Huesle – report finale

Alberto Spadoni
alberto.spadoni3@studio.unibo.it

Giugno 2024

Il presente elaborato ha lo scopo di realizzare una versione digitale e distribuita del gioco da tavolo Mastermind [1].

Tale versione rappresenta una variante rispetto a quella originale e prevede che i due sfidanti abbiano lo stesso ruolo e che debbano competere per la risoluzione di una partita. Per farlo, devono a turni cercare di indovinare un codice composto da una sequenza di 4 colori sulla base dei suggerimenti forniti a partire dai tentativi già effettuati. Tale codice viene generato casualmente dal backend dell'applicazione.

La modalità di gioco risulta asincrona in quanto i due giocatori non devono per forza essere online e collegati alla partita nello stesso momento per far si che il gioco possa andare avanti. L'interazione tra di essi avviene per mezzo della rete Internet e attraverso i client di gioco, che sono eseguiti sui singoli dispositivi dei giocatori.

L'implementazione in questione consentirà di creare due tipologie di partite: una pubblica e una privata. In seguito alla creazione, una partita potrà iniziare solo quando entrambi i giocatori si saranno uniti ad essa. Da quel momento in poi, il gioco fornirà 10 tentativi totali (5 per utente) entro i quali dovrà essere indovinata la sequenza di colori. Vincerà la partita il primo che indovina la sequenza nelle mosse a sua disposizione. Al contrario, il match terminerà in pareggio se nessuno riesce a svelare il codice.

Oltre alle operazioni di gioco, l'applicazione permetterà agli utenti di registrarsi, autenticarsi, modificare alcune informazioni del profilo e visionare le partite in corso e quelle terminate, nonché lo stato attuale degli avversari.

1 Obiettivi

L'obiettivo principale di questo progetto consiste nel riprendere un precedente elaborato e riscriverne una parte, facendo uso di un diverso linguaggio di programmazione e impiegando nuove tecnologie.

Al sistema di partenza verrà riformulato completamente il lato backend, ponendo l'accento su tutti quegli aspetti che risultano fondamentali nel design di un sistema

distribuito solido. Inoltre, si prevede di renderlo indipendente dalle disconnessioni dei client, ma al tempo stesso di garantire loro consistenza al momento della riconnessione.

Un altro obiettivo che si mira a raggiungere è l'integrazione della build automation per tutte le componenti del sistema. Ciò consentirà di utilizzare strumenti specifici per la compilazione e l'esecuzione automatizzata dei codici sorgente. In particolare, il client utilizzerà *npm*, mentre l'intero backend sarà organizzato come un progetto *Gradle* [4].

Infine, per raggiungere appieno gli obiettivi appena descritti, è prevista una breve ma necessaria fase di revisione del frontend per garantire un perfetto adattamento con il nuovo artefatto.

1.1 Risultato atteso e deliverables

In questa sezione viene presentata la situazione iniziale, per poi descrivere brevemente come si intende strutturare il risultato finale.

Il contesto da cui nasce il presente elaborato ha come protagonista un sistema che implementa la versione digitale di Mastermind discussa in precedenza. Esso presenta un'architettura client-server, formata da tre microservizi:

1. **Client** scritto in JavaScript con il supporto del framework React sotto forma di Single-Page Application.
2. **Server** sviluppato in JavaScript ed eseguito su Node.js.
3. **Database** non relazionale MongoDB [9] per la memorizzazione degli utenti e dei dati di gioco.

Il risultato atteso prevede di mantenere l'architettura di partenza e di continuare ad orientarla ai microservizi. Il client rimarrà pressoché invariato, se non per alcuni piccoli adattamenti sul fronte della comunicazione che saranno necessari per assicurarne il corretto funzionamento con il nuovo backend. Il server, al contrario, verrà completamente riscritto in Java e suddiviso a sua volta in due microservizi:

- il primo gestirà gli utenti, le dinamiche di gioco e il database,
- il secondo fungerà da interfaccia tra i client ed il microservizio appena introdotto. In particolare, si occuperà di ricevere le richieste HTTP, di inoltrarle al microservizio appena citato e di provvedere a recapitare le relative risposte. Esso verrà implementato sotto forma di server HTTP con il supporto del framework Vertx Web [19].

L'API RESTful esposta dal backend seguirà la struttura di quella del sistema di partenza e la sua descrizione dettagliata, che segue la specifica OpenAPI, verrà discussa nella sezione 4.3.

Per quanto riguarda la comunicazione tra le componenti del sistema, si prevede di sfruttare le seguenti tecnologie:

- i due microservizi che compongono il backend interagiscono grazie a RabbitMQ, ovvero un Message-Oriented Middleware (MOM) che abilita lo scambio di messaggi attraverso la rete tra entità eterogenee [13];
- per quanto riguarda la comunicazione client-server, oltre al protocollo HTTP usato per l'API, verrà impiegato anche WebSocket [20], grazie alla libreria `SockJS` [17] integrata in Vertx Web. Questo permetterà una comunicazione bidirezionale in tempo reale tra il client e il server, migliorando l'efficienza e la reattività del sistema

A fine progetto, gli artefatti che verranno consegnati sono:

- codice sorgente del frontend e del backend,
- i Dockerfile e docker-compose.yml relativi al client ed al server per facilitare il deployment dell'intero sistema,
- i files per integrare la build automation in entrambi i sorgenti forniti: `package.json` lato frontend e files Gradle per la parte di backend.

1.2 Scenari

Nel contesto di questo progetto, è possibile considerare due scenari principali di interazione.

Scenario 1: Utente Accedendo all'applicazione, un utente che non si è ancora autenticato, può creare un account (fornendo email, nome utente e password) oppure consultare le regole di gioco. Una volta completata la registrazione, può effettuare il login per accedere a tutte le funzionalità dell'applicazione. Questo processo di registrazione e autenticazione è fondamentale per garantire un'esperienza di gioco personalizzata e sicura.

Nel momento in cui l'utente effettua l'accesso al sistema, diventa anche in grado di modificare le proprie informazioni di profilo e di personalizzare alcuni aspetti della visualizzazione. In particolare, esso può impostare una foto profilo tra una collezione già presente e modificare l'indirizzo email e la password. Inoltre, ha anche la possibilità di agire su due impostazioni di accessibilità, ovvero la scelta tra un tema chiaro piuttosto che uno scuro e l'attivazione/disattivazione della modalità per daltonici. Quest'ultima fa si che i colori della schermata di gioco siano perfettamente distinguibili da una persona che soffre di daltonismo.

Infine, l'utente autenticato può effettuare la disconnessione e tornare nella situazione iniziale.

Scenario 2: Giocatore Una volta effettuato l'accesso al sistema, l'utente diventa un giocatore e può iniziare a interagire con esso in modo più profondo. Egli può scegliere di creare una partita pubblica, dove l'avversario viene scelto casualmente dal server, o una partita privata, accessibile solo tramite un codice numerico fornito dal gioco.

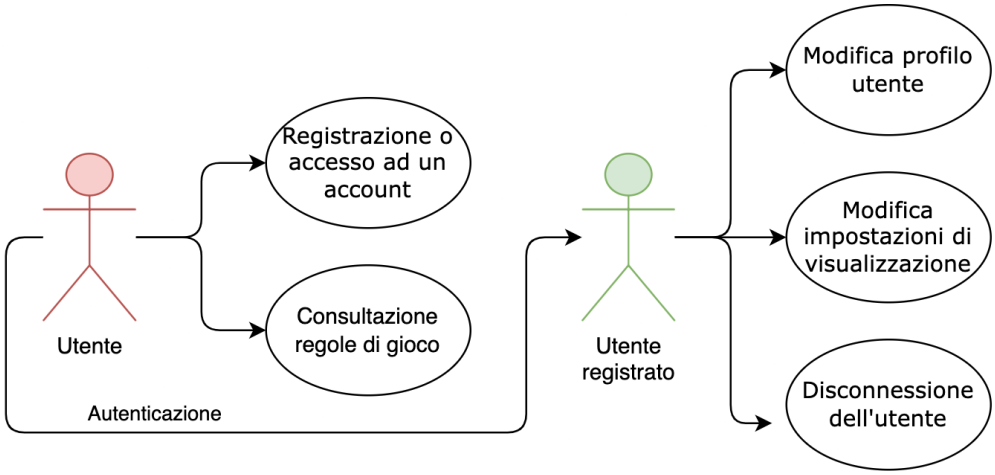


Figure 1: *Use Case Scenario dal punto di vista di un utente*

Dalla schermata principale è possibile visionare lo storico delle partite in corso e di quelle terminate, così come lo stato attuale di tutti gli avversari. Per ogni match, è possibile visualizzare un riepilogo che mostra i tentativi effettuati, compresi di suggerimenti, e l'eventuale esito della partita. Da questa schermata è anche possibile abbandonare il match, concedendo la vittoria all'avversario, ed accedere alla schermata di gioco.

Durante una partita, il giocatore deve cercare di indovinare un codice composto da una sequenza di 4 colori, basandosi sui suggerimenti forniti dai tentativi precedenti. Si hanno a disposizione 10 tentativi totali per indovinare la sequenza e il giocatore che farà la prima mossa viene identificato dal gioco, per poi procedere a turni alterni. Colui che indovina per primo il codice di colori vince la partita. Se nessuno dei giocatori riesce a indovinare la sequenza, la partita termina in pareggio.

1.3 Politiche di valutazione

Per valutare l'*efficacia* del codice prodotto verrà adottato un approccio integrato che combina Unit Tests automatizzati e test sul campo con coinvolgimento degli utenti. In particolare, il codice del backend verrà verificato mediante JUnit [6] e i test copriranno sia le funzionalità relative agli utenti sia quelle inerenti le operazioni di gioco. Verranno predisposti anche dei test specifici per verificare il comportamento del server HTTP, soprattutto in ciò che riguarda la gestione dei cookies e degli headers. Il frontend, invece, verrà testato grazie agli utenti che proveranno il sistema cercando di coprire tutti i casi d'uso possibili.

Per garantire la *qualità* del codice prodotto, si intende applicare opportune tecniche e pattern di design del software, oltre a effettuare refactoring periodici. Questo permetterà di ottenere un codice chiaro, privo di ripetizioni e il più possibile manutenibile e scalabile.

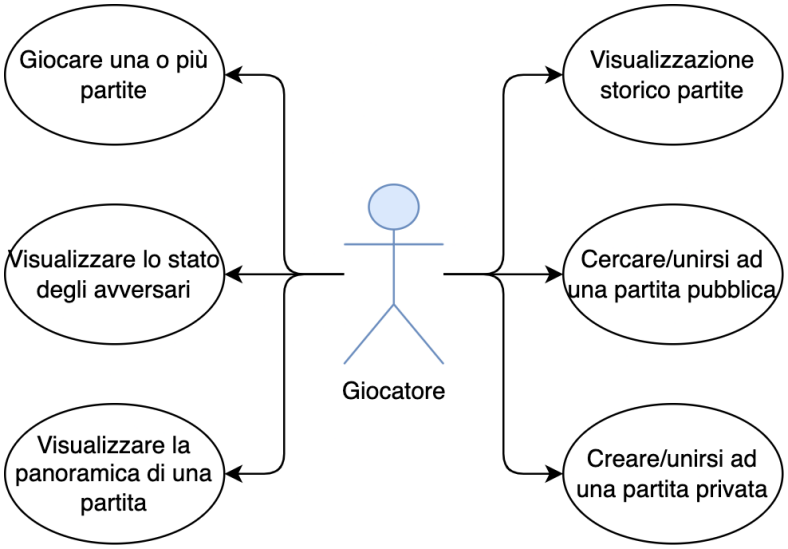


Figure 2: *Use Case Scenario dal punto di vista di un giocatore*

2 Analisi dei requisiti

Verranno ora presentati tutti i requisiti scaturiti durante la fase di analisi, formulati sotto le ipotesi implicite che i giocatori abbiano una conoscenza di base del gioco da tavolo Mastermind e che dispongano di una connessione internet il più stabile possibile.

2.1 Requisiti impliciti

Essi sono necessari per garantire un'esperienza utente soddisfacente e un funzionamento stabile del sistema:

- il sistema dovrebbe avere un'interfaccia utente reattiva, accattivante e facile da usare, dato che il gioco sarà utilizzato da vari tipi di utenti.
- Il sistema deve fornire agli utenti feedback chiari e tempestivi in merito all'esito delle azioni degli stessi, alla natura dei possibili errori e alle azioni compiute dagli avversari. Questo ed il precedente requisito riguardano una porzione del progetto che non è centrale in questo elaborato. Sono stati comunque inseriti per completezza;
- Il sistema dovrebbe essere compatibile con una varietà di dispositivi e piattaforme, consentendo ai giocatori di accedere ad esso da diversi tipi di client e sistemi operativi;
- Il sistema deve garantire che tutte le azioni degli utenti siano sincronizzate correttamente tra i client e il server, in modo che ogni giocatore abbia una visione coerente sullo stato del gioco;

- Il sistema deve essere in grado di memorizzare e recuperare in modo persistente i dati relativi alle partite in corso, inclusi lo stato del gioco, i risultati delle partite precedenti e le preferenze degli utenti;

2.2 Requisiti utente

Dal punto di vista degli utenti, il sistema dovrà permettere di:

- registrare un account con un nome utente, una email ed una password;
- autenticare un utente con la rispettiva coppia nome utente – password;
- modificare a posteriori dell'indirizzo email e della password;
- impostare una foto profilo, tra una collezione di immagini preimpostate;
- modificare l'aspetto generale dell'interfaccia, tra un tema scuro ed uno chiaro, e abilitare o disabilitare la modalità per daltonici;
- disconnettere un utente;
- dare la possibilità di eliminare l'account.

2.3 Requisiti funzionali

Di seguito sono riportati i requisiti riguardanti le funzionalità specifiche che il gioco deve fornire:

- possibilità di cercare una partita pubblica e, se già presente un giocatore in attesa, unirsi ad essa;
- possibilità di creare una partita privata, caratterizzata da un codice numerico di 5 cifre;
- possibilità di unirsi ad una partita privata, previo inserimento del corretto codice di accesso;
- visionare una panoramica relativa ad una specifica partita, la quale mostri i giocatori coinvolti, i tentativi effettuati fino a quel momento e l'esito della stessa;
- capacità di reagire in tempo reale ogniqualvolta si scatena uno di questi eventi: viene individuato un giocatore per iniziare una nuova partita, un avversario effettua un tentativo o abbandona un match, un giocatore cambia il suo stato;
- facoltà di abbandonare una partita in corso, concedendo automaticamente la vittoria all'avversario;
- fornire un feedback per quanto riguarda lo stato attuale degli avversari correnti, ovvero mostrare se essi sono online, offline, o stanno giocando una partita.

2.4 Requisiti non funzionali

Questi requisiti descrivono le qualità globali del software di gioco:

- il sistema dovrà essere il più possibile robusto nei confronti di eventuali crash dei microservizi;
- il backend dovrà essere in grado di gestire numerosi giocatori attivi nello stesso momento, riuscendo a elaborare tutte le richieste con una latenza bassa;
- il sistema dovrà essere in grado di gestire in maniera corretta la connessione e disconnessione degli utenti durante il gioco;
- il sistema dovrà risultare facilmente estendibile;
- in caso di perdita di connessione, il sistema dovrà sempre garantire la consistenza dei dati a discapito della disponibilità.

2.5 Requisiti di comunicazione

Questi requisiti modellano gli aspetti della comunicazione tra le varie componenti del sistema:

- il sistema dovrà includere meccanismi per la rilevazione di problemi di connessione, sia lato frontend che lato backend, gestendo opportunamente tali condizioni;
- le due componenti del backend dovranno interagire necessariamente per mezzo di un sistema a scambio di messaggi, implementato mediante il middleware RabbitMQ;
- l'API RESTful esposta dal server web dovrà rispettare gli standard formali propri del protocollo HTTP e dovrà essere opportunamente documentata;
- il sistema dovrà predisporre una funzionalità per la sincronizzazione in tempo reale tra client e a tale scopo dovrà sfruttare il protocollo WebSocket.

3 Design

Come già accennato nella sezione 1, la struttura generale di questo elaborato si concretizza in un'architettura client-server orientata ai microservizi. La porzione del frontend è stata ripresa quasi completamente dall'elaborato di partenza, pertanto questa sezione ne fornirà solo una descrizione sommaria, che si concentrerà maggiormente sulle modifiche apportate. La porzione del server, invece, verrà analizzata in maniera approfondita, trattando tutti gli aspetti relativi al design, ovvero la struttura, il comportamento e l'interazione con il client.

In Figura 3 è mostrata una visione ad alto livello dell'architettura del sistema nella sua interezza e delle modalità di interazione tra i vari componenti.

- **Devices**: rappresentano i dispositivi dei giocatori sui quali viene eseguito il frontend. Essi interagiscono con il backend mediante due protocolli: per le normali operazioni di gioco sfruttano HTTP, mentre per le comunicazioni in tempo reale utilizzano WebSocket.
- **Web-service**: svolge il ruolo di API Gateway, cioè da punto di ingresso per i dispositivi, e si occupa di mediare le comunicazioni tra essi e il Game-service.
- **Game-service**: può essere visto come il cuore del lato backend in quanto gestisce tutto ciò che riguarda la logica di gioco e le operazioni relative agli utenti. Inoltre, interagisce con il database MongoDB per garantire la persistenza dei dati.
- **RabbitMQ**: è il middleware che consente ai due servizi del backend di comunicare attraverso la rete, tramite scambio di messaggi.

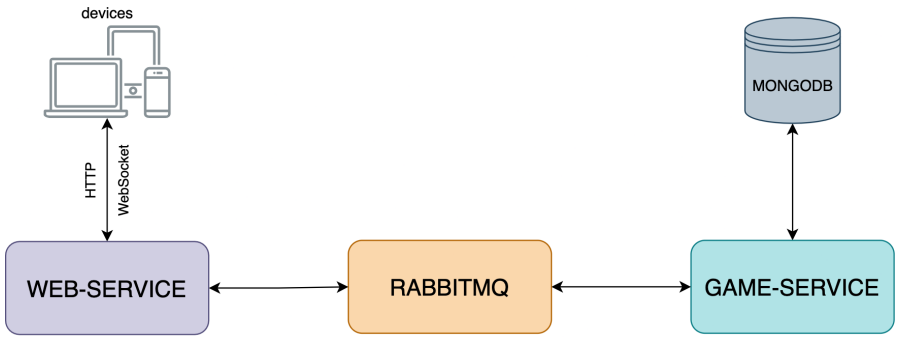


Figure 3: *Panoramica della struttura del progetto*

3.1 Frontend

Come è già stato ribadito, questa porzione di elaborato appartiene ad un altro progetto ed è stata ripresa nella sua interezza, per poi effettuare alcune modifiche e miglioramenti in quegli aspetti inerenti i sistemi distribuiti. Seguirà una breve descrizione della struttura generale della stessa, per poi discutere i contributi di questo progetto.

Il frontend è stato implementato sotto forma di una Single-Page Application (SPA), ovvero un'applicazione web formata da una singola pagina HTML che effettua il render dei vari componenti in maniera dinamica, e quando avviene un cambio di stato ricarica solo la porzione coinvolta. Questa scelta di design possiede alcuni vantaggi, tra cui una maggiore reattività dell'interfaccia grafica, una migliore esperienza dell'utente grazie alla fluidità delle transizioni e una riduzione del carico sul server, che migliora le prestazioni complessive dell'applicazione.

Per la costruzione di un'interfaccia grafica reattiva, intuitiva ed accattivante è stata adoperata la libreria Material UI [8] che fornisce componenti React già pronti, in grado di costruire strutture grafiche complesse in tempi brevi. Inoltre, tutta la UI del sistema

risulta essere responsive, consentendo così di fruire appieno del frontend su qualsivoglia dispositivo.

In qualità di SPA, è stato implementato un efficiente sistema di navigazione tra le pagine, sfruttando l'estensione di React chiamata react-router-dom. Essa permette di definire una serie di rotte aventi un percorso specifico, alle quali sono associate le diverse pagine dell'applicazione [15]. Tale meccanismo fa si che, all'atto di cambio schermata, React non ricarichi tutta la pagina ma modifichi solamente i componenti da visualizzare a schermo. Le varie rotte sono state strutturate in maniera annidata e dividendo quelle protette (cioè che richiedono l'autenticazione) da quelle pubbliche. L'annidamento delle rotte permette di condividere interi componenti React con tutte le pagine figlie. Questa caratteristica è stata usata, ad esempio, per l'imposizione di un layout comune a tutta l'applicazione che comprende un header in cima e il corpo delle schermate subito sotto. Inoltre, tutte le rotte protette sono state a loro volta annidate al di sotto di due rotte fittizie, le cui funzioni sono quella di mantenere l'autenticazione in tutte le pagine dell'applicazione che lo richiedono, e quella di verificare che l'utente abbia effettuato l'accesso prima della visualizzazione di una delle schermate protette. In figura 4 viene riportata una rappresentazione schematica delle varie sezioni del frontend, esplicitando anche la gerarchia tra rotte.

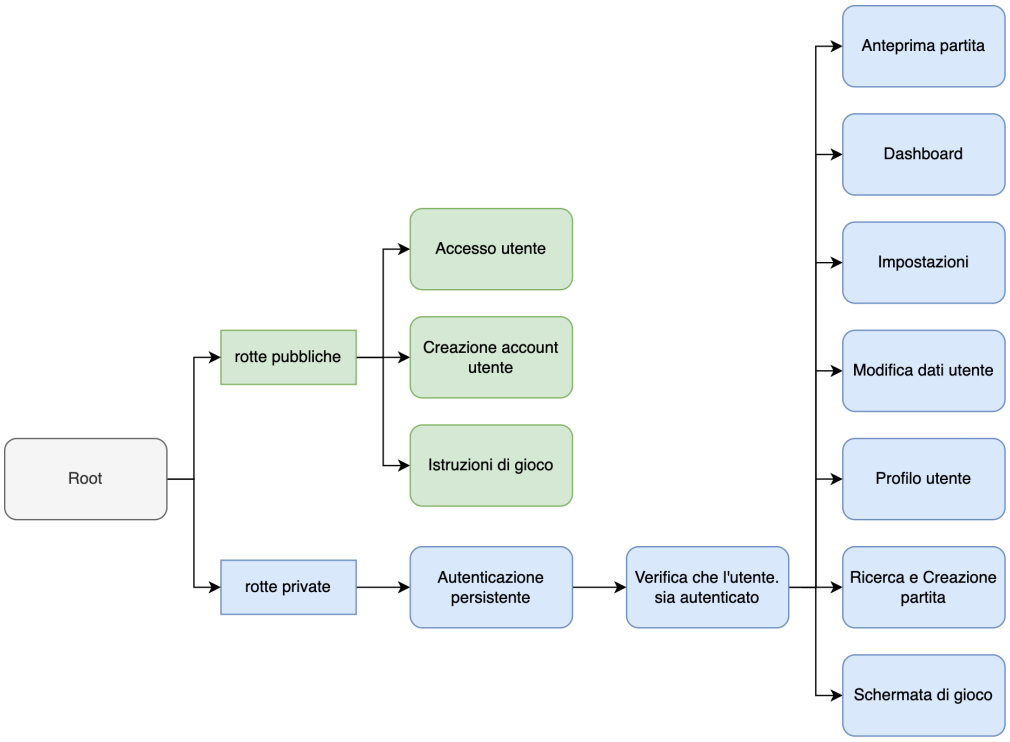


Figure 4: *Struttura schematica del frontend*

Verranno ora presentati i contributi apportati al frontend nel contesto di questo progetto, che sono relativi agli aspetti di comunicazione con il server e alla robustezza nei

confronti dei problemi di rete.

3.1.1 Comunicazione WebSocket lato client

Nel elaborato iniziale, il client aveva già la capacità di comunicare con il server tramite WebSocket, consentendogli di ricevere aggiornamenti sugli eventi di gioco in tempo reale. Ad esempio, quando l'avversario sottometteva un tentativo, il giocatore in attesa del proprio turno riceveva immediatamente un messaggio dal server che scatenava l'aggiornamento dell'interfaccia grafica per riflettere il cambiamento di stato. Nonostante ciò, è stato inevitabile rivedere questo aspetto poiché la libreria utilizzata per implementare WebSocket sul lato server è stata sostituita da *SockJS*, abbandonando *socket.io*. Tale decisione è stata presa in virtù del fatto che il framework *Vertx Web*, impiegato nel backend, fornisce un'implementazione di *SockJS* che si integra perfettamente con esso e con il suo eventbus. La migrazione è stata agevolata dall'ausilio della libreria *eventbus-bridge-client* [3], la quale consente a un frontend JavaScript di connettersi ed interagire con l'eventbus in modo trasparente. Essa, dietro le quinte, utilizza *SockJS* per stabilire una connessione WebSocket con l'istanza di Vertx sul server, facilitando così l'invio di messaggi e la registrazione di gestori su indirizzi specifici che consentono di reagire agli eventi relativi alla ricezione dei messaggi. Inoltre, tale libreria possiede un meccanismo in grado di mantenere attiva la connessione inviando periodicamente al server un messaggio di ping. Quest'ultimo, se non riceve nessun *heartbeat* per un tempo superiore a quello definito in fase di configurazione (il cui valore di default è 5 secondi), scatena un evento di `SOCKET_IDLE` a significare il fatto che quella socket risulta inattiva. Questa caratteristica è molto importante nell'ambito dei sistemi distribuiti in quanto permette di mettere in atto dei meccanismi per rilevare e gestire in maniera corretta i problemi di connessione che possono manifestarsi. In questo progetto è stata impiegata per identificare le disconnessioni improvvise dei client, come spiega meglio la sezione 3.1.2.

Tuttavia, è importante notare che *socket.io* offre una maggiore maturità e una più ampia gamma di funzionalità rispetto a *SockJS*, il che ha reso necessaria l'introduzione di alcuni meccanismi per ridurre il divario tra le due tecnologie. In particolare, si è dovuto far fronte all'implementazione del concetto di **stanza**, non presente in *SockJS*, che consente al server di inviare messaggi verso un preciso sottoinsieme di client connessi, identificati dalla stanza a cui appartengono. Sebbene l'implementazione di questo meccanismo risieda nel backend, lato frontend si è dovuto gestire il processo di unione ad una stanza fittizia, in modo che i client ricevano solo ed esclusivamente i messaggi destinati a loro. Tale stanza è identificata tramite lo username del giocatore e l'accesso ad essa avviene in maniera implicita impostando, per quel client, l'indirizzo di ricezione dei messaggi su una specifica stringa. Essa è ottenuta tramite la concatenazione di un prefisso condiviso con il server e il nome dell'utente. Ad esempio, se il prefisso condiviso fosse la stringa "huesle" e l'utente in questione si chiamasse "bob", il nome della stanza ad esso associata sarebbe "huesle-bob". In maniera analoga, l'utente "alice" sarebbe associato alla stanza "huesle-alice", e così via.

3.1.2 Robustezza ai problemi di rete

Per rendere i client robusti ai problemi di connessione che possono verificarsi sono stati impiegati due accorgimenti, che si differenziano per la natura della criticità che vanno a risolvere. Infatti, è possibile individuare due tipologie di problemi di rete: quelli imputabili al client, ad esempio per una improvvisa mancanza di segnale, e quelli imputabili al server o ad una congestione della rete Internet.

Nel primo caso, è possibile identificare un problema di rete grazie all'evento *Window.offline* [11], che viene lanciato non appena il browser perde l'accesso alla rete. Lo scatenarsi di questo evento viene intercettato da uno speciale componente React che funge da overlay per tutte le schermate della web app. In questo modo, ogniqualvolta il client risulta offline, l'applicazione visualizza un messaggio relativo al problema ed impedisce all'utente di compiere qualsiasi azione, fino a che la connessione non torna disponibile.

Nel secondo caso, viene rilevato un problema di rete grazie all'utilizzo di *SockJS*, che dispone di un sistema di ping verso il server per mantenere attiva la connessione. Se i messaggi di ping non ricevono risposta entro un certo lasso di tempo, si scatena un evento di chiusura della socket che viene gestito dal client innanzitutto informando l'utente del problema. In più, vengono disabilitate tutte quelle funzionalità che richiedono una connessione funzionante, come la ricerca di una partita, l'apertura di un match o ancora la modifica delle impostazioni. L'utente quindi, potrà comunque interagire con l'applicazione ma in maniera limitata. Non appena la socket sarà riaperta, le limitazioni verranno automaticamente eliminate.

I due accorgimenti appena descritti garantiscono consistenza dei dati in tutte quelle situazioni dove si verifica un problema di rete. Questa situazione, però, porta con se la necessità di fare un compromesso. Infatti, come sancisce il CAP Theorem [2], ogni volta che si verifica una partizione di rete è necessario compiere una scelta: garantire la consistenza a discapito della disponibilità oppure mantenere la disponibilità a discapito della consistenza? In questo contesto, si è optato per la prima strada in quanto una temporanea mancanza di disponibilità non causa particolari problemi alla User Experience, vista la natura asincrona della modalità di gioco.

3.2 Backend

Per rispettare i requisiti individuati in fase di analisi, si è optato per una struttura del backend formata da due componenti, che risultano indipendenti l'uno dall'altro e che interagiscono mediante scambio di messaggi. Ciò ha permesso un'efficiente organizzazione del codice e una migliore separazione della logica di gioco da quella necessaria per rendere il sottosistema accessibile da remoto.

Scendendo più nel dettaglio, le due porzioni del backend svolgono le seguenti funzioni:

game può essere considerato il cuore di questo sottosistema in quanto si occupa di implementare tutta la logica di gioco, di gestire gli utenti e di interfacciarsi con il database. Ha anche in compito di effettuare le operazioni di *mashalling* e *unmarshalling*, applicate,

rispettivamente, ai dati in uscita dal server e a quelli in entrata. Inoltre, definisce tutte le entità necessarie alla modellazione del gioco e dei suoi giocatori.

webservice questo componente risulta fondamentale per rendere il sistema accessibile da remoto. Infatti, esso funge da gateway in quanto ha la funzione di ricevere le richieste provenienti dai client, inoltrarle a `game` e restituire le rispettive risposte. In particolare, esso incapsula tutta la logica di gestione di un server HTTP che espone un'API RESTful. Quindi si occupa di definire le rotte, gestire i cookies e supportare l'autenticazione JSON Web Token (JWT) (quest'ultima grazie all'implementazione interna al framework Vertx [18]). Un'altra importante caratteristica di `webservice` è il fatto che funge anche da server WebSocket, implementando un sistema di notifiche che consente ai client di essere aggiornati in tempo reale in merito all'accadimento dei principali eventi di gioco.

3.2.1 Struttura

Verrà ora analizzata la struttura delle due componenti appena descritte, con il supporto di diagrammi delle classi in formato UML. Si consideri che essi non conterranno tutti i singoli attributi e metodi di classe ma solo quelli ritenuti chiave per la comprensione della struttura e della logica applicativa. Ad esempio, verranno omessi tutti i *getters* e *setters*.

Struttura di `game`

In figura 5 è mostrato un diagramma relativo al *modello* dei dati scelto per questo progetto. Esso descrive tutte le entità necessarie per modellare gli elementi di gioco ed i giocatori, mostrando la struttura delle classi impiegate durante l'implementazione e le relazioni tra di esse. Le entità spiegate in dettaglio sono:

- **Match**: rappresenta una partita ed è l'entità centrale. Dispone di un id generato casualmente che viene usato nel database per distinguere univocamente i match. Inoltre, possiede la lista dei tentativi sottomessi, il codice segreto da indovinare e lo stato di quel match. Espone un solo metodo il quale viene chiamato ogni volta che sul server arriva un tentativo da sottomettere. Tale procedura verifica innanzitutto che il mittente coincida con il giocatore del turno corrente. Se coincide, provvede a calcolare i suggerimenti sulla base del codice segreto e verifica se è stato raggiunto il numero massimo di tentativi a disposizione. Nel caso ciò non sia ancora accaduto, controlla se la soluzione è stata indovinata. In caso affermativo, imposta lo stato della partita in modo che indichi un vincitore, altrimenti invoca il metodo per cambiare turno. Infine, se i tentativi disponibili sono esauriti ma la soluzione è ancora ignota, imposta lo stato della partita in modo che indichi un pareggio;
- **MatchStatus**: questa classe racchiude una serie di informazioni inerenti il progresso di una partita, ovvero la lista dei due giocatori coinvolti, il nome utente di chi deve fare la prossima mossa, lo stato in cui si trova la partita e se essa è stata o meno abbandonata da uno dei due sfidanti;

- **MatchState**: rappresenta un *Enum* che definisce i nomi dei diversi stati in cui può trovarsi una partita:
  1. `PLAYING` la partita è in corso: nessuno ha ancora indovinato il codice segreto e la stessa non è stata abbandonata;
  2. `VICTORY` la partita è terminata con un vincitore. Se uno dei due giocatori abbandona il match, esso termina con questo stato, concedendo la vittoria all'altro sfidante;
  3. `DRAW` la partita è terminata con un pareggio.

- **Player**: è la classe che rappresenta gli utenti del gioco che, una volta autenticati, diventano dei giocatori. Essi, nel database, vengono identificati univocamente mediante il nome utente, attributo che non può essere modificato in seguito alla creazione dell'account. Altri attributi presenti sono l'indirizzo email, la password, l'immagine del profilo, alcune impostazioni di visualizzazione, un booleano che indica se il profilo è stato eliminato e, infine, token di refresh per JWT. La password non viene mai gestita in chiaro in quanto ne viene calcolato l'hash fin dall'inizializzazione dell'oggetto Player. Il metodo pubblico `verifyPassword()` consente di verificare se una data stringa in chiaro corrisponde alla password scelta in fase di registrazione e lo fa comparando l'hash della password in input con quello memorizzato nell'oggetto;

- **SecretCode**: modella il codice segreto che deve essere indovinato dagli sfidanti per vincere la partita. Si compone di una sequenza di quattro colori, identificati dal nome della tonalità e scelti casualmente tra un insieme di 6 colori. La procedura `generateColorSequence()` produce sequenze di colori che ammettono ripetizioni di uno o più valori.

- **Attempt**: modella un tentativo sottomesso da un giocatore, caratterizzato dal nome dell'utente che lo ha inviato, dalla sequenza dei 4 colori scelti e dai suggerimenti relativi a quest'ultima. Essi vengono calcolati grazie al metodo `computeHints (attempt)`, il quale compara il tentativo in input con il codice segreto associato a quella partita;

- **Hints**: classe che contiene i suggerimenti relativi ad un tentativo. È caratterizzata dal numero di colori giusti in posizione giusta e da quello dei colori giusti ma in posizione sbagliata. Entrambe le informazioni vanno considerate in base al codice segreto.

- **AccessibilitySettings**: rappresenta due impostazioni di visualizzazione a disposizione dell'utente che consentono di migliorare l'esperienza di fruizione dell'applicazione. La prima riguarda il tema, potendo scegliere tra tonalità chiare oppure scure. La seconda risulta essenziale per i giocatori che soffrono di daltonismo in quanto consente di giocare usando una palette appositamente creata affinché tutti i suoi colori siano perfettamente distinguibili, indipendentemente dalla tipologia di daltonismo a cui si è affetti.

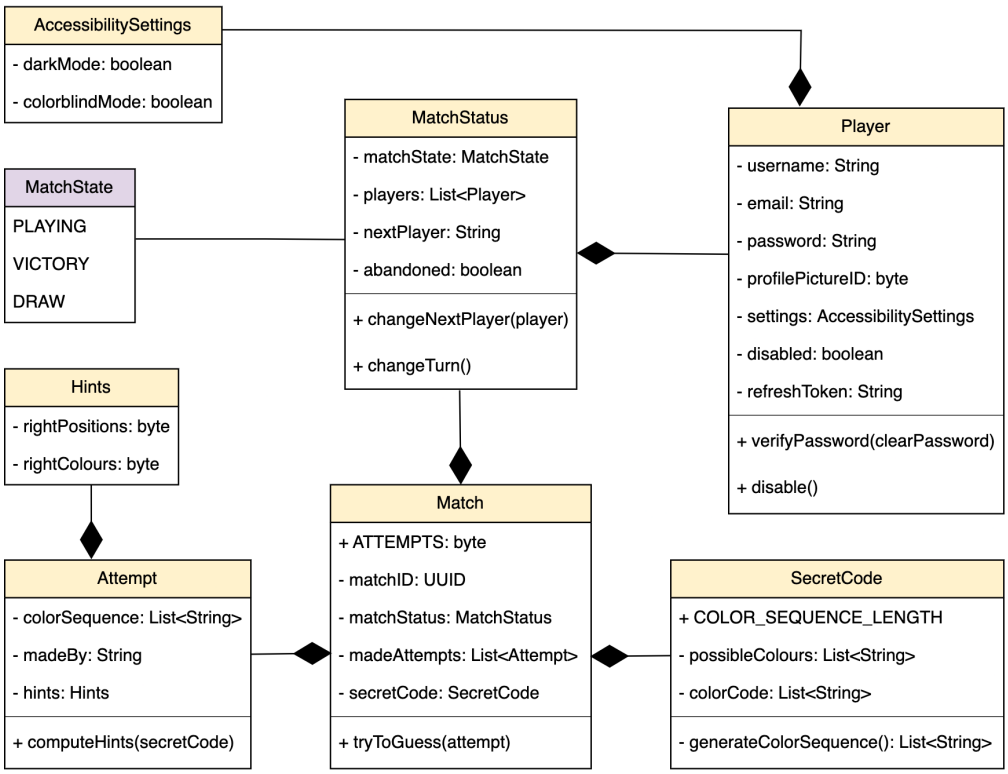


Figure 5: *Diagramma delle classi in UML relativo al modello dei dati*

Per quanto riguarda l'architettura di `game`, che ne descrive il funzionamento indipendentemente dal modello dei dati, è stata concepita una struttura a callbacks suddivisa in due gestori: uno per le operazioni che riguardano gli utenti e uno per quelle inerenti il gioco. Questi gestori sono incaricati di associare i tipi di messaggi (`MessageType`) ai metodi della logica applicativa, i quali sono definiti in appositi controller. Questi ultimi sono suddivisi in base alla categoria delle operazioni che gestiscono e in questo contesto ne esistono quattro tipologie che si occupano di:

1. gestire gli utenti (registrazione, accesso, ecc.),
2. implementare la logica di gioco (effettuare un tentativo, chiedere la lista delle partite relative ad un utente, ecc.),
3. modificare le impostazioni utente (abilitare il tema scuro, cambiare la foto profilo, ecc.),
4. ottenere alcune statistiche (conteggio delle partite vinte, perse e finite in pareggio, relative ad un utente).

È presente un `MessageType` per ogni callback definita e l'architettura qui descritta, visualizzabile in figura 6, fa si che l'arrivo di un messaggio, originatosi sul frontend, sia seguito dalla invocazione della callback associata al tipo della richiesta appena ricevuta.

La capacità del sistema nel ricevere i messaggi è garantita dall'integrazione con RabbitMQ. A tal fine, è stato sviluppato un componente dedicato che incapsula i dettagli implementativi relativi alla libreria ufficiale Java. Esso agisce come un'interfaccia tra il backend e il sistema di messaggistica, consentendo una comunicazione affidabile e scalabile tra le diverse parti del sistema. Volendo essere più specifici, questo componente implementa il lato server del pattern Remote Procedure Call (RPC) [16] e per questo motivo è responsabile di ricevere i messaggi provenienti dal frontend, di eseguire la callback associata al tipo di quel messaggio e infine di rispondere con il risultato ottenuto.

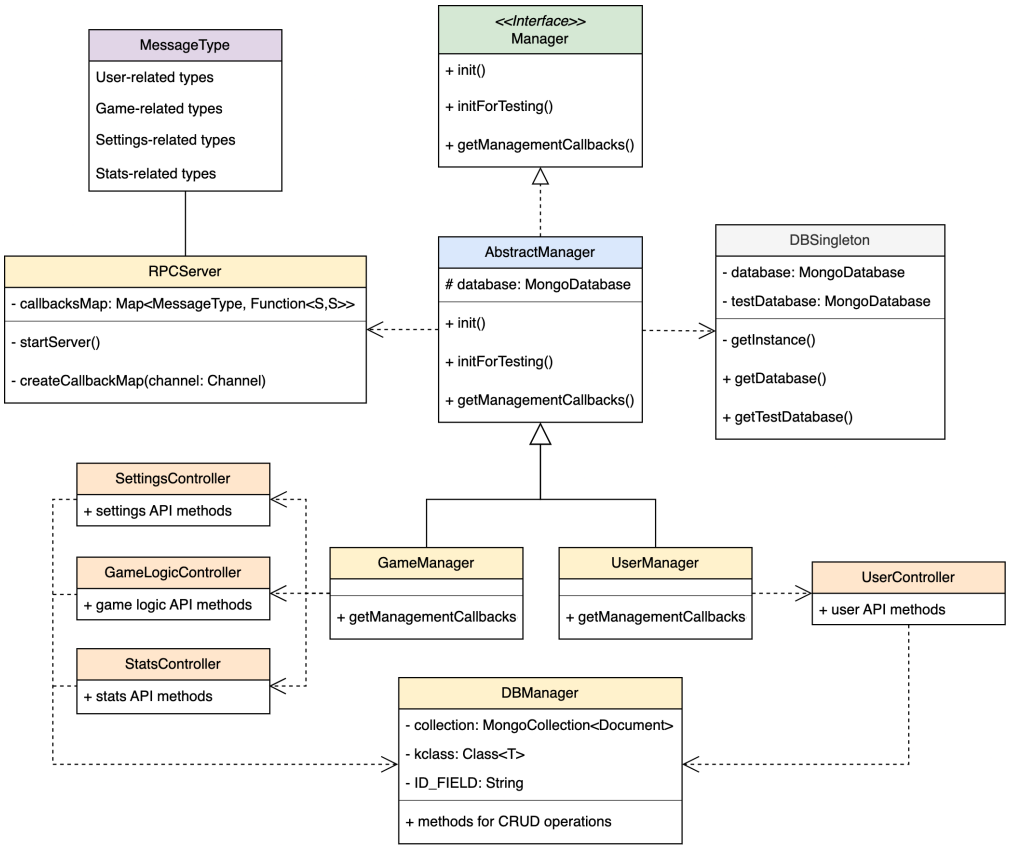


Figure 6: *Diagramma delle classi in UML relativo alla struttura di* `game`

Un altro aspetto interessante riguarda la persistenza dei dati, soprattutto in caso di malfunzionamenti improvvisi di una o più parti del sistema. Per raggiungere questo obiettivo, tutti i cambiamenti di stato ed i risultati prodotti dalle operazioni che esegue `game` vengono sempre memorizzati nel database, compresi i dati relativi agli utenti. Durante il processo di sviluppo di questa funzionalità è nata l'esigenza di avere un componente riutilizzabile che incapsulasse tutta la logica per effettuare le operazioni CRUD, astraendo completamente da quelli che sono i dettagli implementativi dei driver MongoDB per Java [10]. In particolare, questa classe prende il nome di DBManager e consente di eseguire le procedure di lettura e scrittura su una specifica collezione, caratterizzata dal tipo di

dato che memorizza, scelto al momento della creazione dell'oggetto. Questo significa che occorre avere un'istanza di tale componente per ogni collezione che si vuole manipolare, anche se esse appartengono allo stesso database.

Normalmente, per ottenere una collezione su cui lavorare, è necessario innanzitutto aprire una connessione verso il database che la contiene. Dal momento che più componenti del backend hanno la necessità di accedere alla base di dati, si è pensato di introdurre un'architettura che stabilisse tale connessione solo una volta, per poi condividere l'istanza del database con tutti gli oggetti DBManager del caso. A questo proposito, si è scelto di predisporre una classe che segue il pattern singleton per mantenere l'oggetto relativo ad uno specifico database. Ciò garantisce di stabilire solo una connessione con il microservizio che ospita MongoDB, indipendentemente dal numero di classi che ne richiedono l'utilizzo.

Struttura di `webservice`

Questo è la porzione del sistema che funge da interfaccia tra il frontend e il backend, consentendo di fare il salto nel distribuito. Infatti, come già accennato in precedenza, tale componente gestisce un server HTTP che definisce tutte le rotte previste dalla API e che controlla gli aspetti inerenti tale protocollo, come cookies, verifica degli headers, parsing del corpo delle richieste e identificazione dei *path params*. Il server appena citato offre anche un'ulteriore rotta rispetto a quelle previste dalla API che consente di instaurare una connessione tramite il protocollo WebSocket tra il backend ed il frontend.

Questa parte del sistema sfrutta in maniera profonda il framework Vertx, proprio per questo è stato possibile concepire una struttura basata su *Verticles*, ovvero componenti autonomi e riutilizzabili, caratterizzati da un modello di esecuzione ad event loop. Per l'implementazione concreta di `webservice` si è scelto di impiegare due Verticles: il primo si occupa di eseguire il server HTTP, di fatto implementando l'API, mentre il secondo viene utilizzato per gestire le comunicazioni tramite WebSocket allo scopo di inviare notifiche in tempo reale al frontend.

Similmente a quanto si è detto per `game`, anche il sottosistema in oggetto ha la necessità di essere integrato con RabbitMQ. Infatti, una volta ricevute le richieste HTTP dal frontend, esso le deve propagare al backend per poter essere gestite. A tal scopo, è stato predisposto un componente che rappresenta la controparte del server RPC, citato in precedenza, e che prende il nome di client RPC. Esso, in sostanza, si occupa di iniziare una richiesta invocando il metodo remoto `call()` sul server, per poi attendere il relativo risultato.

Una visione schematica della struttura del sottosistema in oggetto viene mostrata in figura 7, nella quale il colore rosso denota le classi che racchiudono un Verticle.

Server HTTP e WebSocket Il server HTTP di questo sistema è formato da una gerarchia di rotte, suddivisibili in due categorie: rotte pubbliche e rotte private. Le prime risultano accessibili da chiunque e coprono le operazioni relative agli utenti. Le seconde, invece, richiedono un'autenticazione per poter essere accedute e gestiscono le operazioni

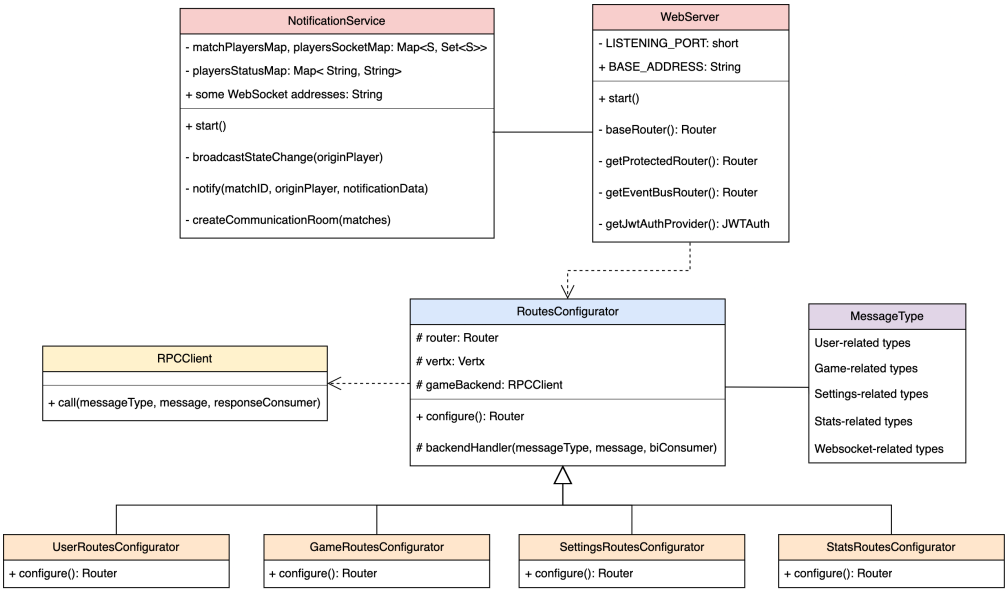


Figure 7: *Diagramma delle classi in UML relativo alla struttura di* `webservice`

di gioco. Vi è poi un'altra rotta, posta in cima alla gerarchia, che funge da punto di ingresso per le connessioni WebSocket.

Per rendere il sottosistema più organizzato e modulare, gli handler delle varie rotte sono stati raccolti in molteplici *Router*. Questi oggetti, messi a disposizione da Vertx Web, consentono di suddividere logicamente le diverse funzionalità del web server in unità separate e coese, facilitando la gestione e il mantenimento del codice. Inoltre, essi consentono di applicare politiche di gestione delle richieste diverse per diverse parti dell'API, contribuendo a una progettazione modulare e scalabile. Questa funzionalità è stata impiegata per proteggere l'accesso alle cosiddette rotte protette. In particolare, è stato predisposto un handler che viene invocato ogni volta che il server riceve una richiesta per una rotta di questo tipo e, prima di elaborarla, verifica l'autenticazione dell'utente tramite JWT.

Più in generale, tutti i Router sono stati raccolti in apposite classi, dette Configurator. Queste classi agevolano il processo di configurazione delle rotte in quanto separano la logica in molteplici componenti, migliorando la leggibilità del codice e rendendo il processo di costruzione della gerarchia dell'API semplice e intuitivo. Scendendo più nel concreto, è possibile individuare la classe RoutesConfigurator che si occupa di effettuare alcune operazioni iniziali e che funge da super-classe per altri 4 Configurators. Essi sono coloro che effettivamente raccolgono tutti gli handler relativi alle varie rotte e definiscono tutta la logica necessaria ad implementare l'API RESTful. Questa suddivisione rispecchia quella introdotta in `game` per quanto riguarda i controllers.

La struttura appena descritta può essere visualizzata nella figura 8.

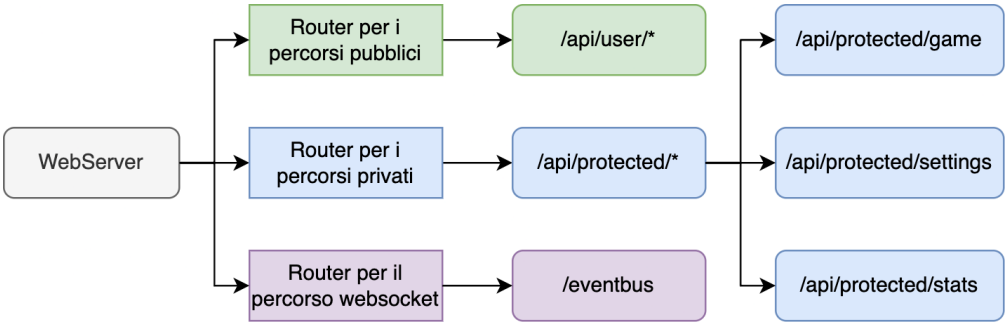


Figure 8: *Gerarchia delle rotte definite nel server web*

Sistema di notifiche Il Verticle NotificationService si occupa di incapsulare tutta la logica necessaria a gestire la comunicazione WebSocket tra client e server. Per questo motivo espone dei metodi in grado di inviare messaggi in tempo reale ad uno specifico utente oppure ad un sottoinsieme di essi, implementando, di fatto, il concetto di stanze discusso nella sezione 3.1.1.

3.3 Comportamento

In questa sezione verrà descritto il funzionamento del backend, specificandone il comportamento durante i diversi flussi di esecuzione. Visto che la descrizione inizierà dal momento in cui il sistema viene avviato, si vuole far notare come le due porzioni del backend possono potenzialmente essere eseguite in maniera totalmente indipendente l'una dall'altra, anche su macchine diverse. L'unico requisito che richiedono coincide con l'avere un server RabbitMQ attivo e raggiungibile dalle macchine che eseguono i microservizi.

Comportamento di `game`

All'avvio di questo microservizio, la classe Main provvede ad istanziare i due gestori, ovvero GameManager e UserManager. Questa operazione, in maniera automatica, inizializza la classe astratta AbstractManager che essi ereditano, la quale si occupa di due fondamentali aspetti:

1. il primo coincide con l'inizializzazione della classe DBSingleton, la quale instaura una connessione con MongoDB, per poi ottenere l'oggetto relativo al database che consentirà di lavorare sulle sue collezioni;

2. il secondo consiste nel mandare in esecuzione i server RPC in modo che il microservizio sia effettivamente in grado di ricevere le richieste. Ogni gestore ha il proprio RPCServer, in questo modo le operazioni relative agli utenti e quelle inerenti la logica di gioco possono essere gestite in parallelo. Una descrizione dettagliata riguardo il funzionamento del server RPC è riportata nella sezione 4.1.

Tornando al funzionamento di `game`, i due manager, in fase di inizializzazione, stabiliscono l'associazione tra i `MessageType` ed i metodi presenti nei rispettivi controllers. Questa è una fase molto importante nel processo di avvio in quanto "istruisce" il sistema su come reagire ai diversi tipi di messaggi che riceve, indicandogli la giusta callback da eseguire.

A questo punto il processo di avvio è terminato e nel caso in cui non ci siano stati errori di inizializzazione, `game` risulta effettivamente attivo e funzionante. Da questo momento in poi, esso entra in uno stato di attesa delle richiesta da elaborare.

I problemi che potrebbero verificarsi durante il processo di avvio riguardano l'inizializzazione del database e del server RPC. In particolare, potrebbe verificarsi che uno dei due o entrambi i servizi MongoDB e RabbitMQ non siano stati avviati correttamente o risultino irraggiungibili. In questo contesto il backend non è in grado di funzionare correttamente e va riavviato, assicurandosi che i servizi appena citati risultino attivi.

Elaborazione delle richieste Una volta che il componente `game` risulta correttamente avviato, è possibile descrivere come esso si comporta ogni volta che viene ricevuto un messaggio di richiesta.

Innanzitutto, si è deciso di introdurre delle classi di *request* e di *result* per ogni macrocategoria di messaggi da gestire. Le prime hanno il compito di incapsulare i dati relativi alla richiesta, che verranno utilizzati dai controllers. Le seconde, invece, servono per racchiudere i risultati prodotti dal backend, da restituire a `webservice`.

Fatta questa premessa, tutte le callbacks definite nei vari controllers presentano un pattern comune che ne denota il comportamento. Infatti, indipendentemente dal tipo di richiesta che occorre gestire, l'invocazione dei metodi suddetti segue il seguente flusso:

- per prima cosa viene eseguito il processo di deserializzazione, il quale converte una stringa in formato JSON nel corrispondente oggetto Java. La stringa di partenza rappresenta il corpo della richiesta generata dal frontend e ricevuta tramite RabbitMQ, mentre l'oggetto finale si concretizza in una classe di *request*;

- ora, la callback ha a disposizione tutti gli input di cui necessita per potre eseguire. Pertanto, il processo di deserializzazione è seguito dalla vera e propria gestione della richiesta, che comprende un'insieme di computazioni, accesso al database, ecc. Questa fase è fortemente dipendente dalla natura dell'operazione da risolvere, pertanto non può essere generalizzata;

- non appena la precedente fase di elaborazione termina, prima di restituire la risposta, viene istanziata la classe di *result* appropriata e ne viene fatto il marshalling. Questo processo, detto anche "serializzazione", trasforma un oggetto Java nella corrispondente rappresentazione in JSON. Solo a questo punto è possibile restituire il risultato dell'operazione appena eseguita, che verrà restituito al mittente tramite RPCServer.

Il flusso appena descritto è applicabile anche nelle situazioni in cui si generano degli errori a runtime, come ad esempio il lancio di un'eccezione. Infatti, i metodi dei controllers

sfruttano il costrutto `try-catch-finally` per incapsulare tutta la logica di elaborazione. In particolare, nella porzione del `finally` si controlla se è stata generata una risposta. In caso affermativo, significa che l'esecuzione del corpo del metodo è andata a buon fine. In caso negativo, invece, significa che la procedura è stata interrotta prima del previsto e viene quindi generata una risposta d'errore. Questo piccolo accorgimento fa si che le callbacks generino sempre un risultato, indipendentemente dall'esito dell'elaborazione.

Comportamento di `webservice`

Il processo di avvio di questo microservizio è caratterizzato dal deploy dei Verticles da cui è composto. Più nello specifico, il primo ad essere lanciato è WebServer, il quale provvede a:

- configurare le rotte che compongono l'API, partendo da un Router radice. Ad esso vengono agganciati sia l'handler per le comunicazioni WebSocket, sia tutti i subrouter presenti nei Configurator;
- creare un server HTTP impostando come handler principale il suddetto Router radice, per poi metterlo in ascolto sulla porta 8080.

Durante la configurazione delle rotte, in particolare quando vengono istanziate le classi Configurators per l'ottenimento dei relativi Router, si attua una fase di vitale importanza per il backend. Infatti, all'interno della classe astratta RoutesConfigurator viene messo in esecuzione RPCClient, il quale abilita questo microservizio ad interagire con `game`. I dettagli implementativi relativi a questo modulo vengono discussi in dettaglio nella sezione 4.1.

Non appena il web server risulta effettivamente pronto a ricevere richieste HTTP, si attua l'ultima fase del processo di avvio di `webservice`, che vede la messa in esecuzione dell'altro Verticle di cui esso è composto, ovvero NotificationService. Quest'ultimo ha il compito di gestire tutte le comunicazioni WebSocket tra frontend e backend e il suo comportamento risulta quello di rimanere in attesa di specifici messaggi sull'eventbus, per poi reagire ad essi tramite opportuni handlers.

3.4 Interazione

In questa sezione viene descritto come interagiscono i componenti del sistema durante le principali fasi di funzionamento, corredando ogni scenario con il corrispondente diagramma di sequenza UML.

3.4.1 Interazioni lato utente

Registrazione di un utente Il processo di registrazione inizia quando il frontend invia una richiesta HTTP di tipo POST all'API, inserendo nel corpo un oggetto JSON contenente lo username, l'indirizzo email e la password in chiaro. Questa richiesta viene intercettata da `webservice`, il quale delega RPCClient di recapitarla a `game` tramite RabbitMQ, specificando il tipo di messaggio `REGISTER_USER`. Nell'ambito di `game`,

non appena l'RPCServer di UserManager riceve la richiesta di registrazione, scatena l'invocazione della callback corrispondente. Essa ha l'effetto di creare un nuovo giocatore, di aggiungerlo al database e infine di terminare con esito positivo. Il risultato viene quindi restituito a `webservice`, il quale provvede a terminare la richiesta HTTP.

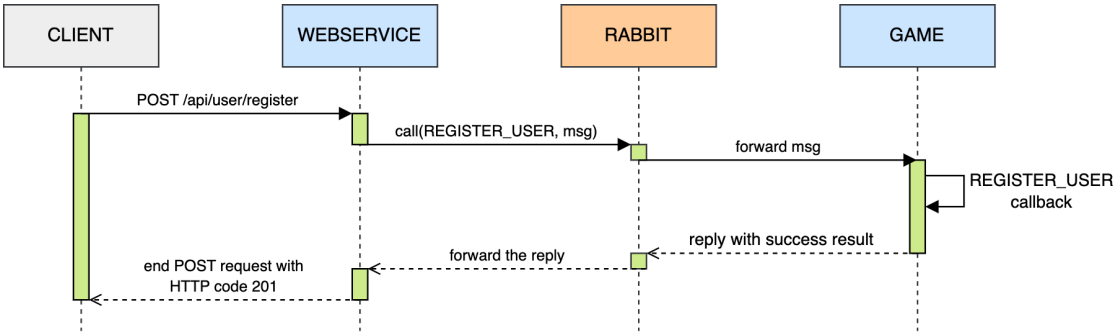


Figure 9: *Diagramma di sequenza per la registrazione di un utente*

Autenticazione di un utente Il processo di login risulta molto simile per quanto riguarda la parte di invio della richiesta. Gli unici cambiamenti risiedono nel corpo della stessa, che ora non contiene l'email, ed nel tipo di azione, il quale è pari a `LOGIN_USER`. Il metodo di `game` associato a questa operazione ha il compito di verificare l'esistenza dell'utente nel database, di verificare che le credenziali siano corrette e infine di autenticare l'utente. Quest'ultimo passaggio si concretizza nella generazione dei token per implementare il meccanismo di autenticazione JWT. La callback termina con la restituzione dei suddetti dati di accesso al server HTTP, il quale provvede ad inserire il refreshToken in un apposito cookie, e l'accessToken nel corpo della risposta HTTP. Quest'ultimo verrà usato dal client per autenticare le future richieste alle rotte protette.

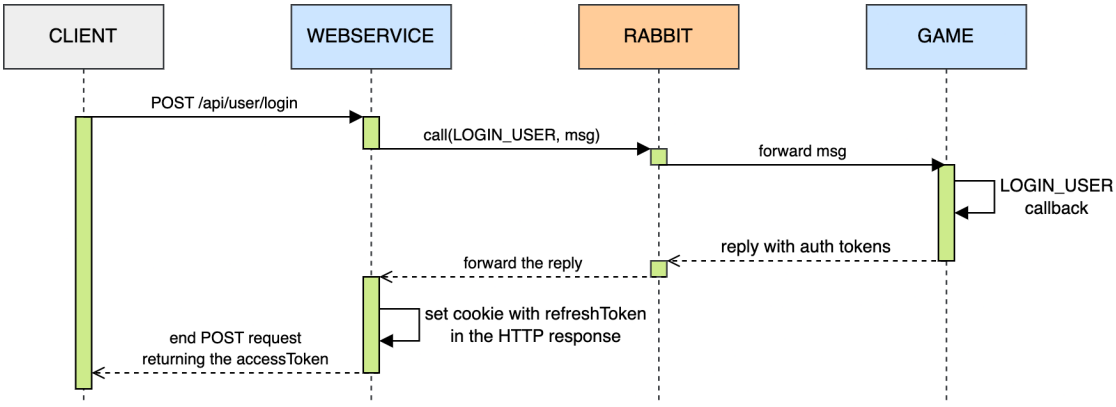


Figure 10: *Diagramma di sequenza per il login di un utente*

Disconnessione di un utente Per effettuare il logout di un utente si sfrutta l'apposita chiamata di tipo GET all'API. All'arrivo della richiesta, `webservice` verifica prima di tutto la presenza del cookie contenente il refreshToken. Se esso è presente, ne viene estratto il valore per poi invocare il relativo handler sul backend. Al contrario, se non viene trovato alcun cookie, significa che l'utente ha già effettuato la disconnessione in precedenza.

Lato `game`, per gestire correttamente il processo di logout viene anzitutto verificata la presenza nel database di un utente associato al refreshToken ricevuto. In caso di corrispondenza positiva, si elimina il valore di tale token dal database e si restituisce un messaggio di successo.

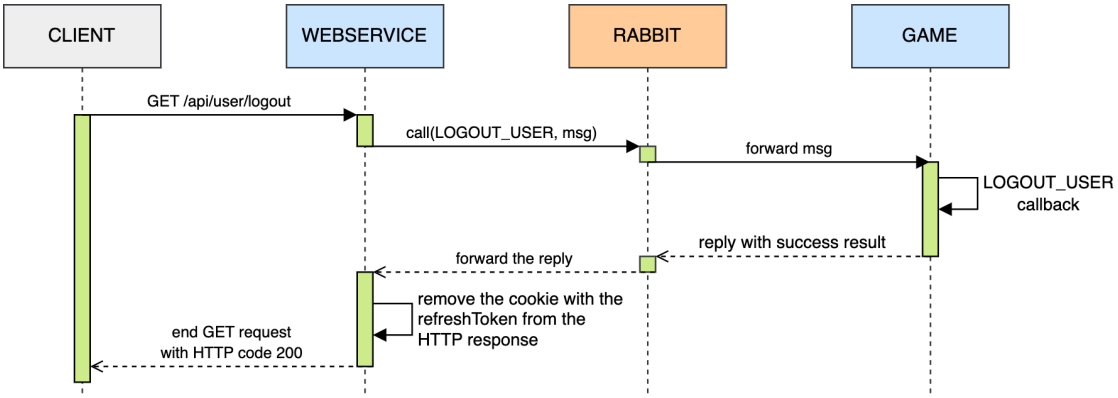


Figure 11: *Diagramma di sequenza per la disconnessione di un utente*

3.4.2 Interazioni lato partita

Da qui in poi, tutte le interazioni sono soggette all'autenticazione tramite JWT. Per questo motivo, il client deve accompagnare ogni richiesta HTTP con l'apposito header contenente il token di accesso. Mentre il backend, prima di elaborare qualsiasi richiesta verso queste rotte, deve verificare che essa sia accompagnata dall'header "Authorization Bearer" e che le informazioni di accesso in esso contenute siano valide.

Ricerca o creazione di una partita Questa azione è piuttosto articolata e comprende una serie di casistiche possibili. In sostanza, la sua funzione è quella di cercare una partita pubblica a cui unirsi oppure crearne una nuova. Quest'ultima può essere sia pubblica, sia privata. L'unico caso non gestito dalla presente azione è quello di ricerca di una partita privata, scenario esaminato nella prossima interazione.

All'arrivo della richiesta POST proveniente dal client, `webservice` esamina il body della stessa ed estrapola il valore del booleano "secret". Questo booleano viene impostato dal client per specificare se la richiesta ha a che fare con una partita pubblica o privata.

A questo punto viene invocata la relativa callback presente nel modulo `game`, dove risiede la vera e propria gestione della ricerca o creazione di un match. Come prima cosa, si verifica la presenza di una richiesta pendente effettuata dallo stesso utente. Ciò

serve per prevenire di effettuare più di una volta la stessa richiesta. Nel caso in cui non vi sia una richiesta pregressa dello stesso tipo, si procede verificando se nel database è presente una pendingRequest fatta da un altro giocatore. Qui si aprono due strade possibili:

- se ne viene trovata una che coincide con il tipo di partita richiesta, si procede alla creazione di una nuova partita, usando come giocatori l'utente corrente e quello che aveva sottomesso la pendindRequest. A questo punto si procede rimuovendo la richiesta pendente dal database e si conclude restituendo un messaggio di successo. In questo caso è stata creata una partita pubblica.

- nel caso in cui non viene trovata nessuna pendingRequest pertinente per il tipo di richiesta, occorre crearne una, in modo che possa essere soddisfatta in futuro da un altro giocatore. Per far ciò, si procede all'inserimento nel database di una nuova richiesta pendente, il cui tipo di partita viene stabilito in base al valore di "secret", estrapolato all'inizio . Più nel dettaglio, se tale attributo ha il valore `false`, la pendingRequest si riferisce ad una partita pubblica ed è caratterizzata dal solo username dell'utente che la ha generata. Al contrario, se l'attributo in questione assume il valore `true`, la richiesta fa riferimento ad una partita privata e, oltre al nome utente, contiene anche un codice. Questo codice viene generato casualmente dal backend e si compone di 5 cifre numeriche. Esso risulta fondamentale per poter soddisfare la richiesta pendente appena creata e, di conseguenza, accedere alla partita in questione.

Al termine dell'esecuzione della callback sul backend, oltre a terminare la richiesta HTTP, `webservice` provvede anche a notificare gli eventuali clients attivi del fatto che è stata creata una nuova partita. Questa segnalazione, fatta tramite il protocollo WebSocket è importante in quanto consente ad un giocatore in attesa di un avversario di essere immediatamente notificato non appena la sua richiesta pendente viene soddisfatta.

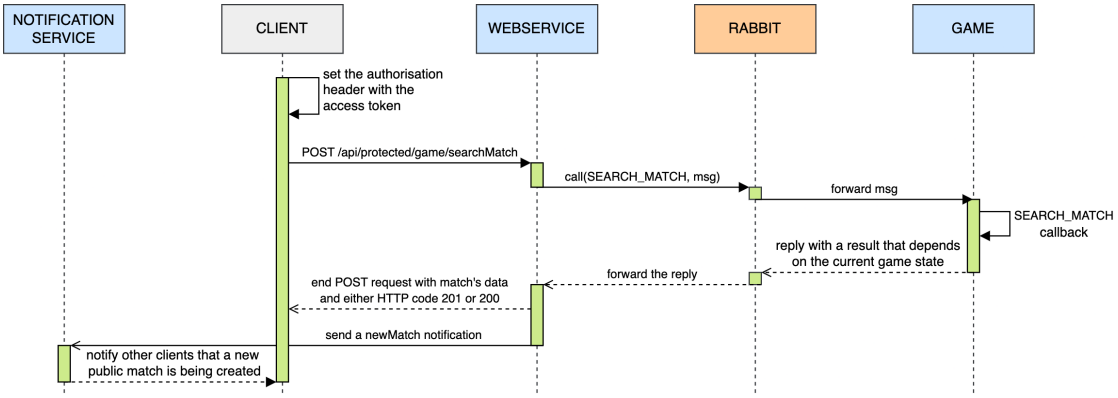


Figure 12: *Diagramma di sequenza per la ricerca di una partita pubblica*

Ricerca di una partita privata Il processo di unione ad una partita privata inizia con il client che effettua una chiamata POST alla rotta apposita dell'API, includendo nel corpo della richiesta il codice secreto per l'accesso alla partita. Quest'ultimo viene intercettato da `webservice` ed inoltrato al backend.

A questo punto, `game` inizia a gestire la richiesta andando a vedere se sul database sono presenti delle pendingRequest associate al codice ricevuto dal client. Se non viene trovato nulla, significa che l'utente ha inserito un codice errato. Al contrario, se la query fornisce un risultato positivo, si procede in maniera simile al caso precedente. Infatti, si provvede anzitutto ad eliminare la richiesta pendente dal database, per poi creare la nuova partita e restituire un messaggio di successo.

Anche in questo caso, in concomitanza con la terminazione della richiesta HTTP, `webservice` provvede a notificare il fatto che è appena stata creata una nuova partita.

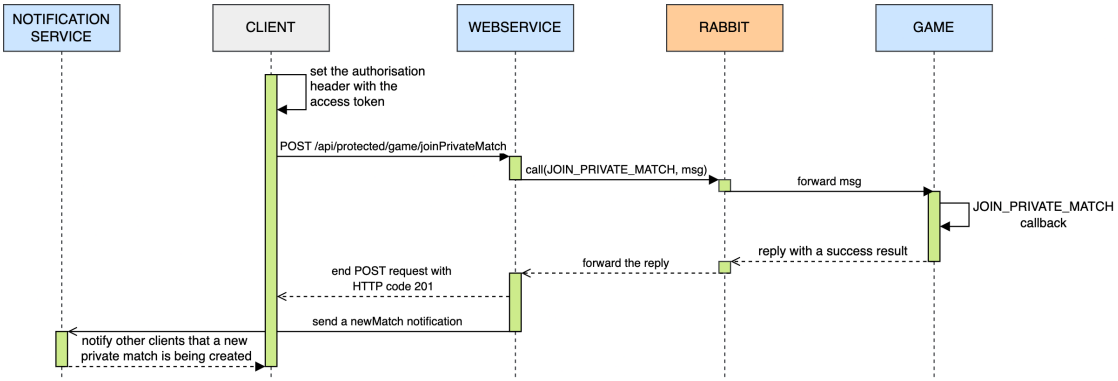


Figure 13: *Diagramma di sequenza per la ricerca di una partita privata*

Abbandono di una partita in corso Questo scenario si verifica quando un utente desidera abbandonare una partita in corso, senza perciò volerla concludere continuando a giocare.

Le fasi iniziali di creazione ed inoltro della richiesta rimangono pressoché invariate. Lato backend, è interessante evidenziare le operazioni che devono essere messe in atto per garantire una corretta gestione dell'abbandono anticipato di un match. In particolare, si inizia verificando la presenza nella base dati di una partita avente lo stesso ID del match per il quale è giunta la richiesta e che abbia come giocatore l'utente che vuole abbandonarla. Se questa verifica va a buon fine, e la partita trovata non risulta già conclusa, si procede impostando su di essa uno stato di vittoria e cambiando il prossimo giocatore con l'altro utente del match. Queste due condizioni consentono di attribuire la vittoria all'avversario dell'utente che ha iniziato la richiesta di abbandono.

Non appena il backend restituisce il controllo a `webservice` per la terminazione della richiesta HTTP, quest'ultimo si occupa anche di emettere una notifica sulla stanza del match appena abbandonato per segnalare all'altro giocatore questo aggiornamento.

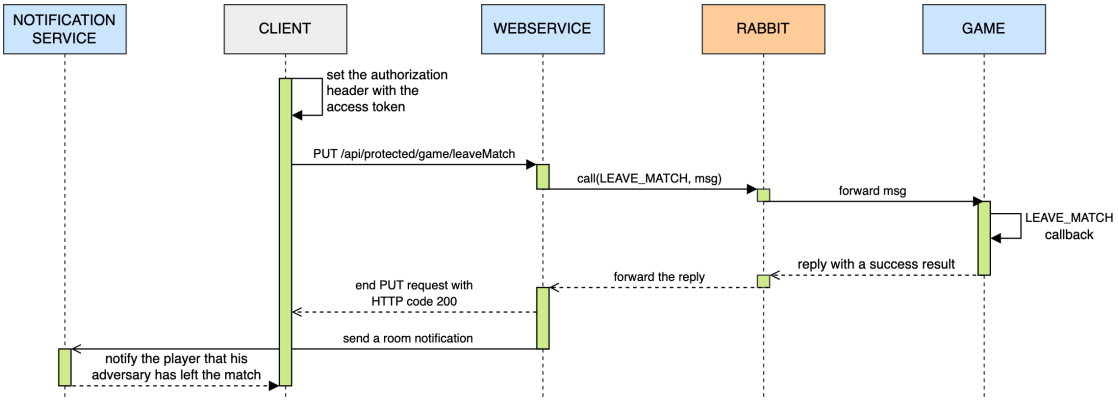


Figure 14: *Diagramma di sequenza per l'abbandono di una partita in corso*

Sottomissione di una mossa Questo scenario descrive l'interazione del sistema quando un utente, durante il processo di gioco, sottomette la propria mossa al server, in modo che possa essere verificata.

Ancora una volta, la struttura generale delle interazioni risulta molto simile ai casi precedenti. I cambiamenti risiedono solo nella callback invocata da `game`. Essa, infatti, per gestire adeguatamente il tentativo sottomesso, procede come segue:

- innanzitutto preleva dal database la partita per la quale si sta sottomettendo il tentativo, in base al suo ID e al fatto che deve includere come giocatore l'utente che ha appena giocato;
- successivamente, verifica che essa sia ancora in corso e che il turno attuale coincida con il giocatore che ha sottomesso il tentativo;
- a questo punto viene elaborato il tentativo, calcolandone i suggerimenti e controllando se è quello vincente;
- infine, la callback termina restituendo lo stato aggiornato della partita e i suggerimenti appena elaborati.

La gestione della richiesta torna ora a `webservice`, il quale deve impacchettare i risultati nel modo in cui se li aspetta il client, per poi restituirglieli terminando la richiesta HTTP. Come ultimo passaggio, il backend deve inviare una notifica all'avversario come feedback per quanto appena accaduto. Più nel dettaglio, possono essere inviate due tipologie di notifiche:

1. se il tentativo appena sottomesso era l'ultimo possibile oppure era quello vincente, la partita risulta ora conclusa e occorre inviare una notifica di tipo `MATCH_OVER`;
2. al contrario, se il tentativo non era né l'ultimo, né quello vincente, si procede ad inviare una notifica di `NEW_MOVE`, la quale serve a segnalare all'avversario che è arrivato il suo turno.

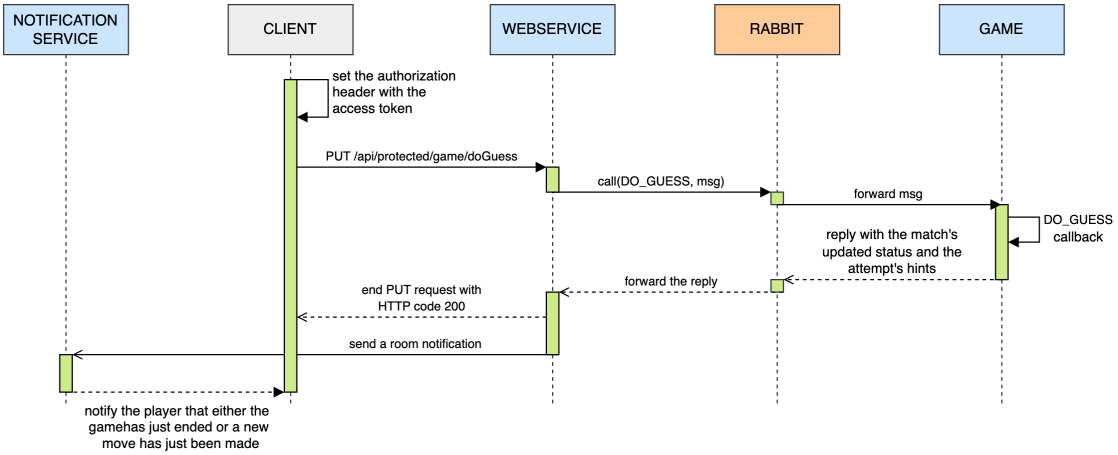


Figure 15: *Diagramma di sequenza per la sottomissione di un tentativo*

3.4.3 Interazioni lato impostazioni

Il sistema in oggetto consente di modificare molteplici valori (relativi al gioco e al profilo utente) ma dal momento che esse risultano tutte molto simili, verrà di seguito descritta l'interazione relativa ad un solo scenario.

Anche in questo caso, essendo queste rotte protette, il backend deve elaborare le richieste in arrivo solo se l'autenticazione JWT è andata a buon fine.

Modifica delle impostazioni Questo scenario descrive le interazioni necessarie per modificare le impostazioni di visualizzazione del gioco, ovvero la modalità dark e quella per daltonici.

Come di consueto, il client inizia la richiesta HTTP, ora di tipo PUT, includendo le impostazioni da aggiornare. Essa viene ricevuta da `webservice`, il quale provvede ad estrapolare i nuovi valori dal corpo della richiesta, per poi inoltrarli a `game`. Quest'ultimo, interagisce con il database per prelevare l'utente che ha richiesto la modifica e per salvare tale cambiamento. L'interazione si conclude con la terminazione della richiesta HTTP da parte di `webservice`.

4 Dettagli implementativi

In questa sezione verranno esplorati quegli aspetti di implementazione non banali e non ancora descritti in precedenza. In particolare, essi tratteranno: l'integrazione di RabbitMQ e la gestione delle comunicazioni WebSocket, entrambi dal punto di vista del backend, e la presentazione della specifica formale dell'API REST.

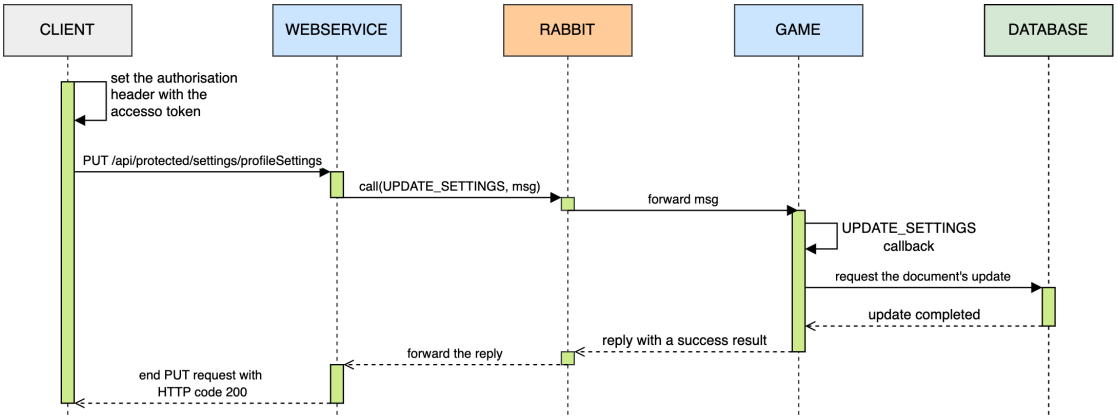


Figure 16: *Diagramma di sequenza per la disconnessione di un utente*

4.1 RabbitMQ

Volendo riprendere la struttura del sistema, è possibile ricordare il fatto che esso dispone di un backend composto da due microservizi. Per migliorare la scalabilità e rendere il sistema ancora più distribuito, si è deciso di sfruttare RabbitMQ per consentire ai due componenti di interagire.

Per incapsulare i dettagli implementativi relativi alla libreria Java del middleware in questione [14], si è deciso di sviluppare due classi che consentono di gestire la comunicazione in maniera robusta ed ottimale. Queste classi fungono da strati di comunicazione tra i microservizi, gestendo le connessioni, le code, gli exchange e i messaggi in modo efficiente e trasparente all'utilizzatore. In particolare, seguendo il pattern RPC, sono stati implementati un componente client e un componente server: il client invia le richieste tramite RabbitMQ e il server le elabora restituendo i risultati. Questa architettura offre numerosi vantaggi. Innanzitutto, consente ai microservizi di essere disaccoppiati, migliorando la manutenzione e facilitando l'evoluzione del sistema. Inoltre, RabbitMQ gestisce automaticamente il bilanciamento del carico e le code di messaggi, garantendo una distribuzione equa delle richieste e una risposta tempestiva anche sotto carichi elevati.

In questo contesto, si ritiene interessante presentare il comportamento delle classi sopra citate, che forniscono una conoscenza completa su come è stato impiegato RabbitMQ nel presente elaborato.

RPCServer Questa classe funge da server RPC, ovvero colui che ha il compito di attendere le richieste del client, elaborarle e restituire i risultati. Inoltre, definisce il metodo remoto invocato dal client. Questo componente si trova all'interno di `game` ed il suo comportamento può essere descritto come segue:

- per prima cosa viene aperta una nuova connessione con il server di RabbitMQ, che deve già essere in esecuzione;

- si procede poi con la creazione di un canale attraverso questa connessione. I canali sono utilizzati per la maggior parte delle interazioni con il middleware;
- il processo continua con la dichiarazione di un *exchange*, ovvero quell'entità che funge da intermediario tra chi produce i messaggi e chi li consuma. Tali messaggi vengono inviati tramite un criterio di instradamento specifico, determinato dal tipo di exchange configurato. Nel presente caso, si è optato per un exchange di tipo "direct" e ciò implica che i messaggi vengano instradati alle code in base a una chiave di routing. Qui, la chiave di routing corrisponde ai diversi tipi dei messaggi da gestire;
- a questo punto, per ogni callback definita nei controllers, ci si mette in ascolto sulla coda avente come chiave di routing il `MessageType` associato a tale metodo. Ciò significa che si avranno tante code quante sono le tipologie di operazioni gestibili;
- ora, il server RPC rimane in attesa dell'arrivo di messaggi in un loop infinito. Il risveglio viene implementato sfruttando il lock implicito presente in tutti gli oggetti Java;
- da questo momento in poi, l'arrivo di un messaggio su una delle code scatenerà l'invocazione della callback associata e la restituzione del relativo risultato su un'apposita coda di risposta.

RPCClient Questa classe rappresenta la controparte del server nel pattern RPC e si occupa di iniziare una richiesta, invocando il metodo remoto sul server, per poi attendere la risposta contenente i risultati. Volendo essere più specifici, questo componente si trova all'interno del microservizio `webservice` si comporta come segue:

- esattamente come accade per RPCServer, anche il client deve per prima cosa stabilire una connessione con il nodo RabbitMQ, la quale viene impiegata per creare un canale che consentirà di interagire con il middleware;
- dopo aver verificato che tale canale risulta effettivamente aperto, il passo successivo coincide con la creazione della coda sulla quale il client si aspetterà di ricevere il risultato, detta *replyQueue*;
- ora viene dichiarato l'exchange che si occuperà di instradare i messaggi, scegliendo anche qui la tipologia "direct". In questo passaggio è importante che ad esso venga associato lo stesso nome di quello usato in RPCServer. In caso contrario, i messaggi non arriverebbero a destinazione;
- a questo punto si attua la fase di invio della richiesta. Si sfrutta il canale creato all'inizio per pubblicare sull'exchange il corpo della richiesta, insieme al tipo di messaggio ad essa associato. Inoltre, vengono anche impostate due proprietà fondamentali, ovvero il nome della coda sulla quale il server dovrà inviare i risultati (*replyQueue*) e un *correlationId* usato per scopi di sicurezza;

- la fase finale consiste nel rimanere in ascolto su *replyQueue*, attendendo l'arrivo dei risultati della richiesta. Per ogni messaggio in arrivo su tale coda, viene eseguito un handler il quale verifica anzitutto che il correlationID ricevuto coincida con quello inviato insieme alla richiesta e, in caso affermativo, provvede ad eseguire un *responseConsumer*. Quest'ultimo viene specificato sotto forma di funzione lamda dalla classe RoutesController e consente di operare una prima elaborazione sui risultati della richiesta, comune a tutte le tipologie di messaggi.

4.2 Comunicazione WebSocket lato server

Arrivati a questo punto, è già ben noto che il backend interagisca con il frontend non solo per mezzo di richieste HTTP ma anche sfruttando WebSocket. Quest'ultimo viene impiegato per tutte quelle comunicazioni minori che devono essere recapitate in breve tempo e che generalmente non richiedono una risposta.

A corredo di quanto detto nella sezione 3.1.1, la quale descrive l'impiego di WebSocket lato frontend, qui si ha l'obiettivo di discutere in dettaglio il protocollo adottato dal backend per gestire correttamente la comunicazione con i clients.

Anzitutto è opportuno sottolineare che l'unico componente responsabile della comunicazione WebSocket è il Verticle NotificationService, all'interno del quale risiede il core del protocollo. Pertanto, ogniqualvolta il WebServer ha la necessità di inviare un messaggio ai clients, dovrà delegare tale Verticle, trasmettendogli la notifica da inoltrare.

Il componente NotificationService supporta sia le notifiche da server a client che quelle da client a server, garantendo una comunicazione bidirezionale che risulta essenziale per un'esperienza di gioco interattiva e reattiva. Questo è realizzato attraverso l'uso dell'eventbus di Vert.x, che facilita la gestione asincrona e distribuita degli eventi.

4.2.1 Notifiche da server a client

Le notifiche da server a client vengono gestite principalmente per informare i giocatori su eventi di gioco come l'inizio di nuove partite, le mosse dei giocatori e la fine delle partite. Essi, più nel dettaglio sono:

- `NEW_MATCH`: evento che si verifica quando viene creata una nuova partita. In particolare, non appena il sistema riesce a creare un match a partire da due giocatori, deve segnalare al client che ha generato la richiesta pendente del fatto che essa è stata soddisfatta e che la partita è stata creata con successo.

- `NEW_MOVE`: evento che si verifica quando viene sottomessa una mossa da parte di un giocatore. A titolo di esempio, si consideri il match tra i giocatori A e B, con A che risulta essere il prossimo a dover giocare. Non appena esso sottomette la propria mossa, il web server invia una notifica a B per comunicargli che l'avversario ha appena giocato ed è ora il suo turno.

- `MATCH_OVER`: evento che si verifica quando una partita viene abbandonata in maniera prematura. In particolare, se un giocatore decide di arrendersi può abbandonare la partita, concedendo la vittoria all'avversario. Questa segnalazione è

importante perché consente all'avversario di capire che quella partita si è conclusa in anticipo e che esso risulta automaticamente il vincitore.

- `CREATE_ROOM`: evento non facente parte di quelli di gioco, come i precedenti, ma è di vitale importanza per garantire un invio delle notifiche controllato e mirato. Viene gestito internamente al backend e non comporta l'invio di una notifica al frontend. Si verifica quando si ha la necessità di creare una stanza di comunicazione WebSocket e ciò accade in seguito alla creazione di un nuovo match.

Il NotificationService ascolta questi eventi (generati da WebServer) sull'indirizzo `WS_EVENTS_ADDRESS` dell'eventbus e, in base al tipo di evento ricevuto, invia notifiche mirate ai client interessati.

Questo approccio ha richiesto l'introduzione del concetto di *stanza* citato nel paragrafo 3.1.1, che viene ora discusso nel dettaglio come parte integrante del protocollo di comunicazione del backend.

Creazione di una stanza Le stanze vengono create per ogni match attivo. Quando NotificationService riceve l'evento `CREATE_ROOM`, viene invocato il metodo `createCommunicationRoom()` per processare la lista dei match attivi e creare le stanze appropriate.

```
private void createCommunicationRoom(JsonArray matches) {
    vertx.executeBlocking(() -> {
        matches.stream()
            .map(JsonObject.class::cast)
            .filter(match ->
                match.getJsonObject("matchStatus")
                .getString("matchState")
                .equals("PLAYING"))
            .forEach(activeMatch -> {
                JsonArray matchPlayers = activeMatch.
                   getJsonObject("matchStatus").getJsonArray
                   ("players");
                String activeMatchID = activeMatch.getString
                   ("_id");
                matchPlayersMap.computeIfAbsent(
                   activeMatchID, matchID ->
                    matchPlayers.stream()
                        .map(JsonObject.class::cast)
                        .map(player -> player.getString("
                           username"))
                        .collect(Collectors.toSet()));
            });
        return null;
    });
}
```

Questo metodo crea un oggetto `matchPlayersMap` che associa l'ID di ogni partita ai rispettivi giocatori. Solo i giocatori coinvolti in un match vengono aggiunti alla stanza di comunicazione di quel match.

Invio di notifiche su di una stanza Quando NotificationService riceve un evento di gioco viene invocato il metodo `notify()`, il quale ha il compito di inviare concretamente la notifica a tutti i clients appartenenti alla stanza del match in questione.

```
private void notify(String matchID, String originPlayer,
    JsonObject notificationData) {
     vertx.executeBlocking(() -> {
         matchPlayersMap.get(matchID).stream()
             .filter(playerInRoom -> !playerInRoom.equals(
                originPlayer))
             .forEach(playerToBeNotified -> vertx.eventBus()
                 .publish(WS_EVENTS_ADDRESS +
                    playerToBeNotified,notificationData.
                    encode()));
         if (notificationData.getString("notificationType").
            equals(MessageType.MATCH_OVER))
             matchPlayersMap.remove(matchID);
         return null;
     });
}
```

Questo metodo recupera l'insieme dei giocatori per un dato match dall'oggetto `matchPlayersMap` , filtra il giocatore originario dell'evento e invia la notifica solo agli altri giocatori della stanza.

Rimozione di una stanza La rimozione di una stanza avviene quando la partita ad essa associata termina. Ciò è sancito dall'evento `MATCH_OVER`, il quale scatena l'invocazione del metodo `notify()` che, dopo aver trasmesso la notifica, provvede anche a rimuovere la stanza da `matchPlayersMap`.

4.2.2 Notifiche da client a server

Oltre a inviare notifiche ai client, NotificationService gestisce anche i messaggi provenienti da essi. Questi possono essere visti come notifiche di stato in quanto consentono al server di sapere in tempo reale lo stato dei giocatori nel sistema e anche di propagare queste informazioni ai vari clients. Gli stati in cui può trovarsi un giocatore sono:

- ***online***: il giocatore risulta attivo, ovvero ha aperto i gioco ed effettuato l'accesso;
- ***offline***: il giocatore non è attivo. Questo si verifica se esso ha chiuso l'applicazione, se ha appena effettuato il logout oppure se ci sono dei problemi di connessione;

- ***playing***: il giocatore è attivo e si trova nella schermata di gioco.

Grazie a questa tipologia di notifiche, ogni giocatore può, in ogni momento, sapere lo stato dei propri avversari, capendo se essi sono attivi, se stanno giocando oppure se si sono disconnessi.

Per implementare questo meccanismo di propagazione dello stato si è ritenuto necessario definire una sorta di protocollo di comunicazione basato sull'eventbus di Vert.x. Questo protocollo stabilisce come i messaggi di stato vengono inviati dai client al server e come il server li elabora e redistribuisce agli altri client. Questo meccanismo è composto da tre scenari, che si verificano in seguito alla ricezione di specifici eventi: registrazione, aggiornamento e propagazione dello stato, disconnessione.

1. **Registrazione di un client**: quando un giocatore effettua l'accesso all'applicazione, il relativo client invia un messaggio di registrazione al server sull'indirizzo `WS_PLAYER_REGISTRATION`. Questo messaggio contiene il nome utente del giocatore e l'indirizzo della sua socket. Il server risponde con lo stato attuale di tutti i clients registrati fino a quel momento, permettendo al nuovo giocatore di avere una visione immediata della situazione corrente.

2. **Aggiornamento e propagazione dello stato**: i giocatori comunicano un cambiamento del loro stato inviando al server un messaggio sull'indirizzo `WS_PLAYER_STATUS`. Questo avviene quando, ad esempio, un client passa da *online* a *offline* o da *online* a *playing*. Questi cambiamenti di stato scatenano sul server un processo che prevede l'aggiornamento di alcune variabili interne e la propagazione della nuova informazione a tutti i clients registrati, escluso quello che ha dato origine al cambiamento.

3. **Disconnessione di un client**: quando un giocatore si scollega in maniera repentina, il server riceve una notifica sull'indirizzo `WS_PLAYER_DISCONNECTION` contenente il riferimento alla socket ad esso associata. Questo scenario si verifica in concomitanza con specifiche situazioni, quali un problema di rete lato client oppure la chiusura improvvisa dell'applicazione da parte dell'utente. In questo frangente, il server deve risalire all'utente disconnesso tramite la sua socket e ciò può esser fatto grazie ad uno specifico oggetto che associa il nome utente con gli indirizzi delle relative socket. Una volta individuato il giocatore, il server procede comunicando agli altri clients che esso non risulta più online.

4.3 API

L'interfaccia per le richieste HTTP messa a disposizione dal backend è già stata ampiamente citata nel corso delle sezioni precedenti. Tuttavia, non si possiede ancora una conoscenza approfondita per quanto riguarda le specifiche rotte, i metodi che supportano, la natura dei risultati, ecc. Questa sezione ha il compito di introdurre brevemente l'API REST in questione e di descriverne la struttura e le funzionalità per mezzo della specifica OpenAPI.

Nella radice della directory di progetto, all'interno della cartella *docs*, è presente un file chiamato `openapi.yaml` che rappresenta la specifica formale dell'API realizzata seguendo lo standard OpenAPI nella sua versione 3.0.3 [12].

Questa API funge da backend per l'applicazione Huesle, fornendo endpoint per la gestione degli account utente, delle operazioni di gioco, delle impostazioni e delle statistiche. Più nel dettaglio, essa include le seguenti funzionalità principali:

1. **Gestione utente**:
   - Registrazione di un nuovo utente,
   - Accesso e disconnessione degli utenti,
   - Aggiornamento dei token di accesso,
   - Eliminazione degli account utente.
2. **Gestione operazioni di gioco**:
   - Ricerca di partite (pubbliche o private),
   - Partecipazione a partite private,
   - Abbandono della partita corrente,
   - Effettuare tentativi nella partita.
3. **Impostazioni utente**:
   - Aggiornamento della password,
   - Aggiornamento dell'email,
   - Aggiornamento dell'immagine di profilo,
   - Recupero e aggiornamento delle impostazioni del profilo (es. modalità scura, modalità daltonici).
4. **Statistiche utente**:
   - Recupero delle statistiche dell'utente (partite vinte, perse, pareggiate).

L'API utilizza JSON Web Token [7] per l'autenticazione e include schemi di sicurezza per proteggere specifici endpoint. Definisce inoltre vari modelli di dati (come User, Match e SecretCode) per ridurre le ripetizioni di codice YAML e per migliorare la leggibilità della specifica.

5 Validazione

Per verificare il corretto funzionamento del sistema si è proceduto con un approccio ibrido, che ha visto l'impiego di tests di unità e di integrazione per la parte di backend, ed il coinvolgimento di piccoli gruppi di utenti per quanto riguarda il frontend. Essendo l'elaborato in oggetto rivolto principalmente agli artefatti che compongono il backend,

di seguito verranno discusse nel dettaglio solamente le modalità di testing relative a quest'ultima parte di sistema.

Anzitutto, come già accennato in precedenza, si è scelto di sfruttare il noto framework JUnit 5 per la scrittura e l'esecuzione di tutti i test relativi ai microservizi del backend. Questa scelta è stata dettata dal fatto che esso risulta un prodotto robusto, ricco di funzionalità e molto apprezzato nel mondo Java. Inoltre, si possedeva già una conoscenza delle sue capacità, cosa che ha ridotto notevolmente il tempo per la scrittura dei tests.

Dal momento che i due microservizi possiedono il loro sotto-progetto Gradle, è stato possibile separare i test del componente `game` da quelli di `webservice`, sia dal punto di vista "fisico" che da quello concettuale. Inoltre, sono stati verificati aspetti diversi nei test dell'uno rispetto a quelli dell'altro.

5.1 Correttezza di `game`

Gli aspetti testati in questo microservizio sono relativi alla logica di gestione degli utenti e a quella di gioco, ovvero tutto ciò che riguarda il funzionamento della parte core del backend. Sono stati previsti sia tests di unità, che verificano il corretto funzionamento di porzioni mirate di sistema, sia tests di integrazione, i quali verificano la correttezza di molteplici parti che lavorano insieme.

5.1.1 Unit Tests

I test di unità sono stati raccolti all'interno della classe GameLogicTests, la quale si concentra sulla correttezza della logica di gioco. Questi test verificano le condizioni iniziali di una partita, il calcolo dei suggerimenti in risposta a un tentativo di indovinare il codice segreto, la simulazione di un pareggio e di una vittoria, il corretto cambio di turno tra i giocatori, le restrizioni sui turni e la validità del codice segreto generato. In sostanza, i test verificano che le regole del gioco siano rispettate e che lo stato della partita sia correttamente aggiornato in risposta alle azioni dei giocatori. Tuttavia, essi astraggono completamente la presenza di un database e il fatto che vi è un altro microservizio che interagisce tramite RabbitMQ. Questi aspetti vengono trattati nei test di integrazione.

5.1.2 Integration Tests

Questa tipologia di test è essenziale in un sistema come questo, cioè provvisto di database e di un middleware che abilita l'interazione tra i microservizi. Infatti, mentre i test di unità si limitano a validare il funzionamento della logica di gioco in maniera isolata, i test di integrazione verificano che le procedure del backend abbiano gli effetti desiderati anche dal punto di vista del database e della comunicazione con il microservizio `webservice`.

Più nel dettaglio, questi tests sono stati raccolti e suddivisi nelle classi UserOperationTests, GameOperationTests e SettingsOperationTests, in modo da rispettare la divisione concettuale propria dei controllers. Esse operano introducendo un'astrazione rispetto al normale comportamento del sistema e lo fanno predisponendo un RPCClient che si occupa di iniziare le richieste da testare, simulando la presenza del server web.

In questo modo è anche possibile verificare il corretto funzionamento dello scambio dei messaggi per mezzo di RabbitMQ e del meccanismo del pattern RPC implementato.

Scendendo più nel concreto, le classi appena citate si occupano di testare i seguenti aspetti:

- **UserOperationTests**: esamina il corretto funzionamento dei processi di login, logout, cancellazione dell'utente e refresh del token di accesso. In particolare, i test al suo interno verificano che le risposte HTTP siano quelle attese, che le informazioni dell'utente siano correttamente memorizzate e recuperate dal database e che i token di accesso siano generati e gestiti correttamente. Inoltre, viene verificato che, dopo il processo di cancellazione, un utente non possa più effettuare il login.

- **GameOperationTests**: esamina il corretto funzionamento dei processi di creazione di una partita, di sottomissione dei tentativi per indovinare il codice segreto e di gestione del cambio di turno tra i giocatori. Inoltre, effettua una simulazione per quanto riguarda la vittoria, il pareggio e l'abbandono di una partita in modo da verificare che queste situazioni vengano gestite come previsto. In sostanza, questa suite di tests verifica aspetti molto simili rispetto a quelli di GameLogicTests, ma in questo caso ci si concentra sul corretto funzionamento del microservizio nel suo complesso, con un focus particolare sullo scambio di messaggi tramite RabbitMQ e sulla memorizzazione delle informazioni nel database.

- **SettingsOperationTests**: esamina la corretta gestione dei processi di reperimento delle impostazioni attuali e di aggiornamento di tali valori. In particolare, questi test verificano che le modifiche alle impostazioni siano correttamente memorizzate nel database e che le risposte HTTP siano quelle attese. Inoltre, viene verificato che, in caso di tentativo di aggiornamento con dati già presenti nel database (come ad esempio un indirizzo email già utilizzato), il sistema risponda opportunamente con un errore.

In questo contesto, l'aver incluso nel processo di testing le procedure di invio e ricezione delle richieste tramite RabbitMQ, si è portato con se un aspetto fondamentale: la serializzazione e deserializzazione dei dati trasmessi. Questi processi di conversione diventano essenziali per garantire una robusta interazione tra i microservizi e la loro correttezza viene verificata implicitamente in fase di esecuzione dei test di integrazione qui discussi. Infatti, se la serializzazione o deserializzazione non funzionasse come previsto, la comunicazione tra i microservizi sarebbe compromessa, portando a fallimenti nei test. Pertanto, pur non essendo l'obiettivo principale dei test in questione, i processi suddetti vengono indirettamente validati, contribuendo a garantire la robustezza e l'affidabilità dell'intero sistema.

5.2 Correttezza di `webservice`

I test fino ad ora descritti verificano la correttezza di tutti gli aspetti del sistema, ad eccezione del comportamento dell'API gateway. Ecco quindi che all'interno del mi-

croservizio in oggetto, sono stati previsti dei test per coprire proprio questo aspetto. In particolare, si è voluta verificare la correttezza del gateway nella gestione dei cookies e degli header per essere sicuri che il meccanismo di autenticazione JWT sia gestito correttamente.

Scendendo più nello specifico, gli scenari sottoposti a verifica sono i seguenti:

- **Login di un utente**: questo test simula lo scenario in cui un utente vuole effettuare l'accesso tramite le credenziali di un account preventivamente registrato. Viene quindi inviata una richiesta HTTP di tipo POST all'endpoint specifico, inserendo nel corpo della stessa un oggetto JSON con le credenziali. Non appena si riceve la risposta, viene verificato il fatto che il server abbia correttamente restituito il cookie contenente il refreshToken e che abbia inserito nel corpo della risposta l'accessToken relativo all'utente che si è appena autenticato. Se tutte queste condizioni vengono rispettate, significa che il processo di login lato web server risulta completato con successo.

- **Logout di un utente**: questo test replica la situazione in cui un utente desidera disconnettersi da un account al quale si è precedentemente autenticato. Per fare ciò, viene inviata una richiesta HTTP di tipo GET all'endpoint appropriato, includendo nell'header il cookie che contiene il refreshToken. Una volta ricevuta la risposta, si verifica che il server abbia rimosso correttamente il suddetto cookie e che abbia inviato un messaggio di conferma nel corpo della stessa.

- **Accesso ad una rotta protetta**: questo test simula lo scenario in cui un utente in possesso del suo accessToken voglia accedere ad una rotta dell'API che risulta protetta dall'autenticazione JWT. A scopo di verifica, è stata scelta come cavia per il test la rotta protetta "*/api/protected/game/getMatches*", la quale consente di ottenere tutti i match associati ad uno specifico utente, il cui username viene estrapolato dall'accessToken. Il processo di verifica inizia con l'invio della richiesta di tipo GET alla rotta sopracitata, avendo cura di includere l'header "Authorization Bearer" contenente il valore di un token di accesso valido e non ancora scaduto. La procedura deve ritenersi conclusa correttamente solo se il web server restituisce una risposta avente come codice di stato il valore 200 e come corpo i dati relativi ai match associati all'utente in questione.

- **Accesso ad una rotta protetta con un token scaduto**: questo test simula la situazione in cui un utente prova ad accedere a una rotta protetta con un token di accesso scaduto. In questo caso, il server web dovremme restituire una risposta di errore, avente come codice di stato il valore 403. Esso comunica al client che il token di accesso risulta scaduto e, pertanto, non può accedere alla rotta richiesta.

- **Accesso ad una rotta protetta senza autorizzazione**: questo test affronta un caso simile al precedente, ma con la differenza che in questo caso l'utente prova ad accedere alla rotta protetta senza fornire alcun parametro di autorizzazione, cioè senza specificare l'header "Authorization Bearer". In questa situazione, la richiesta inevitabilmente fallirà restituendo il codice di stato 500, che indica un

errore interno del server. Questo errore si verifica perché il server si aspetta di ricevere le informazioni di autorizzazione, che però non sono presenti, quindi il processo fallisce.

5.3 Autovalutazione

Il processo di autovalutazione ha rappresentato un aspetto cruciale per garantire la qualità e la robustezza del software sviluppato. Questo metodo ha permesso di riflettere criticamente sul lavoro svolto e di identificare eventuali punti di forza e di debolezza del sistema. L'autovalutazione si è articolata in diverse fasi, ognuna delle quali mirava a fornire un quadro completo e accurato sulle performance del software.

5.3.1 Analisi dei risultati dei test

La prima fase dell'autovalutazione ha riguardato l'analisi dei risultati ottenuti dai test di unità e di integrazione. Grazie all'utilizzo del framework JUnit 5, è stato possibile raccogliere informazioni dettagliate sull'esecuzione dei test, inclusi il numero di quelli superati, falliti e ignorati. Questi dati sono stati analizzati per individuare eventuali pattern di fallimento e per comprendere le aree del sistema che necessitavano di ulteriori miglioramenti. In particolare, si è prestata attenzione ai test di integrazione, data la loro importanza nel verificare la corretta interazione tra i microservizi.

Volendo fornire qualche dato concreto in merito a questo aspetto, è possibile affermare che:

- nell'ambito di `game`, il numero totale di test scritti è pari a 42, con 41 test passati con esito positivo ed 1 test ignorato. Quest'ultimo, è stato scritto solamente per capire la velocità di generazione dell'hash di una password, in base al valore del costo passato come input;
- nell'ambito di `webservice`, invece, il numero di test scritti è pari a 5, così come il numero di quelli passati con successo.

5.3.2 Valutazione della copertura del codice

Un altro aspetto fondamentale nel processo di autovalutazione è stata l'analisi della copertura del codice. Attraverso lo strumento JaCoCo [5], è stato possibile ottenere una stima precisa sulla percentuale di codice coperta dai test. L'obiettivo era raggiungere una copertura superiore al 80% per garantire che la maggior parte delle funzionalità del sistema fosse adeguatamente testata.

Dal momento che il backend del sistema è composto da due microservizi, ognuno dei quali possiede specifici test, è stato necessario analizzare la copertura di entrambi i componenti per ottenere un quadro completo della situazione. Per quanto riguarda `game` si è ottenuta una coverage pari all'82%, mentre per `webservice` solamente del 49%. Quest'ultimo valore risulta così basso in quanto la maggior parte delle funzionalità del microservizio in questione sono state testate direttamente sul campo, tramite l'uso

dell'applicativo. Sono stati sottoposti a test solo gli aspetti più critici, che risultano essere in minoranza rispetto alla moltitudine di funzionalità implementate.

5.3.3 Revisione del codice e refactoring

Durante l'autovalutazione è stata condotta una revisione del codice per assicurarsi che fosse conforme alle migliori pratiche di programmazione e agli standard di codifica. Questo processo ha incluso la verifica della leggibilità del codice, la sua manutenibilità e la presenza di eventuali code smell. Dove necessario, sono stati effettuati interventi di refactoring per migliorare la qualità del codice, riducendo la complessità e migliorando la chiarezza e la coerenza.

5.3.4 Feedback degli utenti

Il coinvolgimento di piccoli gruppi di utenti nel processo di testing del frontend ha fornito un prezioso feedback sull'usabilità e sull'esperienza d'uso del sistema. Questo feedback è stato raccolto attraverso sondaggi e sessioni di test guidate, e ha permesso di identificare problemi non evidenti nei test automatizzati. I miglioramenti suggeriti dagli utenti sono stati valutati e, dove possibile, integrati nel sistema attraverso cicli di sviluppo iterativi.

6 Istruzioni per il deploy

Per mandare in esecuzione tutti gli artefatti software del progetto in questione si è fatto uso del meccanismo di *containerizzazione* messo a disposizione da Docker. Ogni microservizio possiede il proprio container costruito a partire dal relativo `Dockerfile` e tramite l'estensione `docker-compose` è possibile automatizzare la build, la configurazione e l'avvio di tutte le immagini necessarie per mandare in esecuzione il sistema.

In particolare, ecco i diversi container che sono stati previsti per il corretto deployment dell'elaborato in questione:

1. **huesle-database**: rappresenta il database MongoDB essenziale per la persistenza dei dati. Esso, sfrutta l'immagine Docker ufficiale messa a disposizione dai produttori della base di dati;

2. **huesle-rabbitmq**: rappresenta il server RabbitMQ, essenziale affinché i microservizi del backend siano in grado di comunicare tra di loro. Anch'esso sfrutta l'immagine Docker ufficiale messa a disposizione dai produttori del MOM;

3. **huesle-client**: è il container che funge da punto di ingresso all'applicazione, esponendo l'URL per il collegamento al frontend. Viene costruito utilizzando un Dockerfile specifico, basato sull'immagine `node:18-alpine3.18`. All'interno del container, vengono installate tutte le dipendenze tramite `npm` e viene creata la *production build* di tutto il codice JavaScript del frontend. Infine, tutti i file della Single-Page Application vengono serviti sulla porta 3000 utilizzando il tool `serve`.

4. **huesle-game**: container che ospita il microservizio `game` del backend. Viene costruito a partire da un Dockerfile apposito, il quale si basa sull'immagine `gradle:8.6-jdk21` e sfrutta il tool Gradle per compilare ed avviare il codice Java al suo interno.

5. **huesle-webservice**: questo container ospita il microservizio `webservice`, che funge da API gateway. Ha una struttura e un comportamento simili a quelli del container precedente, ma in più espone la porta 8080 all'esterno, consentendo ai client di accedere all'API.

Tutti i container appena descritti non vengono creati e mandati in esecuzione manualmente. Bensì, il progetto è fornito del file `docker-compose.yml` che automatizza completamente il processo di deploy e lo riduce al solo comando

```
docker compose up
```

6.1 Dipendenze temporali

Nell'integrazione dei vari servizi, è stato cruciale introdurre dipendenze temporali. Infatti, affinché i componenti del backend possano operare correttamente, è essenziale che il servizio RabbitMQ sia attivo e pronto a ricevere messaggi. Nonostante docker-compose permetta di definire l'ordine di avvio dei container tramite l'istruzione `depends_on`, questa non verifica se l'applicazione al loro interno è realmente pronta. Di conseguenza, è stato necessario implementare uno script Bash come entrypoint per i container huesle-game e huesle-webservice. Questo script utilizza *netcat* per verificare l'esistenza di un processo in ascolto sulla porta standard di RabbitMQ, confermando così la disponibilità del servizio. Solo a quel punto il container può avviare il suo task specifico.

Di seguito, viene riportato lo script appena discusso:

Listing 1: waitRabbitThenStart.sh

```
#!/bin/bash

is_service_listening() {
    local service_host="$1"
    local service_ports="$2"
    nc -zw3 "$service_host" "$service_ports"
}

until ( ( is_service_listening "${RABBIT_HOST}" "5672" ) );
   do
    echo "Service isn't listening yet. Waiting..."
    sleep 5
done

echo "Services are now listening on the corresponding ports!
   "
```

```
echo "Starting the task $SERVICE with gradle"
gradle "$SERVICE"
```

7 Esempi d'uso

Di seguito vengono riportati alcuni esempi di utilizzo dell'applicazione Huesle con l'obiettivo di mostrare e descrivere la sua interfaccia e le varie procedure di interazione. Gli scenari proposti riprendono quelli già discussi nella sezione 1.2.

7.1 Utente

In questa sezione vengono mostrati tutti i possibili casi d'uso dal punto di vista di un utente.

Quando si avvia l'applicazione, visitando l'URL del server sul quale essa risiede, ci si trova davanti ad una pagina di benvenuto (figura 17) che mostra tutte le possibili azioni che può compiere un utente prima di effettuare l'accesso.

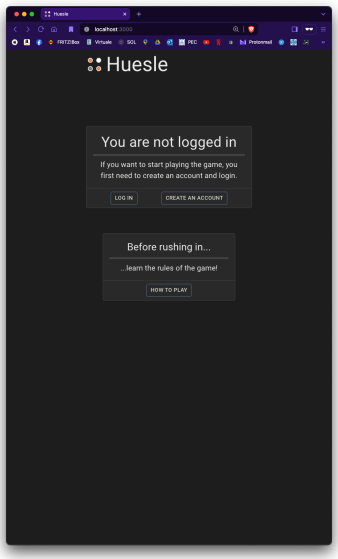
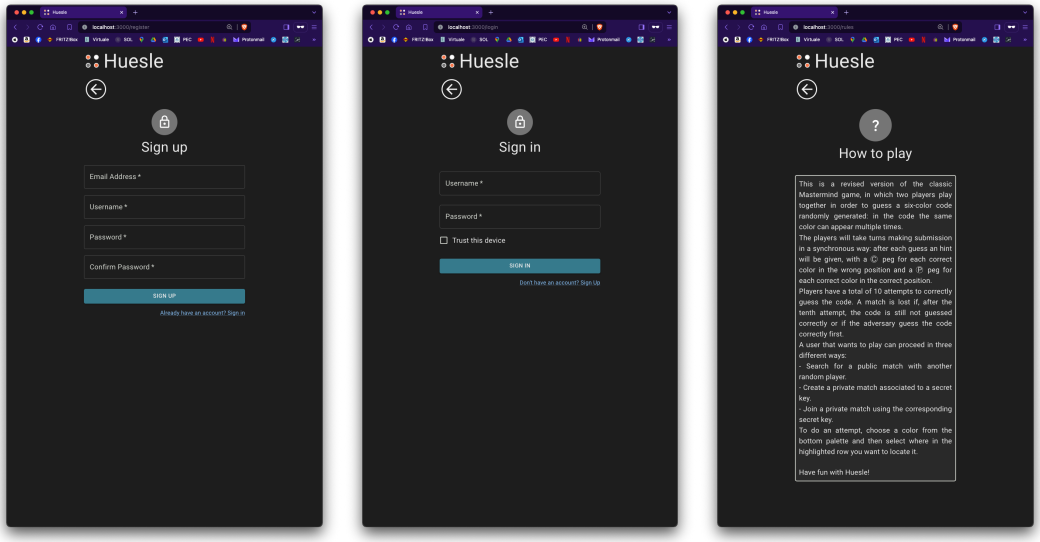


Figure 17: *Schermata di benvenuto*

In figura 18, possono essere visualizzate le schermate relative alle procedure di registrazione ed autenticazione, così come quella contenente le regole del gioco.

In figura 19a vi è la schermata relativa al profilo del giocatore che consente di accedere alla pagina di modifica di alcuni dati utente, di effettuare il logout e persino di eliminare l'account corrente. Inoltre, in essa è visualizzata anche una breve statistica sull'esito delle partite passate per l'utente in questione.

La schermata per la modifica dei dati utente è visualizzabile in figura 19b e consente di cambiare l'immagine del profilo, l'indirizzo email e la password.



(a) *Schermata di registrazione* (b) *Schermata di accesso* (c) *Regole del gioco*

Figure 18: *Schermate relative alle azioni eseguibili senza aver effettuato l'accesso*

Per quanto riguarda le impostazioni dell'applicazione, per ora si limitano soltanto alla possibilità di abilitare o disabilitare la modalità scusa e lo schema colori adatto alle persone affette da daltonismo. In figura 20 si vede la schermata in discussione, in entrambe le versioni del tema.

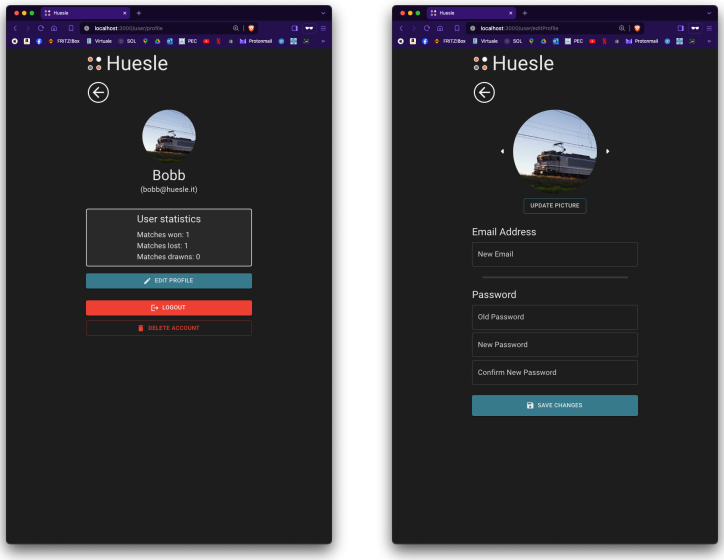
7.2 Giocatore

Una volta che un utente effettua l'accesso tramite le proprie credenziali, diventa un giocatore e viene portato alla schermata principale dell'applicazione, detta dashboard. Essa, visionabile in figura 21, mostra la lista delle partite in corso, lo storico di quelle concluse e lo stato degli avversari. Inoltre, nella parte bassa è presente una barra di navigazione, comune ad altre schermate, la quale consente di accedere al profilo dell'utente e alla pagina per modificare le impostazioni.

Se si clicca sulla riga in corrispondenza di una delle partite mostrate nella dashboard, si apre una pagina che mostra la panoramica di tale partita. Come si vede in figura 22a, tale pagina mostra i nomi e l'immagine di profilo dei due sfidanti e poi un riassunto dell'esito dei rounds che sono già stati completati. In più, presenta il bottone per andare alla schermata di gioco e quello per abbandonare prematuramente la partita.

La figura 22b ritrae la schermata di gioco, composta dai tentativi fatti fino a quel momento e dal pannello di interazione. Quest'ultimo, è composto dall'insieme di colori selezionabili e dal pulsante per sottomettere il tentativo corrente.

Per quanto riguarda la creazione di una nuova partita, l'iter differisce leggermente a seconda che si vuole giocare ad una partita pubblica o a una privata. In entrambi i casi,



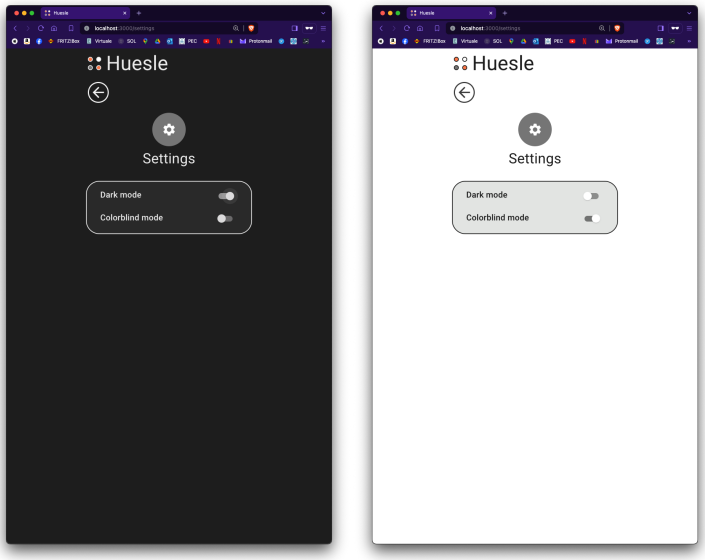
(a) *Profilo utente e statistiche* (b) *Modifica dati utente*

Figure 19: *Schermate del profilo e di modifica dei dati utente*

per accedere a questa funzionalità del gioco, bisogna essere sulla dashboard e cliccare sul pulsante "SEARCH MATCH", situato nella parte alta della stessa. Questa azione porta l'utente nella schermata dalla quale è possibile scegliere quale tipologia di partita si vuole cercare.

1. **Partita pubblica**: questo tipo di partita si avvia con due giocatori qualsiasi, ossia i primi due utenti che desiderano giocare. Le azioni necessarie per creare una partita pubblica sono le seguenti. Si supponga di avere i due utenti A e B, attivi nel sistema ed entrambi con la volontà di giocare. L'utente A inizia selezionando l'opzione per cercare una partita pubblica dalla schermata corrente. Se non ci sono altri utenti che hanno manifestato prima la volontà di giocare con le stesse azioni di A, quest'ultimo viene riportato nella dashboard con l'indicazione che vi è una partita pendente in attesa dell'avversario. Successivamente, l'utente B, desiderando unirsi a un match pubblico, seleziona la stessa opzione di A. Questa volta, però, il sistema rileva che esiste già una richiesta pendente per una partita pubblica. Pertanto, esso procede creando una nuova partita con gli utenti A e B, notificando entrambi di questo aggiornamento. Il processo si conclude con i giocatori che vengono riportati nella dashboard con la possibilità di accedere al match appena creato e iniziare a giocare.

2. **Partita privata**: per la creazione di una partita privata, che è un incontro tra due giocatori specifici, il processo segue questi passaggi. L'utente A seleziona l'opzione per avviare una partita privata. Il sistema genera un codice casuale di 5 cifre e lo visualizza in una finestra di dialogo. L'utente A deve quindi comunicare



(a) *Tema scuro* (b) *Tema chiaro*

Figure 20: *Schermata delle impostazioni*

questo codice all'utente B, rimanendo in attesa fino a quando quest'ultimo inserisce il codice fornito. Durante questo periodo, A può interrompere la ricerca di un avversario in qualsiasi momento. Quando l'utente B decide di unirsi al match inserendo il codice, il sistema avvia la partita, notificando entrambi gli utenti e riportandoli nella dashboard con la possibilità di accedere al match creato e iniziare a giocare.

L'iter descritto al punto 1 è visionabile graficamente in figura 23, mentre quello discusso al punto 2, in figura 24

8 Conclusioni

Il progetto di riscrittura del backend del sistema Huesle ha rappresentato un passo significativo verso l'ottimizzazione e l'evoluzione della piattaforma. Durante il processo, è stato mantenuto il frontend esistente, garantendo così continuità per l'esperienza utente, mentre il backend è stato trasformato da JavaScript a Java, con l'obiettivo di affinare le capacità di realizzazione di un'architettura distribuita, indipendentemente dalle tecnologie impiegate.

La nuova soluzione Java rappresenta un notevole miglioramento rispetto al backend originale in JavaScript. Questo progresso si manifesta sotto vari aspetti chiave. Innanzitutto, la gestione dei tipi di dato è stata significativamente potenziata grazie alla forte tipizzazione di Java, che riduce gli errori di tipo e migliora l'affidabilità del codice. Inoltre, la modularità del codice è stata notevolmente migliorata, permettendo una separazione

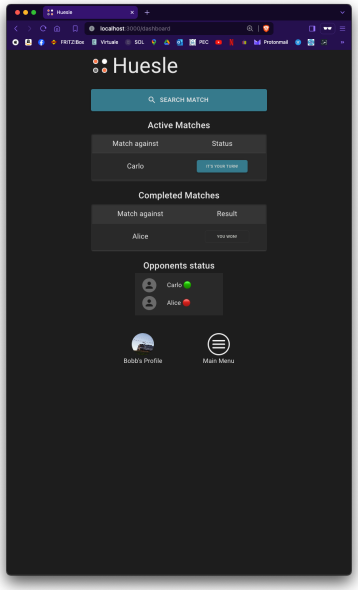


Figure 21: *Schermata della dashboard*

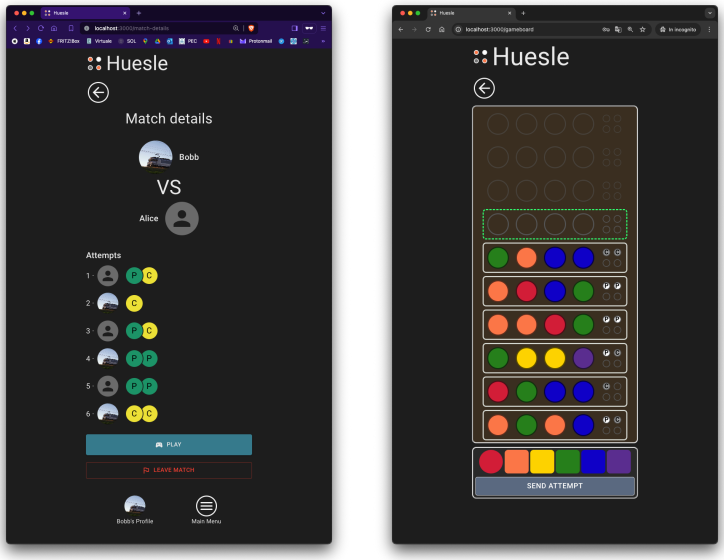
più chiara delle responsabilità e facilitando l'estensione e la manutenzione del sistema. Infine, l'adozione delle buone pratiche di programmazione ha portato a un'architettura più solida e ben strutturata, rendendo il sistema più robusto e scalabile. Questi miglioramenti combinati assicurano che il nuovo backend non solo soddisfi tutti i requisiti attuali, ma sia anche pronto per le sfide future.

Tuttavia, il processo di riscrittura ha presentato diverse sfide. Una delle principali difficoltà è stata garantire un'integrazione fluida tra il nuovo backend e il frontend esistente, evitando interruzioni per gli utenti. Questo ha richiesto una rigorosa analisi della situazione iniziale, seguita da una scrupolosa fase di testing e debug, entrambe fondamentali per assicurare una transizione senza problemi.

8.1 Lavori futuri

Sebbene ci si possa ritenere pienamente soddisfatti del risultato ottenuto, è comunque possibile individuare diverse aree che potrebbero essere coinvolte in un futuro processo di miglioramento del sistema, sia dal punto di vista del frontend che da quello del backend.

- **Chat tra giocatori:** implementare una funzionalità di chat che permetta ai giocatori di comunicare tra loro durante la partita. Questo può migliorare l'interazione sociale e rendere l'esperienza di gioco più coinvolgente.
- **Ampliamento della suite di test:** aumentare la quantità dei tests esistenti in modo da massimizzare il numero di funzionalità verificate. Ad esempio, si potrebbero aggiungere procedure che controllano la correttezza di tutti i (de)serializzatori



(a) *Panoramica di una partita* (b) *Schermata di gioco*

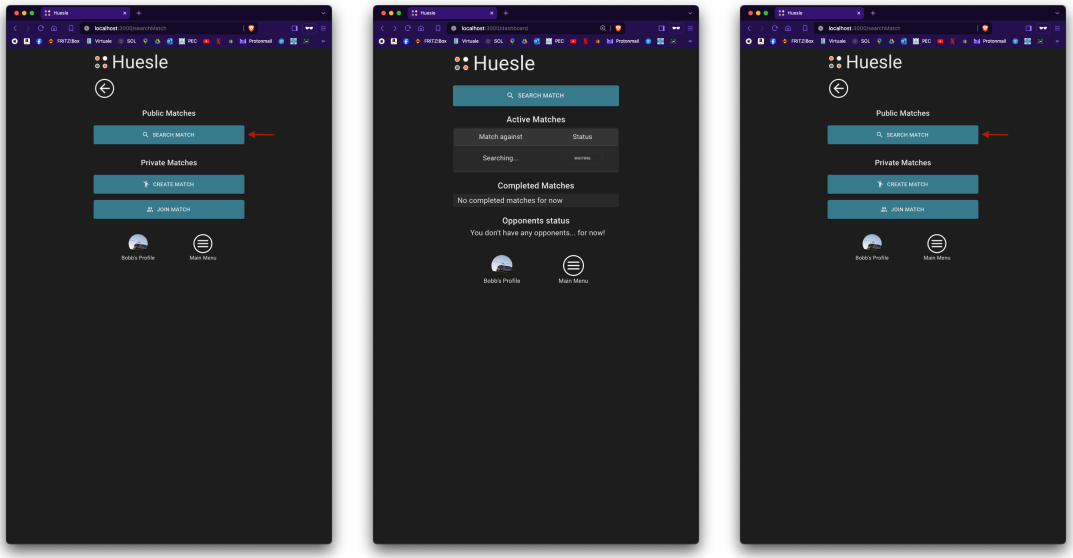
Figure 22: *Schermate relative ad una partita in corso*

presenti in `game` o ancora prevedere dei test per verificare il regolare funzionamento degli endpoint WebSocket. Questi ultimi dovrebbero verificare la correttezza delle comunicazioni in tempo reale tra client e server, garantendo che gli eventi di gioco siano trasmessi correttamente e che le connessioni siano gestite in modo robusto.

- **Scelta della difficoltà di gioco:** aggiungere opzioni di personalizzazione della difficoltà di gioco, permettendo ai giocatori di selezionare la lunghezza del codice da indovinare. Questa funzionalità consentirebbe di adattare il gioco a diversi livelli di abilità e preferenze dei giocatori, rendendo l'esperienza più varia e stimolante.

- **Partite a tempo:** introdurre la possibilità di creare delle partite nelle quali ogni turno ha un limite di tempo entro cui il giocatore deve fare la sua mossa. Questo può aumentare la tensione e la sfida del gioco, costringendo i giocatori a pensare rapidamente e a gestire meglio il loro tempo.

- **Notifiche sugli eventi di gioco:** implementare un sistema di notifiche che informi i giocatori degli eventi di gioco avvenuti mentre non erano online. Questo sistema dovrebbe inviare aggiornamenti sulle mosse degli avversari, sui risultati dei turni e su altri eventi rilevanti, assicurando che gli utenti siano al corrente sul progressi delle partite una volta che si ricollegano al gioco.



(a) *Azione dell'utente A* (b) *Partita pendente di A* (c) *Azione dell'utente B*

Figure 23: *Iter di ricerca e creazione di una partita pubblica*

8.2 Cosa è stato imparato

L'implementazione del sistema in oggetto ha permesso di raggiungere una comprensione elevata ed approfondita delle caratteristiche e delle problematiche di un sistema distribuito. Inoltre, il progetto ha consentito di esplorare in maniera approfondita le potenzialità delle tecnologie Docker, Vertx e RabbitMQ. In particolare, il primo, ha reso il deployment dei vari microservizi semplice e conciso, riducendo la necessità di passaggi e configurazioni complesse. Il secondo, ha permesso di implementare un server web asincrono ed efficiente in maniera semplice e modulare. Infine, il terzo, è stato essenziale per implementare l'architettura a microservizi, consentendo il disaccoppiamento dei vari componenti e l'uso delle tecnologie più pertinenti per ciascuno

La conoscenza acquisita riguardo queste tecnologie sarà sicuramente utile anche per progetti futuri.

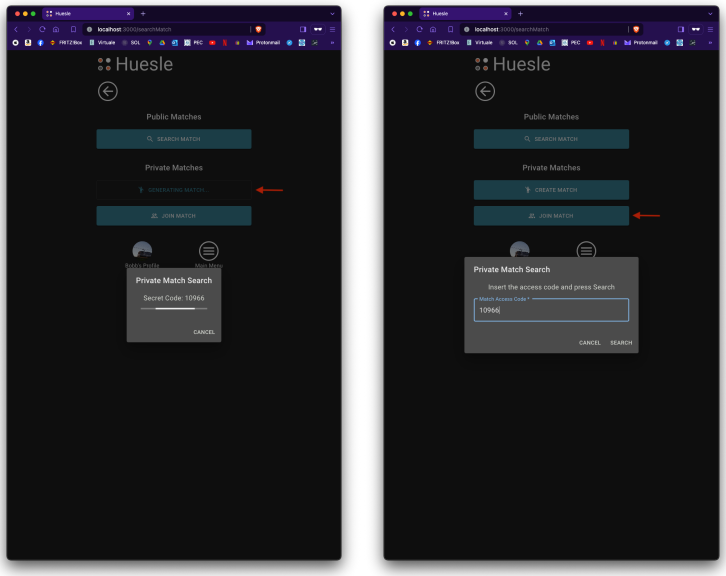
References

[1] Mastermind. `https://en.wikipedia.org/wiki/Mastermind_(board_game)`.

[2] Cap theorem. `https://en.wikipedia.org/wiki/CAP_theorem`.

[3] eventbus bridge client. `https://github.com/vert-x3/vertx-eventbus-bridge-clients/tree/master/javascript/`.

[4] Gradle build tool. `https://gradle.org/`.



(a) *Azione dell'utente A* (b) *Azione dell'utente B*

Figure 24: *Iter di creazione e unione ad una partita privata*

[5] Jacoco. `https://www.eclemma.org/jacoco/`.

[6] Junit 5. `https://junit.org/junit5/`.

[7] Introduction to json web tokens. `https://jwt.io/introduction`.

[8] Material ui. `https://mui.com/`.

[9] Mongodb. `https://www.mongodb.com/`.

[10] Mongodb java driver. `https://www.mongodb.com/docs/drivers/java/sync/current/`.

[11] Window: offline event. `https://developer.mozilla.org/en-US/docs/Web/API/Window/offline_event`.

[12] Openapi 3 specification. `https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.3.md`.

[13] Rabbitmq. `https://www.rabbitmq.com/`.

[14] Libreria rabbitmq per java. `https://mvnrepository.com/artifact/com.rabbitmq/amqp-client`.

[15] React router. `https://reactrouter.com/en/main`.

[16] Remote procedure call. https://www.ionos.com/digitalguide/server/know-how/what-is-a-remote-procedure-call/.

[17] Sockjs. https://github.com/sockjs/sockjs-client.

[18] Libreria vertx-auth-jwt. https://vertx.io/docs/vertx-auth-jwt/java/.

[19] Libreria vertx-web. https://vertx.io/docs/vertx-web/java/.

[20] Websocket. https://en.wikipedia.org/wiki/WebSocket.