109

Protocole de communication

Version 2.0

Historique des révisions

Date	Version	Description	Auteur
2023-03-18	1.0	Description du protocole de communication HTTP et WS	Asimina Koutsos
2023-03-21	1.0	Ajout de détails pour WS dans la partie 2	Asimina Koutsos
2023-04-19	2.0	Correction erreurs sprint 2	Asimina Koutsos
2023-04-20	2.0	Ajout description nouvelles fonctionnalités WS sprint 3 + ajout de détails partie WS	Selim Bellagha

Table des matières

Protocole de communication

1. Introduction

La communication entre le client et le serveur se sépare en deux sous-groupes: la communication par protocole HTTP et la communication par les sockets avec le protocole WebSocket (WS). Les deux protocoles sont utilisés pour les différentes fonctionnalités requises du site web. La communication HTTP est principalement un protocole de communication unidirectionnelle et requiert donc une requête du client pour communiquer avec le serveur. Le protocole WebSocket est bidirectionnel, ce qui veut dire que des données peuvent être envoyées dans les deux sens simultanément et le serveur n'a donc pas besoin d'une requête du client pour communiquer avec lui. Ces deux protocoles ont donc été utiles pour gérer différents aspects du site web. Ci-bas seront détaillés les choix d'utilisation de chaque moyen de communication ainsi que la description des différents types de paquets utilisés au sein de ces protocoles.

2. Communication client-serveur

Le protocole HTTP nécessite une requête du client pour permettre au serveur de retourner des données. Ce protocole est utilisé pour stocker les 3 éléments suivants du site web:

1. Les jeux
2. Les 3 constantes du site
3. L'historique des parties jouées

Ces éléments sont stockés dans des fichiers JSON (games.json, constantes.json, history.json) sur le serveur et on y accède aux éléments de ces fichiers à travers des méthodes HTTP qui seront expliquées et définies dans la section 3 du protocole.

Les jeux sont stockés sous la structure suivante:

```
export interface GameData {
    id: string;
    name: string;
    originalImage: string;
    modifiedImage: string;
    nbDifferences: number;
    differences: Vec2[][];
    isDifficult: boolean;
}
```

Les constantes sont stockés sous la structure suivante:

```
export interface Constants {
    initTime: number;
    penaltyTime: number;
    timeBonus: number;
}
```

Les parties de l'historique sont stockés sous la structure suivante:

```
export interface GameHistory {
```

```
    startDate: string;
    gameLength: string;
    gameMode: string;
    namePlayer1: string;
    namePlayer2: string;
    winnerName: string;
    nameAbandon: string;
}
```

Le protocole HTTP est utilisé pour les fonctionnalités spécifiques suivantes:

- Affichage des jeux sur les pages de configuration et de sélection. Le client fait une requête pour obtenir tous les jeux et ensuite les afficher.
- Ajout et suppression de jeux
- Gérer les clics des joueurs sur les canevas de jeu. Quand un joueur clique sur un canvas, la position en X et en Y du clic, ainsi que l'id du jeu sont envoyés au serveur, le serveur vérifie si ce clic identifie une différence du jeu et renvoie la réponse au client.
- Gérer les 3 constantes de jeu; lorsqu'on veut modifier les constantes, les nouvelles valeurs sont envoyées au serveur à travers une requête et le fichier contenant les constantes stockées est mis à jour.
- Affichage de l'historique dans la page de configuration; le client fait une requête pour obtenir tous les parties de l'historique et ensuite les afficher.
- Ajout des informations (date/heure de début, durée de jeu, nom des joueurs, gagnant, abandon) d'une nouvelle partie terminée à l'historique et suppression de l'historique.

De plus, le protocole HTTP est utilisé pour communiquer au serveur qui lui va communiquer avec la DB pour accéder au meilleurs temps. Ça permettra d'afficher les meilleurs temps de chaque jeu dans les jeux de la page de configuration et sélection, d'envoyer un nouveau topscore au serveur pour vérifier s'il bat les scores stockés dans la BD et de demander la réinitialisation des scores de la BD. Les meilleurs temps sont stockés dans la base de données sous la structure suivante:

```
export interface TopScore {
    position: string;
    gameId: string;
    gameType: string;
    time: string;
    playerName: string;
}
```

Nous avons choisi le protocole HTTP pour les éléments et fonctionnalités décrits plus haut, car ces fonctionnalités ne nécessitent pas forcément une communication bidirectionnelle; le client initie la communication avec le serveur en envoyant des requêtes. Donc, dans les cas où il faut juste chercher des données du serveur comme pour l'affichage des jeux et de l'historique, nous avons trouvé que c'était plus simple d'utiliser un protocole de communication qui utilise principalement une communication unidirectionnelle comme HTTP. Dans les fonctionnalités utilisant HTTP après avoir obtenu les données voulues ou mis à jour des données sur le serveur, le client et le serveur n'ont plus besoin de s'envoyer des données, alors un protocole de communication unidirectionnel nous est tout à fait suffisant.

Le protocole WebSocket est utilisé pour pour gérer le mode 1v1 incluant le chat. Une grande utilité du protocole WS est le fait que ce protocole est bidirectionnel, ce qui permet donc au serveur de communiquer avec le client sans une requête du client. Ceci permet donc la communication en temps réel. Des fonctionnalités spécifiques que cela permet sont:

- la mise en place de salles pour gérer la communication entre clients dans une partie et dans le chat
- la mise en place d'une salle d'attente pour les joueurs qui veulent jouer en 1v1; puisque le protocole WS permet de garder une trace des clients et de leur envoyer des données, on pourra gérer la queue des joueurs qui attendent d'être acceptés dans un jeu et le serveur pourra leur émettre des événements (comme dans le cas où un 2ème joueur est accepté dans une partie créée.
- la gestion des compteurs de différences des 2 joueurs en mode 1v1. Le serveur émet des événements au client pour gérer l'affichage des 2 compteurs. Aussi, lorsqu'un des joueurs atteint le nombre de différences trouvées nécessaire pour gagner la partie, le serveur peut émettre un événement au client pour lui indiquer qui a gagné.

Il sera aussi utilisé dans les fonctionnalités du sprint 3 comme pour émettre un événement qui indique que le temps s'est écoulé dans le mode Temps Limité.

3. Description des paquets

3.1 Description des requêtes HTTP utilisées:

Chaque section contient un tableau résumé des requêtes HTTP utilisées et est suivi par une description détaillée de chaque requête.

3.1.1 Requêtes HTTP vers les jeux stockés dans le fichier games.json

Tableau Résumé des requêtes vers les jeux stockés

MÉTHODE	DESCRIPTION	ROUTE	CORPS REQUÊTE	CODE RETOUR	CORPS RÉPONSE
GET	Retourne la liste de tous les jeux	/api/games		200 OK	liste de jeux: GameData[]
GET	Retourne un jeu en fonction de son id	/api/games/:id	id = req.params.id	200 OK	jeu: GameData
GET	Vérifie si un clic sur le canvas identifie une différence et retourne le résultat	/api/difference/:id	id = req.params.id position en X du clic = req.query.ClickX	200 OK	résultat de la vérification de type {result: boolean, index: number}

			position en Y du clic = req.query.ClickY		
POST	Ajoute un jeu	api/games/send	Objet de type GameData = req.body	201 CREATED	message indiquant l'ajout
DELETE	Supprime un jeu en fonction de son id	api/games/:id	id = req.params.id	200 OK si jeu existe 404 NOT FOUND sinon	message indiquant si supprimé ou non

La requête **GET** sur la route /api/games va retourner la liste de tous les jeux stockés. Dans la réponse de retour sont envoyés le code de retour 200 (OK) et la liste de jeux. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

On obtient les jeux à partir de la méthode getAllGames() qui va lire le fichier games.json contenant les jeux et retourne une promesse avec l'array de jeux (Promise<GameData[]>).

```
this.router.get('/', async (req: Request, res: Response) => {
    try {
        const games: GameData[] = await this.gameService.getAllGames();
        res.status(StatusCodes.OK).send(games);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **GET** sur la route /api/games/:id va retourner un jeu en fonction de son id. Si le jeu est trouvé, la réponse de retour contient le code de retour 200 (OK) et le jeu. Sinon, la réponse de retour contient le code 404 (NOT FOUND) et un message pour indiquer que le jeu voulu n'existe pas. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est obtenu à partir de la méthode getGameById(id: string); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et cette liste sera trié avec "find" pour retourner le jeu avec l'id voulu.

```
this.router.get('/:id', async (req: Request, res: Response) => {
    try {
        const game: GameData = await this.gameService.getGamebyId(req.params.id);
        if (game === undefined) {
            res.status(StatusCodes.NOT_FOUND).json('The requested game does not exist');
        } else {
            res.status(StatusCodes.OK).json(game);
        }
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
```

```
    }
});
```

La requête **GET** sur la route /api/difference/:id est utilisée pour vérifier si un pixel appartient à une différence du jeu lorsque le joueur clique sur le canvas. Dans la requête, req.query.ClickX contient la position en X du clic effectué, req.query.ClickY contient la position en Y du clic effectué et req.params.id contient l'id du jeu auquel le joueur joue. La réponse de retour contient le code de retour 200 (OK) et la réponse sous la forme { result: boolean, index: number}. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

La réponse {result: boolean, index: number} est obtenu à partir de la méthode verificationInPicture(positionX: number, positionY: number, id : string); cette méthode accède à la liste de différences du jeu (champ "differences" dans l'interface GameData) à partir de la méthode getGameById(id). Si les coordonnées du pixel se trouvent dans la liste de différences du jeu, alors la réponse de retour sera { result: true, index: indexDePosition} où indexDePosition sera l'index des coordonnées dans la liste de différences. Si les coordonnées du pixel ne se trouvent pas dans la liste de différences du jeu, alors la réponse de retour sera { result: false, index: -1}.

```
this.router.get('/:id', async (req: Request, res: Response) => {
    try {
        const positionX = Number(req.query.ClickX);
        const positionY = Number(req.query.ClickY);
        const id = req.params.id;
        const response = await this.gameManager.verificationInPicture(positionX,
positionY, id);
        res.status(StatusCodes.OK).send(response);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **POST** sur la route /api/games/send va ajouter un nouveau jeu à la liste de jeux. Le corps de la requête contient le jeu à ajouter. Quand le jeu est créé, la réponse de retour contient le code de retour 201 (CREATED) et un message indiquant que le jeu a été ajouté. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est ajouté à l'aide de la méthode addGame(newGame: Gamedata); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et on ajoute le nouveau jeu à cette liste en effectuant un "push" sur la liste. On écrit ensuite dans le fichier json pour modifier le contenu avec la liste modifiée de jeux.

```
this.router.post('/send', async (req: Request, res: Response) => {
    try {
        const game: GameData = req.body;
        await this.gameService.addGame(game);
        res.status(StatusCodes.CREATED).json('The game was added');
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **DELETE** sur la route /api/games/:id va supprimer un jeu de la liste de jeux en fonction de son id. Le params.id dans la requête contient l'id du jeu à supprimer. Si le jeu existe et est supprimé, la réponse de retour contient le code de retour 200 (OK) et un message indiquant que le jeu a été supprimé. Sinon, la réponse de retour contient le code 404 (NOT FOUND) et un message pour indiquer que le jeu voulu n'existe pas et ne peut donc pas être supprimé. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est supprimé à l'aide de la méthode deleteGame(id: string); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et on trie cette liste pour trouver le jeu à supprimer. Si le jeu existe, le jeu est enlevé de la liste de jeux et on écrit ensuite dans le fichier json pour modifier le contenu avec la liste modifiée de jeux. Dans le cas où le jeu existe et on peut la supprimer, la fonction deleteGame(id) retourne true, sinon elle retourne false;

```
this.router.delete('/:id', async (req: Request, res: Response) => {
    try {
        const gameToDelete = await this.gameService.deleteGame(req.params.id);
        if (gameToDelete === false) {
            res.status(StatusCodes.NOT_FOUND).json('The requested game cannot be
deleted as it does not exist');
        } else {
            res.status(StatusCodes.OK).json('The game with the requested id has been
deleted');
        }
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

3.1.2 Requêtes HTTP vers l'historique des parties dans le fichier history.json

Tableau Résumé des requêtes vers l'historique

MÉTHODE	DESCRIPTION	ROUTE	CORPS REQUÊTE	CODE RETOUR	CORPS RÉPONSE
GET	Retourne la liste des parties de l'historique	/api/history		200 OK	liste des parties: GameHistory[]
POST	Ajoute une partie à l'historique	/api/history	Objet de type GameHistory= req.body	201 CREATED	message indiquant l'ajout
DELETE	Supprime toutes les parties dans l'historique	/api/history		200 OK	message indiquant la suppresion

La requête **GET** sur la route /api/history va retourner la liste de toutes les parties stockées. Dans la réponse de retour sont envoyés le code de retour 200 (OK) et la liste des parties. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

On obtient les parties à partir de la méthode getAllHistory() qui va lire le fichier history.json contenant les parties et retourne une promesse avec l'array de parties (Promise<History[]>).

```
this.router.get('/', async (req: Request, res: Response) => {
    try {
        const history: GameHistory[] = await this.historyService.getAllHistory();
        res.status(StatusCodes.OK).send(history);
   } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
   }
});
```

La requête **POST** sur la route /api/history va ajouter une nouvelle partie à la liste après la fin d'un jeu. Le corps de la requête contient la partie à ajouter. Quand la partie est ajoutée, la réponse de retour contient le code de retour 201 (CREATED) et un message indiquant que la partie a été ajoutée. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est ajouté à l'aide de la méthode addGameHistory(newGameHistory: GameHistory); cette méthode obtient la liste des parties à partir de la méthode getAllHistory() et on ajoute la nouvelle partie à cette liste en effectuant un "push" sur la liste. On écrit ensuite dans le fichier json pour modifier le contenu avec la liste modifiée de parties.

```
this.router.post('/', async (req: Request, res: Response) => {
    try {
        const newGameHistory: GameHistory = req.body;
        await this.historyService.addGameHistory(newGameHistory);
        res.status(StatusCodes.CREATED).json('The game history was added');
     } catch (error) {
         res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
     }
});
```

La requête **DELETE** sur la route /api/history va supprimer la liste complète des parties en historique. Après la suppression, la réponse de retour contient le code de retour 200 (OK) et un message indiquant que l'historique a été supprimé. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est supprimé à l'aide de la méthode deleteAllHistory(); cette méthode remplace le contenu du fichier json avec un array vide supprimant ainsi toutes les parties dans l'array initial.

```
this.router.delete('/', async (req: Request, res: Response) => {
    try {
        await this.historyService.deleteAllHistory();
```

```
        res.status(StatusCodes.OK).json('All history was deleted');
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

3.1.3 Requêtes HTTP vers les constantes de jeu dans le fichier constantes.json

Tableau Résumé des requêtes vers les constantes

MÉTHODE	DESCRIPTION	ROUTE	CORPS REQUÊTE	CODE RETOUR	CORPS RÉPONSE
GET	Retourne les 3 constantes de jeu	/api/games/constants		200 OK	objet de type: Constants
POST	Ajouter les nouvelles constantes	/api/games/constants	Objet de type Constants = req.body	200 OK	

La requête **GET** sur la route /api/games/constants va retourner les 3 constantes de jeu stockées:

- temps initial du compte à rebours
- temps de pénalité pour l'utilisation d'un indice
- temps gagné avec la découverte d'une différence

Dans la réponse de retour sont envoyés le code de retour 200 (OK) et les constantes de jeu. On obtient les constantes de jeu à partir de la méthode getGamesConstants() qui va lire le fichier constantes.json contenant les constantes et retourner une promesse (Promise<Constants>).

```
this.router.get('/constants', async (req: Request, res: Response) => {
   res.status(StatusCodes.OK).json(await this.constantsManager.getGameConstants());
});
```

La requête **POST** sur la route /api/games/constants va remplacer les constantes actuelles avec celles dans le corps de la requête. Le corps de la requête contient les nouvelles constantes. Quand les constantes sont modifiées, la réponse de retour contient le code de retour 200 (OK).

Les constantes sont modifiées à l'aide de la méthode addGameConstants(newConstants: Constants); cette méthode obtient l'objet de type Constants du fichier constants.json et il modifie les valeurs des 3 constantes dans l'objet. On écrit ensuite dans le fichier json pour mettre à jour le contenu avec l'objet modifié.

```
this.router.post('/constants', (req: Request, res: Response) => {
    this.constantsManager.addGameConstants(req.body);
    res.sendStatus(StatusCodes.OK);
});
```

3.1.4 Requêtes HTTP vers les meilleurs temps de la BD

Tableau Résumé des requêtes vers les meilleurs temps

MÉTHODE	DESCRIPTION	ROUTE	CORPS REQUÊTE	CODE RETOUR	CORPS RÉPONSE
GET	Retourne la liste de tous les scores	/api/scores		200 OK	liste de scores: TopScore[]
GET	Retourne la liste des scores en fonction d'un id de jeu et du type (solo ou 1v1)	/api/scores/:gameId/:gameType	gameId, gameType = req.params	200 OK	liste de scores: TopScore[]
POST	Vérifie si le nouveau score bat un des top 3 dans la base de donnée et l'ajoute si oui	/api/scores	Objet de type TopScore = req.body	200 OK	résultat de la vérification de l'ajout de type : { isAdded: boolean, positionIndex: string}
POST	Ajoute des valeurs fictives de meilleurs temps dans la BD pour un nouveau jeu	/api/scores/:gameId	gameId = req.params	201 CREATED	message indiquant l'ajout
DELETE	Réinitialise les meilleurs temps d'un jeu en fonction de son id	/api/scores/:gameId	gameId = req.params	200 OK	message indiquant la réinitialisation

La requête **GET** sur la route /api/scores va retourner la liste de tous les meilleurs temps. Dans la réponse de retour sont envoyés le code de retour 200 (OK) et la liste des temps. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

On obtient les parties à partir de la méthode getAllTopScores() qui va récuperer toutes les meilleurs temps dans la base données et retourne une promesse avec l'array de temps (Promise<TopScore[]>).

```
this.router.get('/', async (req: Request, res: Response) => {
```

```
    try {
        const scores: TopScore[] = await this.scoreService.getAllTopScores();
        res.status(StatusCodes.OK).send(scores);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **GET** sur la route /api/scores/:gameId/:gameType va retourner la liste de temps en fonction de l'id du jeu désiré et du type de jeu soit 'solo' ou '1v1'. La réponse de retour contient le code de retour 200 (OK) et la liste de temps triés. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Les temps sont obtenus à partir de la méthode sortTopScores(id: string, type: string); cette méthode trie la base de données pour trouver les éléments respectant l'id et le type fourni dans la requête. Ces temps sont ensuites triés en ordre croissant de l'attribut 'time' (soit du meilleurs temps au plus pire), puis cette liste est retournée dans une promesse (Promise<TopScore[]>).

```
this.router.get('/:gameId/:gameType', async (req: Request, res: Response) => {
    try {
        const { gameId, gameType } = req.params;
        const response = await this.scoreService.sortTopScores(gameId, gameType);
        res.status(StatusCodes.OK).send(response);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **POST** sur la route /api/scores vérifie si le nouveau temps obtenu par un joueur à la fin d'une partie gagné bat un des top 3 dans la base de données et si c'est le cas, on l'ajoute dans le top 3. Le corps de la requête contient le nouveau topscore obtenu. La réponse de retour contient le code de retour 200 (OK) et un résultat de la structure suivante: `{ isAdded: boolean, positionIndex: string}`. IsAdded indique si le nouveau score a été ajouté à la BD et positionIndex indique la position du nouveau score (soit 1, 2, 3 si le score bat un des top 3 ou -1 si le score n'as pas battu le top 3). Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

La méthode *addScore(newScore: TopScore)* retourne la réponse. Cette méthode appelle la méthode *validateScore(newScore: TopScore)* qui elle appelle *sortTopScores(id: string, type: string)* afin de retourner le top 3 du jeu voulu dans le mode voulu (solo, 1v1). On compare ensuite les temps du top 3 avec le temps du nouveau score pour voir si ce dernier bat un des top 3. Si oui, *validateScore()* retourne la position du score qui a été battu, sinon la méthode retourne '-1'.

AddScore va ensuite traiter le résultat de *validateScore*.

- Dans le cas où le résultat est '-1', { isAdded: false, positionIndex: '-1' } est retourné.

- Dans le cas où le résultat est ‘1’, ‘2’ ou ‘3’, on modifie le top 3 dans la BD pour ajouter le nouveau score dans la position indiquée et { isAdded: true, positionIndex: positionRemplacé} est retourné.

```
this.router.post('/', async (req: Request, res: Response) => {
    try {
        const score: TopScore = req.body;
        const response = await this.scoreService.addScore(score);
        res.status(StatusCodes.OK).send(response);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **POST** sur la route /api/scores/:gameId va ajouter des scores fictifs à un nouveau jeu. Le corps de la requête contient l’id du jeu. Quand les scores sont ajoutés, la réponse de retour contient le code de retour 201 (CREATED) et un message indiquant que les scores ont été ajoutés. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.
Les scores sont ajoutés à l’aide de la méthode addDefaultScores(gameId: string); cette méthode ajoute les valeurs fictives au nouveau jeu. Les valeurs fictives se trouvent dans le fichier default_scores.json, ce fichier est lu et converti en un array d’éléments de type TopScore. Cet array est ensuite ajouté à la BD pour le jeu indiqué.

```
this.router.post('/:gameId', async (req: Request, res: Response) => {
    try {
        const { gameId } = req.params;
        await this.scoreService.addDefaultScores(gameId);
        res.status(StatusCodes.CREATED).json('The default scores were added to the new game');
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **DELETE** sur la route /api/scores/:gameId va réinitialiser les meilleurs temps du jeu indiqué dans la requête.. Après la réinitialisation, la réponse de retour contient le code de retour 200 (OK) et un message indiquant que les meilleurs temps ont été réinitialisés. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Ça se fait à l’aide de la méthode resetOneGame(); cette méthode supprime tous les meilleurs dans la BD ayant comme gameId celui fourni dans la requête et ajoute les valeurs fictives de meilleurs temps dans la BD pour ce jeu.

```
this.router.delete('/:gameId', async (req: Request, res: Response) => {
    try {
        const { gameId } = req.params;
```

```
        await this.scoreService.resetOneGame(gameId);
        res.status(StatusCodes.OK).json('The scores for this game have been reinitialized');
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

3.2 Description des évènements du protocole WebSocket:

Plusieurs événements sont transmis entre le client et le serveur. Ci-bas se trouve la description de la gestion des différents événements par le serveur:

- **'startTimer':** le serveur capte cet événement qui est émis par le client lorsqu'une partie de jeu est commencée. Le serveur lance alors le timer avec le temps de jeu en paramètres de la partie.
    - Pourquoi socket ? Les objets timer sont contenus dans le fichier qui gère les sockets. Ainsi on évite les dépendances puisque se sont les sockets qui gèrent les parties de jeu..

- **'startStopWatch' :** le serveur capte cet événement qui est émis par le client lorsqu'une partie de jeu est commencée. Le serveur lance alors le chronomètre de la partie.
    - Pourquoi socket ? Les objets timer sont contenus dans le fichier qui gère les sockets. Ainsi on évite les dépendances puisque se sont les sockets qui gèrent les parties de jeu.

- **'AddToTimer' :** le serveur capte cet événement émis par un des clients qui vient de trouver une différence. Le serveur reçoit en paramètre le temps à ajouter au timer et l'id de la room auquel il doit l'ajouter si la partie n'est pas en solo.
    - Pourquoi socket ? On doit utiliser les sockets car les timers en solo sont dans une map dont la clé est l'id de la socket en question. On doit alors, dans le cas d'une partie solo, avoir accès à l'id de la socket.

- **'getRealTime':** le serveur émet cet événement au client pour lui indiquer le temps actuel du timer qui est en cours. Il retourne au client le temps du jeu en question en secondes.
    - Pourquoi socket ? Les objets timer sont contenus dans le fichier qui gère les sockets. Ainsi on évite les dépendances puisque se sont les sockets qui gèrent les parties de jeu.

- **'createLobby':** lorsqu'un client joint un jeu pour jouer dans le mode 1v1 en premier et est mis dans la salle d'attente, il émet cet événement au serveur avec comme contenu l'id du jeu qu'il attend et son nom de joueur. Le serveur va capter l'événement et crée une salle; le socket qui a émis l'événement est alors ajouté à la salle comme host. Ça permet donc par la suite de regrouper les 2 sockets qui jouent dans le mode 1v1 ensemble dans une salle.
    - Pourquoi socket ? On utilise ici les sockets pour pouvoir, dès la création du lobby, ajouter la socket à une room avec l'id du lobby.

- **'joinQueue':** lorsqu'un deuxième client joint un jeu pour jouer dans le mode 1v1, il émet cet événement avec comme contenu l'id du jeu à joindre et son nom de joueur pour pouvoir joindre la queue de joueurs qui attendent d'être acceptés par un host dans un jeu 1v1. Le serveur va capter cet événement, ajouter le nom du joueur dans la queue et émet a son tour l'événement **'updateQueue'** au client qui est host de la partie de jeu avec comme contenu la queue qui a été mise à jour. Le client qui hoste la partie capte l'événement 'updateQueue' et met donc à jour sa queue de joueurs en attente pour joindre sa partie.
    - Pourquoi socket ? Pour pouvoir envoyer l'événement 'updateQueue' sans que le host de la partie ne le demande en boucle.

- **'checkPlayersInGame':** le client émet cet événement avec l'id du jeu voulu en contenu pour savoir quels joueurs se trouvent dans un jeu particulier. Le serveur capte l'événement et vérifie s'il y a des salles qui existent pour ce jeu. Il émet ensuite l'événement **'playersInGame'** qui contient le nombre de joueurs dans la salle de jeu s'il y a une salle existante ou 0 comme nombre de joueurs s'il n'y a pas de salle. Le client va ensuite capter cet événement et stocke le nombre de joueurs dans les informations de chaque jeu.
    - Pourquoi socket ? Les objets lobby sont contenus dans le fichier qui gère les sockets. Ainsi on évite les dépendances puisque se sont les sockets qui gèrent les parties de jeu.

- **'deleteRoom':** lorsqu'un client quitte la salle d'attente pour retourner dans la page de sélection de jeux, cet événement est envoyé au serveur qui le capte et enlève ce socket de la liste de salles.
    - Pourquoi socket ? On utilise les sockets pour pouvoir effacer le lobby mais aussi envoyer un événement 'updatePlayers' a toutes les autres sockets.

- **'removeFromQueue':** le client émet cet événement dans 2 situations:
    - Lorsque le host d'un jeu reçoit l'invitation d'un 2ème joueur qui veut rejoindre la partie que le host a créé, le host peut refuser ce joueur et attendre quelqu'un d'autre. Lorsque le host le refuse, il envoie l'événement **'removeFromQueue'** au serveur.
    - Lorsqu'un joueur est dans la salle d'attente et attend d'être accepté par un host d'une partie de jeu existante, il peut décider de quitter la salle d'attente pour retourner dans la page de sélection de jeux. Si c'est le cas, il envoie l'événement **'removeFromQueue'** quand il quitte la salle d'attente.
    - Dans les 2 cas, l'événement est émis avec comme contenu l'id du socket à enlever de la queue et l'id du jeu

  Lorsque le serveur capte cet événement et enlève le socket de la queue de joueurs en attente. Le serveur émet donc l'évènement **'refused'** qui sera capté par le client qui a été enlevé de la queue et ce dernier est redirigé vers la page de sélection de jeux. Le serveur émet ensuite l'événement **'updateQueue'** à l'host qui a créé la partie de ce jeu avec comme contenu la liste mise à jour des joueurs en attente pour joindre sa partie/salle.

    - Pourquoi socket ? Les sockets ici permettent au serveur d'envoyer au joueur un événement 'refused' sans que ce dernier ait à envoyer des requêtes pour vérifier.

- **'addToRoom':** Lorsque le host d'un jeu reçoit l'invitation d'un 2ème joueur qui veut rejoindre la partie que le host a créé, le host peut accepter ou refuser ce joueur. Lorsque le host accepte, il envoie l'événement **'addToRoom'** au serveur avec comme contenu l'id du socket du 2ème joueur à ajouter dans la salle de la partie de jeu.

  Le serveur capte cet événement et ajoute l'id du socket dans la salle du jeu. Il émet ensuite l'événement **'refused'** à tous les autres sockets dans la queue d'attente pour joindre la partie. Les clients captent l'événement **'refused'** qui les redirige vers la page de sélection de jeux.

  Le serveur émet ensuite l'événement **'goToGame'** avec comme contenu l'id de la salle dans laquelle se trouvent les 2 joueurs qui vont jouer 1v1. Les 2 clients captent cet événement qui les dirige vers la page de jeu 1v1.

  En dernier, le serveur émet l'événement **'username'** avec comme contenu les noms des deux joueurs, soit le host et l'invité. L'événement est capté par le client dans la page de jeu 1v1 qui va attribuer les noms des 2 joueurs à des attributs user1 et user2 de la page. On va donc pouvoir utiliser ces 2 attributs dans le fichier html de la page 1v1 pour afficher les noms des 2 joueurs.

  - Pourquoi socket ? Les sockets permettent au host, dès qu'il accepte un joueur, de lui envoyer un event 'goToGamePage' afin de rejoindre la page de jeu sans envoyer des requetes pour savoir s'il a été accepté ou non.

- **'differenceFound'**: lorsqu'un joueur trouve une différence dans le jeu, il émet cet événement avec comme contenu l'id de la salle et l'id de la différence identifiée. Le serveur capte ensuite l'événement et incrémente un compteur du nombre de différences trouvées par le joueur; si le host a émis l'événement, c'est son compteur qui est incrémenté, sinon c'est celui de l'invité. Le serveur va émettre l'événement **'differenceUpdate'** à la salle où se trouvent les 2 joueurs avec comme contenu les valeurs mise à jour des 2 compteurs de différences de l'host et de l'invité ainsi que l'id de la différence. Cet événement est capté du côté client et les valeurs du nombre de différences trouvées par chaque joueur sont mises à jour. Le client vérifie ensuite si un des 2 joueurs a gagné la partie.
  - Pourquoi socket ? Les sockets permettent d'envoyer un événement au deuxième joueur afin de le prévenir qu'une différence a été trouvée sans que ce dernier ait à envoyer des requêtes pour vérifier.

- **'giveUp':** lorsqu'un des 2 joueurs veut abandonner la partie en cours, il émet cet événement avec l'id de la salle comme contenu. Le serveur capte l'événement et vérifie l'id de quel joueur a émis l'événement. Le serveur émet ensuite à l'id du socket de l'autre joueur l'événement **'win'**. Le joueur qui a gagné la partie capte cet événement du côté client et un message de félicitations apparaît pour celui-ci.
  - Pourquoi socket ? Ici les sockets permettent au joueur qui abandonne d'envoyer un événement à l'autre joueur afin de lui informer qu'il a abandonné. Ainsi il n'aura pas à envoyer des requêtes pour vérifier si l'autre joueur a abandonné.
- **'checkLimitedGame':** lorsque le serveur reçoit cet événement, il cherche s'il y a un limitedLobby disponible (ou il n'y a pas de 2eme joueur). S'il n'y en a pas, alors le serveur va en créer un et mettre le joueur dans le lobby. S'il y en a un, le joueur sera mis en 2eme joueur dans le lobby et émettra un événement 'goToCoopGame'.
  - Pourquoi socket ? Les sockets nous permettent d'envoyer un événement au joueur en attente sans qu'il ait à constamment vérifier si un joueur est présent ou non.

- **‘gamesNumber’:** lorsque le serveur reçoit un événement gamesNumber, avec en paramètre ‘id de la room et le nombre de jeu disponible, il va initialiser la variable gamesNumber du lobby en question avec la valeur reçue en paramètre.
    - Pourquoi les sockets ? Le code aurait pu être fait en HTTP mais les lobbys sont dans le service SocketServeurManager, pour limiter les dépendances on utilise les sockets.
- **‘limitedDifferenceFound’:** lorsque le serveur reçoit cet événement, avec en paramètre l’id de la room, il incrémentera alors le nombre de différences trouvés du lobby et diminue le nombre de jeux. Ainsi il calcule aléatoirement un nombre pour le nouveau jeu. Ainsi il enverra aux membres du lobby un événement ‘LimitedDifferenceUpdate’ avec le nombre de différences trouvés et le numéro du nouveau jeu.
    - Pourquoi socket ? Ici les sockets permettent au serveur d’envoyer un événement au joueur qui n' a pas trouvé de différence sans qu’il soit obligé d’envoyer des requêtes pour vérifier si une différence a été trouvée.
- **‘limitedTimeGiveUp’:** lorsque le serveur reçoit cet événement, avec un parametre l’id de la room, il renvoie au deuxième joueur un événement ‘limitedTimeGiveUp’ pour qu’il soit informé que l’autre joueur a abandonné.
    - Pourquoi socket ? On utilise ici les sockets afin d'éviter à chaque joueur de vérifier continuellement si l’autre joueur a abandonné la partie.
- **‘systemMessage’:** lorsque le serveur reçoit cet événement, il verifie si c’est un message d’abandon et envoie le nom du joueur et un message d’abandon, sinon il envoie au joueur qui doit recevoir le message un événement ‘receiveSystemMessage’.
    - Pourquoi socket ? Les sockets permettent ici d’envoyer un message à un joueur sans que ce dernier n'ait à vérifier en continu si un message doit être recu.
- **‘sendChatToServer’**: lorsque le serveur reçoit cet événement, avec en paramètre le message à envoyer, ce dernier envoie un événement ‘receiveChatMessage’aux joueurs de la room avec le message.
    - Pourquoi socket ? Les sockets permettent ici d’envoyer un message à un joueur sans que ce dernier n'ait à vérifier en continu si un message a été envoyé.
- **‘globalMessage’:** lorsque le serveur reçoit cet événement, avec en paramètre le nouveau score d’un joueur, il met en forme le message et le renvoie avec un événement ‘receiveSystemMessage’ vers tous les sockets connectés.
    - Pourquoi socket ? Si tous les joueurs connectés sur le serveur doivent continuellement vérifier qu'un nouveau record à été battu, il y aurait trop de requetes. Les sockets permettent donc d’envoyer les informations que lorsqu' un nouveau record est battu.
- **‘systemMessageSolo’:** lorsque le serveur reçoit cet événement avec un message en paramètre, il traite le message et renvoie le message au joueur concerné avec un événement ‘receiveSystemMessageSolo’.
    - Pourquoi socket ? Ici l’utilisation des sockets n’est pas indispensable, le code aurait pu être fait en HTTP. Le plus important ici est que le serveur envoie le message pour pouvoir avoir un temps non trafiqué dans le chat.