Équipe 204

Projet d’évolution d’un logiciel
Protocole de communication

Version 1.2

Historique des révisions

Date	Version	Description	Auteur
aaaa-mm-jj	x.x	<Détails précis du travail effectué>	<Nom>
2021-02-03	1.0	Premier jet du document de protocole de communication	Antoine Morcel Maxime Fecteau
2021-02-09	1.1	Continuation de la première itération du protocole de communication	Antoine Morcel Maxime Fecteau
2021-04-13	1.2	Révision suite à la finition du back-end	Antoine Morcel

Table des matières

Protocole de communication

1. Introduction

Ce document contient l'information relative à la librairie de communication client-serveur utilisée ainsi que la description de l'interface du serveur. De plus, il contient la description des paquets utiles à chaque fonctionnalité. Nous décrivons premièrement ce que les clients peuvent demander au serveur, ensuite nous décrivons la structure des paquets JSON que les clients doivent envoyer au serveur pour pouvoir communiquer.

2. Communication client-serveur

La communication client-serveur est faite avec l'aide de la librairie Socket.io. Nous utiliserons des structures JSON la majorité du temps pour pouvoir envoyer les informations nécessaires. Pour tout ce qui est gestion de compte nous utilisons un api http classique.

2.0 Gestion des sockets

- Ouverture du socket côté client et connexion au serveur lorsque l'utilisateur se connecte à l'application avec son compte utilisateur.
- Fermeture du socket côté client et déconnexion au serveur lorsque l'utilisateur se déconnecte ou ferme l'application.

2.1 Gestion des salons

- Créer un salon de jeu. Le client avise le serveur qu'il veut créer un salon avec le type de la partie et la difficulté de la partie.
- Rejoindre un salon de jeu. Le client avise le serveur que le joueur rejoint un salon.
- Quitter un salon de jeu. Le client avise le serveur que le joueur quitte un salon.
- Statut des salons de jeu publics. Le client demande au serveur une liste des parties publiques qui sont disponibles.
- Inviter un utilisateur à un salon de jeu. Le joueur envoie une invitation à un autre joueur qui est dans sa liste d'amis et qui est en ligne.

2.2 Gestion de messagerie

- Créer un salon de messagerie.
- Rejoindre un salon de messagerie.
- Quitter un salon de messagerie.
- Détruire un salon de messagerie.
- Envoyer un message au salon de messagerie.
- Envoyer un message au salon de jeu courant.
- Envoyer un message privé à un utilisateur
- Obtenir l'historique de message à un utilisateur. Le client demande les derniers messages transmis entre l'utilisateur qui demande et l'utilisateur spécifié. Entre autres, ceci est pour voir l'historique des messages envoyés à ces amis.
- Obtenir l'historique de messagerie des différents salons. Tous sauf celui de partie.

2.3 Gestion de compte (api Http, contrairement au reste qui utilise socket.io)

- Créer un nouveau compte
- Connexion à un compte
- Déconnexion de l'utilisateur
- Obtenir les informations du compte courant (Nom d'usager, nombre de parties jouées, etc)
- Obtenir les informations du tableau de bord.
- Modifier l'information de compte (nom d'usager, courriel, nom)

2.4 Gestion des dessins

- Créer un trait de dessin. La création d'un trait de dessin se fait en trois étapes distinctes qui ont chacune leurs propre appel de socket (socket.on) pour permettre la transmission fluide et en directe du dessin aux autres clients.
    - Premièrement, lorsque le client commence à faire un trait (onMouseDown), celui relaie l'information de la balise svg créer au serveur, qui lui, la renvoie aux autres clients.
    - Deuxièmement, lorsque le client exécute des mouvements avec son trait, il relaie les modifications de cette balise svg au serveur, qui lui, la renvoie aux autres clients.
    - Troisièmement, lorsque le client finit de dessiner son trait, il le relaie au serveur, qui lui, relaie aux autres clients que ce trait est terminé et que la balise svg ne subira plus de modifications.
- Effacer un trait de dessin. Le client envoie les informations relatives aux traits qui sont effacés par le dessinateur au serveur, qui lui renvoie ces informations aux autres clients.
- Défaire. Le client qui est en train de dessiner indique au serveur qu'il a défait une action.
- Refaire. Le client qui est en train de dessiner indique au serveur qu'il a refait une action.

2.5 Gestion modes jeux

- Début de partie
- Fin de partie
- Sélection de mot pour le dessinateur (s'il n'y a pas de joueur virtuel dans l'équipe active)
- Deviner (la logique côté serveur va changer selon le mode de jeu)
- Assignation de rôle

2.6 Joueur Virtuel

Les joueurs virtuels communiquent avec les clients de manière unidirectionnelle. Ils doivent pouvoir:

- Dessiner des traits (voir la section 2.4)
- Envoyer des messages dans les salons de jeu

Pour que les joueurs virtuels fonctionnent, nous devons pouvoir ajouter des paires de mot-image à la base de donnée de mot ainsi nous devons pouvoir :

- Ajouter une paire de mot-image à partir d'un dessin manuel
- Ajouter un paire de mot-image à partir d'une image jpg/png
- Récupérer la séquence de dessin selon le mode de dessin pour pouvoir afficher un aperçu

3. Description des paquets

Ci-dessous sont la structure des paquets (qui sont sujet à changement) pour les différentes fonctionnalités du serveur que les clients peuvent utiliser.

3.1 Gestion des salons

- Pour créer un salon de jeu on a :

```
CREATE_LOBBY = 'createLobby', (
      lobbyName: string,
      gametype: GameType,
      difficulty: Difficulty,
      privacySetting: boolean
)
```

- Pour rejoindre un salon de jeu on a :

```
JOIN_LOBBY = 'joinLobby', (
      lobbyId: string
)
```

- Pour quitter un salon de jeu on a :

```
    LEAVE_LOBBY = 'leaveLobby'
```

- Pour le statut des salons publics on a :

```
GET_ALL_LOBBIES = 'getListLobby', (
      lobbyOpts: LobbyOpts
      callback: (lobbiesCallback: LobbyInfo[]) => void
)
interface LobbyOpts {
    difficulty?: Difficulty;
    gameType?: GameType;
}

interface LobbyInfo {
    lobbyId: string;
    lobbyName: string;
    ownerUsername: string;
    nbPlayerInLobby: number;
    gameType: GameType;
    difficulty: Difficulty;
    maxSize: number;
}
```

- Pour inviter un utilisateur à un salon de jeu on a :

```
SEND_INVITE = 'sendInviteToFriend', (
      friendId: string
)
```

et

```
ACCEPT_INVITE = 'acceptInviteFromFriend', (
      lobbyIdToJoin: string,
      callback: (lobbyJoined: boolean) => void
)
```

3.2 Gestion de messagerie

- Pour la création de salon de messagerie on a :

```
CREATE_CHAT_ROOM = 'createChatRoom', (
      roomName: string,
      callback: (successfullyCreated: boolean) => void
)
```

- Pour rejoindre un salon de messagerie on a :

```
 JOIN_CHAT_ROOM = 'joinChatRoom', (
      roomName: string,
      callback: (joined: boolean) => void
)
```

- Pour quitter un salon de messagerie on a :

```
LEAVE_CHAT_ROOM = 'leaveChatRoom', (
      roomName: string,
      callback: (left: boolean) => void
)
```

- Pour détruire un salon de messagerie on a :

```
DELETE_CHAT_ROOM = 'deleteChatRoom', (
      roomName: string,
      callback: (successfullyDeleted: boolean) => void
)
```

- Pour envoyer un message à un salon de messagerie on a :

```
SEND_MESSAGE_TO_ROOM = 'SendMsgToRoom', (
      roomName: string,
      message: Message
)
interface Message {
    content: string;
}
```

- Pour envoyer un message au salon courant on a :

```
SEND_MESSAGE = 'SendMsg', (
      sentMsg: Message
)
interface Message {
    content: string;
}
```

- Pour envoyer un message à un utilisateur on a :

```
SEND_PRIVATE_MESSAGE = 'SendPrivateMsg', (
      sentMsg: PrivateMessageTo
)
interface PrivateMessageTo {
    receiverAccountId: string;
    content: string;
}
```

- Pour obtenir l'historique de messagerie d'un salon on a :

```
GET_CHAT_ROOM_HISTORY = 'getChatRoomHistory', (
```

```
      roomName: string,
      page: number,
      limit: number,
      callback: (chatHistory: ChatRoomHistory | null) => void
)
interface ChatRoomHistory {
    roomName: string;
    messages: ChatRoomMessage[];
}
interface ChatRoomMessage {
    senderAccountId: string;
    senderUsername: string;
    timestamp: number;
    content: string;
    roomName: string;
}
```

- Pour obtenir l'historique de message à un utilisateur on a :
GET /api/database/friends/history?page=number&limit=number&otherId=id avec comme headers:

```
authorization : jwt_token
```

le client reçoit

```
interface MessageHistory {
    messages: PrivateMessage[];
}
interface PrivateMessage {
    senderAccountId: string;
    receiverAccountId: string;
    timestamp: number;
    content: string;
}
```

3.3 Gestion de compte

- Pour créer un nouveau compte : POST /api/database/auth/register avec comme corps

```
interface Register {
    firstName: string;
    lastName: string;
    username: string;
    email: string;
    password: string;
    passwordConfirm: string;
}
```

Comme réponse du serveur on a :

```
interface LoginResponse {
    accessToken: string;
    refreshToken: string;
}
```

- Pour se connecter à un compte : POST /api/database/auth/login avec comme corps

```
interface login {
    username: string;
    password: string;
}
```

Comme réponse on a :

```
interface LoginResponse {
    accessToken: string;
    refreshToken: string;
}
```

- Pour se déconnecter : DELETE /api/databse/auth/logout avec comme corps

```
interface Logout {
    refreshToken: string;
}
```

- Pour obtenir les informations du compte : GET /api/database/account avec dans les headers :

```
authorization : jwt_token
```

Le serveur retourne :

```
interface AccountInfo {
    _id: string;
    firstName: string;
    lastName: string;
    username: string;
    email: string;
    friends: AccountFriend[];
    createdDate: number;
    avatar: string;
}
interface AccountFriend {
    friendId: string,
    status: FriendStatus,
    received: boolean
}
```

- Pour obtenir les information du tableau de bord on a : GET /api/database/dashboard avec dans les headers

```
authorization : jwt_token
```

Le serveur retourne les informations du tableau de bord :

```
interface DashBoardInfo {
    _id: string;
    firstName: string;
    lastName: string;
    username: string;
    email: string;
    logins: Logins[];
    gameHistory: GameHistoryDashBoard;
    createdDate: number;
    avatar: string;
}
export interface Logins {
    start: number;
    end?: number;
}
export interface GameHistoryDashBoard {
    games: Game[];
    nbGamePlayed: number;
    winPercentage: number;
    averageGameTime: number;
    totalTimePlayed: number;
    bestScoreSolo: number;
    bestScoreCoop: number;
}
export interface Game {
    gameResult: GameResult;
    startDate: number;
    endDate: number;
    gameType: GameType;
    team: Team[];
}
export interface Team {
    score: number;
    playerNames: string[];
}
```

- Pour supprimer le compte : DELETE /api/database/account avec dans les headers :

```
authorization : jwt_token
```

Le serveur retourne le compte qui vient d'être supprimé

- Pour modifier le compte POST /api/database/account avec dans les headers :

```
authorization : jwt_token
```

et comme corps

```
interface UpdateAccount {
    firstName?: string,
    lastName?: string,
    username?: string,
    email?: string,
}
```

Le serveur retourne le compte modifié comme suit :

```
interface AccountInfo {
    _id: string;
    firstName: string;
    lastName: string;
    username: string;
    email: string;
    friends: AccountFriend[];
    createdDate: number;
    avatar: string;
}
interface AccountFriend {
    friendId: string,
    status: FriendStatus,
    received: boolean
}
```

3.4 Gestion des dessins

- Pour pouvoir commencer un trait on a :

```
START_PATH = 'startPath', (
      startPoint: Coord,
      brushInfo: BrushInfo
)
interface Coord {
    x: number;
    y: number;
}
interface BrushInfo {
    color: string
    strokeWidth: number;
}
```

Le serveur renvoi le début du trait aux autres clients avec :

```
START_PATH_BC = 'startPathBroadcast', (
    id: number;
```

```
    zIndex: number;
    startPoint: Coord;
    brushInfo : BrushInfo;
)
```

- Pour mettre à jour un trait on a :

```
UPDATE_PATH = 'updatePath', (
      updateCoord: Coord
)
```

Le serveur renvoi la mise à jour du trait aux autres clients avec :

```
UPDATE_PATH_BC = 'updatePathBroadcast', (
      updateCoord: Coord
)
```

- Pour terminer un trait on a :

```
END_PATH = 'endPath', (
      endPoint: Coord
)
```

Le serveur renvoi la terminaison du trait aux clients avec :

```
END_PATH_BC = 'endPathBroadcast', (
      endPoint: Coord
)
```

- Pour effacer un trait on a :

```
ERASE_ID = 'erase', (
      id: number
)
```

Le serveur renvoi l'effacement du trait aux clients avec :

```
ERASE_ID_BC = 'eraseBroadcast',(
      id: number
)
```

- Pour ajouter un trait on a :

```
ADD_PATH = 'addPath',(
      id: number
)
```

Le serveur renvoi l'ajout du trait aux clients avec :

```
ADD_PATH_BC = 'addPathBroadcast',(
      id: number;
      zIndex: number;
      startPoint: Coord;
      brushInfo : BrushInfo;
)
```

- La gestion des undo/redo des clients est faite avec les commandes d'effaçage et d'ajout de trait ci-haut.

3.5 Gestion modes jeux

- Pour le début de la partie l'hôte de la partie envoi au serveur :

```
START_GAME_SERVER = 'StartGameFromLobbyToServer'
```

Puis le serveur renvoi aux clients

```
START_GAME_CLIENT = 'StartGameFromServerToClient', (
      players : Player[]
)
interface Player{
  accountId: string | null;
  avatarId: string | null;
  finishedLoading: boolean | null;
  username: string;
  playerRole: PlayerRole;
  teamNumber: number;
  isBot: boolean;
  isOwner: boolean;
}
```

Finalement quand le client a fini de bouger vers la zone de jeu il envoi au serveur :

```
LOADING_OVER = 'loadingOver'
```

Quand chaque joueur a terminé de charger, le serveur commence la partie!

- Une fois la partie terminée le serveur envoi aux clients :

```
END_GAME = 'endGame', (
      reason: ReasonEndGame
)
enum ReasonEndGame {
  PLAYER_DISCONNECT = 'playerDisconnected',
  WINNING_SCORE_REACHED = 'winningScoreReached',
  TIME_RUN_OUT = 'timeRunOut',
  NO_WORDS_FOUND = 'noWordsFound'
}
```

- Pour le mot à dessiner le serveur envoi au client dessinateur :

```
UPDATE_WORD_TO_DRAW = 'updateWordToDraw', (
      word: string
)
```

- Pour deviner (la logique côté serveur va changer selon le mode de jeu) on a :

```
PLAYER_GUESS = 'guess', (
      word: string
```

```
)
```

Qui retourne aux clients :

```
GUESS_BROADCAST = 'GuessBroadcast', (
      guessReturn: GuessMessage
)
interface GuessMessage {
  guessStatus: GuessResponse
  senderUsername: string;
  timestamp: number;
  content: string;
}
enum GuessResponse {
  CORRECT = 'correct',
  CLOSE = 'close',
  WRONG = 'wrong'
}
```

- Pour l'assignation de rôle on a le serveur qui envoi de l'information aux clients :

```
UPDATE_ROLES = 'updateRoles', (
      players : Player[]
)
interface Player{
  accountId: string | null;
  avatarId: string | null;
  finishedLoading: boolean | null;
  username: string;
  playerRole: PlayerRole;
  teamNumber: number;
  isBot: boolean;
  isOwner: boolean;
}
enum PlayerRole {
  DRAWER = 'active',
  GUESSER = 'guesser',
  PASSIVE = 'passive'
}
```

3.6 Joueur Virtuel

- Pour ajouter une paire de mot-image à partir d'un dessin manuel on a :
  POST /api/pictureword/upload/drawing avec comme corps :

```
```

```
interface PictureWordDrawing {
    drawnPaths: PictureWordPath[],
    word: string,
    hints: string[],
    difficulty: Difficulty,
    drawMode: DrawMode
}
interface PictureWordPath {
    brushInfo: BrushInfo
    path: { x: number, y: number }[]
    id: string
}
enum Difficulty {
  EASY = 'easy',
  INTERMEDIATE = 'intermediate',
  HARD = 'hard'
}
enum DrawMode {
    CONVENTIONAL = 'conventional',
    RANDOM = 'random',
    PAN_L_TO_R = 'panlr',
    PAN_R_TO_L = 'panrl',
    PAN_T_TO_B = 'pantb',
    PAN_B_TO_T = 'panbt',
    CENTER_FIRST = 'center'
}
```

Qui retourne l'id de la paire mot-image créée.

- Pour ajouter un paire de mot-image à partir d'une image jpg/png on a :
POST /api/pictureword/upload/picture avec comme corps :

```
export interface PictureWordPicture {
    picture: string, // encodé en base64
    color: string,
    threshold?: number,
    word: string,
    hints: string[],
    difficulty: Difficulty,
    drawMode: DrawMode
}
```

Qui retourne l'id de la paire mot-image créée.

- Pour récupérer la séquence de dessin selon le mode de dessin pour pouvoir afficher un aperçu on a:

GET /api/pictureword/sequence/:id qui retourne :

```
interface DrawingSequence {
    height: number;
    width: number;
    stack: Segment[];
}
interface Segment {
    zIndex: number;
    brushInfo: BrushInfo;
    path: Coord[];
}
```

Les joueurs virtuels utilisent la structure de donné ci-haut pour dessiner. Ils dessinent en utilisant les mêmes fonctions de dessin du serveur, donc

```
START_PATH_BC = 'startPathBroadcast', (
    id: number;
    zIndex: number;
    startPoint: Coord;
    brushInfo : BrushInfo;
)
UPDATE_PATH_BC = 'updatePathBroadcast', (
      updateCoord: Coord
)
END_PATH_BC = 'endPathBroadcast', (
      endPoint: Coord
)
```