École Centrale Lyon

ELC-D3
Applications Web
Rapport

Projet final - Jeu d'échec

Élèves :
Mohammed Adam Khali
Flavien Mairot

Enseignant :
René Chalon
Daniel Muller

2 avril 2024

Table des matières

1 Introduction et présentation du jeu **2**
1.1 Présentation du jeu 2
1.2 Arborescence du projet 3

2 Explication du code **4**
2.1 Communication serveur-client 4
2.2 Fonctionnement des scripts côté serveur 5
2.3 Fonctionnement des scripts côté client 6

3 Conclusion **8**

1 Introduction et présentation du jeu

1.1 Présentation du jeu

Pour notre projet, nous avons décidé de réaliser une application web de jeu d'échec jouable en ligne à deux (figures 1 et 2).

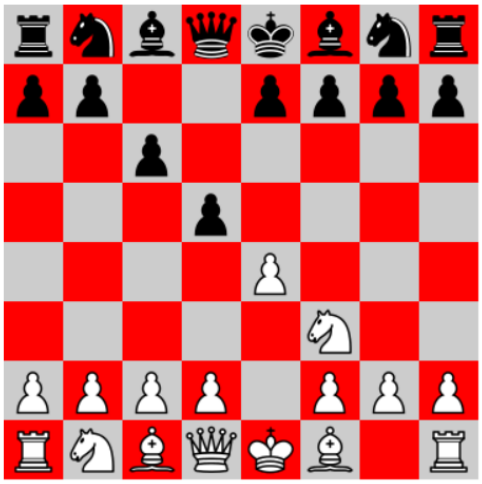

FIGURE 1 – Point de vue du joueur 1

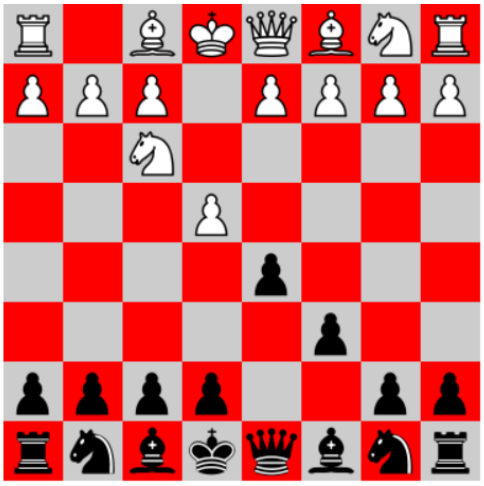

FIGURE 2 – Point de vue du joueur 2

L'application est capable de gérer plusieurs salles en simultané et ainsi d'héberger plusieurs partis sur un seul serveur. Chaque joueur peut donc jouer tour à tour ses pièces et l'application n'effectue que les coups autorisés. Les contributions de chaque joueur sont relayées aux autres grâce à une connexion permanente au serveur via le module socket.io. Nous avons également pris l'initiative de mettre en place une fenêtre de messagerie instantanée en temps direct pour permettre aux joueurs de communiquer entre eux et affichant les différentes informations de la partie (figure 3).
Une partie se termine lorsqu'un des deux joueurs s'est emparé du roi de son adversaire. Dans ce cas, les joueurs ont chacun l'apparition d'un message soit de victoire, soit de défaite et ils sont tous les deux déconnectés du serveur.

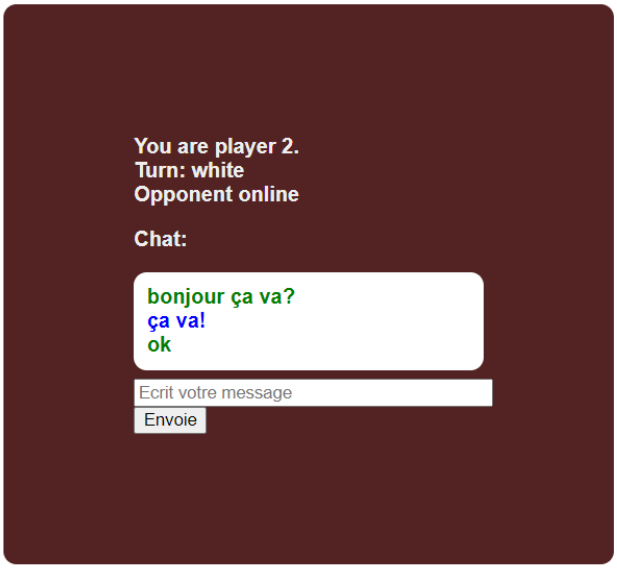

FIGURE 3 – Fenêtre de chat avec les différentes informations de la partie

1.2 Arborescence du projet

Notre projet s'organise sous la forme d'un dossier Chess comprenant plusieurs fichiers et dossiers (figure 4). Ces derniers sont organisés en partie serveur et client. Le code côté serveur, ou backend, comprend les fichiers server.js, user.js, move.js et les dépendances nécessaires au projet. Le code côté client ou frontend, quant à lui, comprend les fichiers index.html et script.js ainsi que les images nécessaires à l'application. Leurs interactions sont les suivantes.

1. **'server.js'** : Contient le code principal du serveur, y compris la gestion des connexions des clients, la logique de jeu, les événements socket.io, etc.
2. **'users.js'** : Gère la gestion des utilisateurs et des salles de jeu. Il contient des fonctions pour ajouter, supprimer et mettre à jour les utilisateurs et les salles, ainsi que pour récupérer des informations sur ces derniers.
3. **'move.js'** : Contient la logique de déplacement des pièces du jeu et détermine si un coup est autorisé ou non.
4. **'node modules'** : Dossier contenant toutes les dépendances Node.js installées pour le projet (non inclus dans la liste pour simplifier).
5. **'index.html'** : Page HTML du jeu, structure principale de la fenêtre du jeu.
6. **script.js** : Script JavaScript, gère-les événements côté client.
7. **'images'** : Dossier contenant les images des pièces en format PNG, pour avoir un jeu homogène plutôt qu'avec l'utilisation des caractères Unicode.

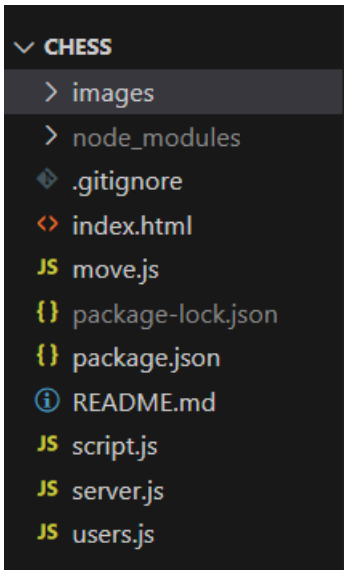

FIGURE 4 – Arborescence du projet

2 Explication du code

2.1 Communication serveur-client

Le serveur de l'application stocke différents salons avec chacun son échiquier et ses deux joueurs. Il n'y en a initialement qu'un seul, à la mise en route du serveur. Mais si plus de joueurs se connectent, d'autres salons sont créés.

Pour structurer la communication serveur-client, nous avons utilisé l'environnement Node JS qui est un environnement d'exécution JavaScript côté serveur utilisant le moteur JavaScript V8 de Google Chrome. Pour ce faire, il est d'abord nécessaire d'installer toutes les dépendances associées au sein du répertoire du jeu d'échec sur un terminal, puis de lancer un serveur via la commande 'node server.js'. La communication s'effectue alors par l'intermédiaire des Websockets. Les Websockets forment un protocole de communication permettant l'échange bidirectionnel et en temps réel de données entre un client et un serveur sur une seule connexion persistante. Pour les implémenter, nous avons utilisé la bibliothèque socket.io (figure 5).

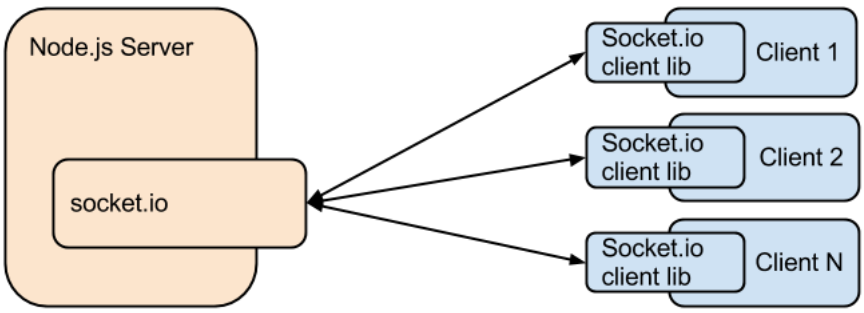

FIGURE 5 – Schéma socket.io

Dans notre application d'échecs, Socket.IO est utilisé pour transmettre les mouvements

des pièces entre les joueurs en temps réel. Lorsqu'un joueur effectue un mouvement, Socket.IO envoie ce mouvement au serveur. Ce dernier vérifie alors que ce coup est bien autorisé. Puis, le cas échéant, il le transmet à tous les autres clients connectés à la même partie, assurant ainsi une expérience de jeu synchronisée.

2.2 Fonctionnement des scripts côté serveur

Dans un premier temps, lorsque le serveur est mis en route, on importe toutes les dépendances nécessaires et on configure le serveur et les routes. Le serveur est créé avec Express.js et un serveur HTTP est attaché à celui-ci. Le port du serveur est défini par défaut sur le port 3000 (figure 6).

```
1   const app = require('express')();
2   const http = require('http');
3   const server = http.createServer(app);
4   const PORT = process.env.PORT || 3000;
5   const io = require('socket.io')(server);
6   const util = require('util')
```

FIGURE 6 – Imports et configuration serveur

On définit alors les différentes routes pour que le client puisse accéder aux fichiers HTML, JavaScript et aux images à partir du répertoire local du projet (figure 7).

```
9
10  //Routes
11  app.get('/', (req, res) => {
12      res.sendFile(__dirname+'/index.html');
13  });
14
15  app.get('/script.js', (req, res) => {
16      res.sendFile(__dirname+'/script.js');
17  });
18
19  app.get('/images/:file', (req, res) => {
20      try { res.sendFile(__dirname+'/images/'+req.params['file']+'.png');}
21      catch (error) { console.log(error); }
22  });
23
```

FIGURE 7 – Configuration des routes

Lorsqu'un client se connecte au serveur, l'événement 'connection' est déclenché. Le serveur vérifie s'il existe des salles de jeu disponibles. Si une salle est disponible, le client est ajouté à cette salle. Sinon, une nouvelle salle est créée. Pour la gestion des joueurs et l'attribution de ces derniers à différents salons, de nombreuses fonctions sont définies à part dans un fichier users.js évoqué précédemment et importé au début du code serveur

principal. Une fois connecté, le serveur envoie au client des informations telles que le joueur attribué (1 ou 2), la disposition actuelle des pièces d'échecs dans la salle et le joueur précédent (figure 8).

```
64      socket.emit('sendPlayer', { player: user.player });
65      socket.emit('sendPositions', {.positions: getRoomPositions(user.room) });
66      socket.emit('lastPlayer', getRoomLast(user.room));
67      io.to(user.room).emit('players', getUsersInRoom(user.room).length);
```

FIGURE 8 – Envoie des données de la partie au client

Le serveur est alors à l'écoute de différents événements que peut émettre le client. Ce dernier peut émettre un message, se déconnecter ou alors tenter un déplacement. Pour ce dernier événement, une fonction 'mouvement' est codé à part et est importé. Elle contient le cœur des règles du jeu dans la mesure où elle prend en entrée l'état de la partie, avec les différentes pièces en jeu, les joueurs et le mouvement tenté, et renvoie un booléen valant true si le coup et autorisé à ce stade de la partie.
Prenons l'exemple de la réception d'un message par le serveur (figure 9). On utilise bien l'objet socket, communiquant entre le serveur et le particulier, pour récupérer le message. On passe ensuite en argument une fonction de gestionnaire d'événement. Fonction anonyme qui récupère l'identifiant de la connexion au serveur puis utilise cette fois l'objet io pour émettre en retour un événement 'message' à tous les clients connecté dans la même salle que l'émetteur du message.

```
101     socket.on('message', (message) => {
102         user = getUser(socket.id);
103         io.to(user.room).emit('message', { message, emiter: user.player });
104     });
```

FIGURE 9 – Gestion de l'événement message reçu par le serveur

Enfin, le script serveur démarre ce dernier sur le port spécifié. Une fois que le serveur est en cours d'exécution et écoute les connexions entrantes, un message de débogage est imprimé dans la console (figure 10).

```
113   server.listen(PORT, () => {
114       console.log('Server running in port', PORT);
115   });
```

FIGURE 10 – Mise en route serveur

2.3 Fonctionnement des scripts côté client

Le code côté client de l'application est responsable de l'interaction avec l'utilisateur, de l'affichage du plateau et de la communication avec le serveur. Dans un premier temps, lorsqu'un client se connecte au serveur via le port adapté, ce dernier établit une connexion avec le serveur via Socket.IO dès le chargement de la page. La route définie précédemment

lui renvoie le fichier HTML central du code client 'index.html'.
Celui-ci contient divers éléments tels que les images png des pièces cachées initialement, la fenêtre de saisie du chat ou même le plateau qui se présentera sous la forme d'un Canvas mais aussi d'un script associé, éléments centraux du projet (figure 11).

```
91      <canvas id="chess" width="800" height="800"></canvas>
92      <script src="/script.js"></script>
```

FIGURE 11 – Canvas du plateau au sein du document HTML

Le script client récupère dans un premier temps l'objet Canvas, grâce à son identifiant, et établie une connexion avec le serveur grâce à la création d'un objet socket. Ce script a pour rôle principal la réception et l'émission d'événements auprès du serveur.
Il est alors nécessaire d'implémenter des listeners sur notre fenêtre pour détecter les interactions de l'utilisateur pour la suite de la gestion des événements (figure 12).

```
30  window.addEventListener('mousemove', (e) => {
31      var rect = canvas.getBoundingClientRect();
32      mouseX = e.clientX - rect.left;
33      mouseY = e.clientY - rect.top;
34  });
35
36  window.addEventListener('mousedown', (e) => {
37      mouseIsPressed = true
38  })
39
40  window.addEventListener('mouseup', (e) => {
41      mouseIsPressed = false
42  })
```

FIGURE 12 – Implémentation des listeners

Via l'objet socket, il est alors nécessaire d'implémenter la gestion des événements transmis par le serveur, notamment la récupération et le stockage des différentes données comme l'arrivée d'un nouveau joueur, ses messages, l'attribution d'un salon, la mise à jour des positions et d'autres (figure 13). Les exemples ci-dessous stockent les données du joueur en cours et des positions, soit dans le texte d'un élément HTML récupéré par id, soit dans une variable globale.

```
56  socket.on('sendPlayer', (data) => {
57      player = data.player;
58      document.getElementById('player').innerHTML = `You are player ${player}.`;
59  });
60
61  socket.on('sendPositions', (data) => {
62      positions = data.positions;
63  });
```

FIGURE 13 – Exemples d'événements reçu par le client lié à la récupération des données de la partie

Les événements transmis du client au serveur sont donc transmis pour certains lorsque des listeners sont appelés. C'est le cas des messages par exemple qui sont envoyés via un événement 'message' transmis par socket lorsque qu'un listener associé à l'élément text-box (qui est en fait un élément HTML input) est déclenché. Ce dernier listener se déclenche lorsque l'évènement 'touche entrée enfoncée' est détecté, il fait alors appel à la fonction inputSend (figure 14).

```
90  function inputSend() {
91      sendMessage(document.getElementById('text-box').value);
92      document.getElementById('text-box').value='';
93  }
94
95  function sendMessage(message) {
96      socket.emit('message', message);
97  }
```

FIGURE 14 – Fonctions liées à l'événement d'envoi de message

La totalité de l'affichage de la partie est gérée par une fonction draw qui est elle-même appelée périodiquement toutes les 10 millisecondes. Cette fonction fonctionne grâce à un objet images qui contient les éléments HTML associés aux images des pièces au format png. La fonction draw parcourt le canvas de manière similaire à un tableau en coloriant les cases selon leur parité afin de reproduire un damier. Ensuite, grâce aux informations récupérées du serveur et aux informations détectées sur la position de la souris, elle dessine les pièces sur le plateau ainsi que, éventuellement, la pièce sélectionnée par le joueur en train de jouer à l'endroit de son curseur.

3 Conclusion

Ce projet d'application d'échecs en temps réel nous aura donc permis de mettre en évidence l'intérêt de Node.js et Socket.IO pour créer une expérience de jeu interactive et synchronisée. En combinant la gestion côté serveur de Node.js avec la communication en temps réel entre client et serveur de Socket.IO via les WebSockets, nous avons pu développer une application web permettant à plusieurs utilisateurs de jouer ensemble à distance aux échecs. Cette approche démontre la capacité de ces technologies à faciliter

la création d'applications web dynamiques et collaboratives, offrant ainsi aux utilisateurs une expérience de jeu immersive en temps réel.