QuizConnect

Plateforme de Jeu en Ligne Synchronisée

Travail de Bachelor

Ali Gökkaya

Juillet 2024

Supervisé par :

Prof. Dr. Jacques Pasquier-Rocha
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Remerciements

Je tiens à exprimer ma profonde gratitude au Prof. Dr. Jacques Pasquier-Rocha, directeur du Groupe Génie Logiciel à l'Université de Fribourg, pour sa supervision et son soutien tout au long de ce projet. Son expertise et ses conseils précieux ont été d'une grande aide dans la réalisation de ce travail.
Je remercie également Quentin Nater, assistant de recherche, et Paul Ricci, un collègue, pour leurs nombreuses recommandations et leur soutien moral.
Enfin, un grand merci à tous ceux qui m'ont soutenu, que ce soit par leur aide, leurs encouragements ou simplement leur compagnie durant cette aventure. Grâce à eux, j'ai pu surmonter les défis de ce projet avec enthousiasme.

Résumé

Sous la supervision du Prof. Dr. Jacques Pasquier-Rocha du Groupe Génie Logiciel de l'Université de Fribourg, Ali Gökkaya a réalisé un travail de Bachelor consacré au développement d'une plateforme de jeu en ligne innovante. Ce projet a pour ambition de permettre aux utilisateurs de créer des comptes personnels, de configurer des sessions de jeu interactives et d'inviter d'autres joueurs via un code unique généré par le serveur. La synchronisation en temps réel de chaque tour de jeu, assurée par un serveur central, garantit une expérience utilisateur fluide et immersive.
L'application front-end a été développée en utilisant React et Bootstrap, offrant une interface utilisateur réactive et adaptée à différents supports. Pour le back-end, Node.js et Go ont été utilisés, apportant performance et évolutivité au système. La communication en temps réel est gérée grâce à WebSocket, permettant une interaction instantanée entre les participants et une gestion efficace des sessions de jeu. Les données relatives aux utilisateurs et aux sessions sont stockées de manière sécurisée dans une base de données SQLite.
Le rapport détaille les motivations du projet, la conception des fonctionnalités, l'architecture du système, ainsi que les aspects techniques liés au développement et à l'expérience utilisateur. Une importance particulière a été accordée à la programmation réactive pour gérer les opérations asynchrones, améliorant ainsi la réactivité et les performances globales de l'application.
Les tests du prototype ont confirmé l'efficacité du système et la pertinence des choix technologiques effectués. Le projet se conclut par des perspectives d'amélioration, notamment l'ajout de nouvelles fonctionnalités et l'optimisation continue des performances. Ce travail illustre l'application concrète de concepts modernes en développement web pour créer une plateforme de jeu en ligne à la fois innovante et performante.

Mots-clés : plateforme de jeu en ligne, synchronisation en temps réel, WebSocket, programmation réactive, systèmes distribués, React, Node.js, Go, SQLite.

Table des matières

Liste des figures

Liste des tableaux

Liste des codes source

1
Introduction

1.1. Motivations et Objectifs

Actuellement en deuxième année de Bachelor en informatique à l'Université de Fribourg, en Suisse, j'ai choisi de consacrer mon travail de Bachelor au développement d'une plateforme de jeu en ligne. Cette plateforme permettra aux utilisateurs de créer des comptes, de générer des sessions de jeu, et d'inviter d'autres joueurs grâce à un code unique attribué par le serveur. Pour mener à bien ce projet, j'ai sollicité le groupe de recherche "Software Engineering Group", dirigé par le Prof. Dr. Jacques Pasquier-Rocha, qui a accepté de superviser mon travail.

L'objectif principal de ce projet est de concevoir un système robuste et interactif, où chaque tour de jeu est synchronisé par le serveur, garantissant ainsi une expérience utilisateur fluide et engageante. Ce projet me permettra de renforcer mes compétences en développement web et en gestion de bases de données, tout en explorant des aspects fondamentaux de la conception de systèmes distribués et interactifs en temps réel. Mon ambition est de livrer un prototype fonctionnel capable de synchroniser plusieurs clients via WebSocket, assurant ainsi une qualité et une performance optimales du jeu.

1.2. Organisation du Document

Chapitre 1 : Introduction

L'introduction présente les motivations et les objectifs de ce travail, accompagnée d'un bref aperçu de la structure de chaque chapitre ainsi que des conventions de formatage employées.

Chapitre 2 : Fonctionnalités et modélisation

Ce chapitre explore l'idée principale du projet ainsi qu'un exemple de projet similaire existant. Il aborde également les cas d'utilisation du prototype et la modélisation de la base de données.

Chapitre 3 : Application du point de vue utilisateur

Cette section analyse l'utilisation du prototype à travers des explications illustrées par des captures d'écran, facilitant ainsi la compréhension des fonctionnalités proposées.

Chapitre 4 : Application du point de vue développeur

Ce chapitre présente la partie développement du projet. Il détaille l'initialisation des technologies et l'architecture globale du prototype, illustrée par des blocs de code pertinents.

Chappitre 5 : Résultats et perspectives d'amélioration

Ce dernier chapitre nous présente les résultats obtenus ainsi que les perspectives d'améliorations possibles de cette application pour une utilisation en production.

1.3. Notations et Conventions

— Le rapport est rédigé en français.

— Le code source apparaît comme suit :

```
// Exemple de code
func helloWorld(){
    fmt.Println("Hello World")
}
```

— Les commandes à exécuter dans le terminal sont présentées comme suit :

```
cd Desktop
```

— L'intégralité du code source est disponible dans le dossier GitHub, dont le lien [1] est communiqué à la fin de ce rapport.

— Lors de la rédaction de ce rapport, ChatGPT d'OpenAI a été utilisé pour corriger les erreurs d'orthographe et de syntaxe, afin de garantir un document plus agréable à lire. À cet effet, le guide d'utilisation de l'IA publié par l'Université de Neuchâtel a été suivi [26].

[1] `https://github.com/agkkaya321/unifrTB`

2

Fonctionnalités et modélisation

2.1. Contexte

Dans ce chapitre, nous allons essayer d'analyser le contexte de mon projet, ainsi que son utilité et les cas d'utilisation du prototype.

2.1.1. Mon Projet

Mon projet consiste en un jeu de questions en ligne qui synchronise plusieurs clients. Tout d'abord, il est nécessaire de créer un compte et de configurer des salons de jeu. Un salon de jeu est composé d'une série de questions créées par l'utilisateur. Ensuite,

l'administrateur du jeu peut lancer la partie et recevoir un code unique qui permet aux autres joueurs de rejoindre le jeu. Un nombre défini de questions est posé aux participants ; les questions peuvent être identiques ou différentes pour chaque joueur. À chaque question, les joueurs doivent répondre comme ils pensent qu'une autre personne le ferait. Toutes ces questions et réponses sont ensuite collectées et projetées sur un grand écran (Displayer). Les participants peuvent voter depuis leur interface pour gagner des points. À chaque tour, ils doivent deviner qui a répondu et à la place de qui.

2.1.2. Projet semblable

Actuellement, sur le marché, il existe plusieurs quiz web qui offrent la synchronisation de plusieurs clients afin de créer une session sous forme interactive. La majeure partie de ces jeux fonctionne sur le principe de questions préparées qui sont posées aux joueurs, et une ou plusieurs bonnes réponses sont attendues. Parmi cette vaste panoplie, l'un des plus connus est Kahoot [15].

2.1.3. Kahoot

Kahoot est une plateforme d'apprentissage ludique qui permet de créer et de participer à des quiz interactifs en ligne. Autant utilisée dans les écoles que dans les entreprises, elle facilite l'engagement et l'apprentissage par le jeu. Avec Kahoot, les utilisateurs peuvent renforcer leurs connaissances tout en s'amusant. Dans mon cadre personnel, je l'utilise surtout dans un cadre de loisir.

À l'origine, cette plateforme est disponible sur le web, mais il existe aussi des applications qui sont codées de manière native pour les plateformes de téléphones. Il est possible de participer à ce jeu de manière simple avec un smartphone.
Ci-dessous, il est possible de voir les trois vues principales de l'application. La première vue (figure 2.1) est celle de la bibliothèque de jeux où se trouvent les quiz qu'un joueur possède. Sur la figure suivante (figure 2.2), on peut voir comment le jeu est affiché sur l'écran principal où il est projeté, et enfin, la vue d'un utilisateur qui vote sur un smartphone sur la figure 2.3.

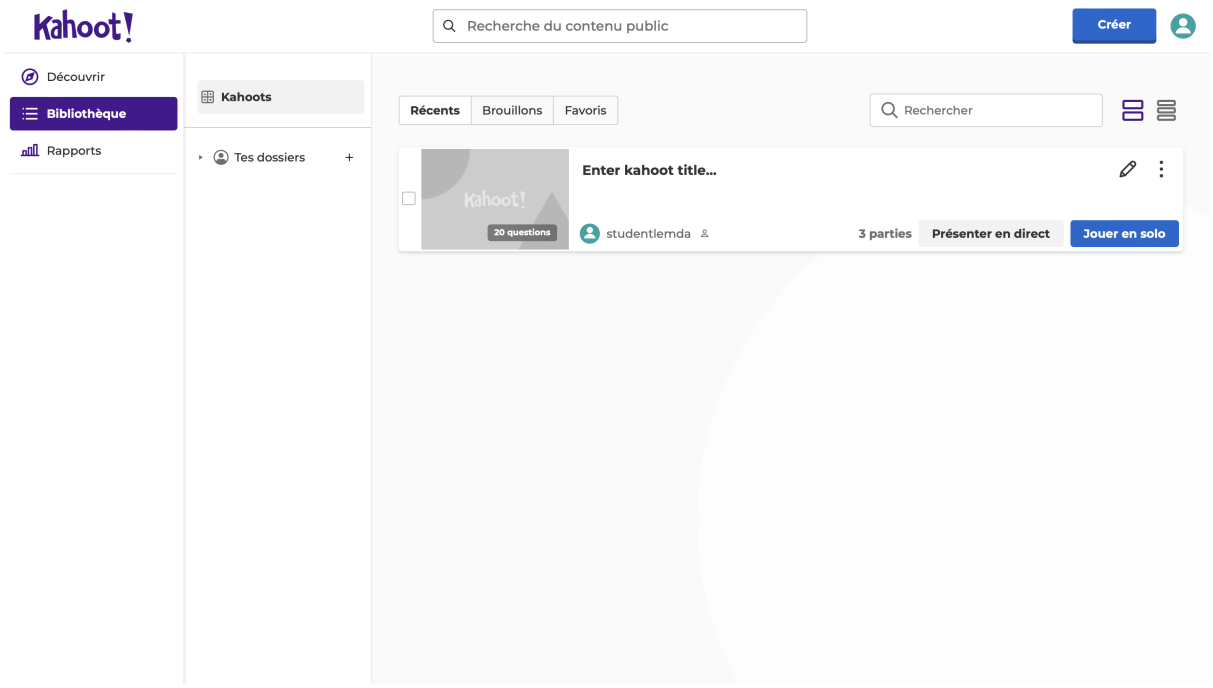

FIGURE 2.1. – Bibliothèque de quiz sur Kahoot

La figure 2.1 illustre la présentation d'une bibliothèque de jeux sur la plateforme Kahoot. Dans cet exemple, nous nous focalisons sur les jeux disponibles. Ma bibliothèque ne contient qu'un seul jeu déjà créé, qui dispose de deux boutons pour le lancer, offrant ainsi deux modes de jeu distincts. Le premier bouton, «Présenter en direct», permet d'accéder au mode multijoueur, tandis que le second, «Jouer en solo», propose un mode solo.

FIGURE 2.2. – Ecran principale avec le quiz affiché

La figure 2.2 illustre la présentation d'une question dans le mode multijoueur. On y observe la question «Quel est le plat préféré d'Ali» en haut de l'écran, suivie des quatre options de réponse de couleurs différentes que les joueurs peuvent sélectionner.

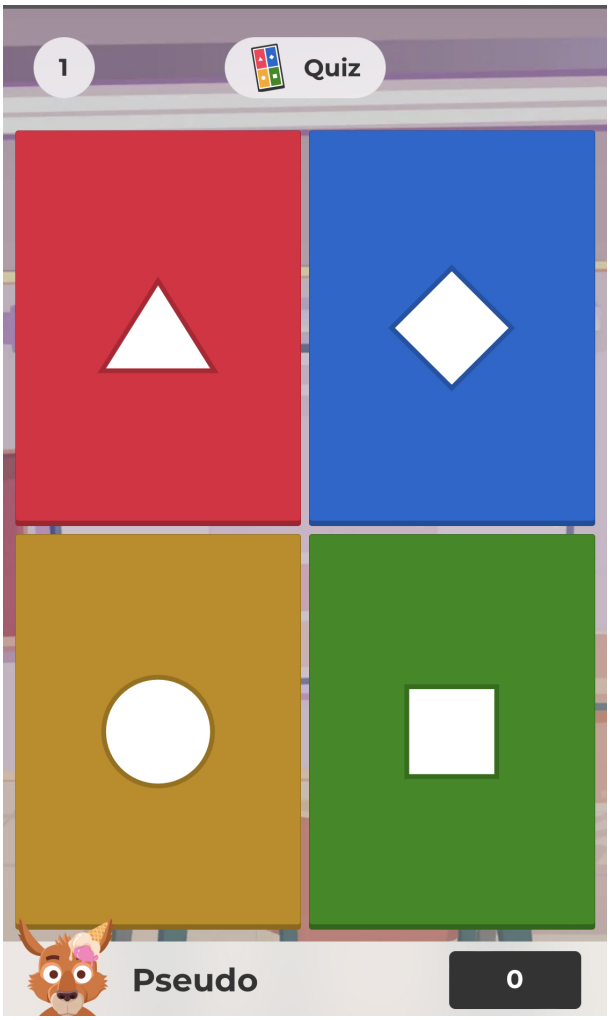

FIGURE 2.3. – Ecran pour voter une réponse

La figure 2.3 présente l'interface utilisateur que les joueurs voient lorsqu'ils votent pour les différentes réponses possibles, identifiées par leurs couleurs respectives. Il est également possible de voir le pseudonyme et le score du joueur sous les réponses.

Le déroulement de Kahoot est le suivant : au préalable, un joueur prépare des questions et réponses en indiquant à chaque fois la ou les bonnes réponses. Ensuite, l'ensemble de ces questions apparaît dans sa bibliothèque de jeux, comme présenté sur la figure 2.1. Plusieurs joueurs peuvent rejoindre le jeu via un lien communiqué. Une fois que tous les joueurs sont connectés, le jeu projette toutes les questions dans l'ordre sur un écran visible par tous, comme dans l'exemple de la figure 2.2, où nous voyons la question "Quel est le plat préféré d'Ali" avec les réponses Kebab, Lasagne, Pâtes ou Café & Croissant. Pendant que cela est projeté, chaque joueur a accès, depuis le client avec lequel il s'est connecté, à une vue comme celle de la figure 2.3, où il peut voter via les couleurs ou les formes associées à la bonne réponse. Ce processus se répète pour toutes les questions.

2.1.4. Principes de base

Dans la majorité des cas, le déroulement d'un quiz web suit le schéma suivant : tout d'abord, une session de jeu est créée, à laquelle les différents participants se connectent et se synchronisent. Ensuite, les questions sont présentées successivement, permettant aux joueurs de répondre. À la fin de chaque tour, les résultats et les scores sont affichés avant de passer à la question suivante.

2.1.5. Principes retenus

Il est essentiel de respecter certains principes de base pour garantir que le jeu soit cohérent et compréhensible pour les joueurs. Voici une liste des principes de base retenus dans mon application :

1. **Système de login et d'inscription** qui permet aux joueurs de créer leur propre bibliothèque de jeux.
2. **Système de synchronisation** via WebSocket qui permet une communication en direct entre les joueurs et le serveur.
3. **Un écran d'affichage** qui englobe et résume l'état actuel du jeu.
4. **Système de points** qui est affiché aux joueurs à la fin de chaque manche pour leur donner un aperçu.

2.1.6. Déroulement de mon jeu

Voici une explication du déroulement de mon jeu, illustrée par le schéma de la figure 2.4. Dans ce schéma :

— Les losanges avec un plus signifient que tous les événements précédents doivent se produire avant de continuer.
— Les losanges avec une étoile signifient que l'on passe à l'activité suivante seulement si la condition indiquée est remplie.
— Un cercle fin représente le début du jeu.
— Un cercle épais représente la fin du jeu.

1. **Connexion et préparation :**
   — Le jeu commence par la création d'une partie par un administrateur. Cet utilisateur configure un écran partagé appelé *Displayer*, visible par tous les joueurs (voir figure 3.14).
   — Le *Displayer* affiche un code unique à 5 chiffres généré par le serveur. Ce code est utilisé par les autres joueurs pour rejoindre la session via la section *Join* de leur interface.
   — Chaque joueur doit entrer un pseudonyme et le code unique pour rejoindre la session. Une fois connectés, leurs noms apparaissent dynamiquement sur le *Displayer*, confirmant leur participation.
   — Une fois que tous les joueurs sont connectés, chacun doit appuyer sur le bouton *Start* pour indiquer qu'il est prêt (voir figure 3.16). Lorsque tous les participants ont validé leur état, le jeu commence automatiquement.

2. **Phase de questions :**
   - Une fois le jeu lancé, chaque joueur reçoit trois questions aléatoires sur son interface personnelle (voir figure 3.18).
   - Les questions incluent le nom d'un autre joueur (mis en évidence en rouge). Chaque participant doit répondre comme s'il était ce joueur, en imaginant ce qu'il dirait.
   - Les réponses sont soumises via le bouton *Submit*. Tous les joueurs doivent avoir répondu à leurs trois questions pour que le jeu passe à l'étape suivante.
   - Exemple : Une question pourrait être *"Quel est ton plat préféré ?"*, et le joueur doit répondre en supposant être la personne désignée en rouge.
3. **Phase de quiz interactif :**
   - Une fois les réponses collectées, elles sont anonymement affichées sur le *Displayer* avec la question correspondante (voir figure 3.19).
   - Chaque joueur doit deviner deux éléments :
     a) *Qui* a répondu à la question.
     b) *Pour qui* la réponse a été donnée (le joueur en rouge).
   - Ces choix se font via des menus déroulants sur leur interface individuelle (voir figure 3.21).
   - Une fois les votes soumis, le *Displayer* affiche les bonnes réponses et attribue des points :
     - 1 point pour avoir correctement deviné l'auteur de la réponse.
     - 1 point pour avoir correctement deviné pour qui la réponse a été donnée.
   - Cette phase se répète pour toutes les questions de la session.
4. **Affichage des scores et fin du jeu :**
   - Après chaque question, les réponses correctes ainsi que les scores de tous les joueurs sont affichés sur le *Displayer* pendant une durée de 10 secondes.
   - Une fois que toutes les questions ont été traitées, le *Displayer* présente un récapitulatif final avec les scores globaux de chaque joueur, permettant de visualiser les résultats définitifs de la session (voir figure 3.24).
   - Après l'affichage des scores finaux, tous les participants sont automatiquement redirigés vers la page d'accueil, où ils peuvent décider de créer ou de rejoindre une nouvelle partie.

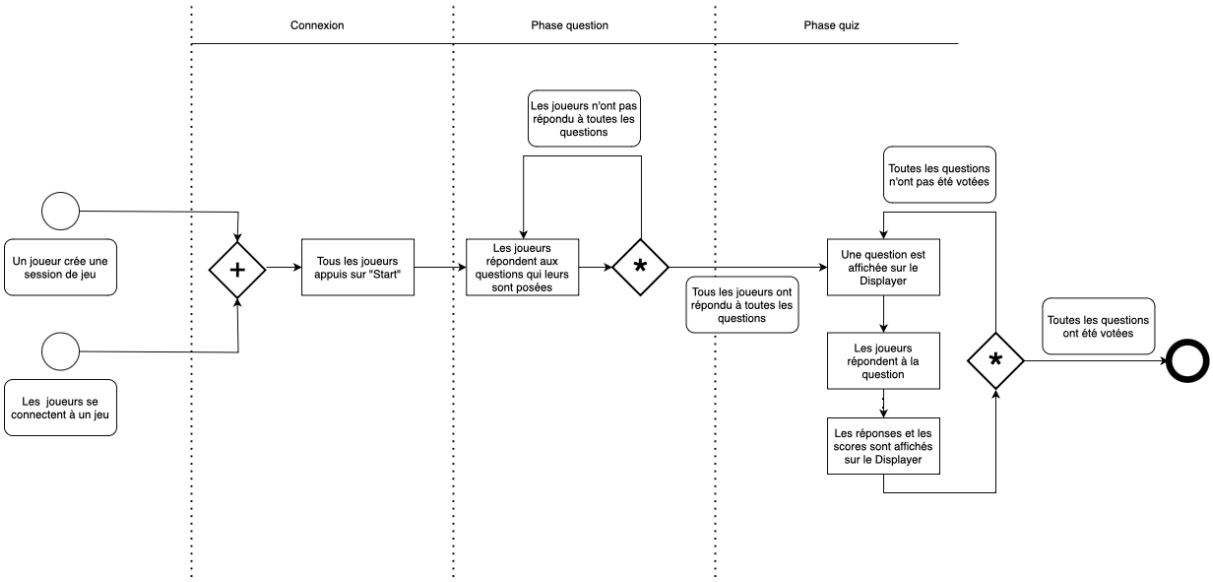

FIGURE 2.4. – Déroulement du Jeu

2.2. Cas d'utilisation du client (sans compte)

Ce premier cas d'utilisation décrit la situation d'un joueur qui arrive pour la première fois sur le site sans s'être enregistré au préalable. Voici le diagramme des cas d'utilisation de l'application du point de vue d'un client qui n'a pas de compte (figure 2.5).

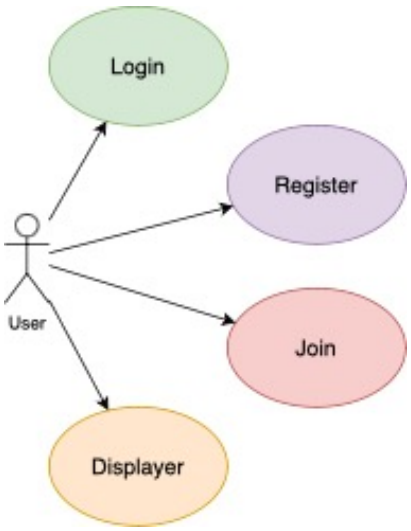

FIGURE 2.5. – Cas d'utilisation du client sans connexion

2.2.1. Login

Ce premier cas d'utilisation permet au client de se connecter et d'accéder à plus de fonctionnalités en tant qu'administrateur de jeu. Cela se fait via une interface de connexion qui sera détaillée dans les prochains chapitres. Une authentification est réalisée en ba-

ckend, ce qui permet d'augmenter automatiquement les droits du client, lui donnant ainsi accès aux fonctionnalités réservées à l'administrateur de jeu.

2.2.2. Register

Ce deuxième cas d'utilisation permet au client de s'enregistrer dans la base de données en fournissant un nom, un e-mail et un mot de passe. Ces informations sont ensuite stockées afin de pouvoir authentifier un client existant lors de l'utilisation de la fonction de connexion.

2.2.3. Join

Le cas d'utilisation "Join" est le point d'entrée à l'application en tant que joueur. Cette fonctionnalité permet de se connecter au serveur à l'aide d'un code de 5 chiffres fourni par l'administrateur de jeu et d'un pseudonyme. Une fois connecté avec ce code, l'interface se transforme en celle d'un joueur, permettant ainsi de participer activement et de prendre des décisions dans le jeu.

2.2.4. Displayer

Le cas d'utilisation "Displayer" est utilisé pour transformer l'interface en tableau de bord, affichant le déroulement du jeu et les scores. Il est essentiel d'en avoir au moins un par partie de jeu. Pour accéder à cette fonctionnalité, il faut insérer le même code à 5 chiffres, mais cette fois-ci en utilisant la fonctionnalité "Displayer".

2.3. Cas d'utilisation du client (avec compte)

Une fois qu'un client utilise la fonction "Register" ou "Login", une authentification côté serveur a lieu pour valider le client. Si l'authentification est réussie, le client accède à davantage de fonctionnalités en tant qu'administrateur de jeu. Voici les cas d'utilisation de l'application en tant qu'administrateur de jeu illustrés sur la figure 2.6.

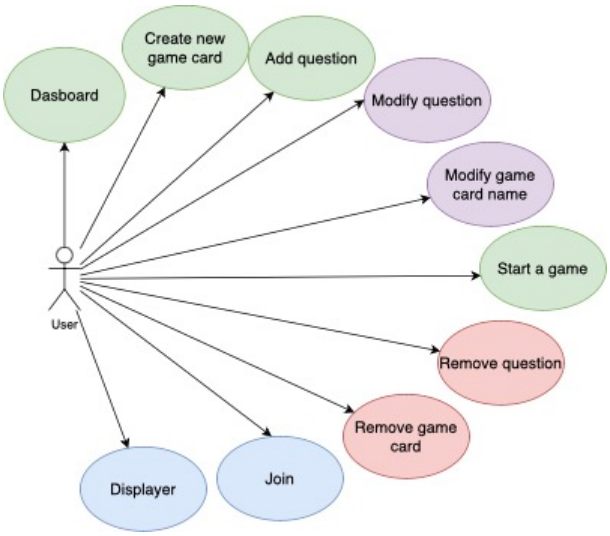

FIGURE 2.6. – Cas d'utilisation client avec connexion

2.3.1. Dashboard

Le Dashboard est la fonctionnalité qui permet à l'administrateur de visualiser et de configurer les cartes de questions associées à son compte. Les cartes de questions représentent les différentes sessions de jeu que l'utilisateur a créées. Elles sont composées d'un titre et de plusieurs questions.

2.3.2. Game card

Il existe plusieurs fonctionnalités autour des cartes de jeu. Un joueur authentifié peut créer, supprimer ou modifier une carte de jeu à sa guise. Une carte de jeu est composée d'un titre et de questions, qui peuvent être modifiés à tout moment.

2.3.3. Question

Sur le même principe que les cartes de jeu, les questions sont également configurables. À tout moment, il est possible d'ajouter, de supprimer ou de modifier une question.

2.3.4. Start a game

Cette fonctionnalité est la plus importante du jeu, car elle permet de lancer une partie. Chaque carte de jeu possède une fonctionnalité "start" qui crée un salon de jeu et génère un code à 5 chiffres. Ce code permet aux joueurs de se connecter au salon de jeu qui vient d'être créé.

2.3.5. Displayer & Join

Ces deux fonctionnalités sont communes aux joueurs authentifiés et non authentifiés. Leur utilité est identique à celle expliquée précédemment.

2.4. Modélisation de la base de données

La modélisation des données est une étape primordiale, car elle constitue la base sur laquelle repose presque tout le reste du projet. Elle structure de manière claire et logique les données en définissant les entités, leurs attributs et les relations entre elles. Un plan bien construit permet de réduire les risques d'erreurs liés aux éléments de code ajoutés par la suite. Dans le cadre de ce projet, un modèle relationnel a été choisi pour représenter la base de données, illustré par un diagramme Entité-Relation utilisant la "Modified Chen Notation" pour indiquer les cardinalités. Cette approche, bien qu'initialement proposée par Peter Chen en 1976, reste une méthode efficace pour modéliser les objets du monde réel et leurs interactions au sein d'une base de données. [3][18]
Pour expliquer ce modèle, la syntaxe est la suivante :

- **Rectangles** : représentent les entités. Dans le cadre de mon projet, ce sont les utilisateurs, les cartes de jeu et les questions.
- **Traits** : représentent les relations, indiquant comment les entités sont connectées.
- **Cardinalités** : donnent une idée de la quantité d'entités interconnectées. Elles sont représentées aux extrémités des relations.
- **1** : signifie exactement un
- **c** : aucun ou un
- **m** : signifie un ou plusieurs

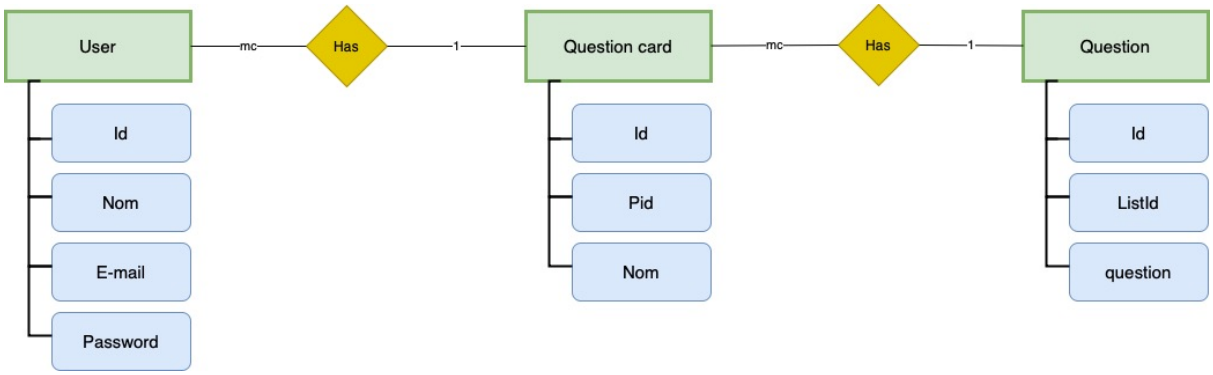

FIGURE 2.7. – Modèle entité-relation

En suivant les consignes données plus haut, nous pouvons interpréter le diagramme ERM de la manière suivante :

- Un "User" posède aucun un ou plusieurs "Question card"
- "Question card" posède aucun un ou plusieurs "Question"

Des détails supplémentaires sur la configuration de la base de données seront fournis dans la section dédiée à l'implémentation.

3

Application du point de vue utilisateur

3.1. Contexte

Dans ce chapitre, nous allons examiner l'expérience utilisateur de mon prototype. Mon application a été développée avec React[27] et Bootstrap[21], en se concentrant sur l'expérience utilisateur. Le terme "client" désigne l'interface utilisateur, qui peut être visualisée sur divers supports tels que les ordinateurs, les téléphones portables, etc. L'un des principaux avantages de Bootstrap est que le code est "responsive". Ce terme signifie que le site web s'adapte à la taille de l'écran et propose une vue différente en fonction du support utilisé.
Il y aura en grande partie deux vues distinctes pour l'application, car dans le design développé, une vue est adaptée aux grands écrans (ordinateurs ou plus grands) et une autre aux petits écrans(smartphones). Chaque sous-chapitre présentera une fonctionnalité du prototype, accompagnée d'une illustration de son utilisation.

3.2. Menu

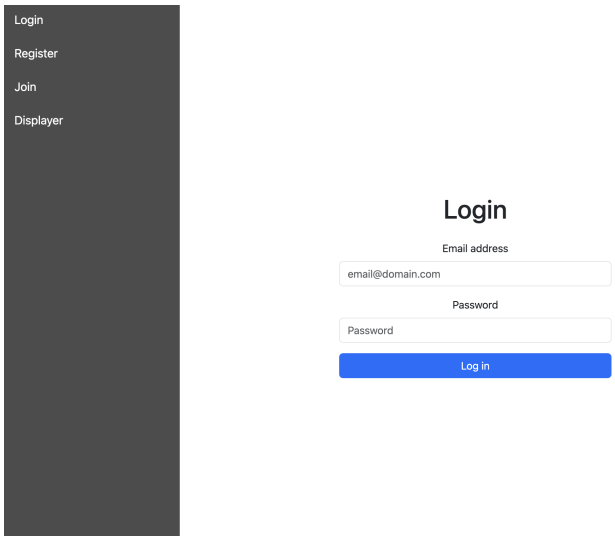

FIGURE 3.1. – Menu vue grand écran

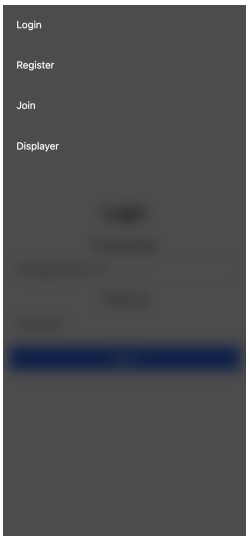

FIGURE 3.2. – Menu vue mobile

Tout d'abord, nous allons examiner le menu. Mon application comporte quatre segments. Il s'agit d'une application "SPA" (Single Page Application)[2], ce qui signifie que la vue est dynamique et change en fonction des choix effectués. Lorsque l'utilisateur n'est pas connecté, quatre segments sont disponibles : "Register", "Login", "Join" et "Displayer". En revanche, une fois connecté, il ne reste plus que trois segments : "Dashboard", "Join" et "Displayer". Toutes ces fonctionnalités sont expliquées en détail dans les sections suivantes de mon rapport.
Le menu dispose de deux vues distinctes. Pour l'activer, il suffit d'appuyer sur le bouton "Menu" situé en haut à gauche de l'écran. La figure 3.1 de gauche montre le menu sur un grand écran, où il occupe seulement une petite partie à gauche de l'écran, tandis que sur une vue mobile, comme illustré dans la figure 3.2 de droite, le menu couvre l'intégralité de l'écran.

3.3. Register

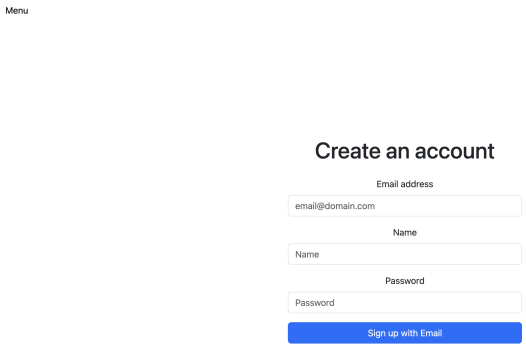

FIGURE 3.3. – Register vue grand écran

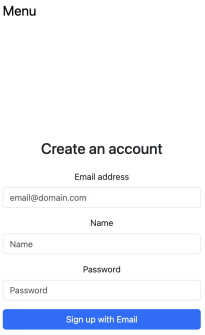

FIGURE 3.4. – Register vue mobile

Cette fonctionnalité permet la création d'un nouvel utilisateur. Mon application peut également être utilisée sans compte, mais si un joueur souhaite devenir administrateur de partie, il est obligé de créer un compte. Avoir un compte lui donne accès à des fonctionnalités supplémentaires telles que la création d'une carte de jeu (une série de questions), la modification de ses cartes de jeu existantes, et la gestion d'une bibliothèque contenant toutes ses cartes de jeu. L'aspect le plus important est la possibilité de lancer une partie. Pour s'inscrire dans le système, il suffit d'appuyer sur "Register" dans le menu, puis de remplir les trois champs requis : l'email, le nom, et le mot de passe. Il est important de fournir une adresse email valide, car le système vérifie que la chaîne de caractères correspond bien au format d'une adresse email. Une fois les informations saisies, il suffit d'appuyer sur le bouton "Sign up" pour finaliser l'inscription.

3.4. Login

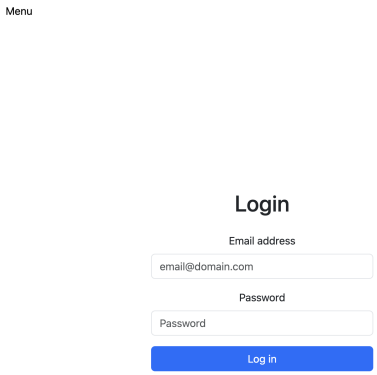

FIGURE 3.5. – Login vue grand écran

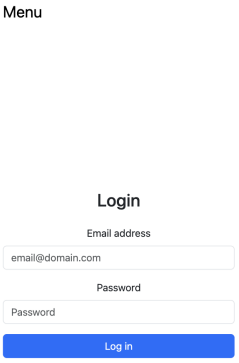

FIGURE 3.6. – Login vue mobile

Après avoir créé un compte, le système conserve les données de l'utilisateur, permettant ainsi au joueur de les utiliser pour se connecter ultérieurement. Cette connexion se fait simplement en insérant l'email et le mot de passe choisis lors de l'inscription. Les deux vues correspondantes sont illustrées dans les figures de ce chapitre.

3.5. Dashboard

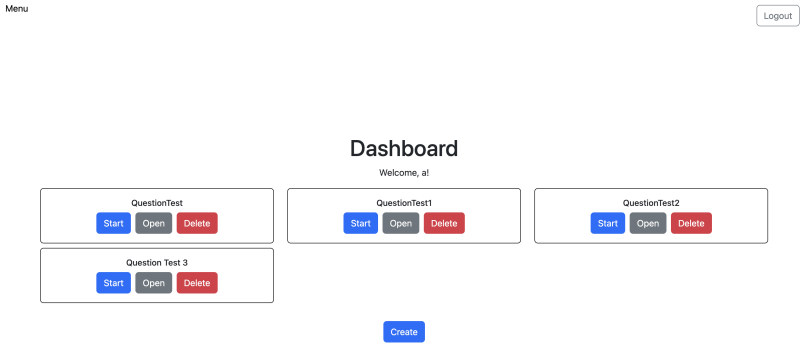

FIGURE 3.7. – Dashboard vue grand écran

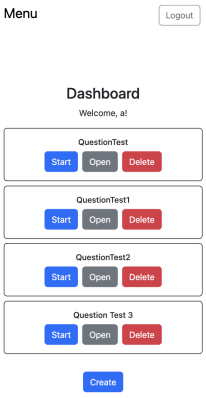

FIGURE 3.8. – Dashboard vue mobile

Dans le Dashboard, on trouve toutes les cartes de jeu existantes. Sur la vue grand écran, comme illustré sur la figure 3.7 de gauche, les cartes sont alignées en colonnes de trois, centrées au milieu de la page. Cette structure est conservée, même lorsqu'il n'y a qu'une seule carte. En revanche, sur la vue mobile, comme montré sur la figure 3.8 de droite, les

cartes sont alignées en une seule colonne, pour s'adapter à la taille de l'écran.
Chaque carte comporte trois boutons. Le bouton "Start", en bleu, permet de lancer une partie. Le bouton "Open/Close", en gris, permet d'afficher ou de masquer les questions associées à la carte. Le bouton "Delete", en rouge, supprime la carte. En bas de la page, un bouton "Create", en bleu, centré, permet de créer une nouvelle carte. Enfin, on retrouve en haut à gauche le bouton "Menu", ainsi qu'un bouton "Logout" pour se déconnecter.

3.6. Create a game card

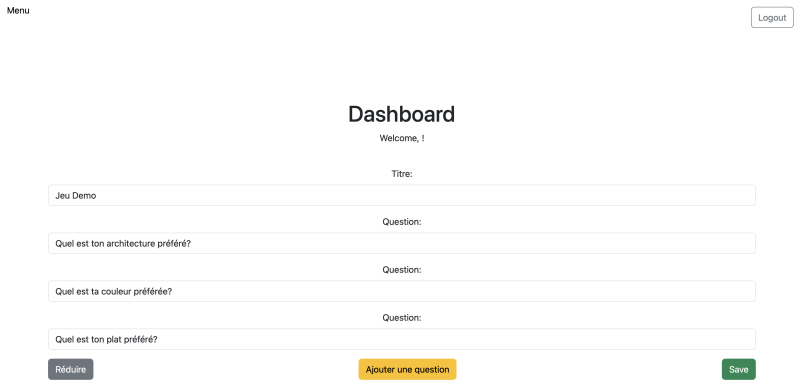

FIGURE 3.9. – Vue de création de question

Lorsque l'on appuie sur le bouton "Create" pour créer une nouvelle carte de jeu, la vue présentée ressemble à celle illustrée sur la figure 3.9. L'utilisateur peut entrer un titre, comme "JeuDemo" dans l'exemple, mais ce titre est entièrement personnalisable. En appuyant sur le bouton "Ajouter une question", le système ajoute un champ permettant d'insérer une nouvelle question. Une fois toutes les questions ajoutées, l'utilisateur peut appuyer sur le bouton "Save", en vert, pour sauvegarder la carte de jeu. Il y a également un bouton "Réduire" qui permet de masquer cette section de création.

3.7. Configuration

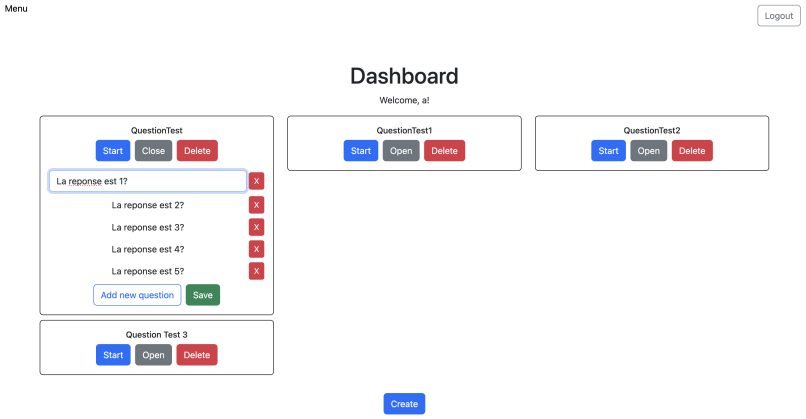

FIGURE 3.10. – Configurartion vue grand écran

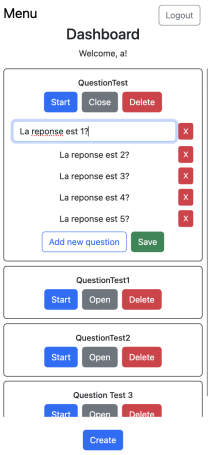

FIGURE 3.11. – Configuration vue mobile

Dans la section "Dashboard", il est également possible d'éditer les cartes de jeu déjà existantes. Il suffit de cliquer sur les champs de texte pour les modifier, ou d'appuyer sur le bouton "Delete/X" en rouge pour supprimer complètement l'élément. Il est recommandé d'appuyer sur "Save" après avoir effectué les modifications, bien que la plupart des changements soient pris en compte dès que le champ de modification n'est plus actif.
Lorsque l'utilisateur clique sur un champ contenant une question ou sur le titre du jeu, celui-ci devient un champ éditable entouré en bleu, comme illustré sur les deux figures de l'exemple. Une fois les modifications effectuées, le système met à jour la base de données avec les changements nécessaires.

3.8. Start

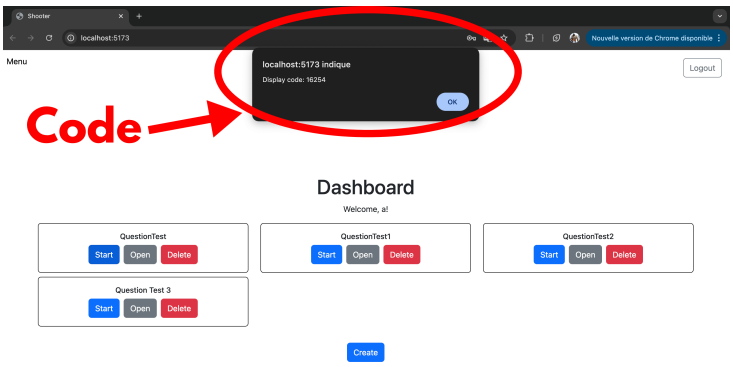

FIGURE 3.12. – Start vue grand écran

La fonctionnalité la plus utile de l'application est la possibilité de démarrer une session de jeu. Cette action s'effectue via le bouton "Start" présent sur chaque carte de jeu. Lorsque ce bouton est activé, le serveur crée une session en utilisant les questions associées à la carte de jeu sélectionnée et génère un code à 5 chiffres. Ce code s'affiche sous forme d'alerte dans le navigateur avec le message "Display code : *****", comme illustré à la figure 3.12. Ce code est ensuite utilisé pour se connecter à la session de jeu et participer.

3.9. Displayer

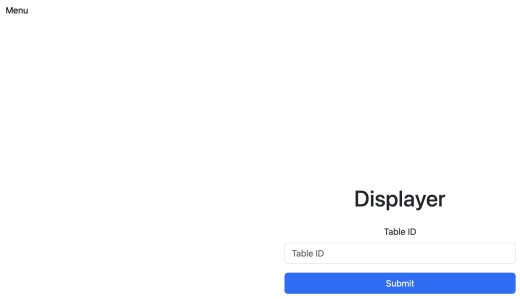

FIGURE 3.13. – Displayer non connecté

Une fois que le code à cinq chiffres servant d'identifiant de session est obtenu, comme illustré à la figure 3.12, il est nécessaire de créer une interface "Displayer" pour permettre l'affichage du jeu. Pour ce faire, il suffit de cliquer sur la section "Displayer" dans le menu, puis de saisir le code dans le formulaire prévu à cet effet. En cliquant sur "Submit", le serveur vérifie automatiquement l'existence de la session correspondante et s'y connecte si celle-ci est valide.

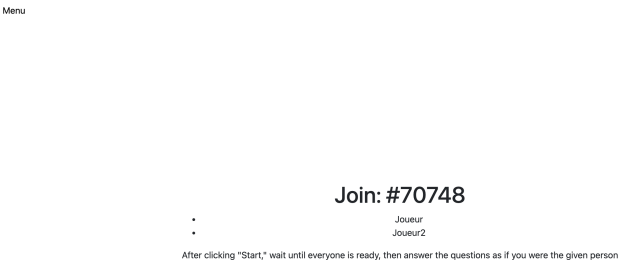

FIGURE 3.14. – Displayer connecté

Si la session est valide, l'affichage changera, comme montré sur la figure 3.14. L'interface affichera le même code que les joueurs utiliseront pour se connecter à la session. La logique derrière cette fonctionnalité repose sur la nécessité d'avoir un ou plusieurs "Displayer(s)" pour agir comme maître de jeu, ainsi un minimum de trois à quatre joueurs est recommandé pour démarrer une partie. Le "Displayer" est généralement affiché sur un grand écran visible par tous, ce qui facilite la connexion des autres joueurs, qui peuvent suivre les instructions visibles sur cet écran principal.
De plus, les joueurs connectés sont affichés dynamiquement avec leur nom, comme illustré sur la seconde figure. Chaque fois qu'un nouveau joueur rejoint la session, son nom est automatiquement mis à jour et visible sur l'interface. Cela permet de suivre en temps réel qui est connecté à la session de jeu.

3.10. Join

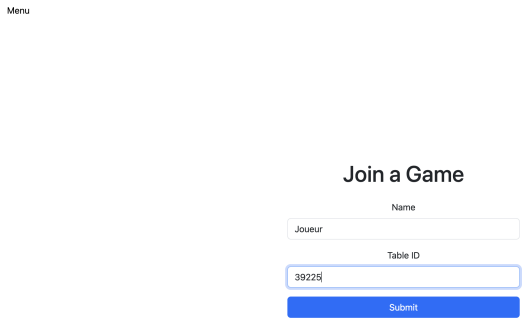

FIGURE 3.15. – Join vue grand écran

FIGURE 3.16. – Join vue mobile connecté

Si le "Displayer" a déjà été activé, les joueurs peuvent se connecter en accédant à la section "Join" dans le menu. Lorsqu'ils accèdent à la vue "Join", un formulaire s'affiche, demandant un nom et un code. Le joueur doit saisir le nom qu'il souhaite utiliser, visible par les autres participants, ainsi que le code affiché par le "Displayer". Une fois ces informations renseignées, il peut appuyer sur le bouton "Submit" pour rejoindre la session de jeu, comme montré dans l'exemple de la figure 3.15.
Lorsque le code est valide, la vue change et le joueur est connecté, comme illustré dans l'exemple de la figure 3.16. À ce moment, il peut voir tous les joueurs connectés à la session, accompagnés des noms qu'ils ont choisis. Un bouton "Start" avec une bordure verte et initialement transparent apparaît également. Chaque joueur doit appuyer sur ce bouton "Start" lorsque tout le monde est prêt à jouer. Le système synchronise tous les boutons "Start". Une fois que tous les participants ont appuyé sur ce bouton, le jeu commence.
Pour savoir si un joueur a déjà appuyé sur le bouton, il suffit de vérifier sa couleur : une fois cliqué, le bouton devient entièrement vert. Il est également possible de cliquer une seconde fois sur le bouton pour indiquer que le joueur n'est plus prêt, ce qui ramènera le bouton à son état initial.

3.11. Question

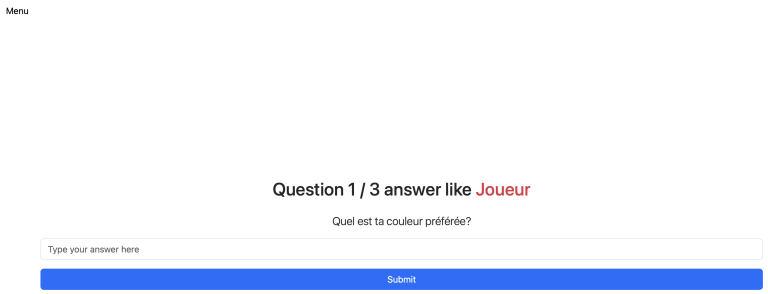

FIGURE 3.17. – Question vue grand écran

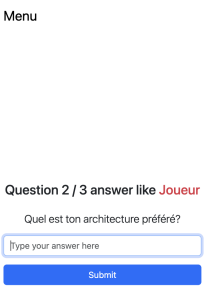

FIGURE 3.18. – Question vue mobile connecté

Le jeu commence par une série de questions aléatoires posées à chaque joueur, avec un total de trois questions par joueur. Chaque joueur peut recevoir soit la même question, soit une question différente. De plus, il est demandé de répondre à la question comme le ferait un autre joueur.
Pour mieux comprendre, on peut se référer à l'exemple de la figure 3.17. Le joueur doit répondre à la question "Quelle est ta couleur préférée ?" en se mettant à la place de la personne désignée, ici "Joueur". Ainsi, le joueur qui est en train de jouer doit imaginer comment "Joueur" répondrait à cette question. Ce processus se répète trois fois, à chaque fois avec une nouvelle question aléatoire et un autre joueur aléatoire.
Dans les figures de ce chapitre, nous voyons le même exemple sur deux clients différents. Pour le premier joueur, il s'agissait de la première question, tandis que pour le second, c'était la deuxième question. La personne pour laquelle il faut répondre est bien mise en évidence en rouge, comme montré sur les figures. Une fois que la réponse est écrite dans le champ prévu à cet effet, le joueur peut appuyer sur "Submit" pour valider sa réponse.

3.12. Response

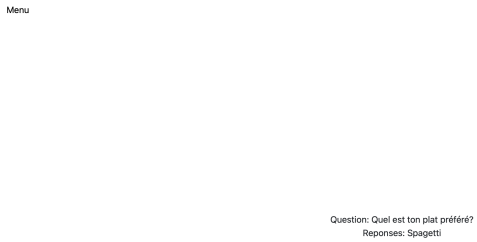

FIGURE 3.19. – Displayer réponse

Une fois que tous les joueurs ont répondu aux questions, le jeu passe à la phase de quiz. Durant cette phase, les questions/réponses fournies par les joueurs sont affichées sur le "Displayer". Par exemple, dans la figure 3.19, la question affichée est "Quel est ton plat préféré ?" et la réponse donnée est "Spagetti".
Chaque joueur doit alors deviner qui a répondu et pour qui. Autrement dit, ils doivent identifier quel joueur a répondu à cette question en se mettant à la place de quel autre joueur. Ce processus se répète avec les différentes réponses fournies par les joueurs au cours du jeu, rendant le quiz à la fois interactif et amusant, car il s'agit de deviner les dynamiques de réponse entre les participants.

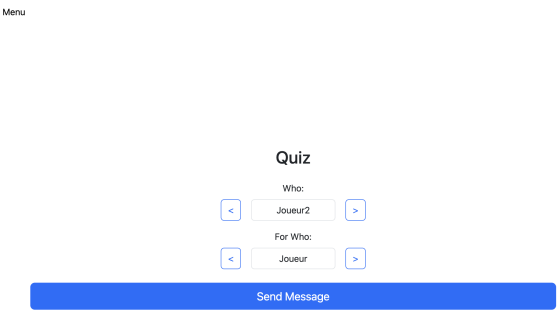

FIGURE 3.20. – Réponse vue grand écran

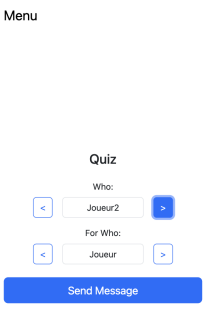

FIGURE 3.21. – Réponse vue mobile

Pour participer à la phase de quiz, chaque joueur accède à l'interface "Quiz", comme illustré dans les figures 3.20 et 3.21. Sur cette interface, deux menus déroulants sont disponibles : le premier permet de sélectionner le joueur qui a posé la question, et le second permet de choisir pour qui ce joueur a répondu. Chaque menu affiche les noms de

tous les joueurs connectés à la session, et les choix peuvent être modifiés en utilisant les flèches situées à droite et à gauche de chaque menu.

Une fois que le joueur a fait ses choix pour répondre à la question et à la réponse affichée sur le "Displayer", il peut appuyer sur le bouton "Send message" pour soumettre sa réponse. Ce système permet à chaque joueur de voter et de tenter de deviner les relations entre les réponses et les joueurs impliqués.

3.13. Wait

FIGURE 3.22. – Wait vue grand écran

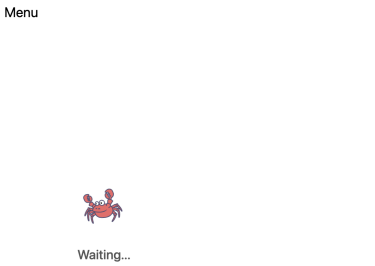

FIGURE 3.23. – Wait vue mobile connecté

Durant chaque round, lorsque qu'un joueur attend que tous les autres aient répondu, une petite animation d'attente est affichée. Celle-ci met en scène un animal aléatoire qui oscille de chaque côté, rendant l'attente plus agréable à regarder. Au total, douze animaux différents sont utilisés dans cette animation, chacun sélectionné de manière aléatoire à chaque fois. Ces animations ont été téléchargées gratuitement au format SVG, permettant une intégration légère et fluide dans l'interface du jeu.[23]

3.14. Score

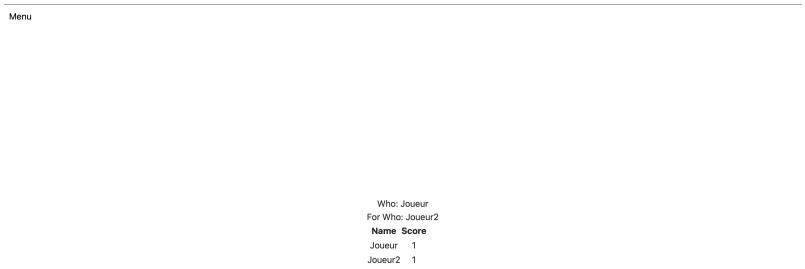

FIGURE 3.24. – Vue de Score

À la fin de chaque round, les scores des différents joueurs sont affichés sur le "Displayer". Chaque joueur peut gagner un point par bonne réponse, ce qui signifie qu'il est possible de gagner jusqu'à deux points par round (un point pour avoir correctement deviné qui a répondu à la question et un point pour avoir deviné pour qui la réponse a été donnée). Cette mécanique permet de comptabiliser les performances des joueurs tout au long du jeu, créant une compétition amusante et dynamique.

La fin du jeu survient lorsque toutes les questions ont été répondues. Les scores et la réponse à la dernière question sont alors affichés une dernière fois, puis tous les clients connectés à cette session de jeu sont automatiquement redirigés vers la page d'accueil de l'application.

4

Application du point de vue développeur

4.1. Contexte

Pour commencer, l'objectif de ce chapitre est de présenter l'organisation de mon code. Mon application est composée de trois parties distinctes : un front-end en React et deux back-ends en NodeJs et Go. Cette architecture a été choisie afin de tirer parti des points forts de chaque technologie. Ainsi, j'ai opté pour une architecture distribuée, basée sur des microservices.

Le front-end, développé en React, est écrit en JSX[10]. Le back-end en NodeJs utilise ExpressJS[12] pour gérer l'authentification et la base de données, tandis que le serveur en Go, avec le framework Gorilla[6], agit comme un microservice chargé de la gestion dynamique des sessions de jeu. Cette organisation permet une meilleure répartition des responsabilités et une utilisation optimisée des différentes technologies.

Du côté de la base de données, j'ai implémenté SQLite [5] pour des raisons de prototypage et d'efficacité. SQLite offre une solution légère et simple à mettre en place, tout en étant facilement adaptable à un serveur SQL plus robuste pour une mise en production. Cette approche me permet de valider rapidement les fonctionnalités de l'application tout en conservant la possibilité de migrer vers une solution plus évolutive si nécessaire.

Sur la figure 4.1, je vais détailler le fonctionnement de mon prototype en lien avec l'architecture de la base de données et son intégration dans l'application.

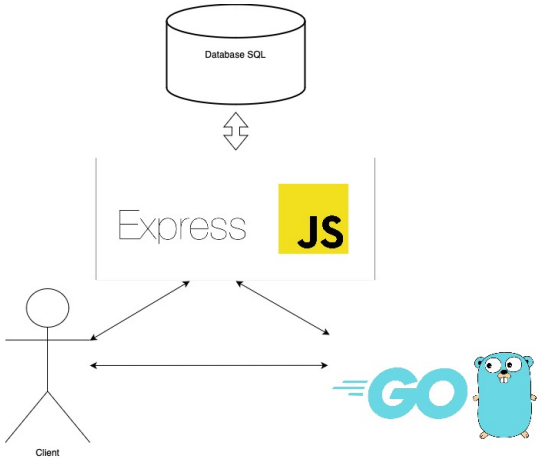

FIGURE 4.1. – Architecture prototype

4.2. Front-end

4.2.1. Analyse des frameworks front-end

Le choix d'un framework front-end est une décision cruciale dans le cadre du développement d'un projet web. De nombreux frameworks sont disponibles, chacun présentant des avantages et des inconvénients selon plusieurs critères : facilité d'utilisation, performances, qualité de la documentation, et popularité. Cette analyse repose à la fois sur des études réalisées par la communauté en ligne [1] et sur des données provenant d'évaluations comparatives de frameworks [25].

Critères d'analyse

Facilité d'utilisation L'ergonomie et la facilité de prise en main d'un framework constituent un aspect essentiel, particulièrement dans les environnements où le temps de développement est limité. Selon les données recueillies du tableau 4.1 [25], React et Vue.js sont souvent perçus comme des frameworks relativement simples à utiliser. Vue.js est largement plébiscité pour sa courbe d'apprentissage douce, tandis que React, bien que nécessitant la compréhension de concepts comme JSX et les hooks, reste accessible aux développeurs familiarisés avec JavaScript. Angular, en revanche, est considéré comme plus complexe en raison de sa structure rigide et d'un écosystème plus lourd, ce qui allonge le temps d'apprentissage.

Performances Les performances sont un critère majeur, en particulier pour les applications à grande échelle. Toujours selon l'analyse comparative du tableau 4.1 [25], React optimise les performances en utilisant le *Virtual DOM*, un mécanisme qui permet de ne mettre à jour que les parties nécessaires du DOM réel, rendant les interfaces plus réactives. Svelte adopte une approche différente en compilant le code au moment du build,

éliminant ainsi le besoin d'un *Virtual DOM*, ce qui lui confère un avantage supplémentaire en termes de performances. Vue.js, bien qu'offrant des performances respectables, se situe légèrement en dessous de React et Svelte dans des cas d'utilisation plus complexes. Angular, avec sa structure plus lourde, peut voir ses performances diminuer dans des applications de grande envergure.

Documentation La qualité de la documentation influence directement la productivité des développeurs. React et Angular se distinguent par une documentation très riche et exhaustive. Soutenu par Facebook, React dispose d'une documentation bien structurée ainsi que d'une communauté dynamique offrant une multitude de ressources supplémentaires. Vue.js propose également une documentation claire et bien organisée, bien que légèrement plus limitée en termes de ressources comparé à React. Svelte, malgré ses performances remarquables, souffre d'une documentation plus réduite, ce qui peut représenter un obstacle pour les développeurs moins expérimentés. Cette tendance est confirmée par les données [25], où React et Angular dominent largement en matière de documentation.

Popularité et communauté La popularité d'un framework est un indicateur de sa pérennité et de la taille de son écosystème. Selon les statistiques récentes représenté sur le graphique de la figure 4.2 [1], React est de loin le framework front-end le plus utilisé, adopté par des entreprises de premier plan comme Facebook, Netflix et Airbnb. Cette popularité se traduit par une communauté très active, un support constant et un écosystème en pleine expansion. Vue.js, bien qu'il ait une communauté plus modeste, gagne en popularité, particulièrement pour des projets de taille moyenne. Angular, bien que toujours présent dans des environnements institutionnels, voit sa popularité décliner en raison de frameworks plus légers et flexibles. Quant à Svelte, il est en pleine croissance mais reste un choix moins courant pour l'instant, selon les tendances du dépôt [25].

Comparaison des frameworks

L'analyse des données issues des études [1, 25] permet d'établir un panorama clair des forces et faiblesses de chaque framework :

- **React** : Il se distingue par sa popularité, son vaste écosystème et ses bonnes performances grâce à l'utilisation du *Virtual DOM*. Il reste relativement simple à utiliser pour ceux qui maîtrisent JSX et les hooks. Sa documentation est complète et la communauté qui le soutient est très active.
- **Vue.js** : Reconnu pour sa simplicité et sa courbe d'apprentissage progressive, Vue.js est une excellente option pour des projets de taille moyenne. Bien que légèrement moins performant que React et Svelte dans des scénarios plus complexes, il reste un excellent choix pour ceux qui recherchent un framework rapide à mettre en œuvre.
- **Angular** : Ce framework est plus complexe et présente une structure rigide, ce qui peut être un désavantage dans certains cas. Cependant, il est toujours très utilisé dans les grandes organisations, en particulier pour des applications d'entreprise de grande envergure.
- **Svelte** : Innovant dans le domaine des performances, Svelte offre des résultats impressionnants grâce à sa compilation directe. Cependant, son écosystème et sa

documentation encore limités peuvent freiner son adoption à grande échelle pour l'instant.

Conclusion

Après une analyse approfondie de ces différents critères, React semble être le choix le plus approprié pour ce projet. Sa combinaison de performances élevées, d'une documentation solide et d'une communauté active en fait un outil particulièrement adapté au développement d'applications web modernes et évolutives. Vue.js peut également être une alternative intéressante pour des projets de plus petite envergure ou nécessitant une mise en place rapide. Cependant, pour assurer la scalabilité et la durabilité du projet, React se révèle être la solution la plus robuste. De plus, il est à noter que React propose également React Native, un framework permettant de développer des applications mobiles natives. Par conséquent, l'apprentissage de React constitue un choix stratégique optimal dans le cadre d'un travail de bachelor.

TABLE 4.1. – Comparaison des frameworks JavaScript

Framework	Ease of Use	Performance	Compatibility	Documentation
React	High	High	High	Extensive
Angular	Moderate	High	Moderate	Comprehensive
Vue.js	High	High	High	Extensive
Ember.js	Moderate	Moderate	Moderate	Comprehensive
jQuery	High	Moderate	High	Extensive

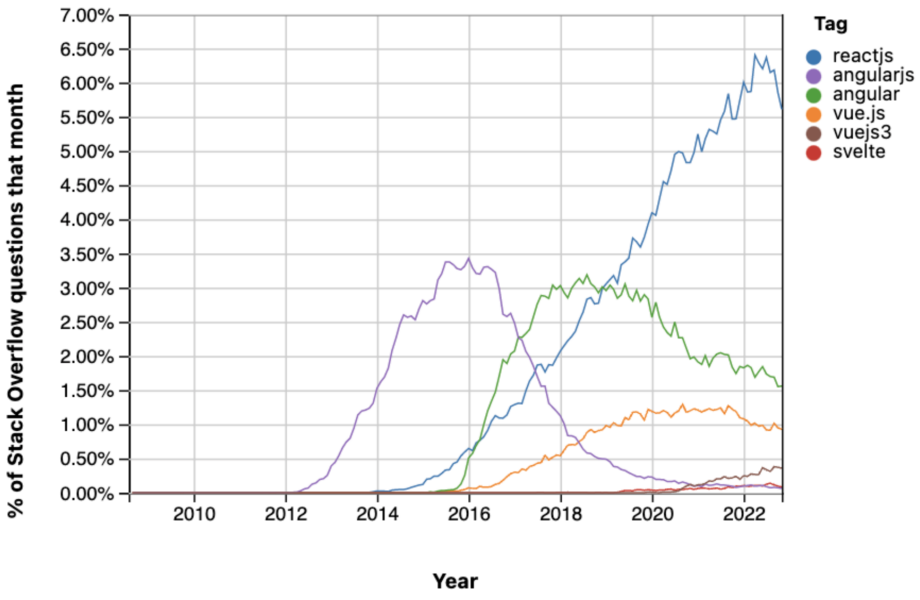

FIGURE 4.2. – Anlayse framework web : graphique popularité

4.2.2. React

React [19] est une bibliothèque JavaScript open source, développée par *Meta Open Source*, qui offre des outils puissants pour créer des interfaces utilisateur interactives et dynamiques. Depuis son introduction en 2013, elle s'est imposée comme une technologie de référence pour le développement d'applications web modernes. Son principal avantage réside dans sa capacité à structurer les interfaces sous forme de composants réutilisables, facilitant ainsi la gestion et l'évolution des interfaces complexes.

L'un des concepts fondamentaux de React est son approche déclarative, où les développeurs définissent l'apparence et le comportement de l'interface à un moment donné. React se charge ensuite de mettre à jour les éléments concernés lorsque l'état change. Cette gestion est optimisée grâce à l'utilisation du *Virtual DOM* (Document Object Model virtuel), qui permet d'améliorer les performances en réduisant les manipulations directes du DOM réel.

Le DOM

Le DOM (Document Object Model) est une interface de programmation qui représente la structure d'un document HTML ou XML sous la forme d'un arbre hiérarchique. Chaque élément d'une page web, comme un titre, un paragraphe ou une image, est représenté sous forme de nœud dans cet arbre. Le DOM permet aux langages de programmation, comme JavaScript, de manipuler dynamiquement les éléments d'une page, en ajoutant, modifiant ou supprimant des nœuds. Cependant, les manipulations directes du DOM peuvent être coûteuses en termes de performances, surtout lorsqu'il s'agit de grandes applications web. C'est ici que React se distingue en introduisant le *Virtual DOM*, une représentation en mémoire du DOM réel, qui optimise les mises à jour en ne modifiant que les éléments nécessaires.

Philosophie et design

L'architecture de React est basée sur un *design pattern* orienté autour des composants modulaires. Chaque composant représente une unité fonctionnelle de l'interface utilisateur, telle qu'un bouton, un formulaire ou une section de la page. Ces composants sont réutilisables et peuvent être agencés dans différentes parties de l'application, ce qui simplifie la modularité du code et en facilite la maintenance. Cette approche permet de diviser une application complexe en plusieurs composants plus petits, chacun gérant son propre état et son propre rendu. Cela rend l'interface plus facile à maintenir et à développer, tout en améliorant la lisibilité du code.

En parallèle, React adopte un modèle de gestion unidirectionnelle des données, appelé *one-way data binding*. Ce modèle garantit que les données circulent du composant parent vers les enfants via les *props* (propriétés). Cela assure une meilleure prévisibilité des flux de données, facilitant le contrôle et la gestion des interactions complexes.

De plus, React utilise le concept de *state* pour gérer l'état interne des composants. Lorsqu'un état change, React évalue uniquement le composant concerné et met à jour l'interface de manière efficace grâce au *Virtual DOM*. Ce mécanisme améliore les performances en réduisant le besoin de rechargements complets de la page, offrant ainsi une expérience utilisateur plus fluide.

SPA

Une autre caractéristique essentielle de React est sa capacité à construire des applications monopages, ou *Single Page Applications* (SPA). Dans une SPA, tout le contenu est chargé dans une seule page HTML initiale, et les vues de l'application sont actualisées de manière dynamique sans nécessiter de rechargement complet de la page. Ce processus est géré efficacement par le *Virtual DOM*, qui ne met à jour que les parties modifiées de l'interface, garantissant ainsi des transitions rapides et fluides.

Grâce à ce modèle, les utilisateurs peuvent naviguer entre les différentes sections de l'application sans interruptions visibles, ce qui améliore considérablement l'efficacité et réduit les temps de chargement. Ce type d'architecture permet de créer des applications web proches des applications natives, tout en bénéficiant de la rapidité et de la légèreté des technologies web.

4.2.3. Mise en place

Dans cette section, nous allons procéder à la mise en place du projet et à l'installation des dépendances nécessaires pour le développement. Pour ce faire, nous utiliserons Node.js [11], React, et Vite [28], un outil de construction web moderne qui remplace Webpack pour offrir des temps de démarrage plus rapides et une meilleure expérience de développement. L'objectif est de créer un environnement de développement performant et bien structuré pour construire une application web réactive avec React.

Pré-requis Avant de commencer, assurez-vous d'avoir Node.js installé sur votre machine. Si ce n'est pas le cas, vous pouvez le télécharger depuis le site officiel [11], puis vérifier l'installation avec la commande suivante dans le terminal :

```
node -v
npm -v
```

Ces commandes permettent de vérifier que Node.js et NPM (Node Package Manager) sont bien installés en affichant les versions correspondantes installées sur votre machine.

Mise en place du projet L'étape suivante consiste à initialiser le projet et à installer les dépendances nécessaires. Nous allons utiliser Vite pour sa rapidité et son efficacité dans la gestion des projets React. Voici les étapes détaillées :

1. **Initialisation du projet avec Vite** Pour créer un nouveau projet React avec Vite, ouvrez votre terminal et exécutez la commande suivante pour initialiser un nouveau projet :

```
npm create vite@latest nom-du-projet -- --template react
```

Cette commande utilise 'vite' pour créer un nouveau projet React. Remplacez 'nom-du-projet' par le nom de votre choix pour le projet. Le paramètre '–template react' indique que nous souhaitons initialiser un projet avec le modèle React fourni par Vite.

2. Installation des dépendances Une fois le projet créé, déplacez-vous dans le répertoire du projet avec la commande :

```
cd nom-du-projet
```

Ensuite, installez toutes les dépendances du projet en exécutant :

```
npm install
```

Cette commande installera toutes les bibliothèques nécessaires pour faire fonctionner le projet React. Les dépendances seront listées dans le fichier 'package.json', qui inclut des bibliothèques comme React, React-DOM et Vite, entre autres.

3. Lancement du serveur de développement Maintenant que toutes les dépendances sont installées, vous pouvez lancer le serveur de développement en utilisant la commande suivante :

```
npm run dev
```

Cela démarrera un serveur de développement à l'aide de Vite, et vous fournira une URL (généralement 'http ://localhost :3000') où vous pourrez visualiser votre application en temps réel dans un navigateur.

Vérification du bon fonctionnement Une fois toutes ces étapes réalisées, vous pouvez vérifier que votre projet fonctionne correctement en accédant à 'http ://localhost :3000' dans votre navigateur. Vous devriez voir l'application React de base fournie par Vite.

À partir de là, vous êtes prêt à commencer le développement de votre application en ajoutant vos propres composants et fonctionnalités.

4.2.4. Code

Dans cette section, nous allons analyser la structure de mon code ainsi que son fonctionnement. Chaque sous-section contiendra des schémas et des blocs de code associés pour illustrer les concepts présentés. Les schémas représenteront la hiérarchie des composants dans React, en mettant en lumière les relations parent-enfant entre eux.

Les titres des sous-sections seront corrélés au nom des composants qu'ils représentent, facilitant ainsi la compréhension et la navigation dans la structure du projet. Seules les parties nécessaires à la compréhension de l'architecture et du fonctionnement du code seront présentées dans ce rapport. Le code source complet, ainsi que les détails supplémentaires, sont disponibles sur Github [1] pour ceux qui souhaitent l'examiner dans son intégralité.

[1] https ://github.com/agkkaya321/unifrTB

App

Le composant `App` est le composant racine de mon prototype. Son rôle principal est de gérer l'état de l'application afin d'afficher l'un des six composants enfants, en fonction des interactions de l'utilisateur. Ces composants sont les suivants :

1. **DropdownMenu** : Ce composant représente le menu déroulant permettant à l'utilisateur de naviguer entre les différentes vues de l'application. La gestion de l'état de la vue est assurée par le hook `useState`, avec deux variables : `currentPage` et `setCurrentPage`. Comme illustré à la ligne 3 du code 4.1, ces variables sont passées en tant qu'arguments au composant `DropdownMenu`. Une variable locale dans le composant `App` contrôle l'affichage de la vue active, et la fonction `setCurrentPage` est transmise aux sous-composants pour leur permettre de modifier cet état. À chaque changement d'état, le DOM est re-rendu pour refléter la nouvelle vue. L'évaluation des vues à afficher se fait via une série de conditions `if`, visibles aux lignes 9, 12, 13, 17, 20, et 21 du code.
2. **Dashboard** : Ce composant est responsable de l'affichage du tableau de bord, qui regroupe les informations et actions principales de l'utilisateur connecté. Il contient également deux paramètres, qui seront détaillés dans le chapitre dédié aux hooks.
3. **Login** : Ce composant gère le formulaire de connexion, permettant à l'utilisateur de s'authentifier.
4. **Auth** : Ce composant est chargé de l'enregistrement des utilisateurs et redirige vers la vue appropriée en fonction de leur état (connecté ou non).
5. **Jeu** : Ce composant représente l'interface principale du jeu, affichant les éléments interactifs pour les joueurs.
6. **Displayer** : Ce composant gère l'affichage du mode *Displayer* de l'application, qui présente des informations aux utilisateurs dans un contexte spécifique.

Le composant `App` assure donc une navigation dynamique entre ces différentes vues, en fonction de l'état global de l'application. Cet état est principalement géré à travers le hook `useState`, qui permet de déterminer quel composant doit être affiché à un moment donné.

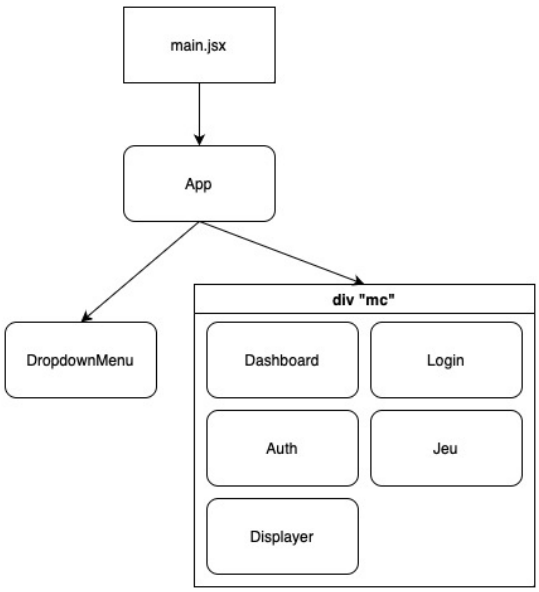

FIGURE 4.3. – Relation components

```jsx
<>
  <DropdownMenu
    setCurrentPage={setCurrentPage}
    isAuthenticated={isAuthenticated}
  />
  <div className="mc">
    {isAuthenticated ? (
      <>
        {currentPage === "dashboard" && (
          <Dashboard user={user} handleLogout={handleLogout} />
        )}
        {currentPage === "page3" && <Jeu />}
        {currentPage === "displayer" && <Displayer />}
      </>
    ) : (
      <>
        {currentPage === "page1" && (
          <Login onLogin={handleLogin} login={login} error={error} />
        )}
        {currentPage === "page2" && <Auth register={register} />}
        {currentPage === "page3" && <Jeu />}
        {currentPage === "displayer" && <Displayer />}
      </>
    )}
  </div>
</>
```

Listing 4.1 – Return of App.jsx

DropdownMenu

Le composant `DropdownMenu` est un menu permettant de naviguer entre les différentes vues de l'application. Il est illustré dans les figures 3.1 et 3.2. Son fonctionnement repose

sur un principe simple. Comme on peut le voir à la ligne 1 du code 4.2, une fonction `setCurrentPage` est passée en paramètre.

De la ligne 5 à la ligne 13, nous voyons un composant `<li>` qui représente le segment *Login* du menu. À la ligne 7 et 9, on peut observer qu'au clic sur cet élément, la fonction `setCurrentPage("page1")` est appelée. Cela a pour conséquence de modifier la variable `currentPage` dans le composant principal `App`. Ce même mécanisme est utilisé pour naviguer vers les autres composants.

Le tout repose sur un mécanisme de gestion de l'état : chaque changement de la variable `currentPage` provoque une mise à jour du DOM virtuel, qui entraîne le rendu de la vue correspondante.

```
1  const DropdownMenu = ({ setCurrentPage, isAuthenticated }) => {
2    ...
3    return (
4      ...
5      <li
6        className="ddi"
7        onClick={() => {
8          toggleDropdown();
9          setCurrentPage("page1");
10       }}
11     >
12       Login
13     </li>
14     ...
15   );
16 };
```

Listing 4.2 – DropdownMenu.jsx

Dashboard

Le composant **Dashboard** est responsable de l'affichage principal de l'application, comme illustré dans les figures 3.7 et 3.8. Il comprend deux sous-composants principaux : **QCard** et **CreateCQ**. De plus, **QCard** a également un sous-composant appelé **Question**.

Le **Dashboard** utilise un hook personnalisé appelé `useQ` pour récupérer les données des cartes de jeu. Ensuite, il utilise le composant `QCard` pour afficher ces données et permettre à l'utilisateur d'interagir avec elles. L'affichage lui-même est relativement simple, puisqu'il s'agit principalement de HTML et de JSX de base, mais l'intérêt se trouve dans la manière dont les données sont traitées avec les hooks.

Comme on peut le voir à la ligne 2 du code 4.3, le **Dashboard** fait appel à `useQ` pour obtenir les données des cartes de jeu. Ces données sont ensuite traitées ligne par ligne, notamment aux lignes 6 et 8, où chaque carte est affichée séparément grâce au composant `QCard`. Chaque carte de jeu est ainsi traitée comme une instance distincte. Le sous-composant `Question` fonctionne de la même manière pour gérer l'affichage des questions à l'intérieur de chaque carte.

Le composant `CreateCQ` fonctionne sur un principe similaire, mais avec un objectif différent : permettre la création de nouvelles cartes de jeu et de nouvelles questions, ajoutant ainsi des fonctionnalités dynamiques à l'application.

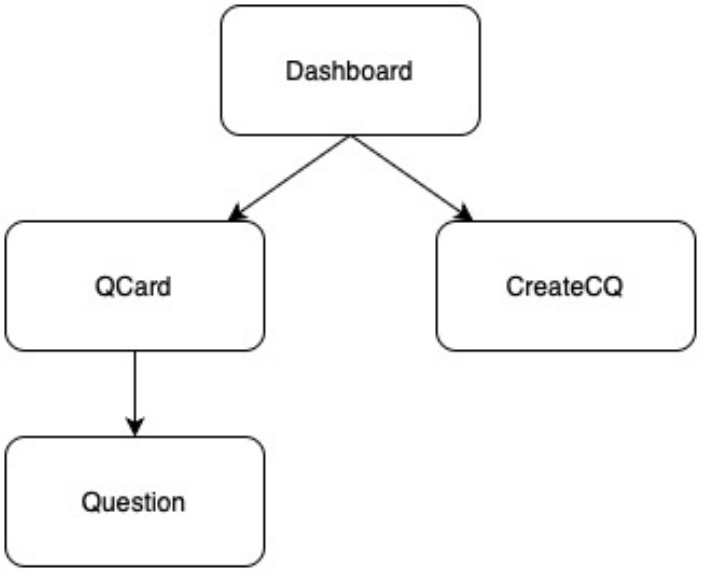

FIGURE 4.4. – Relation components Dashborad

```
1  const Dashboard = ({ user, handleLogout }) => {
2    const { data, error, handleRefresh } = useQ();
3
4    return (
5      <div className="row">
6        {data.map((qcard, index) => (
7          <div className="col-md-4" key={qcard.id || index}>
8            <QCard qcard={qcard} refresh={handleRefresh} />
9          </div>
10       ))}
11     </div>
12   );
13 };
```

Listing 4.3 – Dashboard.jsx

Login

Pour commencer l'analyse du code présenté à la ligne 1 du code 4.4, nous remarquons que plusieurs paramètres sont passés en arguments lors de l'appel du composant `Login`. Ces paramètres sont essentiels pour gérer les différentes fonctionnalités de la page de connexion. À la ligne 2 et 3, on observe l'initialisation d'un hook `useState` qui va stocker les informations saisies par l'utilisateur, comme l'email et le mot de passe. Ce hook permet de conserver et de mettre à jour ses données au fur et à mesure que l'utilisateur les entre.

Aux lignes 27 et 37, on peut voir comment les champs `email` et `password` sont mise à jour à chaque modification par l'utilisateur. Cela est géré en temps réel grâce au hook `useState`, qui écoute les changements dans les champs de saisie. Le formulaire contient deux champs de type `input` pour l'email et le mot de passe, ainsi qu'un bouton `submit`, ce qui correspond à la structure classique d'une page de connexion.

Lorsqu'un utilisateur saisit des informations dans les champs, ces données sont stockées dans les variables gérées par le `useState`, ce qui permet de récupérer ces informations lors

de la soumission du formulaire. Lorsque le bouton `submit` est cliqué, la première étape est de vérifier que les champs ne sont pas vides et que les informations sont correctement formatées. Ensuite, la fonction `login` est déclenchée. Cette fonction fait partie du hook `useAuth`, comme illustré dans le code 4.5.

Dans le code 4.5, à la ligne 1, on peut observer que la fonction `login` prend en paramètres l'email et le mot de passe. Elle envoie ensuite une requête POST à l'aide de la bibliothèque `axios` pour authentifier l'utilisateur. Si la réponse du serveur est positive (c'est-à-dire si les identifiants sont corrects), la fonction effectue les modifications nécessaires pour connecter l'utilisateur.

```
1  function Login({ onLogin, login, error }) {
2    const [email, setEmail] = useState("");
3    const [password, setPassword] = useState("");
4
5    const handleSubmit = async (e) => {
6      e.preventDefault();
7
8      try {
9        const result = await login(email, password);
10       if (result.success) {
11         onLogin(result.user);
12       }
13     } catch (error) {
14       console.error("Login failed:", error);
15     }
16   };
17
18   return (
19   ...
20     <form onSubmit={handleSubmit}>
21       <input
22         type="email"
23         className="form-control"
24         id="email"
25         placeholder="email@domain.com"
26         value={email}
27         onChange={(e) => setEmail(e.target.value)}
28         required
29       />
30       ...
31       <input
32         type="password"
33         className="form-control"
34         id="password"
35         placeholder="Password"
36         value={password}
37         onChange={(e) => setPassword(e.target.value)}
38         required
39       />
40       ...
41       <button type="submit" className="btn btn-primary w-100">
42         Log in
43       </button>
44     </form>
45     ...
46   );
```

```
}
```

Listing 4.4 – Login.jsx

```
const login = useCallback(async (email, password) => {
  setLoading(true);
  setError(null);

  try {
    const response = await axios.post(
      "http://localhost:3000/login",
      { email, password },
      { withCredentials: true }
    );

    if (response.data.success) {
      // Save the token in cookies for persistence
      Cookies.set("token", response.data.accessToken, { expires: 7 });

      setIsAuthenticated(true);
      setUser(response.data.user); // Assuming user data is returned
      checkAuth();
      return { success: true };
    } else {
      throw new Error("Invalid credentials");
    }
  } catch (err) {
    setError(err.message);
    return { success: false, error: err.message };
  } finally {
    setLoading(false);
  }
}, []);
```

Listing 4.5 – Login Function

Auth

Le composant `Auth` correspond à la page d'inscription, comme illustré dans les figures 3.3 et 3.4. La logique de fonctionnement de la page d'inscription est très similaire à celle de la page de connexion, à l'exception de la fonction du hook utilisé, qui diffère ici.

Dans le cas de l'inscription, le composant `Auth` utilise la fonction `register`, comme indiqué dans le code 4.6. Aux lignes 5 à 10, on peut voir qu'une requête POST est envoyée au serveur pour enregistrer un nouvel utilisateur. Si la réponse du serveur est positive (c'est-à-dire que l'inscription est réussie), une session est démarrée et un cookie est créé pour conserver les informations de l'utilisateur.

```
const register = useCallback(async (email, name, password) => {
  setLoading(true);
  setError(null);

  try {
    const response = await axios.post(
      "http://localhost:3000/register",
      { email, name, password },
```

```
9       { withCredentials: true }
10    );
11
12    if (response.data.success) {
13      // Save the token in cookies for persistence
14      Cookies.set("token", response.data.accessToken, { expires: 7 }); // Token valid
            for 7 days
15
16      setIsAuthenticated(true);
17      setUser(response.data.user);
18      return { success: true };
19    } else {
20      throw new Error("Registration failed");
21    }
22  } catch (err) {
23    setError(err.message);
24    return { success: false, error: err.message };
25  } finally {
26    setLoading(false);
27  }
28}, []);
```

Listing 4.6 – Register Function

Jeu

Le composant `Jeu` est lui-même composé de deux composants principaux : `Game` et `FormJeu`. Le composant `Game` représente la partie active lorsqu'un joueur a rejoint une session de jeu, comme illustré dans les figures 3.17 et 3.20. De son côté, le composant `FormJeu` correspond à la page d'accueil permettant de rejoindre une session, également appelée la page `Join` comme sur la figure 3.15.

En analysant le code 4.7, on peut constater, entre les lignes 27 et 31, qu'une variable `sessionJeu` est utilisée pour déterminer quelle vue afficher : soit le jeu, soit la page d'inscription à la session. À la ligne 5, une instance de WebSocket est créée via le hook `useWebSocket`, permettant d'établir la connexion entre le client et le serveur. Cette connexion est activée à la ligne 10 lorsque l'utilisateur tente de rejoindre une session.

Un point important du code se trouve à la ligne 14, où le hook `useEffect` entre en jeu. Ce hook, propre à React, permet d'exécuter du code lorsque certaines variables changent d'état. Dans notre cas, la variable `message` (définie à la ligne 23) est surveillée. Lorsque la connexion WebSocket démarre, cette variable est continuellement mise à jour pour interpréter les différentes interactions entre le serveur et le client. Chaque message reçu déclenche une mise à jour et exécute une logique spécifique pour gérer les événements du jeu en temps réel.

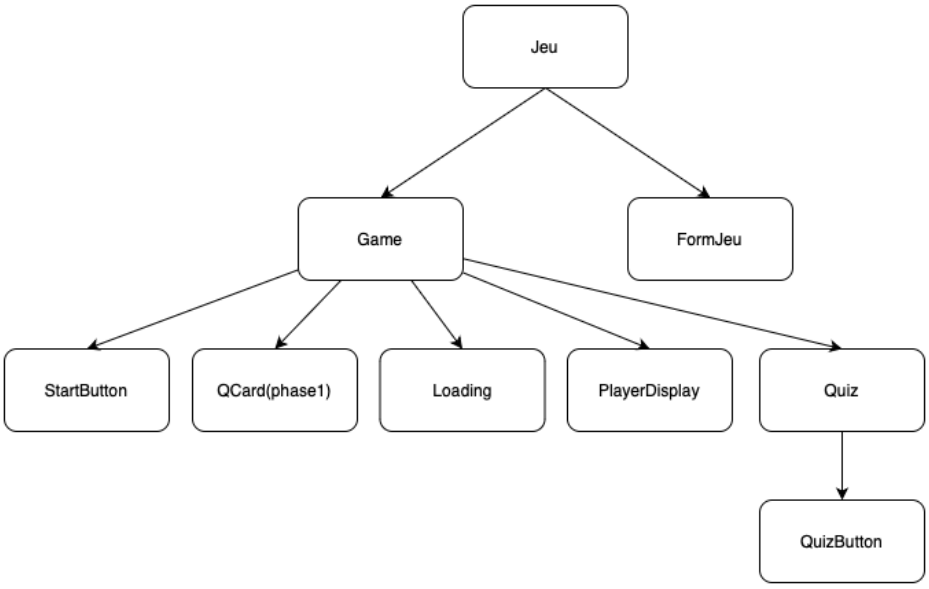

FIGURE 4.5. – Relation components Jeu

```
function Jeu() {
  const [sessionJeu, setSessionJeu] = useState(false);
  const [formData, setFormData] = useState(null);

  const { message, sendMessage, setUrl } = useWebSocket();

  useEffect(() => {
    if (formData) {
      const { name, tableId } = formData;
      setUrl(`ws://localhost:8080?id=${name}&table=${tableId}`);
    }
  }, [formData, setUrl]);

  useEffect(() => {
    console.log(message);
    if (message && sessionJeu != true) {
      setSessionJeu(true);
    }

    if (message.type == "end") {
      setSessionJeu(false);
    }
  }, [message]);

  return (
    <>
      {sessionJeu ? (
        <Game message={message} sendMessage={sendMessage} />
      ) : (
        <FormJeu onSubmit={handleFormSubmit} />
      )}
    </>
  );
```

```
}
```

Listing 4.7 – Jeu Function

Un point intéressant à observer dans les sous-composants est la manière dont l'application gère les différents états pendant le déroulement du jeu. Comme montré dans le code 4.8, extrait du composant `Game`, on trouve une variable `phaseJeu`, qui est mise à jour en fonction des messages reçus du serveur via les hooks WebSocket et l'analyse du composant `Game`.

Cette gestion des messages est également présente dans le code 4.9, où le hook `useEffect` est utilisé de manière efficace pour surveiller les changements de la variable `phaseJeu`. Chaque fois qu'un nouveau message est reçu du serveur, le hook `useEffect` s'assure que l'état de `phaseJeu` est mis à jour. Ce changement impacte directement l'état de la page et détermine quels composants doivent être affichés, ajustant ainsi l'interface en fonction de la progression du jeu.

```
<>
  {phaseJeu == 0 && (
    <div>
      {nbJoueurs.length === 0 ? (
        <p>Waiting players...</p>
      ) : (
        <PlayerDisplay nbJoueurs={nbJoueurs} />
      )}
      <StartButton sendMessage={sendMessage} />
    </div>
  )}

  {phaseJeu == 1 && (
    <QCard
      questions={questions}
      nbJoueurs={nbJoueurs}
      sendMessage={sendMessage}
      setIsLoading={setIsLoading}
    />
  )}

  {phaseJeu == 2 && (
    <Quiz
      nbJoueurs={nbJoueurs}
      setIsLoading={setIsLoading}
      sendMessage={sendMessage}
    />
  )}
</>
```

Listing 4.8 – Game Phases

```
useEffect(() => {
  switch (message.type) {
    case "nbJoueur":
      setNbJoueurs(message.content || []);
      break;
    case "questionsRound":
      setQuestions(message.content);
      break;
```

```
    case "changePhase":
      setPhase(message.content);
      if (message.content != phaseJeu) {
        setIsLoading(false);
        setPhase(message.content);
      }
      break;
    case "question":
      setIsLoading(false);
      break;
    default:
      break;
  }
}, [message]);
```

Listing 4.9 – useEffect for Message Handling

Displayer

Le composant `Displayer` suit une architecture presque identique à celle de `Game`, à l'exception de quelques différences mineures, notamment un nombre réduit de sous-composants et certaines spécificités visuelles. Toutefois, d'un point de vue logique et structurel, ces deux composants partagent une conception très similaire. Le rôle principal de `Displayer` est d'afficher les informations relatives au jeu pour les utilisateurs, tout en conservant la même logique de gestion d'état et de mise à jour dynamique des composants que `Game`.

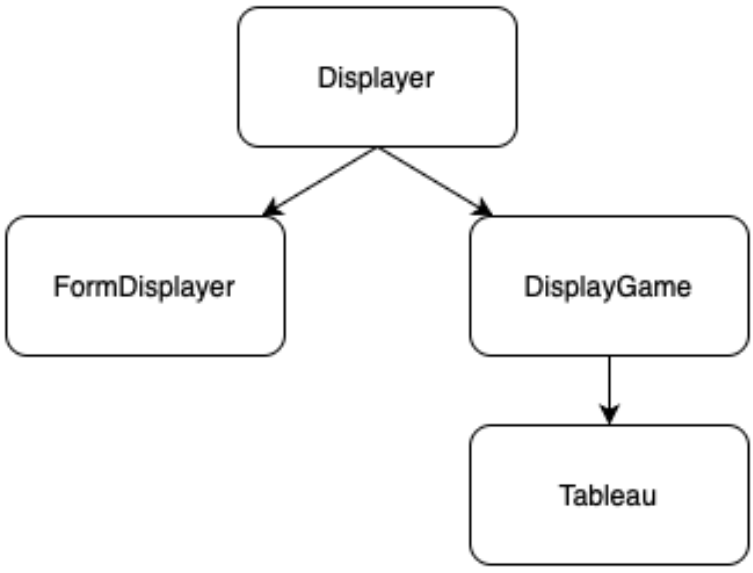

FIGURE 4.6. – Relation components Displayer

Hooks

Les hooks en React offrent une approche innovante pour gérer des aspects essentiels du cycle de vie des composants, notamment l'état et les effets secondaires. Parmi les outils

clés, les hooks `useState` et `useEffect` jouent un rôle fondamental. Le hook `useState` permet de gérer l'état local au sein des composants fonctionnels, en fournissant une méthode intuitive pour déclarer des variables d'état et les mettre à jour en réponse aux interactions ou aux changements internes. Quant à `useEffect`, il est utilisé pour prendre en charge les effets secondaires, tels que les appels API, les abonnements à des services externes ou encore les manipulations du DOM. Ce hook s'exécute après chaque rendu et peut être configuré pour s'ajuster en fonction des changements d'un ensemble spécifique de dépendances.

Outre les hooks natifs, React permet également de créer des `hooks personnalisés`, qui sont simplement des fonctions JavaScript utilisant les hooks existants pour encapsuler des logiques réutilisables. Cela permet de simplifier et de centraliser des processus complexes ou répétitifs, tels que la gestion d'un formulaire ou les appels d'API, tout en favorisant une meilleure lisibilité du code et en réduisant les duplications. Ces hooks personnalisés respectent la philosophie de React qui consiste à découpler la logique de l'interface utilisateur. En regroupant des fonctionnalités spécifiques dans ces hooks, les développeurs peuvent créer des composants plus modulaires, maintenables et évolutifs, tout en facilitant la réutilisation du code dans l'ensemble de l'application.

Hooks personnalisés

Dans le schéma 4.7, les losanges verts représentent des **hooks personnalisés** implémentés dans l'application React, chacun étant identifié par le nom indiqué à l'intérieur des losanges. Par exemple, on retrouve des hooks tels que `useAuth`, `useWebSocket`, et `msgDecode`, chacun étant conçu pour encapsuler une logique spécifique, réutilisable à travers divers composants. Ces hooks permettent de gérer des aspects essentiels du fonctionnement de l'application, comme l'authentification ou la communication via des WebSockets.

Les lignes vertes présentes sur le diagramme symbolisent les relations entre les différents composants et ces hooks personnalisés. Plus précisément, elles indiquent soit les composants qui **utilisent directement** les hooks, soit ceux qui les **transmettent en tant que paramètres** à leurs composants enfants. Cela met en évidence la manière dont ces hooks personnalisés sont partagés et appliqués à travers la hiérarchie des composants, facilitant ainsi la réutilisation du code tout en centralisant la gestion de certaines logiques complexes, telles que la gestion des messages ou l'authentification, dans des fonctions isolées et modulaires. Cette approche améliore la clarté et la maintenabilité du code, tout en permettant une modularité accrue des fonctionnalités.

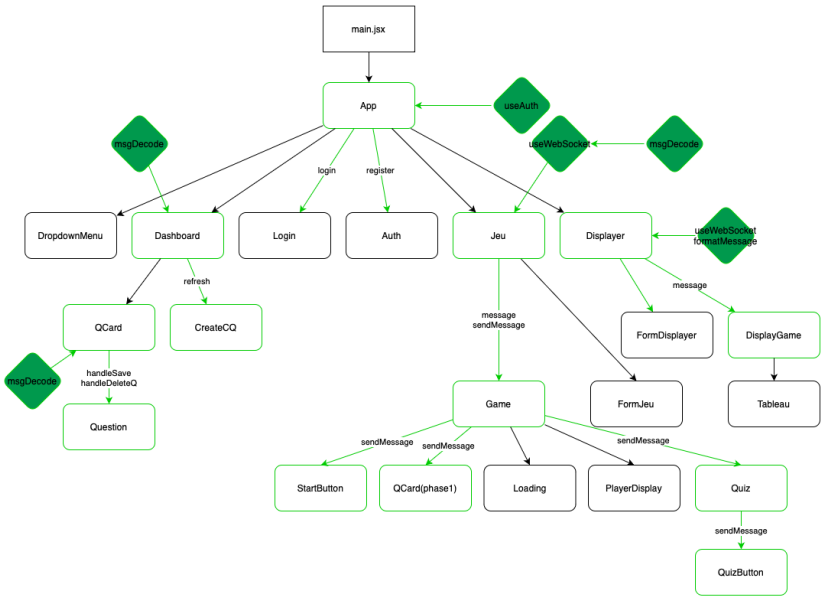

FIGURE 4.7. – Relation components Hooks

4.3. Back-end

4.3.1. Architecture

Pour mon architecture de back-end, j'ai opté pour la mise en place de deux serveurs distincts, l'un développé en `Node.js` [11] avec le framework `Express` [12], et l'autre en `GoLang` [14] utilisant le package `Gorilla` [6]. Ce choix d'architecture repose sur une approche de microservices, où chaque service est dédié à des tâches spécifiques. L'objectif principal est de garantir la scalabilité et la distribution des services, des critères essentiels dans les infrastructures modernes.

J'ai choisi `Express` pour agir en tant que proxy vers la base de données et pour gérer les connexions entrantes de manière efficace. Ce serveur `Node.js` est particulièrement adapté à la gestion des requêtes HTTP en raison de sa simplicité et de son écosystème mature. D'autre part, `GoLang` a été sélectionné pour gérer les connexions `WebSocket`, qui jouent un rôle clé dans la synchronisation des données en temps réel via une architecture multi-thread. `Go` est particulièrement bien adapté à cette tâche grâce à sa gestion native de la concurrence, ce qui permet de traiter efficacement de nombreuses connexions simultanées.

Deux aspects majeurs ont guidé ce choix architectural : la **distributivité** et la **concurrence**. En répartissant les responsabilités entre ces deux serveurs, j'assure non seulement une meilleure gestion des ressources, mais aussi une architecture capable de répondre aux exigences d'un environnement distribué tout en exploitant la puissance du multi-threading pour gérer les opérations en temps réel de manière performante.

La distributuion

La **distribution** dans le contexte des systèmes informatiques fait référence à l'architecture où les ressources et les services sont répartis sur plusieurs serveurs ou nœuds,

plutôt que centralisés sur une seule machine. Cette approche est au cœur des architectures modernes de microservices, qui fragmentent une application en plusieurs services indépendants, chacun exécutant une fonction précise. Chaque service peut être déployé, mis à jour et maintenu indépendamment des autres, offrant ainsi une flexibilité et une scalabilité accrues.

L'utilisation des microservices facilite la *scalabilité horizontale*, où de nouveaux services peuvent être ajoutés pour répondre à la demande croissante sans compromettre la stabilité de l'application. De plus, les microservices permettent de gérer les échecs de manière isolée : si un service rencontre un problème, cela n'affecte pas nécessairement le reste du système, augmentant ainsi la **résilience** de l'architecture [20].

Les recherches académiques montrent que les architectures de microservices sont particulièrement adaptées pour les environnements **distribués** dans le cloud, où les ressources sont dynamiquement allouées et répliquées selon les besoins [8]. Cela contraste avec les architectures monolithiques, qui tendent à être moins flexibles et plus complexes à mettre à l'échelle. Ainsi, la **distribution** des services au sein des microservices permet non seulement une meilleure utilisation des ressources matérielles et logicielles, mais aussi une plus grande capacité à évoluer et à répondre aux besoins des utilisateurs de manière efficace.

4.3.2. NodeJS

`Node.js` est un environnement d'exécution JavaScript côté serveur qui permet de développer des applications réseau performantes et évolutives. Il repose sur le moteur JavaScript V8 de Google, ce qui lui confère une grande rapidité d'exécution. L'un des principes fondamentaux de `Node.js` est son modèle événementiel et non-bloquant, qui permet de gérer un grand nombre de connexions simultanées avec une faible empreinte mémoire [24].

Philosophie et Points forts

Node.js est particulièrement utilisé pour la création de serveurs web légers, le développement d'API RESTful, ainsi que la gestion de connexions en temps réel avec des protocoles tels que WebSocket [16]. Il est également très populaire dans les architectures de microservices, grâce à sa capacité à gérer des processus en parallèle avec un faible coût en termes de ressources.

La philosophie derrière Node.js repose sur l'idée de "non-blocking I/O", permettant d'exécuter des opérations d'entrée/sortie sans bloquer l'exécution du programme. Ce modèle favorise la scalabilité, notamment pour les applications nécessitant de multiples connexions réseau simultanées. L'un des principaux points forts de Node.js est sa grande vitesse et son efficacité dans les applications nécessitant des E/S intensives, telles que les serveurs HTTP ou les gestionnaires de fichiers [22].

Points faibles

Cependant, Node.js n'est pas sans faiblesses. L'un de ses principaux inconvénients est la difficulté de gérer des tâches computationnelles lourdes en raison de son modèle monothread. Bien que le modèle événementiel soit idéal pour les E/S, il n'est pas adapté aux

calculs intensifs, pouvant ainsi entraîner des goulets d'étranglement dans des applications nécessitant beaucoup de traitement de données [17].

Mon usage de Node

Dans le cadre du prototype que nous allons analyser, `Node.js` a principalement été utilisé en tant que proxy vers la base de données, comme l'illustre la figure 4.1. Ce rôle de proxy permet à l'application de gérer efficacement les requêtes et les connexions entre l'application web et la base de données. Lorsque l'utilisateur se connecte à l'application, celle-ci interagit avec une API RESTful, qui expose différents endpoints. Ces endpoints permettent la gestion des opérations de création, lecture, mise à jour et suppression (CRUD), assurant une communication fluide avec la base de données sous-jacente.

Voici un tableau détaillant les principaux endpoints de l'API RESTful utilisée dans l'application :

Méthode HTTP	**Endpoints**
POST	`/login`, `/register`, `/logout`, `/CQ`, `/Question`, `/newTable`
GET	`/check-auth`, `/QCards`
DELETE	`/Question/:questionId`, `/CQ/:qcatId`
PATCH	`/Title`, `/Question`

TABLE 4.2. – Liste des endpoints exposés par l'API RESTful de l'application.

Dans ce tableau, les méthodes HTTP telles que `POST`, `GET`, `DELETE`, et `PATCH` sont associées à différents endpoints, chacun permettant une interaction spécifique avec l'application. Les requêtes `POST` sont utilisées pour des actions de création, telles que l'inscription (`/register`) ou l'ajout de nouvelles entrées (`/Question`, `/newTable`). Les requêtes `GET` permettent de récupérer des informations, comme vérifier l'authentification avec `/check-auth` ou obtenir une liste de cartes de questions avec `/QCards`. Les requêtes `DELETE` sont destinées à supprimer des ressources spécifiques, identifiées par un `ID`, tandis que les requêtes `PATCH` permettent de mettre à jour des éléments tels que les titres ou les questions existantes.

Ce modèle d'API RESTful assure une organisation claire et concise des interactions avec la base de données, tout en facilitant la gestion des utilisateurs et des données de l'application.

Code

Nous allons commencer par analyser un endpoint de chaque type, puis nous examinerons l'interaction avec la base de données, et enfin nous aborderons la partie la plus complexe, à savoir le mécanisme d'authentification via les tokens et les cookies.

POST Dans le code présenté à la liste 4.10, nous examinons le endpoint `/login` du prototype. À la ligne 1, l'endpoint est déclaré en utilisant la méthode `app.post`, ce qui indique qu'il répond aux requêtes HTTP POST. La fonction est déclarée comme asynchrone avec le mot-clé `async`, nécessaire pour exécuter des opérations qui peuvent prendre du temps, comme l'accès à la base de données.

La fonction utilise la syntaxe de déstructuration de JavaScript pour extraire l'`email` et le `password` du corps de la requête, comme le montre la ligne 2. Il s'agit d'une fonctionnalité moderne de JavaScript qui simplifie l'extraction des propriétés d'un objet.

À la ligne 5, la fonction `isInDb(email, password)` est appelée pour vérifier si les identifiants fournis par l'utilisateur existent dans la base de données. Le mot-clé `await` est utilisé ici, ce qui suspend l'exécution de la fonction asynchrone jusqu'à ce que la promesse renvoyée par `isInDb` soit résolue. Cela est nécessaire car les opérations de base de données sont asynchrones et peuvent prendre du temps à s'exécuter.

Si l'utilisateur est authentifié (c'est-à-dire que `user.isAuth` est vrai), un `accessToken` est généré à la ligne 8 en utilisant la méthode `jwt.sign`. Ce jeton d'accès contient l'identifiant et le nom de l'utilisateur, est signé avec une clé secrète, et a une durée d'expiration d'une heure. Ce jeton sera utilisé pour authentifier les requêtes suivantes.

Si l'authentification échoue, le serveur répond avec un statut `401 Unauthorized` et un message JSON indiquant que les identifiants sont invalides. Toute erreur rencontrée pendant le processus est capturée par le bloc `catch`, qui envoie une réponse `500 Internal Server Error`.

```
1  app.post("/login", async (req, res) => {
2    const { email, password } = req.body;
3    try {
4      // Check user info in the database
5      const user = await isInDb(email, password);
6      if (user.isAuth) {
7        // Generate a JWT for the authenticated user
8        const accessToken = jwt.sign(
9          { id: user.authID, name: user.userName },
10         jwtSecret,
11         { expiresIn: "1h" }
12       );
13       res.json({ success: true, accessToken });
14     } else {
15       res.status(401).json({ success: false, message: "Invalid credentials" });
16     }
17   } catch (error) {
18     res.status(500).json({ success: false, message: "Internal server error" });
19   }
20 });
```

Listing 4.10 – POST /login

GET La fonction GET présentée dans le code 4.11 permet de récupérer les cartes de jeu qu'un utilisateur possède. Contrairement à l'endpoint POST précédent, cet endpoint inclut une vérification d'authentification. À la ligne 1, la fonction middleware `authenticateJWT` est utilisée pour vérifier que la requête est autorisée à accéder aux informations demandées. Ce middleware vérifie le JSON Web Token (JWT) fourni par le client pour s'assurer que l'utilisateur est authentifié.

Lorsque l'utilisateur se connecte avec ses identifiants, une session d'authentification est créée, et un JWT est émis. Ce jeton est ensuite inclus dans les en-têtes des requêtes suivantes pour authentifier l'utilisateur.

À la ligne 4, l'identifiant de l'utilisateur est obtenu à partir de l'objet de requête (`req.user.id`). Cet identifiant a été incorporé dans le JWT lors du processus de connexion et est maintenant utilisé pour effectuer des requêtes à la base de données spécifiques à cet utilisateur.

La fonction récupère ensuite de manière asynchrone les cartes de l'utilisateur à partir de la base de données en appelant `getQ(req.user.id)`. Pour chaque carte, elle récupère les questions associées en appelant `getQuestions(card.id)`. Elle construit un tableau de `Q`, chacun contenant une carte et ses questions.

Si l'opération réussit, le serveur répond avec le tableau `Q` au format JSON à la ligne 11. Si une erreur se produit pendant le processus, elle est capturée par le bloc `catch`, et le serveur répond avec un statut `500 Internal Server Error`.

```
1  app.get("/Question", authenticateJWT, async (req, res) => {
2    try {
3      let Q = [];
4      let cards = await getQ(req.user.id);
5
6      for (const card of cards) {
7        const questions = await getQuestions(card.id);
8        const qcard = { card: card, questions: questions };
9        Q.push(qcard);
10     }
11     res.json(Q);
12   } catch (error) {
13     console.error("Error fetching Q:", error);
14     res.status(500).send("Internal Server Error");
15   }
16 });
```

Listing 4.11 – GET /Question

DELETE L'endpoint présenté dans le code 4.12 introduit une logique supplémentaire par rapport aux exemples précédents. À la ligne 1, l'endpoint est déclaré en utilisant `app.delete`, indiquant qu'il répond aux requêtes HTTP DELETE. Le middleware *authenticateJWT* est de nouveau utilisé pour s'assurer que l'utilisateur est authentifié avant de poursuivre.

Cet endpoint inclut un paramètre dynamique `:qcardID` dans son URL, qui représente l'identifiant de la carte à supprimer. À la ligne 2, `qcardID` est extrait des paramètres de la requête en utilisant `req.params.qcardID`, et converti en entier. Cet identifiant spécifie la carte particulière que l'utilisateur souhaite supprimer.

À la ligne 3, la fonction `deleteCard(QCARDID)` est appelée pour interagir avec la base de données et supprimer la carte spécifiée. Cette fonction effectue généralement une opération de base de données pour supprimer l'enregistrement de la carte correspondant à `QCARDID`.

Cet endpoint permet aux utilisateurs authentifiés de supprimer leurs propres cartes. La gestion des erreurs et la réponse au client ne sont pas explicitement montrées dans cet extrait de code, mais dans un environnement de production, il serait important de gérer les erreurs potentielles et de fournir un retour approprié au client.

```
1  app.delete("/CQ/:qcardID", authenticateJWT, (req, res) => {
2    const QCARDID = parseInt(req.params.qcardID);
3    deleteCard(QCARDID);
```

```
});
```

Listing 4.12 – DELETE /CQ

PATCH Le endpoint PATCH présenté dans le code 4.13 permet aux utilisateurs authentifiés de modifier le titre d'une ressource, telle qu'une carte. L'endpoint est déclaré à la ligne 1 en utilisant `app.patch`, ce qui indique qu'il répond aux requêtes HTTP PATCH à l'URL `"/Title"`. Le middleware `authenticateJWT` assure que seuls les utilisateurs authentifiés peuvent accéder à cet endpoint.

Au sein de la fonction asynchrone, l'`id` et le `title` sont extraits du corps de la requête en utilisant la syntaxe de déstructuration de JavaScript, comme le montrent les lignes 2 et 3. L'`id` représente l'identifiant de la ressource à mettre à jour, et `title` est le nouveau titre fourni par l'utilisateur.

À la ligne 5, le code vérifie si le titre est fourni. Si le titre est manquant, le serveur répond avec un statut `400 Bad Request` et un message d'erreur JSON indiquant que le titre est requis. Cette validation garantit que la requête contient toutes les informations nécessaires pour poursuivre.

À la ligne 8, la fonction `updateTitleInDatabase(id, title)` est appelée pour effectuer l'opération de mise à jour dans la base de données. Cette fonction est attendue (`await`), car il s'agit probablement d'une opération asynchrone impliquant un accès à la base de données.

Si la mise à jour réussit, le serveur répond avec un statut `200 OK` et un message JSON confirmant la mise à jour, comme le montre la ligne 10. Si une erreur se produit pendant le processus, le bloc `catch` à la ligne 12 enregistre l'erreur et répond avec un statut `500 Internal Server Error`, accompagné d'un message d'erreur JSON.

Cet endpoint illustre comment gérer les mises à jour partielles des ressources en utilisant la méthode HTTP PATCH, qui est destinée à appliquer des modifications partielles à une ressource.

```
app.patch("/Title", authenticateJWT, async (req, res) => {
  try {
    const { id } = req.body;
    const { title } = req.body;

    if (!title) {
      return res.status(400).json({ error: "Title is required" });
    }
    const updateResult = await updateTitleInDatabase(id, title);

    return res.status(200).json({ message: updateResult.message });
  } catch (error) {
    console.error("Error updating title:", error);
    res
      .status(500)
      .json({ error: "An error occurred while updating the title" });
  }
});
```

Listing 4.13 – PATCH /Title

Token d'authentification

Pour gérer les authentifications au sein de l'application et contrôler l'accès des joueurs aux différentes cartes de jeu, j'utilise le *JSON Web Token* (JWT). Le JWT est un standard ouvert permettant l'échange sécurisé d'informations sous forme de token signé. Il est couramment utilisé pour l'authentification dans les applications web modernes, car il offre un mécanisme stateless. Cela signifie que le serveur n'a pas besoin de stocker des informations de session, ce qui allège la gestion côté serveur et améliore la scalabilité.

Dans mon application, j'ai implémenté un *middleware* Express.js qui crée un JWT lors de la connexion de l'utilisateur. Ce token est ensuite inclus dans les en-têtes HTTP pour toutes les requêtes ultérieures nécessitant une authentification. Le middleware vérifie la validité du token pour autoriser ou refuser l'accès aux ressources protégées. Le code du middleware est présenté ci-dessous (voir code 4.14).

```
app.use(express.json());

// Middleware pour vrifier le JWT
function authenticateJWT(req, res, next) {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];

  if (!token) {
    return res.sendStatus(401); // Aucun token fourni
  }

  jwt.verify(token, jwtSecret, (err, user) => {
    if (err) {
      console.error("Erreur de vrification du token :", err.message);
      return res.sendStatus(403); // Token invalide
    }

    req.user = user; // Stocke les informations de l'utilisateur dans la requte
    next();
  });
}
```

Listing 4.14 – Middleware de vérification du JWT

Le fonctionnement du middleware est le suivant :

- Le token JWT est extrait de l'en-tête `Authorization` de la requête HTTP, au format `Bearer <token>`.
- S'il n'y a pas de token présent, le serveur renvoie une réponse avec un statut HTTP 401 (*Unauthorized*), signalant que l'authentification est requise.
- Le token est vérifié à l'aide de la clé secrète `jwtSecret`. Si le token est invalide ou expiré, le serveur renvoie une réponse HTTP 403 (*Forbidden*).
- Si le token est valide, les informations de l'utilisateur sont stockées dans l'objet `req.user`, afin qu'elles puissent être utilisées dans les requêtes ultérieures. Ensuite, le middleware appelle la fonction `next()` pour passer au middleware ou au gestionnaire de route suivant.

Le JWT permet de vérifier efficacement l'identité de l'utilisateur tout en assurant qu'il a les droits nécessaires pour accéder aux ressources demandées. Grâce à sa nature stateless,

le JWT réduit également la charge sur le serveur, ce qui améliore les performances et la scalabilité de l'application.

Communication avec la base de données

Pour illustrer la manière dont je communique avec la base de données, je vais utiliser la fonction `isInDb`, que nous avons vue dans le code 4.10. Cette fonction montre comment mon serveur interagit avec la base de données en envoyant une requête SQL pour vérifier les informations d'un utilisateur. Le code de cette fonction est présenté dans le listing 4.15. Les autres fonctions qui interagissent avec la base de données suivent une logique similaire, seules les requêtes SQL changent.

```
async function isInDb(email, password) {
  const sql = "SELECT * FROM users WHERE email = ? AND pass = ?";
  try {
    const row = await getQuery(sql, [email, password]);
    if (row) {
      return { isAuth: true, authID: row.id, userName: row.nom };
    }
    return { isAuth: false };
  } catch (err) {
    throw new Error(‘Erreur lors de la vrification de l’utilisateur : ${err.message}‘)
        ;
  }
}
```

Listing 4.15 – Vérification des informations de l'utilisateur dans la base de données

Cette fonction interagit avec la base de données SQLite pour vérifier si un utilisateur existe avec l'adresse e-mail et le mot de passe fournis. Voici son fonctionnement détaillé :

- **Paramètres d'entrée** : La fonction reçoit l'adresse e-mail et le mot de passe saisis par l'utilisateur lors de la tentative de connexion.
- **Requête SQL** : Une requête SQL est construite pour sélectionner toutes les colonnes de la table `users` où l'e-mail et le mot de passe correspondent. Les paramètres de la requête sont passés de manière sécurisée, via des placeholders (points d'interrogation), afin de prévenir les attaques par injection SQL.
- **Exécution de la requête** : La fonction utilise `getQuery`, une fonction asynchrone qui interagit directement avec la base de données, pour exécuter la requête et récupérer les résultats.
- **Traitement des résultats** :
  - Si une ligne correspondant est trouvée, la fonction renvoie un objet indiquant que l'utilisateur est authentifié (`isAuth: true`), ainsi que l'identifiant et le nom de l'utilisateur.
  - Si aucune correspondance n'est trouvée, la fonction renvoie `isAuth: false`.
- **Gestion des erreurs** : En cas d'erreur lors de l'exécution de la requête, une exception est levée avec un message d'erreur détaillé, ce qui facilite le débogage.

La fonction `getQuery` est responsable de l'exécution de la requête SQL et de la gestion de la communication avec la base de données. Cette abstraction permet de réutiliser

`getQuery` pour d'autres opérations (comme les insertions, mises à jour, suppressions) tout en conservant un code propre et maintenable.

En utilisant des requêtes SQL paramétrées, cette méthode renforce la sécurité de l'application en évitant les attaques par injection SQL. De plus, la structure asynchrone améliore la lisibilité du code et permet une meilleure gestion des opérations de base de données dans un environnement web asynchrone.

4.3.3. Go

Le langage Go, souvent appelé Golang, est un langage de programmation moderne créé par Google en 2009. Sa philosophie repose sur la simplicité, la performance et l'efficacité, tout en favorisant la lisibilité du code. Conçu pour répondre aux besoins des systèmes distribués et du cloud computing, Go se distingue par sa gestion concurrente intuitive grâce à ses goroutines, permettant de gérer des milliers de tâches simultanément avec une consommation minimale de ressources. Un de ses grands atouts est la compilation rapide et le faible temps d'exécution, ce qui en fait un outil idéal pour développer des applications performantes à grande échelle. En plus de sa rapidité, Go prône un modèle de conception minimaliste, se concentrant sur l'essentiel et évitant les complexités inutiles, ce qui permet aux développeurs de produire du code fiable et maintenable. Utilisé dans des projets critiques comme Docker et Kubernetes, Go s'est imposé comme un incontournable dans le développement d'infrastructures et de systèmes backend robustes [7].

Mise en place

L'installation et l'initialisation d'un projet Go se font avec une simplicité remarquable, vous permettant de vous plonger rapidement dans le développement. Tout d'abord, accédez au site officiel de Go [2] et téléchargez la distribution adaptée à votre système d'exploitation. Une fois l'installation terminée, vérifiez-la en exécutant la commande suivante dans votre terminal :

```
go version
```

Pour initialiser un nouveau projet, commencez par créer un répertoire de travail pour votre projet, puis lancez la commande :

```
go mod init nom_du_projet
```

Cette commande crée le fichier `go.mod`, essentiel pour la gestion des dépendances du projet. Il s'agit d'une pratique moderne et modulaire qui assure une meilleure organisation et reproductibilité.

Ensuite, il vous suffit de créer un fichier `main.go` dans lequel vous pouvez écrire votre fonction principale `main()`. Pour exécuter votre application, utilisez simplement la commande :

```
go run main.go
```

[2] `https://golang.org/dl/`

Installation d'un package Go facilite également l'ajout de bibliothèques tierces à votre projet via la gestion des dépendances. Par exemple, pour installer un package, tel que `gorilla/mux` comme j'ai utilisé dans mon application, un routeur HTTP populaire, il vous suffit d'exécuter la commande suivante dans le répertoire de votre projet :

```
go get github.com/gorilla/mux
```

Cette commande récupère le package spécifié et l'ajoute automatiquement au fichier `go.mod`, garantissant ainsi que votre projet conserve une trace des versions des dépendances utilisées. Vous pouvez ensuite importer ce package dans votre fichier `main.go` :

```
import "github.com/gorilla/mux"
```

Cela vous permet d'accéder aux fonctionnalités du package et de les intégrer dans votre application. Pour réutiliser et gérer ces dépendances lors du développement futur ou sur d'autres machines, Go gère automatiquement le téléchargement et la mise à jour des packages via son module de gestion intégré [4].

Code

Dans cette section, nous étudions le microservice qui gère la synchronisation des clients et le contrôle du flux du jeu dans mon prototype. La figure 4.8 présente la structure des classes principales de l'application, leurs interactions et les routines Go déclenchées lors de l'initialisation.

Les différentes classes et leurs interactions sont décrites ci-dessous :

- **main.go** : Cette classe détient les instances des classes `Table` et `Client` lors de leur création, puis ne conserve que les tables, tandis que les clients sont stockés dans les tables. Lors de son initialisation, elle déclenche deux routines Go principales :
  1. `newTable.GameStart()` : Lance une nouvelle partie en initialisant les ressources nécessaires.
  2. `verifyActivity()` : Vérifie l'activité des tables, garantissant que celles-ci restent synchronisées avec le serveur.
- **table.go** : Cette classe contient deux instances importantes : `GameManager` et `Client`. Elle joue un rôle central dans la gestion du jeu, en orchestrant la communication entre les clients et le gestionnaire de jeu (`GameManager`).
- **gameManager.go** : Gère la logique de jeu à haut niveau. Cette classe possède une instance de `Table` (pointeur vers une instance de `Table`), ce qui lui permet d'accéder à toutes les données relatives à l'état du jeu. Elle ne lance pas de routine Go directement, mais elle est souvent appelée par `table.go` pour la coordination du jeu.
- **client.go** : Cette classe gère les interactions avec les clients. Elle possède des méthodes importantes qui permettent la transmission des messages et des informations entre le serveur et les clients. Deux routines Go sont particulièrement importantes ici :
  1. `client.SendToClient()` : Envoie des informations aux clients connectés.
  2. `client.ClientReceiveMessage()` : Gère la réception des messages envoyés par les clients.

L'ensemble des classes de l'application communique principalement à travers des méthodes spécifiques, ce qui garantit un flux de messages et d'événements bien structuré. Les routines Go permettent une gestion asynchrone et concurrente des interactions avec les clients, assurant ainsi un niveau de réactivité et de performance adapté aux besoins du microservice.

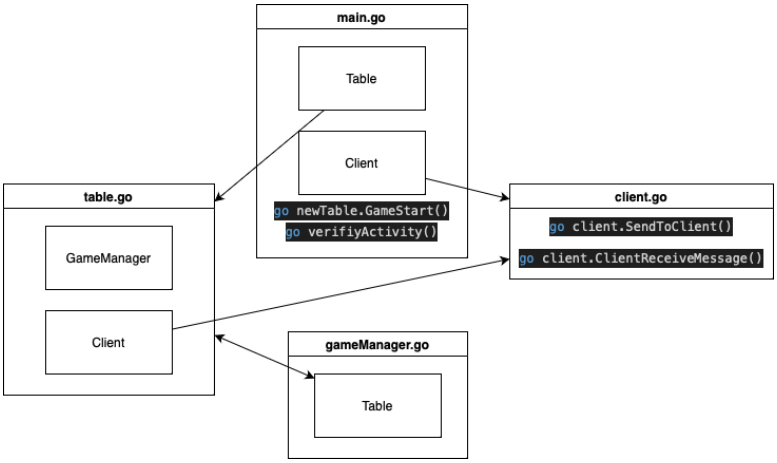

FIGURE 4.8. – Structure des classes en Go et leurs interactions

Il existe deux types d'activités principales dans la partie Go de mon application : la création d'une table, déclenchée par la partie Node, et la connexion en tant que joueur, effectuée par les utilisateurs via la partie React. Concentrons-nous ici sur la création d'une table, comme illustré dans le diagramme 4.9. Ce processus débute lorsqu'une requête est envoyée à l'endpoint `/newTable` en utilisant la méthode `POST`. La fonction `postHandler` est alors responsable de la réception et du traitement de cette requête et de crée une table. La fonction, dans le code 4.16 montre la fonction `NewTable` qui est chargé de crée une table.

Cette structure Table contient plusieurs éléments essentiels pour la gestion du jeu, notamment un pointeur vers une instance de `gameManager`, qui orchestre les différentes phases du jeu. De plus, elle possède une liste de clients, qui représentent les joueurs connectés à la table. D'autres variables importantes sont également incluses, mais ce qui retient particulièrement notre attention ce sont les deux canaux (*channels*) propres à Go : `clientAddedCh` et `clientremoveCh`. Ces canaux permettent une communication asynchrone entre les différentes parties du programme, facilitant ainsi la gestion efficace de l'ajout et de la suppression de clients en temps réel, sans bloquer les autres opérations.

Pour conclure, `main.go` retourne l'ID de la table au client après avoir initialisé les composants nécessaires. La fonction `/newTable` renvoie l'instance `Table`, prête à être utilisée dans le flux de jeu. Une fois la table récupérée, `main.go` lance une *goroutine*, comme illustré à la ligne 3 du code 4.17. Une *goroutine* permet de lancer un thread concurrent en Go, offrant une exécution simultanée tout en évitant les complexités habituelles associées à la gestion des threads.

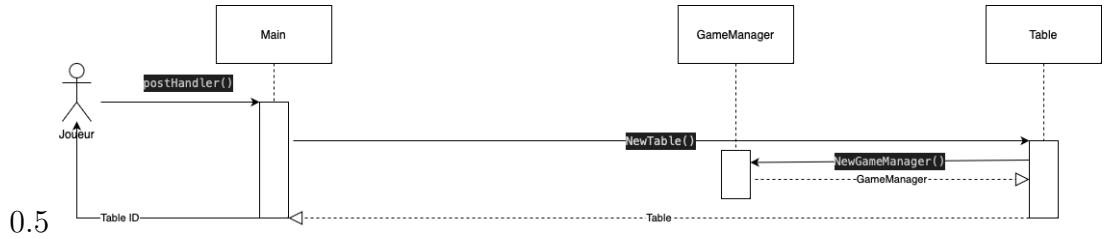

FIGURE 4.9. – Création de la Table

```go
func NewTable(id string, questions QuestionsData) *Table {
    t := &Table{
        ID: id,
        Questions: questions,
        Clients: []*Client{},
        DisplayViews: []*Client{},
        Phase: 0,
        clientAddedCh: make(chan struct{}, 1),
        clientremoveCh: make(chan struct{}, 1),
    }

    // Initialisation du GameManager avec rfrence  la Table
    t.gameManager = NewGameManager(t)
    return t
}
```

Listing 4.16 – postHandler en Go

```go
newTable := models.NewTable(getTableID(), questionsData)
tables = append(tables, newTable)
go newTable.GameStart()
```

Listing 4.17 – Table Go routine

La seconde activité, qui concerne la connexion en tant que client, se déroule comme illustré dans le diagramme de flux présenté à la figure 4.10. Le processus commence par la réception de la requête dans la fonction `homeConnect`, qui a pour rôle de créer une instance de la classe `Client`. Ce client nouvellement créé s'enregistre ensuite auprès d'une entité `Table`, établissant ainsi la première étape de la connexion.

La seconde étape de cette requête est la mise en place de la connexion WebSocket. Celle-ci est essentielle pour permettre la communication en temps réel, régie par un protocole spécifique. Une fois la connexion WebSocket établie, les messages échangés sont pris en charge par une fonction spécifique de la classe `Client`, qui fonctionne en tant que *goroutine* distincte. Cela permet une réception asynchrone des messages sans interrompre le flux principal du programme.

Les messages reçus sont ensuite décryptés à l'aide de la classe `Message`. Dans mon application, tous les messages suivent un format standardisé : chaque message comporte un champ `type` et un champ `content`, assurant ainsi une uniformité dans la structure des données échangées. La classe `Message` a pour fonction de formaliser et d'harmoniser ces messages, garantissant une cohérence dans l'envoi et la réception des informations.

Une fois le message décrypté, il est transmis au `GameManager`, qui se charge de le traiter en fonction de son type. Si, à ce stade, le `GameManager` doit répondre ou envoyer des

informations, il crée une nouvelle instance de la classe `Message`, encapsule les données appropriées, puis transmet cette instance à la classe `Client`. Ce dernier se charge alors d'envoyer le message via la connexion WebSocket, assurant ainsi un échange fluide et structuré entre les différentes entités de l'application.

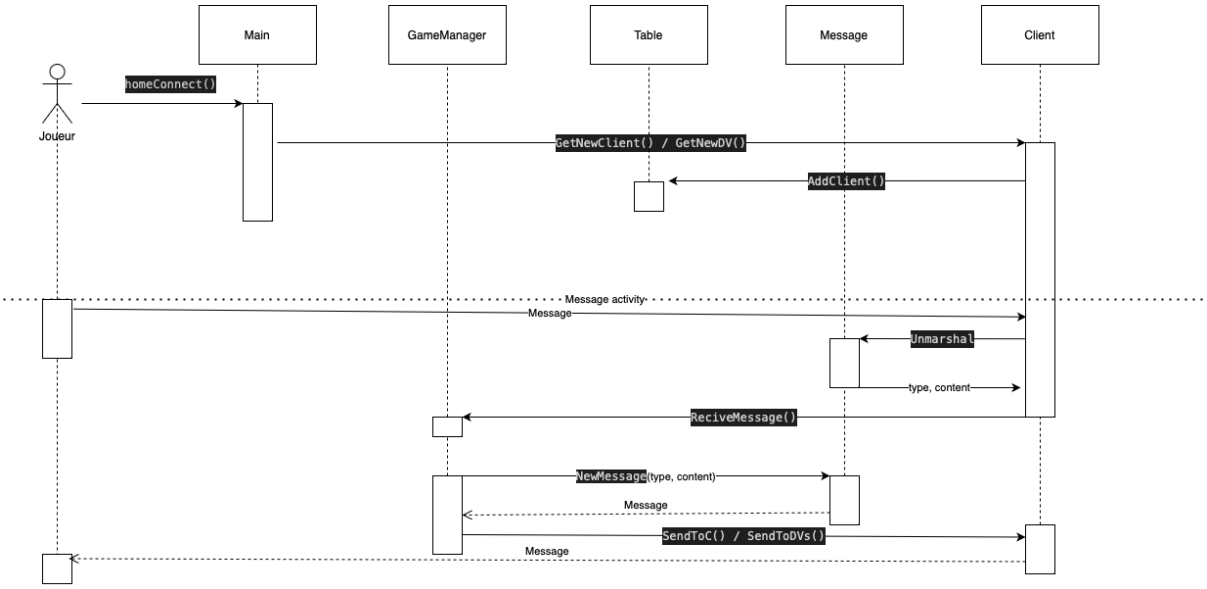

FIGURE 4.10. – Message flow

4.3.4. Problème rencontré

Lors de l'établissement de la connexion WebSocket entre mon client développé en React et mon serveur écrit en Go, j'ai rencontré un problème spécifique : React ne met pas à jour son affichage lorsque des informations consécutives arrivent rapidement sur le navigateur Safari. Ce comportement est dû à une mise à jour de React 18 et au moteur JavaScript de Safari, qui ignore le flux d'informations entrant et ne traite que le dernier message reçu. Ce phénomène, documenté dans un article sur GitHub [9], explique que cette optimisation vise à réduire les mises à jour inutiles de l'interface utilisateur, mais pose un problème dans les scénarios où chaque message est crucial. En effet, Safari, en tentant d'optimiser les performances, fusionne les mises à jour rapides, ce qui entraîne une perte de données intermédiaires. Cela crée une latence apparente ou un décalage dans la synchronisation entre l'état du client et celui du serveur.

Pour remédier à ce problème, j'ai expérimenté plusieurs modifications du côté front-end, mais sans succès. J'ai donc développé la solution suivante, illustrée dans les blocs de code 4.18 et 4.19. L'idée principale consiste à restructurer la gestion des messages côté serveur. Plutôt que d'envoyer les informations directement au client à chaque mise à jour, j'ai implémenté une file d'attente de messages, où chaque message est mis en mémoire tampon. Cette file est ensuite parcourue et les messages sont envoyés à des intervalles réguliers, garantissant que chaque information parvient au client de manière contrôlée et dans le bon ordre. Ainsi, même si Safari ignore certains flux rapides, la cohérence des données est maintenue grâce à cette gestion optimisée du flux de messages. De plus, une vérification de la réception correcte des messages côté client permet de s'assurer

qu'aucune information critique n'est perdue, offrant ainsi une meilleure synchronisation entre le client et le serveur.

```
func (c *Client) SendToClient() {
    for {
        if !c.Queue.IsEmpty() {
            msg, err := c.Queue.Dequeue()
            if err == nil {
                jsonData, err := json.Marshal(msg)
                if err != nil {
                    log.Println("JSON marshal error:", err)
                    continue
                }

                c.mu.Lock()
                err = c.Conn.WriteMessage(websocket.TextMessage, jsonData)
                c.mu.Unlock()

                if err != nil {
                    log.Println("Write error:", err)
                    continue
                }
            }
        }
        time.Sleep(750 * time.Millisecond)
    }
}
```

Listing 4.18 – SendToClient

Dans la fonction `GetNewClient()`, illustrée au code 4.19, on peut constater à la ligne 9 que lors de la création d'un nouveau client, celui-ci est initialisé avec un objet de type `Queue`, qui est une file d'attente de messages. Cette structure `Queue` est définie dans le fichier `fifo.go` de mon code source et implémente les opérations de base d'une file d'attente FIFO (First-In, First-Out). Ensuite, à la ligne 14, une nouvelle goroutine est lancée pour exécuter la fonction `SendToClient()`. Cette fonction, représentée dans le code 4.18, fonctionne de la manière suivante.

```
func GetNewClient(id string, table *Table, conn *websocket.Conn) *Client {
    client := &Client{
        ID: id,
        Table: table,
        Conn: conn,
        Channel: make(chan string),
        Done: make(chan struct{}),
        Score: 0,
        Queue: Queue{},
    }
    table.AddClient(client)

    go client.SendToClient()

    go client.ClientReceiveMessage()
    ...
}
```

Listing 4.19 – GetNewClient

La fonction `SendToClient()` est essentielle pour gérer l'envoi des messages au client de manière contrôlée. Elle exécute une boucle infinie qui, à chaque itération, vérifie si la file d'attente `Queue` du client n'est pas vide. Si un message est disponible, il est retiré de la file grâce à la méthode `Dequeue()`. Le message est ensuite sérialisé en format JSON à l'aide de `json.Marshal(msg)`. En cas de succès, le message sérialisé est envoyé au client via la connexion WebSocket en utilisant `c.Conn.WriteMessage()`. Pour assurer la sécurité des threads lors de l'écriture sur la connexion, un mutex (`c.mu.Lock()`) est utilisé pour verrouiller la ressource partagée. Après l'envoi du message, la fonction attend 750 millisecondes avant de continuer, ce qui régule le flux de messages envoyés au client.

Cette approche garantit que les messages sont envoyés au client à un rythme contrôlé, évitant ainsi les problèmes d'optimisation du navigateur Safari qui peuvent entraîner la perte de messages lorsqu'ils arrivent trop rapidement. En utilisant une file d'attente et en espaçant les envois, on s'assure que chaque message important est reçu et traité par le client, améliorant ainsi la fiabilité de l'application.

Par ailleurs, la gestion asynchrone des messages grâce aux goroutines permet au serveur de continuer à traiter d'autres tâches sans être bloqué par l'envoi des messages. Cela améliore les performances globales du serveur et assure une meilleure réactivité de l'application.

En conclusion, en adaptant la gestion du flux de messages côté serveur et en mettant en place une communication plus robuste entre le client et le serveur, il a été possible de contourner les limitations imposées par les optimisations de React et du navigateur Safari. Cette solution assure une meilleure synchronisation et fiabilité de l'application, tout en maintenant des performances satisfaisantes.

5 Conclusion

5.1. Résultats

L'objectif principal de ce travail de bachelor était de développer un prototype d'application web capable de synchroniser plusieurs clients web de manière efficace. Pour atteindre cet objectif, nous avons adopté une démarche structurée et méthodique, articulée autour de plusieurs étapes clés.

Dans un premier temps, nous avons réalisé une analyse approfondie d'un projet similaire existant. Cette étude comparative nous a permis d'identifier les solutions déjà proposées, de comprendre leurs forces et faiblesses, et de définir les caractéristiques uniques que notre application pourrait apporter. Parallèlement, nous avons élaboré les cas d'utilisation pertinents, ce qui a facilité la modélisation précise de la base de données nécessaire pour supporter les fonctionnalités envisagées.

Ensuite, nous avons développé le prototype en détaillant chaque fonctionnalité à travers des figures et des exemples concrets. Chaque page de l'application a été décrite minutieusement, expliquant non seulement son apparence et son interface utilisateur, mais aussi son fonctionnement interne et son rôle dans l'écosystème global de l'application. Cette approche détaillée a permis de mettre en lumière le parcours utilisateur et de garantir une cohérence dans l'expérience offerte.

Dans la phase suivante, nous avons analysé les technologies utilisées pour la conception du prototype. Après avoir étudié divers frameworks web, nous avons opté pour React. Ce choix s'est basé sur la richesse des ressources disponibles, la qualité de la documentation, et la forte communauté de développeurs soutenant le framework. Nous avons illustré la structure générale du projet à l'aide de blocs de code et de schémas, ce qui a permis de clarifier l'architecture mise en place et les interactions entre les différents composants du front-end.

Concernant le back-end, nous avons exploré une architecture distribuée en utilisant Go et Node.js. Cette combinaison technologique a été choisie pour tirer parti des performances de Go en matière de traitement concurrent et de la flexibilité de Node.js pour la gestion des services web. Nous avons présenté la structure du back-end à travers plusieurs diagrammes et extraits de code, détaillant les mécanismes de communication, la gestion des données, et les protocoles de synchronisation entre les clients.

Il est important de reconnaître que, bien que fonctionnelle, notre implémentation présente des limites inhérentes à la portée et au cadre temporel d'un travail de bachelor. Certaines

optimisations et fonctionnalités avancées n'ont pas pu être intégrées dans cette version du prototype. Néanmoins, le travail accompli fournit une base solide pour de futures améliorations et extensions.

Dans la section suivante, nous aborderons les perspectives d'amélioration et les orientations potentielles pour le développement ultérieur de cette application. Nous discuterons des optimisations possibles, des fonctionnalités supplémentaires envisageables, et des pistes pour renforcer la robustesse et la scalabilité du système.

5.2. Perspectives d'amélioration

Dans le prolongement de ce travail de bachelor, plusieurs axes d'amélioration peuvent être envisagés pour optimiser le prototype développé.

Premièrement, l'intégration de tokens d'authentification pour le microservice représente une amélioration notable, en particulier pour l'utilisation sur des appareils mobiles. Si cette implémentation n'apporterait pas de changements significatifs pour un utilisateur sur ordinateur, elle améliorerait considérablement l'expérience mobile. En effet, lorsque l'écran d'un téléphone portable s'éteint, la connexion WebSocket est interrompue, ce qui peut perturber le déroulement du jeu ou de l'application. L'utilisation de tokens permettrait de maintenir l'état de la session et de rétablir plus facilement la connexion, offrant ainsi une meilleure continuité d'utilisation sur les dispositifs mobiles.

Deuxièmement, le renforcement de la sécurité des données utilisateur est essentiel, surtout dans la perspective d'une mise en production. Actuellement, le cryptage des mots de passe n'est pas implémenté dans le front-end, car le prototype n'est pas destiné à un environnement de production immédiat. Toutefois, il serait indispensable d'ajouter une fonction de cryptage dans le code React pour protéger les informations sensibles des utilisateurs. L'adoption de protocoles de sécurité standards, tels que le hachage des mots de passe avec des algorithmes éprouvés, contribuerait à sécuriser l'application contre les vulnérabilités potentielles et les tentatives d'accès non autorisées.

Enfin, l'aspect esthétique et ergonomique du front-end pourrait être amélioré pour offrir une expérience utilisateur plus agréable et intuitive. Bien que le design actuel soit fonctionnel, une collaboration avec un designer professionnel permettrait d'optimiser l'interface utilisateur. Des améliorations pourraient inclure une meilleure organisation visuelle, l'utilisation cohérente de palettes de couleurs, et l'application de principes de design centrés sur l'utilisateur. Cela augmenterait non seulement l'attrait visuel de l'application, mais également sa facilité d'utilisation et son adoption par un public plus large.

En conclusion, ces perspectives d'amélioration visent à renforcer la performance, la sécurité et l'attrait de l'application. En les mettant en œuvre, le prototype serait mieux préparé pour une éventuelle mise en production et une utilisation à plus grande échelle, répondant ainsi aux exigences des utilisateurs et des standards actuels de développement web.

Lien vers le projet

Le code source et les documents relatifs à ce travail de bachelor sont disponibles sur GitHub : https://github.com/agkkaya321/unifrTB

A

License of the Documentation

References

[1] Bluebird International. Most popular web frameworks in 2023, 2023. Consulté en septembre 2024. 27, 28

[2] Cristian Bocutiu. *React.js Essentials*. Springer International Publishing, 2020. 15

[3] Peter P. S. Chen. The entity-relationship model : Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1) :9–36, 1976. 13

[4] DigitalOcean Community. Importing packages in go. `https://www.digitalocean.com/community/tutorials/importing-packages-in-go`, 2024. Accessed : 2024-09-17. 53

[5] SQLite Consortium. Sqlite - self-contained, serverless, zero-configuration, transactional sql database engine. `https://www.sqlite.org/`, 2024. Accessed : 2024-09-11. 26

[6] Go Developers and Gorilla Web Toolkit Contributors. Go with gorilla - web toolkit for go. `https://www.gorillatoolkit.org/`, 2024. Accessed : 2024-09-11. 26, 44

[7] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, Boston, 2015. 52

[8] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices : Migration of a mission critical system. In *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW 2017)*, pages 29–36, 2017. 45

[9] Facebook. React github issue #26713. `https://github.com/facebook/react/issues/26713`, 2024. Accessed : 2024-09-17. 56

[10] Inc. Facebook. React - a javascript library for building user interfaces (jsx). `https://reactjs.org/docs/introducing-jsx.html`, 2024. Accessed : 2024-09-11. 26

[11] Node.js Foundation. Node.js official website, 2024. Consulté en septembre 2024. 31, 44

[12] Node.js Foundation and OpenJS Foundation. Node.js with express - web application framework for node.js. `https://expressjs.com/`, 2024. Accessed : 2024-09-11. 26, 44

[13] Free Software Foundation. The gnu free documentation license. `https://www.gnu.org/licenses/fdl.html`. [Accessed : 23-Sep-2024].

[14] Google. The go programming language. `https://golang.org`, 2024. `https://golang.org`. 44

[15] Kahoot. Site web de kahoot. `https://kahoot.com`, 2024. Consulté le 29 août 2024. 5

[16] Jason Korban. *Real-time Web Application Development Using Node.js*. Packt Publishing Ltd., 2013. 45

[17] Dmitry Kozlovski et al. Analyzing the limitations of node.js in high-performance computing. *ACM Transactions on Internet Technology (TOIT)*, 20(3) :1–24, 2020. 46

[18] Andreas Meier. *Introduction pratique aux bases de données relationnelles*. Presses polytechniques et universitaires romandes, Lausanne, deuxième édition edition, 2006. 13

[19] Inc. Meta Platforms. React - a javascript library for building user interfaces. `https://react.dev`, 2024. Accessed : 2024-09-11. 30

[20] Sam Newman. *Building Microservices : Designing Fine-Grained Systems*. O'Reilly Media, 2015. 45

[21] Mark Otto and Jacob Thornton. Bootstrap. `https://getbootstrap.com/`, 2011. Accessed : 2024-09-05. 14

[22] Praveen Kumar Raj and Chandrasekhar Annavarapu. Node.js for scalable internet of things applications. In *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 197–201. IEEE, 2018. 45

[23] SVGRepo. Svgrepo - free svg vectors and icons. `https://www.svgrepo.com`, 2024. Accessed : 2024-09-09. 24

[24] Stefan Tilkov and Steve Vinoski. Node.js : Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6) :80–83, 2010. 45

[25] Thomas Tkrotoff. Front-end frameworks comparison. `https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190`, 2019. Consulté en septembre 2024. 27, 28

[26] Université de Neuchâtel. Chatgpt-4 : Avantages et limites de l'utilisation dans la recherche et l'écriture, 2024. Consulté le 17 septembre 2024. 3

[27] Jordan Walke. React : A javascript library for building user interfaces. `https://reactjs.org/`, 2013. Accessed : 2024-09-05. 14

[28] Evan You. Vite official website, 2024. Consulté en septembre 2024. 31