LOG2990-311

Protocole de communication

Version 1.0

Historique des révisions

[Dans l'historique de révision, il est nécessaire d'écrire le nom du ou des auteurs ayant travaillé sur chaque version.]

Date	Version	Description	Auteur
2023-03-21	1.0	Écriture complète du document.	Jérôme Chabot, Simon-Alexandre Bastin, Wassim Hamadache, Marsel Bakashov

Table des matières

Protocole de communication

1. Introduction

Ce protocole de communication a pour objectif de présenter la structure, le format et les règles de transmission de données utilisées dans notre application web. Notre projet repose sur une communication client-serveur, qui s'appuie sur deux protocoles distincts : HTTP et WebSocket. Le protocole HTTP étant ponctuel et unidirectionnel, il a surtout été utilisé pour la gestion de fonctionnalités principales et pour les interactions simples avec les utilisateurs. Tandis que WebSocket est une communication bidirectionnelle en temps réel entre le client et le serveur. Cela en fait une technologie adaptée aux fonctionnalités de chat en direct, aux jeux multijoueurs et aux autres applications nécessitant une communication en temps réel entre les utilisateurs. Cette communication permet de répondre aux besoins fonctionnels spécifiques de notre projet, en permettant notamment la gestion de parties multijoueurs et la mise en place d'un service de clavardage. Dans ce document, nous allons décrire en détail les choix de communication que nous avons faits, ainsi que les raisons de ces choix. Nous présenterons également les fonctionnalités déjà implémentées dans les Sprints 1 et 2, ainsi que les fonctionnalités à venir dans le Sprint 3 à travers le contenu des différents types de paquets utilisés.

2. Communication client-serveur

D'abord, pour l'ensemble des fonctionnalités requises au premier sprint, nous avons utilisé le protocole HTTP. Du côté client, les fonctions sont présentes dans *communication.service*. Du côté serveur, les différents contrôleurs interprètent les requêtes.

La gestion de la minuterie, la gestion des jeux existants, comme l'ajout et la suppression des jeux ainsi que les requêtes pertinentes pour l'algorithme de détection des différences se font avec le protocole HTTP. Cela inclut notamment la sauvegarde d'images sur le serveur. De plus, le chargement des informations au début d'une partie en solo sont faites grâce au même protocole. Enfin, lors d'un clic pendant une partie, une requête est faite auprès du serveur pour vérifier si la coordonnée fait partie d'une différence, et toutes les coordonnées faisant partie de ladite différence sont retournées si c'est le cas.

En revanche, le protocole WebSocket a été utilisé pour la messagerie ainsi que pour la gestion d'une partie multijoueur.

En ce qui concerne le service de chat, on a décidé d'utiliser la communication bidirectionnelle des websocket pour gérer les différents événements en lien avec le clavardage. Nous avons un service, ChatService, qui s'occupe d'émettre les différents événements lors de différents actionnements. Parmi ces événements, nous pouvons retrouver un abandon d'une partie, un message d'utilisateur, un message du système, joindre une partie, etc. Le serveur reçoit ce emit et s'occupe à son tour de renvoyer l'information nécessaire au client, pour qu'il puisse mettre à jour l'instance nécessaire et l'afficher.

Pour la gestion de partie multijoueur, nous utilisons un service nommé WaitingRoomService qui s'occupe d'appeler de créer les écouteurs et émettre les messages au serveur avec un autre service, SocketClientService, qui gère le websocket. WaitingRoomService est un service qui permet de regrouper tous les appels nécessaires au serveur par rapport à la salle d'attente. Ce service permet notamment de savoir si l'utilisateur créé ou rejoint la partie, si la partie est pleine, etc.

Au niveau du jeu nous utilisons un service nommé MultiplayerService qui fait usage de communication bidirectionnelle par websockets afin de s'occuper des événements émis par les joueurs en multijoueur. MultiplayerService permet, entre autres, de détecter les erreurs et d'abandonner une partie. Le service renvoie l'information au client afin de mettre à jour l'instance.

3. Description des paquets

HTTP

Pour les requêtes HTTP, nous avons séparé les requêtes en 4 contrôleurs. Toutes les requêtes ont le préfixe *http://localhost:3000/api*. Pour des raisons de simplicité, seuls les suffixes sont indiqués dans le présent rapport.

D'abord, *algo.controller* contient une grande partie des requêtes. Les suffixes de requêtes pour ce contrôleur commencent toutes par 'algo-controller'. Ce début de suffixe n'est pas spécifié dans les explications reliées à ce contrôleur pour faciliter la lecture.

La fonction *getImage* est exécutée lorsqu'une requête GET avec le suffixe '/image/:name' est lancée. Le paramètre 'name' est de type *string*, et il désigne le nom que l'image porte dans les fichiers du serveur. Dans notre application, toutes les images ont le même nom que leur ID, alors l'ID de l'image est fourni en chaîne de caractères pour obtenir la bonne image. Cette requête est particulièrement utilisée pour montrer les jeux existants dans les menus, alors l'URL de l'image de gauche (ou image originale) est retournée dans la réponse.

La fonction *getAllImages* est exécutée lorsqu'une requête GET avec le suffixe '/image' est lancée. Cette fonction est similaire à la précédente, mais celle-ci renvoie toutes les images de gauche (ou originales) existantes. Plus précisément la réponse est un objet contenant 'data', un tableau d'URLs d'images, et 'games', un tableau contenant les informations pertinentes d'un jeu. L'interface utilisée pour les informations d'un jeu contient les attributs suivants: 'gameId', 'gameName', 'gameDifficulty', 'numberOfDiff', 'bestTime1', 'bestTime2' et 'bestTime3'.

La fonction *deleteTemps* est exécutée lorsqu'une requête DELETE avec le suffixe '/temp' est lancée. Lorsque nous lançons la validation de différences dans la création d'un jeu, des images sont enregistrées dans un répertoire temporaire contenant l'image de gauche, de droite, l'image de différences ainsi que le coordonnées des différences. En lançant cette requête, ces fichiers sont supprimés. Il n'y a pas de réponse associée à cette requête.

La fonction *deleteReal* est exécutée lorsqu'une requête DELETE avec le suffixe 'real/:gameId' est lancée. Le paramètre gameId est une chaîne de caractères du ID du jeu à être supprimé. Cette requête est utilisée pour supprimer du serveur un jeu existant. Il n'y a pas de réponse associée à cette requête.

La fonction *save* est exécutée lorsqu'une requête PATCH avec le suffixe 'save' est lancée. Le corps de la requête contient le nom du jeu, la difficulté associée au jeu ainsi que le nombre de différences. Cette requête transfère les informations des répertoires temporaires pour sauvegarder les jeux dans les répertoires réguliers pour les jeux. Il n'y a pas de réponse associée à cette requête.

La fonction *validate* est exécutée lorsqu'une requête POST avec le suffixe 'validate' est lancée. Le corps de la requête est un objet de l'interface *ImgData*, contenant l'URL de l'image de gauche et de droite, le nom du jeu, ainsi que le rayon d'élargissement choisi. Cette fonction est la plus importante de l'algorithme de détection de différences. Elle retourne un objet contenant l'URL de l'image de différences, la difficulté ainsi que le nombre de différences dans le jeu reçu.

La fonction *getModifiedPixels* est exécutée lorsqu'une requête GET avec le suffixe '/pixels/:gameName' est lancée. Le paramètre *gameName* est une chaîne de caractères contenant le nom du jeu dans les fichiers du serveur. Comme précisé plus tôt, ce nom de fichier est équivalent à l'ID du jeu. Cette fonction obtient les pixels des images de gauche et de droite sous la forme d'un tableau de 3 dimensions, la 3e dimension étant les valeurs RGBA de chaque pixel. Cette requête donne une réponse contenant les pixels de l'image de gauche et de droite.

Ensuite, *app-router.controller* contient quelques requêtes plus générales. Les suffixes de requêtes pour ce contrôleur commencent toutes par ‘app-router-controller’. Ce début de suffixe n’est pas spécifié dans les explications reliées à ce contrôleur pour faciliter la lecture.

La fonction *getMistake* est exécutée lorsqu’une requête POST avec le suffixe 'mistake' est lancée. Le corps de la requête est un objet suivant l’interface Attempt, qui contient le nom du jeu du côté serveur (qui est donc l’ID), ainsi que les coordonnées X et Y d’un clic. Cette requête est lancée pendant les parties lorsqu’un clic est effectué par l’utilisateur. Cette requête fournit une réponse qui est soit un tableau de coordonnées ou bien *null*. Si l’endroit du clic correspond à une différence du jeu spécifié, toutes les coordonnées de cette différence sont retournées. Si ce n’est pas le cas, la réponse contient *null*.

La fonction *getMistakes* est exécutée lorsqu’une requête GET avec le suffixe 'mistake/:id' est lancée. Le paramètre *id* est une chaîne de caractères correspondant au nom du jeu côté serveur (qui est donc l’ID). Cette fonction retourne simplement un tableau contenant les différences. Les différences sont des tableaux de coordonnées. La réponse de cette requête est donc un tableau de deux dimensions de coordonnées.

Pour poursuivre, *game-card.controller* contient une requête pour charger des jeux. Les suffixes de requêtes pour ce contrôleur commencent toutes par ‘game-card-controller’. Ce début de suffixe n’est pas spécifié dans les explications reliées à ce contrôleur pour faciliter la lecture.

La fonction *getGameCardTest* est exécutée lorsqu’une requête GET avec le suffixe '/:id' est lancée. Le paramètre *id* est une chaîne de caractères correspondant au nom du jeu côté serveur (qui est donc l’ID). Cette fonction obtient les informations d’un jeu en fonction de son ID. La réponse de cette requête est un objet de type *GameInformation* contenant les attributs suivants: ‘gameId’, ‘gameName’, ‘gameDifficulty’, ‘numberOfDiff’, ‘bestTime1’, ‘bestTime2’ et ‘bestTime3’, tel que spécifié ci-haut.

Enfin, *timer.controller* contient des requêtes pertinentes à l’utilisation de la minuterie. Les suffixes de requêtes pour ce contrôleur commencent toutes par ‘timer-controller’. Ce début de suffixe n’est pas spécifié dans les explications reliées à ce contrôleur pour faciliter la lecture.

La fonction *getTimerId* est exécutée lorsqu’une requête GET avec le suffixe '/id' est lancée. Cette fonction génère une instance de minuterie et retourne l’ID de la minuterie créée.

La fonction *getTimer* est exécutée lorsqu’une requête GET avec le suffixe '/:id' est lancée. La paramètre *id* correspond à l’ID de la minuterie à récupérer. La minuterie en question est dans la réponse de cette requête.

La fonction *deleteTimer* est exécutée lorsqu’une requête DELETE avec le suffixe '/:id' est lancée. La paramètre *id* correspond à l’ID de la minuterie à supprimer. Cette fonction arrête la minuterie ayant comme ID le paramètre fourni. Aucune réponse n’est attendue de cette fonction.

Sprint 3

La fonction *deleteAllReal* est exécutée lorsqu’une requête DELETE avec le suffixe 'real' est lancée. Cette requête est utilisée pour supprimer du serveur tous les jeux existants. Il n’y a pas de réponse associée à cette requête.

La fonction *deleteAllBestTimes* est exécutée lorsqu’une requête DELETE avec le suffixe 'bestTimes' est lancée. Cette requête est utilisée pour supprimer du serveur les meilleurs temps pour tous les jeux. Il n’y a pas de réponse associée à cette requête.

La fonction *deleteGameHistory* est exécutée lorsqu’une requête DELETE avec le suffixe 'gameHistory' est lancée. Cette requête est utilisée pour supprimer du serveur l’historique des parties. Il n’y a pas de réponse associée à cette requête.

La fonction *addGameToHistory* est exécutée lorsqu'une requête PATCH avec le suffixe 'gameHistory' est lancée. Le corps contient un objet suivant l'interface *gameHistoryInstance*. Cette interface contient la date et l'heure du début de la partie, la durée totale de la partie, le mode , le nom du premier joueur (ou du joueur solo), le nom du deuxième joueur (si c'est une partie à deux), le gagnant de la partie (si c'est une partie à deux), le temps utilisé pour gagner (si applicable). La fonction met aussi à jour le meilleur temps si le temps de cette partie fait partie du podium. Cette requête est utilisée pour ajouter une entrée à l'historique des parties. Il n'y a pas de réponse associée à cette requête.

WebSocket

Client

Le protocole WebSocket a notamment été utilisé pour la gestion de la boîte de clavardage via le ChatService. Ce dernier s'occupe de gérer les différents événements du chat et il contient quelques fonctions spécifiques au chat.

La fonction *sendUserMessage* prend en paramètre un message et une couleur. Cette fonction permet d'émettre un objet { message: string, color: string, isSystem: false } à travers le sujet RxJS messageSource pour envoyer un message d'utilisateur. Cette fonction est appelée lorsque l' utilisateur valide une entrée via le component chat-box. L'attribut color est toujours attribué au noir, mais il a été gardé, car c'était pertinent de garder la possibilité de changer la couleur selon l'utilisateur pour des implementations futures.

La fonction *sendSystemMessage* prend en paramètre un message et une couleur. Cette fonction émet un objet { message: string, color: string, isSystem: true } à travers le sujet RxJS messageSource pour envoyer un message système. Les messages systèmes sont différenciés des messages utilisateurs, car leur apparence est différente, ils sont gras. Cette fonction est appelée au différents événements du système, comme l'utilisation d'un indice ou encore la trouvaille d'une différence. L'attribut couleur permet d'utiliser une couleur différente pour chaque événement.

La fonction *joinRoom* prend en paramètre un nom de salle. Cette fonction émet l'événement ChatEvents.JoinRoom via la connexion socket avec le nom de salle fourni, pour permettre à un utilisateur de rejoindre une salle de discussion.

La fonction *sendRoomMessage* prend en paramètre le nom de la salle, un message et le type d'expéditeur. Cette fonction émet l'événement ChatEvents.RoomMessage via la connexion socket avec les informations fournies, pour permettre à un utilisateur d'envoyer un message à une salle de discussion. Le type de l'expéditeur permet de savoir si on a affaire à un joueur ou au système pour savoir quelle fonction *sendMessage* il faudra utiliser.

La fonction *removeDifference* prend en paramètre un tableau de coordonnées. Cette fonction émet les coordonnées fournies via le sujet RxJS foundDifference pour permettre de supprimer les différences trouvées de l'image.

La fonction *removeHint* ne prend pas de paramètres. Cette fonction émet une notification vide via le sujet RxJS useHint pour permettre de supprimer un indice.

La fonction *sendRemoveData* prend en paramètre le nom de la salle et un tableau de coordonnées. Cette fonction émet l'événement ChatEvents.FoundDifference via la connexion socket avec les informations fournies, pour permettre à l'utilisateur de supprimer les différences trouvées de l'image pour tous les autres utilisateurs de la salle de discussion.

La fonction *sendUsername* prend en paramètre le nom de la salle et le nom d'utilisateur. Cette fonction émet l'événement ChatEvents.GetUsernameAdversary via la connexion socket avec les informations fournies, pour permettre à l'utilisateur d'obtenir le nom de l'adversaire dans la salle de discussion.

La fonction *sendRemoveHiunt* prend en paramètre le nom de la salle. Cette fonction émet l'événement

ChatEvents.removeHint via la connexion socket avec les informations fournies, pour permettre à d'envoyer un évènement d'utilisation d'indice au serveur.

La fonction *sendSurender* prend en paramètre le nom de la salle. Cette fonction émet l'événement ChatEvents.Surender via la connexion socket avec le nom de la salle fourni, pour permettre à l'utilisateur d'abandonner la partie et de se retirer de la salle de discussion.

La fonction *formatTime* prend en paramètre une date. Cette fonction formate la date fournie au format "(HH:mm:ss)" et retourne une chaîne de caractères pour afficher l'heure de manière conviviale dans les messages de chat.

La fonction *onIsGameCreator* permet de vérifier si l'utilisateur est le créateur de la salle. Elle prend en paramètre le nom de la salle et une fonction de rappel qui prend en paramètre une réponse booléenne. Cette fonction émet l'événement WaitingRoomEvents.IsGameCreator avec le nom de la salle fourni via la connexion socket, et elle attend une réponse de type booléen via le même événement avant d'appeler la fonction de rappel avec cette réponse.

La fonction *onIsRoomFull* permet de vérifier si la salle est pleine. Elle prend en paramètre le nom de la salle et une fonction de rappel qui prend en paramètre une réponse booléenne. Cette fonction émet l'événement WaitingRoomEvents.IsRoomFull avec le nom de la salle fourni via la connexion socket, et elle attend une réponse de type booléen via le même événement avant d'appeler la fonction de rappel avec cette réponse.

La fonction *onOpponentJoinedGame* permet de détecter quand un adversaire a rejoint la salle de discussion. Elle prend en paramètre une fonction de rappel qui prend en paramètre une réponse sous forme de chaîne de caractères. Cette fonction écoute l'événement WaitingRoomEvents.OpponentJoinedGame via la connexion socket, et elle appelle la fonction de rappel avec la réponse lorsque cet événement est émis.

La fonction *onOpponentLeftGame* permet de détecter quand un adversaire a quitté la salle de discussion. Elle prend en paramètre une fonction de rappel qui ne prend pas de paramètre. Cette fonction écoute l'événement WaitingRoomEvents.OpponentLeftGame via la connexion socket, et elle appelle la fonction de rappel lorsque cet événement est émis.

La fonction *onResponseOfCreator* permet de détecter quand le créateur de la salle a répondu à une demande d'invitation. Elle prend en paramètre une fonction de rappel qui prend en paramètre un objet de la forme { creatorAnswer: boolean; gameRoomName: string }. Cette fonction écoute l'événement WaitingRoomEvents.ResponseOfCreator via la connexion socket, et elle appelle la fonction de rappel avec l'objet de réponse lorsque cet événement est émis.

La fonction *onCreatorLeftGame* permet de détecter quand le créateur de la salle a quitté la salle de discussion. Elle prend en paramètre une fonction de rappel qui ne prend pas de paramètre. Cette fonction écoute l'événement WaitingRoomEvents.CreatorLeftGame via la connexion socket, et elle appelle la fonction de rappel lorsque cet événement est émis.

La fonction *onGetCreatorUsername* permet de récupérer le nom d'utilisateur du créateur de la salle. Elle prend en paramètre une fonction de rappel qui prend en paramètre le nom d'utilisateur sous forme de chaîne de caractères. Cette fonction écoute l'événement WaitingRoomEvents.GetCreatorUsername via la connexion socket, et elle appelle la fonction de rappel avec le nom d'utilisateur lorsque cet événement est émis.

La fonction *onGiveCreatorUsername* permet de notifier que l'utilisateur est le créateur de la salle. Elle prend en paramètre une fonction de rappel qui ne prend pas de paramètre. Cette fonction écoute l'événement WaitingRoomEvents.GiveCreatorUsername via la connexion socket, et elle appelle la fonction de rappel lorsque cet événement est émis.

La fonction *onDeleteRoom* prend en paramètre une fonction de rappel qui sera appelée lorsque

l'événement WaitingRoomEvents.DeleteGame sera émis par le serveur. Cette fonction permet de gérer la suppression d'une salle et exécute le rappel fourni.

La fonction *getGameRooms* ne prend pas de paramètres et émet l'événement WaitingRoomEvents.GetGameRooms via la connexion WebSocket pour obtenir la liste des salles disponibles. Le résultat est renvoyé à l'aide de la fonction callbackGetGameRooms.

La fonction *joinGameRoom* prend en paramètre le nom d'une salle et émet l'événement WaitingRoomEvents.JoinGameRoom via la connexion WebSocket pour permettre à un utilisateur de rejoindre une salle .

La fonction *responseOfCreator* prend en paramètres la réponse du créateur de la salle (booléen) et le nom de la salle. Cette fonction émet l'événement WaitingRoomEvents.ResponseOfCreator via la connexion WebSocket avec les informations fournies pour permettre au créateur de la salle de répondre à une demande de participation à la salle.

La fonction *creatorLeftGame* prend en paramètre le nom de la salle. Cette fonction émet l'événement WaitingRoomEvents.CreatorLeftGame via la connexion WebSocket avec le nom de la salle fourni pour informer les autres utilisateurs que le créateur de la salle a quitté la salle.

La fonction *opponentLeftGame* prend en paramètre le nom de la salle. Cette fonction émet l'événement WaitingRoomEvents.OpponentLeftGame via la connexion WebSocket avec le nom de la salle fourni pour informer les autres utilisateurs que l'adversaire a quitté la salle.

La fonction *opponentJoinedGame* prend en paramètres le nom de la salle et le nom d'utilisateur de l'adversaire. Cette fonction émet l'événement WaitingRoomEvents.OpponentJoinedGame via la connexion WebSocket avec les informations fournies pour informer les autres utilisateurs qu'un adversaire a rejoint la salle.

La fonction *leaveGameRoom* prend en paramètre le nom de la salle. Cette fonction émet l'événement WaitingRoomEvents.LeaveGameRoom via la connexion socket avec le nom de la salle fourni, pour permettre à l'utilisateur de quitter la salle.

La fonction *leaveQueueRoom* prend en paramètre le nom de la salle d'attente. Cette fonction émet l'événement WaitingRoomEvents.LeaveQueueRoom via la connexion socket avec le nom de la salle d'attente fourni, pour permettre à l'utilisateur de quitter la salle d'attente.

La fonction *giveCreatorUsername* prend en paramètre le nom de la salle et le nom de l'utilisateur créateur de la salle. Cette fonction émet l'événement WaitingRoomEvents.CreatorUsername via la connexion socket avec les informations fournies, pour permettre à l'utilisateur de recevoir le nom du créateur de la salle.

Serveur

Du côté serveur, le chat.gateway fourni a été utilisé tout en lui ajoutant les @SubscribeMessage et les fonctions additionnelles qui nous étaient nécessaires.

La fonction *broadcastAll* est appelée lorsqu'un client envoie un message à diffuser à tous les clients en utilisant l'événement ChatEvents.BroadcastAll. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le message envoyé par le client. Cette fonction émet un événement ChatEvents.MassMessage à tous les clients connectés (y compris celui qui a initié l'événement) avec le message envoyé.

La fonction *joinRoom* est appelée lorsqu'un client souhaite rejoindre une salle de chat en utilisant l'événement ChatEvents.JoinRoom. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant l'ID de la salle de chat à rejoindre. Cette fonction utilise la méthode join de l'objet Socket pour ajouter le client à la salle de chat.

La fonction *roomMessage* est appelée lorsqu'un client désire envoyer un message dans sa salle de discussion en utilisant l'événement ChatEvents.RoomMessage. Elle prend deux paramètres : un objet Socket et un objet contenant les propriétés roomName (une chaîne de caractères représentant l'ID de la salle de chat), message (une chaîne de caractères représentant le message envoyé par le client) et senderType (une chaîne de caractères représentant le type d'utilisateur qui a envoyé le message). Une fois que la fonction confirme l'existence de la salle, elle envoie le message à tous les clients connectés à celle-ci via un événement ChatEvents.RoomMessage et la méthode '.to' du socket.

La fonction *removeDifference* est appelée lorsqu'un client trouve une différence dans une image en utilisant l'événement ChatEvents.FoundDifference. Elle prend deux paramètres : un objet Socket et un objet contenant les propriétés roomName (une chaîne de caractères représentant l'ID de la salle de chat) et data (un tableau d'objets Coordinate représentant les coordonnées de la différence trouvée). Une fois que la fonction confirme l'existence de la salle, cette fonction envoie les données de la différence à tous les clients connectés à la salle de chat en émettant un événement ChatEvents.FoundDifference à l'aide de la méthode'.to' du socket.

La fonction *getUsername* est appelée lorsqu'un client a besoin du nom de l'adversaire dans une partie multijoueur. Elle prend deux paramètres : un objet Socket et un objet contenant les propriétés roomName (une chaîne de caractères représentant l'ID de la salle de chat) et username. Une fois que la fonction confirme l'existence de la salle, cette fonction envoie le username de l'adversaire à tous les clients connectés à la salle de chat en émettant un événement ChatEvents.GetUsernameAdversary à l'aide de la méthode'.to' du socket. Cette méthode permet notamment de récupérer le nom de l'adversaire pour l'afficher dans la partie et dans la salle d'attente.

La fonction *surrender* est appelée lorsqu'un client abandonne une partie multijoueur. Elle prend deux

paramètres : un objet Socket et un objet contenant le roomName (une chaîne de caractères représentant l'ID de la salle de chat). Une fois que la fonction confirme l'existence de la salle, cette fonction envoie le roomName à tous les clients connectés à la salle de chat en émettant un événement ChatEvents.Surrender à l'aide de la méthode'.to' du socket. Par la suite, il est possible d'appliquer la logique lors de l'abandon d'une partie.

La fonction *removeHint* est appelée lorsqu'un client utilise un indice une partie. Elle prend deux paramètres : un objet Socket et le roomName (une chaîne de caractères représentant l'ID de la salle de chat). Une fois que la fonction confirme l'existence de la salle, cette fonction envoie une notification vide à tous les clients connectés à la salle de chat en émettant un événement ChatEvents.RemoveHint à l'aide de la méthode'.to' du socket. Elle sera utilisée afin de synchroniser l'affichage des indices lors des parties en coopération.

La fonction *isGameCreator* est appelée lorsque le client souhaite savoir s'il est le créateur d'une salle d'attente. Elle utilise l'événement WaitingRoomEvents.IsGameCreator et prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle. Cette fonction émet un événement WaitingRoomEvents.IsGameCreator à l'objet Socket avec la réponse à la question.

La fonction *isRoomFull* est appelée lorsque le client souhaite savoir si une salle est pleine. Elle utilise l'événement WaitingRoomEvents.IsRoomFull et prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle. Cette fonction émet un événement WaitingRoomEvents.IsRoomFull à l'objet Socket avec la réponse à la question.

La fonction *joinGameRoom* est appelée lorsqu'un client souhaite rejoindre une salle d'attente en utilisant l'événement WaitingRoomEvents.JoinGameRoom. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle à rejoindre. Cette fonction utilise la méthode join de l'objet Socket pour ajouter le client à la salle, puis émet un événement WaitingRoomEvents.GetGameRooms à tous les clients connectés à la salle d'attente, via la méthode 'broadcastAllGameRooms' du service waitingRoomService.

La fonction *leaveGameRoom* est appelée lorsqu'un joueur quitte une salle d'attente en utilisant l'événement WaitingRoomEvents.LeaveGameRoom. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle quittée. Cette fonction utilise la méthode leave de l'objet Socket pour retirer le joueur de la salle de jeu. Elle met également à jour la file d'attente de la salle en utilisant le service WaitingRoomService, puis elle diffuse toutes les salles mises à jour à tous les joueurs connectés en utilisant la méthode broadcastAllGameRooms du service.

La fonction *leaveQueueRoom* est appelée lorsqu'un joueur quitte une file d'attente de jeu en utilisant l'événement WaitingRoomEvents.LeaveQueueRoom. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la file d'attente quittée. Cette fonction utilise la méthode leave de l'objet Socket pour retirer le joueur de la file d'attente. Elle diffuse ensuite toutes les salles mises à jour à tous les joueurs connectés en utilisant la méthode broadcastAllGameRooms du service WaitingRoomService.

La fonction *creatorLeftGame* est appelée lorsqu'un joueur qui a créé une salle de d'attente quitte cette dernière en utilisant l'événement WaitingRoomEvents.CreatorLeftGame. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle de jeu quittée. Cette fonction utilise la méthode leave de l'objet Socket pour retirer le joueur de la salle de jeu. Elle émet également un

événement WaitingRoomEvents.CreatorLeftGame à tous les joueurs connectés à la salle de jeu en utilisant la méthode to du socket. Enfin, elle gère la mise à jour de la file d'attente de la salle en utilisant le service WaitingRoomService.

La fonction *opponentLeftGame* est appelée lorsqu'un adversaire quitte une salle d'attente. Elle prend deux paramètres : un objet Socket et une chaîne de caractères représentant le nom de la salle d'attente. Lorsque cette fonction est appelée, elle retire le socket de la salle d'attente en appelant la méthode leave, et émet un événement WaitingRoomEvents.OpponentLeftGame à tous les clients connectés à la salle d'attente en utilisant la méthode to du socket. Cette fonction utilise également les services WaitingRoomService pour diffuser toutes les salles et mettre à jour la salle d'attente.

La fonction *getGameRooms* est appelée lorsqu'un client demande la liste de toutes les salles de jeu disponibles en utilisant l'événement WaitingRoomEvents.GetGameRooms. Cette fonction prend un paramètre : un objet Socket. Elle récupère toutes les salles de jeu à partir de l'objet Server et les envoie au client en émettant un événement WaitingRoomEvents.GetGameRooms avec les données des salles de jeu.

La fonction *opponentJoinedGame* est appelée lorsqu'un adversaire rejoint une salle d'attente en cours. Cette fonction prend deux paramètres : un objet Socket et un objet contenant les propriétés roomName (une chaîne de caractères représentant le nom de la salle), et username (une chaîne de caractères représentant le nom d'utilisateur de l'adversaire). Une fois que la fonction confirme l'existence de la salle, elle ajoute le socket à la salle en utilisant la méthode join et émet un événement WaitingRoomEvents.OpponentJoinedGame à tous les clients connectés à la salle de jeu en utilisant la méthode "to" du socket. Cette fonction utilise également les services WaitingRoomService pour mettre à jour la salle d'attente et supprimer la salle de jeu de la liste des salles de jeu disponibles.

La fonction responseOfCreator est appelée lorsqu'un client répond à une invitation de jeu créée par un autre client en utilisant l'événement WaitingRoomEvents.ResponseOfCreator. Elle prend deux paramètres : un objet Socket et un objet contenant les propriétés creatorAnswer (un booléen représentant la réponse du client) et roomName (une chaîne de caractères représentant le nom de la salle d'attente). Si creatorAnswer est vrai, elle envoie un événement WaitingRoomEvents.ResponseOfCreator à tous les clients connectés à la salle d'attente avec les propriétés creatorAnswer et gameRoomName à l'aide de la méthode "to" du socket. Elle utilise également la méthode leave du socket pour retirer le client de la salle d'attente et la méthode updateQueueRoom du service waitingRoomService pour mettre à jour la liste d'attente. Si creatorAnswer est faux, la fonction envoie un événement WaitingRoomEvents.ResponseOfCreator avec les propriétés creatorAnswer et roomName à tous les autres clients connectés à la salle d'attente à l'aide de la méthode "to" du socket. Enfin, la fonction utilise la méthode broadcastAllGameRooms du service waitingRoomService pour mettre à jour la liste des salles de jeu.

La fonction *creatorUsername* est appelée lorsqu'un client souhaite récupérer le nom d'utilisateur du créateur de la salle d'attente en utilisant l'événement WaitingRoomEvents.CreatorUsername. Elle prend deux paramètres : un objet Socket et un objet contenant les propriétés roomName (une chaîne de caractères représentant le nom de la salle d'attente) et creatorUsername (une chaîne de caractères représentant le nom d'utilisateur du créateur de la salle d'attente). Ensuite, elle envoie le nom d'utilisateur du créateur de la salle d'attente à tous les clients connectés à cette dernière en émettant un événement WaitingRoomEvents.GetCreatorUsername à l'aide de la méthode '.to' du socket.

Sprint 3

Pour le sprint 3, les WebSocket seront utilisés pour implémenter plusieurs autres fonctionnalités. Voici quelques issues qui pourraient être complétés à l'aide de WebSocket. Il est important de savoir qu'il est probable que l'on ajoute d'autres fonctions utilisant des WS, que l'on en néglige certaines qui seront cités ci-dessous ou que l'on utilise HTTP au lieu de WS pour certaines fonctionnalités.

Mode temps limité

Le système doit incrémenter le compteur de différence pour les deux joueurs. Une fonction sendIncrementDiffCounter pourrait être appelée lorsque l'un des deux individus dans la salle trouve une différence avec le roomName en paramètre. Cette fonction émettait un événement ChatEvents.IncrementDiffCounter via la connextion socket avec les informations fournies. Ensuite, le serveur aurait une fonction incrementDiffCounter. Une fois que la fonction confirme l'existence de la salle, cette fonction envoie une notification vide à tous les clients connectés à la salle de chat en émettant un événement ChatEvents.IncrementDiffCounter à l'aide de la méthode'.to' du socket. Le client appellera la fonction removeHint lorsqu'il recevra cet événement.

Le système doit ajouter un temps supplémentaire défini dans la vue de configuration à chaque différence trouvée. Pour implémenter cette fonctionnalité, on pourrait envoyer la roomName et le temps à ajouter au serveur en émettant un événement TimerEvents.IncrementTimer. Ensuite le serveur devrait confirmer l'existence de la salle et envoyer une notification avec le temps ajouter au client avec la a méthode '.to' du socket en émettant un événement TimerEvents.IncrementTimer. Le client s'occupera alors d'incrémenter le temps pour les joueurs dans la salle.

Le système doit permettre au joueur restant de continuer la partie en mode *solo*. Lorsque l'un des individus abandonne la partie, on pourrait utiliser notre fonction *surrender* mentionnée ci-haut. Il suffirait simplement d'actualiser l'interface du joueur et de ne pas automatiquement terminer la partie.

Remise des données à leur état inital

Le système doit permettre la suppression de toutes les fiches de jeu. Lorsqu'un utilisateur déclenche un événement de suppression de tous les jeux, on mettrait une notification vide avec l'événement deleteAllGames. Le serveur enverra une notification vide globale, donc un broadcast, en émettant 'événement deleteAllGames. Lorsque le serveur recevra cet événement, il s'occupera de supprimer l'ensemble des jeux.

Le système doit permettre la réinitialisation des meilleurs temps de tous les jeux. Lorsqu'un utilisateur déclenche la réintialisation des meilleurs temps de jeux de tous les jeux, une fonction sendResetLeaderboard pourrait être appelée, qui enverra un événement LeaderboardEvents.ResetAllLeaderboards via la connexion socket avec les informations nécessaires, telles que le nom de la salle de jeu. Ensuite, le serveur devrait avoir une fonction resetLeaderboard, qui vérifiera l'existence de la salle et enverra une notification vide à tous les clients connectés en émettant un événement LeaderboardEvents.ResetAllLeaderboards. Le client appellera la fonction updateLeaderboard lorsqu'il recevra cet événement pour afficher les temps réinitialisés.

Le système doit permettre la réinitialisation à travers un bouton sur la fiche de jeu. Le système doit remplacer les meilleurs temps par des temps fictifs accompagnés de noms de joueurs fictifs. Lorsqu'un utilisateur déclenche la réintialisation des meilleurs temps de jeux d'un jeu spécifique, une fonction

sendResetLeaderboard pourrait être appelée, qui enverra un événement LeaderboardEvents.ResetLeaderboard via la connexion socket avec les informations nécessaires, telles que le nom de la salle de jeu. Ensuite, le serveur devrait avoir une fonction resetLeaderboard, qui vérifiera l'existence de la salle et enverra une notification vide à tous les clients connectés en émettant un événement LeaderboardEvents.ResetLeaderboard. Le client appellera la fonction updateLeaderboard lorsqu'il recevra cet événement pour afficher les temps réinitialisés.

Le système doit tenir compte des nouveaux meilleurs temps à la fin d'une partie de jeu réinitialisé. Lorsque le joueur termine une partie de jeu, le système enregistre le temps de jeu et le compare aux temps enregistrés précédemment pour le jeu. Si le temps du joueur est plus rapide que les meilleurs temps enregistré,. un événement LeaderboardEvents.UpdateLeaderboard via la connexion socket avec les informations nécessaires, telles que le nom de la salle de jeu et le nouveau meilleur temps de jeu. Ensuite, le serveur devrait avoir une fonction updateLeaderboard, qui vérifiera l'existence de la salle et enverra une notification avec le nouveau meilleur temps de jeux à tous les clients connectés en émettant un événement LeaderboardEvents.UpdateLeaderboard. Le client appellera la fonction updateLeaderboard lorsqu'il recevra cet événement pour afficher les temps actualisés.

Vue de configuration - constantes de jeu

Le système doit appliquer les constantes à l'ensemble du système après leur modification. Lorsqu'un utilisateur modifie les constantes de jeux et les enregistre, une fonction sendTimeConstants pourrait être appelée, qui enverra un événement TimeConstants.UpdateTimeConstants via la connexion socket avec les informations nécessaires, telles que les temps modifiés. Ensuite, le serveur devrait avoir une fonction updateTimeConstants qui enverra une notification avec les nouveaux temps de jeu avec l'événement TimeConstants.UpdateTimeConstants. Le client appellera la fonction updateTimeConstants lorsqu'il recevra cet événement pour afficher les temps réinitialisés.

Indice de jeu

Ici, la seule utilisation pertinente des sockets et pour actualiser l'affichage des sockets restants pour les utilisateurs dans la salle, mais cela a déjà été fait et expliqué avant.

Historique des parties jouées

Le système doit mettre à jour l'historique après la fin d'une partie. Lors de la fin d'une partie, une fonction sendNewHistoryGame pourrait être appelée, qui enverra un événement History.UpdateHistory via la connexion socket avec les informations nécessaires, telles que la date, l'heure, la durée, le mode de jeu, le nom du premier joueur, le nom du deuxième et le nom du gagnant. Ensuite, le serveur devrait avoir une fonction updateHistory qui enverra une notification avec la date, l'heure, la durée, le mode de jeu, le nom du premier joueur, le nom du deuxième et le nom du gagnant avec l'événement History.UpdateHistory. Le client appellera la fonction addHistorygame lorsqu'il recevra cet événement pour ajouter la nouvelle ligne à l'historique.

Meilleurs temps

L'utilisation des WS dans cette issue est très semblable à l'issue Remise des données à leur état initial. Toutefois, il faudra ajouter le gestionnaire de la base de données MongoDB afin d'être à jour avec celle-ci.

Message de partie (global)

Le système doit envoyer un message aux joueurs de toutes les parties actives lorsqu'un joueur se classifie

pour les tableaux des meilleurs temps de son jeu. Pour ce faire, il faudra simplement modifier l'envoie de message dans les fonctions existantes. En effet, actuellement on envoie seulement aux utilisateurs de la salle active. Il faudra simplement utiliser la méthode 'broadcast' du socket avec la logique existante.