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 finale	Banna, Michael Chowdhury, Rasel Doghri, Aziz Greige, Jason Moukheiber, Éric Zahreddine, Nour
2023-04-20	2.0		Chowdhury, Rasel Greige, Jason Zahreddine, Nour

Table des matières

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. 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 gérer le mode multijoueur et solo en mode temps limités, ainsi que classique. En effet, cela inclus la messagerie et le chat en ligne

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. En effet, les éléments que l'on veut récupérer sont l'entièreté des parties, pour les constantes, pour la récupération de l'historique de jeu, pour obtenir un jeu grâce à son id et pour créer un jeu. On utilise aussi le protocole HTTP afin de supprimer une partie en fonction de son id, mettre à jour le jeu et les constantes.

3. Description des paquets

Games.controller.ts

Les paramètres des requêtes HTTP sont toujours les mêmes qui sont : (req: Request, res: Response)

Requête	Description
this.router.get('/', async (req: Request, res: Response)	Cette requête GET a la route “/”. Elle sert à retourner toutes les parties à l’aide de la méthode getAllGames() de gameService. On l’utilise du côté client dans ConfigSelectContentComponent dans le ngOnInit() et dans GamesDisplayComponent dans la méthode fetchGames(). Si jamais il y a une erreur, le statut 500 est envoyé.
this.router.get('/consts', async (req: Request, res: Response)	Cette requête GET a la route “/consts”. Elle sert à retourner toutes les constantes de jeu à l’aide de la méthode getConsts() de gameService. On l’utilise du côté client dans GameConstantsDialogComponent dans le ngOnInit(). Si jamais il y a une erreur, le statut 500 est envoyé.
this.router.get('/history', async (req: Request, res: Response)	Cette requête GET a la route “/history”. Elle sert à retourner l'historique de jeu à l’aide de la méthode getHistory() de gameService. On l’utilise du côté client dans GameHistoryComponent dans le ngOnInit(). Si jamais il y a une erreur, le statut 500 est envoyé.
this.router.get('/:id', async (req: Request, res: Response)	Cette requête GET a la route “/:id”. Elle sert à retourner le jeu selon son id à l’aide de la méthode getGameById(id) de gameService. On l’utilise du côté client dans GameHistoryComponent dans le ngOnInit(). Si jamais il y a une erreur, le statut 500 est envoyé. Si la partie n’est pas trouvée, le statut 404 est envoyé accompagné du message ‘Game not found’.
this.router.post('/', async (req: Request, res: Response)	Cette requête POST a la route “/”. Elle sert à créer un jeu à l’aide de la méthode createGame(game) de gameService. On l’utilise du côté client dans CreationPageComponent dans la méthode create(). Si jamais il y a une erreur, le statut 500 est envoyé. Si la partie a bien été créer, le statut envoyé est 201. Si le corps de la requête ne contient pas d’éléments, le statut 400 est envoyé.
this.router.delete('/history', async (req: Request, res: Response)	Cette requête DELETE a la route “/history”. Elle sert à effacer l’historique des jeux à l’aide de la méthode deleteHistory() de gameService. On l’utilise du côté client dans GameHistoryComponent dans la méthode deleteHistory(). Si jamais il y a une erreur, le statut 500 est envoyé. Si l’historique a bien été effacé, le statut envoyé est 200 avec un message disant ‘{deletedCount} history records deleted’. Si l’historique est déjà vide, le statut 404 est envoyé suivi du message ‘No history found’
this.router.delete('/:id', async (req: Request, res:	Cette requête DELETE a la route “/:id”. Elle sert à

Response)	effacer une partie selon son id à l'aide de la méthode deleteGame(id) de gameService. On l'utilise du côté client dans GameItemComponent dans la méthode deleteGame(). Si jamais il y a une erreur, le statut 500 est envoyé. Si la partie a bien été effacé, le statut envoyé est 200 avec un message disant 'Game deleted'. Si la partie n'est pas trouvée, le statut 404 est envoyé suivi du message 'Game not found'.
this.router.put('/consts', async (req: Request, res: Response)	Cette requête put a la route "/consts". Elle sert à mettre à jour les constantes des jeux à l'aide de la méthode updateConsts(initialTime, penalty, timeGainPerDiff) de gameService. On l'utilise du côté client dans GameConstantsDialogComponent dans la méthode saveConsts() et dans ConfigPageComponent dans la méthode resetConsts(). Si jamais il y a une erreur, le statut 500 est envoyé. Si jamais les constantes ont bien été mise à jour, le statut 200 est envoyé.

Diff.controller.ts

Requête	Description
this.router.post('/find-differences', async (req: Request, res: Response)	Cette requête POST a la route "/find-differences". Elle sert à envoyer les différences entre les deux images à l'aide de la méthode findDifferences(img0, img1, radius) de diffService. On l'utilise du côté client dans CreationPageComponent dans la méthode onDetectDifferences(). Si les différences ont bien été envoyé, le statut envoyé est 201.

Socket

	Émetteur	Receveur	Paramètres	Description
(this.sio.sockets.sockets.get(sock) as any).emit('player-refuse');	Le créateur de la partie	Le joigneur de la partie	X	Cette requête est émise lorsque le créateur du jeu refuse le joueur qui à essayer de rejoindre la partie. Elle est émise à game-page.
this.sio.emit('update-game-button', { gameId: data.gameId, gameOn: false });	Le créateur de la partie	Le joigneur de la partie	X	Cette requête est émise lorsque le créateur d'un jeu spécifique créer un jeu 1v1, le bouton se met à jour afin de devenir rose et affiche rejoindre partie 1v1 à ceux qui veulent rejoindre la partie.

this.sio.on('connection', (socket)	x	Tous les sockets	socket	'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.
socket.on('create-game', async (data: any)	x	Le créateur de la partie	(data:any) data.player0 data.gameId	'create-game' a lieu lorsqu'un créateur décide de créer une partie en coopérative en mode Classique. Elle est donc appelée dans la fonction createMultiGame() de game-item.component.ts.
this.sio.emit('update-game-button', { gameId: data.gameId, gameOn: true });	Le créateur de la partie	Le joigneur de la partie	(data:any) data.gameId data.player0	Lorsque le joueur 1 crée la partie, la méthode émet update-game-button qui à été expliqué précédemment.
this.sio.to(data.gameId).emit('game-created');	Le serveur	Tous les sockets	data.gameId	Lorsque le jeu a été créé, on émet game-created au jeu spécifique à l'aide de son id.
this.sio.emit('update-game-button', { gameId: data.gameId, gameOn: true });	Le serveur	Tous les sockets	gameId: data.gameI, gameOn: true	Cette requête est émise lorsque le créateur d'un jeu accepte un joueur et que la partie commence. Le bouton retourne en bleu.
socket.on('join-game', async (data: any)	x	Tout le monde sauf le créateur	(data:any) data.player1 data.gameId	Ce socket s'active lorsqu'on veut rejoindre un jeu.
this.sio.to(matchId).emit('random-games', { games, constants, player0: data.player0, player1: data.player1 });	Le serveur	Tous les sockets dans la salle correspondent au id *matchId*	games, constants, player0: data.player0, player1: data.player1	Le socket émet random-games à games
socket.to(data.gameId).emit('player-joined', { players: this.joinersMap[data.gameId] });	Le créateur de la partie	Tous les sockets dans la salle correspondant au data.gameId	(data:any) Le date est utilisé pour accéder au nom joueur 2 et au id de la partie data.player1 data.gameId	Cette requête est émise quand le joueur rejoint la partie à l'aide de son id.
socket.on('join-approval', async (data: any)	x	Le créateur de la partie	(data:any) data.gameId	'join-approval' a lieu lorsque le créateur de la partie attend que le joigneur vienne
this.sio.to(data.gameId).emit('player-refuse')	Le créateur de la partie	Le joueur qui a tente de joindre la partie	data.gameId	On émet 'player-refuse' à la salle de jeu en fonction de son id.
(this.sio.sockets.sockets.get(joiner.id) as any).emit('player-refuse');	Le créateur de la partie	Le joigneur de la partie	x	On émet 'player-refuse' lorsque le créateur de la partie refuse la personne qui essaye de joindre.
this.sio.to(matchId).emit('game-started', {});	Le serveur	Tous les sockets dans la salle correspondent au id *matchId*	x	Le serveurde la partie émet un message à tous les socekts présents dans la salle
socket.on('validate-coords', async ({ x, y, found })	x	Tous les sockets	x, y, found	On emet 'validate-coords' lorsqu'on veut valider les coordonnées du clique
socket.emit('validate-coords', { res, x, y });	Le serveur	Tous les sockets presents dans la salle	Res, x, y	Le serveur 'validate-coords' lorsqu'il veut valider les cliques à l'interieur d'une salle

socket.to(Array.from(socket.rooms)[1]).emit('notify-difference-found', { diff: res });	Le serveur	Tous les sockets presents dans la room en question	{diff: res}	'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.
socket.to(Array.from(socket.rooms)[1]).emit('notify-difference-error');	Le serveur	Tous les sockets presents dans la room en question	x	'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.
socket.on('validate-coords-tl', async ({ x, y, found })	x	Tous les sockets	x, y, found	On emet ‘validate-coords-tl’ lorsqu’on veut valider les coordonnées du clique
socket.emit('validate-coords-tl', { res, x, y });	Le serveur	Tous les sockets presents dans la salle	Res, x, y	Le serveur ‘validate-coords-tl’ lorsqu’il veut valider les cliques à l’interieur d’une salle
socket.on('cancel-from-client', this.cancelFromClient);	x	Le créateur de la partie	This.cancelFromClient	Le créateur de la partie recoit un message comme quoi le client a cancelé la partie
socket.on('cancel-from-joiner', (gameId) => {	x	Le créateur de la partie	gameId	Le créateur de la partie recoit un message comme quoi le client a cancelé le processus de joignement de partie
socket.leave(gameId);	Le joueur de la partie voulant quitter la partie	Tous les sockets dans la salle correspondant au id gameId	x	Le joueur de la partie émet son intention de quitter la partie, demande qui est traité par la suite par le serveur
socket.to(gameId).emit('player-joined', { players: this.joinersMap[gameId] }); });	Le serveur	Tous les sockets dans la salle correspondant au id gameId	Players:this.joinersMap[gameId]	Le serveur émet un message qui laisse savoir le créateur de la partie qu’un joueur a join la partie
socket.on('game-deleted', (data: any)	x	Le créateur de la partie	data:any data.gameId	Le créateur de la partie recoit un message comme quoi la partie a été supprimé
this.sio.emit('game-deleted', { gameId: data.gameId });	Le serveur	Tous les sockets	gameId: data.gameId	Le serveur émet ‘game-deleted’ lorsque le jeu est supprimé
socket.on('delete-all-games', async ()	x	x	x	Le créateur de la partie recoit un message comme toutes les jeux ont été supprimé
this.sio.emit('game-deleted', { gameId: game });	Le serveur	Tous les sockets	gameId: game	Le serveur émet ‘game-deleted’ lorsque le jeu est supprimé
this.sio.emit('refresh-games');	Le serveur	Tous les sockets	x	Le serveur émet ‘refresh-games’ lorsqu’un raffraichissement est nécessaire
socket.on('game-end', async (data)	x	Les joueurs dans la partie	Data Data.winner, Data.playerAbabandoned	Les joueurs dans la partie recoit un message comme quoi la partie est terminé
this.sio.to(matchId).emit('enemy-abandon');	Le serveur	Tous les sockets dans la salle correspondant au id matchId	matchId	Le serveur de la partie lance un message comme quoi son adversaire a quitté la partie
this.sio.to(matchId).emit('new-record', { newRecord });	Le serveur	Tous les sockets dans la salle correspondant	newRecord	Le serveur de la partie lance un message comme quoi un nouveeau record de meileur

		au id matchId		temps a été instauré
socket.on('denied', (socketJoiner)	x	Le socket qui tente de joindre la partie	socketJoiner	Le serveur émet un message comme quoi le socket du joigneur ne peut rejoindre la partie
this.sio.to(socketJoiner.id).emit('player-refuse');	Le créateur de la partie	Tous les sockeys dans la salle correspondant au id socketJoiner.id	x	Le créateur de la partie émet un message comme quoi un joueur en question a un acces refuse dans la partie
socket.on('room-message', (data: any)	x	Tous les sockets dans une salle en question	data:any data.message	Le serveur émet un message qui peut etre vu par tous les sockets d'une salle en question
socket.to(Array.from(socket.rooms)[1]).emit('room-message', { message: data.message });	Le joueur ayant envoyé un message dans le chat	Les joueurs présents dans une salle en question	{message:data.message}	Le message émit par un joueur quelconque dans une salle peut seulement etre vu par les autres joueurs de la meme salle
socket.on('global-message', (data: any)	x	Tous les sockets	Data:any	Le serveur émet un message global à tous les sockets
socket.broadcast.emit('global-message', data);	Le joueur qui a lancé le message	Tous les sockets sauf l'émetteur	data	Un message global est envoyé a tous les sockets sauf évidemment celui qui l'a lancé
socket.on('reset-game-history', async ()	x	Tous les sockets	x	Le serveur émet un message comme quoi l'historique de jeu sera réinitialisé
socket.on('reset-scores', async (data: any)	x	Tous les sockets	Data:any Data.gameId,	Le serveur émet un message comme quoi le pointage de tous les joueurs sera réinitialisé
socket.on('disconnecting', ()	x	Tous les sockets	x	Le serveur émet un message comme quoi les sockets présents seront déconnectés
socket.to(room).emit('cancel-from-joiner');	Le serveur	Pour les sockets dans la salle correspondant au id room	x	Le servur émet un message comme quoi, à tous les sockets dans la salle corresponsant au id room, que le rejoingnement d'une partie a été cancelé

Brève descriptions des méthodes utilisant HTTP et WebSocket:

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.

- '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.