209

Protocole de communication

Version 1.0

Historique des révisions

Date	Version	Description	Auteur
2023-03-18	1.0	L’introduction	Radwan Rahman
2023-03-19	1.0	communication client & desc. paquets	Radwan Rahman
2023-03-20	1.2	paquets HTTP	Radwan Radwan
2023-03-20	1.2	paquets HTTP	Skander Hannachi
2023-03-20	1.2	paquets WebSocket	Mohamad Awad

Table des matières

Protocole de communication

1. Introduction

Avec le projet 2, nous avons un aperçu progressive des implications d'un projet de développement web. En effet, avec chaque sprints qui nous impose une progression graduelle de notre site de jeu de trouver les différences, ce qui nous défie de démontrer les atouts que nous avons acquis tout au long de notre scolarité. A partir de nos connaissances sur le sur-ensemble de Javascript, nous devions utiliser le cadriciel Angular qui est basé sur une approche modulaire et sur les composantes. En premier lieu, nous allons aborder la communication entre le client et le serveur, de l'évolution de la méthode utilisée pour faire cette dernière. Par la suite, nous allons faire la description des paquets utilisés au sein de nos protocoles de communication, c'est-à-dire, les informations qui sont envoyés et reçus lors de la communication réseau.

2. Communication client-serveur

Pour notre projet, nous avons opté initialement pour les requêtes HTTP. Avec la fin du premier sprint, nous avons eu une rétrospection collective, et avons décidé d'adopter une manière différente pour certaines fonctionnalités de procéder avec la communication client-serveur. Effectivement, nous avons opté pour un protocole bi-directionnel, le WebSocket. Ce dernier nous offre une connexion persistante entre le client et le serveur. Ce dernier, implémenté dans notre projet avec la librairie SocketIO, nous a apporté un grand contraste entre l'état de notre projet avant et après la remise du premier sprint. Nous avons plusieurs raisons à ce choix brusque. Avec les 2 derniers sprints à l'horizon, nous avons pu analyser les requis finaux, et leurs implications pour aboutir à cette conclusion.

Au premier sprint, nous n'avons fait que des paquets de requêtes HTTP. Les paquets avaient le type de méthode, la route, le corps et/ou le(s) paramètre(s) et/ou une réponse associée. Nous avons plusieurs paquets qui ont été renvoyés du client, et chacuns de ses paquets sont expliqués et schématisés ci-dessous. Avec la fin de la première remise, nous avons eu un changement d'idées, et avons décidé de changer quelques types de paquets pour la communication client-serveur. En effet, nous avons, pour les deux derniers sprints, décidé d'implémenter des web-sockets. Cette décision a été prise après avoir lu les requis du deuxième et du troisième sprint. Pour les paquets websockets, la structure est ainsi: le nom de l'événement, sa source et son contenu (si lieu). En outre, il est aisé de constater que nous faisons usage de salles pour notre site web. Ces dernières nous permettent de regrouper des sockets du côté du serveur. Par la suite, le serveur communique avec un sous-ensemble de ses clients, comme par exemple notre fonctionnalité de clavardage. Les attributs principaux du socket (socket.id, socket.rooms et socket.data) sont important pour tout opérations utilisant les websockets.

3. Description des paquets

Pour permettre une visualisation des paquets, nous avons des schémas des paquets de protocoles HTTP, ainsi que les protocoles Websockets.

Voici les paquets de communication entre le client et le serveur qui ont été créé:

3.1 Protocole HTTP

Cas d'utilisation : *Envoie et réception des fichiers dans la vue de création*

Dans la vue de création, un utilisateur a pour option de téléverser des images afin de créer une fiche de jeu qui inclut les URL des images téléversées, soit l'image originale et l'image altérée. Le serveur doit également envoyer un feedback afin de valider le nombre de différences générées par la sélection/création des images utilisées pour la fiche. Si le nombre de différences respecte la contrainte imposée par le projet, soit entre 3 et 9 différences, la génération de la fiche de jeu aura lieu. Dans le cas contraire, l'utilisateur sera prévenu que ses images entrées ne respectent pas la norme et devra téléverser de nouvelles images jusqu'à ce que le nombre de différences soit valide.

Contrôleurs utilisés ainsi que ses routes utiles au cas d'utilisation

ImageController **/image**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
POST	/compare	Body	Cette route a pour but d'intercepter les deux fichiers d'image téléversés (contenu dans le corps de la requête), de comparer les deux images en faisant appel au service de détection des différences et de renvoyer au client le nombre de différences trouvées entre les deux images.

SheetController /sheet		
TYPE	ENDPOINT	UTILITÉ
POST	/	Cette route a pour but d'intercepter les deux fichiers d'image téléversés, elle se fera appelée par le client associé lors de la confirmation que les deux images contiennent bel et bien le nombre de différences requis. De plus, le middleware se chargera d'effectuer une création d'un objet *Sheet.* Cet objet est une interface respectant le schéma des objets contenus dans la base de données, il représentera une partie créée par l'utilisateur et sera stocké.

Cas d'utilisation : *Affichage des parties créées dans la vue de sélection*

Lorsque l'utilisateur se dirige vers la vue de sélection, il devrait pouvoir obtenir un affichage complet des parties présentes dans la base de données afin qu'il puisse sélectionner la fiche de jeu sur laquelle il voudra jouer.

Contrôleurs utilisés ainsi que ses routes utiles au cas d'utilisation

SheetController /sheet			
TYPE	ENDPOINT	PARAMS/ BODY	UTILITÉ
GET	/		Cette route a pour but de renvoyer au client l'ensemble des fiches de jeu présentes dans la base de données afin de les présenter à l'utilisateur, le middleware fera un appel find générique pour retourner un tableau contenant l'ensemble des fiches de jeu. La réponse contiendra ce tableau.

Cas d'utilisation : *Suppression d'une fiche de jeu dans la vue de création*

Sur la vue de création, l'utilisateur a pour option de cliquer sur un bouton de suppression associé à une fiche de jeu en particulier. Il enverra ainsi une requête au serveur pour le prévenir que cette fiche est à retirer de la base de données.

Contrôleurs utilisés ainsi que ses routes utiles au cas d'utilisation

SheetController **/sheet**			
TYPE	ENDPOINT	PARAMS/ BODY	UTILITÉ
GET	/:id	PARAMS	Cette route a pour but d'abord de récupérer le paramètre *id* envoyé lors de la requête afin de supprimer la fiche de jeu associée à l'identifiant envoyé. Le middleware va effectuer une requête de suppression vers la base de données sur la fiche de jeu voulue. Sur une suppression réussie le serveur renvoie OK et sinon NOTFOUND.

Cas d'utilisation : *Récupération de la fiche de jeu ainsi que des images utilisées au lancement d'une partie*

Lorsque l'utilisateur aura sélectionné la partie sur laquelle il désire jouer, il sera redirigée vers la vue du jeu. À ce moment, le client demandera au serveur de lui servir les images de jeu, originales et altérées ainsi que la fiche complète de jeu.

Contrôleurs utilisés ainsi que ses routes utiles au cas d'utilisation

SheetController **/sheet**			
TYPE	ENDPOINT	PARAMS/ BODY	UTILITÉ
GET	/:_id	PARAMS	Cette route a pour but d'abord de récupérer le paramètre *_id* envoyé lors de la requête. Le middleware récupérera alors depuis la base de donnée la fiche de jeu associée à l'identifiant voulu et se chargera de la transmettre au client avec le message OK, dans le cas où la fiche n'existe pas une réponse NOTFOUND sera renvoyée au client.

ImageController **/image**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
GET	/:filename	PARAMS	Cette route a pour but de récupérer le nom du fichier associé à l'image désirée et de renvoyer au client le fichier dans son intégralité pour pouvoir jouer dessus avec la réponse OK. Dans le cas contraire, la réponse NOTFOUND sera renvoyée au client.

Cas d'utilisation : *Enregistrement des scores sur une fiche de jeu*

Contrôleurs utilisés ainsi que ses routes utiles au cas d'utilisation

SheetController **/sheet**			
TYPE	ENDPOINT	PARAMS/ BODY	UTILITÉ
Patch	/	BODY	Cette route a pour but d'abord de récupérer dans le corps de la requête un objet partiel (patron proxy) de la fiche de jeu sur laquelle la partie s'est jouée, cet objet servira à apporter des modification sur la fiche de jeu notamment en sauvegardant un meilleur joueur avec son score s'il y a lieu. Le middleware fera ainsi une mise à jour de la fiche associée sur la base de données et retourne une réponse OK en cas de succès et NOTFOUND en cas d'échec.

cas d'utilisation: *Envoie et réception des fichiers dans la vue de création*

Services client utilisés ainsi que ses routes utiles au cas d'utilisation :

ImageService **/image**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
POST	/compare	BODY	Cette route accepte un objet sheet avec des images à comparer. Sa retourne une requête POST au serveur, qui permet d'envoyer l'objet FormData en tant que corps de la requête. En cas de succès, on a OK, sinon

SheetService **/sheet**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
POST	/create	PARAMS	Cette route a pour but d'accepter un objet FormData comme parametre. Il envoie une requete HTTP POST au serveur, pour pouvoir

Cas d'utilisation : *Affichage des parties créées dans la vue de sélection*

Services client utilisés ainsi que ses routes utiles au cas d'utilisation :

SheetService **/sheet**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
GET	/get	PARAMS	Cette route a pour but de retourner un observable d'un array d'objets Sheet. Il envoie une requete HTTP GET au serveur, et s'attend à recevoir un array d'objets sheet en reponse. Il inclus aussi un handler d'erreur qui retourne un array vide en cas d'erreur.

Cas d'utilisation : *Suppression d'une fiche de jeu dans la vue de création*

Service client utilisés ainsi que ses routes utiles au cas d'utilisation :

SheetService **/image**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
DELETE	/id	PARAMS	Cette route a pour but d'accepter un paramètre id. Il supprime la fiche de jeu que l'on ne desire plus avoir. Il inclut aussi handler d'erreur qui utilise un opérateur catchError pour retourner une reponse vide en cas d'erreur.

Cas d'utilisation : *Récupération de la fiche de jeu ainsi que des images utilisées au lancement d'une partie*

Service client utilisés ainsi que ses routes utiles au cas d'utilisation :

SheetService **/sheet**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
GET	/id	PARAMS	Cette route a pour but d'accepter un parametre id. Il envoie un GET au serveur. Ceci est pour avoir Il inclus aussi un handler d'erreur utilisant un opérateur catchError pour reto8unr un objet Sheet vide en cas d'erreur.

ImageService **/image**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
GET	/	BODY	Cette route a pour but de faire un get au serveur avec le type de reponse 'blob' pour indiquer que le type de reponse devrait etre un objet binaire representant l'image. En d'autre terme, il prend du serveur le nom du fichier avec l'image du jeux qui est désiré.

Cas d'utilisation : *Enregistrement des scores sur une fiche de jeu*

Service client utilisés ainsi que ses routes utiles au cas d'utilisation :

SheetService **/sheet**			
TYPE	ENDPOINT	PARAMS / BODY	UTILITÉ
PATCH	/id	BODY	Cette route a pour but d'avoir le paramètre id du sheet on veut update, et un FormData avec le nom de parametre etant sheetForm, qui contient la donnee mise a jour. Il retounre un observable de type objet any. Elle enovie une requete HTTP PATCH au serveur avec le corps ayant le FormaData.

Avec la fin de la première remise, nous avons eu un changement d'idées, et avons décidé de changer complètement la manière de procéder avec la communication client-serveur. En effet, nous avons, pour les deux derniers sprints, décidé d'utiliser les web-sockets. Cette décision a été prise après avoir lu les requis du deuxième et du troisième sprint. Pour les paquets websockets, la structure est ainsi: le nom d'événement, sa source et son contenu (si lieu). En outre, il est aisé de constater que nous faisons usage de salles pour notre site web. Ces dernières nous permettent de regrouper des sockets du côté du serveur. Par la suite, le serveur communique avec un sous-ensemble de ses clients, comme par exemple notre s Un socket peut communiquer avec les autres de sa salle ○ Par défaut chaque socket a sa propre salle qui correspond à son id

3.2 Protocole WebSocket:

Pour assurer une émission/réception robuste d'événements du côté client , nous utilisons un service Socket Client Service

Ce service est injectable et permet la communication en temps réel avec un serveur en utilisant la librairie Socket.IO. Ce service peut être injecté dans n'importe quelle page ou composant dans le client qui nécessite une connexion au serveur.

Le service fournit une interface générique pour communiquer avec le serveur, ce qui nous permet de se concentrer sur la mise en œuvre des fonctionnalités dont ils ont besoin sans se soucier des détails de l'établissement d'une connexion avec le serveur. Le service dispose de plusieurs méthodes, notamment isSocketAlive(), qui vérifie si la connexion est active, connect(), qui établit une connexion avec le serveur, disconnect(), qui ferme la connexion avec le serveur, on<T>(event: string, action: (data: T) => void), qui enregistre un écouteur d'événement, et send<T>(event: string, data?: T), qui envoie un événement au serveur avec une charge utile de données facultative.

Étant donné l'injection de dépendance , toutes les composantes injectant ce service partageront le même socket et donc chacune peut recevoir tous les événements destinés aux salles de ce dernier ,chaque composante choisit de gérer les événements qui les intéressent.

Côté serveur, nous utilisons une implémentation de classe TypeScript d'un gateway WebSocket. Elle utilise la librairie Socket.IO pour gérer les messages WebSocket entrants et sortants.

La classe ChatGateway implémente les interfaces OnGatewayConnection, OnGatewayDisconnect et OnGatewayInit, ce qui lui permet de gérer les événements de connexion et de déconnexion WebSocket et d'initialiser le serveur WebSocket. Elle définit également plusieurs gestionnaires d'événements de messages à l'aide du décorateur @SubscribeMessage.

La classe ChatGateway est responsable de la création et de la gestion des salles de jeux, de la gestion de la logique de jeu, et de l'envoi et de la réception de messages entre les clients.

La classe ChatGateway agit en tant que médiateur entre les clients et la logique de jeu, et gère l'état du jeu et la communication entre les clients.

Voici les paquets WebSocket de communication entre le client et le serveur.

Cas d'utilisation : Créer un jeu en Solo

NOM DU EVENT	BODY	UTILITÉ	Source/Destination
createSoloGame	Objet JS {name,sheetId, roomName}	Lorsqu'un joueur clique sur le bouton "jouer" dans la fiche de jeu, un événement est envoyé au serveur. Cet événement contient le nom du joueur, les informations de la fiche de jeu, ainsi que le nom de la salle de jeu qui communique avec le serveur. Le serveur crée alors une instance de l'interface Playroom, y ajoute les informations du jeu, et l'ajoute à son tableau de "rooms" où il stocke les informations des salles. La fiche de jeu permet de naviguer à la page de jeu en conservant le nom de la salle dans le URL	**Source: Game-Card Component** **Destination: Serveur**
players	Players[2] Interface Player {name,socketId, differencesFound}	À l'initialisation de la page de jeu(mode solo et 1v1) ,le serveur envoie à celle-ci , après la création de l'instance de Playroom , les informations des joueurs , pour qu'elle puisse les conserver, render les noms sur la vue du jeu et mettres à jour les différences trouvées vis à vis la progression du jeu	**Source: Serveur** **Destination: Game-Page Component**
numberOfDifferences	number	À l'initialisation de la page de jeu(mode solo et 1v1), le serveur envoie à celle-ci le nombre total de différences pour qu'elle puisse l'afficher sur la vue.	**Source: Serveur** **Destination: Game-Page Component**

Cas d'utilisation : Créer un jeu en Mode 1 contre 1.

NOM DU EVENT	BODY	UTILITÉ	Source/Destination
joinGridRoom	void	À l'initialisation du carrousel , celui-ci envoie un message au serveur pour le faire joindre la salle des carrousels.	**Source: Game-Card-Grid Component** **Destination: Serveur**
gameJoinable	sheetId : string	Dès que le bouton "créer" est cliqué ,le carrousel reçoit un event (enfant-parent) de la fiche du jeu et envoie au serveur la fiche du jeu, le serveur crée une salle portant le nom `GameRoom${sheetId}` et ajoute le socket du client créateur dans cette salle où il peut recevoir , accepter ou refuser un client qui veut joindre la partie	**Source: Game-Card-Grid Component** **Destination: Serveur**
Joinable	sheetId: string	le serveur renvoie à tous les carrousels à l'exception de celui où il y a eu la création de la partie, l'information qui leur permettra de changer les boutons "créer" en "joindre"	**Source: Serveur** **Destination: GridRoom (BroadCast)**
joinGame	JoinGame (interface) {name, sheetId}	En cliquant sur le bouton "joindre", le carrousel signale que son utilisateur souhaite joindre la partie, il donne le id de la fiche de jeu qu'il veut joindre et son nom pour qu'il soit envoyé au créateur, il fait joindre le socket demandant dans la salle du jeu créé.	**Source: Game-Card-Grid Component** **Destination: Serveur**
UserJoined	playerName : string	le carrousel utilise son dialog service pour afficher les noms des joueurs qui demandent de joindre la partie dans une file d'attente.	**Source: Server** **Destination: `GameRoom${sheetId}`**
UserJoined	playerName : string	le carrousel utilise son dialog service pour afficher les noms des joueurs qui demandent de joindre la partie dans une file d'attente.	

playerRejected	{playerName, sheetId}	Un client créateur rejette une demande de joindre, le serveur reçoit le nom du joueur à faire sortir de la salle correspondante à la fiche de jeu.	**Source: Game-Card-Grid Component** **Destination: Serveur**
playerConfirmed	{player1:string, player2:string,sheetId}	Un client créateur accepte une demande de joindre, envoie au serveur les noms des deux joueurs et la fiche du jeu , le serveur crée une instance de l'interface PlayRoom avec les noms des joueurs	
MultiRoom Created	{player2:string, roomName}	Le client reçoit cet event et vérifie si il est correspondant au client accepté et renvoie un signal avec le même body au serveur , dans le cas contraire le client envoie un évènement "***quit***" pour quitter la salle	**Source: Serveur** **Destination:** **`GameRoom${sheetId}`**
Player2Joined	{player2:string, roomName}	Le serveur reçoit les information du deuxième joueur et ajoute son socket à la salle créée pour le jeu et son nom	**Source: Game-Card-Grid Component (Player2) Destination:Serveur**
quit	sheetId: string	Le serveur reçoit les informations de tous les autres clients qui n'ont pas été acceptés et les fait sortir de la salle de liste d'attente.	
JoinedRoom	PlayRoom (interface)	Les deux joueurs participants au jeu reçoivent les informations de leurs playroom et utilisent une partie de ces informations comme paramètre dans le URL de la navigation vers la vue du jeu , ces paramètre seront utilisés par la vue du jeu pour pouvoir communiquer avec le serveur en progression du jeu. Le serveur renvoie à la vue du jeu les événements ***players*** et ***numberofDifferences*** comme décrit dans le cas d'utilisation de mode solo.	**Source: Serveur** **Destination:** **`GameRoom${sheetId}`**

Scénario Alternatifs dans le même cas

- À tout moment , le client créateur peut annuler le jeu, le serveur reçoit son signal et le renvoie à la salle des carroussels qui vont remettre le bouton ***joindre*** à ***créer***
- À tout moment , un client dans une liste d'attente d'un jeu peut annuler sa demande , il envoie son signal au serveur qui le transmet à son tour au client créateur et il sera retiré de la liste d'attente.

Cas d'utilisation : Jouer une partie.

Comme mentionné dans les deux cas précédents , à l'initialisation de la Game-Page Component , celle-ci reçoit du serveur les noms des joueurs et le nombre des différences totales pour les utiliser dans l'affichage.
La Play-Area Component qui fait partie de la game-page, injecte un service Game-Logic Service qui est responsable de la communication avec le serveur pour tout ce qui est validation de clique et

NOM DU EVENT	BODY	UTILITÉ	Source/Destination
click	data = {x: click.offsetX,y: click.offsetY,roomName: tplayerName: name, };	Quand un joueur clique sur une des deux images, on envoie l'information du click au serveur , ainsi que la salle de jeu et le nom du joueur	Source: Play Area.Game-Logic Service (client) Destination: Serveur
clickFeedBack	coordonnées du click , informations du joueur (son socketId , son nom et les nombres de différences trouvées à date)	le serveur, ayant déterminé si le click est valide ou non , il renvoie les informations nécessaire à la salle de jeu, seul le joueur ayant trouvé la différence reçoit un audio et visuel feedback sur son écran. Au cas ou le click n'est pas valide , un message visuel et sonor error parvient sur l'écran du joueur ayant cliqué. Les deux clients marquent la différence comme trouvée	Source: Serveur Destination: Play Area.Game-Logic Service (client)
gameDone	message : string	le serveur vérifie si les différences sont toutes trouvées en mode solo ou la moitié est trouvée en mode 1v1 et envoie un signal pour terminer la partie et annoncer le gagnant	Source:Serveur Destination :Area.Game-Logic Service (client)
clock	Date()	cet event est émis chaque seconde, la game-page component le reçoit et sourtrait le temps du début de la partie , pour avancer son ***timer***	Source:Serveur Destination: Game-Page Component

Scénario Alternatif : en Mode 1v1 , à tout moment un joueur peut quitter la partie , la méthode handleDisconnect : cherche la salle de du socket de ce joueur dans son tableau des PlayRooms et fait sortir le joueur de la salle , si le jeu

est terminé , le serveur supprime la salle de jeu, sinon , le serveur émet un message au joueur restant que son adversaire a quitté et qu'il a gagné.

Cas d'utilisation : Clavardage et messages de parties.

NOM DU EVENT	BODY	UTILITÉ	Source/Destination
roomMessage	ChatMessage : { content: string, type: string}	Quand un joueur envoie un message dans la boîte des messages la chat-zone component émet un évenement parent-enfant à la vue du jeu , qui le transmettera à son tour au serveur	Source : Game-Page Component Destination : Serveur
		Quand le serveur broadcast le message , il est reçu par l'autre client dans sa **game-page component,** qui va le passer comme **input** à la chat-zone component , le type est utilisé par le css pour déterminer la couleur et la position du message. De même quand un click est validé le serveur envoie à la salle de jeu un ***roomMessage*** de type ***game***	Source : Serveur Destination : Game-Page Component

Cas d'utilisation : Jeu en Mode Temps Limité

NOM DU EVENT	BODY	UTILITÉ	Source/Destination
indice	void	Le serveur reçoit ce signal et renvoie les coordonnées d'une différence non trouvée	Source : Game-Page-Component Destination:Serveur
indice	{ coords }	Le joueur reçoit l'indice, la game-page fait clignoter la différence et applique une pénalité en secondes sur le ***CountDown***	Source:Serveur Destination : Socket du joueur ayant demandé l'indice
sheetDone	void	Le serveur reçoit ce signal et le renvoie une nouvelle fiche de jeu au client	Source : Game-Page-Component Destination:Serveur
sheetDone	{sheetId : string}	Le client reçoit la nouvelle sheetId et utilise le même protocole http pour naviguer vers une nouvelles page de jeu qui conservera sa dernière mise à jour du ***CountDown***	Source:Serveur Destination : Socket du joueur ayant demandé l'indice
timeUP	Historique du jeu {startTime , endTime, player1, player2? , winner (player)? , gameMode, sheetID[]}	le serveur reçoit ce signal et conserve toute l'historique du jeu dans la base de données	Source : Game-Page-Component Destination:Serveur