Équipe 103

Protocole de communication

Version 1.7

Historique des révisions

Date	Version	Description	Auteur
2023-03-14	1.0	ajout de la description des paquets concernant la salle d'attente en protocole SocketIO	Alexis Nicolas
2023–03-15	1.1	Description des routes HTTP	Elouan Guyon
2023-03-16	1.2	Ajout de la description des paquets concernant les messages de systèmes et les messages dans le chat	Jasper Lai
2023-03-17	1.3	Description du protocole WS pour le chat, les événements déclenchés lors d'un jeu et les événements globales	Jasper Lai
2023-03-18	1.4	Ajout de l'introduction	Julien Légaré Jasper Lai
2023-03-19	1.5	Ajout de communication match. mise à jour de la salle d'attente. Ajout d'une justification pour le choix de SocketIo pour la salle d'attente. Justification de SocketIo pour la suppression de partie	Alexis Nicolas
2023-03-19	1.6	Modification des routes dans la section 3 et ajout des futurs routes pour le sprint 3	Elouan
2023-03-21	1.7	Déplacement de certains events de chat à match, correction de la table matière, ajout broadcast pour l'événement de meilleur temps	Jasper

Table des matières

Protocole de communication

1. Introduction

1.1. Mise en contexte

Dans le cadre du cours LOG2990 - Projet de logiciel d'application Web, nous avons été chargés de créer une version internet du jeu des sept différences. Pour réussir à mener ce projet à jour, nous avons dû commencer par analyser et comprendre la tâche qui nous était demandée grâce au document de vision qui nous était fourni, ainsi qu'avec l'aide des descriptions de tâches présentes sur GitLab, puis ensuite amorcer le développement en nous familiarisant avec le mode de travail en intégration continue, une méthode avec laquelle peu d'entre nous étaient familiers.

Pour nous aider dans le développement de notre application, nous avons dû nous familiariser avec le langage de programmation TypeScript, ainsi que la plateforme Angular qui permet de créer des applications web en réseau. Nous avons aussi dû utiliser des protocoles que nous n'avions jamais utilisés auparavant, comme le protocole WebSocket ainsi que des protocoles que nous avions vu tel que HTTP.

Ce document contient donc les protocoles de communication client-serveur utilisés, ainsi que la description des différents paquets utilisés, ce qui permet une compréhension générale des choix que nous avons décidé de faire pour notre projet.

1.2 Structure du document

Pour élaborer nos choix de méthodes de communication client-serveur, nous avons décidé de décrire en premier lieu les différents protocoles (HTTP et WebSocket) utilisés ainsi que la raison de leur utilisation. Nous spécifions aussi les tâches qui nécessitent l'utilisation de ces protocoles dans cette section. En deuxième lieu, nous décrivons le contenu, le contexte d'utilisation et les spécificités de chaque requête HTTP et de chaque paquet WebSocket.

2. Communication client-serveur

2.1 Protocole HTTP

On utilise HTTP pour l'ensemble des fonctionnalités implémentées au sprint 1. Commençons par la création d'une partie. Le client doit pouvoir téléverser des images au serveur et ensuite valider la création d'une partie ou bien décider de ne pas la créer. Tout cela se fait avec HTTP. On utilise également HTTP pour vérifier si le clic d'un joueur est sur une différence ou non. Également, lorsqu'un joueur active le mode triche, il y a une requête HTTP qui est effectuée au serveur afin d'obtenir les différences. La suppression d'une partie utilise à la fois HTTP et WS. La très grande majorité du travail est fait avec HTTP, mais WS est nécessaire pour déterminer en temps réel le nombre de parties qui sont jouées. Si on supprime une partie qui a actuellement des joueurs, on veut supprimer l'objet partie, mais on ne va supprimer l'objet des différences qu'une fois qu'il n'y a aucun joueur qui est en train de jouer la partie, car on veut que les joueurs puissent terminer leur partie même si le jeu a été supprimé.

2.2 Protocole WebSocket

Le protocole WebSocket est utilisé pour le clavardage entre deux joueurs, les événements déclenchés par les joueurs lors d'une partie, les événements globaux et la salle d'attente lors d'une partie multijoueur.

Au niveau du clavardage et les événements déclenchés par les joueurs lors d'une partie (différence, erreur et indice), le protocole WebSocket est utile vu qu'il nous permet de regrouper

deux sockets clients dans une chambre avec un identifiant unique. Grâce à cette organisation, nous sommes capables de partager des données concernant les différences trouvées, les erreurs commises et les messages écrits d'un joueur entre deux joueurs spécifiques. Nous sommes aussi capables de savoir quel joueur à déclencher quel événement ou à envoyer quel message grâce au identifiant unique de chaque socket. De cette manière, nous sommes capables de déterminer la logique du jeu multijoueur ainsi que l'apparence des messages dans la boîte à message. De plus, cette organisation permet d'avoir plusieurs parties indépendantes simultanément. Similairement, pour les événements globaux, le protocole WebSocket nous est utile pour la notification de toutes personnes dans une partie vu que nous pouvons les regrouper dans une room et émettre cette notification à tous les participants qui sont présentement dans une partie.

Nous utilisons également SocketIo pour la salle d'attente dans la vue de sélection, car les rooms de WebSockets nous permettent de regrouper tous le sockets qui regardent une fiche de jeu dans une même room, et ainsi pouvoir leur émettre en temps réel l'événement qu'une personne souhaite héberger une partie. le bouton «créer une partie» deviendra «rejoindre» pous tous ceux qui regardent cette fiche. Le protocole WebSocket nous simplifie également la tâche lorsque l'hôte souhaite accepter une demande de partie et qu'il doit émettre plusieurs évènements ciblés pour plusieurs groupes différents, car nous pouvons envoyer des events à tous les clients dans une salle, mais seuls ceux qui L'écoutent le recevront. Cela est utile lorsque l'hôte n'est plus un hôte de la salle d'attente. Tous ceux dans la salle d'attente savent que l'hôte a quitté la salle et reçoivent aussi le même message que ceux qui font seulement regarder la fiche de jeu afin de changer leur bouton vers «créer».

Nous avons choisi de faire la création d'une partie en protocole HTTP, mais la suppression d'une partie se fait en SocketIo, car nous avions besoin d'utiliser le principe de rooms pour émettre aux clients regardant une partie et ceux dans une salle d'attente qu'elle n'est plus disponible pour jouer.

3. Description des paquets

3.1. HTTP

3.1.1. Route /api/stage

Tous les endpoint qui sont créés à partir de stage sont liés aux objets de parties.

Tableau 1 : Liste des routes partant de /api/stage.

Route	Méthode	Contenu de la requête	Réponse de la requête
get /api/stage	`async getStages( @Query('index')index:number @Query('endIndex')endIndex: number, @Res() res: Response)`	Index initial et index final en query	Toutes les informations sur les parties comprises entre les deux index. Retourne 200 si ça fonctionne et 500 s'il y a une erreur
get /api/stage/info	`async getNbOfStages()`	Aucun	Le nombre de parties total. Retourne 200 si ça fonctionne et 500 s'il y a une erreur

get /api/stage/:gameCardId	`async stagegetStageById(@Param()param, @Res() res: Response)`	id de la partie à retourner	La partie avec l'id passé en requête (retourne 200). Ne renvoie rien et le code 404 si la partie n'existe pas et 500 s'il y a une erreur
Post /api/stage	`async createGame( @Body()game:GameCardDto, @Res() res: Response)`	On passe dans le corps des informations nécessaires à la création d'une partie	On retournera la partie créée et le code 201. Si on ne passe pas de corps dans la requête, on retourne le code 400. On retourne 500 s'il y a une erreur
Post /api/stage/image/:radius	`async uploadImages(@UploadedFiles() files:ImageUploadDto, @Param() param, @Res() res: Response)`	On passe deux dto d'images ainsi que le rayon à appliquer sur l'image de différence	Si on crée un objet de différence valide, on retourne un objet ServerGeneratedGameInfo et le code 201. On renvoie 200 si la requête est bonne mais que le nombre de différences est invalide. On renvoie 400 si on ne reçoit pas les fichiers et 500 s'il y a une erreur
get /api/stage/image/:imageName	`async getImage(@Param()param, @Res() res: Response)`	Nom de l'image qu'on veut afficher	On renvoie l'image qui a le nom passé dans la requête et 200. On renvoie 404 si l'image n'existe pas et 500 s'il y a une erreur
delete /api/stage/image/:imageName	`async deleteImage(@Param() param, @Res() res: Response)`	Nom de l'image qu'on veut supprimer	Si on supprime l'image, on retourne 204. On retourne 404 si l'image n'existe pas et 500 s'il y a une erreur de traitement.
patch /api/stage/constants (sprint 3)	`async getStageupdateConstants( @Query('countdown') countdown:number, @Query('penalityTime') penalityTime:number, @Query('differenceFoundTime')differenceFoundTime:number, @Res() res: Response)`	3 informations optionnelles: Temps compte à rebours, le temps de pénalité indice et le temps gagné avec différence trouvée	On renvoie 200 si on arrive à mettre les informations à jour et 500 s'il y a une erreur
get /apî/stage/constants	`async getStagesConstants()`	aucun	Retourne les 3 constantes de jeu dans un array de 3 nombres.. Retourne 200 si ça fonctionne et 500 s'il y a une erreur

3.1.2. Route /api/game-click. **Tous les endpoint qui sont créés à partir de game-click sont liés aux objets de différences.**

Tableau 2 : Liste des routes partant de /api/game-click.

Route	Méthode	Contenu de la requête	Réponse de la requête
get /api/game-click	`async validateDifference( @Query('x')clickPositionX :number,@Query('y') clickPositionY: number, @Query('id') id: string)`	La position X et Y d'un pixel d'une image en query et l'id de cette image.	Renvoie un objet ClickDifferenceVerification et le code 200. Retourne 500 s'il y a un erreur
get /api/game-click/:id	`async getDifferencesFromId( @Param('id') stageId:string)`	id d'un objet de différence (une partie et son objet de différences ont le même id)	Retourner l'objet et le code 200. Renvoie 500 s'il y a une erreur
delete /api/game-click/:id	`async deleteDifferences( @Param('id') id: string)`	id de l'objet de différence à supprimer	Retourne le code 200 ou 500 s'il y a une erreur
get /api/game-click/hint/:id	`async getDifferences(@Param('id'), @Query('differenceNumber') differenceNumber:number)`	id d'un objet de différence et le numéro de la différence en query	On renvoie un array qui contient la bonne différence et 200. Si on donne un numéro de différence invalide on retourne 400. Si on donne un id invalide on renvoie 404 et si il y a une erreur de traitement on retourne 500

3.1.3. Route /api/best-time. **Tous les endpoint qui seront créés à partir de sont liés aux objets de meilleurs temps.**

Tableau 3 : Liste des routes qui vont partir de /api/best-time.

Nom de l'évènement	Méthode	Contenu de la requête	Retourne
patch /api/best-time/:id	`async updateBestTime( @Param('id')id:string @Query('gameDuration) gameDuration:number, @Query('playerName') playerName:string,`	L'id de l'objet à potentiellement mettre à jour. La durée de la partie et le nom du joueur qui a gagné la partie	Retourne 200 si tout s'exécute et 500 s'il y a une erreur

	`@Res() res: Response)`		
delete /api/best-time/:id	`async resetBestTimes( @Param('id')id:string @Res() res: Response)`	Id de l'objet de meilleurs temps qu'on veut réinitialiser	Si on supprime l'objet de meilleurs temps, on retourne 204. On retourne 404 si l'objet n'existe pas et 500 s'il y a une erreur de traitement.
delete /api/best-time	`async resetBestTimes( @Res() res: Response)`	**Aucun**	Si on supprime tous les objets de meilleurs temps, on retourne 204. On retourne 500 s'il y a une erreur de traitement.

3.2. WebSocket

3.2.1. Waiting-Room

Dans cette section de la salle d'attente. l'hôte est référé comme un client pour le serveur mais qui a la particularité d'être celui qui crée une partie et qui attend qu'un autre client lui demande de le joindre

Tableau 4: événements WebSockets contenus dans la salle d'attente pour rejoindre une partie en ligne

Nom de l'évènement	Source	Contenu	Informations supplémentaires
scanForHost	Client	**stageIds: string[]** liste des Id des stages que le client souhaite recevoir des évènements pendant qu'ils les regarde	
hostGame	Hôte	**stageId: string** Id du stage dont ll'hôte souhaite héberger la salle d'attente	
unhostGame	Hôte	vide	l'hôte arrête d'héberger la salle d'attente
matchCreated	Serveur	**stageId: string** Id du stage dont un hôte a créé une salle d'attente	
matchDeleted	Serveur	**stageId: string** Id du stage dont il est désormais possible de créer une salle d'attente en tant qu'hôte	Envoyé lorsque l'hôte soit arrête d'héberger sa salle d'attente, soit a accepté un adversaire
joinHost	Client	**JoinHostInWaitingRequest{** **stageId: string;** **playerName: string;** **}** Représente l'Id du stage que le client souhaite faire une demande ainsi que	

		le nom sous lequel il apparaît	
quitHost	Client	vide	le client ne souhaite plus rejoindre l'hôte dans sa partie
requestMatch	Serveur	**PlayerInformations {** **playerName: string;** **playerSocketId: string;** **}** informations utiles sur le joueur souhaitant rejoindre la salle d'attente de l'hôte	
unrequestMatch	Serveur	**socketId: string** Id du socket du client qui se retire de la salle d'attente	
acceptOpponent	Hôte	**PlayerInformations** - playerName est le nom de l'hôte qu'il transmet au client accepté. - playerSocketId est l'identificateur du client accepté	envoyé automatiquement en mode temps limité
declineOpponent	hôte	**opponentId: string** identificateur du client dont la demande est refusée par l'hôte	
matchAccepted	Serveur	**AcceptationInformation{** **playerName: string;** **playerSocketId: string;** **roomId: string;** **}** informations sur l'hôte de la salle d'attente à transmettre au joueur accepté, plus identificateur de la room qui comprend les 2 joueurs	
matchRefused	Serveur	**refusedReason: string** raison pour laquelle le client ou l'hôte se fait sortir de force de la salle d'attente	
matchConfirmed	Serveur	**roomId: string** identificateur de la room qui comprend les 2 joueurs	
deleteGame	Client	**stageId: string** Id du stage dont le client souhaite supprimer le la base de donnée	
deleteAllGames	Client	**aucun**	le serveur utilisera le deleteGame mais pour toutes les gameCards
gameDeleted	Serveur	**aucun**	À envoyer à tous les clients qui regardent la fiche de jeu

			correspondant au stage supprimé. leur forçant à rafraichir leur liste

3.2.2. Match

Le gateway match sert pour la communication d'informations que le serveur devrait connaître pour les joueurs qui sont entrain de jouer

Tableau 5: Événements WebSockets liés aux parties en cours

Nom de l'évènement	Source	Contenu	Informations supplémentaires
createSoloGame	Client	**stageId: string** Id du stage dont un joueur a commencé une partie en mode solo	sert à compter le nombre de joueurs dans chaque stage en tout temps dans le serveur
timer	Serveur	**timestamp: number** numéro représentant le temps écoulé en secondes depuis le début d'une partie	
stopTimer	Client	**room: string** numéro de la room qui lie 2 joeurs en mode multijoueurs ou Id du joueur en solo	les joueurs solo se font envoyer leur temps dans leur propre room
changeStage	Serveur	**stageId: string** Id du stage à afficher ensuite	utiliser pour envoyer le prochain stage à envoyer pour le mode temps limité
win	Joueur et Serveur	**Cas joueur** **room: string** Nom de la chambre à émettre une notification d'un gagnant et d'une fin de partie. **Cas serveur** **socketId: string** Identifiant du socket gagnant à transmettre aux deux joueurs	
difference	Joueur et Serveur	**Cas joueur:** **data:** **MultiplayerDifferenceInformation {** **differencesPosition: number;** **lastDifferences: number[];** **room: string** **}** Information sur la position de la différence dans le array des différences, la différence ainsi que la room dans laquelle se situe les deux joueurs pour leur transmettre les données.	difference est principalement utilisé pour transmettre les données de la différence aux deux joueurs ainsi que quel joueur à trouvé la différence.

		Cas Serveur: **data: PlayerDifference {** **differencesPosition: number;** **lastDifferences: number[];** **socket: string** **}** Information sur la position de la différence dans le array des différences, la différence ainsi que le socket du joueur qui a trouvé la différence.	

3.2.3. Chat

Dans cette section, il n'y a pas de distinction entre les deux joueurs. C'est-à-dire que n'importe quel joueur peut déclencher n'importe quel événement à n'importe quel moment.

Tableau 4: événements WebSockets en lien avec la messagerie chat ainsi que les évènements de points

Nom de l'évènement	Source	Contenu	Informations supplémentaires
validate	Joueur	**message: string** Message de l'utilisateur dans le clavardage à valider.	(0 < longueur du message < 200)
WordValidated	Serveur	**validate: Validation {** **isValidated: boolean;** **originalMessage: string;** **}** Information sur la validation du message ainsi que le message du approuvé.	En cas d'échec, le serveur retourne une valeur prédéfinie. Sinon, le message original est transmis pour être ajouté aux messages dans le clavardage.
event	Joueur	**data: RoomEvent {** **room: string;** **event: string;** **isMultiplayer: boolean** **}** Information sur le type de notification à montrer, la chambre à émettre le message et le type de jeu.	Event englobe n'importe quelle event qui est déclenché par un joueur lors de son interaction avec le jeu (Différence ou erreur). Principalement utile pour montrer les événements dans la boite à messagerie.
hint	Joueur	**room: string** Nom de la chambre à émettre une notification dans le cas qu'un joueur utilise un indice	

roomMessage	Joueur et Serveur	**Cas Joueur** **RoomManagement {** **room: string;** **message: string;** **}** Information sur la chambre à émettre le message ainsi que le message **Cas Serveur** **RoomMessage {** **socketId: string;** **message: string;** **event: string;** **}** Information sur le socket source du message, le message et le type de message.	**Cas joueur:** Utiliser seulement si l'événement validate valide que le message est de format correct. **Cas serveur:** Utiliser dans 3 context, soit un événement de type notification (erreur/différence trouver et indice), un événement de type abandon ou un événement de type message
Abandon	Serveur	**RoomMessage {** **socketId: string;** **message: string;** **event: string;** **}** Information sur le socket source qui a abandonné la partie, le message d'abandon (heure de l'abandon) et l'événement (abandon)	Déclencher uniquement lorsqu'un socket client se déconnecte. (Quitter ou rafraichir la page de jeu)
newTime	Serveur et Joueur	**Cas joueur** **BestTime {** **playerName : string;** **gameName: string;** **mode: string** **time: string** **}** Information sur le temps, le joueur qui a battue un record de temps sur le serveur à émettre à tous joueur en jeu. **Cas Serveur** **BestTime {** **playerName : string;** **gameName: string;** **mode: string** **position: string** **}** Information à émettre à tous joueur en cours de jeu.	Opération de type broadcast sur tous les joueurs qui sont présentement en jeux