Équipe 102

Fais-moi un dessin
Protocole de communication

Version 1.6

Historique des révisions

Date	Version	Description	Auteur
2021-02-03	1.0	Première version de la section 1	Ming Xiao Yuan
2021-02-15	1.1	La communication client vers le serveur est complète	Benjamin Boucher-Charest
2021-02-16	1.2	La communication serveur vers le client est complète	Justin Caisse
2021-02-17	1.3	Révision de la communication client vers le serveur	Laura Beaudoin et Vlad Drelciuc
2021-02-18	1.4	Révision de la communication serveur vers le client	Benjamin Boucher-Charest et Vlad Drelciuc
2021-04-14	1.5	Corrections à la suite des commentaires de l'appel d'offre	Vlad Drelciuc
2021-04-19	1.6	Corrections à la suite des commentaires de l'appel d'offre	Andi Podgorica

Table des matières

Protocole de communication

1. Introduction

Le présent document, intitulé *Protocole de communication*, a pour but de représenter schématiquement les communications effectuées entre les composantes de l'application *Fais-moi un dessin*. La section 2 (Communication client-serveur) vise à schématiser notre choix de moyen de communication entre les utilisateurs de cette application et le serveur *Firebase,* tandis que la section 3 (Description des paquets) vise à expliciter les différents types de paquets utilisés au sein de notre protocole de communication.

2. Communication client-serveur

Les clients communiqueront avec le serveur à l'aide de l'API de Firebase. Il n'y aucune restriction (port ou adresse IP) sur les connexions utilisateurs, puisque le serveur doit être accessible pour n'importe quel utilisateur qui désire jouer. Firebase donne l'option d'utiliser des requêtes HTTP ou WebSocket. Nous utiliserons une combinaison des deux afin de balancer l'automatisation de certaines tâches asynchrones (avec des requêtes HTTP) et les changements en temps réel avec WebSocket.

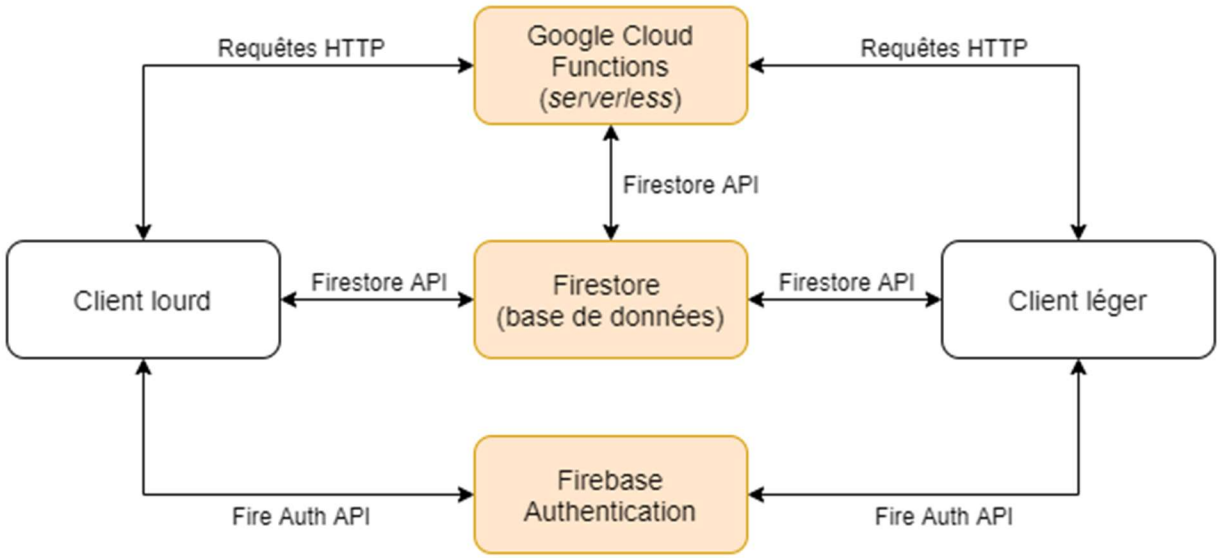

Fig. 1 - Image de la communication entre les clients et Firebase

Firebase Authentication est utilisé pour la gestion des comptes et l'authentification. Firestore est utilisé comme base de données. Firestore est accessible directement depuis les clients lourd et léger à travers son API pour les changements mineurs nécessitant du temps réel (ex : envoi et réception de messages, envoi de traits dessinés, etc.). Pour les requêtes de type asynchrone (ex : la création de lobbys, la gestion des parties, les calculs du classement de joueurs, etc.), les clients contactent des REST API "endpoints" liés à des Cloud Functions qui gèrent ces requêtes plus complexes.

3. Description des paquets

3.1 Définitions

3.1.1 Protocole TCP/IP

Le protocole de communication utilisé pour la transmission d'informations est le TCP/IP. La quantité de données dans un paquet sera limitée par le protocole (environ 1500 bytes de données par paquet), de même que la détection d'erreurs de transmission. Un paquet TCP/IP devrait respecter la forme suivante :

Description	Type de Message	Données

Fig. 2 - Image d'un paquet TCP/IP

3.1.2 Payload

Il s'agit des données du paquet. Dans la communication client vers le serveur, le payload va contenir l'information nécessaire pour soit remplacer un champ, ajouter un nouvel élément, ou supprimer un élément.

Voici les données possibles que l'on peut sauvegarder sur Firestore :

Array	Boolean	Geopoint
Map	Null	Number
Reference	String	Timestamp

Fig. 3 - Tableau des données possibles dans une base de données Firestore

3.1.3 Type

Le type va diriger le fonctionnement du payload lorsqu'il sera reçu dans Firestore. Trois types de paquets Firestore sont offerts pour un paquet :

- *ADDED*

  Ce type de paquet est assigné lorsqu'un utilisateur ajoute un élément à un document dans une collection Firestore, sinon il est également possible de créer un document. La nomenclature d'appellation pour les requêtes de type *ADDED* est la suivante : *add*

- *MODIFIED*

  Ce type est assigné lorsqu'un élément existant dans un document est modifié dans le Firestore. Une modification peut être limitée à un champ ou plusieurs champs à la fois. La nomenclature d'appellation pour les requêtes de type *MODIFIED* est la suivante : *set*

- *REMOVED*

  Ce type est assigné lorsqu'un élément doit être supprimé d'un document dans Firestore. La nomenclature d'appellation pour le type *REMOVED* est la suivante: *delete*

3.1.4 Éléments du Firestore

- *Collection*
  Une collection contient des documents dans Firestore. Afin d'avoir des documents, il est nécessaire d'avoir une collection existante.

- *Document*
  Un document contient des champs dans Firestore. Il a une limite de 1 Mb pour des données. Afin d'avoir des champs, il est nécessaire d'avoir un document existant. Il regroupe ensemble les champs d'un élément spécifique.

- *Champ*
  Un champ est contenu dans un document Firestore et il peut contenir une valeur, selon les types disponibles - **section 3.1.2**.

3.2 Google Cloud Functions

Les clients communiquent avec le backend “serverless” de Google Cloud Functions à travers des requêtes HTTP. Toutes les requêtes HTTP doivent être du type indiqué entre parenthèses (GET, POST, PUT ou DELETE), ainsi qu’elles doivent contenir les paramètres spécifiés ci-dessous dans leur corps (body) et être adressées à :

https://us-central1-chat-b68d4.cloudfunctions.net/{nom-de-la-requete}

Il est important de spécifier que les clients écoutent les changements sur Firestore (par exemple sur la partie en cours ou sur le chat en cours), alors même si la Cloud Function ne retourne pas de résultat, tant qu’elle modifie un document sur Firestore, tous les clients seront mis à jour automatiquement.

Tableau 1: Présentation de la requête leaderboard

Nom de la requête	**Description**	**Paramètres**	**Réponses possibles**	**Valeur de retour**

leaderboard
(GET)
	Retourne un classement des meilleurs joueurs pour différents modes de jeu (Classique et Solo Sprint) et différentes périodes de temps (dernières 24h, derniers 7 jours, derniers 30 jours et depuis toujours). Ces statistiques sont compilées à partir des parties enregistrées dans les collections Firestore.	(Aucun)	200 (OK)	{
				classicDay: Entry[],
				classicWeek: Entry[],
				classicMonth: Entry[],
				classicAll: Entry[],
				soloDay: Entry[],
				soloWeek: Entry[],
				soloMonth: Entry[],
				soloAll: Entry[]
				}
				où Entry = {
				player: string,
				score: number
				}

Tableau 2: Présentation de la requête createSoloGame

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

createSoloGame (POST)	Fait la création une partie de type Solo Sprint à partir des données passées en paramètre. Fait également la création d'un canal de discussion sur lequel le client pourra s'abonner pour envoyer ses guess. Initialise la machine à états du mode Solo Sprint.	{
		host: string,
		difficulty: string
		}
			200 (Game was created)
			400 (Invalid body params)
				{
				gameId: string
				}

Tableau 3: Présentation de la requête correctSoloGuess

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

correctSoloGuess (POST)	Requête appelée lorsque le joueur d'un Solo Sprint devine correctement un mot. Met à jour les statistiques de la partie (nombre de bons guess et des guess totaux) et ajoute du temps supplémentaire selon les paramètres de la partie. Met à jour le document correspondant à la partie dans Firestore.	{
		gameId: string,
		correctAttemptsThisRound: number,
		badAttemptsThisRound: number
		}
			200 (Game was updated)
			400 (Invalid body params)
				(Aucune, mais la partie est modifiée sur Firestore)

Tableau 4: Présentation de la requête noMoreSoloAttempts

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

noMoreSoloAttempts (POST)	Appelé lorsque le joueur a épuisé ses tentatives de deviner un mot durant une partie Solo Sprint. Met à jour les statistiques de la partie et génère une nouvelle image aléatoire que le joueur virtuel pourra dessiner. Met à jour le document correspondant à la partie dans Firestore.	{
		gameId: string,
		badAttemptsThisRound: number
		}
			200 (Game was updated)
			400 (Invalid body params)
				(Aucune, mais la partie est modifiée sur Firestore)

Tableau 5: Présentation de la requête endSoloGAme

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

endSoloGame (POST)	Appelé lorsque le temps a expiré ou que le joueur a quitté la partie Sprint Solo. Son score sera enregistré sur son profil, de même que l'expérience et la monnaie virtuelle gagnées durant la partie. Met à jour les documents correspondants à la partie et au profil de	{
		gameId: string,
		badAttemptsThisRound: number
		}
			200 (Game is done)
			400 (Invalid body params)
				(Aucune, mais la partie est modifiée sur Firestore)

	l'utilisateur dans Firestore.			

Tableau 6: Présentation de la requête createLobby

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

createLobby
(POST)
	Créer un lobby dans lequel les équipes seront formées avant le début d'une partie Classique. Retourne l'id de ce lobby qui permettra au client de rechercher et de s'abonner au nouvel document dans Firestore.	{
		difficulty: string,
		host: string,
		isPublic: boolean,
		mode: string,
		name: string
		}
			200 (Lobby was created)
			400 (Invalid body params)
				{
				lobbyId: string
				}

Tableau 7: Présentation de la requête createGameFromLobby

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

createGameFromLobby
(POST)
	Appelé une fois que le propriétaire d'un lobby remplit ce dernier et qu'il est prêt à commencer la partie Classique. Valide que le lobby est complet et respecte les contraintes de la machine à états. Si tout est correct, la partie est créée dans Firestore et l'attribut gameId du lobby	{
		lobbyId: string
		}
			200 (Game was created)
			400 (Invalid lobbyId)
			403 (Lobby not ready)
				(Aucune, mais la partie est créé sur Firestore et l'attribut gameId du lobby est modifié)

	est modifié, de sorte à notifier tous les clients que la partie est commencée.			

Tableau 8: Présentation de la requête guessWord

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

guessWord (POST)	Appelé lorsqu'un joueur d'une partie Classique tente de deviner un mot. Met à jour les statistiques de la partie et change l'état de la machine à états (en fonction de l'exactitude du guess). Met à jour le document de la partie dans Firestore.	{
		gameId: string,
		guesserUsername: string,
		word: string
		}
			200 (Game was updated)
			400 (Invalid gameId)
			403 (Not allowed to guess)
			404 (No image found)
				{
				isCorrect: boolean
				}

Tableau 9: Présentation de la requête timerExpired

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

timerExpired (POST)	Appelé lorsque le timer d'un round arrive à zéro. Fera avancer la machine à état de la partie Classique. Mettra à jour le document de la partie dans Firestore.	{
		gameId: string
		}
			200 (Game was updated)
			400 (Invalid body params)
				(Aucune, mais la partie est modifiée sur Firestore)

Tableau 10: Présentation de la requête quitGame

Nom de la requête	Description	Paramètres	Réponses possibles	Valeur de retour

quitGame
(POST)
	Appelé lorsqu'un joueur décide de quitter la partie Classique alors que celle-ci n'est pas terminée. Va sauvegarder le score, l'expérience et la monnaie virtuelle de l'équipe adverse, qui aura automatiquement la victoire. Les membres de l'équipe qui ont abandonné n'auront aucun point. Mettra à jour les documents de la partie et les profils de tous les joueurs de la partie dans Firestore.	{
		gameId: string,
		quitterUsername: string
		}
			200 (Game was stopped)
			400 (Invalid params)
			403 (Quitter not in game
				(Aucune, mais la partie est modifiée sur Firestore)

3.3 Firestore

Les clients peuvent également faire des requêtes directement à la base de données Firestore pour obtenir des informations et des changements en temps réel. Nous privilégions l'utilisation de requêtes directes vers la base de données lorsqu'aucune manipulation complexe nécessitant l'intervention du backend Google Cloud Functions n'est requise.

3.3.1 Gestion des messages

Les clients observent la collection des messages. Lorsqu'un nouveau message est ajouté, si celui-ci appartient au canal/canaux sur lequel l'utilisateur est connecté, l'interface sera mise à jour pour afficher ce nouveau message.

Tableau 11: Envoyer un message

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
addMessage	Le client ajoute un message à la collection de Firestore.	channel: string, content: string, senderID: string, timeStamp: Date*	(Aucune)

* Le timeStamp d'un message est ajouté par Firestore à la réception du message pour s'assurer que tous les clients reçoivent les messages dans le même ordre.

Tableau 12: Charger l'historique d'un canal de clavardage

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
getHistory	Le client demande de recevoir l'historique des messages d'un canal particulier.	channel: string	Array *contenant des objets avec les attributs suivants :* channel: string, content: string, senderID: string, timeStamp: Date

3.3.2 Gestion des canaux de communication

Les clients observent la liste des canaux sur le profil de l'utilisateur. Lorsqu'un utilisateur rejoint un canal, ce canal est ajouté dans la liste de ses canaux sur son profil. Lorsqu'il y a un changement (ajout ou retrait) dans cette liste, l'interface est mise à jour.

Tableau 13: Création d'un canal de clavardage

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
createChannel	Le client ajoute un canal à la collection de Firestore.	host: string, isPublic: boolean, name: string, shareKey: string*	(Aucune)

* shareKey est une clé unique de 5 ou 6 caractères pouvant être facilement partageable entre les utilisateurs pour rejoindre un canal

Tableau 14: Rejoindre un canal de clavardage

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
joinChannel	Le client ajoute la clé d'un canal dans la liste des canaux contenue dans le profil utilisateur du joueur.	username: string, channels: string[]*	(Aucune)

* channels est la liste mise à jour

Tableau 15: Quitter un canal de clavardage

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
quitChannel	Le client enlève la clé d'un canal qu'il souhaite quitter de la liste des canaux contenue dans le profil utilisateur du client.	username: string, channels: string[]*	(Aucune)

* channels est la liste mise à jour

3.3.3 Gestion des lobbies

La création de lobbies est gérée par une Cloud Function. Les clients observent le lobby dans lequel l'utilisateur se trouve et met à jour automatiquement l'interface graphique lorsque des changements sont observés (par exemple lorsqu'un nouvel utilisateur rejoint le lobby, il apparaît dans la liste des membres sur tous les autres clients).

Tableau 16: Obtenir les détails d'un lobby

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
getLobby	Retourne tous les détails liés au lobby correspondant à la clé fournie par le client.	shareKey: string***	difficulty: string, gameId: string,* host: string, isPublic: boolean, mode: string,** name: string, players: Map<string, number>, shareKey: string***

* gameId est nul par défaut. Lorsque le propriétaire du lobby décide de lancer la partie, une Cloud Function va s'occuper de créer la partie, puis va y ajouter son id ici. Étant donné que les changements au lobby sont écoutés par tous les clients présents, lorsque ceux-ci voient une valeur apparaître dans le champ gameId, ils savent que la partie va commencer et vont aller écouter les changements sur la partie correspondante.

** mode contient les quatre compositions d'équipes possibles pour le mode Classique : PPvPP (2 humains contre 2 humains), RPvPP (robot + humain contre 2 humains), PPvRP (2 humains contre robot + humain) et RPvRP (robot + humain contre robot + humain). Cet attribut est utilisé par la machine à état gérée par une des Cloud Functions.

*** shareKey est une clé unique de 5 ou 6 caractères pouvant être facilement partageable entre les utilisateurs pour rejoindre un canal

Tableau 17: Obtenir la liste de tous les lobbys publics

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
getPublicLobbies	Retourne la liste de tous les lobbys publics que l'utilisateur peut rejoindre.*	(Aucun)	Array *contenant des objets avec les attributs suivants :* difficulty: string, gameId: string, host: string, isPublic: boolean, mode: string, name: string,

			players: Map<string, number>, shareKey: string

* l'utilisateur ne peut rejoindre que des lobbys qui ne sont pas déjà pleins et dont la partie n'est pas déjà commencée.

Tableau 18: Rejoindre un lobby

Nom de la requête	Description	Paramètres	Valeur de retour
joinLobby	Le client ajoute l'utilisateur à la liste des players d'un lobby existant.	lobbyId: string, players: Map<string, number>*	(Aucun)

* players a été mis à jour pour contenir l'utilisateur souhaitant rejoindre le lobby, ainsi que sa position dans le lobby.

Tableau 19: Quitter un lobby

Nom de la requête	Description	Paramètres	Valeur de retour
quitLobby	Le client enlève l'utilisateur de la liste des players d'un lobby existant.	lobbyId: string, players: Map<string, number>*	(Aucun)

* players a été mis à jour pour contenir l'utilisateur souhaitant rejoindre le lobby, ainsi que sa position dans le lobby.

3.3.4 Gestion des parties Classiques

La création et la mise à jour des *parties classiques* est gérée exclusivement par les Cloud Functions. Les clients ne font qu'observer et réagir aux changements. Une *partie classique* contient les attributs suivants :

- currentImageId: string, (ID de l'image qui est présentement dessinée)
- difficulty: string,
- host: string,
- mode: string, (PPvPP, RPvPP, etc. voir description dans section 3.3.3)
- nextEvent: Date, (Timestamp de la fin de l'état actuel. Utilisé par la machine à états)
- players: Map<string, number>, (Map mettant en relation les joueurs et leur position)
- roles: Map<string, number>, (Map mettant en relation les joueurs et leur rôle: dessinateur, etc.)
- round: number,
- scores: Map<string, number>, (Map mettant en relation les joueurs et leur score)
- start: Date,
- state: number, (État de la machine à états)
- users: string[] (Liste des joueurs. Utile pour filtrer l'historique des joueurs.)

3.3.5 Gestion des parties Solo Sprint

Similairement au mode Classique, la création et la mise à jour des parties Sprint Solo est gérée exclusivement par les Cloud Functions. Les clients ne font qu'observer et réagir aux changements. Une partie Sprint Solo contient les attributs suivants :

- correctAttempts: number, (nombre de bon guess)
- currentImageId: string, (ID de l'image qui est présentement dessinée)
- difficulty: string,
- end: Date,
- host: string,
- score: number,
- start: Date,
- state: number, (État de la machine à états)
- totalAttempts: number (nombre de guess totaux)

3.3.6 Gestion des traits

Lors d'une partie Classique, la synchronisation des traits passe par une collection dans Firestore. Lorsqu'une partie classique démarre, une Cloud Function va créer un trait ayant le même id que la partie. Les clients vont s'abonner aux changements sur ce trait. Lorsqu'un joueur dessine, en réalité il met à jour ce trait (ou des segments du trait pour donner une impression de dessin en temps réel). Les autres clients sont notifiés du changement et vont le reproduire de leur côté en temps réel.

Tableau 20: Mise à jour des traits

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
sendStrokes	Le client met à jour les attributs du dernier trait (ou segment de trait) qui est dessiné.	id: string, path: Pair<number, number>[]*, color: string, size: number, opacity: number, type: StrokeType**	(Aucune)

* Le path est un tableau d'objets ayant les attributs { x: number, y: number }. Le path représente les coordonnées des points tracés par leur joueur qui dessine.

** StrokeType est un enum pouvant avoir les valeurs suivantes : Added, Done, Erased, Undo, Redo ou SnapshotUpdate. Cette enumération permet de synchroniser les instructions pour reproduire le dessin à travers tous les clients abonnés sur le trait. Ainsi, les clients reçoivent en réalité une séquence d'instructions permettant de reproduire le trait de crayon, le trait d'effacer ou l'utilisation du undo-redo de leur côté.

Il est à noter qu'un trait ne représente pas nécessairement un segment complet. Un trait pourrait être une partie du segment, puisque ceux-ci sont envoyés à des intervalles régulières dans le temps, créant une impression de dessin en temps réel.

Tableau 21: Réception des mises à jour de trait

Nom de la requête	Description	Paramètres	Valeur de retour
getStrokes	Le client s'abonne pour recevoir les mises à jour des traits envoyés par le dessinateur.	id: string	id: string, path: Pair<number, number>[], color: string, size: number, opacity: number, type: StrokeType

3.3.7 Gestion de la librairie de paires mot-image

Durant une partie (Classique ou Solo Sprint), l'attribut currentImageId décrit aux sections 3.3.4 et 3.3.5 permet de retrouver l'image qui est en train d'être dessinée. Cette image provient d'une libraire de paires mots-images créés préalablement. Lors d'un changement de round, une Cloud Function s'occupera de remplacer currentImageId par l'id d'une nouvelle image de même difficulté. Les clients seront notifiés du changement et iront chercher les détails de la nouvelle image. Cette dernière pourra alors être utilisée par les joueurs virtuels ou pour demander des indices.

Tableau 22: Ajouter une paire mots-images dans la base de données

Nom de la requête	Description	Paramètres	Valeur de retour
addImage	Le client ajoute une entrée de paire mot-image à la collection de Firestore.	authorId: string, word: string, difficulty: string, hints: string[], paths: Pair<number, number>[]*	(aucune)

* Similaire au paths décrits à la section 3.3.6

Tableau 23: Rechercher une paire mots-images dans la base de données

Nom de la requête	Description	Paramètres	Valeur de retour
getImage	Le client cherche une paire de mots-images dans la collection de Firestore.	id: string*	authorId: string, word: string, difficulty: string, hints: string[], paths: Pair<number, number>[]*

* id proviendrait de currentImageId dans une partie Classique ou Solo Sprint

3.3.8 Gestion des profils utilisateurs

Le client s'abonne aux changements sur le profil de l'utilisateur en cours. Il pourra ainsi mettre à jour l'interface en temps réel en cas de changement (par exemple lorsque l'utilisateur rejoint un canal de discussion). L'utilisateur pourra aussi modifier certaines préférences sur son profil, tel que le fait d'activer ou non la musique d'arrière-plan ou la voix des joueurs virtuels. De plus, l'ajout de l'expérience et de la monnaie virtuelle en fin de partie est géré par une Cloud Function.

Tableau 24: Consulter le profil utilisateur d'un joueur

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
getProfile	Le client envoie à Firestore une requête pour le profil d'utilisateur. Celui-ci est retrouvé avec le nom d'utilisateur unique passé en paramètre.	username: string	avatar: string, channels: string[], experience: number, firstName: string, lastName: string, money: number, selectedBadges: string[], toggleMusic: boolean, toggleVP: boolean, unlockedAvatars: string[], unlockedBadges: string[]

Tableau 25: Modifier les badges

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
setBadgeList	Le client envoie à Firestore une requête pour modifier les badges attachés au profil utilisateur.	username: string, selectedBadges: string[]	(Aucune)

Tableau 26: Débloquer un badge

Nom de la requête	**Description**	**Paramètres**	**Valeur de retour**
unlockBadge	Le client envoie à Firestore une requête pour modifier la liste des badges qu'il	username: string, unlockedBadges: string[]	(Aucune)

	possède après en avoir débloqué un nouveau.		

Tableau 27: Modifier l'avatar

Nom de la requête	Description	Paramètres	Valeur de retour
setAvatar	Le client envoie à Firestore une requête pour modifier l'avatar attaché au profil utilisateur.	username: string avatar: string	(Aucune)

Tableau 28: Acheter un avatar du magasin

Nom de la requête	Description	Paramètres	Valeur de retour
buyAvatar	Le client envoie à Firestore une requête pour modifier la liste d'avatars qu'il possède.	username: string, unlockedAvatars: string[], money: number	(Aucune)

Tableau 29: Activer ou désactiver la musique d'arrière-plan

Nom de la requête	Description	Paramètres	Valeur de retour
toggleMusic	Le client envoie à Firestore une requête pour modifier ses préférences pour la musique d'arrière-plan.	username: string toggleMusic: boolean	(Aucune)

Tableau 30: Activer ou désactiver la voix des joueurs virtuels

Nom de la requête	Description	Paramètres	Valeur de retour
toggleVirtualPlayerVoice	Le client envoie à	username: string	(Aucune)

	Firestore une requête pour modifier ses préférences pour la voix des joueurs virtuels.	toggleVP: boolean	

3.3.9 Gestion de l'historique de connexion

Les méthodes d'enregistrement des connexions et déconnexions permettent d'afficher l'historique de connexion de l'utilisateur sur son profil. Ces méthodes sont appelées automatiquement par le client lorsque l'utilisateur ouvre ou ferme l'application.

Tableau 31: Ajout d'une entrée à l'historique de connexion

Nom de la requête	Description	Paramètres	Valeur de retour
addLogin	Le client ajoute une entrée d'historique de connexion dans la collection de Firebase.	login: Date, user: string	(Aucune)

Tableau 32: Ajout d'une entrée à l'historique de déconnexion

Nom de la requête	Description	Paramètres	Valeur de retour
updateLogoutTime	Le client met à jour son entrée d'historique de connexion dans la collection de Firebase pour y indiquer sa date de déconnexion.	logoff: Date user: string	(Aucune)

3.4 Fire Authentification

La gestion des comptes se fait dans un service indépendant appelé Fire Authentication. Les clients s'y connectent directement à travers son API. Les attributs du profil client (autre que le nom d'utilisateur et le mot de passe) sont stockés dans Firestore.

3.4.1 Création de compte

Tableau 33: Ajouter un compte dans le Fire Authentification

Nom de la requête	Description	Paramètres	Valeur de retour
createProfile	Le client envoie à Fire Authentification une requête pour ajouter un profil d'utilisateur.	avatar: string, channels: string[], experience: number, firstName: string, lastName: string, money: number, selectedBadges: string[], toggleMusic: boolean, toggleVP: boolean, unlockedAvatars: string[], unlockedBadges: string[]	(Aucune)

3.4.2 Connexion

Tableau 34: Authentifier le compte lors d'une connexion

Nom de la requête	Description	Paramètres	Valeur de retour
logIn	Le client envoie à Fire Authentification une requête pour authentifier la connexion de l'utilisateur.	username: string password: string	(Aucune)

3.4.3 Déconnexion

Tableau 35: Déconnexion d'un compte

Nom de la requête	Description	Paramètres	Valeur de retour
logOut	Le client envoie à Fire Authentification une requête pour détruire son token de connexion.	username: string	(Aucune)

3.5 Traitement des erreurs

Lorsqu'une valeur de retour est *void* pour les appels au serveur, les requêtes écoutent toujours pour des erreurs. Les appels sont envoyés dans une *Promise,* alors la valeur de retour est *Promise<void>* pour les requêtes et cela garantit un comportement contrôlé lors d'une erreur. La même chose est ajoutée aux requêtes qui ont une valeur de retour autre que *void*. Ainsi, il est impossible pour l'application d'entrée dans un état indéfinie à la suite d'une valeur de retour erronée.