205

Fais-moi un dessin
Protocole de communication

Version 1.4

Historique des révisions

Date	Version	Description	Auteur
2021-02-09	1.0	Première ébauche du document	Augustin Bouchard
2021-02-10	1.1	Ajout des deux diagrammes + texte dans la section communication client-serveur	Simon Ayotte
2021-02-17	1.2	Ajout plusieurs descriptions de paquets	Guilhem Dubois
2021-02-18	1.3	Ajout diagrammes paquets + diagrammes web sockets	Simon Ayotte
2021-02-19	1.4	Version finale avant la remise d'appel d'offre	Équipe 205

Table des matières

Protocole de communication

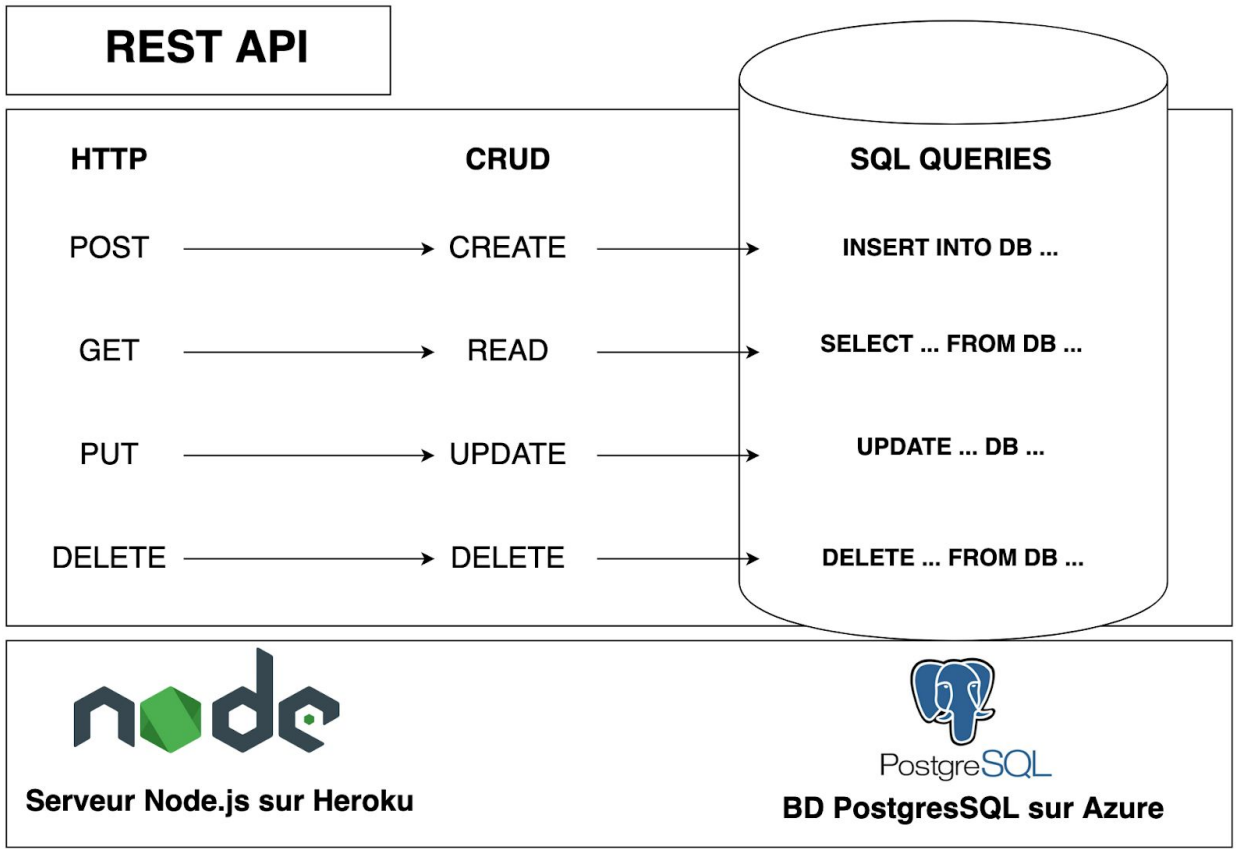
1. Introduction

Le présent document présente le protocole de communication de notre application *Fais moi un dessin.* On y présente des diagrammes facilitant la compréhension des communications client-serveur.

2. Communication client-serveur

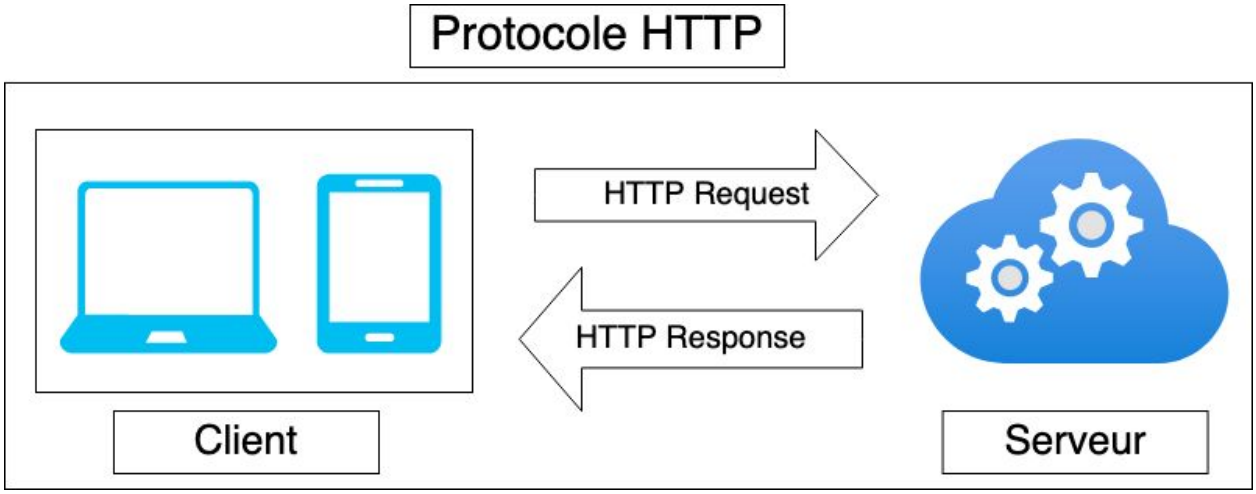
2.1 REST API, CRUD et protocole HTTP

La communication entre le serveur et les clients se fait principalement avec les requêtes HTTP (POST, GET, PUT, DELETE) qui sert d'interface entre notre serveur node.js et les clients (Electron/Android). Ces opérations font partie de la famille des opérations CRUD (CREATE, READ UPDATE, DELETE).



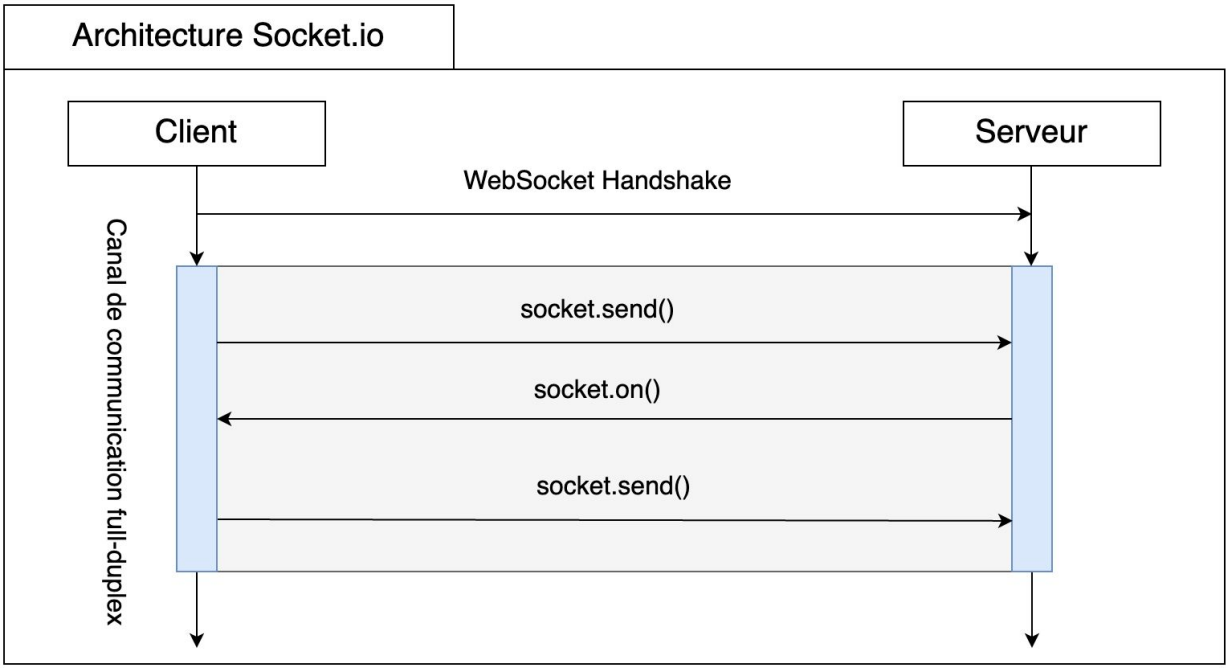
Le client peut donc envoyer des informations au serveur par l'entremise des requêtes POST. Il peut également recevoir de l'information du serveur par l'entremise de requête GET. Les requêtes sont par la suite gérer par le serveur qui s'assure de valider et d'envoyer des queries à la BD PostgreSQL si il le faut. Les requêtes suivent donc le protocole HTTP.

Pour la majorité des opérations entre les clients et le serveur où le client a besoin d'un accès à une information du serveur ou bien d'une validation du serveur, nous allons utiliser les ree



2.2 Communication avec WebSockets

Lorsque nous avons besoin d'effectuer plusieurs opérations de communication entre le client et le serveur dans un court laps de temps, nous optons plutôt pour l'utilisation des WebSockets dans notre application à l'aide de la librairie Socket.io. Nous allons particulièrement faire usage des WebSockets pour la fonctionnalité du clavardage ainsi que pour la transmission en temps réel des éléments SVG sur le canevas de dessin dans une partie.

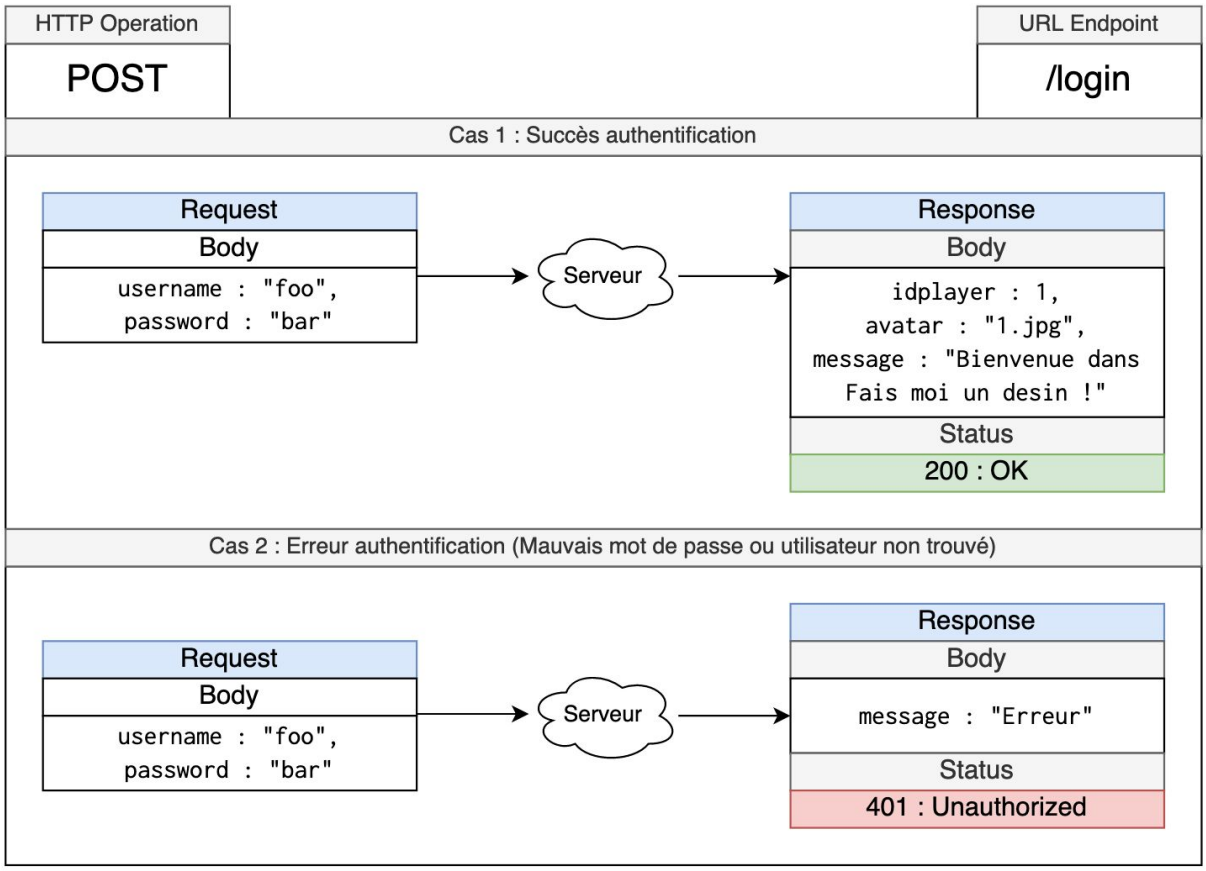


Les WebSockets nous permettent d'établir un canal de communication full-duplex entre le client et le serveur ce qui facilite la transmission de données pour certaines fonctionnalités. Socket.io fonctionne par événements, le client envoie donc des données sous la forme d'événements qui sont écoutés par le serveur. Par la suite, le serveur peut répondre au client.

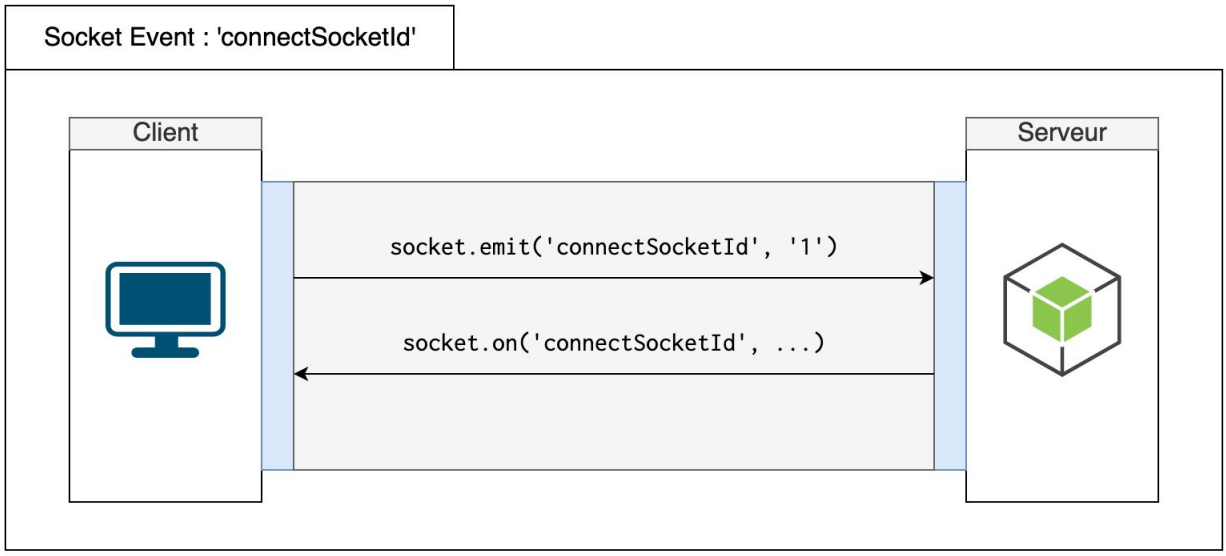
3. Description des paquets

Pour l'envoi des requêtes et des réponses HTTP, notre application privilégie le format JSON comme structure de communication étant donné que c'est une structure extrêmement flexible et facile d'utilisation, particulièrement avec le serveur et le client lourd qui sont écrit en TypeScript.

3.1 Connexion d'un utilisateur

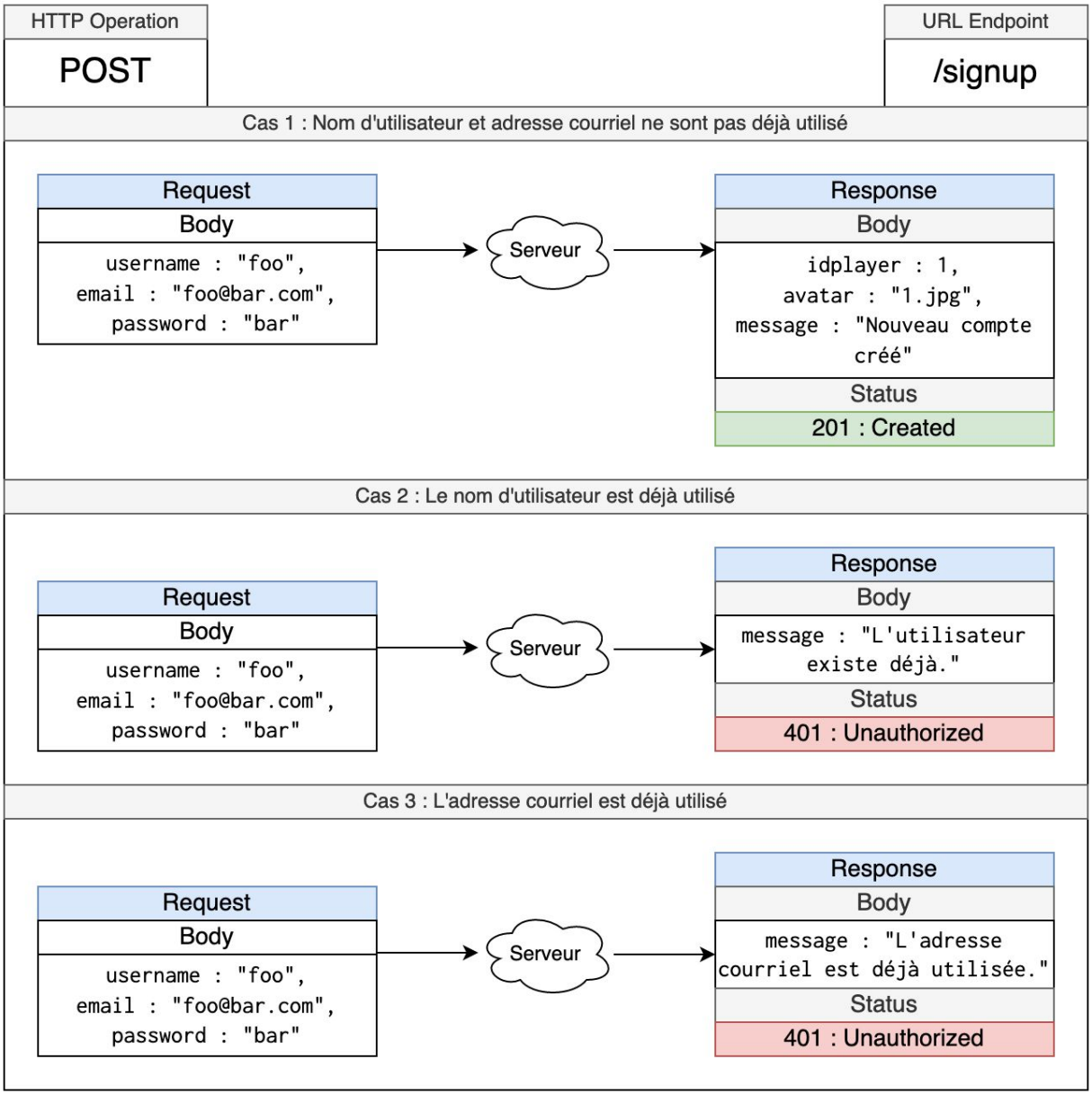


Une fois confirmé, le client démarre une connexion par WebSocket au serveur et envoie un événement afin que le serveur puisse associer la connexion à l'utilisateur:
Événement: connectSocketid
Méthode de requête: WebSocket
Data: idplayer: identifiant de l'utilisateur



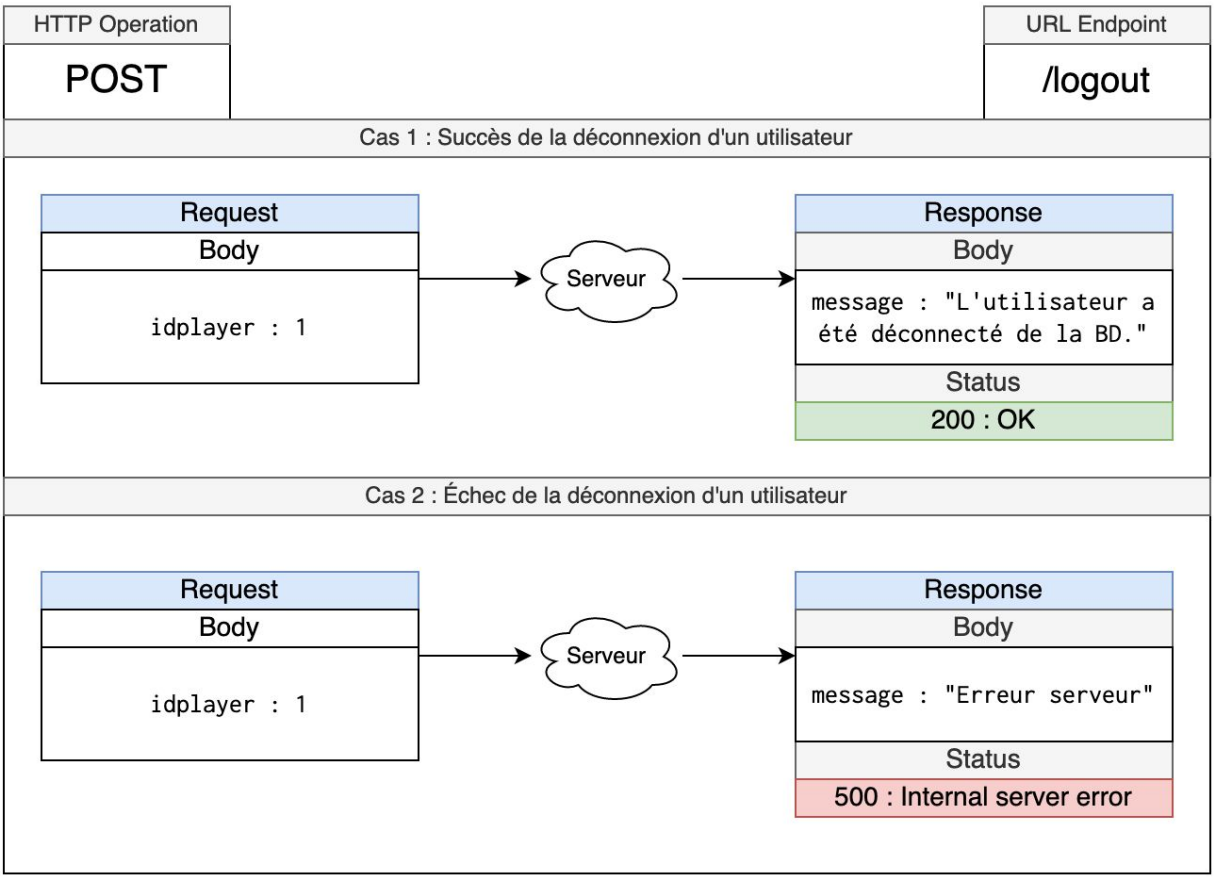
Par la suite, le serveur reconnaîtra que tout message envoyé par cette connexion WebSocket correspond à cet utilisateur.

3.2 Inscription d'un utilisateur



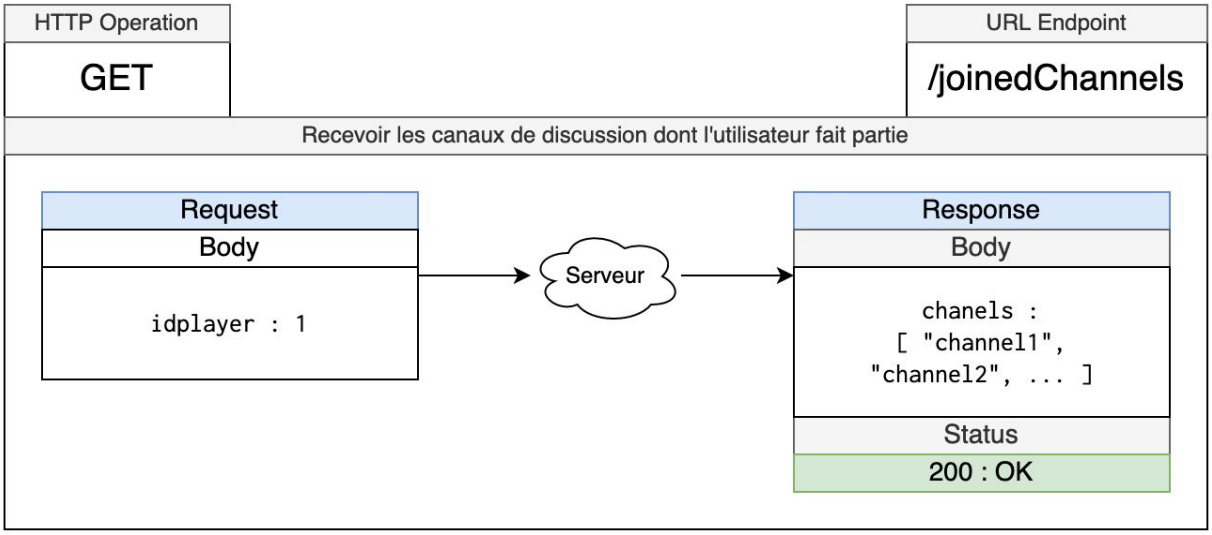
L'événement "connectSocketid" est ensuite envoyé tel qu'expliqué dans la section 3.1.

3.3 Déconnexion d'un utilisateur



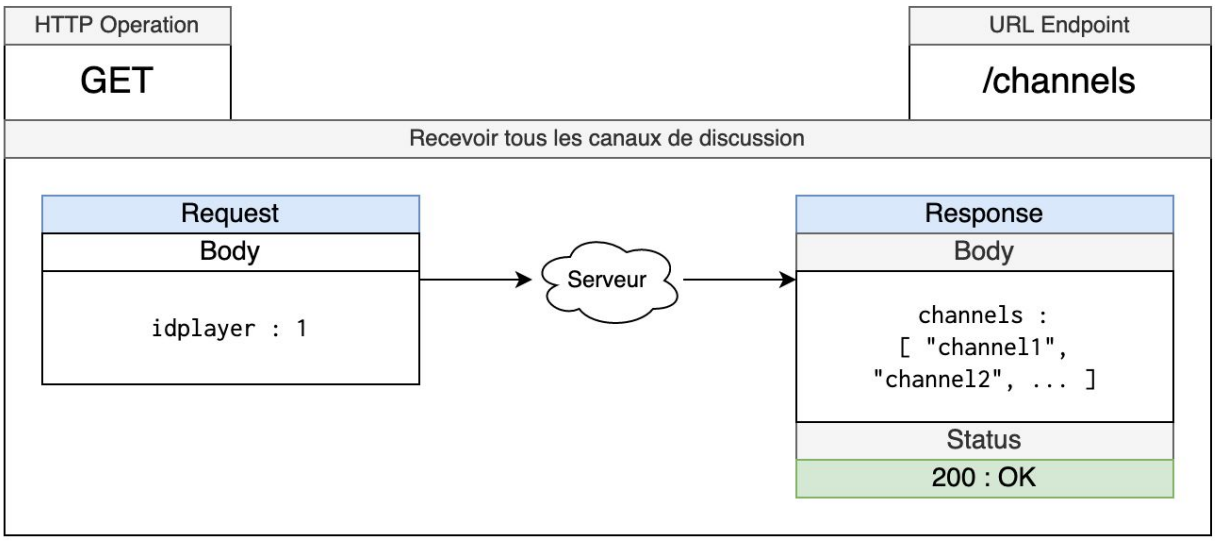
3.4 Réception des canaux de discussion

Lorsque l'utilisateur se connecte, il reçoit tous les canaux de discussion dont il fait partie:



Pour chaque canal, le client écoute un événement pour recevoir les messages du canal (voir section 3.9).

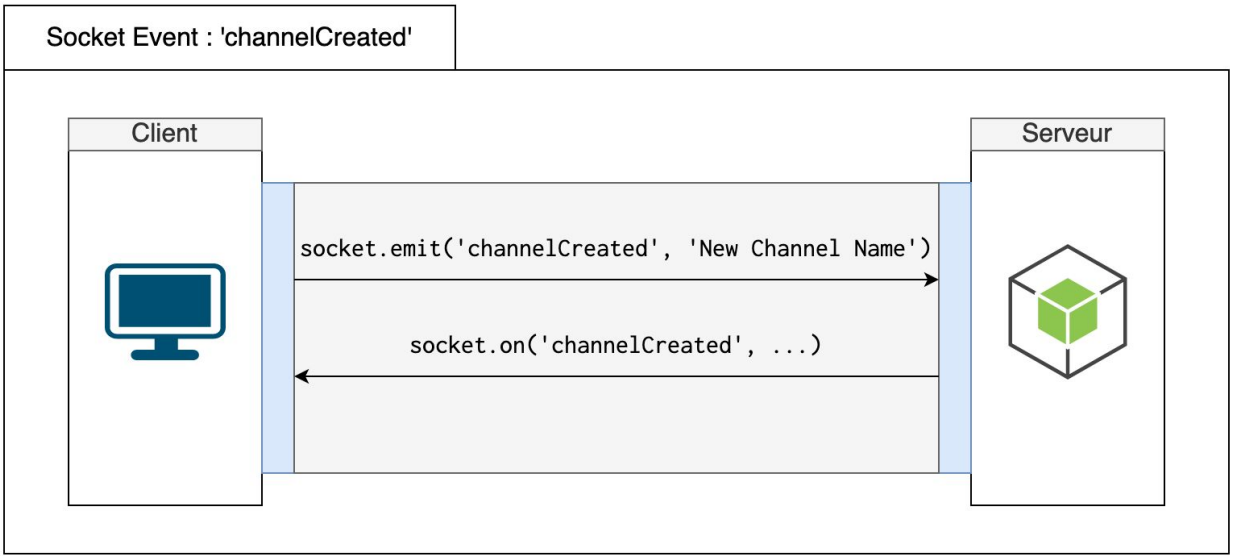
Il peut aussi recevoir tous les canaux de discussion dans l'application :



Le client peut alors s'inscrire aux évènements de création/suppression de canaux (voir sections 3.5 et 3.6)

3.5 Création d'un canal de discussion

Événement: channelCreated
Méthode de requête: WebSocket
Data: name: nom du canal



Par la suite, le client rejoint automatiquement le canal (voir section 3.7).

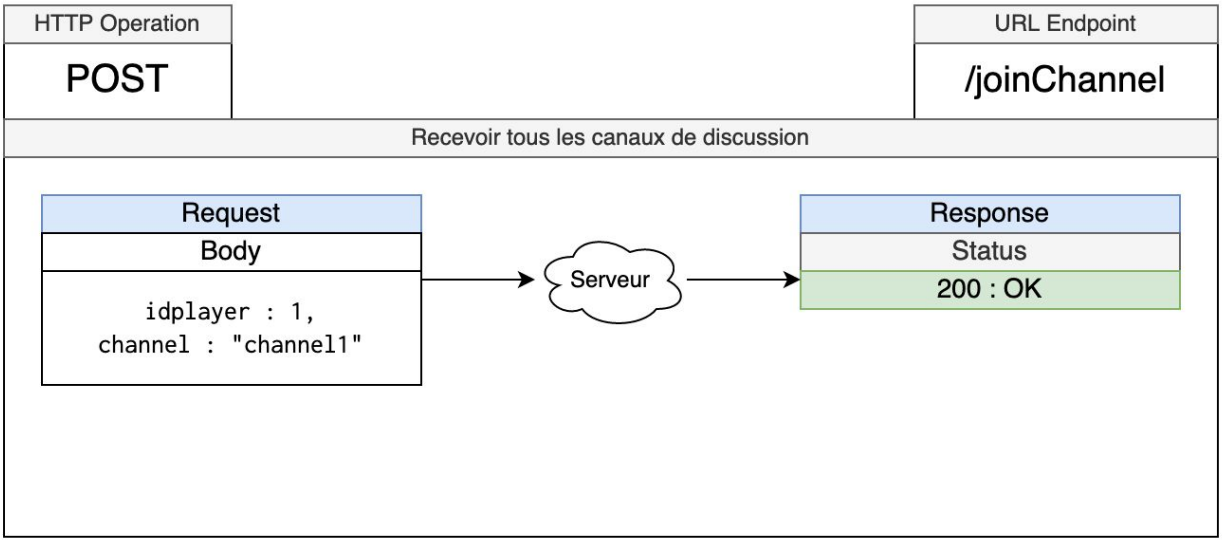
3.6 Suppression d'un canal de discussion

Événement: channelDeleted
Méthode de requête: WebSocket

Data: name: nom du canal

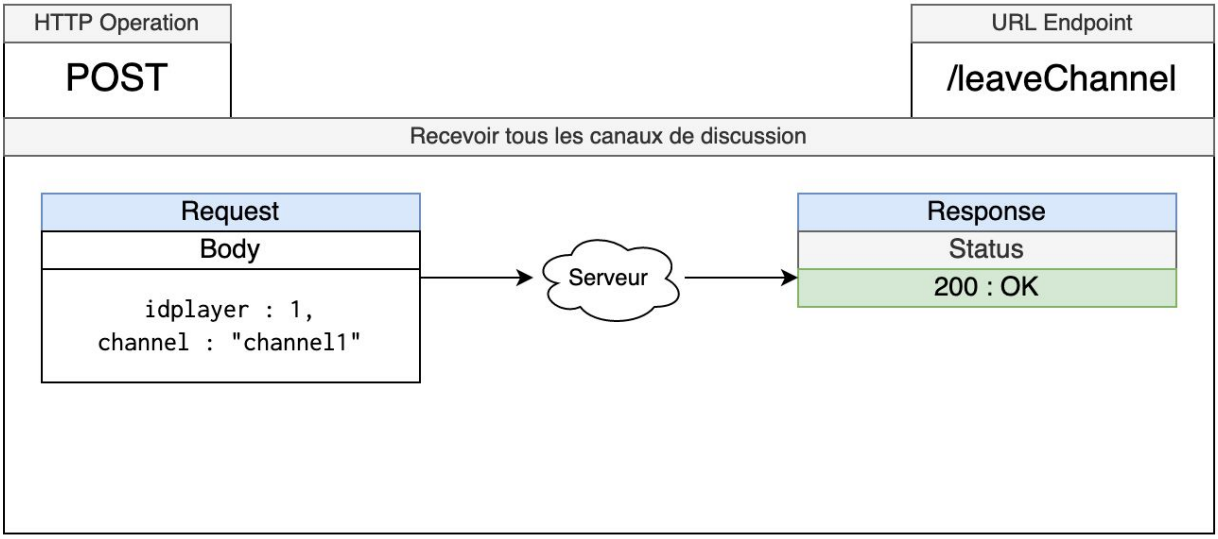
Le client arrête aussi d'écouter l'événement qui reçoit les messages du canal (voir section 3.9).

3.7 Joindre un canal de discussion



Par la suite, le client écoute l'événement pour recevoir les messages du canal (voir section 3.9).

3.8 Quitter un canal de discussion

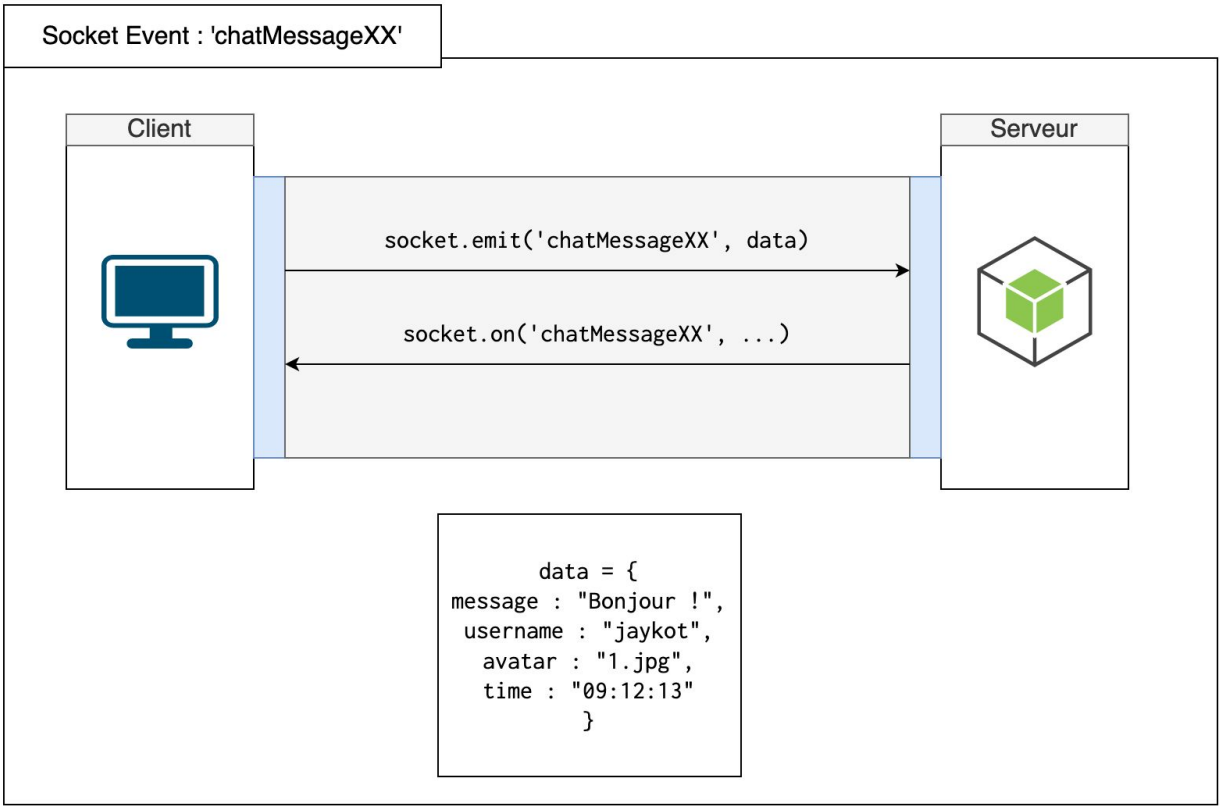


Le client arrête aussi d'écouter l'événement qui reçoit les messages du canal (voir section 3.9).

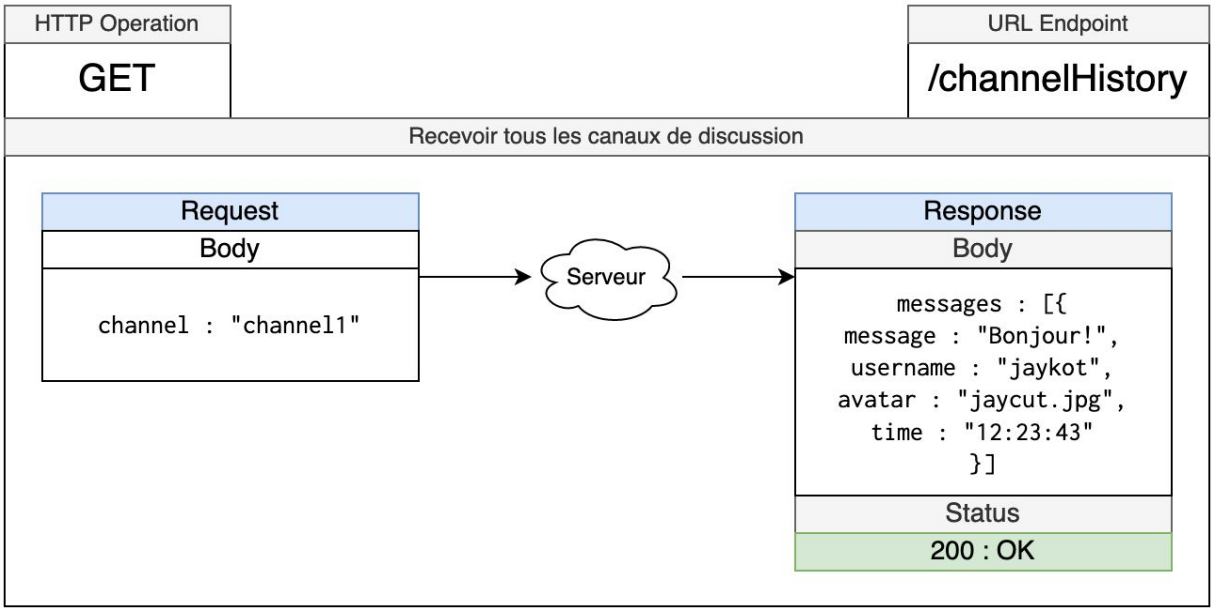
3.9 Envoi/Réception de messages

Chaque client passe par un même événement de WebSocket pour envoyer/recevoir des messages d'un canal de discussion. Le serveur reçoit le socket et le broadcast à tous les clients observant cet événement.

Événement: " chatMessageXX " où XX est l'identifiant du canal de discussion
Data: {
message: contenu du message
username: nom d'utilisateur de l'expéditeur
avatar: nom du fichier d'image de l'avatar de l'expéditeur
time: temps d'envoi sous format heure:minute:seconde
}



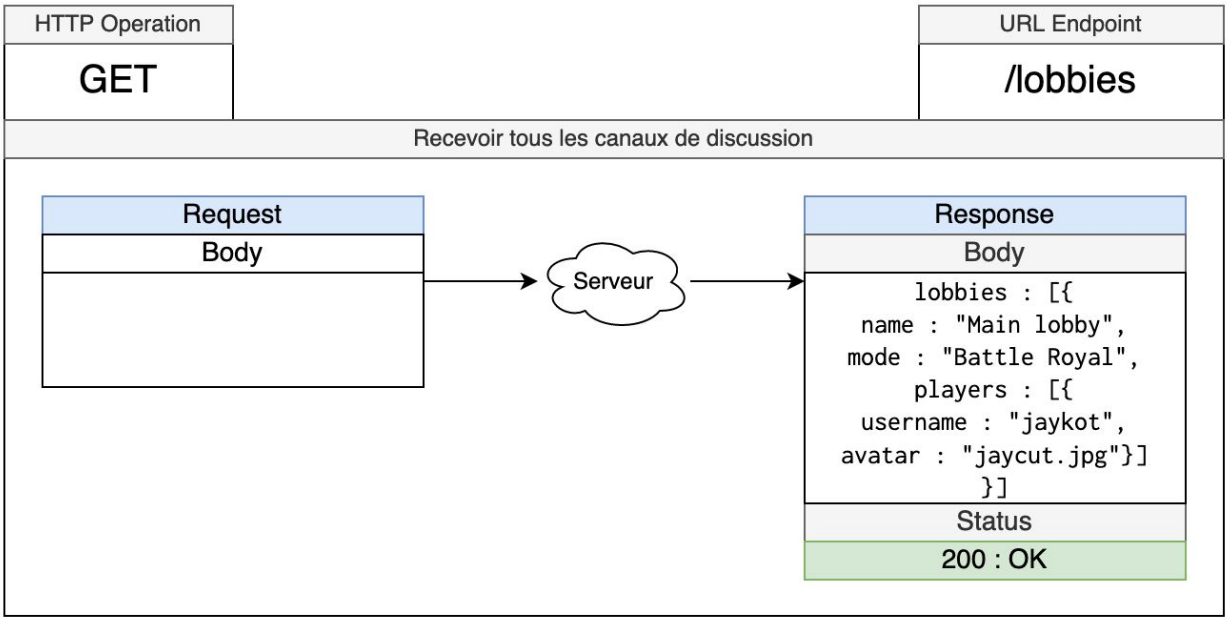
3.10 Historique des messages d'un canal de discussion



3.11 Réception des lobbys disponibles

Route: /lobbys

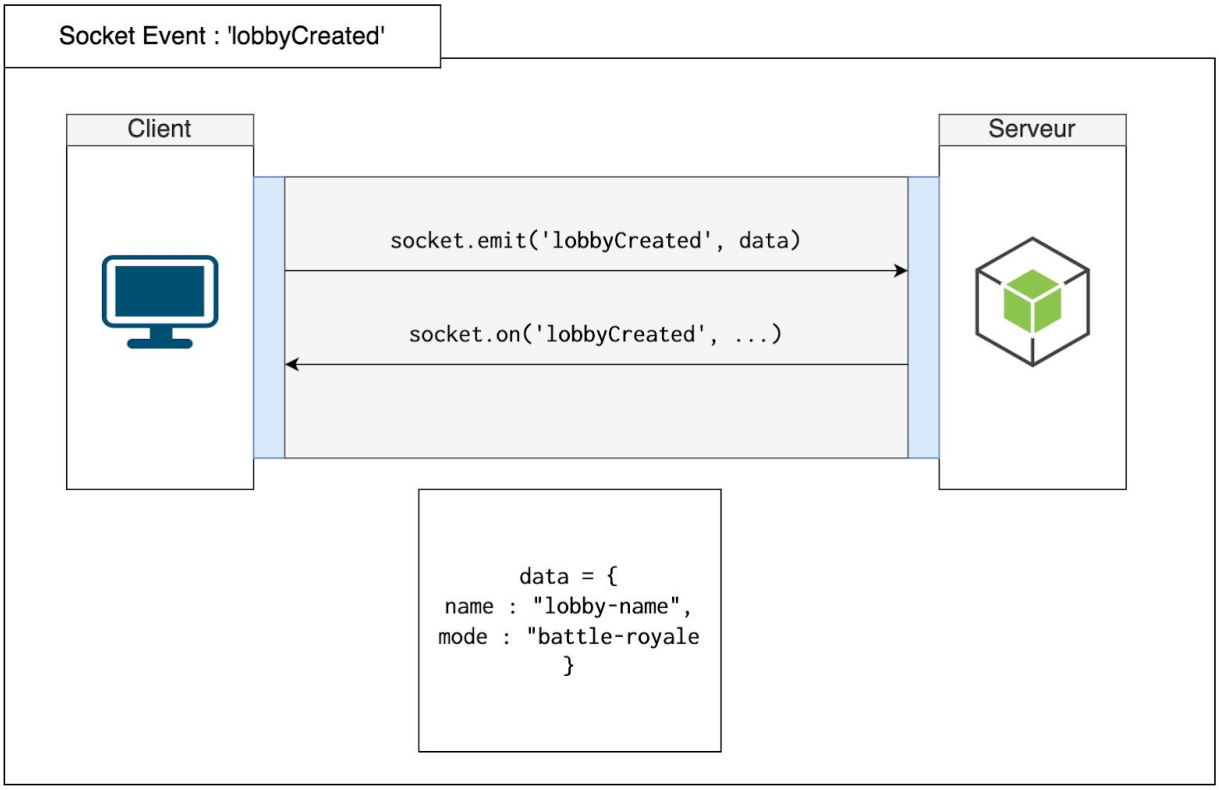
Méthode de requête: GET (HTTP)
Réponse:
Status: 201
Body: {
	lobbys: liste des lobbys de format {
		name: nom du lobby
		mode: identifiant représentant le type de lobby
		players: liste des joueurs dans le lobby de format {
			username: nom de l'utilisateur
			avatar: nom du fichier d'image de l'avatar de l'utilisateur
		}
	}
}



Le client peut alors s'inscrire aux évènements de création/suppression de lobbys (voir sections 3.12 et 3.13)

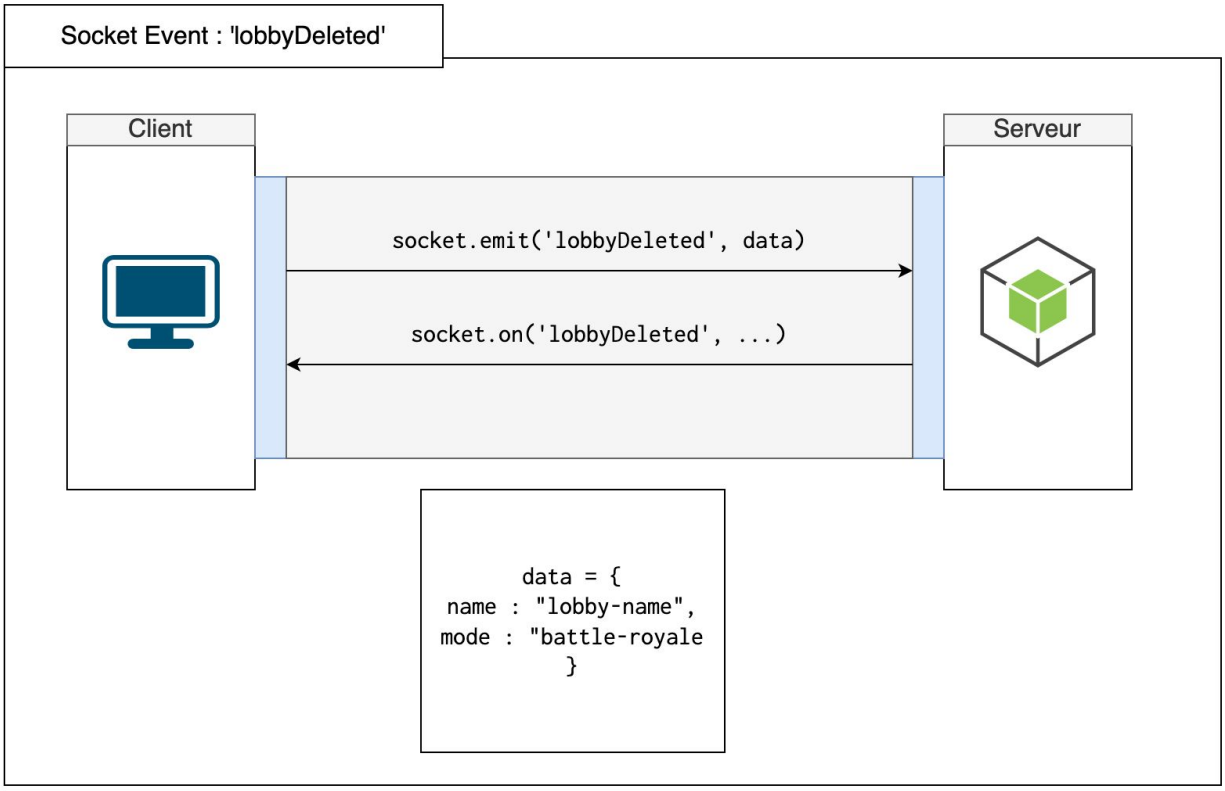
3.12 Création d'un lobby

Événement: lobbyCreated
Méthode de requête: WebSocket
Data: {
	name: nom du lobby
	mode: type de lobby
}



3.13 Suppression d'un lobby

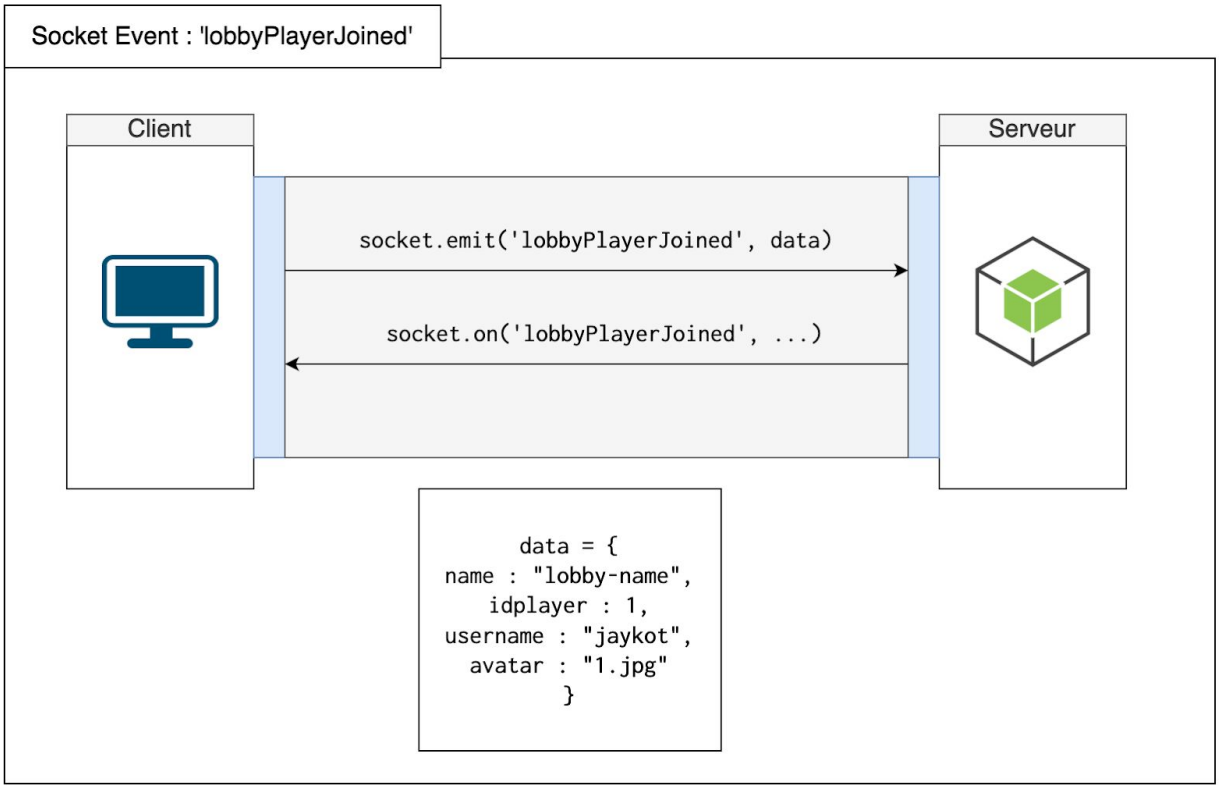
Événement: lobbyDeleted
Méthode de requête: WebSocket
Data: name: nom du lobby



3.14 Joindre un lobby

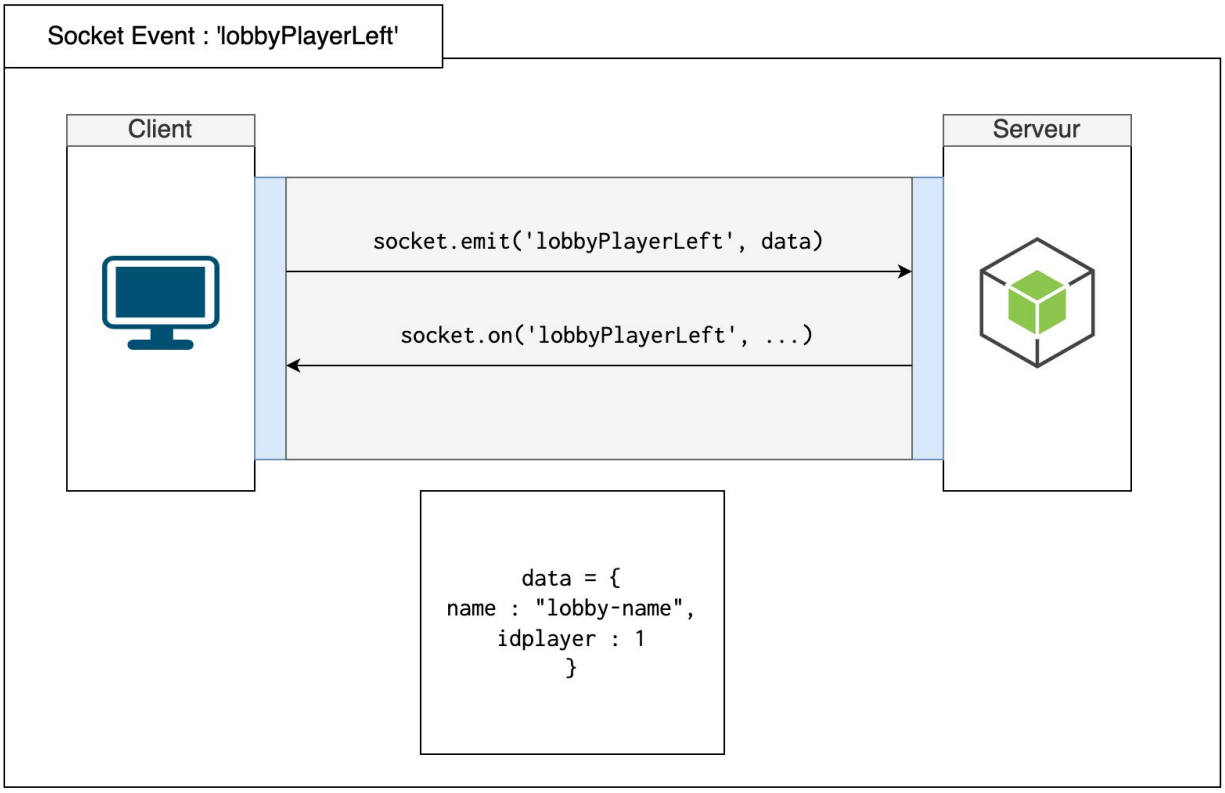
Événement: lobbyPlayerJoined
Méthode de requête: WebSocket
Data: {
name: nom du lobby
idplayer: identifiant de l'utilisateur
username: nom d'utilisateur
avatar: nom du fichier d'image de l'avatar de l'utilisateur

}

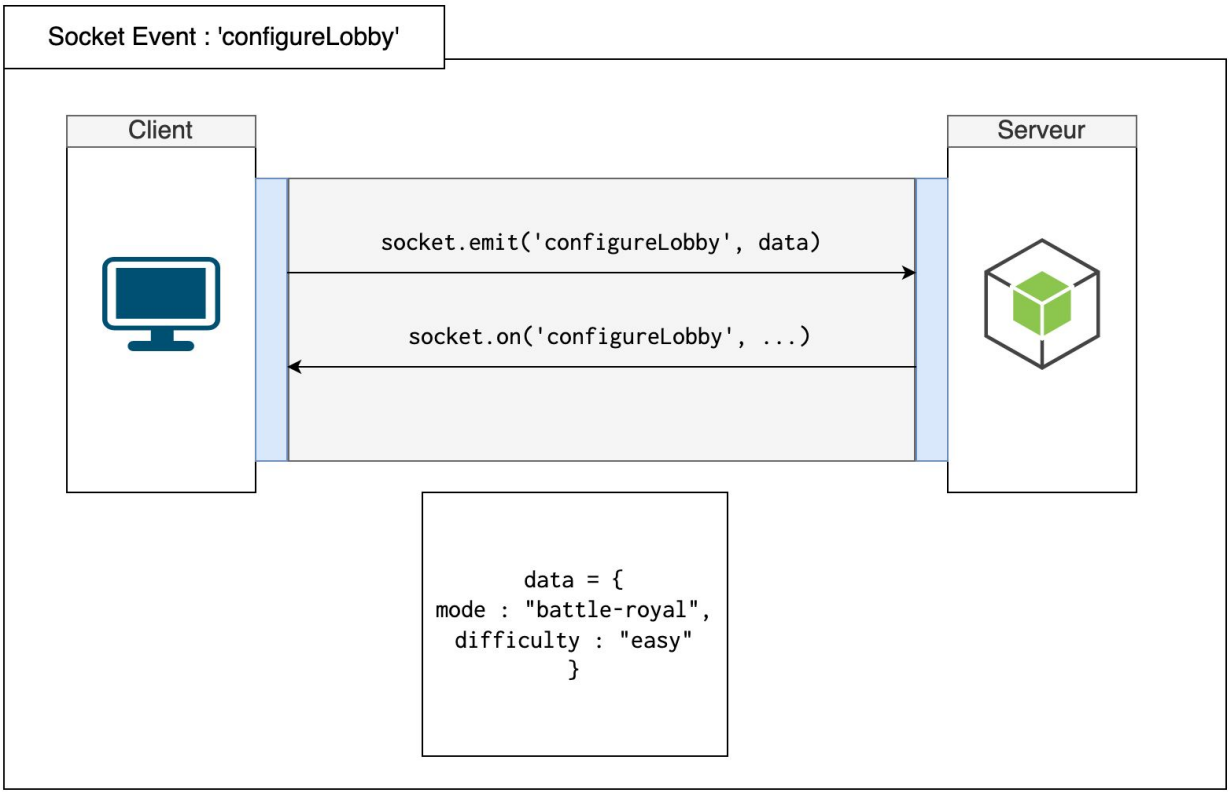


3.15 Quitter un lobby

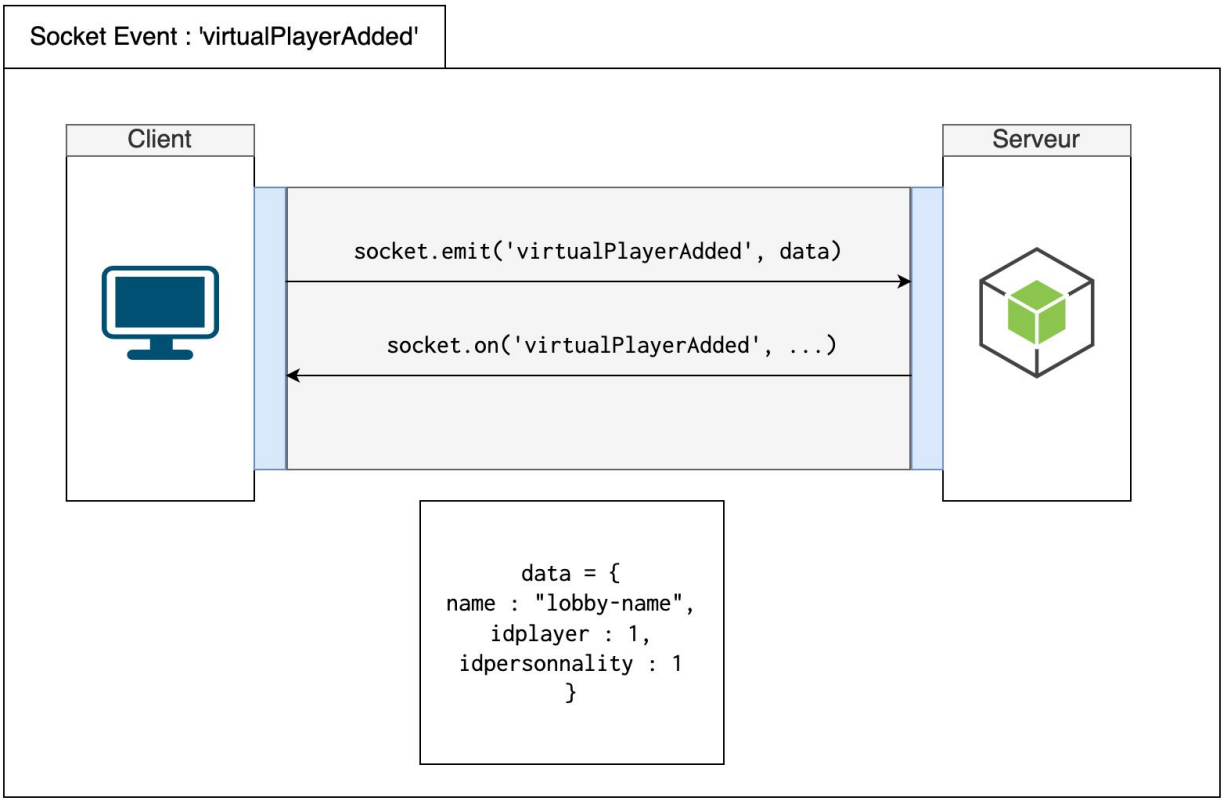
Événement: lobbyPlayerLeft
Méthode de requête: WebSocket
Data: {
name: nom du lobby
idplayer: identifiant de l'utilisateur
}


Socket Event : 'lobbyPlayerLeft'
Client
Serveur
socket.emit('lobbyPlayerLeft', data)
socket.on('lobbyPlayerLeft', ...)
data = {
name : "lobby-name",
idplayer : 1
}

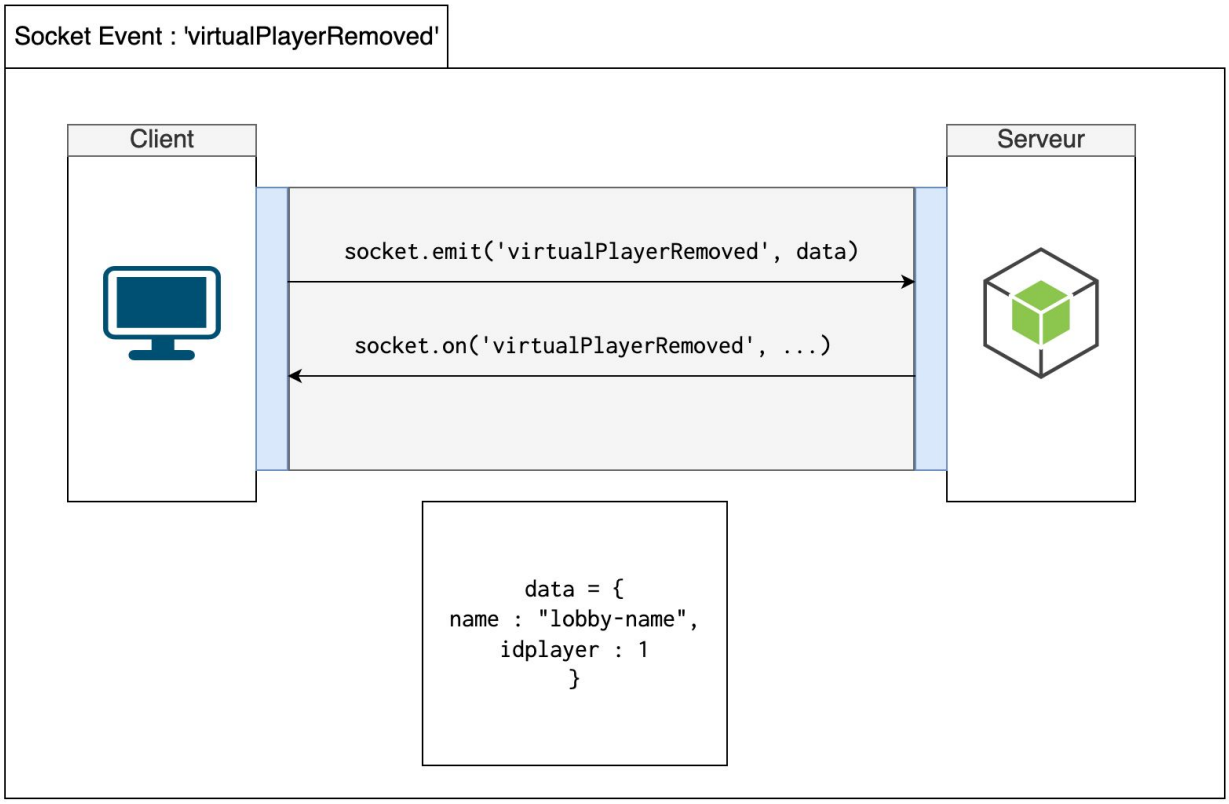
3.16 Configuration d’un lobby



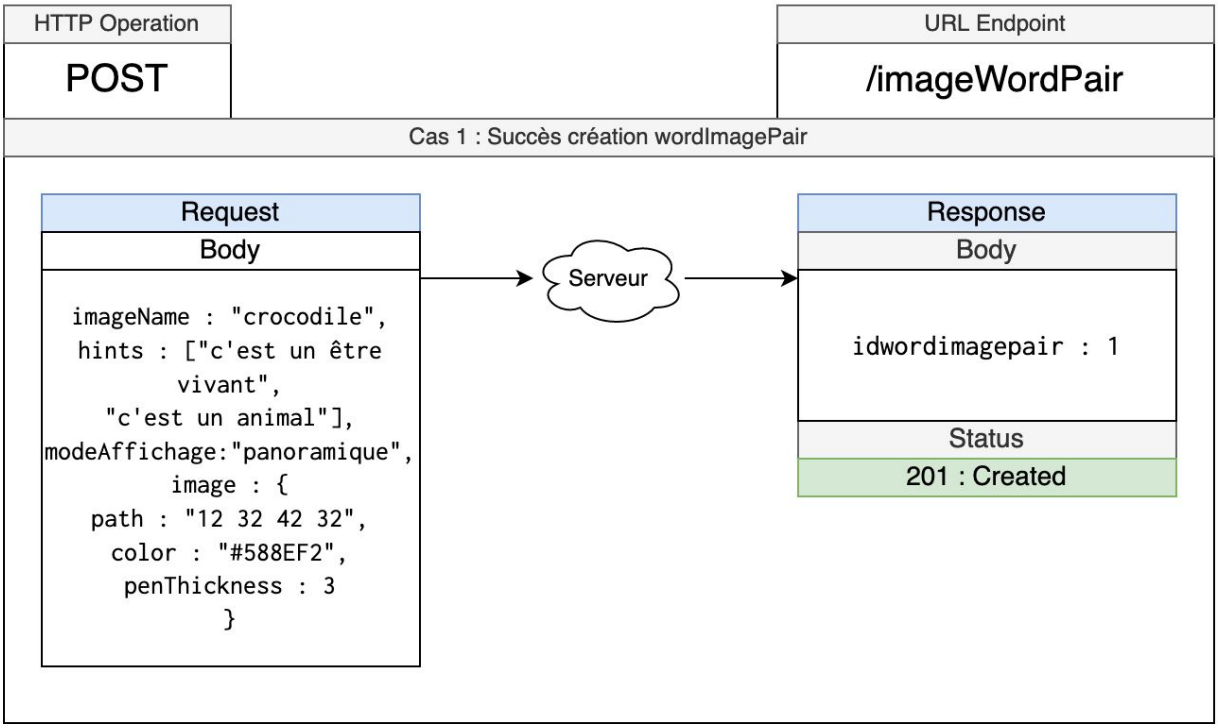
3.16.1 Ajouter un joueur virtuel au lobby



3.16.2 Enlever un joueur virtuel au lobby

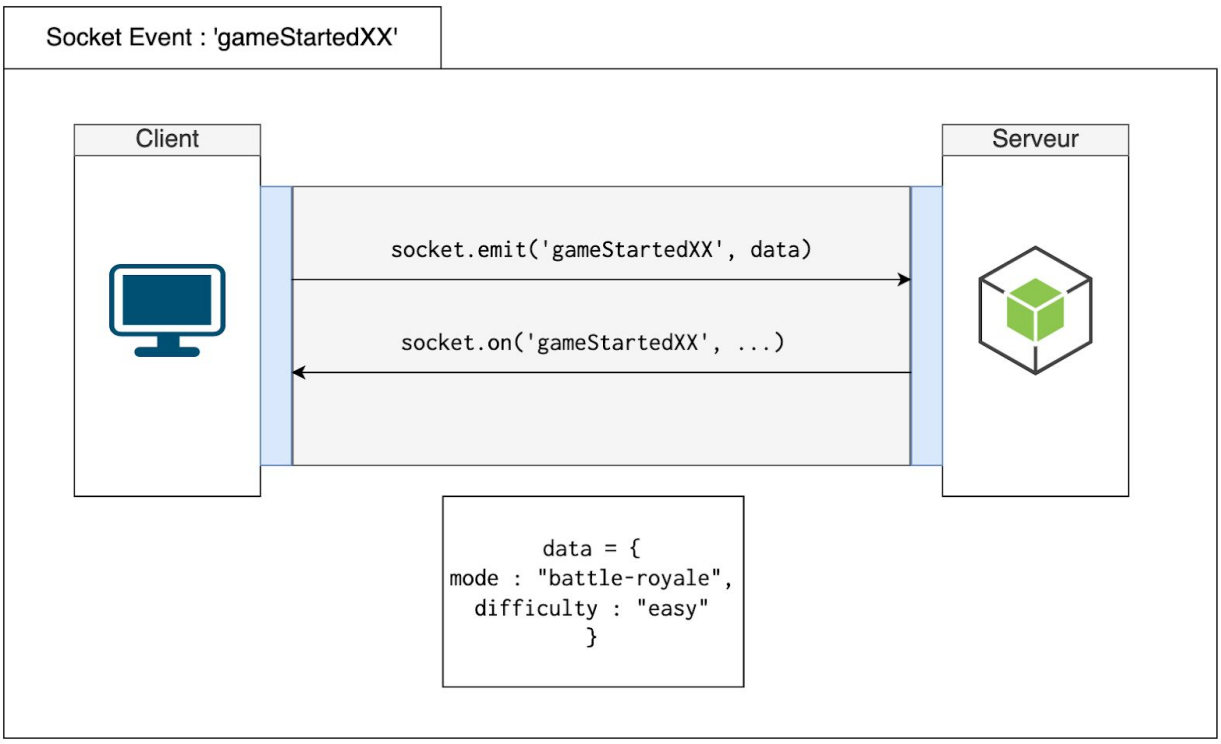


3.17 Création d'une paire mot-image

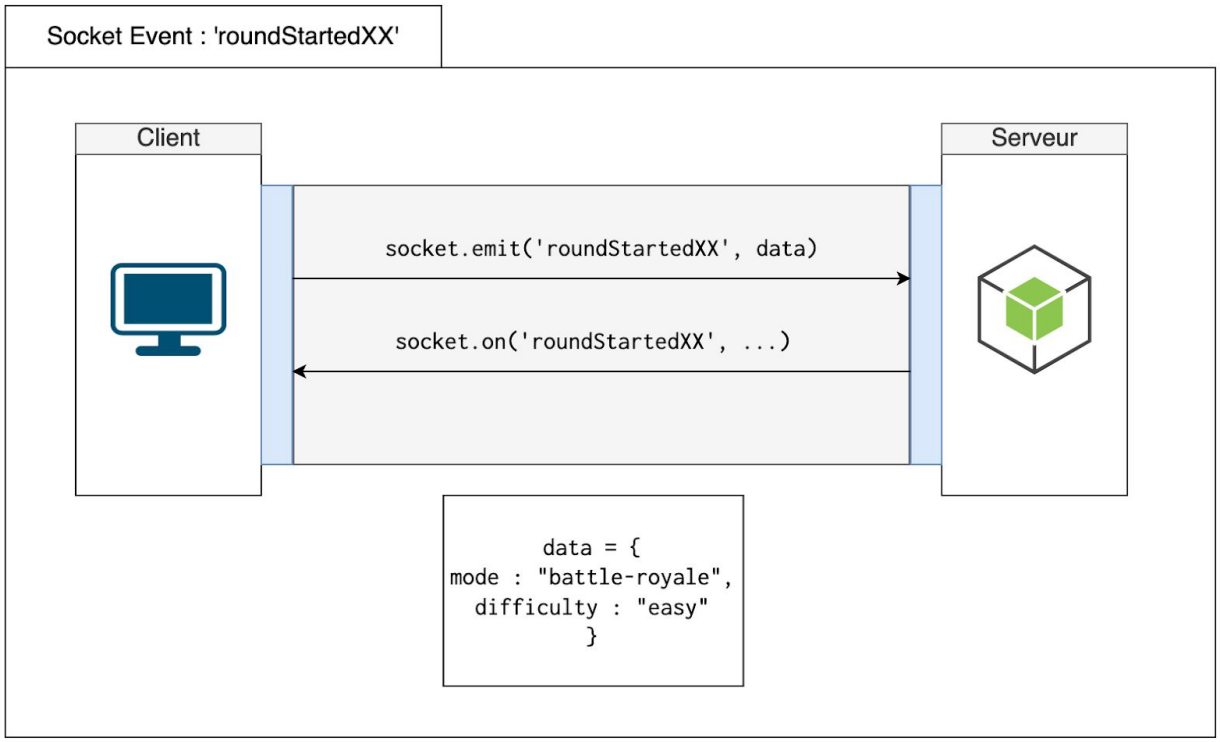


3.18 Déroulement d’une partie

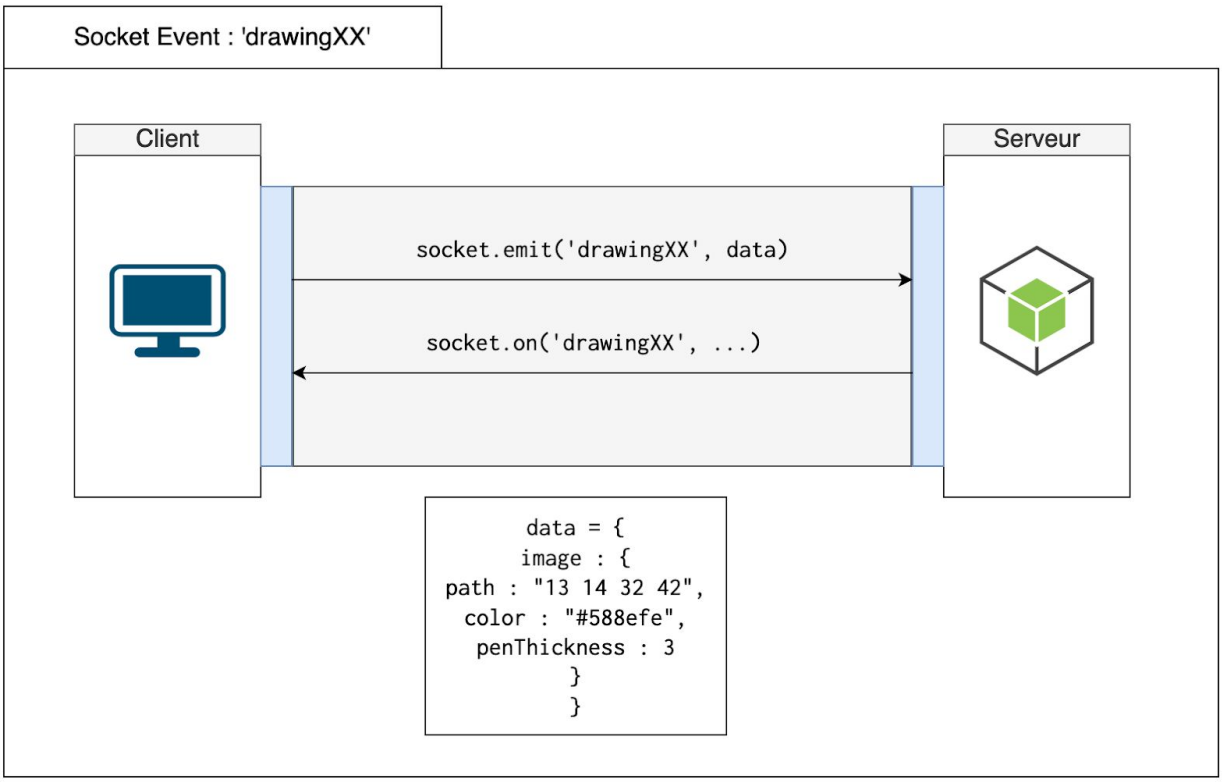
3.18.1 Début d’une partie



3.18.2 Début d’une manche

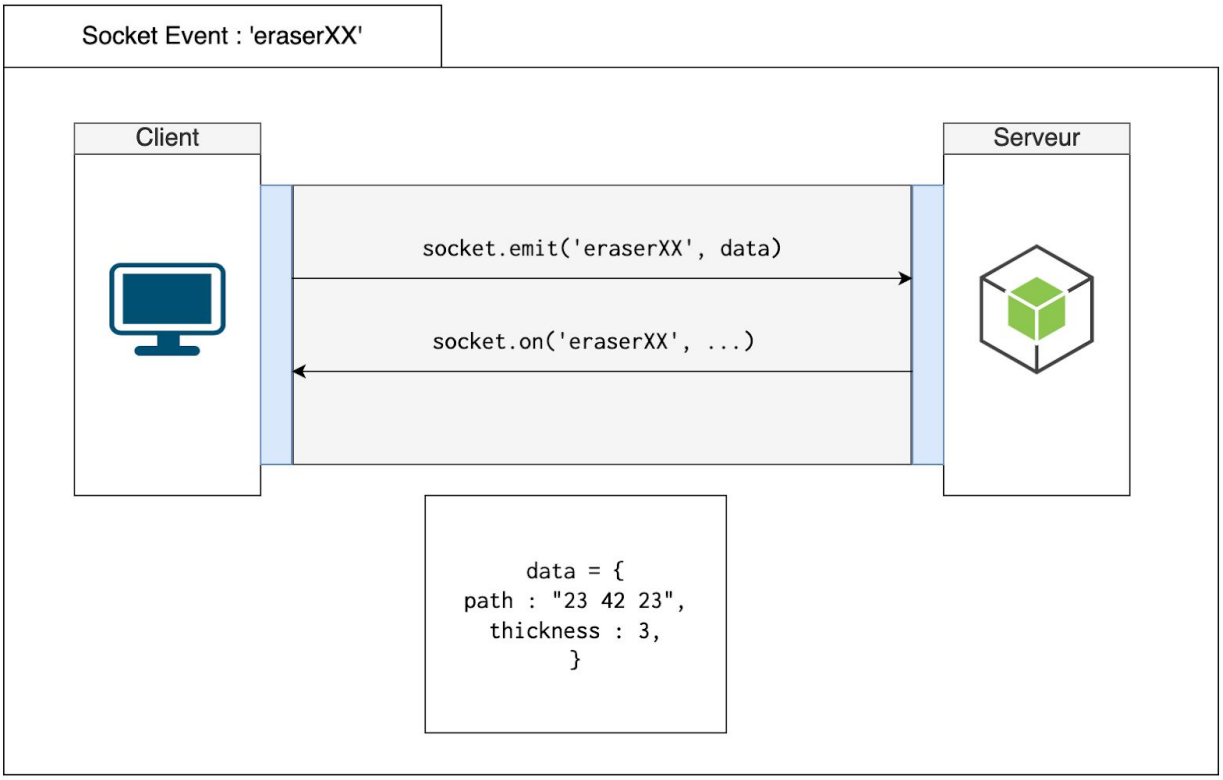


3.18.3 Envoie du dessin aux joueurs dans le lobby



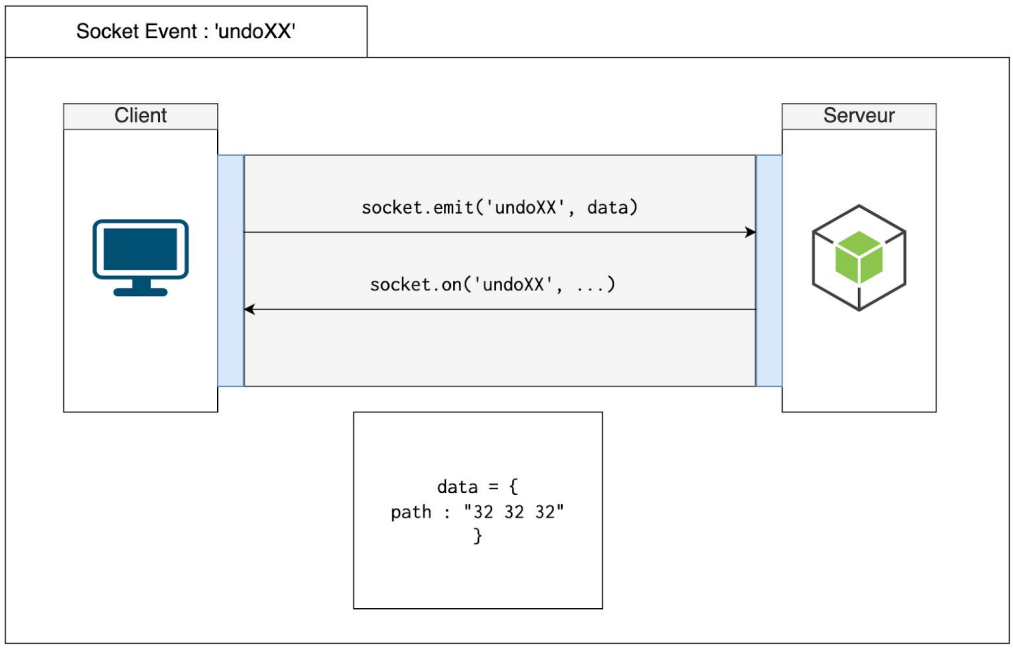
Pour l'envoie du dessin, le "XX" à la fin du drawing correspond à l'id du lobby de la partie. C'est un event qui sera envoyé assez souvent par le client pour assurer un fluidité dans la partie.

3.18.4 Utilisation de l’efface dans le dessin

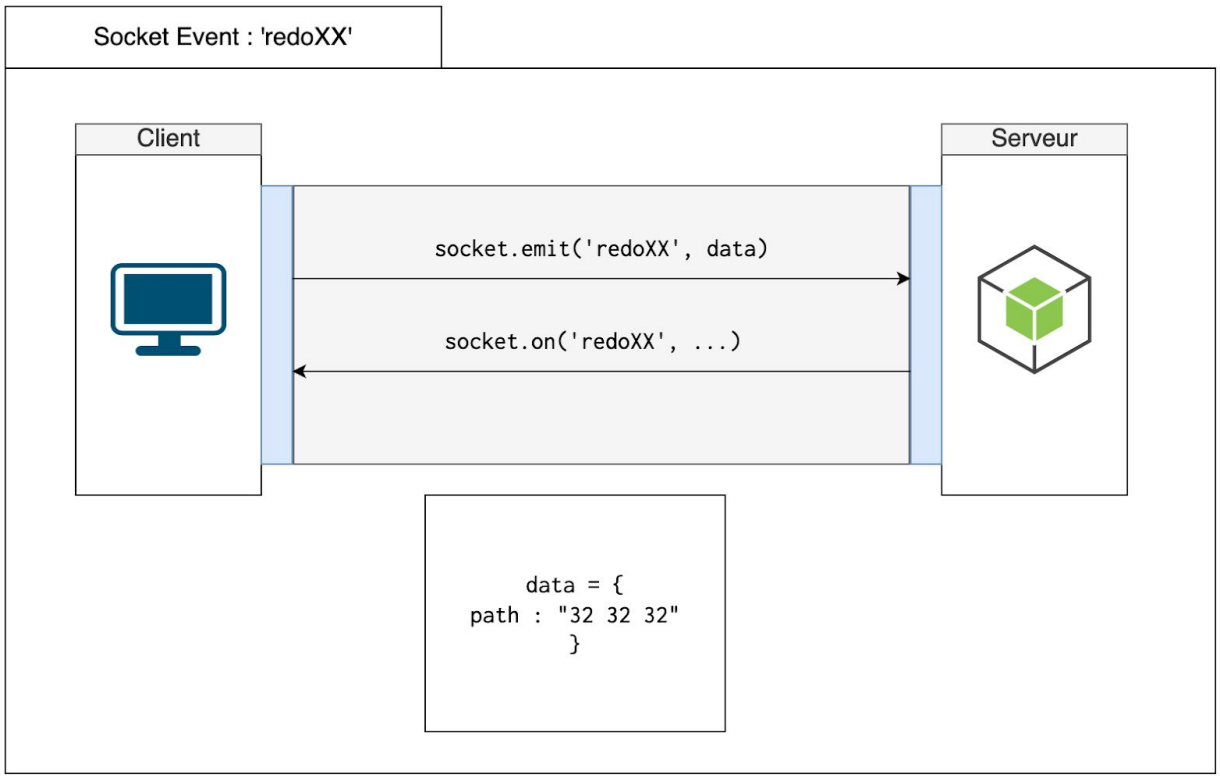


Encore une fois, XX correspond au id du lobby de la partie.

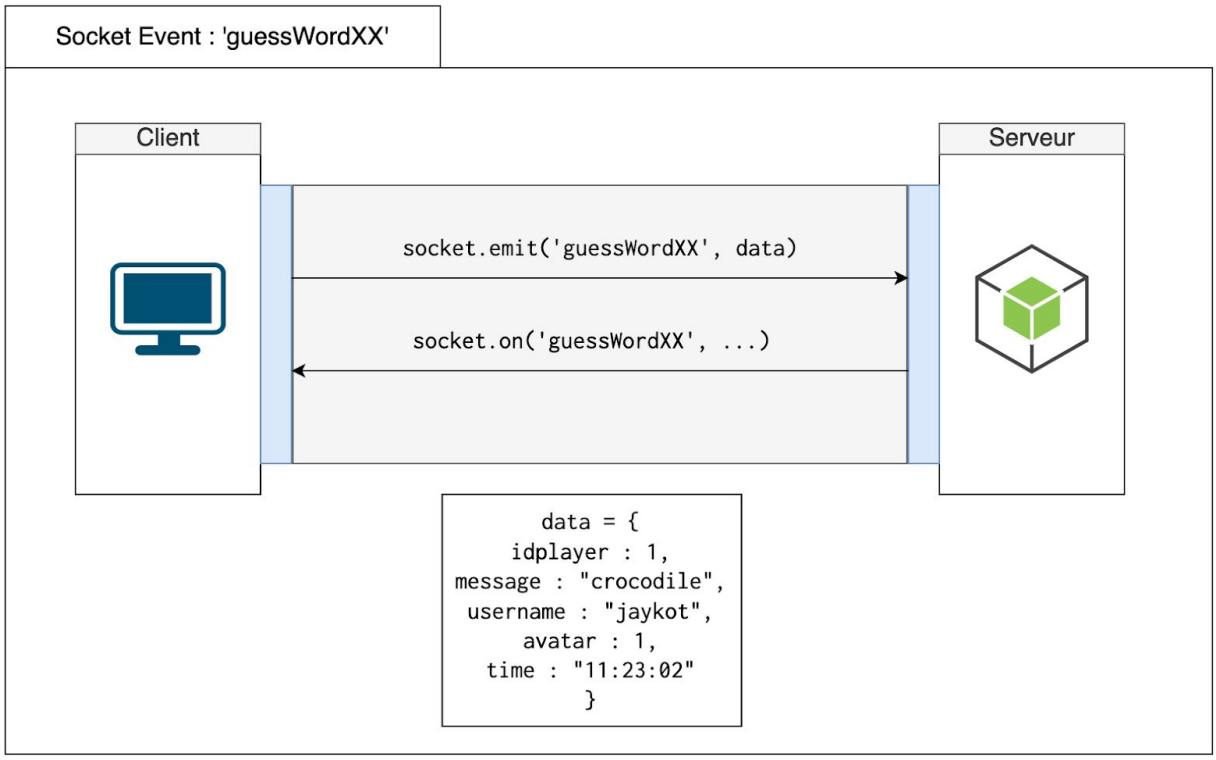
3.18.5 Utilisation du Undo dans le dessin



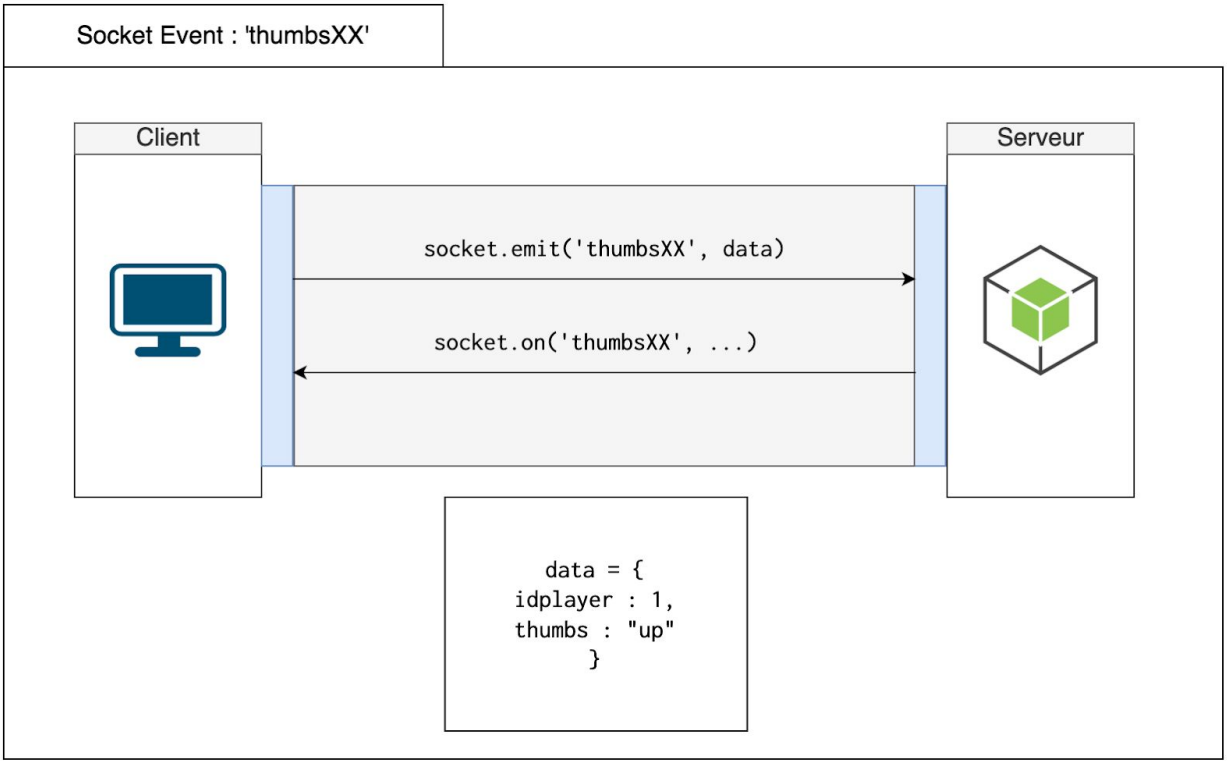
3.18.6 Utilisation du Redo dans le dessin



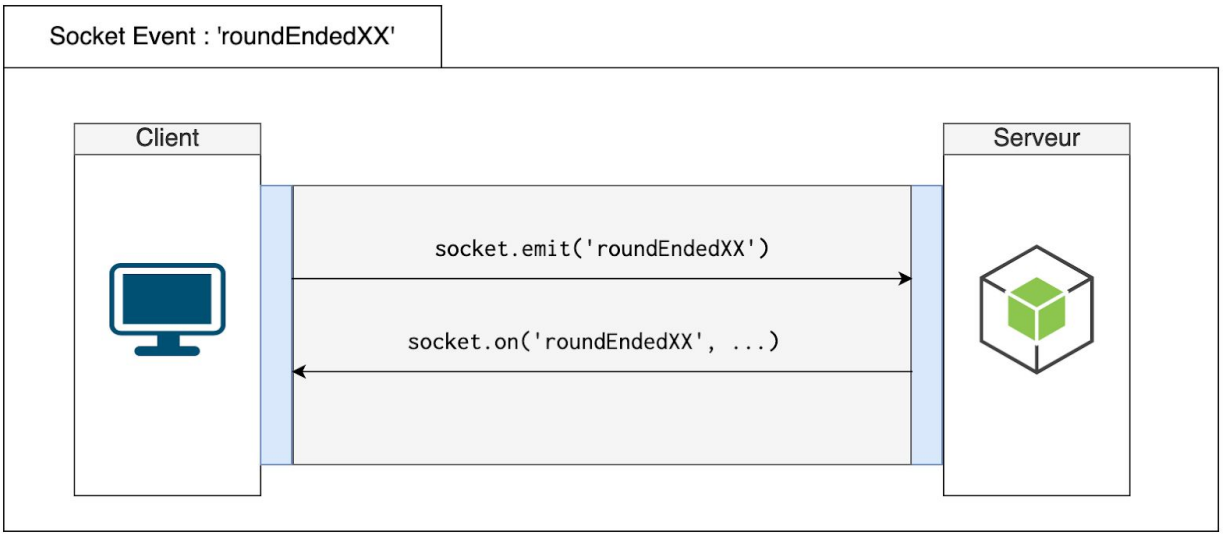
3.18.7 L’utilisateur devine le mot dessiné



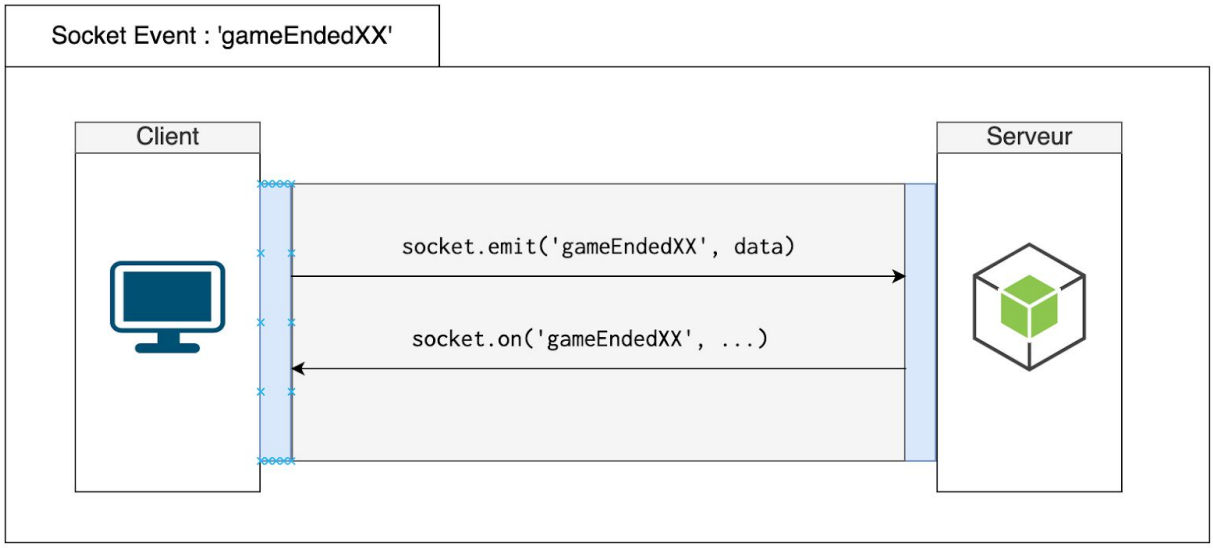
3.18.3 L’utilisateur envoie un pouce au dessinateur



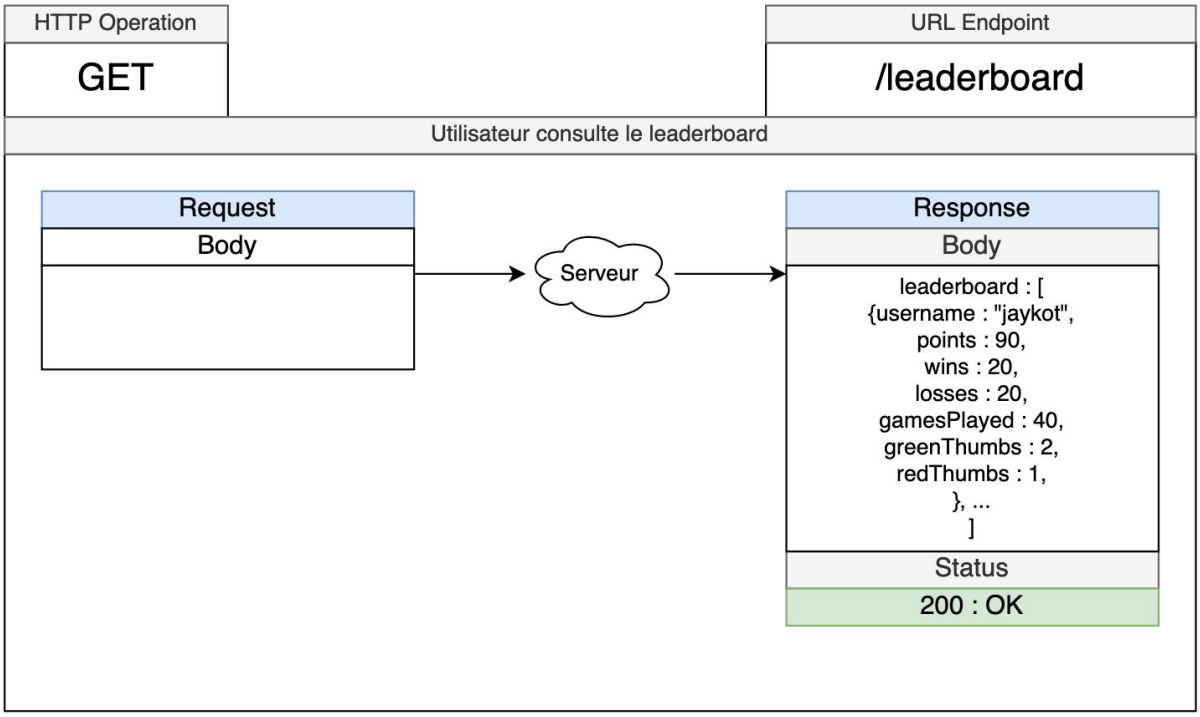
3.18.3 Fin d’une manche



3.18.3 Fin d’une partie



3.19 Consultation du leaderboard



3.20 Consultation de la liste d’amis