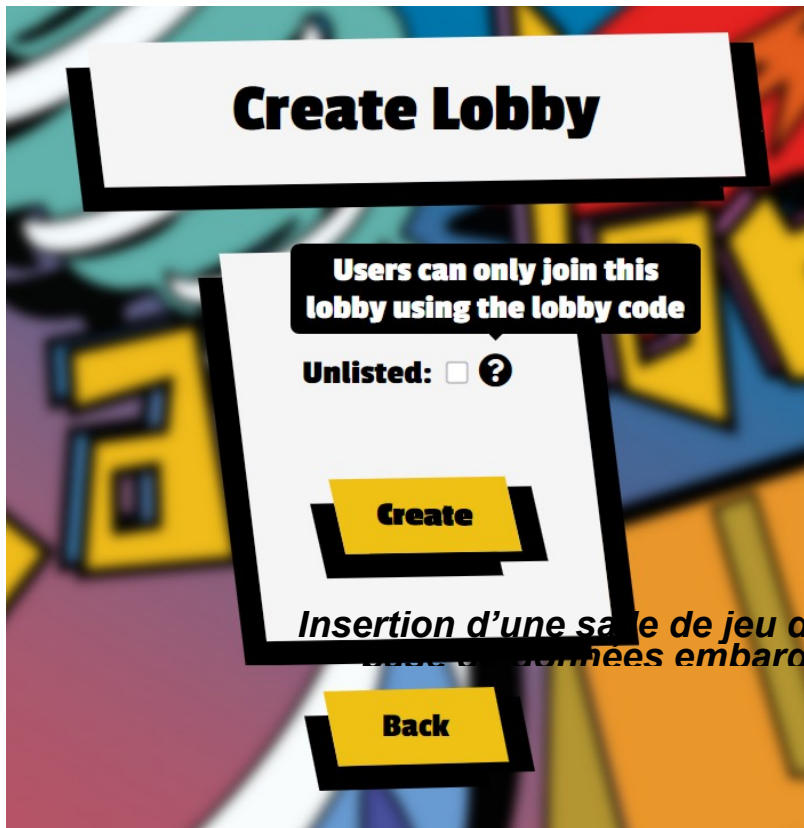
```

Scène en légère surbrillance si

Utilisation d'éléments customs dans le LITM

Ciblage des joueurs dans la scène





Vue de création d'une salle de jeu

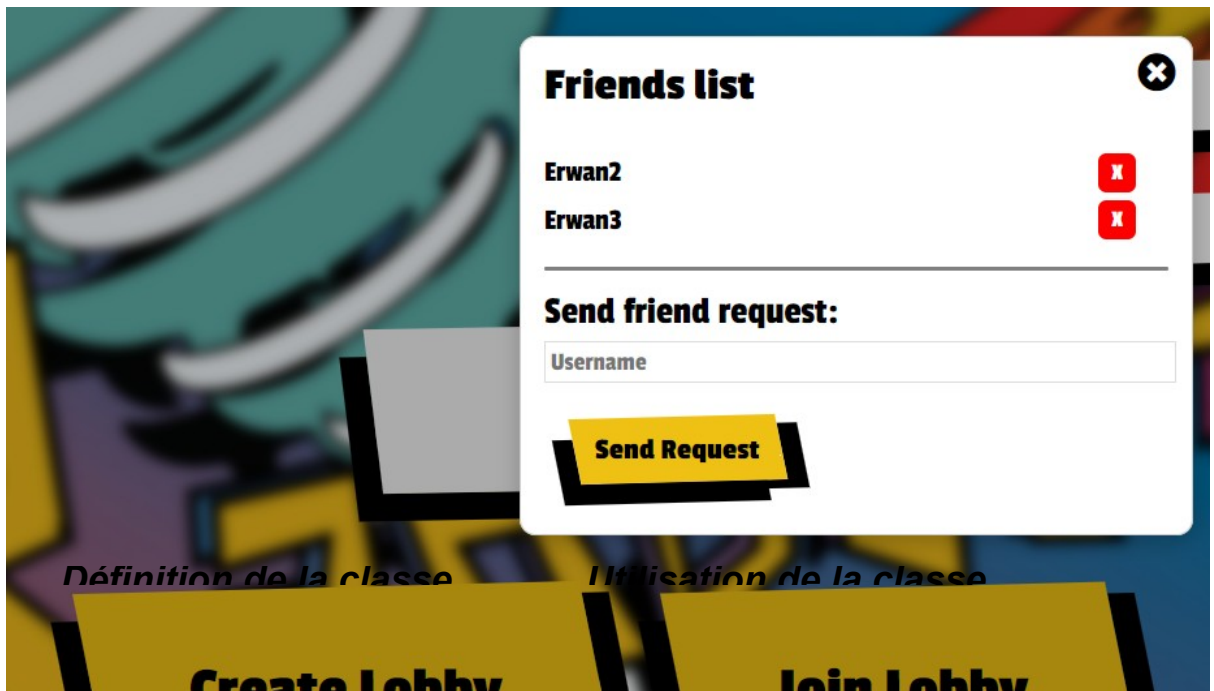
```
let lobbies: Collection<Lobby> = backend_db.lobbies_collection();

if get_lobby_id_for_user(account_id, &lobbies).is_some() {
    return Err(ErrorConflict("User is already in a lobby !"));
}

let lobby_id: String = generate_lobby_id(&lobbies)
    .map_err(error::InternalServerError)?;

let mut lobby: Lobby = Lobby::new(lobby_id.clone(), json.unlisted);
lobby.users.insert(account_id);

lobbies.insert_one(&lobby);
```



Vue du panel de gestion des amis

```
export class GameEvent {
  /**
   * @type {EventManager | null}
   * set by EventMgr when pushed to the queue
   */
  mgr = null;
  timeout = 250; // in ms
  #started = 0;

  constructor() {}

  // reimplement logic here
  async run() {}

  // read-only
  get started() { return this.#started; }

  /**
   * called by EventMgr
   * do not reimplement this function, reimplement run() instead
   */
  execute() {
    // if timeout < 0, it's up to run() to call notifyMgr() when done
    if (this.timeout >= 0) {
      setTimeout(() => { this.onTimeout(); }, this.timeout);
      this.#started = Date.now();
      this.run();
    }

    // tells the EventMgr to execute next event in queue
    onTimeout() { this.mgr.executeNext(); }
  }
}

export class PutCardInPile extends GameEvent {
  constructor(player, card) {
    super();
    this.timeout = 400;
    this.player = player;
    this.card = card;
  }

  async run() {
    if (this.card != null) {
      this.card.active = false;
      // transfer card to discard pile
      this.player.discard_cards.push(this.card);
      // remove card from hand if it's not fake
      const idx = this.player.cards.indexOf(this.card);
      if (idx != -1) {
        this.player.cards.splice(idx, 1);
      }
      this.player.updateHandCardPositions();
      this.player.emitCardCountChange();
      this.player.updateDiscardCardPositions();
      this.player.emitDiscardCountChange();
    }
  }
}
```

```
body {
  font-size: x-large;
}

/* [...] */

/* MOBILE VIEW */
@media all and ((max-width: 480px) or (max-height: 480px)) {
  body {
    font-size: medium;
  }

  /* [...]
  Comparaison de la taille et de l'agencement
  des éléments en fonction des dimensions de
  l'écran
  */
}
```

Définition de la taille de la police du corps de la page en fonction des dimensions de l'écran

