Équipe 106

Projet 3 : Fais-moi un dessin
Protocole de communication

Version 1.2

Historique des révisions

Date	Version	Description	Auteur
2020-02-03	1.0	Introduction et communication client-serveur	Hubert et Zakari
2020-02-06	1.1	Description des paquets	Hubert et Zakari
2020-04-12	1.2	Mise à jour du protocole de communication finale	Hubert et Zakari

Table des matières

Protocole de communication

1. Introduction

Ce document présente les différents protocoles de communication que notre jeu multiplateforme utilisera pour fonctionner adéquatement. Après quelques rencontres en équipe, nous avons choisi d'utiliser l'architecture REST et les sockets pour la communication entre nos clients (lourd et léger) et le serveur. D'abord, les choix que nous avons effectués seront présentés et justifiés dans la section communication client-serveur, puis la description détaillée et le contenu des paquets seront éclaircis dans la section description des paquets.

2. Communication client-serveur

La communication entre le client et le serveur sera faite par deux principales technologies : REST API et sockets. Dans la majorité des cas, la connexion sera assurée par les sockets, car ils permettent d'établir une discussion stable entre le client et le serveur. REST API sera utilisée pour les fonctionnalités de création d'un jeu, de gestion du profil utilisateur et d'accès au classement des modes de jeu. Les sockets seront implémentés à l'aide de la librairie Socket.IO parce qu'elle simplifie la communication entre les clients grâce à la création de canaux personnalisés (*rooms*). Les sockets seront utilisés pour les fonctionnalités de clavardage et lors du déroulement du jeu.

Le serveur sera conçu sous l'environnement de développement Node.js et les informations importantes qui doivent être gardées en mémoire telles que les profils et les jeux seront sauvegardés dans une base de données. MongoDB sera utilisé comme système de gestion de base de données, car il est orienté documents et les requêtes à utiliser pour extraire l'information nécessaire seront relativement simples.

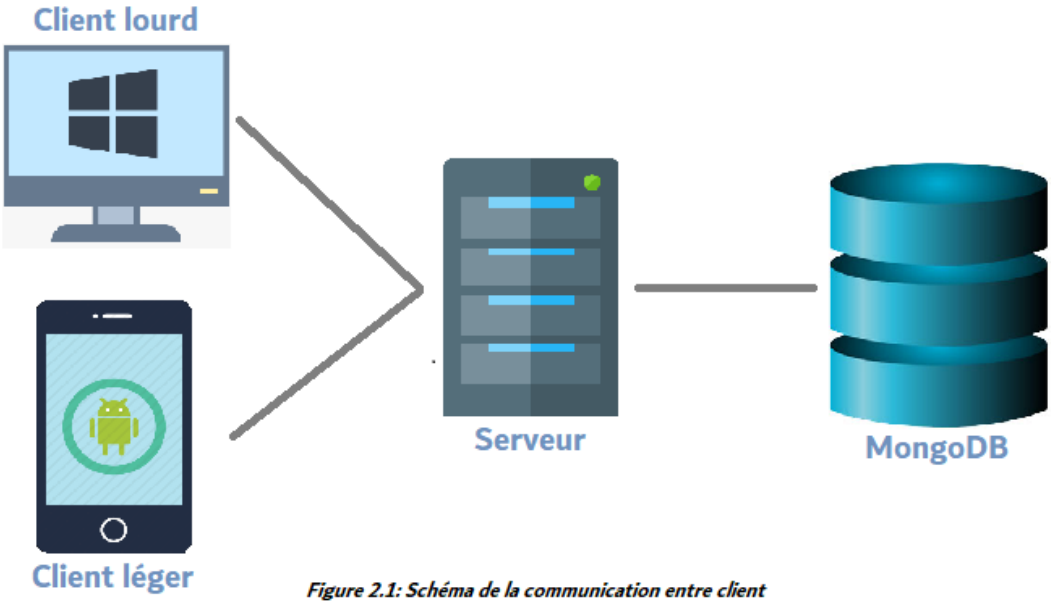

Figure 2.1: Schéma de la communication entre client lourd, client léger, serveur et base de données

Le type de format utilisé pour l'envoi et la réception de données est le format JSON, tant pour les requêtes HTTP que pour les évènements des sockets. Le client et le serveur s'assurent de sérialiser l'objet avant l'envoi et de le désérialiser après réception. Cela permet d'être cohérent au niveau du type de données utilisées et de répliquer des classes dans un langage différent. Plus précisément, une classe avec le même nom avec les mêmes noms d'attributs et les mêmes types d'attributs aura le même format JSON que ce soit en C#, Kotlin ou TypeScript. La traduction du JSON en objet se fera grâce à la librairie Gson en Kotlin et au cadriciel Json.NET en C#. De plus, les JSON se traduisent automatiquement en objet TypeScript.

3. Description des paquets

Dans cette section, les paquets utilisés pour la communication entre les clients et le serveur seront décrits en détail. Tout d'abord, les paquets des requêtes HTTP seront présentés, ensuite les paquets de la communication par sockets avec Socket.IO seront présentés. À noter : la plupart des évènements Socket.IO ont un évènement rétroaction associé. Ces retours du serveur (objet Feedback, voir annexe) contiennent l'état du traitement de la requête. De cette façon, on peut contrôler les changements de vues de l'interface selon le succès de la requête. Ils n'ont pas été présentés dans ce document puisqu'ils n'apportent pas de contenu significatif.

3.1 Paquets utilisés pour gérer le profil de l'utilisateur (HTTP)

Pour gérer le profil utilisateur, nous avons décidé de choisir le pseudonyme comme identifiant unique d'un utilisateur. De cette façon, nous pouvons chercher les utilisateurs dans la base de données en ayant uniquement leur pseudonyme. Les paquets qui permettent de gérer le

profil utilisateur sont détaillés dans le tableau I.

Tableau I - Paquets concernant la gestion du profil de l'utilisateur

Requête	Chemin (/profile...)	Description	Contenu paquet
POST	/create	Création d'un nouvel utilisateur.	- user / PrivateProfile*
GET	/private/**:username**	Récupération des informations privées d'un utilisateur.	- user / PrivateProfile*
DELETE	/**:username**	Suppression du profil utilisateur associé au username fourni.	
GET	/stats/**:username**	Récupération des statistiques.	- stats / Stats*

* Voir annexe pour la description des objets commun au serveur et au client

3.2 Paquets utilisés pour gérer les jeux créés (HTTP et Socket.IO)

Pour la création des jeux, le mot à deviner sera l'identifiant unique des jeux créés. De ce fait, on peut récupérer un jeu avec le mot à deviner. De cette façon, il n'y aura pas de redondance dans les dessins des joueurs virtuels. Par exemple, il ne pourra pas y avoir deux fois le dessin d'une orange, car l'utilisateur sera averti qu'il y a déjà un dessin pour ce mot lors de la création. De plus, la vitesse à laquelle le temps s'écoule est définie selon la difficulté, mais le temps de dessin total sera déterminé selon le temps de la manche lors de la partie. Les paquets permettant de gérer les jeux créés sont détaillés dans le tableau II et III.

Tableau II - Paquets concernant la gestion des jeux créés (HTTP)

Requête	Chemin (/game...)	Description	Contenu paquet
POST	/create	Création d'un jeu.	- createGame / CreateGame*

* Voir annexe pour la description des objets commun au serveur et au client

Tableau III - Paquets pour la prévisualisation (Socket.IO)

ID de l'événement	Description	Contenu paquet

preview	Prévisualisation d'un jeu (Voir le tableau VIII pour les évènements de dessin en temps réel).	- gamePreview / GamePreview*

* Voir annexe pour la description des objets commun au serveur et au client

3.3 Paquet utilisé pour consulter le classement (HTTP)

Pour les classements, un joueur pourra consulter le classement top 10 par mode de jeu. Les différents classements sont mis à jour à la fin de chaque partie, donc on a seulement besoin d'une requête qui permet de récupérer le top 10 pour un mode de jeu en question (le mode de jeu est identifié par l'énumération MatchMode, voir annexe). Par exemple, on peut récupérer le top 10 du mode 1 contre 1 en faisant la requête GET au chemin /rank/Hubert/. Le top 10 récupéré contient le nombre de points des 10 meilleurs joueurs. Il contient également le nombre de points du demandeur et sa position dans le dernier élément du tableau. Le paquet concernant le classement est présenté dans le tableau IV.

Tableau IV - Paquet concernant le classement

Requête	Chemin (/profile...)	Description	Contenu paquet
GET	/rank/**:username/:matchMode**	Récupération du top 10 d'un mode et de son propre classement dans ce mode.	- ranks / Rank[]*

3.4 Paquets utilisés pour la connexion et la déconnexion d'un utilisateur (Socket.IO)

Pour la connexion d'un utilisateur, ce dernier envoie un paquet contenant le nom d'utilisateur et le mot de passe. Si ces identifiants correspondent à un profil dans la base de données, l'identifiant du socket envoyé est synchronisé avec l'utilisateur dans un objet Map qui contient les utilisateurs connectés. Lors de la déconnexion, l'utilisateur et l'identifiant du socket sont supprimés de l'objet Map. Les paquets concernant la connexion et la déconnexion d'un utilisateur sont présentés dans le tableau V.

Tableau V - Paquets pour la connexion/déconnexion d'un utilisateur

ID de l'évènement	Description	Contenu paquet
sign_in	Connexion de l'utilisateur à partir de son pseudonyme et de son mot de passe.	- username / string - password / string

sign_out	Déconnexion de l'utilisateur	

3.5 Paquets utilisés lors de la création d'une partie (Socket.IO)

Pour créer une partie, le joueur hôte décide de certains paramètres tels que le nombre de manches, le mode de jeu et le temps limite des manches. Les joueurs sont mémorisés dans une liste associée à la partie du côté serveur et sont ajoutés à celle-ci lorsqu'ils rejoignent la partie. Le joueur hôte est défini par l'indice 0 de cette liste. Une fois la partie créée et un nombre minimum de joueurs (selon le mode), l'hôte de la partie peut démarrer une partie. Les autres joueurs peuvent rejoindre la partie à l'aide d'une liste de parties disponibles mise à jour. Les utilisateurs peuvent également quitter la partie. Si l'hôte quitte la partie, le premier joueur l'ayant rejoint devient l'hôte et ainsi de suite si ce dernier quitte la partie. Si l'hôte est seul dans la partie et qu'il la quitte, la partie est supprimée. Les paquets pour la création d'une partie sont présentés dans le tableau VI.

Tableau VI - Paquets pour la création d'une partie

ID de l'évènement	Description	Contenu paquet
create_match	Création d'un match (par l'hôte uniquement) avec nombre de manches, mode de jeu et le temps des manches.	- rounds / number - gameMode / MatchMode* - timeLimit / number
join_match	Joindre un match préalablement créé par l'hôte à l'aide d'un identifiant de la partie connu par l'hôte.	- matchId: string
leave_match	Quitter un match dans lequel l'utilisateur est déjà présent.	
add_vp	Ajouter un joueur virtuel à la partie.	
remove_vp	Retirer un joueur virtuel de la partie.	
update_matches	Peut être demandé avec une requête "get_matches", sinon c'est envoyé lorsqu'il y a un changement dans les parties disponibles.	- matchInfos / MatchInfos[]

* Voir annexe pour la description des objets commun au serveur et au client

3.6 Paquets utilisés lors du déroulement d'une partie (Socket.IO)

Le joueur courant (drawer) est celui qui dessine dans tous les modes. Dans les modes de sprint et en 1 contre 1, le joueur dessinateur est toujours un joueur virtuel préalablement établi. Dans les deux modes de sprint, le seul évènement utilisé pour faire les changements de tour est

“update_sprint”. Dans le mode mêlée générale, les changements de tour se font en deux temps avec l’évènement “turn_ended” qui envoie les scores de la dernière manche et les options de mots pour le dessinateur de la prochaine manche. Ensuite, l'événement “turn_started” permet d’envoyer le temps de la manche et le mot à deviner. En effet, pour que les joueurs qui devinent aient une idée de la longueur du mot on leur envoie le mot sous forme de barres de soulignement (ex: banana devient _ _ _ _ _ _).

Tableau VII - Paquets concernant le déroulement d’une partie d’un mode régulier

ID de l’évènement	Description	Contenu paquet
match_started	Envoyé par le serveur aux joueurs du salon lorsque l’hôte de la partie décide de débuter la partie.	- nbrOfRounds / number (pour l’affichage dans l’interface round 5 of **10**)
match_ended	Envoyé par le serveur lorsque la partie est terminée.	- players / Player[]*
start_turn	Envoyé par le joueur dessinateur après qu’il ait choisi son mot.	- word / string
turn_started	Envoyé par le serveur lorsque le joueur courant a choisi son mot et que la manche peut débuter. Indique au client de lancer son chronomètre.	- word / string - time / number (en secondes)
turn_ended	Envoyé par le serveur lorsque la manche est terminée. Indique au client d’arrêter son chronomètre.	- currentRound / number - players / Player[]* - choices / string[] - drawer / string
hint	Envoyé par le serveur si un des joueurs demande un indice dans le chat.	- hint / string
guess	Envoyé par un client désirant essayer de deviner le mot. Une réponse du serveur est ensuite envoyée pour lui indiquer s’il a deviné le mot.	- guess / string
unexpected_leave	Envoyé par le serveur lorsqu’un joueur quitte la partie au milieu de celle-ci et que les requis du mode ne sont plus respectés. Dans ce cas la partie se termine.	

* Voir annexe pour la description des objets commun au serveur et au client

3.7 Paquets utilisés pour le dessin en temps réel (Socket.IO)

Pour le dessin en temps réel, plusieurs événements sont nécessaires pour reproduire le dessin d’un joueur virtuel sur l’interface d’un client ou bien pour transférer les traits d’un autre client.

La logique sur le serveur est assez simple, mais les clients doivent écouter ces évènements et les insérer adéquatement dans la surface de dessin.

Tableau VIII - Paquets utilisés pour le dessin en temps réel

ID de l'évènement	Description	Contenu paquet
stroke	Trait envoyé du client vers le serveur.	- stroke / Stroke*
point	Point envoyé du client vers le serveur.	- stylusPoint / StylusPoint*
erase_stroke	Point où l'efface de trait est appliquée du client vers le serveur.	- stylusPoint / StylusPoint*
erase_point	Point où l'efface de point est appliquée du client vers le serveur.	- stylusPoint / StylusPoint*
clear	Demande au serveur d'effacer la surface de dessin de tous les autres clients du canal.	
new_stroke	Trait envoyé du serveur vers tous les clients du canal.	- stroke / Stroke*
new_point	Point envoyé du serveur vers tous les clients du canal.	- stylusPoint / StylusPoint*
new_erase_stroke	Point envoyé du serveur vers tous les clients du canal pour effacer les traits intersectés.	- stylusPoint / StylusPoint*
new_erase_point	Point envoyé du serveur vers tous les clients du canal pour effacer les points intersectés.	- stylusPoint / StylusPoint*
new_clear	Demande aux clients du canal d'effacer toute leur surface de dessin.	

* Voir annexe pour la description des objets commun au serveur et au client

3.8 Paquets utilisés pour le clavardage (Socket.IO)

Pour le clavardage, l'utilisateur peut créer un canal de discussion ou en joindre un existant. Une fois dans ce canal, il va charger l'historique de ces messages précédant son arrivée. Puis, il pourra à son tour envoyer un message. À tout moment, un utilisateur peut quitter un canal de discussion (sauf s'il est lié à une partie). Pour la suppression d'un canal de discussion (qui n'est pas lié à une partie), seul un utilisateur seul dans un canal de discussion peut le supprimer. De plus, il est important de noter que les canaux de discussion peuvent être privés ou publics. Les canaux publics sont visibles lorsque l'utilisateur recherche un canal à rejoindre dans la barre de

recherche et les canaux privés sont non visibles et peuvent seulement être rejoints suite à l'invitation d'un autre joueur.

Tableau IX - Paquets utilisés pour le clavardage

ID de l'évènement	Description	Contenu paquet
create_chat_room	Création d'un canal de discussion par un utilisateur. Un nom doit être attribué au canal et il ne doit pas déjà exister dans la liste des canaux déjà présents (roomId sert d'identifiant unique aux canaux de discussion).	- createRoom / CreateRoom*
join_chat_room	L'utilisateur rejoint un canal de discussion existant.	- roomId / string
leave_chat_room	L'utilisateur quitte un canal de discussion dans lequel il est déjà présent.	- roomId / string
delete_chat_room	Suppression d'un canal de discussion. La suppression va fonctionner uniquement si on est le dernier utilisateur dans le canal de discussion.	- roomId / string
send_message	Envoi d'un message dans le canal de discussion.	- message / ClientMessage*
new_message	Diffusion du message dans le canal de discussion.	- message / Message*
rooms_retrieved	Envoie le nom des canaux de discussion publics que l'utilisateur peut rejoindre.	- roomIds / string[]
send_invite	Envoyé au serveur par un utilisateur désirant inviter un autre utilisateur à rejoindre son canal de discussion privé.	- invitation / Invitation*
receive_invite	Envoyé à un utilisateur lorsqu'un autre utilisateur l'invite à rejoindre son canal de discussion privé.	- invitation / Invitation*
avatar_updated	Envoyé à tous les canaux de discussion dans lequel l'utilisateur est présent pour mettre à jour l'avatar.	- avatarUpdate / AvatarUpdate*

* Voir annexe pour la description des objets commun au serveur et au client

4. Annexe

Feedback: Objet
{
    status: boolean
    log_message: string (message à montrer à l'utilisateur)

}

PrivateProfile: Objet
{
	firstname : string
	lastname : string
	username : string
	password : string
	avatar : string
	(représente une image d’un de nos avatars, ex : BANANA, APPLE, etc.)
}

PublicProfile: Objet
{
	username : string
	avatar : string
	(représente une image d’un de nos avatars, ex : BANANA, APPLE, etc.)
}

Stats : Objet
{
	matchPlayed: number
	victoryPercentage: number
	averageMatchTime: number (en secondes)
	totalTimePlayed: number (en secondes)
	bestSprintSoloScore: number
	connections: string[]
	deconnections: string[]
	matchesHistory: MatchHistory[]
}

MatchHistory: Objet
{
	startTime: number
	endTime: number
	matchMode: MatchMode
	playerNames: string[]
	winner: Rank
	myScore: number
}

Rank: Objet
{
	username: string
	score: number
	pos: number

}

MatchInfos: Objet
{
	matchId: string
	host: string
	nbRounds: number
	timeLimit: number (en seconds)
	matchMode: MatchMode
	players: PublicProfile[] (username, avatar)
}

Player: Objet
{
	user: PublicProfile
	score: UpdateScore
	isVirtual: boolean
}

UpdateScore: Objet
{
	scoreTotal: number
	scoreTurn: number
}

StartTurn: Objet
{
	timeLimit: number
	word: string
}

EndTurn: Objet
{
	currentRound: number
	players: Player[]
	choices: string[]
	drawer: string

}

UpdateSprint: Objet
{
	guess: number (nombre d’essais restants)
	time: number (temps mis à jour)
	word: string (remplacé par des barres de soulignement)
	players: Player[]

```
}

Level : énumération
{
        Easy = 0
        Medium = 1
        Hard = 2
}

Mode: énumération
{
        Classic = 0
        Random = 1
        Panoramic = 2
        Centered = 3
}

Shape: énumération {
        Rectangle = 0,
        Ellipse = 1
}

Game : Objet
{
        word: string,
        drawing: Stroke[],
        clues: string[],
        level: Level,
}

CreateGame: Object {
        word: string,
        drawing: Stroke[],
        clues: string[],
        level: Level,
        mode: Mode
        option: number
}

GamePreview: Object {
   drawing: Stroke[],
   mode: Mode
   option: number
}

MatchMode: énumération
```

{
	freeForAll = 0
	sprintSolo = 1
	sprintCoop = 2
	1vs1 = 3
}

ClientMessage : Object
{
	roomId: string
	content: string
}

CreateRoom: Object
{
	id: string
	isPrivate: boolean
}

Message : Object
{
	username: string
	content: string
	date : number (timestamp)
	roomId: string
}

Invitation: Object
{
	id: string,
	username: string
	(peut être le sender ou le receiver dépendamment de l’événement)
}

AvatarUpdate: Object
{
	roomId : string
	updatedProfile : PublicProfile (contient le nouvel avatar)
}

Stroke: Object
{
	DrawingAttributes: DrawingAttributes
	StylusPoints: StylusPoints[]...
}

```
DrawingAttributes: Object {
        Color: string
        Width: number
        StylusTip: Shape
        Top: number
}

StylusPoint: Object {
        X: number
        Y: number
}
```