Protocole de communication

Version 1.0

Historique des révisions

Date	Version	Description	Auteur
2023-03-17	1.0	Commencer l'introduction et la communication client-serveur	Chowdhury, Rasel
2023-03-18	1.0	Ajout de détails, modification et finalité de l'introduction et de la communication client-serveur.	Zahreddine, Nour
2023-03-19	1.0	Description des paquets	Zahreddine, Nour
2023-03-20	1.0	Vérification de tout le document en équipe et dernières modifications pour obtenir la version final	Banna, Michael Chowdhury, Rasel Doghri, Aziz Greige, Jason Moukheiber, Éric Zahreddine, Nour

Table des matières

1. Introduction 4
2. Communication client-serveur 4
3. Description des paquets 5

Protocole de communication

1. Introduction

Le protocole de communication assure la connexion entre le client d'une application et son serveur. Dans le cadre de notre projet, les protocoles HTTP et WebSocket sont utilisés comme moyens de communication entre le client et le serveur, fonctionnant de manières distinctes. Effectivement, la performance du projet repose grandement sur le protocole employé : les WebSockets sont mis en œuvre pour l'envoi continu de données, tandis que l'envoi de données via HTTP s'effectue sur demande. Ce document abordera ces deux protocoles, leurs cas d'utilisation respectifs et les avantages comparatifs de chaque approche. De plus, nous discuterons également des divers paquets échangés lors de la communication, ainsi que des informations transmises et reçues à travers ces paquets.

2. Communication client-serveur

Au cours du projet, la communication entre le client et le serveur est assurée par des requêtes HTTP et l'utilisation de WebSockets. Le protocole HTTP est employé de manière limitée, étant donné qu'il nécessite une requête chaque fois que le client et le serveur doivent échanger des informations. Les Sockets sont privilégiés pour les requêtes importantes, qui requièrent un envoi et une réception de données immédiats dès leur disponibilité.

Dans ce projet, les WebSockets sont utilisés pour la messagerie et le chat en ligne. En effet, il est important d'assurer une communication instantanée entre le client et le serveur lors de l'envoi ou la réception de messages. Cette méthode offre plusieurs bénéfices, tels que la rapidité des messages transmis en temps réel et une communication directe entre le serveur et le client. D'autre part, la mise en place d'un jeu 1v1, y compris les salles d'attente, est gérée par le protocole WebSocket. Ce dernier permet de regrouper divers joueurs et de les placer dans une salle d'attente simultanément. Les joueurs doivent également être en mesure de visualiser les autres participants présents dans la salle d'attente, une information fournie directement par le serveur sans qu'une demande ne soit nécessaire. Plusieurs autres aspects du projet exploitent le protocole WebSocket, comme l'envoi d'images depuis le serveur vers le client, la synchronisation de l'écran de jeu requérant une communication continue entre le serveur et le client lorsqu'une différence est détectée, la réinitialisation des chronomètres de jeu et l'affichage permanent du nom du joueur ayant le meilleur score. En revanche, le protocole HTTP est principalement utilisé pour les éléments stockés dans la base de données, car ces informations ne doivent pas toujours être récupérées et sont sollicitées lors d'une requête du client vers le serveur. Cela concerne notamment l'historique des parties, qui est chargé seulement lorsqu'une requête client est adressée au serveur.

3. Description des paquets

Games.service

get collection()

Dans la classe GamesService, la méthode collection() est définie comme un getter qui renvoie la collection de jeux contenue dans la base de données. La méthode va utiliser l'objet dbService pour accéder à la base de données et récupérer la collection de jeux grâce à this.dbService.db.collection(DB_CONSTS.DB_COLLECTION_GAMES).

async getAllGames()

La méthode getAllGames() est utilisée pour récupérer tous les jeux de la base de données, afin de renvoyer une réponse en format JSON au client. La classe GamesController sera appelée permettant de gérer la requête du client au serveur en attribuant le path GET/ grâce à la méthode this.router.get('/', async (req: Request, res: Response).

Async getGameById(id : unknown)

La méthode getGameById() est utilisée pour récupérer un jeu de la base de données en fonction de son id et le renvoyer sous forme de réponse JSON au client. La classe GamesController sera appelée permettant de gérer la requête du client au serveur en attribuant le path GET/:id grâce à la méthode this.router.get('/:id', async (req: Request, res: Response). La méthode collection() sera aussi utilisée pour accéder à la collection de jeux contenant le jeu que l'on désire en utilisant findOne(). Finalement, le jeu sera récupéré sous la forme d'un fichier JSON.

Async CreateGame(gameInfo : CreateGame)

La méthode createGame() est utilisée pour créer un nouveau jeu et l'enregistrer dans la base de données. La classe GamesController sera appelée permettant de gérer la requête du client au serveur en attribuant le path POST/ grâce à la méthode this.router.post('/', async (request, response). D'autre part, la méthode contient une fonction randomUUID qui est utilisée afin de générer les différences pour les enregistrer grâce aux méthodes contenues dans diff.service(saveImages() et findDifferences()). Les nombreuses méthodes employées vont nous permettre de créer le jeu en fonction du nombre de différences, en établissant par exemple la difficulté du jeu. La méthode collection() sera aussi utilisée pour accéder à la collection de jeux pour enregistrer l'objet de jeu créé en utilisant insertOne(). L'objet jeu sera ensuite envoyé en réponse JSON au client.

async deleteGame(id: string)

La méthode deleteGame() est utilisée pour supprimer un jeu de la base de données en fonction du paramètre id. La classe GamesController sera appelée permettant de gérer la requête du client au serveur en attribuant le path DELETE/:id grâce à la méthode this.router.delete('/:id', async (request, response). La méthode collection() sera aussi utilisée pour accéder à la collection de jeux afin de supprimer l'objet jeu d'après son id en utilisant findOneAndDelete(). L'objet jeu sera ensuite envoyé en réponse JSON au client.

async validateCoords(id: string, x: number, y: number, found: string[] = [])

La méthode validateCoords() est utilisée pour valider si une coordonnée fait partie des différences d'image d'un jeu spécifique dans la base de données. La classe GamesController sera appelée permettant de gérer la requête du client au serveur en attribuant le path GET/validate grâce à la this.router.get('/validate', async (req: Request, res: Response). Les paramètres que prend validateCoords() doivent correspondre aux informations qui sont dans la base de données afin que ce soient des différences. La méthode get collection() sera aussi utilisée pour accéder à la collection de jeux afin de chercher le jeu à partir des coordonnées. Si ceux-ci sont trouvés dans la base de données, on envoie une réponse JSON au client. La réponse sera différente dépendamment de si oui ou non, les différences sont trouvées.

async connectToServer(uri: any)

La méthode connectToServer()permet de se connecter au serveur MongoDB à l'aide de l'URI. Le but de cette méthode est de pouvoir initialiser une nouvelle instance MongoClient en prenant en paramètre l'URI. On appelle ensuite la méthode connect() pour établir une connexion avec le serveur MongoDB. Lorsque la connexion est établie, la base de données sera définie sur l'instance de base de données du client.

Socket manager

handleIter(room: string, cb: (sock: any) => void)
cancelFromClient = async (data: any)

La méthode handleIter() prend un ID de salle et un callback en paramètre. Elle s'occupe de récupérer tous les sockets de la salle de jeu et les itère, en appelant le callback de rappel fourni pour chaque socket. Nous avons aussi la méthode cancelFromClient() qui s'occupe de l'annulation d'un jeu côté client. Elle prend un objet data en paramètre. La méthode fonctionne grâce à handleIter() qui va parcourir les sockets de la salle de jeu et va émettre l'événement 'player-refuse', causant la suppression des sockets se trouvant dans la salle de jeu. Elle va ensuite émettre l'événement 'update-game-button', mettant à jour l'état du jeu dans la base de données. En effet, isGameOn deviendra false.

handleSocket : void

- 'connexion' a lieu lorsqu'un client se connecte au serveur à l'aide d'un websocket. Une fois que le client se connecte, celui-ci est capable d'envoyer et recevoir des messages du serveur grâce à l'objet socket qui a été créé.

- 'create-game' a lieu lorsqu'un joueur crée un nouveau jeu. Le serveur émet l'événement 'update-game-button' aux clients, mettant ainsi 'gameOn' à vrai, marquant le début de la partie. Ensuite, le jeu est immédiatement ajouté à la base de données et le joueur peut commencer à jouer.

- 'join-game' a lieu lorsqu'un joueur rejoint une partie. Il est ensuite placé dans la salle d'attente et le serveur émet l'événement pour que les autres clients soient au courant qu'il rejoint la salle.

- 'join-approval' a lieu lorsqu'un joueur accepte ou refuse une invitation à jouer. Si le joueur accepte, le serveur crée un ID et retire les autres joueurs pour les placer dans une autre salle. Le serveur n'oublie pas d'ajouter le jeu dans la base de données et émet un événement de début de partie pour commencer à jouer. Si le joueur refuse, le serveur émet un événement refusant le joueur et le retire de la salle d'attente.

- 'cancel-from-client' a lieu lorsqu'un joueur annule une partie côté client. Le serveur émet un événement annulant la partie et retire tous les joueurs de la page de jeu. Le serveur met ensuite la base de données à jour et gameOn devient false.

- 'cancel-from-joiner' a lieu lorsqu'un joueur voulant rejoindre une partie l'annule. Le serveur émet un événement d'annulation et retire tous les joueurs de la page de jeu.

- 'abandon-game' a lieu lorsqu'un joueur abandonne la partie. Le serveur émet un événement d'abandon de l'adversaire à l'autre joueur et retire tous les joueurs de la page de jeu.

- 'game-deleted' a lieu lorsqu'un jeu est supprimé. Le serveur émet un événement qui supprime le jeu à tous les clients et retire tous les joueurs de la page de jeu.

- 'game-end' a lieu lorsqu'une partie se termine. Le serveur retire tous les joueurs de la page de jeu.

- 'difference-found' a lieu lorsqu'un joueur trouve une différence dans le jeu. Le serveur émet un événement 'notify-difference-found' à l'autre joueur et affiche les coordonnées de différence.

- 'difference-error' a lieu lorsqu'un joueur fait une erreur en essayant de trouver une différence. Le serveur émet un événement de notification d'erreur de différence aux joueurs.

- 'room-message' a lieu lorsqu'un joueur envoie un message dans la page de jeu grâce au clavardage. Le serveur émet un événement 'room-message' à l'autre joueur contenant le message.
- 'erase-game-history' a lieu lorsqu'un joueur supprime les historiques de jeu. Le serveur émet un événement qui efface l'historique.

- 'déconnexion' a lieu lorsqu'un joueur se déconnecte du serveur.

Nous avons plusieurs composantes dans le coté client utilisant des web sockets.

Game-page

Tout d'abord, nous avons la méthode configureMultiCreatorEvents() qui s'occupe de la configuration des listeners d'événements pour les jeux 1v1. Cette méthode configure les listeners d'événements pour qu'ils puissent écouter 'player-refuse', 'game-started', 'game-created', 'player-joined', 'cancel-from-joiner' et 'enemy-abandon', qui ont été décrits dans handleSocket(). Ensuite, la méthode abandonGame() envoie un événement 'abandon-game' au serveur lorsque le joueur quitte le jeu et getPlayerName() renvoie le nom du joueur par son id.

SocketClientService

La première méthode est socket() utilisée pour simplement stocker l'instance de socket. Ensuite, gameRoomId() est utilisée pour identifier la salle de jeu en temps réel grâce à son id, isSocketAlive() vérifie si le socket est actif, la méthode connect() crée une instance de socket et se connecte au serveur à l'aide de l'URL et disconnect() déconnecte le socket du serveur. De plus, removeAllListeners() supprime tous les listeners de l'instance de socket, on<T>(event: string, action: (data: T) => void) attache un listener à l'instance de socket pour un événement en particulier et écoute lorsque l'événement est déclenché, off<T>(event: string, action: (data: T) => void) supprime un listener de l'instance de socket pour un événement en particulier et send<T>(event: string, data?: T) envoie un message au serveur en fonction de l'événement. Des données sont envoyées avec le message si elles sont présentes, car les données sont facultatives.

Local-message.service

Le LocalMessagesService est un service que l'on utilise pour le stockage des messages générés par le jeu dans un tableau local, comme les messages qui apparaissent lorsque le joueur trouve une différence. Dans GamePageComponent, le SocketClientService est injecté et utilisé pour configurer des listeners d'événements, afin d'envoyer des messages (par exemple : un message sera envoyé lorsque le joueur trouve une différence ou encore lorsqu'un joueur abandonne la partie).

Game-item

La première méthode configureBaseSocketFeatures() utilise les fonctionnalités du socket, tel que l'événement 'update-game-button' mettant à jour la partie. En deuxième lieu, nous avons createGame() qui crée une nouvelle partie en mode solo, nous avons aussi deleteGame() qui fait en sorte que l'on peut supprimer la partie donc lorsque le client approuve la suppression du jeu, cela envoie une requête "delete" au serveur et envoie un événement pour supprimer le jeu au socket. Ensuite, nous avons createMultiGame() créant une nouvelle partie 1v1, lorsque le joueur confirme la création du jeu, un événement 'create-game' est envoyé au socket, nous avons joinMultiGame() qui permet au joueur de rejoindre une partie multijoueur existante, envoyant un événement "join-game" au serveur de socket.

Play-area

Dans play-area, nous avons tout d'abord le constructeur qui se connecte au service de socket, dans notre cas, la méthode handleDifferenceNotifications() va écouter "notify-difference-found" et "notify-difference-error" émis par le service de socket, en appelant incrementDifferences() et addMessage(). En dernier lieu, mouseHitDetect() détecte si le bouton de la souris a été cliqué et fait un appel au serveur si une différence a été trouvée. Si le serveur lui répond qu'une différence a été trouvée, la différence clignote et envoie l'événement ‘difference-found’ au serveur, ce qui appelle incrementDifferences(). Dans le cas contraire, un message d'erreur apparaît et l'événement "difference-error" est envoyé au serveur, ce qui appelle addMessage().

Messages

Tout d'abord, le constructeur initialise localMessages et utilise un listener pour recevoir des messages grâce à socketClientService. Les méthodes reliées au socket dans la composante messages sont entre autres sendMessage(), qui émet un événement de sortie représentant le message. Tout comme le constructeur, receiveChatMessage() configure un listener pour les messages grâce à socketClientService et ajoute le message dans localMessages. Finalement, sendChatMessage envoie un message au serveur grâce à socketClientService, pour être ensuite ajouté au localMessages.