Guess the Song

Marco Avagnano
marco.avagnano@studio.unibo.it

Simone Fontanella
simone.fontanella@studio.unibo.it

24 novembre 2022

Indice

Elenco delle figure

Questo progetto nasce dall'idea del noto programma televisivo Sarabanda. Più precisamente abbiamo preso spunto dall'ultima fase di gioco, il 7x30, dove il concorrente deve indovinare 7 canzoni in 30 secondi. Per semplificare la fase di gioco abbiamo realizzato una versione rivisitata in cui l'utente ha a disposizione 30 secondi per ogni canzone e il nome dell'autore. Il gioco é suddiviso in turni, il numero di turni é selezionabile dal menù iniziale, ad ogni turno il giocatore deve indovinare una sola canzone e riceve dei punti in caso di risposta corretta. Alla fine di ogni turno viene visualizzata la classifica parziale con il totale dei punti ricevuti da ogni utente, e il tempo complessivo impiegato per rispondere. Vince chi alla fine ha più punti, o in caso di pareggio chi ha impiegato di meno complessivamente per rispondere.

Capitolo 1

Analisi dei requisiti

1.1 Obiettivi

L'obiettivo del progetto è quello di sviluppare un'architettura client-server in grado di riprodurre il gioco appena descritto. Il server ha il compito di inviare le canzoni, validare le risposte, assegnare i punteggi e redigere la classifica alla fine di ogni turno. Per la prioritizzazione dei nostri obiettivi utilizzeremo il metodo MoSCoW:

1. MUST:
    - Modellazione di tutti i sistemi;
    - Creazione e gestione delle lobby lato Server;
    - Sincronizzazione server-client
    - Validazione delle risposte lato Server
    - Gestione degli stati di gioco lato Server;
    - Creazione di un client Web;
    - Creazione di Test per verificare il corretto funzionamento del sistema
2. SHOULD:
    - Creazione di un client web responsive;
3. WOULD:
    - Sistema di autenticazione;
    - Possibilità di consultare i report delle partite passate;
4. WON'T:
    - Un'interfaccia grafica e curata nel dettaglio e personalizzata.

Per l'implementazione del progetto abbiamo deciso di utilizzare lato server il tool-kit Vert.x e Junit per i test, mentre per il client la libreria React.js. Sempre lato server é stata utilizzato gradle per automatizzare la compilazione e l'esecuzione del progetto, e per la gestione delle dipendenze.

1.2 Divisione del lavoro

La fase di progettazione é stata svolta da tutti i membri del gruppo, successivamente in fase di sviluppo un membro del gruppo si é occupato della parte di gioco mentre l'altro delle lobby. Una volta terminata una prima versione funzionante del gioco, il primo membro si é occupato dei punti opzionali riguardanti gli aiuti in gioco, mentre il restante si é occupato della gestione dell'autenticazione di un utente e del salvataggio nel database in remoto dello storico delle partite.

Capitolo 2

Tecnologie

- `Gradle`
- `Git`
- `VertX`
- `Junit`
- `ReactJS`
- `SockJS`
- `MongoDB`
- `Docker`

2.1 Gradle

Gradle è uno strumento di automazione della compilazione di applicativi Java, che permette di scaricare dipendenze, compilare e testare codice in modo veloce e richiedendo all'utente un'interazione minima. Gradle permette infatti di utilizzare librerie open source senza doverle importare esplicitamente, ma semplicemente andando a specificare all'interno del file build.gradle la repository in cui tale codice è presente. Nel nostro caso Gradle è stato utilizzato per dichiarare in modo semplice e immediato le librerie relative a Verxt, MongoDB, GSON. Come specificato sopra, l'uso di Gradle non solo garantisce un modo immediato per la gestione delle dipendenze, ma permette di compilare e testare il codice per mezzo di comandi utilizzabili tramite terminale.
Esempi di comandi gradle:

```
$ gradle run
$ gradle test
```

Figura 2.1: Logo di Gradle

2.2 Vert.x

Eclipse Vert.x è un tool-kit per creare applicazioni reattive sulla JVM. Sviluppato inizialmente da Tim Fox nel 2012, Vert.x é ora gestito da Eclipse FOundation. Le prime iterazioni del progetto miravano a creare un "Node.js per la Java Virtual Machine", dopodiché gli sviluppatori hanno deciso di deviare il lavoro nello sviluppo di un tool-kit basato sulla programmazione asincrona specifico per la JVM. Vert.x non é considerato un framework poiché non fornisce un 'implementazione predefinita all'applicazione, ma si é liberi di utilizzarlo come una libreria all'interno del codice. Questo strumento non ha strumenti di compilazione consigliati, né pattern predefiniti per la stesura del codice, un'applicazione Vert.x é un insieme di moduli che forniscono solo le funzionalità necessarie e nulla di più. Non ha alcun tipo di dipendenze ad esempio con API di database o altro. Un progetto Vert.x é strutturato in moduli, in particolare per il nostro applicativo abbiamo utilizzato:

- `Vert.x core`: fornisce le API per la programmazione asincrona, non-blocking I/O, streaming e accessi ai protocolli network come TCP, UDP, DNS, HTTP, o WebSockets.
- `Vert.x web`: é un insieme di elementi utili alla creazione di applicazioni Web. Le principali feature sono : Routing, espressioni regolari, Request body handling, Event-bus bridge, supporto a SockJs.

Figura 2.2: Logo di Vert.x

2.3 Git

Git è un DVCS (Distributed Version Control System), cioè sistema software per il controllo di versione distribuito. Viene tipicamente usato per coordinare il lavoro sullo stesso software tra più sviluppatori, facendo in modo che sia possibile mantenere il codice sempre aggiornato nonostante venga modificato da più sviluppatori nello stesso

momento. Questo sistema permette di coordinare più facilmente il lavoro e in alcuni casi di consultare passate versioni del software. Inoltre è possibile consultare lo storico dei vari cambiamenti in modo rapido ed efficace grazie ai commit fatti dai membri del progetto, che consistono in una descrizione significativa delle modifiche compiute.

2.4 ReactJS

React è una libreria JavaScript per costruire interfacce utente caratterizzata dalla dichiaratività , efficienza e flessibilità. Grazie a React è possibile costruire interfacce complesse a partire da piccoli pezzi di codice chiamati componenti. Attraverso un componente andiamo a definire quella che risulta essere a tutti gli effetti una classe. Il nostro componente, definito da uno o più costruttori, contiene uno stato; questo stato verrà mantenuto, ed eventualmente aggiornato, durante tutto il ciclo di vita (lifecycle) del componente. È possibile ricevere dati ed istruzioni da altri componenti attraverso le props. Per integrare codice HTML dentro i nostri componenti React usa una sintassi speciale chiamata JSX che rende queste strutture più facili da scrivere.

2.5 SockJS

Vert.x-Web viene fornito con un gestore socket SockJS integrato, chiamato event-bus bridge. Estende efficacemente il bus degli eventi Vert.x sia lato server che lato client Javascript. Questo crea un bus di eventi distribuito che si estende su più istanze di Vert.x. Viene fornita una semplice libreria javascript chiamata "vertx-eventbus.js" che consente di inviare e pubblicare messaggi sul bus di eventi e registrare i gestori per ricevere messaggi. La libreria è disponibile su NPM, quindi può essere facilmente utilizzata con bundler o strumenti di compilazione.

2.6 Junit

JUnit è un framework di unit testing per il linguaggio di programmazione Java. JUnit è stato creato da Kent Beck insieme ad Erich Gamma. Da allora ha ispirato ed è stato modello guida per diversi framework di unit testing per altri linguaggi. Il framework è attualmente alla versione 5, che è organizzata in 3 sotto-progetti / moduli e necessita di java versione 8 o più recente. La versione 4 ha portato modifiche strutturali rispetto alla versione 3, con la quale è incompatibile. Le classi che costituiscono il framework appartengono a package diversi per le versioni 3 e 4; junit.framework fino a 3.8, org.junit dalla 4. Esempio di utilizzo di Junit 5

Figura 2.3: Logo di JUnit

2.7 MongoDB

MongoDB è un potente sistema di database open source e gratuito non relazionale, popolare per l'archiviazione di alti volumi di dati. È stato rilasciato 12 anni fa nel 2009 da 10gen (ora MongoDB Inc.) con una Server Side Public License. È un programma di database NoSQL scritto in C++, Python e JavaScript con compatibilità multipiattaforma. Supporta sistemi operativi come Windows, macOS e Linux e linguaggi come C, PHP, Java, Ruby, Node.js e altri. MongoDB differisce dai sistemi di database tradizionali in termini di come i dati vengono memorizzati. Invece di memorizzare i dati in righe e colonne, MongoDB adotta un design orientato ai documenti che rappresenta i dati in vari documenti e collezioni simili a JSON. Questi documenti contengono una serie di coppie di valori o chiavi di diversi tipi, come documenti annidati e array. Le coppie chiave/valore possono essere strutturate diversamente da un documento all'altro. MongoDB offre maggiore sicurezza, affidabilità ed efficienza oltre alla flessibilità di modificare la struttura o lo schema dei dati. In cambio si ottengono requisiti superiori in termini di velocità e archiviazione.

Figura 2.4: Logo di MongoDB

2.8 Npm

Npm é il più grande repository di software al mondo. Sia sviluppatori open source che organizzazioni private utilizzano npm per gestire i loro progetti software Npm consiste in tre diversi componenti:

- `Il sito web`
- `CLI (Command Line Interface)`: serve ad installare pacchetti, inizializzare ed avviare applicazioni
- `Registro`: un database contenente i vari pacchetti Javascript

Capitolo 3

Design

3.1 Struttura

La struttura del server é basata sulla classica modalità client-server. I client si connettono ad un server centrale, che é il fulcro di tutta l'architettura, in caso di malfunzionamenti da parte di quest'ultimo non é possibile usufruire del servizio. In particolare è stato adattato un approccio di tipo stateful in quanto risponde alle richieste dei client in modo coerente dipendentemente dallo stato attuale in cui si trova. Questa scelta é dovuta al fatto che nel gioco, il server deve essere l'unica entità a controllarne lo stato e lo svolgimento così da impedire possibili brogli.

3.1.1 Diagramma delle classi

Il nostro sistema attraverso la classe `LobbyVerticle` permette all'utente di creare o unirsi ad un game. Per tenere traccia dei vari game si utilizza come supporto la classe `LobbyManager`. Un game è rappresentato dalla classe `GameVerticle`. In un game sono presenti da 1 ad un massimo di 5 player. Il `Player` che ha creato la lobby diventa l'Admin del game ed é l'unico in grado di far partire il gioco. Inoltre un'istanza di `GameVerticle` mantiene tutti i dati necessari per la gestione della partita. Per ogni partita il `GameVerticle` richiede a `SpotifyApiVerticle` le canzoni (tante quanto il numero di turni) che poi verranno sottomesse ai player sottoforma di indovinello. Ogni partita ha un numero di turni prestabilito all'inizio. La logica del gioco viene gestita attraverso GameController, che possiede l'oggetto della classe Turn inizializzato ad ogni round. Quest'oggetto tiene traccia del numero di giocatori all'interno del turno corrente, i secondi impiegati per ogni risposta e il countdown del turno. Alla fine della partita verrà generato un report con la classifica e verrà inviato al database. I player hanno poi la possibilità di creare un account ed accedervi per visionare lo storico delle proprie partite.

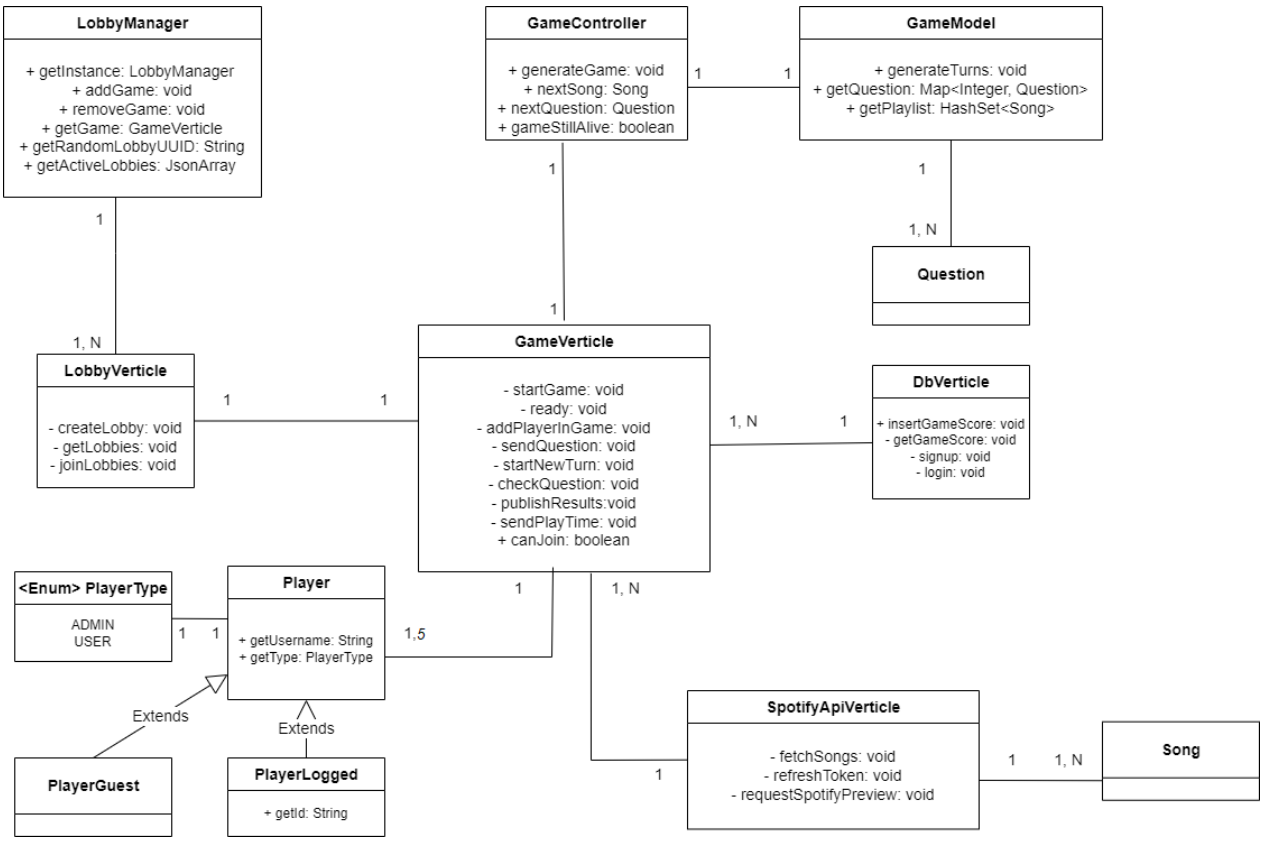

Figura 3.1: Design UML del sistema

3.2 Behaviour

Di seguito verranno descritte tutte le fasi del nostro programma.

3.2.1 Creazione della lobby

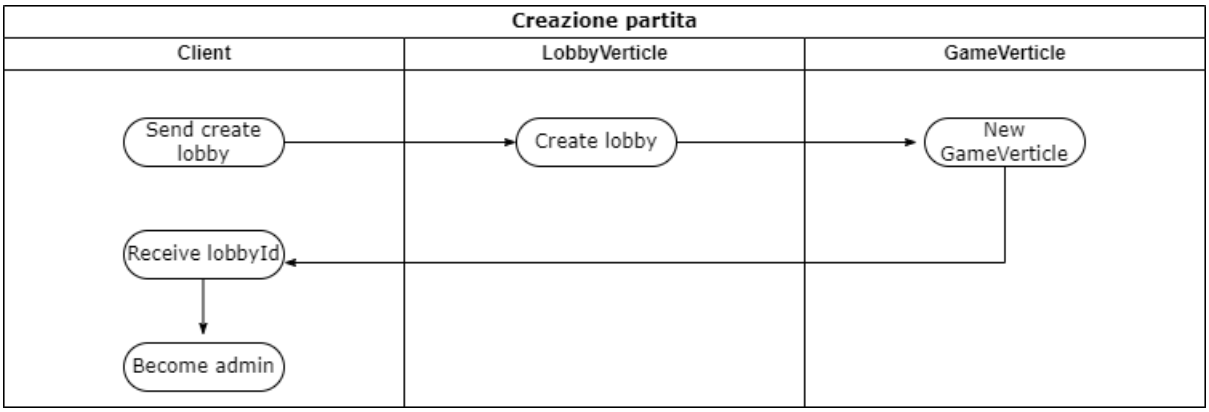

Figura 3.2: Diagramma di attività della creazione di una partita

In Figura 3.2 viene presentato un activity diagram che modella la creazione di una partita. Inizialmente il `Client` invia al `LobbyVerticle` la richiesta per creare una lobby. Il verticle inizializza un nuovo `GameVerticle` e invia al `Client` l'identificatore della lobby. Il `Client` diventa l'admin della lobby.

3.2.2 Ingresso nella lobby

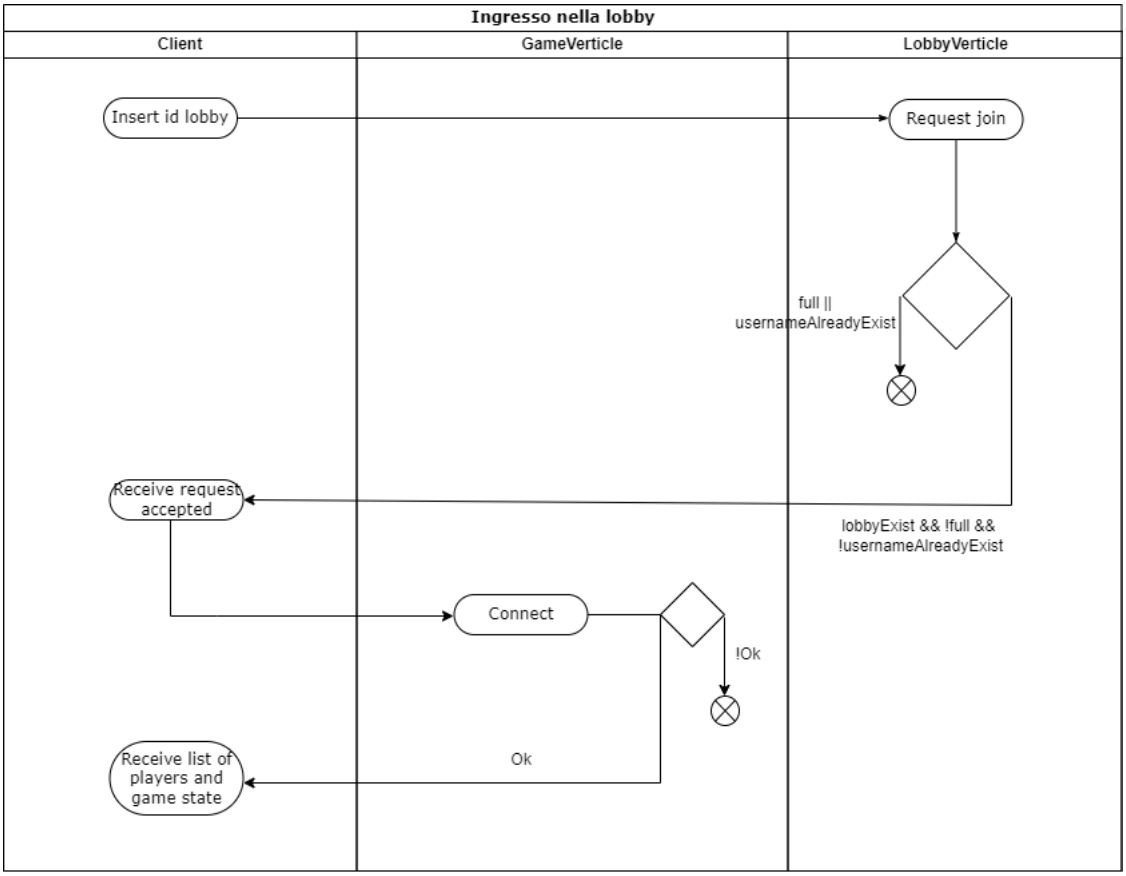

Figura 3.3: Diagramma di attività dell'ingresso in una lobby

In Figura 3.3, é mostrato l'activity diagram dell'ingresso nella lobby da parte di un utente. Il `Client` conoscendo l'id della lobby invia al `LobbyVerticle` la richiesta per accedervi. Il `LobbyVerticle` controlla se la lobby esiste e non é piena, in caso affermativo verifica se non é presente un utente con lo stesso username. Se le condizioni vengono soddisfatte il `LobbyVerticle` comunica al `Client` la possibilità di accedere alla lobby, il `Client` richiede al `GameVerticle` di essere aggiunto, e riceve le informazioni sulla lobby da parte del `Server` o in caso di partita in corso, lo stato del gioco.

3.2.3 Inizio partita

In Figura 3.4 viene presentato l'activity diagram dell'inizio di una partita. L'unico `Client` in grado di poter avviare un match é l'admin. Quest'ultimo notifica il `GameVerticle`, che a sua volta comunica a tutti i `Client` presenti nella lobby che la partita sta iniziando, a loro volta ricevuta tale notifica i `Client` uno ad uno confermano

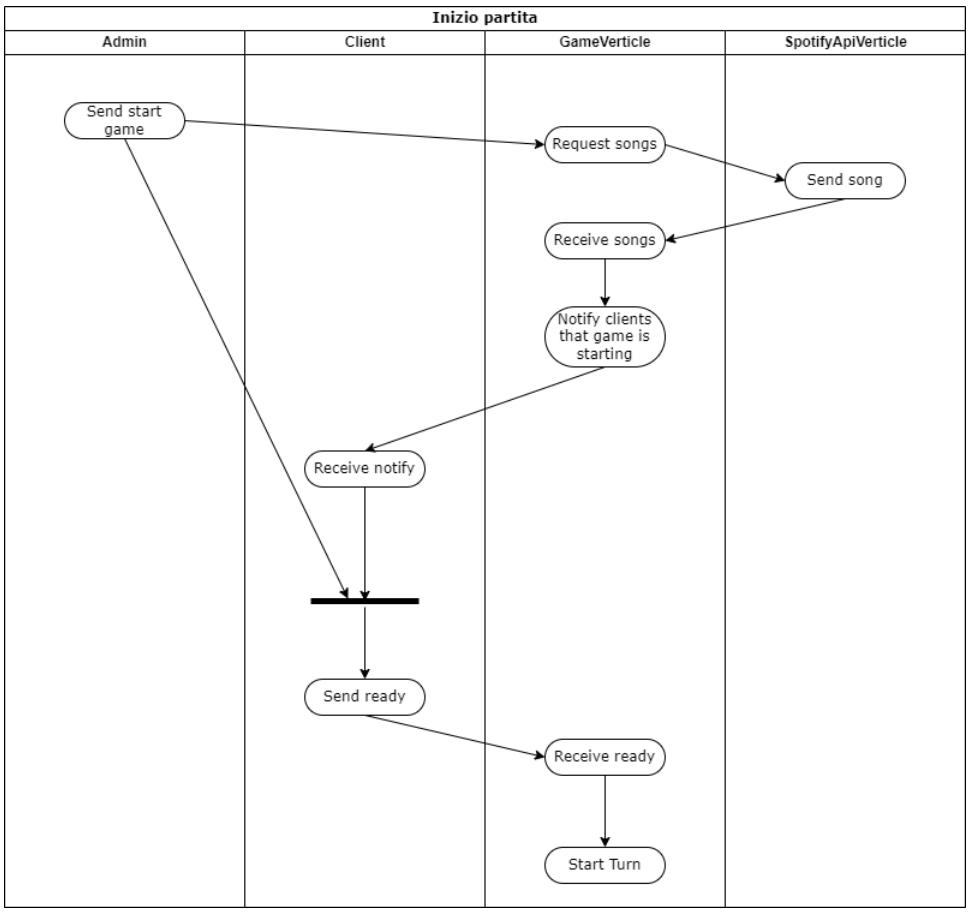

Figura 3.4: Diagramma di attività dell'inizio di una partita

di essere pronti a ricevere le domande. Il `GameVerticle` ricevuto un numero di adesioni pari agli utenti in gioco pubblica i dati del primo round.

3.2.4 Svolgimento di un round

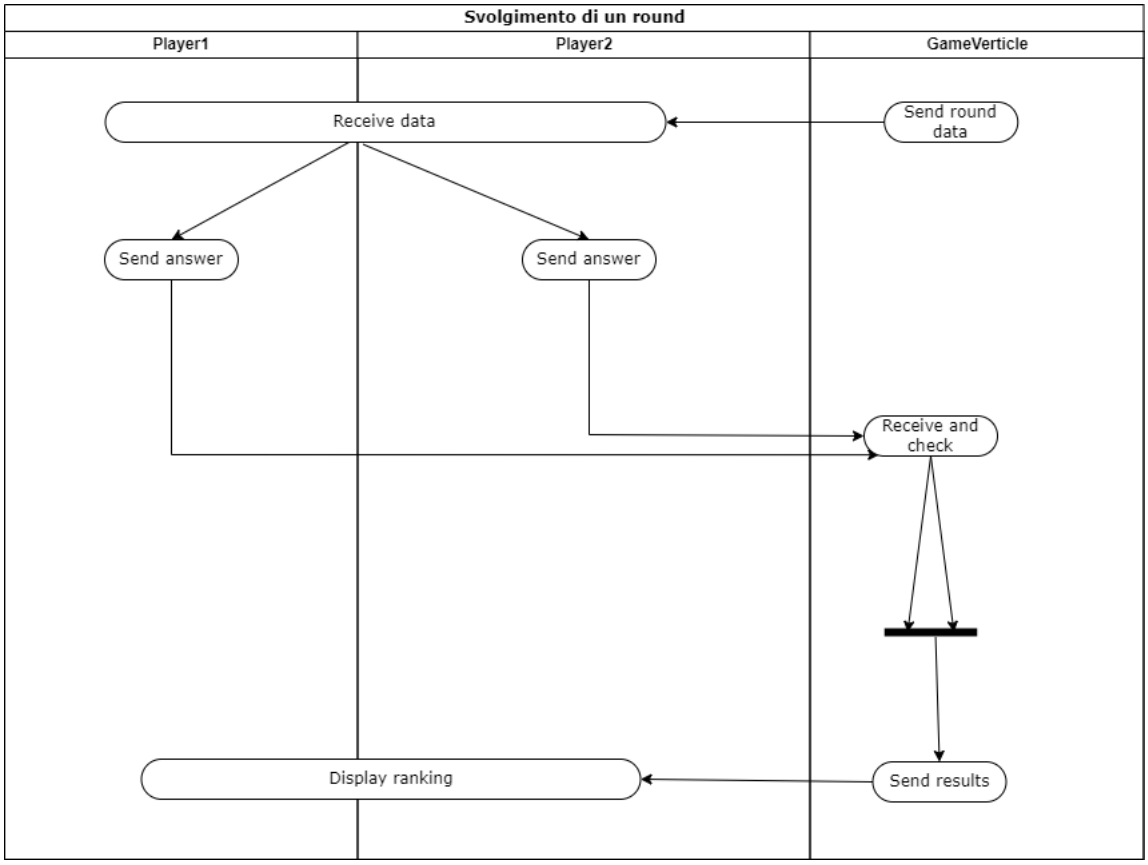

Figura 3.5: Diagramma di attività dello svolgimento di un round

La Figura 3.5 va a descrivere il flusso completo delle attività che avviene durante un round. All'inizio del turno il `GameVerticle` invia le domande e i 30 secondi di canzone da ascoltare. Il `Client` ha a disposizione 30 secondi per ascoltare e inviare al server la risposta (titolo della canzone). Il `Server` ad ogni risposta ne verifica la correttezza. Per poter terminare il round il `GameVerticle` necessita che ogni `Client` abbia risposto oppure che il tempo si sia esaurito. Al termine del round, il server pubblica i risultati ed ogni `Client` mostra a schermo la scoreboard parziale.

3.3 Interaction

Per la prima interazione, il `Client` manderà una richiesta HTTP Post al `Server`, richiedendo di creare una nuova lobby, oppure di entrare in nuova già esistente. Se il `Client` ha richiesto la creazione, il `Server` risponderà con il numero di porta della lobby corrispondente. Viceversa se viene richiesto l'accesso da parte di un utente, verrà controllato se non esiste un altro giocatore con lo stesso username, in caso affermativo l'utente viene aggiunto al gioco.

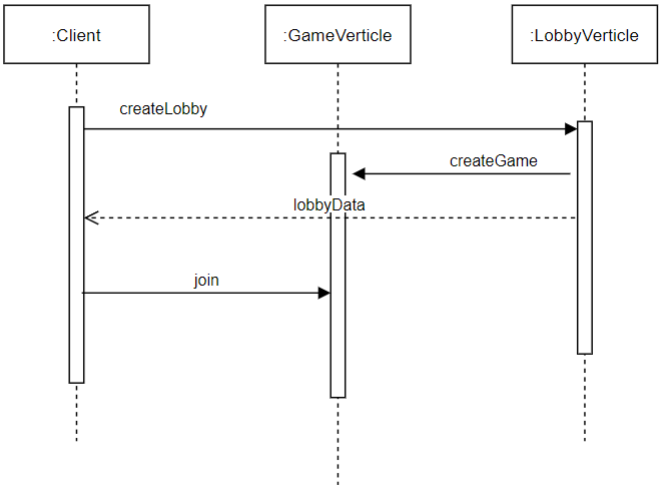

Figura 3.6: Diagramma di sequenza che mostra i messaggi scambiati tra un utente e il server durante la creazione della lobby

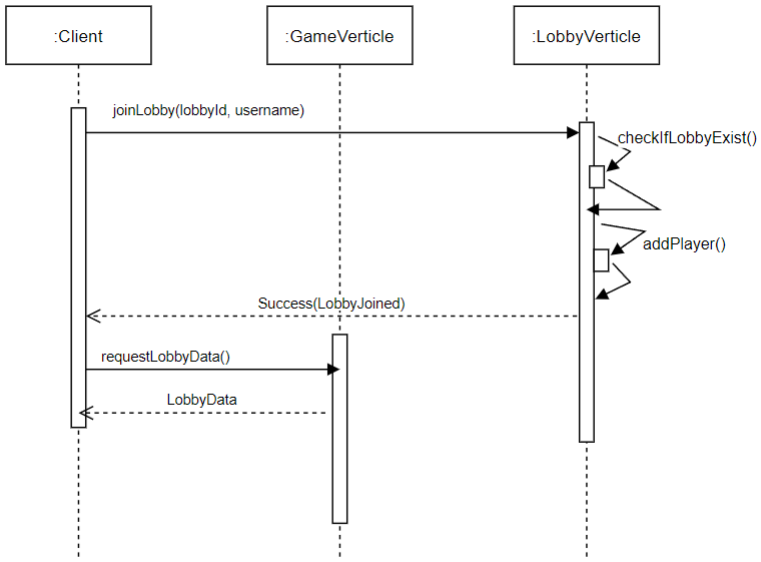

Figura 3.7: Diagramma di sequenza che mostra i messaggi scambiati tra un utente e il server durante l'accesso ad una lobby

Una volta che un utente ha avuto accesso alla lobby verrà istanziata una websocket per lo scambio dei messaggi tra client e server in modalità full-duplex. Subito dopo l'accesso il client comunicherà costantemente con il Server, attraverso l'utilizzo di un ping implementato tramite websockets. Il ping é necessario per verificare se un utente ha perso la connessione con il Server, in questo gli altri Client verranno avvisati tramite un messaggio broadcast. Per i messaggi è stato scelto il formato JSON e decidendo un modo formale di leggere e scrivere i dati. Di seguito i messaggi scambiati fra le due parti per l'operazione di ping:

SERVER

```
{
    body:{
        type: "ping",
        username: "mario.rossi"
    }
}
```

CLIENT

```
{
    headers: {
        status: "SUCCESS"
    },
     body:{
        type: "ping"
    }
}
```

In Figura 3.8, vengono mostrati i messaggi scambiati durante lo svolgimento del gioco tra Client e Server.

L'admin invia al GameVerticle l'intenzione di iniziare la partita e tutti gli altri giocatori vengono notificati. A questo punto ogni utente richiede tramite *ready* la domanda e l'url del turno, il Server raccolte tante richieste quante il numero di utenti nel gioco pubblica le domande. Esempio di richiesta e risposta delle domande:

CLIENT

```
Indirizzo eventbus: ready.lobbyID
{
     body:{
        username: "mario.rossi"
    }
}
```

SERVER

```
Indirizzo eventbus: question.lobbyID.mario.rossi
{
    body:{
       question: "Michael Jackson",
       stream: "https://open.spotify.com/preview/5ChkMS8OtdzJe"
    }
}
```

Il Client invia poi la risposta e attende la fine del turno. L'utente ha anche la possibilità di richiedere un aiuto in gioco, mostriamo un esempio dove vengono richieste le prime tre lettere della risposta:

CLIENT

```
Indirizzo eventbus: suggestion.lobbyID
{
```

```
    body:{
        username: "mario.rossi",
        suggestion: "letters"
    }
}

SERVER
{
    headers:{
        status: "SUCCESS",
    },
    body:{
      letters: "Bil"
    }
}
```

Alla fine di ogni turno il Server pubblica i risultati a cui ogni utente può accedere, un JSON contente la risposta corretta, se un utente ha indovinato, i punteggi parziali del turno, i punteggi totali accumulati, i secondi per rispondere e il tempo totale accumulato per rispondere (*sendResults*).

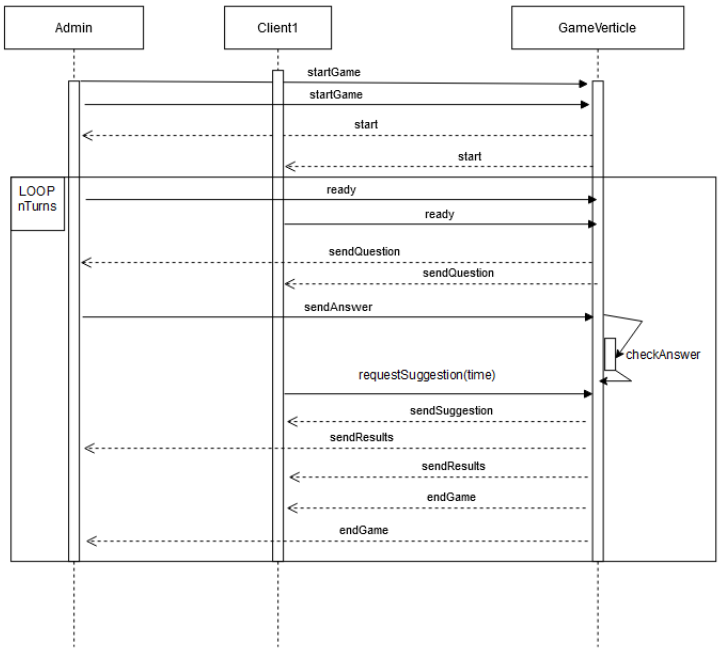

Figura 3.8: Diagramma di sequenza che mostra i messaggi scambiati tra utenti e server durante lo svolgimento di una partita

3.3.1 Sincronismo e sicurezza

Il focus principale di questo progetto é sincronizzare gli utenti con il server affinché abbiano la stessa quantità di tempo per rispondere durante i vari round del gioco. Il controllo del flusso di gioco é nelle mani del server, esso mantiene per ogni utente il tempo a disposizione per rispondere. E' stata scelta quest'implementazione, piuttosto che avere un tempo unico per ogni giocatore, a causa degli aiuti in gioco. Infatti un utente ha la possibilità di richiedere un tempo extra, che si andrà a sommare al tempo rimanente del turno, per evitare possibili imbrogli questa parte é completamente gestita dal server. Il server controlla che tutti gli utenti abbiano risposto o in caso contrario che il tempo rimanente per l'utente o gli utenti che hanno richiesto il tempo extra sia scaduto, in caso affermativo termina il round. Un'altra funzionalità implementata é quella di permettere ad un utente di aggiungersi ad una partita già iniziata, e sincronizzarsi con il tempo di gioco. Un utente può effettuare il join in qualsiasi istante del gioco, verrà calcolato un punteggio pari a 0 e un tempo di risposta pari a 30 per i turni precedenti.

3.4 Parte opzionale

3.4.1 Struttura database

Il database deve essere in grado di memorizzare gli utenti che si registrano al sistema e mantenere in memoria i dati delle partite passate. Come DBMS si è optato per MongoDB, un database di tipo non relazione orientato ai documenti. La struttura comprende un documento Users che mantiene al suo interno la collezione di tutti gli utenti registrati al gioco. In particolare tale documento contiene l'id, l'username, la password e la lista delle partite giocate.

Un documento Games che rappresenta la collezione di tutte le partite effettuate nel gioco, mantiene l'id, la data in cui si è svolta la partita e la lista dei giocatori partecipanti con i relativi punteggi.

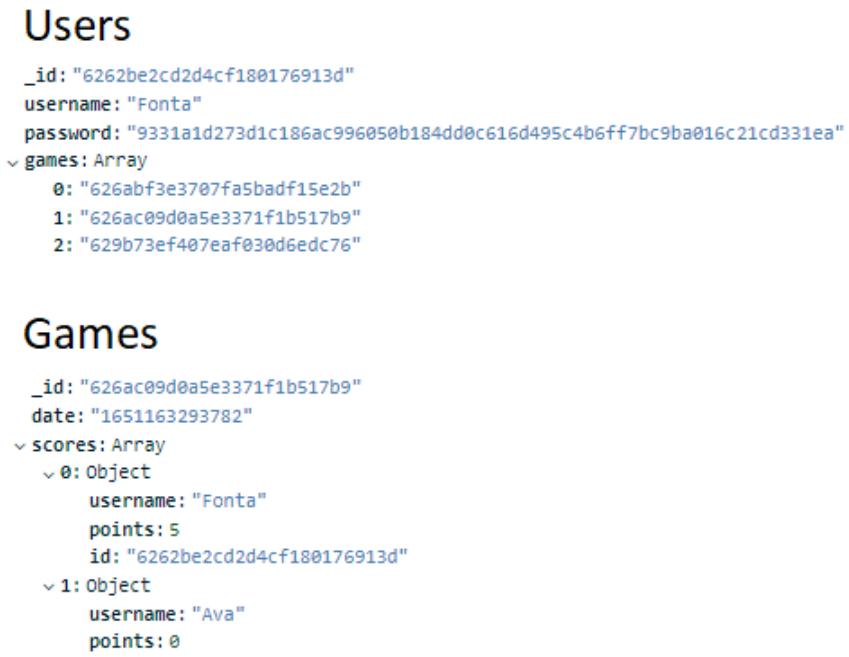

Figura 3.9: Esempio di documenti del database

3.4.2 Autenticazione utenti

Un utente ha la possibilità di accedere al gioco sia come ospite che come utente registrato. La registrazione permette di associare lo storico delle partite ad un determinato e univoco username. Per mantenere al sicuro le password degli utenti é stato utilizzato un algoritmo di cifratura chiamato Bcrypt. Bcrypt utilizza una funzione di hashing basata su un algoritmo a chiave simmetrica chiamato Blowfish.

3.4.3 Aiuti in gioco

Sono stati implementati due tipi di aiuti: richiesta di tempo addizionale (10 secondi) oppure una sotto stringa della risposta, in genere le prime tre lettere della parola. Durante lo svolgimento di un round il player ha la possibilità di richiedere un solo aiuto, per tutta la durata della partita può utilizzarne al massimo 3. Il tempo addizionale é sicuramente il più complicato da implementare in quanto ogni utente nella partita può richiederlo in istanti di tempo differenti. Per l'implementazione dunque si é scelto di assegnare ad ogni player un proprio tempo di gioco, che viene incrementato ogni qualvolta venga richiesto tale aiuto. Se anche solo un utente richiede l'aiuto il tempo complessivo del round viene portato da 30 a 40 secondi ed é necessario attendere la fine in caso quest'ultimo non dia la risposta.

3.4.4 Chat

Come ultima feature opzionale é stata implementata una chat di gioco. Ogni lobby ha una propria chat condivisa con gli utenti in partita. Se un utente accede al gioco a partita in corso la chat non recupererà i messaggi scambiati in precedenza.

3.5 Testing

Per rendere la struttura dell'applicazione più solida abbiamo costruito dei test, utilizzando il framework Junit, progressivamente durante lo sviluppo del progetto. Inoltre abbiamo testato il sistema con l'ausilio di vari client di prova per verificare il funzionamento e l'interazione con il sistema. Non è stato però possibile testare il comportamento del sistema nel caso in cui il numero di client che interagiscono contemporaneamente con il sistema sia elevato.

3.5.1 DatabaseVerticle Test

Si tratta di test relativi al servizio di autenticazione nel gioco. Per prima cosa si verifica se un utente può registrarsi al sistema con due tipi di test, signupFailed e signupSuccess. Nel primo caso un utente prova a registrarsi al sistema immettendo un username già presente, ci si aspetta una risposta con status code 409 per far sì che il test sia passato. Per il signupSuccess invece si verifica che la risposta restituisce uno status code 200.

Anche per il login abbiamo due test, loginSuccess e loginFailed. Nel primo caso si crea un oggetto json con nome utente e password presenti nel database e si verifica che la risposta restituisce uno status code 200. Nel secondo caso si crea un oggetto json con username presente nel database ma con password errata e si verifica che la risposta del server restituisce uno status code 401 Unauthorized.

3.5.2 SpotifyApiVerticle Test

I test sono relativi ad un servizio che ha il compito di comunicare tramite API con Spotify. La funzione di SpotifyAPiVerticle é quella di scaricare una playlist di preview di canzoni e renderle disponibili al server. L'unico test effettuato riguarda appunto questa funzione, testFetchSongs. Essendo un metodo asincrono all'interno di SpotifyApiVerticle il test ha bisogno di una promise, che verrà completata ad operazioni ultimate. Il test verifica attraverso l'handler della promise se l'operazione ha avuto successo, in caso affermativo il test é passato.

3.5.3 LobbyVerticle Test

Questa classe di test ha il compito di verificare la creazione di una lobby all'interno del sistema. Innanzitutto si crea una richiesta http di creazione della lobby che verrà inoltrata al LobbyVerticle. Successivamente si attende uno status code di 200 che la richiesta è stata eseguita correttamente. Un altro test che viene eseguito è il join della lobby. Per prima cosa si crea una nuova lobby. La risposta del server fornirà l'id della lobby creata che verrà usato in una nuova richiesta da parte di un nuovo utente per entrare nella lobby. Il server risponde con un messaggio di riuscita o no nell'operazione di join, in caso positivo la risposta contiene lo status code di 200 quindi il test è passato.

3.5.4 GameVerticle Test

- `deployVerticle`: in questo test si verifica innanzitutto che il deployment di SpotifyApiVerticle abbia successo, in quanto operazione necessaria per l'avvio del

server, attraverso l'utilizzo di una promise. Se l'operazione ha successo si passa ad aggiungere i player al gioco, nel test ne vengono aggiunti 2. Uno é l'admin della partita che simula una richiesta tramite eventbus al server per iniziare la partita. Se la risposta contiene nell'header il campo "STATUS" con valore "SUCCESS" il test ha successo.

- `testReceivedData`: in questo test viene verificato che le domande arrivino correttamente al client. Per prima cosa é necessario che i due giocatori comunichino al server di essere pronti per la ricezione, dopodichè attraverso l'eventbus viene inviato un JsonObject con due campi "question" contenente il nome dell'artista e "stream" contenente l'url per riprodurre la preview. Il test verifica che appunto siano presenti questi due campi, in caso affermativo il test é passato.

- `testCheckAnswer`: la struttura del test é simile al deployVerticle, viene simulato l'invio di una risposta tramite eventbus e si aspetta dal server un JsonObject con status "SUCCESS". In caso affermativo il test é passato.

- `testCheckSuggestion`: viene simulata una richiesta di un aiuto in gioco da parte del client. La richiesta contiene un JsonObject con all'interno un campo "suggestion" che indica il tipo di aiuto richiesto, può essere di tipo "letter" o "time", se il server risponde con status "SUCCESS" il test ha esito positivo

- `testTwiceSuggestionsInTurn`: vengono fatte due richieste da parte dei due utenti in gioco, uno per l'aiuto "letter" l'altro per "time".

3.6 Deployment

Il primo passo è clonare il repository al seguente indirizzo: https://gitlab.com/pika-lab/courses/ds/projects/ds-project-avagnano-fontanella-ay2122.git.

3.6.1 Database

Per avviare un' istanza del database MongoDB, bisogna per prima cosa posizionarsi nella cartella dove è presente il file di configurazione di docker

```
cd mongo-docker
```

Assicurarsi che nessun servizio sia in esecuzione sulla porta 27017 e in seguito eseguire il comando

```
docker-compose up
```

3.6.2 Server

Grazie all'utilizzo di Gradle il deployment del server risulta automatizzato e intuitivo. È sufficiente che la macchina sulla quale si vuole eseguire il server abbia installato un JRE con Java dalla versione 11 in poi. Succevviamente bisogna posizionarsi all'interno della cartella del server con il comando:

```
cd guessthesong-server
```

Per lanciare il server usare il comando:

```
./gradlew run
```

3.6.3 Client

Per quanto riguarda il lato client sarà sufficiente l'installazione di npm, che provvederà autonomamente all'import delle dipendenze necessarie e al setup dell'ambiente relativo a Node.js. Per prima cosa bisogna posizionarsi all'interno della cartella del client con:

```
cd guessthesong-client
```

Successivamente si va a importare tutte e librerie utilizzate dal client con:

```
npm install
```

Infine per lanciare il client usare il comando:

```
npm start
```

3.7 Esempi D'uso

Dalla homepage dell'applicativo web l'utente può effettuare il login o registrarsi per tenere traccia dello storico delle proprie partite. Il menu permette la creazione di una nuova lobby, come mostrato in Figura 3.11, scegliendo il numero di turni della partita, il numero di giocatori e il genere di playlist di spotify tra quelle messe in evidenza. La figura 3.12 mostra i dati della lobby dal punto di vista dell'admin: i giocatori che sono entrati, l'id e il bottone che permette di iniziare la partita. La schermata di gioco é mostrata in Figura 3.13, sulla sinistra sono presenti la textbox per rispondere, la barra di riproduzione della preview, con la quale é possibile riavvolgere la canzone e i due aiuti in gioco, i 30 secondi extra e le prime 3 lettere del titolo della canzone. Mentre a destra è presente la chat di gioco. A termine di ogni turno, come mostrato in Figura 3.14, vi é la scoreboard parziale, con i punti accumulati da ogni utente fino a quel turno, i secondi impiegati per rispondere, le risposte date e il titolo corretto della canzone. Infine la Figura 3.15, mostra i punteggi finali e il vincitore, quest'ultima schermata permette di tornare al menu iniziale.

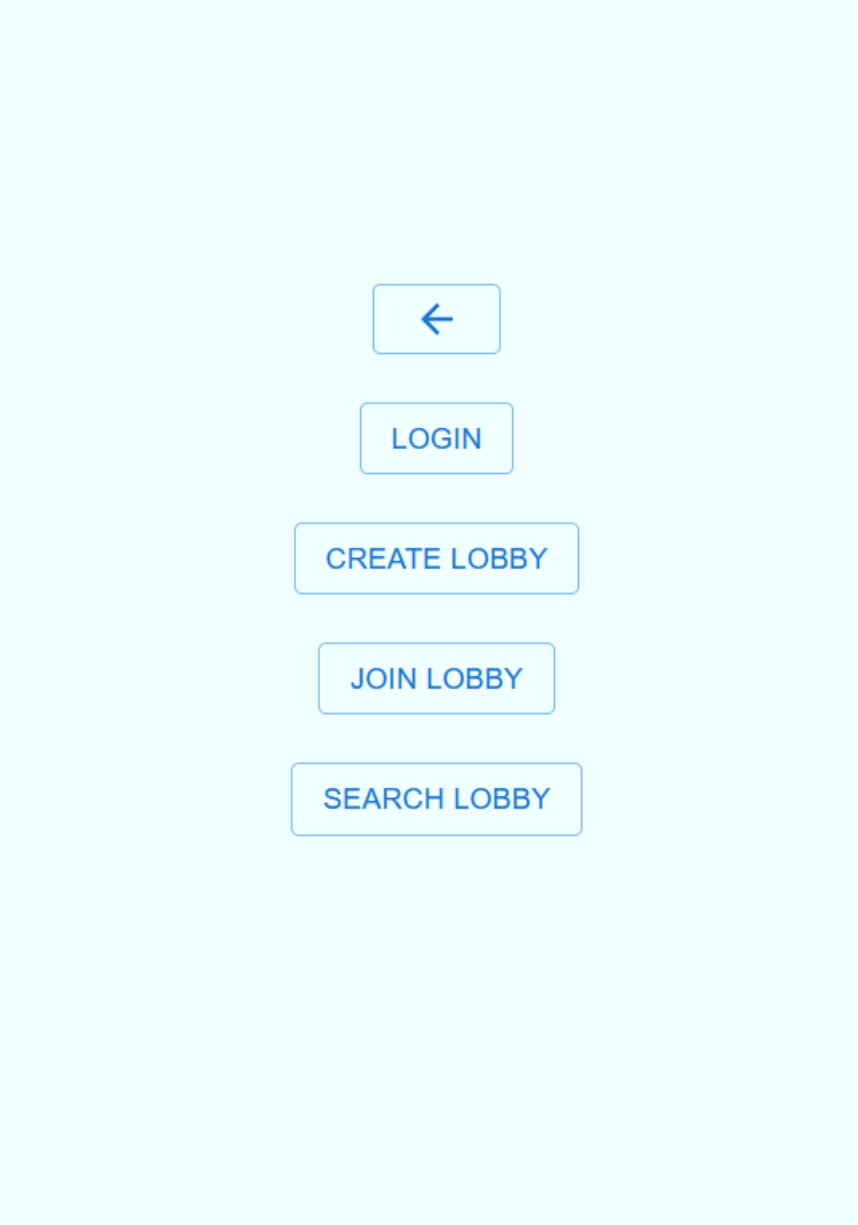

Figura 3.10: Home page del gioco

Figura 3.11: Menu di creazione della lobby

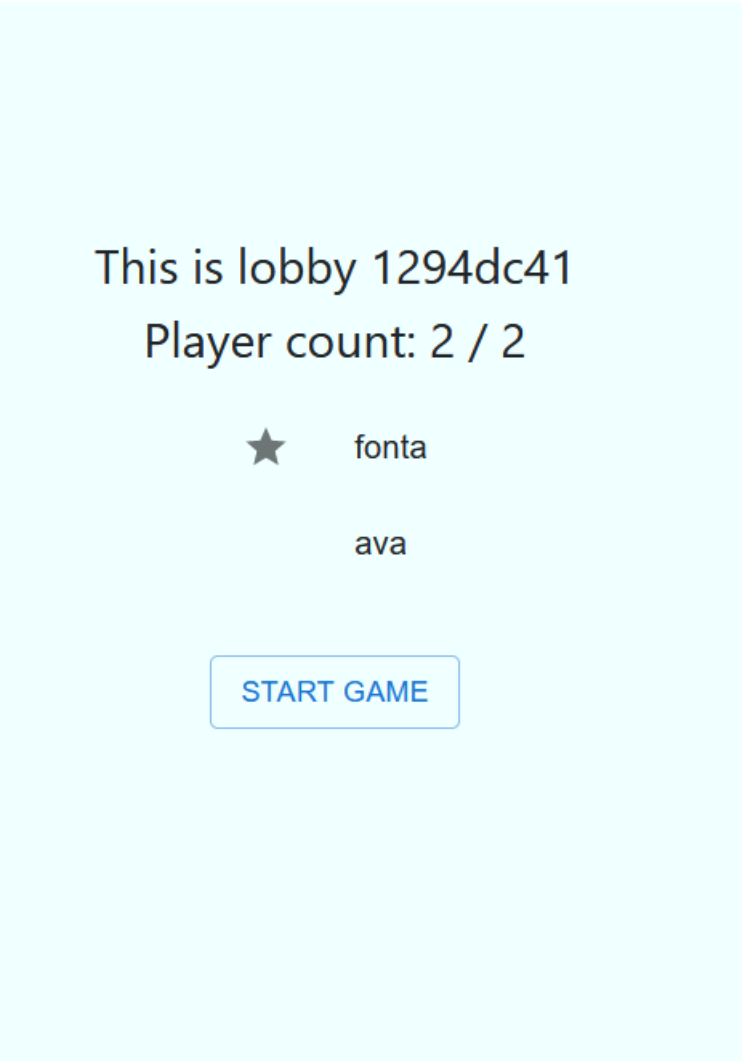

Figura 3.12: Schermata della lobby e avvio del gioco

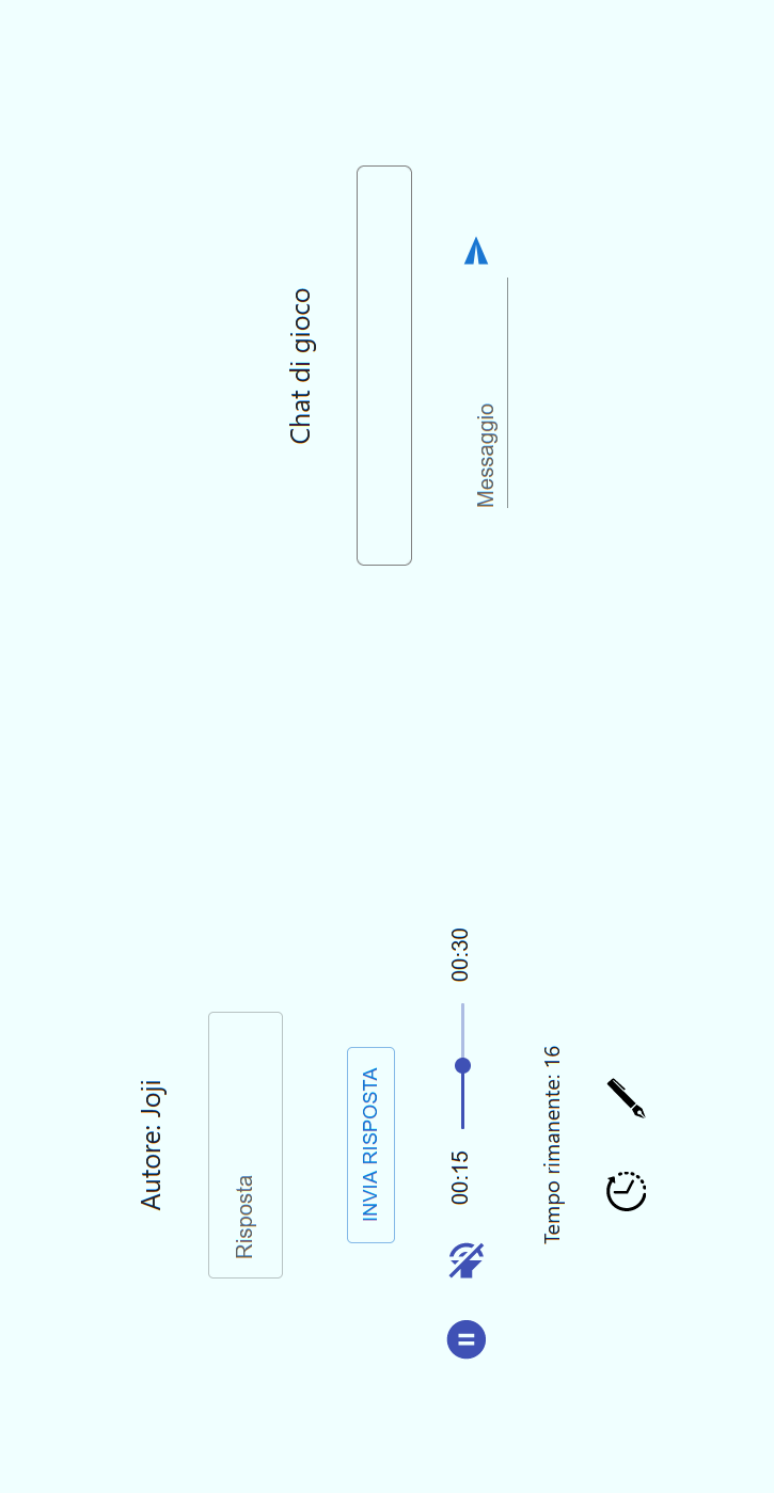

Figura 3.13: Schermata di gioco e chat

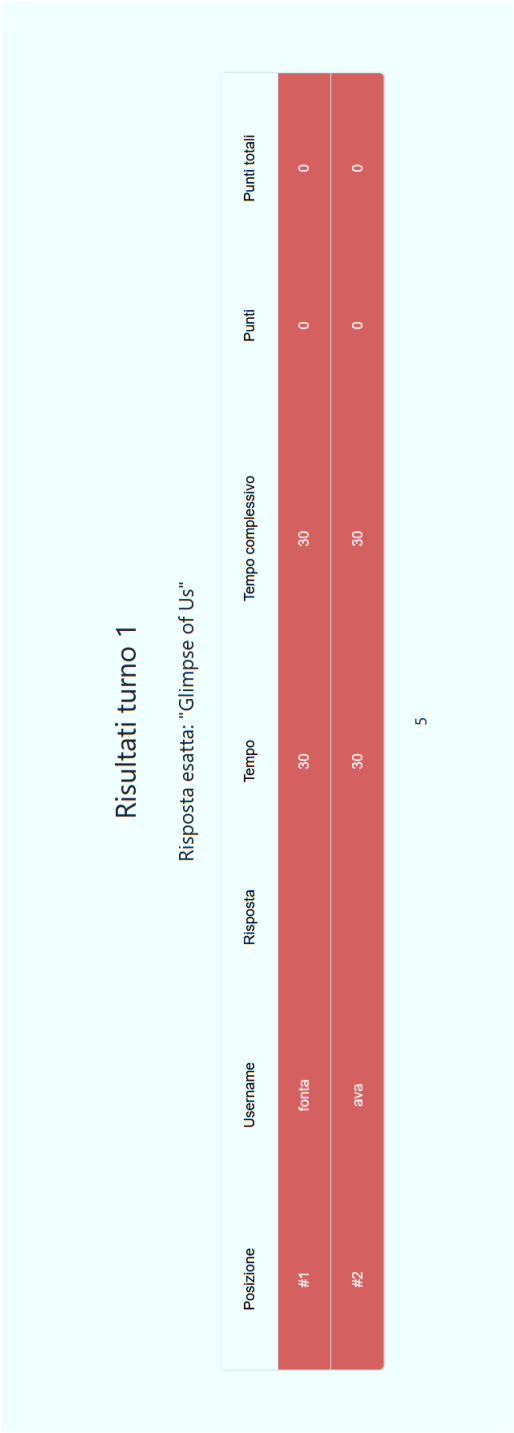

Figura 3.14: Schermata dei risultati parziali

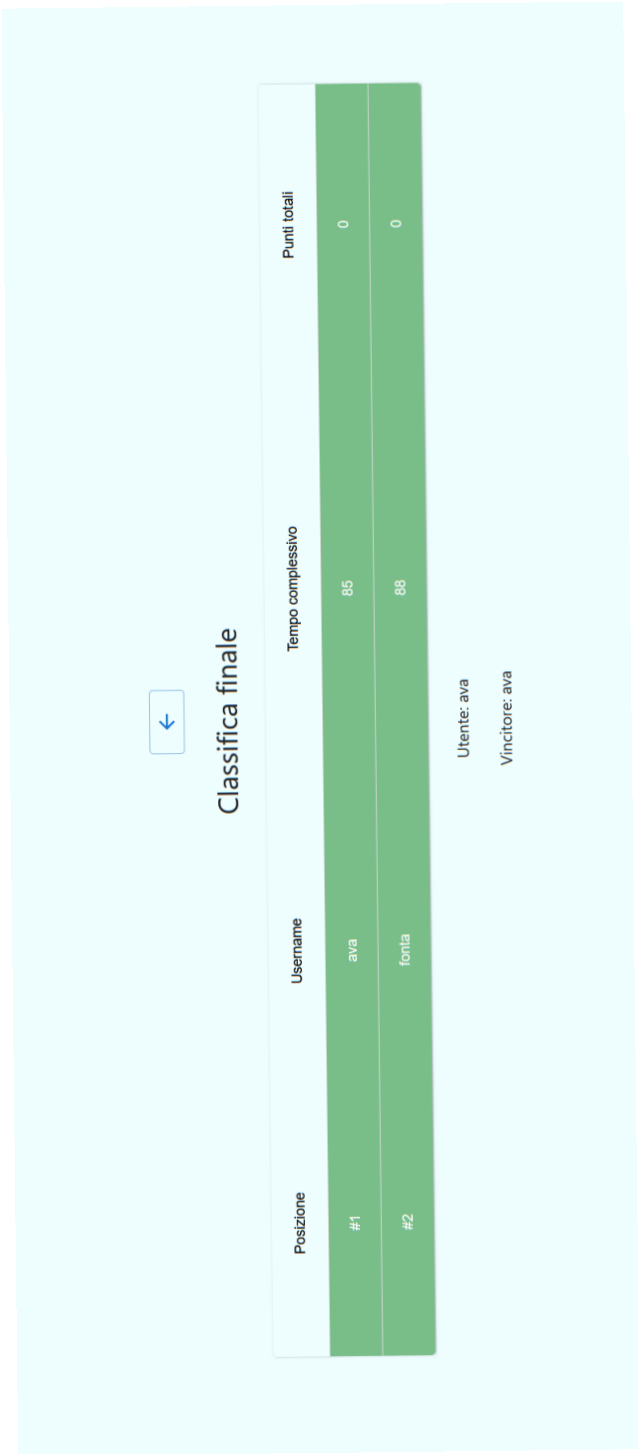

Figura 3.15: Classifica finale e vincitore

3.8 Conclusioni

Il progetto ha richiesto una quantità di tempo superiore alle aspettative per essere ultimato, é stato impiegata una quantità di tempo considerevole per familiarizzare con le nuove tecnologie quali Vert.x, React e MongoDb. Abbiamo ottenuto un risultato soddisfacente, il gioco risulta reattivo, coinvolgente e divertente da giocare. Al di fuori dell'elaborato é stato un piacere dedicarsi a questo progetto, in quanto fan del programma televisivo Sarabanda, a cui il gioco é ispirato. Di seguito riportiamo il piano di lavoro svolto. In primo luogo abbiamo stabilito i requisiti del sistema, questa prima fase é stata accompagnata da una discussione sugli aspetti logici del gioco, sul tipo di architettura che il sistema doveva rispettare e la gestione delle risorse da parte del server. Stabiliti i requisiti fondamentali e opzionali, e scelta l'architettura, in comune accordo abbiamo tracciato un diagramma di sequenza che ci ha permesso di comprendere al meglio le dinamiche di interazione fra client e server. Dopodiché abbiamo concordato le parti da sviluppare e ci siamo concentrati sull'implementazione di una versione base del progetto. Terminata la prima versione sono stati effettuati dei test in locale, per verificare che tutte le componenti comunicassero in maniera corretta e che la logica del gioco corrispondesse a quanto concordato. Ci siamo successivamente concentrati nell'aggiunta di funzionalità opzionali, quali chat di gioco o aiuti in partita, e come prova finale abbiamo testato il gioco sulla stessa rete locale con 5 utenti.

3.8.1 Sviluppi Futuri

La struttura del nostro progetto permette l'aggiunta di nuove funzionalità nel gioco, come ad esempio nuovi aiuti in gioco oppure aggiunta di nuove playlist.

Un altro sviluppo riguarda la migliore gestione della grafica generale e dell'interfaccia utente del sistema, con l'integrazione di elementi di gamification (avatar per gli utenti, punti che si guadagnano con le partite, playlist sbloccabili).

Una parte da sviluppare ulteriormente è la gestione dell'utente nel sistema. L'utente non ha una pagina dedicata, non ha la possibilità di inserire informazioni riguardanti a sé e non esiste un sistema dedicato per inviare i codici delle lobby ad un amico se non usando applicazioni di terze parti.

L'ultimo sviluppo potrebbe riguardare il test del sistema da parte di centinaia/migliaia di utenti allo scopo di verificare che sotto stress il sistema funzioni correttamente e sia performante.

3.8.2 Cosa abbiamo imparato

Lo sviluppo di questo progetto ci ha fatto sicuramente comprendere meglio come funziona la programmazione distribuita e asincrona. In particolare l'utilizzo di Vert.x ci ha facilitato lo sviluppo di un server in grado di gestire richieste asincrone provenienti da più client, attraverso tecniche come Promise, Future, executeBlocking per operazioni computazionalmente dispendiose in termini di tempo e risorse. Da sottolineare l'importanza degli EventBus, fondamentali per la comunicazione tra client e server che ci

permettono un'elevata reattività. Lo svolgimento dell'elaborato ci ha permesso di comprendere infine, l'importanza dei test che ci hanno aiutato a verificare il funzionamento del sistema nelle varie fasi di realizzazione del progetto.