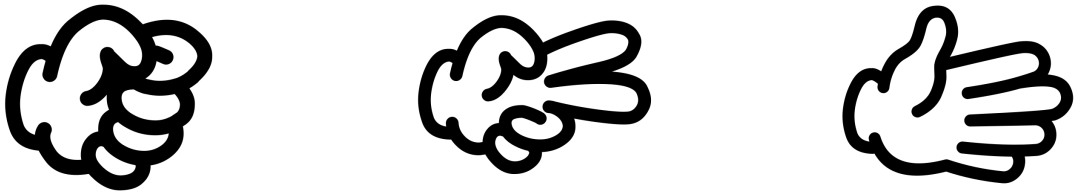

UNIVERSITÀ
DI PARMA

# Rock, Scissors, Paper

Francesco Gaetano Mincione

Matricola 310972



# Indice

# 1 Introduzione

Lo scopo di questa relazione è spiegare il processo di realizzazione e la logica del progetto. L'idea di base è semplice: far giocare due utenti uno contro l'altro ad una partita al meglio di cinque a sasso, carta, forbice. Essendo semplice si è voluto dare l'idea di un gioco che mira a non prendersi troppo sul serio sia a livello visivo che, soprattutto, uditivo con musiche solenni di sottofondo ed enfatizzando le scelte e risultati simulando una telecronaca.

## 1.1 Modello Client-Server e Socket

Il gioco è basato su un'**architettura Client-Server**: il client gestisce l'input dell'utente, ovvero la scelta della mossa, mostra quella fatta dall'avversario ed il punteggio aggiornato di conseguenza. Si occupa anche di gestire tutto il comparto sonoro, dalle musiche in background al commento che cambia a seconda dell'esito. Nel server invece è contenuta la logica del gioco, che semplicemente confronta le scelte fatte dal giocatore e ne comunica gli esiti ad entrambi. Client e server comunicano tramite socket che gestisce una comunicazione event-based.

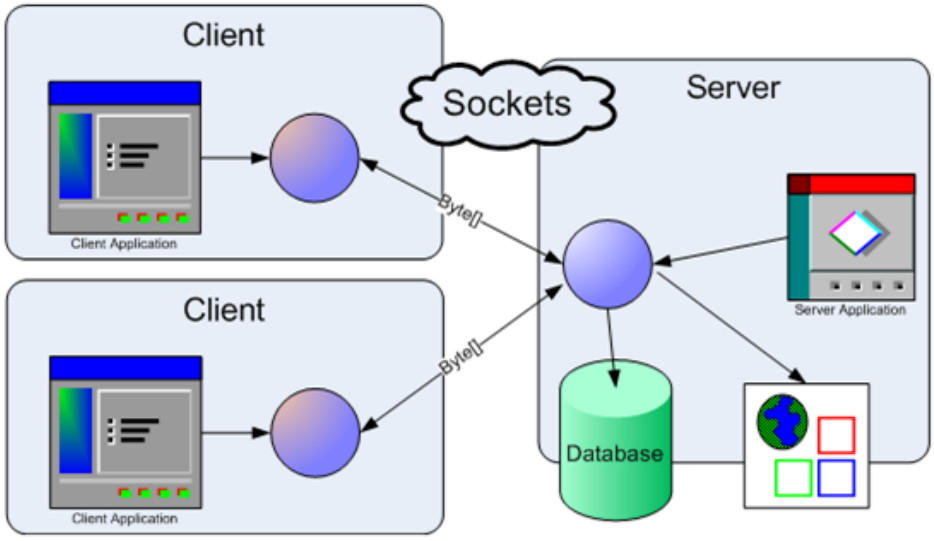



Figure 1: Architettura Client-Server

## 1.2 Linguaggi, librerie e programmi

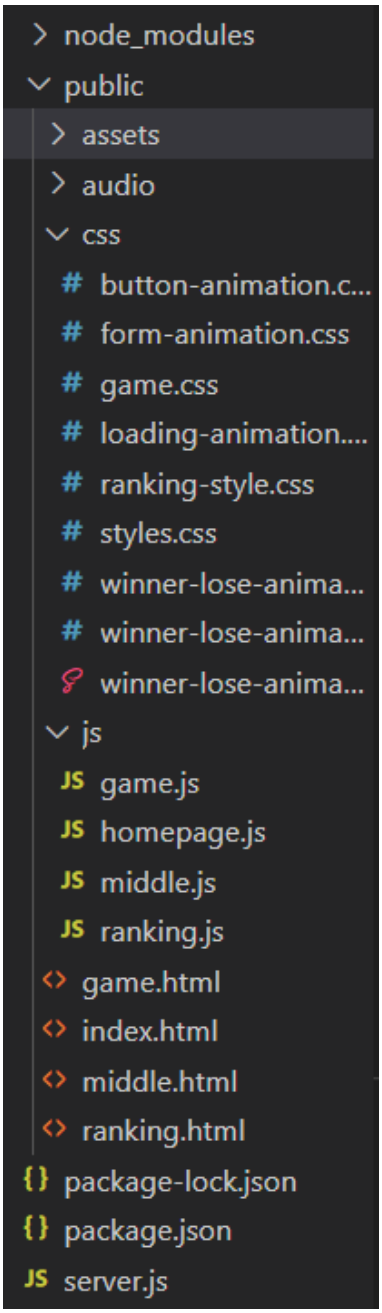
I linguaggi impiegati per la scrittura del progetto sono stati **HTML, CSS, JavaScript**. Per la classifica all-time e per le credenziali degli utenti è stato creato il database *NoSQL* **MongoDB** contenente entrambe le collezioni. È stato impiegato anche **GIMP** per la personalizzazione di alcune immagini utilizzate.
Le librerie utilizzate sono:

- **Node.js e Express:** node è stato utilizzato per la realizzazione del server e per la gestione del gioco. Express invece permette la renderizzazione dei file lato client

- **mongoose:** per la connesione al database MongoDB
- **bcrypt:** per l'hashing delle password inserite dagli utenti
- **body-parser:** per le operazioni di GET e POST
- **Sass:** un preprocessore di css per definire lo stile di alcune scritte
- **jQuery:** utilizzata per la manipolazione degli elementi presenti nell'HTML del gioco

## 1.3 Struttura del progetto

Di seguito una spiegazione generale di come è stato suddiviso il codice che compone il progetto nella sua interezza. La suddivisione è stata fatta in ottica di avere modularità, scalabilità e riutilizzabilità del codice tutto.




*server.js:* è il cuore pulsante di tutto il progetto, ed è suddivisibile in 3 macroaree di funzionamento: **setup e aggiornamento del database, gestione dei path,** (ad esempio per il redirecting tra le varie pagine)**, gestione delle connessioni e richieste dei client** (in particolare quelle riguardanti il gioco).

*index.html/homepage.js:* si tratta dei file che gestiscono la pagina iniziale contenenti il form per l'iscrizione e login dell'utente.

*middle.html/middle.js:* una volta effettuato il login l'utente verrà reindirizzato in questa pagina, contenente a sua volta due bottoni che rimandano l'utente alla pagina della classifica o a quella per creare/cercare una stanza.

*ranking.html/ranking.js:* apposita pagina per mostrare la classifica all-time all'utente.

*game.html/game.js:* sono i file dove avviene prima la creazione della stanza, mostrato il codice da condividere all'avversario per poter giocare, e dopodiché la gestione del gioco: dalla presa degli input dell'utente, al mostrare la scelte dell'avversario e l'aggiornamento dello score oltre che l'esito della partita. Viene gestita anche la parte di telecronaca. Il tutto avviene attraverso lo scambio di messaggi col server in modo tale che questi siano in grado di mostrare e sentire gli esiti nel modo corretto

all'utente.

Un discorso a parte va fatto per la parte dei file *css*: si è cercato di gestire lo stile dei background e il posizionamento di alcuni pulsanti delle singole pagine utilizzando *styles.css*, mentre per lo stile di alcuni dei singoli elementi, come bottoni, form ed alcune scritte, si è preferito dedicarci degli specifici css. La pagina del gioco ha invece un suo css dedicato visto l'utilizzo dinamico di alcuni dei suoi elementi.

Chiudono poi questa panoramica sulla struttura del progetto le cartelle *assets*, contenente le immagini utilizzate, e *audio*, contenente soundtracks e file audio utili a simulare la telecronaca.

# 2 Funzionamento del gioco

Le regole del gioco sono esattamente quelle di sasso, carta, forbice iterate in una partita al meglio di 5 (dunque chi vince per primo tre round risulta il vincitore). In questa sezione si vuole dunque andare nel merito del codice per spiegare nel dettaglio l'interazione tra client e server durante il match tra i due user.

## 2.1 Creazione stanza

Una volta che l'utente è stato reindirizzato in *game.html* può scegliere se creare una stanza o aggiungersi ad una già esistente. Se l'utente preme *"Create Game"* attiva l'evento "onclick" nell'html che chiamerà la funzione **createGame()** definita in *game.js*: questa ha lo scopo di informare il server che un utente sta creando una stanza e ne fornisce lo username, salvato in precedenza in fase di login.

**game.html:**

```
66              <button class="form-control  glow-on-hover" onclick="createGame()">Create Game</button>
```

**game.js:**

```
7   function createGame() {
8     player1 = true;
9     const playerUsername = localStorage.getItem("username");
10    socket.emit("createGame", {playerUsername: playerUsername});
11  }
```

Il server di risposta crea un id della stanza (attraverso la funzione **makeid(length)** che, semplicemente, da una stringa contenente tutto l'alfabeto maiuscolo e minuscolo e i numeri da 0 a 9 ne seleziona randomicamente sei caratteri), crea la stanza con quell'id e vi ci aggiunge l'utente. Lo username di quest'ultimo viene anche aggiunto in una variabile **usernameArray**[] che tornerà utile per la trasmissione dei risultati associati agli utenti. La cosa importante comunque è che così facendo sia per il server che per il client **chi crea la stanza sarà da questo momento considerato come player1, mentre chi entra poi sarà il player2**. Infine trasmette al creatore della stanza che questa è stata creata e gli fornisce l'id da condividere.

**server.js:**

```
193   socket.on("createGame", (data) => {
194     const roomUniqueId = makeid(6);
195     usernameArray.push(data.playerUsername);
196     console.log(data.playerUsername);
197     rooms[roomUniqueId] = {};
198     socket.join(roomUniqueId);
199     socket.emit("newGame", { roomUniqueId: roomUniqueId });
200   });
```

Lato client una volta ricevuto "newGame" apparirà all'utente un form con l'id che può essere copiato tramite un apposito tasto e condiviso privatamente con l'altro utente con il quale si vuole giocare.

## 2.2 Unirsi alla stanza


Si vuole ora discutere il codice che si occupa di far unire un utente alla partita. Questo dovrà inserire il codice fornitogli da chi ha creato la stanza come mostrato nella figura a sinistra e premere il tasto *"Join Game"*. Lato client la logica usata è molto simile alla precedente: una volta che è stato inserito il codice e premuto il tasto attiverà l'evento "on click" nel *game.html* che chiama la funzione **joinGame()** implementata in *game.js*: questa prende lo username dell'utente e l'id inserito e li trasmette al server. Il server a questo punto, dopo aver controllato che nella stanza vi sia effettivamente il player1, aggiunge l'utente e informa entrambi che sono pronti per giocare tramite l'emit di "playersConnected". Di seguito i codici:

**game.html:** *"on click" viene chiamata la funzione*

```
69          <button class="form-control  glow-on-hover" onclick="joinGame()">Join Game</button>
```

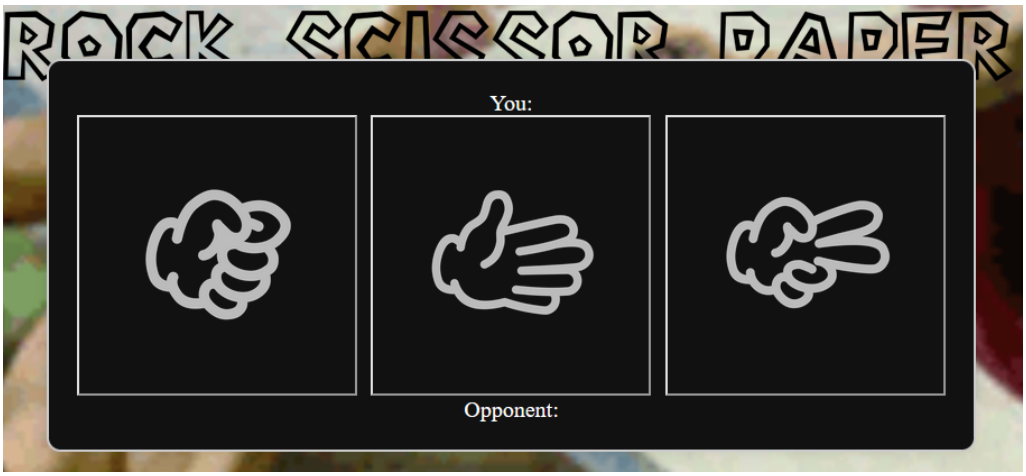
**game.js:** *username e id inserito vengono inviati al server*

```
13  function joinGame() {
14    const playerUsername = localStorage.getItem("username");
15    roomUniqueId = document.getElementById("roomUniqueId").value;
16    socket.emit("joinGame", { roomUniqueId: roomUniqueId, playerUsername: playerUsername });
17  }
```

**server.js:** *dopo aver inserito l'utente nella stanza informa entrambi che si può cominciare a giocare*

```
202   socket.on("joinGame", (data) => {
203     if (rooms[data.roomUniqueId] != null) {
204       usernameArray.push(data.playerUsername);
205       console.log(data.playerUsername);
206       socket.join(data.roomUniqueId);
207       socket.to(data.roomUniqueId).emit("playersConnected", {});
208       socket.emit("playersConnected");
209     }
210   });
```

## 2.3 Comunicazione delle scelte




A questo punto il client "pulisce" l'area, attiva gli effetti audio e di telecronaca (di cui si parlerà in una sezione apposita) facendo apparire all'utente una finestra con 3 bottoni, ai quali sono associati i valori di "rock", "scissors" e "paper" che, una volta premuti, chiameranno la funzione **sendChoices(rpsValue)** implementata in *game.js*. Questa svolge tre compiti: informa il server della scelta presa, simula la telecronaca e mostra all'utente unicamente il bottone con la scelta presa da quest'ultimo. Per quanto detto in precedenza il client sa già distinguere se chi sta facendo la scelta è il player1 o il player2 in quanto al momento della creazione della stanza una variabile **player1** è stata settata a *true*, e in base a ciò informerà il sever se la scelta è stata fatta dal primo piuttosto che dal secondo nell'**emit**. Per gli aggiornamenti visivi "dinamici" si è preferito utilizzare *jQuery* per trattare gli elementi prensenti nell'html invece poiché molto più manipolabili e responsivi rispetto all'uso classico (come ad esempio l'utilizzo di **getElememtById(..)**).


```
function sendChoice(rpsValue) {
  const choiceEvent = player1 ? "p1Choice" : "p2Choice";
  socket.emit(choiceEvent, {
    rpsValue: rpsValue,
    roomUniqueId: roomUniqueId,
  });
 ///CODICE TELECRONACA OMESSO
  let playerChoiceButton = document.createElement("button");
  playerChoiceButton.style.display = "block";
  playerChoiceButton.classList.add(rpsValue.toString().toLowerCase());
  playerChoiceButton.innerText = rpsValue;
  $("#player1Choice").empty();
  $("#player1Choice").append(playerChoiceButton);

}
```

A questo punto il server in ascolto associa le rispettive scelte utenti nella stanza e chiama la funzione **declareRoundWinner(roomUniqueId)** di cui si parlerà nella prossima sezione.

```
socket.on("p1Choice", (data) => {
  console.log("P1choice", data);
  let rpsValue = data.rpsValue;
  if (rooms[data.roomUniqueId]) {
    rooms[data.roomUniqueId].p1Choice = rpsValue;
    console.log(rooms[data.roomUniqueId].p2Choice);
    if (rooms[data.roomUniqueId].p2Choice) {
      declareRoundWinner(data.roomUniqueId);
    }
  }
});
```

```
socket.on("p2Choice", (data) => {
  console.log("P2choice", data);
  let rpsValue = data.rpsValue;
  if (rooms[data.roomUniqueId]) {
    rooms[data.roomUniqueId].p2Choice = rpsValue;
    if (rooms[data.roomUniqueId].p1Choice) {
      declareRoundWinner(data.roomUniqueId);
    }
  }
});
```

## 2.4 La funzione declareRoundWinner

La funzione **declareRoundWinner(roomUniqueId)** contiene a tutti gli effetti tutta la logica del gioco:

- Prende dalla stanza le scelte fatte dagli utenti e le invia alla funzione **determineRoundResult (p1choice, p2choice)**

```
let p1Choice = rooms[roomUniqueId].p1Choice;
let p2Choice = rooms[roomUniqueId].p2Choice;
let roundResult = determineRoundResult(p1Choice, p2Choice);
```

- Come da nome **determineRoundResult(..)** confronta, seguendo le regole del gioco sasso, carta, forbice, le scelte fatte dai giocatori e ritorna il vincitore
- A questo punto il server incrementa lo score e informa il client tramite l'**emit** "result", fornendogli lo score, entrambe le scelte e usernames

```
269   // Update scores based on round result
270   if (roundResult === "p1") {
271     player1Score++;
272   } else if (roundResult === "p2") {
273     player2Score++;
274   }
275
276   // Emit round result to clients
277   io.sockets.to(roomUniqueId).emit("result", {
278     winner: roundResult,
279     player1Score: player1Score,
280     player2Score: player2Score,
281     p1Choice: p1Choice,
282     p2Choice: p2Choice,
283     player1Name: usernameArray[0],
284     player2Name: usernameArray[1],
285   });
```

- Nella funzione viene anche fatto il controllo degli score per determinare la condizione di **gameOver**: questa avviene quando uno dei due giocatori vince tre partite, in tal caso viene nel "result" nel messaggio agli utenti oltre alle cose già dette in precedenza viene comunicato che il gioco è finito e il vincitore della partita.

```
if (player1Score === 3 || player2Score === 3) {
  const gameWinner = player1Score === 3 ? "p1" : "p2"; // Identify the actual winner
  let gameOverText = player1Score === 3 ? "You win!" : "Opponent wins!";
  io.sockets.to(roomUniqueId).emit("result", {
    winner: "gameOver",
    gameWinner: gameWinner,
    player1Score: player1Score,
    player2Score: player2Score,
    gameOverText: gameOverText,
    player1Name: usernameArray[0],
    player2Name: usernameArray[1],
  });
```

- Infine, nel game over, il server aggiorna il database con gli score fatti da entrambi i giocatori e libera le stanza.

```
303     // Update scores in the database
304     try {
305       await Score.updateOne({ username: usernameArray[0] }, { $inc: { score: player1Score } });
306       console.log(usernameArray[0]);
307       await Score.updateOne({ username: usernameArray[1] }, { $inc: { score: player2Score } });
308       console.log(usernameArray[1]);
309     } catch (error) {
310       console.error('Error updating scores:', error);
311     }
312
313     // Reset scores for the next game
314     player1Score = 0;
315     player2Score = 0;
316     usernameArray = [];
317   }
```

## 2.5 Svolgimento gioco lato client

A questo punto il client, che era in ascolto, non appena arriva la "result", utilizzando i dati fornitogli dal server, si occupa di fare le seguenti cose:

- Mostrare al giocatore visivamente la scelta dell'avversario, andando quindi a manipolare con *jQuery* gli elementi nell'html definiti in un'area apposita per il giocatore avversario, che consiste nel mostrare un bottone con l'immagine della scelta


```
119   let dest = "";
120   if (!player1) {
121     dest = "p1Choice";
122   } else {
123     dest = "p2Choice";
124   }
125   console.log(data);
126   switch (data[dest]) {
127     case "Rock":
128       let $rock = $("<button/>", {
129         id: "rockButton",
130         class: "rock",
131       });
132       $("#player2Choice").append($rock);
133       break;
```

Figure 2: In questo caso viene mostrata la scelta "rock" fatta dall'avversario

- Allo stesso modo aggiorna lo score, presi i dati dallo scambio di messaggi seleziona ed aggiorna l'elemento presente nell'*game.html*

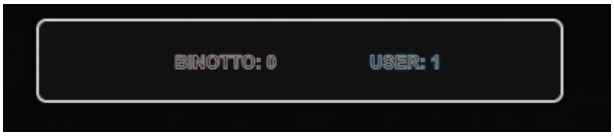



Figure 3: Cosa vede l'utente

```
<div id="scoreBox" class="score-box" style="display: none;">
        <p id="player1Score" class="score-text" style="font-size: 20px;" >Player1: 0</p>
        <p id="player2Score" class="score-text" style="font-size: 20px;">Player2: 0</p>
</div>
```

Figure 4: game.html

```
160   // Update the player scores
161   player1Score = data.player1Score;
162   player2Score = data.player2Score;
163
164   // Update player names and scores
165   $("#player1Score").text(`${data.player1Name}: ${data.player1Score}`);
166   $("#player2Score").text(`${data.player2Name}: ${data.player2Score}`);
167
168   $("#scoreBox").css("display", "block");
```

Figure 5: game.js

- Se i giocatori non hanno ancora raggiunto i 3 punti resetta l'area e ripropone all'utente le scelte con i tre bottoni ricominciando quanto descritto in 2.3

- Nel caso in cui invece fra i dati del messaggio ricevuto vi fosse il **gameOver**, il client a seconda se l'utente ha vinto o perso (informazione sempre ricevuta dal server) lo reindirizza in una schermata che gli comunicherà il risultato con una scritta "winner" o "you lost!" e, dopo 10 secondi, lo riporta in *middle.html*


Questo viene utilizzato sfruttando il metodo **setTimeout**, il quale permette permette di chiamare una funzione dopo *ms* indicati. Nell'ordine (seguendo codice sotto): viene svuotata l'area di gioco, controllato se il giocatore ha vinto o perso in modo da scegliere cosa far vedere e sentire all'utente, nell'esempio si fa vedere che nel caso il vincitore sia il player1 viene preparata la scritta da mostrare e riprodotte un motivetto e la telecronaca.

```
setTimeout(() => {
  if (data.winner === "gameOver") {

    // Clear player's choice buttons
    $("#gamePlay").empty();
    $("#scoreBox").remove();

    const winnerTextElement = $("#winnerText");

    if (data.gameWinner === "p1" && player1) {
      winnerTextElement.text("Winner").addClass("winner-text").removeClass("loser-text");
      $(".animation-container").fadeIn();
        const winFanfare = document.getElementById("winFanfare");
        winFanfare.currentTime = 0; // Reset the playback time
        winFanfare.play();
      setTimeout(() => {
        const victorySound = document.getElementById("victorySound");
        victorySound.currentTime = 0; // Reset the playback time
        victorySound.play();
        }, 1000);
```

Dopodiché la scritta viene mostrata e, sempre usando **setTimeout**, viene simulato un caricamento per il reindirizzamento alla pagina *middle.html*

```
215     // Display the winner/loser text
216     winnerTextElement.css("display", "block");
217
218     setTimeout(function() {
219       window.location.href = 'middle.html'; // Redirect to middle.html
220     }, 10000);
```

# 3 Telecronaca e soundtracks

Si è voluto dare un spazio apposito per soundtrack e telecronaca, seppur breve, perché li si considera elementi molto nell'ottica di migliorare l'esperienza dell'utente: l'idea, come detto all'inizio, è quella di enfatizzare il gioco tutto. Le soundtracks di sottofondo sono state prese dal gioco *Pokémon Battle Revolution* (gioco per Wii del 2006) che accompagnano le lotte dei mostriciattoli nell'arena, mentre gli spezzoni di telecronaca sono stati presi da *SuperSmash Bros Ultimate* (gioco per switch del 2018), anche questa usata per commentare i combattimenti in arena mentre i personaggi si scontrano fra loro.

## 3.1 Funzionamento

Il funzionamento è molto semplice: ogni file audio viene caricato come elemento nel file html in cui verrà usato utilizzando il parametro **preload**, che consente al browser ad effettuare una richiesta per una risorsa senza bloccare l'evento di caricamento del documento. Nel caso in cui la sountrack debba partire per prima viene anche indicato il parametro **autoplay loop**.In generale poi questi elementi, specialmente per quanto riguarda le pagine inerenti al gioco, verranno poi manipolate, lato client, a seconda dell'utilizzo nei rispettivi file javascript. Verrano ora mostrati e discussi degli utilizzi significativi.

```
<audio id="backgroundMusic" autoplay loop preload="auto">
    <source src="audio/middle.mp3" type="audio/mpeg">
</audio>
```

## 3.2 Soundtracks partita

Quando comincia la partita, ovvero quando i giocatori si trovano davanti alle tre scelte da effettuare per la prima volta, viene messa in pausa la soundtrack di sottofondo e successivamente

ne viene avviata una scelta in mondo randomico tra tre, sempre in ottica di migliorare l'esperienza del giocatore:

```
const backgroundMusic = document.getElementById("backgroundMusic");
backgroundMusic.pause();

const soundtracks = ["audio/soundtrack1.mp3", "audio/soundtrack2.mp3", "audio/soundtrack3.mp3"];

// Randomly select a soundtrack
const randomIndex = Math.floor(Math.random() * soundtracks.length);
const randomSoundtrack = soundtracks[randomIndex];

// Update the audio source and play
backgroundMusic.src = randomSoundtrack;
backgroundMusic.play();
```

## 3.3 Gestione della telecronaca

La telecronaca è stata aggiunta per enfatizzare scelte e risultati del round e della partita dell'utente. Per quanto riguarda le scelte la logica è sempre quella di associare l'attivazione di un del commento all'interno della funzione **sendChoice(rpsValue)** già discussa in 2.3: viene selezionato l'elemento nell'html, si setta **currentTime** a 0 in modo tale da far cominciare l'audio della telecronaca al secondo zero e e viene fatta riprodurre con **play()**.

```
266     const niceCallSound = document.getElementById("niceCall");
267     niceCallSound.currentTime = 0; // Reset the playback time
268     niceCallSound.play();
```

Figure 6: In questo caso ogni volta che l'utente preme un tasto il telecronista dirà "NICE CALL"

Per gli esiti dei round si è optato invece di non farli vedere visivamente all'utente (se non attraverso lo score) ma renderli noti attraverso la telecronaca. La logica usata per farlo è stata la stessa appena descritta con l'aggiunta di un piccolo ritardo di *1000 ms* attraverso **setTimeout()** per rendere la simulazione della telecronaca più credibile. Stesso discorso per quanto riguarda il commento alla fine sull'esito della partita.

```
109     } else {
110       winnerText = `It's a draw`;
111       setTimeout(() => {
112       const drawSound = document.getElementById("draw");
113       drawSound.currentTime = 0; // Reset the playback time
114       drawSound.play();
115       }, 1000);
116     }
```

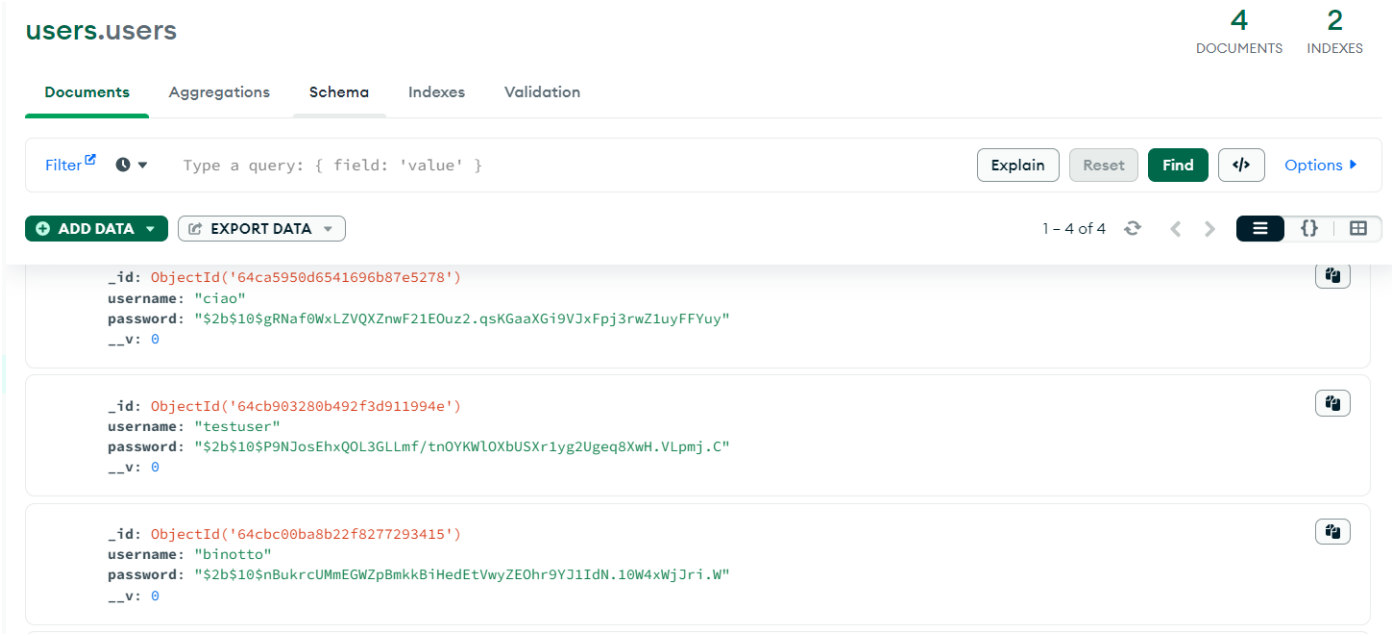
Figure 7: Nel caso in cui l'esito del round sia un pareggio il telecronista dirà "DRAW!"

# 4 MongoDB per il database


Per la raccolta dei dati si è deciso di utilizzare **MongoDB**, un DBMS di tipo *NoSQL*. Sebbene all'inizio la scelta è stata fatta più per motivi di curiosità nel provare qualcosa di diverso rispetto a quanto già visto in altri corsi con i database di tipo relazione, alla fine a livello di scrittura del codice è risultata una scelta comoda principalmente per i metodi forniti da **mongoose** per la gestione dei database legati all'utilizzo di *node.js*. Solitamente questo tipo di database comunque viene usato per la velocità e la scalabilità che offre nell'inserimento e recupero dei dati, ma non è stato questo il caso poiché il database è composto solamente di due collezioni e neanche così densamente popolate. Inoltre per la visualizzazione del database è scaricabile anche **MongoDBCompass**, utile per vedere se i dati vengono effettivamente caricati e aggiornati correttamente.

## 4.1 Struttura e collezioni

Come già detto la struttura del database è molto semplice: è composta da due collezioni, la prima *users* contente lo username, posto come unique, con la password associata (criptata usando **bcrypt**, che si basa su una cifratura Blowfish), la seconda *scores* lo username e il relativo score. Entrambi vengono creati e manipolati dal server.




## 4.2 Iscrizione e Login

L'iscrizione e login consistono in uno scambio di messaggi tra client e server. Per quanto riguarda l'iscrizione dopo che l'utente ha immesso i dati e premuto il tasto *subscribe* viene

fatta una richiesta al server usando una richiesta asincrona (**fetch**) per l'inserimento dell'utente.

```
50    subscribeButton.addEventListener("click", async (event) => {
51        event.preventDefault();
52
53        const username = document.getElementById("username").value;
54        const password = document.getElementById("password").value;
55        try {
56          const response = await fetch("/subscribe", {
57            method: "POST",
58            headers: {
59              "Content-Type": "application/json",
60            },
61            body: JSON.stringify({ username, password }),
62          });
```

Il server, ricevuta la richiesta, controlla se il nome inserito dall'utente esiste già nella collezione User attraverso il metodo **findOne** di mongoose: se non è presente lo salva (attraveso il metodo **save()** assieme alla password inserita e contemporaneamente aggiorna Score inizializzandone il punteggio a 0 e restituisce un messaggio di successo, altrimenti restituisce messaggio di fallimento.

```
app.post('/subscribe', async (req, res) => {
  const { username, password } = req.body;

  try {
    // Check if the user already exists in the database
    const existingUser = await User.findOne({ username: username });
    if (existingUser) {
      return res.status(409).json({ message: 'Username already exists. Please choose a different username.' });
    }

    // Create a new instance of the User model with the provided data
    const newUser = new User({ username: username, password: password });
    // Save the new user to the database
    await newUser.save();

    // Add the new user to the scores collection with an initial score of 0
    const newScore = new Score({ username: username, score: 0 });
    await newScore.save();

    return res.status(201).json({ message: 'User subscribed successfully!' });
  } catch (error) {
    console.error('Error during subscription:', error);
    return res.status(500).json({ error: 'Internal server error' });
  }
});
```

A seconda della risposta poi il client informa l'utente con l'oppurtuno messaggio a seconda se l'iscrizione è avvenuta con successo o meno.

```
64        if (response.ok) {
65          // If subscription was successful, show an alert message
66          alert("Subscription successful!");
67        } else {
68          // If subscription failed, display the error message from the server response
69          const data = await response.json();
70          alert(data.message);
71        }
72      } catch (error) {
73        console.error("Error during subscription:", error);
74        alert("An error occurred during subscription. Please try again later.");
75      }
```

Per quanto riguarda il login, il discorso è molto simile per quanto riguarda il botta e risposta tra client e server, cambiano solamente i tipi di controlli da effettuare sul database: dapprima viene controllato se lo username esiste, in caso contrario si informa l'utente, dopodiché viene confrontata la password inserita con quella con quella presente nella collezione. Qui viene usata la possibilità offerta da **mongoose** di poter associare un proprio metodo alla collezione. In questo caso viene passata la password inserita al metodo **comparePassword(password)** che al suo interno usa il metodo **compare** di **bcrypt** che permette il confronto tra le password hashate, essendo che nel database sono salvate come tali.

```
68  // Method to compare provided password with the hashed password
69  userSchema.methods.comparePassword = function (candidatePassword) {
70    return bcrypt.compare(candidatePassword, this.password);
71  };
```

## 4.3 Display e aggiornamento della classifica

Il database viene manipolato anche nel caso in cui l'utente voglia vedere la classifica *All-time* e al termine di di una partita fra due utenti aggiornando il punteggio di entrambi. Per quanto riguarda il primo caso la comunicazione che avviene tra client e server è molto simile a quanto già descritto nel paragrafo precedente: il client (*ranking.js*) tramite una funzione **fetchRankingData** far richiesta al server, questo risponde utilizzando il metodo **find** di mongoose sulla collezione dei primi dieci giocatori col punteggio più alto, il client poi si occupa di mostrarla creando una lista con ogni entry passata dal server.

```
app.get('/getRanking', async (req, res) => {
  try {
    // Retrieve the top 10 scores from the scores collection, sorted in descending order
    const rankingData = await Score.find({}, 'username score').sort({ score: -1 }).limit(10);

    // Return the ranking data as a JSON response
    res.status(200).json(rankingData);
  } catch (error) {
    console.error('Error getting ranking data:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Figure 8: server.js

```
async function fetchRankingData() {
    try {
        // Fetch the ranking data from the server
        const response = await fetch('/getRanking');
        const rankingData = await response.json();

        // Clear any existing content from the ranking list
        rankingList.innerHTML = '';

        // Loop through the ranking data and add each entry to the ranking list
        rankingData.forEach((entry, index) => {
            const listItem = document.createElement('li');
            listItem.textContent = `${index + 1}. ${entry.username} - ${entry.score}`;
            rankingList.appendChild(listItem);
        });
```

Figure 9: ranking.js


Figure 10: Cosa vede l'utente

Per quanto riguarda l'aggiornamento della classifica, questo viene fatto immediatamente dopo la fine della partita tra due utenti, aggiornando lo score utilizzando il metodo di mongoose **updateOne** dove il primo parametro funziona da "filtro" e il secondo indica cosa aggiornare.

```
302     // Update scores in the database
303     try {
304       await Score.updateOne({ username: usernameArray[0] }, { $inc: { score: player1Score } });
305       console.log(usernameArray[0]);
306       await Score.updateOne({ username: usernameArray[1] }, { $inc: { score: player2Score } });
307       console.log(usernameArray[1]);
308     } catch (error) {
309       console.error('Error updating scores:', error);
310     }
```