Lupus
Final Report

Andrea Serafini
andrea.serafini11@studio.unibo.it

November 2023

Questo progetto nasce con l'obiettivo di creare una versione digitale del gioco Lupus, conosciuto anche come Mafia [1], un gioco conosciuto ormai da tutti, molto popolare tra i gruppi di universitari.

Il gioco prevede che ci siano almeno 6 giocatori e un narratore esterno che orchestri la partita. Prima di iniziare la partita, il narratore assegna casualmente un ruolo a ciascun giocatore. Generalmente questa assegnazione viene effettuata con l'ausilio di carte da gioco o token fisici, poiché è fondamentale per lo svolgimento che i ruoli rimangano segreti per tutta la partita. I ruoli presenti nel gioco possono variare a seconda del numero di giocatori o del tipo di partita che si vuole effettuare.

Durante lo svolgimento del gioco ci sarà un alternarsi di fasi diurne e notturne nelle quali i diversi personaggi potranno intraprendere azioni specifiche. Il narratore avrà il compito di supervisionare l'alternarsi delle fasi di gioco, tenere traccia delle decisioni dei giocatori e delle loro azioni.

Nella versione originale e più semplice del gioco i ruoli sono quelli di *cittadino* e *mafioso*, per cui si andranno a creare due squadre con obiettivi contrapposti, identificare i mafiosi e uccidere i cittadini. A ogni turno di notte il narratore dovrà chiedere ai mafiosi chi vogliono uccidere all'insaputa dei cittadini, mentre durante il turno di giorno tutti i giocatori dovranno decidere chi accusare di essere un mafioso, consapevoli però che i mafiosi cercheranno di influenzare a loro favore la decisione.

La partita termina nel momento in cui tutti i giocatori di una delle due squadre hanno la meglio sugli altri, avendo ucciso tutti i cittadini o imprigionato tutti i mafiosi. A questo punto sarà sufficiente distribuire nuovamente i ruoli per iniziare una nuova partita.

Indice

Elenco delle figure

1 Analisi preliminare

1.1 Obiettivi

L'obiettivo finale del progetto è quello di creare un applicativo che, basandosi su un'architettura Client-Server ibrida, permetta di facilitare il ruolo di narratore nel gioco descritto in precedenza. Le specifiche che seguiranno derivano da un'analisi preliminare del problema e non saranno quindi definitive, ma potranno essere riviste durante la fase di sviluppo.

Il sistema distribuito si baserà su di un'architettura *peer-to-peer* con discovery server per la gestione della singola partita, durante la quale tutti i Client partecipanti, in maniera mista tra Web e Mobile, condivideranno le informazioni di gioco mantenendo così uno stato aggiornato e coerente tra loro. Il Server avrà poi un ruolo più centrale per quanto riguarda la persistenza del sistema, grazie alla realizzazione di un database, occupandosi di salvare credenziali e informazioni relative agli utenti, oltre che informazioni sui risultati di gioco ed eventuali statistiche. Infine durante il progetto si andrà a familiarizzare con i *container* di Docker con l'obiettivo di facilitare il deployment del sistema.

1.2 Deliverables

Al termine dello sviluppo sono attesi i seguenti artefatti:

- database
- server
- client WEB
- client mobile

Le tecnologie che si ipotizza di utilizzare per lo sviluppo di questi sono: MongoDB per quanto riguarda il database, Node.js e in particolare il framework Express.js per l'implementazione del Server, e per il Client la libreria Javascript React per lo sviluppo della versione Web in combinazione con React Native per la versione Mobile. Infine verrà utilizzato Gradle per quanto riguarda la building automation.

1.3 Work plan

Essendo un progetto individuale non sarà possibile parallelizzare le attività, per cui si procederà a uno sviluppo incrementale delle componenti del sistema, alternando fasi di implementazione e di testing, suddividendo le iterazioni per funzionalità nel seguente modo:

- sistema di gioco peer-to-peer (web client) con discovery server
- persistenza e storico tramite DB

- (opzionale) mobile client

Prima di avviare la fase di implementazione si svolgerà una fase di analisi approfondita e progettazione del sistema nella quale verranno definite le specifiche di funzionamento dell'applicativo e i suoi requisiti, oltre che ai dettagli dell'architettura stessa.

2 Analisi dei requisiti

Nelle sezioni seguenti verranno elencati i requisiti identificati per il progetto, suddivisi tra *funzionali* e *non funzionali*, così da avere a disposizione uno strumento di monitoraggio per il processo di sviluppo.

2.1 Requisiti funzionali

L'utente finale dovrà avere la possibilità attraverso il sistema di compiere una serie di azioni, raccolte nel seguente elenco:

- accedere al sistema tramite credenziali univoche
- creare un nuovo *party* [2]
- unirsi a un *party* esistente
- finalizzare il *party* e accedere alla *lobby* [3]
- (opzionale) selezionare la tipologia di partita
- avviare la partita
- selezionare un giocatore da uccidere/accusare
- visualizzare il risultato a partita terminata
- (opzionale) salvare il risultato

2.2 Requisiti non funzionali

Il sistema dovrà poi avere le seguenti caratteristiche:

- il sistema dovrà essere scalabile
- l'architettura ibrida dovrà essere gestita in maniera efficiente
- la comunicazione dovrà essere "sicura" tra i peers e il server
- il sistema dovrà essere in grado di gestire in maniera corretta la connessione e disconnessione degli utenti durante il gioco

2.3 Tecnologie e paradigmi

La maggior parte delle tecnologie impiegate per lo sviluppo di questo progetto sono state definite nelle prime fasi di design, oltre che in fase di proposta dello stesso. Alcune librerie o specifici framework invece sono stati il risultato di scelte nate durante la fase di implementazione. Per lo sviluppo e l'implementazione del sistema è stato scelto lo stack MERN, una variazione dell'originale stack MEAN [4] nella quale Angular viene sostituito da React. Questa architettura permette di creare in maniera semplificata una struttura su tre livelli (*front end, back end, database*) utilizzando solamente JavaScript e JSON.

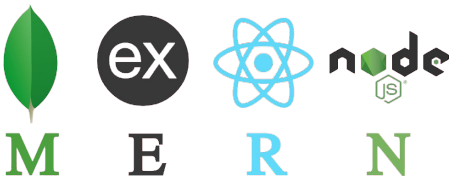

Figura 1: Tecnologie stack MERN

Qui di seguito verranno descritte alcune tra le principali tecnologie utilizzate suddividendole tra quelle trasversali a tutto il progetto e quelle specifiche delle componenti di server, database e client.

2.3.1 Tecnologie trasversali

NPM Node.js [5] è un *runtime system* multi piattaforma per l'esecuzione di codice JavaScript, costruito sul motore JavaScript V8 di Google Chrome. Node.js dispone di una grande quantità di moduli scritti completamente in Javascript. Essendo il progetto open-source è inoltre possibile per gli sviluppatori aggiungere i propri moduli in modo da renderli disponibili pubblicamente.

Il gestore di pacchetti predefinito per l'ambiente si chiama Node Package Manager. *npm* può essere richiamato tramite linea di comando usando la seguente sintassi:

```
npm <command> [args]
```

Il comando base per ottenere un pacchetto è:

```
npm install packet_name
```

Tutte le dipendenze e i conflitti vengono gestiti automaticamente [6]. Grazie a Node è anche possibile per esempio creare un progetto React utilizzando il comando seguente:

```
npx create-react-app project
```

npx è uno strumento integrato in npm in grado di eseguire pacchetti, anche se non sono ancora installati nel sistema. Sarà poi possibile avviare il server di sviluppo utilizzando i seguenti comandi:

```
cd project
npm start
```

L'interfaccia sarà poi visualizzabile all'indirizzo *http://localhost:3000*.

Figura 2: npm logo

Socket.io Socket.io è una libreria JavaScript che fornisce un servizio molto simile alle originali Web-Socket. Socket.io offre delle API JavaScript cross-browser che permettono la creazione di un canale di comunicazione full-duplex tra domini a bassa latenza tra il browser e il server web.

Socket.io tenta di utilizzare prima di tutto le WebSocket native. In caso fallisca (in caso di incompatibilità coi sistemi o a causa di altri problemi) ricorrerà all'utilizzo di polling HTTP.

Socket.io è stato progettato per funzionare con tutti i browser moderni (97% di compatibilità nel 2020) e in ambienti che non supportano il protocollo WebSocket, ad esempio dietro proxy aziendali restrittivi.

Oltre alle funzionalità offerte da una tradizionale WebSocket, Socket.io offre inoltre le seguenti features:

- reliability (switch a polling HTTP nel caso la connessione WebSocket non possa essere stabilita)
- riconnessione automatica
- buffering dei pacchetti
- acknowledgments
- broadcast a tutti i client o a un sottoinsieme di essi (Room)
- multiplexing

Figura 3: Socket.io logo

Docker Docker è un sistema open-source tramite il quale è possibile automatizzare il processo di deployment di applicazioni all'interno di contenitori software. Docker implementa API di alto livello per gestire *container* che eseguono processi in ambienti isolati.

Utilizzando i container dunque le risorse possono essere isolate, i servizi limitati e i processi avviati in modo da avere una prospettiva completamente privata del sistema operativo, col loro proprio identificativo, file system e interfaccia di rete. Più container condividono lo stesso kernel, ma ciascuno di essi può essere costretto a utilizzare una certa quantità di risorse, come la CPU, la memoria e l'I/O.

L'utilizzo di Docker per creare e gestire i container può semplificare la creazione di sistemi distribuiti, permettendo a diverse applicazioni o processi di lavorare in modo autonomo sulla stessa macchina fisica o su diverse macchine virtuali. Ciò consente di effettuare il deployment di nuovi nodi solo quando necessario.

Al fine di creare un container Docker sarà necessario specificare un *Dockerfile* per ogni servizio erogato. Similmente ai gitignore per il versioning git, è possibile definire dei *dockerignore* per segnalare a docker quali file ignorare durante la copia dei file all'interno del container. Per gestire invece applicazioni composte da più servizi sarà possibile utilizzare *Docker Compose*, uno strumento che permette con un solo comando di avviare un intero sistema isolato gestendo nello specifico dipendenze tra servizi, volumi e reti.

Figura 4: Docker logo

2.3.2 Tecnologie Server

Node.js Node.js [5] è un *runtime system* multipiattaforma per l'esecuzione di codice JavaScript, costruito sul motore JavaScript V8 di Google Chrome, progettato per creare applicazioni di rete scalabili. Grazie al suo funzionamento molte connessioni possono essere gestite contemporaneamente, per ognuna delle quali verrà invocata una callback, rendendo Node attivo solo al momento necessario.

Node.js implementa un'architettura event-driven, facendo dunque affidamento su un event loop. Non esiste alcuna chiamata per avviare il ciclo: Node.js entra semplicemente nel ciclo degli eventi dopo aver eseguito lo script di input e, analogamente, esce dal ciclo di eventi quando non ci sono più callback da eseguire. Questo comportamento è simile a JavaScript in browser: il ciclo degli eventi è nascosto all'utente. Per natura dell'event loop, Node è single-threaded, ma è possibile, su necessità, effettuare delle fork per sfruttare al meglio i core offerti dalla macchina creando nuovi thread.

Figura 5: Node.js logo

Express Express è un framework web per Node.js che semplifica lo sviluppo di applicazioni web e API. Fornisce una serie di funzionalità per gestire richieste HTTP, definire *rotte*, elaborare dati di input e gestire le risposte. Express è estremamente flessibile e leggero, consentendo agli sviluppatori di creare rapidamente applicazioni web scalabili e con ottime prestazioni.

Una delle caratteristiche principali di Express è il concetto di *middleware*, ovvero funzioni che possono essere eseguite prima, durante o dopo il processo di gestione delle richieste. Questo consente agli sviluppatori di rendere modulare il codice e aggiungere facilmente funzionalità come autenticazione e gestione degli errori.

Express segue il paradigma di programmazione event-driven di Node.js e si integra perfettamente con esso. Può essere utilizzato insieme ad altri moduli Node.js per gestire aspetti specifici delle applicazioni web, come la gestione delle sessioni utente o la connessione al database.

Grazie alla sua vasta adozione nella comunità Node.js, Express ha una vasta gamma di plugin e middleware disponibili, che permettono agli sviluppatori di estendere facilmente le funzionalità base del framework per adattarsi alle esigenze specifiche del progetto.

In sintesi, Express è un potente framework per lo sviluppo di applicazioni web con Node.js, che offre una combinazione di flessibilità, prestazioni e facilità d'uso.

Figura 6: Express logo

Mongoose Mongoose è una libreria per Node.js che permette di creare degli *Schema* per rappresentare i dati da archiviare nel sistema. Ogni Schema è associato a una collezione nel Database di MongoDB. Mongoose viene utilizzato per la creazione del proprio model, essendo possibile creare delle istanze dallo Schema attraverso delle *Factory* e utilizzarli come dei semplici oggetti Javascript. Oltre a offrire metodi aggiuntivi già pronti per salvare i dati all'interno del database è possibile creare funzioni che solo oggetti appartenenti a un relativo schema possono richiamare, rendendo la modellazione simile all'object-oriented.

Figura 7: Mongoose logo

2.3.3 Tecnologie Database

MongoDB MongoDB è un DBMS NoSQL, cioè non utilizza un meccanismo di persistenza relazionale come un tradizionale SQL. Il modello NoSQL non è unico e può dunque utilizzare varie strutture dati per sostituire le tabelle con campi uniformi utilizzate in SQL.

In particolare MongoDB utilizza un modello orientato al documento, dove le informazioni sono memorizzate in una struttura gerarchica ad albero e un qualsiasi numero di campi con qualsiasi lunghezza può essere aggiunto. I campi a loro volta possono contenere aggregati di dati composti da più elementi o da strutture annidate.

I DBMS orientati al documento offrono alcuni vantaggi, specialmente in ambito web, rispetto ai tradizionali RDBMS. Si ottengono maggior flessibilità dei dati, utile per avere meno rigidità in fase di sviluppo o, in generale, per scenari in cui i dati memorizzati non sono sempre uniformi, e una maggior facilità nella trasposizione in strutture dati nel codice in quanto i JSON, utilizzati da MongoDB, trovano una corrispondenza uno a uno con esse. Il trade-off nell'avere una struttura meno rigida è però il rischio di duplicazione di dati e inconsistenze, per cui è richiesta al progettista una maggiore cautela nella manipolazione di dati.

Figura 8: MongoDB logo

2.3.4 Tecnologie Client

React.js React è un framework open-source che permette di implementare applicazioni web seguendo i principi della programmazione a oggetti. In modo particolare risulta essenziale descrivere tre concetti chiave:

- **JSX:** è un'estensione della sintassi di JavaScript.[7] Permette di unire gli aspetti di html come linguaggio di template agli aspetti di JavaScript come linguaggio di scripting in una forma che ne aumenta semplicità e leggibilità. Gli elementi JSX vengono utilizzati nelle definizioni delle funzioni di rendering semplificando la costruzione della UI. Attraverso JSX si può richiamare un componente React tramite con un meccanismo analogo ai tag in html.

- **Componenti:** attraverso un componente[8] si va a definire quella che risulta essere a tutti gli effetti una classe. Il componente, definito da uno o più costruttori,

contiene uno stato, il quale verrà mantenuto, ed eventualmente aggiornato, durante tutto il ciclo di vita dello stesso. È possibile ricevere dati e istruzioni da altri componenti attraverso le props.

- **Stato:** è un insieme di proprietà di un componente.[9] Queste proprietà possono variare a seguito dell'interazione con altri componenti o come azione del componente stesso nel caso esso esegua delle azioni a cadenza temporale. In React lo stato risulta inoltre fondamentale ai fini di ottenere un aggiornamento delle interfacce performante. Ogni componente React deve obbligatoriamente definire una funzione *render()*. Attraverso di essa verrà ritornato il contenuto da renderizzare. React cambierà il contenuto della UI, utilizzando quindi risorse, solamente quando vi saranno delle modifiche nel contenuto ritornato dalla funzione *render()*. Utilizzando dunque le proprietà che definiscono lo stato del componente all'interno di questa funzione sarà possibile ridurre al minimo il numero di volte in cui l'applicazione verrà renderizzata, riflettendo i cambiamenti di stato del componente.

React infine introduce anche i componenti funzione, che di fatto svolgono la stessa funzione dei componenti precedentemente descritti, ma con una sintassi più concisa.

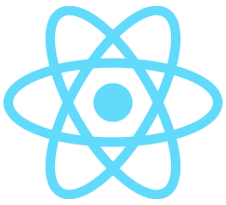

Figura 9: React logo

Redux Redux [10] è un *contenitore di stato* per applicazioni JavaScript. Gode di quattro caratteristiche fondamentali per progetti portata medio-grande:

- **Deterministico:** aspetto fondamentale nelle applicazioni web di ogni genere è il determinismo. L'oneroso compito di far collaborare tutti i componenti al fine di ottenere un comportamento predicibile viene largamente semplificato dall'utilizzo di questo framework.

- **Centralizzato:** avere stato e logica centralizzati permette di ottenere rapidamente delle feature di fondamentale importanza, come funzioni di "annulla" e "ripeti" che permettono di muoversi agilmente tra lo storico degli stati. Un approccio centralizzato garantisce inoltre una consistenza maggiore dei dati, rendendo possibile modificarli solamente attraverso specifiche funzioni, le azioni, create durante la definizione della struttura dati.

- **Debug oriented:** attraverso semplici plugin browser come Redux DevTools risulta immediato il debug dell'applicazione. Attraverso questi tool si ottiene una visione completa dello stato dei dati e della sua evoluzione nel tempo, riuscendo a

identificare con precisione quando e soprattutto perché lo stato abbia subito delle modifiche.

- **Flessibile:** Redux funziona con ogni layer di UI e, essendo ormai uno strumento consolidato, dispone di un solido supporto e una vasta proposta di plugin e pacchetti aggiuntivi per ogni esigenza.

Per comprendere come queste caratteristiche si concretizzino risulta fondamentale comprendere tre concetti chiave nella struttura di Redux:

- **Store:** è un oggetto che contiene l'intera struttura ad albero dello stato.[11] Fornisce metodi per la lettura dello stato corrente.
- **Reducer:** definiscono la struttura dello store.[12] Vi possono essere più reducer all'interno di una stessa applicazione, al fine di meglio suddividere lo stato.
- **Azioni:** permettono di modificare il contenuto dello store.[13] Essendo queste l'unico modo per alterare lo stato attuale, qualsiasi componente che voglia agire sullo stato deve passare per le azioni definite.

Ci sono alcune motivazioni che portano alla scelta di utilizzare questo strumento in aggiunta allo stato messo a disposizione da React. Quest'ultimo pur essendo uno strumento molto potente, pone davanti a delle limitazioni.

Non è raro che più componenti all'interno dell'applicazione debbano fare riferimento allo stesso dato, per cui la presenza di uno stato comune evita di dover implementare meccanismi ad hoc di sincronia tra gli stati dei due componenti. Lo stato di ogni componente, attraverso strumenti preesistenti, sarà dunque sincronizzato con la struttura principale.

React inoltre, data la sua natura orientata alla programmazione reattiva, incoraggia la propagazione dell'informazione solo in una direzione: da un componente padre verso un componente figlio. La presenza delle azioni Redux risolve questo problema in quanto ogni cambiamento apportato allo stato Redux si rifletterà su tutti i componenti React che utilizzano quel particolare dato, in maniera trasparente nei confronti della loro gerarchia.

Figura 10: Redux logo

PeerJS PeerJS è una libreria JavaScript open-source nata per semplificare la creazione di applicazioni *peer-to-peer* in modo trasparente e affidabile. Utilizzando WebRTC, ovvero Web Real-Time Communication, PeerJS permette di stabilire connessioni dirette tra i browser senza la necessità di server intermediari. Questo approccio decentralizzato riduce la latenza e il carico di lavoro sul server, migliorando l'efficienza e la scalabilità delle applicazioni P2P.

Sfruttando PeerJS è possibile creare e gestire connessioni P2P in modo semplice, avendo a disposizione API intuitive per aprire, accettare e gestire le connessioni. La libreria gestisce automaticamente aspetti complessi come il *signaling* e il trasporto dei dati attraverso i peer connessi, consentendo di concentrarsi maggiormente sulla logica dell'applicazione anziché sulle complessità della comunicazione in tempo reale.

Tra le funzionalità offerte utili per la creazione di questo tipo di applicazioni le principali sono la trasmissione di dati in tempo reale, la condivisione di file e la comunicazione audio/video. Inoltre, essendo basato su WebRTC, PeerJS è compatibile con la maggior parte dei browser moderni e fornisce quindi una soluzione sicura e di facile utilizzo.

Per poter connettere i peer l'un l'altro è necessario però avere un server che svolga il compito di *connection broker*, per cui non riceverà nessun dato della rete P2P. Per quanto riguarda l'hosting del server, PeerJS offre due opzioni:

- utilizzare il PeerServer Cloud fornito direttamente dagli sviluppatori della libreria, che è una versione cloud-hosted gratuita del PeerServer ufficiale. Questa opzione è conveniente per chi non ha specifiche necessità e preferisce una soluzione facile da implementare,
- eseguire il proprio PeerServer, utilizzando il codice sorgente open-source scritto in Node.js disponibile online. Questa opzione offre maggiore controllo e flessibilità, ma richiede competenze tecniche per l'installazione e la configurazione.

Figura 11: PeerJS logo

3 Design

In questo capitolo verranno descritti gli aspetti di progettazione sui quali si basa l'elaborato svolto.

3.1 Struttura

Per lo sviluppo del sistema è stato scelto di utilizzare un'architettura ibrida tra *Client-Server* e *Peer-to-Peer*. Come si può vedere in figura 12 il server sarà il punto di snodo tra i client e il database, svolgendo inoltre la funzione di *discovery server* per quanto riguarda la rete p2p. Ogni client sarà quindi in comunicazione con il server per le funzioni di autenticazione, discovery e interazioni con il database, ma una volta entrato in un *party* sarà connesso direttamente agli altri client. In questo modo il carico di lavoro per il server centrale sarà ridotto al minimo, poiché le informazioni relative alle partite saranno condivise solo all'interno delle reti p2p create. Così facendo si introduce anche un livello di tolleranza in caso di errori all'interno della rete in quanto, dopo aver svolto il ruolo di discovery, ovvero dopo aver inviato gli indirizzi dei peer già presenti all'interno di un party al nuovo peer che fa richiesta di entrarvici e viceversa, se il server dovesse non essere più raggiungibile i peer potrebbero comunque avviare la partita e continuare a giocare perdendo solamente la funzionalità di salvataggio al termine della stessa. Inoltre la singola rete p2p potrà ripristinare lo stato originale della partita nel caso in cui si verifichi il fallimento di più nodi poiché tutti i peer condivideranno uno stato coerente e completo del loro stato.

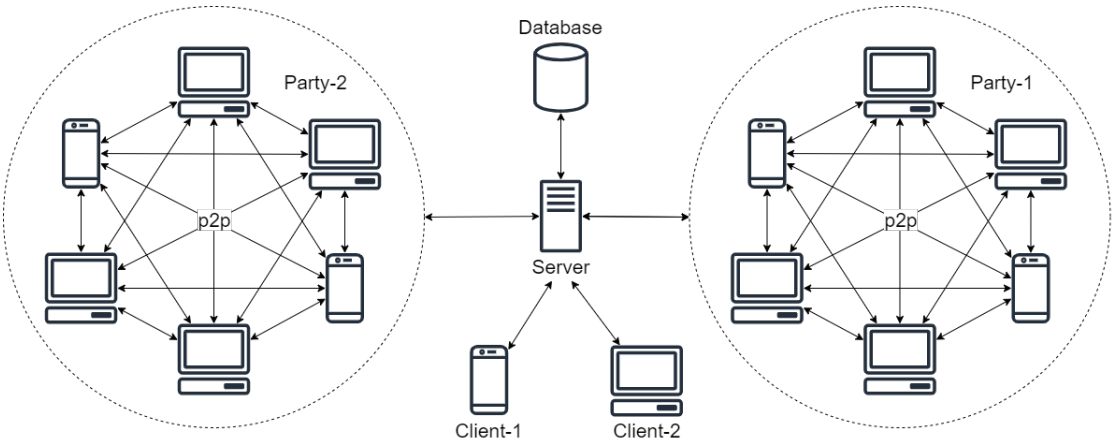

Figura 12: Sintesi architettura del sistema

3.1.1 Server

Il server come detto in precedenza si basa sul framework Express, in accordo con lo stack MERN, e svolge il ruolo di *discovery* per i client che si connettono. Sfruttando la libre-

ria Mongoose all'avvio viene effettuata la connessione al database, dopodiché vengono impostate le route per le richieste HTTP e gli handler per i messaggi ricevuti attraverso le web-socket.

Al fine di evitare una crescita incontrollata delle comunicazioni nel caso il sistema venga scalato a un utilizzo più esteso sono state utilizzate le *rooms* [14] messe a disposizione dalla libreria, ovvero dei canali di comunicazione ai quali le socket possono unirsi che permettono al server di inviare messaggi a specifici sottoinsiemi di client senza effettuare broadcast delle informazioni. La singola room corrisponde come concetto a quello di party descritto in precedenza, rappresentando quindi un gruppo di giocatori logicamente isolato dagli altri.

Il server implementato sarà di tipo *stateful*. Alla base di questo stato ci saranno due liste contenenti strutture dati come riportate nelle figure 13 e 14. Lo stato degli utenti autenticati verrà mantenuto all'interno del campo*sessionStorage*, il quale assocerà a un identificativo univoco utilizzato come chiave, un valore corrispondente alle informazioni di accesso del giocatore, in questo caso *username* e stato di connessione. Le informazioni riguardo la suddivisione in stanze verranno invece memorizzate in una lista di oggetti *room* composti dall'identificativo della stanza stessa, una lista di giocatori, identificati da *username* e *peerId*, e dallo stato di chiusura.

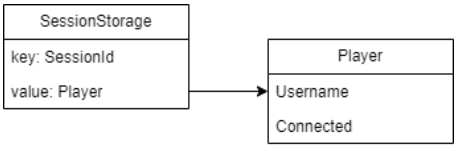

Figura 13: Schema sessione

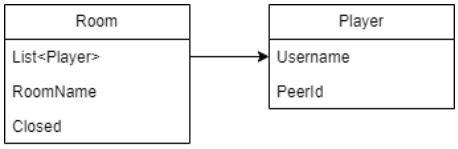

Figura 14: Schema stanze

3.1.2 Database

Il database utilizzato dal sistema è stato strutturato in maniera semplice con l'obiettivo di familiarizzare con le tecnologie impiegate, raggiungendo comunque i requisiti che erano stati prefissati.

Come si può vedere nella figura 15 le informazioni salvate si riassumono in due classi, *user* e *game*. La prima contiene le informazioni relative al singolo utente, ovvero:

- **username:** nome scelto dall'utente in fase di registrazione, univoco all'interno del database.
- **password:** la password impostata che viene salvata come hash utilizzando la libreria *bcript* [15].
- **goodWins/badWins:** statistiche aggregate relative alle vittorie ottenute in base alla squadra di appartenenza.
- **playedGames:** una lista contenente gli identificativi delle partite giocate dall'utente.

Alcune delle informazioni salvate possono essere considerate ridondanti, come ad esempio il campo *playedGames*, ma sono state pensate per uno stato futuro del sistema nel quale il numero di entità di *game* sarà molto elevato, causando così una possibile complessità computazionale crescente per la ricerca di tutte le partite nelle quali ha partecipato uno specifico giocatore.
La classe *game* è composta da:

- **gameCode:** codice univoco utilizzato per identificare la partita.
- **winners:** la squadra vincitrice.
- **players:** un array contenente gli username dei giocatori partecipanti e il loro relativo ruolo.
- **history:** un array la "storia" della partita, ovvero tutti i suoi avvenimenti.

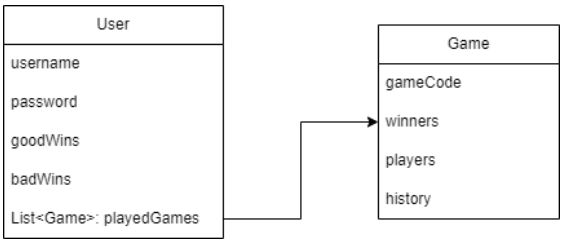

Figura 15: Diagramma delle classi

3.1.3 Client

Anche il client rispetta lo stack scelto per il sistema e si basa quindi sul framework React. La sua struttura interna si basa su due macro comportamenti coesistenti, comportandosi come client nei confronti del server centralizzato e come peer nei confronti degli altri client connessi.

Redux Per quanto riguarda lo stato del client è stata invece sfruttata la libreria Redux in aggiunta al concetto di stato presente in React. Gli oggetti creati e utilizzati in questo caso, visibili nella figura 16, sono i seguenti:

- **user:** contiene tutte le informazioni relative all'utente.
  - **username:** nome univoco identificativo.
  - **room:** il codice identificativo del party selezionato.
  - **token:** identificativo della sessione.
  - **stats:** contiene le statistiche ricevute dal server.
- **util:** raccolta di booleani utili al sistema per definirne lo stato.

  - **socketConnected:** stato della socket verso il server.
  - **peerConnected:** stato della connessione del peer.
  - **isLoading:** stato di caricamento, attesa di informazioni.
  - **cardVisible:** necessità di mostrare la carta all'avvio della partita.

- **game:** contiene tutte le informazioni relative alla singola partita.
  - **gameCode:** codice univoco generato per identificare la partita.
  - **partyClosed:** *True* se il party è completo e chiuso.
  - **players:** array dei giocatori in partita.
  - **phase:** fase della partita.
  - **history:** array contenente lo storico della partita.
  - **wolfNumber:** numero di lupi impostati.
  - **extras:** informazioni su quali personaggi extra sono stati impostati.

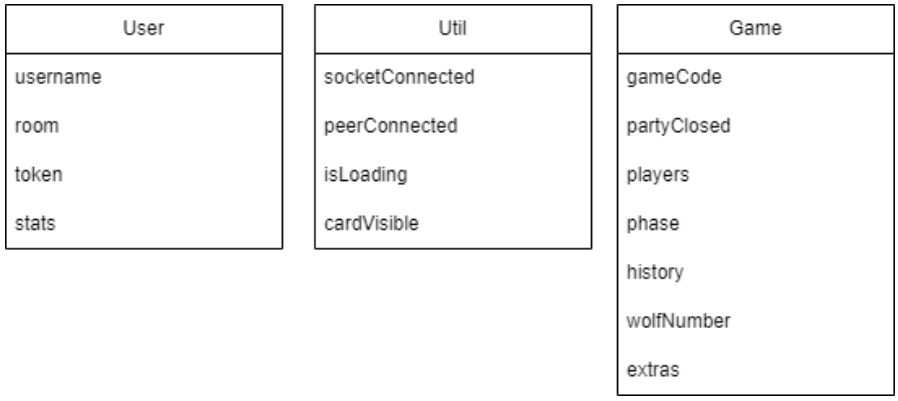

Figura 16: Diagramma stato di Redux

3.1.4 Interfaccia

Lo sviluppo dell'interfaccia utente è stato effettuato in maniera incrementale consultando amici e colleghi del corso per meglio definire le componenti fondamentali e gli scenari di utilizzo. Dopo una prima fase di bozzetti su carta sono stati disegnati i wireframe del progetto utilizzando Figma. La parte di definizione e implementazione ha poi tenuto conto dei principali criteri di accessibilità.

Wireframe Le interfacce sono state sviluppate seguendo il criterio *mobile-first*, qui di seguito verranno riportati esempi di visualizzazione sia mobile che da desktop. Tutte le schermate avranno una *navbar* contenente il logo dell'applicazione e un menu a tendina che potrà contenere diverse funzionalità a seconda dello stato del sistema.

La prima schermata che si presenterà all'utente sarà quella di accesso, visibile nelle figure 17 e 21, dalla quale sarà possibile registrarsi inserendo delle nuove credenziali o accedere attraverso quelle inserite in precedenza.

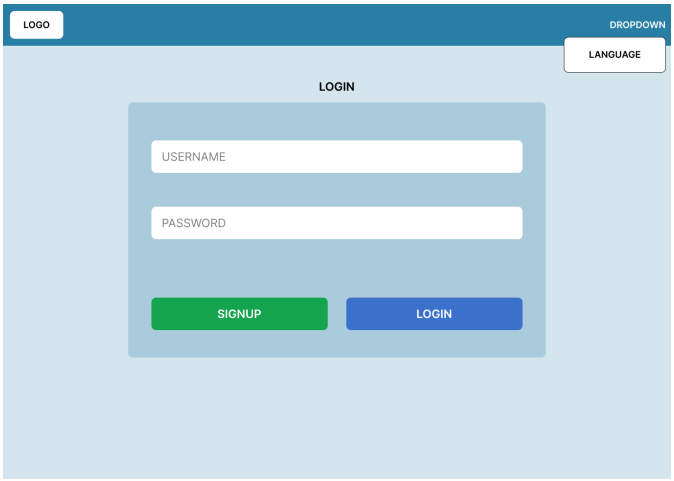

Figura 17: Schermata di login

A seguire una volta effettuato il login l'utente potrà inserire un codice identificativo di una room da creare o alla quale unirsi, attraverso un form visibile nella figura 18. Nella stessa figura è mostrata come esempio anche una notifica che il sistema potrà mandare all'utente sfruttando la libreria *react-notifications*[16].

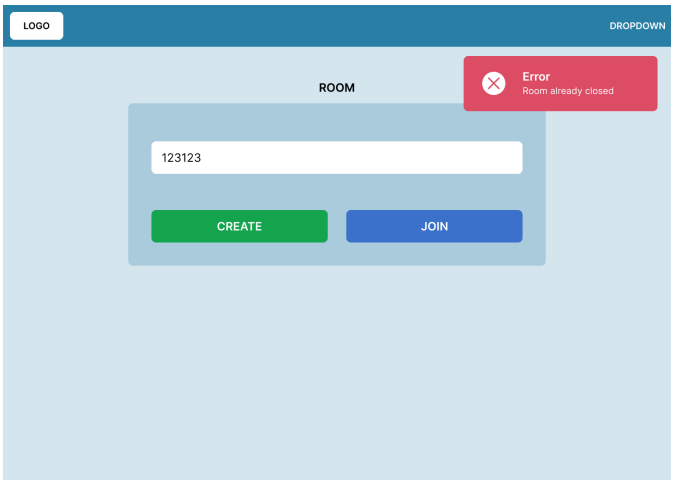

Figura 18: Schermata di selezione della room

Una volta all'interno all'utente verrà mostrata una schermata come quella in figura 19 nella quale si avrà la possibilità di chiudere la room una volta raggiunto il numero minimo di giocatori richiesto. Successivamente in base alle specifiche di implementazione

del gioco sarà data la possibilità attraverso un modale di modificare le impostazioni di gioco, come il numero di lupi o eventuali personaggi extra che si vogliono utilizzare.

Figura 19: Schermata interna alla room

Dopo aver avviato la partita la schermata di gioco sarà quella mostrata nella figura 20, strutturata su tre colonne contenenti le informazioni necessarie, l'elenco dei giocatori partecipanti, la carta rappresentante il ruolo del giocatore e lo storico degli avvenimenti della partita. In base alla fase di gioco nella parte alta della schermata verrà poi mostrato un ulteriore elenco dei giocatori tra i quali scegliere chi si vuole votare. Nella visualizzazione mobile mostrata nella figura 22 le colonne verranno disposte verticalmente in maniera reattiva in base alla dimensione della finestra.

Figura 20: Schermata di gioco

Figura 21: Schermata di login per dispositivo mobile

Figura 22: Schermata di gioco per dispositivo mobile

3.2 Comportamento

In questa sezione si descriverà sinteticamente il comportamento delle due parti principali del sistema.

3.2.1 Server

Qui di seguito nella figura 23 viene riportato il diagramma degli stati che il server attraversa durante la sua esecuzione. Una volta avviato, dopo una fase di inizializzazione e import, il server effettuerà un tentativo di collegamento al database. Nel caso dovesse fallire l'esecuzione viene terminata in quanto non è prevista la possibilità di utilizzare il sistema senza un database.

Una volta effettuata la connessione e impostati gli handler il server si mette in ascolto sulla porta 8080, se non diversamente specificato in fase di avvio.

Raggiunto questo stato che può essere considerato finale il server si occuperà soltanto di ricevere messaggi dai client, elaborare la risposta, aggiornando se necessario il proprio stato interno, per poi inviarla al mittente oppure come messaggio di broadcast a seconda delle necessità.

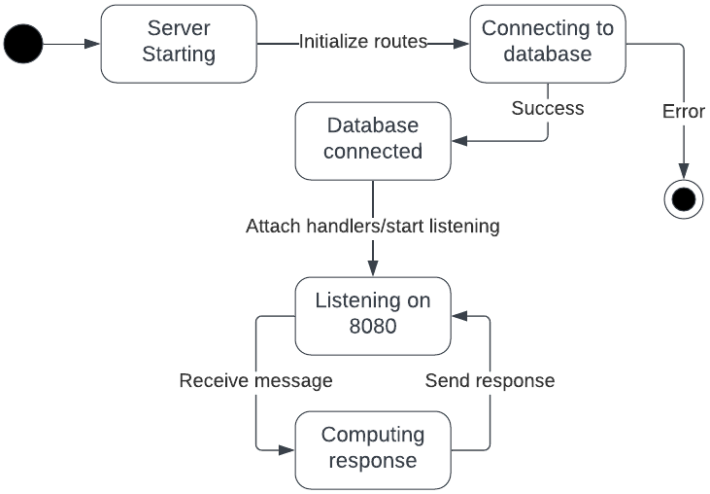

Figura 23: Diagramma degli stati del server

3.2.2 Client

La figura 24 riporta invece i possibili stati del client. Si può osservare come all'avvio il client non effettui immediatamente il collegamento al server, bensì come attenda la prima azione, che può essere quella di autenticarsi o di registrarsi come nuovo utente, per effettuare il primo tentativo. Nel caso in cui il collegamento dovesse fallire il sistema ritornerebbe allo stato di partenza senza interrompere l'esecuzione.

Una volta effettuata la connessione e l'autenticazione lo stato interno del client viene aggiornato con le nuove informazioni ottenute. Successivamente selezionando il codice di una stanza il client entrerà in uno stato di inizializzazione necessario per avviare la sua componente *peer*. Una volta andata a buon fine la procedura il client nello stato di *lobby* attenderà la chiusura della stessa per passare alla fase di gioco effettiva. A questo punto una volta definita la composizione del party ci sarà un alternarsi di due stati, impostazione dei settaggi di gioco e gioco effettivo, nei quali ogni modifica effettuata dall'utente verrà inviata a tutti gli altri peer connessi e analogamente rimarrà in ascolto per ogni modifica della quale può essere notificato.

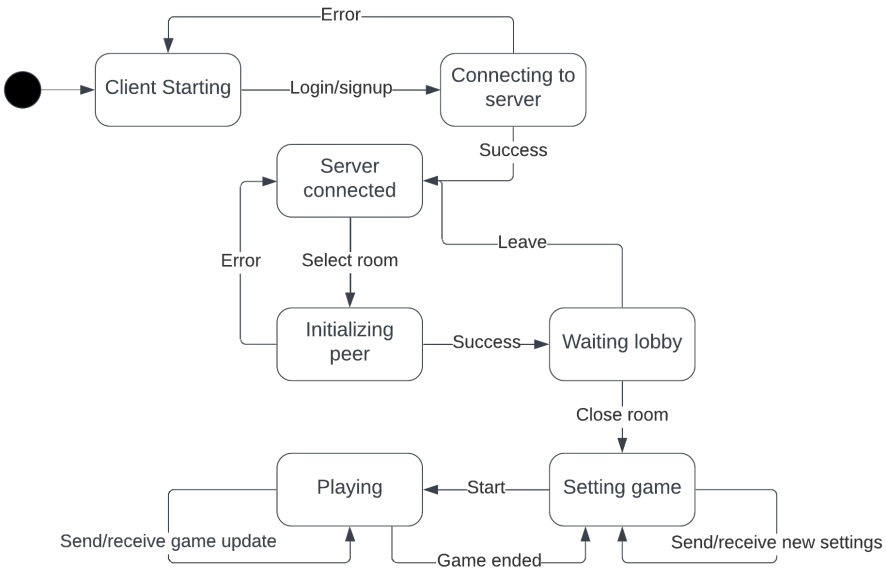

Figura 24: Diagramma degli stati del client

3.3 Interazione

Le interazioni tra i componenti che verranno descritte qui di seguito sono quella tra un client e il server, analizzando quindi la procedura di autenticazione, e quella tra due peer durante le fasi di gioco. È stato ritenuto superfluo approfondire l'interazione tra il server e il database in quanto triviale nella sua implementazione.

3.3.1 Client-Server

L'interazione tra client e server avrà inizio nel momento in cui l'utente attraverso l'interfaccia effettuerà la registrazione o l'accesso al sistema. Nell'esempio riportato in figura 25 viene riportata l'interazione tra le due parti del sistema ipotizzando uno scenario di utilizzo standard.

La connessione alla socket del server viene richiesta dal client nel momento in cui invia il primo messaggio, che può essere di *signup* o *login*. Il server invia poi in risposta all'accesso la sessione dell'utente, la quale viene anche memorizzata per permettere di ristabilire la connessione in caso di fallimenti.

Il client può poi richiedere le statistiche relative all'utente che saranno inviate in risposta dal server.

Per procedere all'inizio del gioco il client invierà un messaggio *room* contenete l'identificativo della lobby nella quale vuole entrare, dopodiché, in caso il codice non corrisponda a una stanza già chiusa nella quale non si era presenti in precedenza, il server invierà un messaggio di conferma al client stesso. A livello di comunicazione il codice della room

verrà poi utilizzato per identificare un canale, indicato come *:Room* in figura, sul quale verrà effettuata una broadcast a tutti i client già connessi comunicando l'ingresso di un nuovo peer. Analogamente al client verranno inviate le informazioni di tutti i peer già presenti.

Nel momento in cui il client comunica al server di voler chiudere la room viene anche in questo caso effettuata una broadcast allo specifico canale. Da questo punto in poi le interazioni tra client e server si sospendono, e hanno inizio le interazioni di gioco che saranno descritte nella sezione successiva. Infine terminata la partita il client potrà comunicare al server la volontà di salvarla nel database

Nel caso in cui un utente lasciasse la partita prematuramente allora verrà comunicata la distruzione del peer al server che si occuperà di notificare il canale. L'ultima interazione mostrata è quella di logout, la quale ha come obiettivo quello di comunicare al server di cancellare le informazioni relative alla sessione dell'utente.

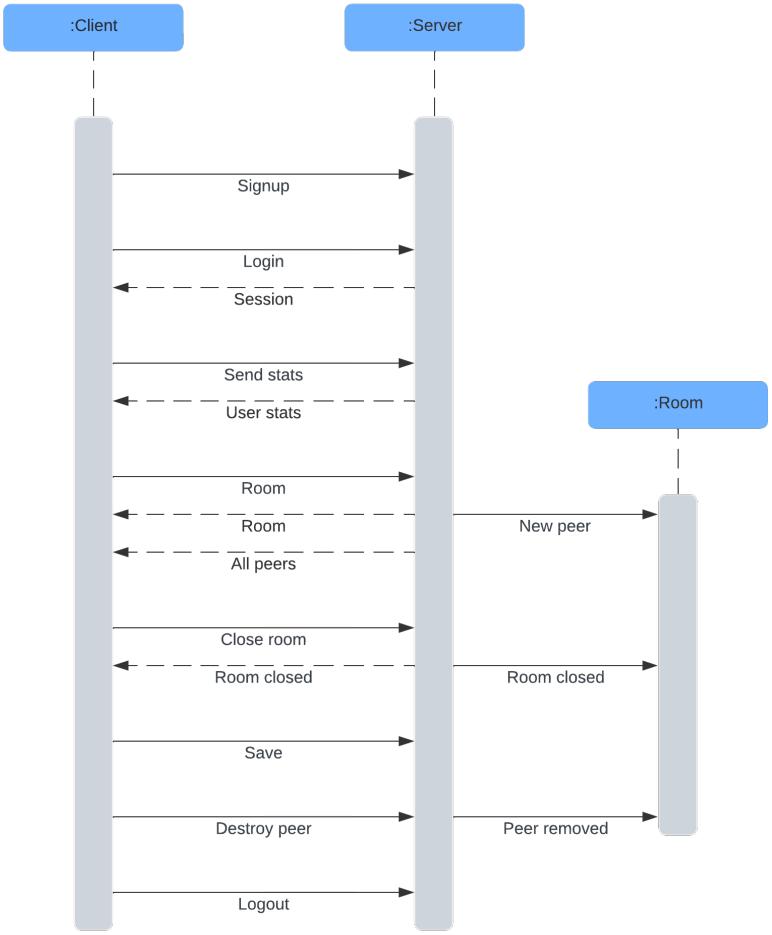

Figura 25: Diagramma di sequenza client-server

3.3.2 Peer-to-Peer

L'interazione tra i peer è limitata alle fasi di gioco. Nella figura 26 è stata riportata una traccia di riferimento semplificata che include nell'insieme *Client-Server* anche l'ultima parte dell'interazione illustrata in precedenza. Si può notare che per mantenere una maggior chiarezza nella figura è stato scelto di rappresentare solo due nodi *Peer_1* e *Peer_2* all'interno della rete, nonostante le comunicazioni vengano inviate in modalità broadcast per una più rapida diffusione dell'informazione.

La prima delle fasi visibile nell'insieme *P2P* è quella di selezione delle impostazioni di gioco, durante la quale le informazioni saranno mantenute coerenti tra i peer inviando messaggi a ogni altro partecipante alla stanza per tutte le modifiche effettuate. Una volta effettuate le modifiche necessarie sarà sufficiente cliccare sul bottone di avvio della partita per generare e successivamente inviare il *GameCode*, entrando così nell'insieme *Game* delle comunicazioni.

In questa parte di gioco in base al ruolo del partecipante potrà essere chiesto di effettuare una votazione, la quale verrà condivisa con gli altri utenti attraverso l'invio della lista di giocatori *Players* aggiornata con il proprio voto. Seguendo la logica del gioco contestualmente all'ultimo voto della fase verranno inviati anche due messaggi ulteriori, uno per indicare la fase successiva e uno per aggiornare lo storico degli avvenimenti della partita. Questi messaggi raccolti nell'insieme *Game* verranno ripercorsi ciclicamente fino alla vittoria di una delle due parti, segnata dall'invio del passaggio alla fase conclusiva.

A questo punto tutti i peer avranno la possibilità di inviare le informazioni della partita al server per salvarle sul database in maniera univoca utilizzando il codice generato all'inizio. Con questo messaggio si chiude l'insieme *Game Loop* che può essere percorso nuovamente mantenendo l'insieme di giocatori definito prima della chiusura della stanza.

Oltre alle comunicazioni illustrate fin'ora può risultare interessante sottolineare una funzione specifica implementata per l'invio cumulativo di tutte le informazioni della partita fino a quel momento, studiata per permettere a un utente di rientrare in gioco in caso di fallimento del sistema. Questa funzione a differenza delle altre interazioni invia i messaggi solo al singolo peer in fase di riconnessione e non in broadcast a tutti i nodi come per gli altri

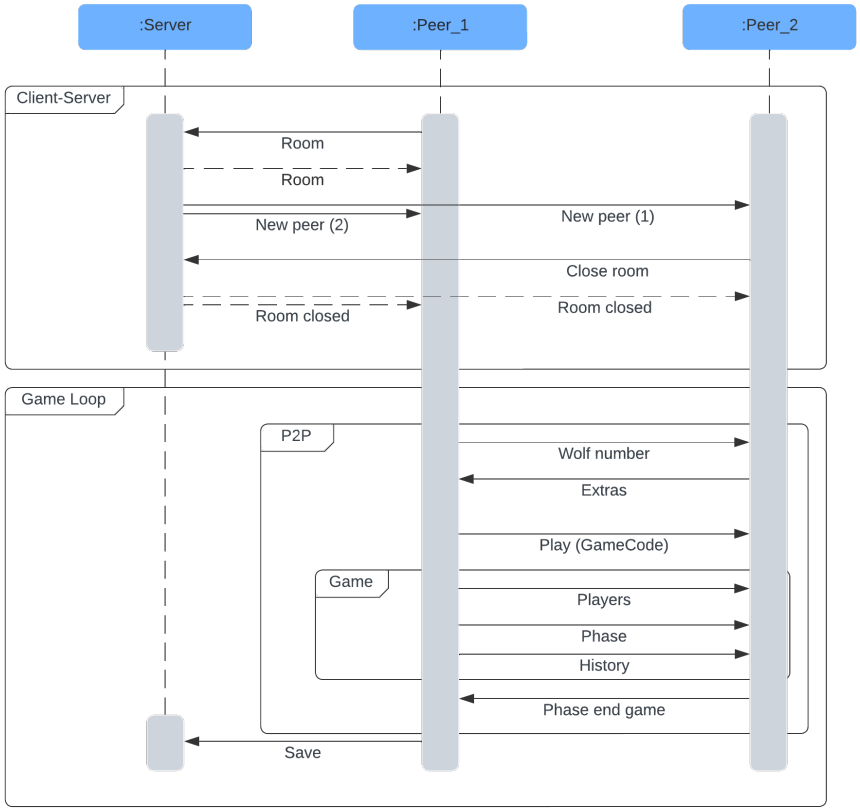

Figura 26: Diagramma di sequenza peer-to-peer

4 Dettagli implementativi

Nel seguente capitolo verranno riportati alcuni estratti del codice implementato per il progetto, con l'obiettivo di evidenziare aspetti affrontati durante lo sviluppo che sono stati ritenuti interessanti.

4.1 Server

4.1.1 Routes

Nell'estratto seguente è possibile vedere come siano definite le rotte all'interno del server. Si dividono in tre gruppi, quelle relative alle informazioni di utenti e partite, definite in file separati, e quella definita per gestire le richieste a '/'.

```
    app.use(require('./routes/user'));
    app.use(require('./routes/game'));
    app.get('/', (req, res) => {
```

```
    res.send('LUPUS' + '<br/>'
                        + 'Discovery server running');
});
```

La struttura del file *./routes/user.js* riportata qui di seguito mostra sinteticamente come il router venga impostato per le richieste *get* e *post* relative agli utenti.

```
const router = express.Router();

router.post("/user", (req, res) => {...});

router.get("/user/:username", (req, res) => {...});

router.get("/users", (req, res) => {...});

module.exports = router;
```

4.1.2 Socket.IO

Per quanto riguarda invece la comunicazione attraverso websocket, lato server utilizzando il seguente codice dopo aver inizializzato *io* è possibile andare ad applicare gli handler per i vari messaggi.

```
//On connection assign handlers
io.on('connection', (socket) => {

    socket.on('disconnect', () => {...} });
    socket.on('logout', () => {...} });
    socket.on('login', (message) => {...} });
    ...
});
```

4.1.3 Mongoose

Gli Schema sul database sono stati creati utilizzando Mongoose, qui di seguito viene riportato lo Schema utilizzato per le informazioni relative agli utenti.

```
// Create User Schema
const UserSchema = new Schema({
    username: {
        type: String,
        required: true,
```

```
            index: { unique: true }},
        password: {
            type: String,
            required: true},
        goodWins: {
            type: Number,
            required: true},
        badWins: {
            type: Number,
            required: true},
        playedGames: {
            type: Array,
            required: true},
    });
```

Sfruttando i middleware messi a disposizione da Mongoose è stato possibile implementare una versione specifica di *.pre('save', ...)*. All'interno di questo hook si è andati, utilizzando la libreria *bcrypt* [17], a generare e conseguentemente salvare la versione cifrata della password grazie all'omonima funzione di hashing [15].

Subito sotto si trova l'implementazione della funzione *comparePassword*, implementata come metodo aggiuntivo dell'istanza dello UserSchema. Anche questa funzione fa uso della libreria vista in precedenza per confrontare la password inserita dall'utente con quella salvata nel database.

```
    UserSchema.pre("save", function (next) {
        var user = this;
        // only hash the password if it has been modified
        if (!user.isModified('password')) return next();

        // generate a salt
        bcrypt.genSalt(SALT_WORK_FACTOR,
            function (err, salt) {
                if (err) return next(err);

                // hash the password using our new salt
                bcrypt.hash(user.password, salt,
                    function (err, hash) {
                        if (err) return next(err);

                        // override the cleartext password
                        // with the hashed one
                        user.password = hash;
                        next();
                    });
```

```
        });
});

UserSchema.methods.comparePassword =
    function (candidatePassword, cb) {
        bcrypt.compare(candidatePassword, this.password,
            function (err, isMatch) {
                if (err) return cb(err);
                cb(null, isMatch);
            });
    };
```

Per illustrare come è stato utilizzato lo Schema appena descritto viene di seguito riportato un estratto del codice proveniente dalle route mostrate in precedenza. Si può vedere la creazione di una nuova istanza di User, il salvataggio di un'istanza sul database e la query di ricerca utilizzata per ricercare un utente attraverso il suo username.

```
const newUser = new User({
    username: req.body.username,
    password: req.body.password,
    goodWins: 0,
    badWins: 0,
    playedGames: []
});

// Create a new User
User.create(newUser)
    .then(function (dbUser) {
        res.json(dbUser);
    })
    .catch(function (err) {
        res.json(err);
    });

User.findOne({ username: req.params.username })
    .then(function (dbUser) {
        res.json(dbUser);
    })
    .catch(function (err) {
        res.json(err);
    });
```

4.2 Client

4.2.1 sessionStorage

Gli oggetti web storage *sessionStorage* e *localStorage* permetto di salvare le coppie chiave/valore nel browser. La particolarità di questi spazi di memorizzazione è che i dati rimarranno memorizzati anche in seguito al ricaricamento della pagina, nel caso del sessionStorage, e anche in seguito a un riavvio del browser, per il localStorage. Proprio per questo limite il primo viene utilizzato molto meno spesso del secondo, nonostante proprietà e metodi siano gli stessi. Il sessionStorage esiste infatti solo all'intero della tab del browser corrente. Un'altra tab con la stessa pagina avrà un archiviazione differente. I dati sopravvivono all'aggiornamento della pagina, ma non alla sua chiusura e successiva riapertura.

In questa versione del codice è stato scelto di utilizzare il sessionStorage per permettere di mantenere più sessioni attive di utenti diversi all'interno dello stesso browser, sacrificando appunto la persistenza dell'autenticazione alla chiusura del browser.

```
const sessionID = sessionStorage.getItem("sessionID");

sessionStorage.setItem("sessionID", sessionID);

sessionStorage.removeItem("sessionID")
```

4.2.2 Redux

Di seguito è possibile vedere il codice relativo allo stato di Redux per quanto riguarda le informazioni dell'utente, raccolte in un unico slice, con le relative azioni.

```
var initialState = {
    username: null,
    room: null,
    token: null,
    stats: null
}

const userSlice = createSlice({
    name: 'user',
    initialState,
    reducers: {
        setUsername(state, action) {
            state.username = action.payload
        },
        setRoom(state, action) {
            state.room = action.payload
```

```
        },
        setToken(state, action) {
            state.token = action.payload
        },
        setStats(state, action) {
            state.stats = action.payload
        },
    },
})

export const { setUsername, setRoom, setToken, setStats }
    = userSlice.actions
```

5 Validazione e auto-valutazione

Successivamente in questo capitolo verranno descritti brevemente alcuni tra gli strumenti utilizzati per effettuare test sul codice sviluppato.

Oltre a questi tools sono stati effettuati vari test con utenti durante le diverse fasi di implementazione. Basandosi sul bacino di utenti per il quale è stato pensato il sistema, sono state consultate per questi test persone interne all'università, anche iscritte ad altri corsi e quindi senza uno specifico background informatico. Nelle prime fasi sono state richieste consulenze riguardo le scelte grafiche, presentando alcuni wireframes, e per quanto riguarda le specifiche a livello di regole di gioco e usabilità. Successivamente sono stati effettuati test di accessibilità chiedendo di accedere al sistema da dispositivi diversi e testare le funzionalità basilari come registrazione, autenticazione e persistenza delle sessioni. Nelle fasi finali i test sono stati impostati come vere e proprie demo di scenari di utilizzo reali, giocando partite complete toccando tutte le funzionalità implementate.

5.1 Requisiti

Qui di seguito verranno riportati gli elenchi presentati in fase di definizione dei requisiti analizzando quali di essi siano stati soddisfatti dal livello di implementazione raggiunto.

Requisiti funzionali

- ✓ accedere al sistema tramite credenziali univoche
- ✓ creare un nuovo party
- ✓ unirsi a un party esistente
- ✓ finalizzare il party e accedere alla lobby
- ✓ avviare la partita

- ✓ selezionare un giocatore da uccidere/accusare
- ✓ visualizzare il risultato a partita terminata

I due requisiti funzionali identificati in fase di analisi come opzionali sono stati alla fine soddisfatti, in quanto il sistema permette sia di personalizzare le impostazioni della partita, sia di salvarla al termine della stessa.

- ✓ (opzionale) selezionare la tipologia di partita
- ✓ (opzionale) salvare il risultato

Requisiti non funzionali

- ✓ il sistema dovrà essere scalabile
- ✓ l'architettura ibrida dovrà essere gestita in maniera efficiente
- ✓ la comunicazione dovrà essere "sicura" tra i peers e il server
- ✓ il sistema dovrà essere in grado di gestire in maniera corretta la connessione e disconnessione degli utenti durante il gioco

5.2 Deliverables

Al termine dello sviluppo gli artefatti prodotti sono stati:

- ✓ Il database
- ✓ Il Server
- ✓ Il Client WEB
- × Il Client Mobile

5.3 Redux DevTools

Per effettuare i test necessari a verificare che lo stato dell'applicazione fosse correttamente gestito sono stati usati gli strumenti offerti da Redux DevTools [18]. Questi strumenti per sviluppatori permettono di migliorare il flusso di sviluppo di Redux o qualsiasi altra architettura che gestisca cambiamenti di stato. Può essere utilizzato come estensione per i principali browser, Chrome, Edge e Firefox, come app standalone oppure come componente React integrato nel client.

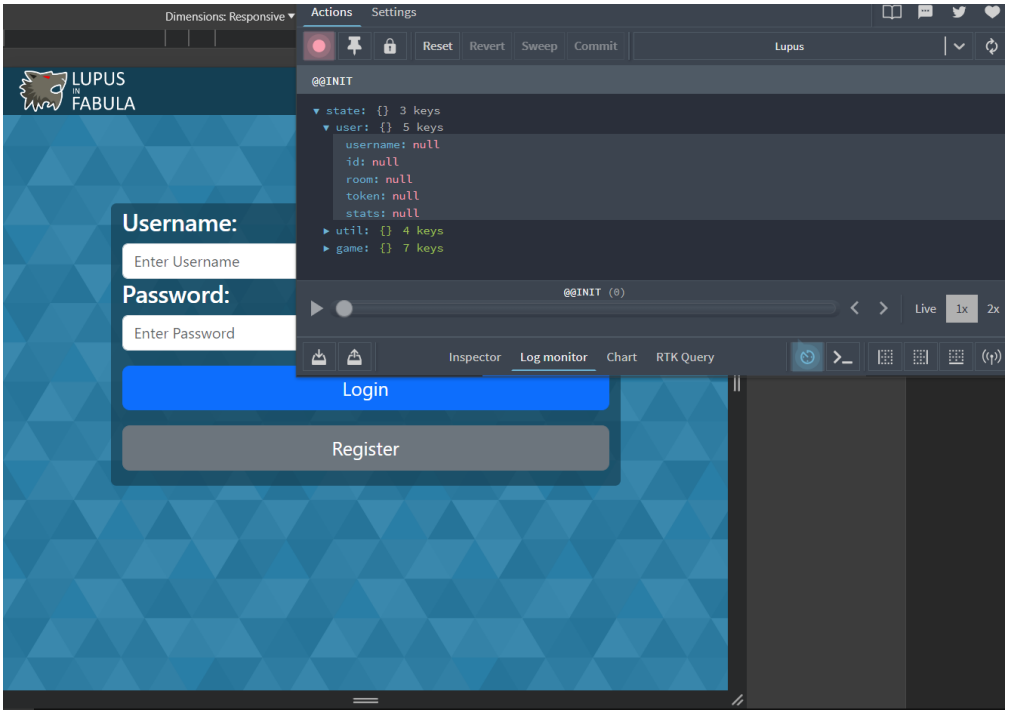

Figura 27: Redux DevTools in uso

Per lo sviluppo di questo progetto è stata utilizzata l'estensione messa a disposizione per Chrome, come mostrato nell figura 27, nella quale è possibile vedere lo stato in tempo reale di Redux.

Figura 28: Redux DevTools logo

5.4 Axe DevTools

Per testare aspetti di accessibilità sono stati invece utilizzati gli strumenti messi a disposizione da axe DevTools [19]. Questi tools permettono, sempre attraverso un'estensione

disponibile per il browser Chrome, di analizzare vari aspetti di accessibilità relativi a una specifica pagina web o a una sua sottoparte.

L'utilizzo dei tools è visibile nella figura 29, nella quale si può inoltre osservare la struttura di suddivisione utilizzata per attribuire un livello di gravità al problema riscontrato.

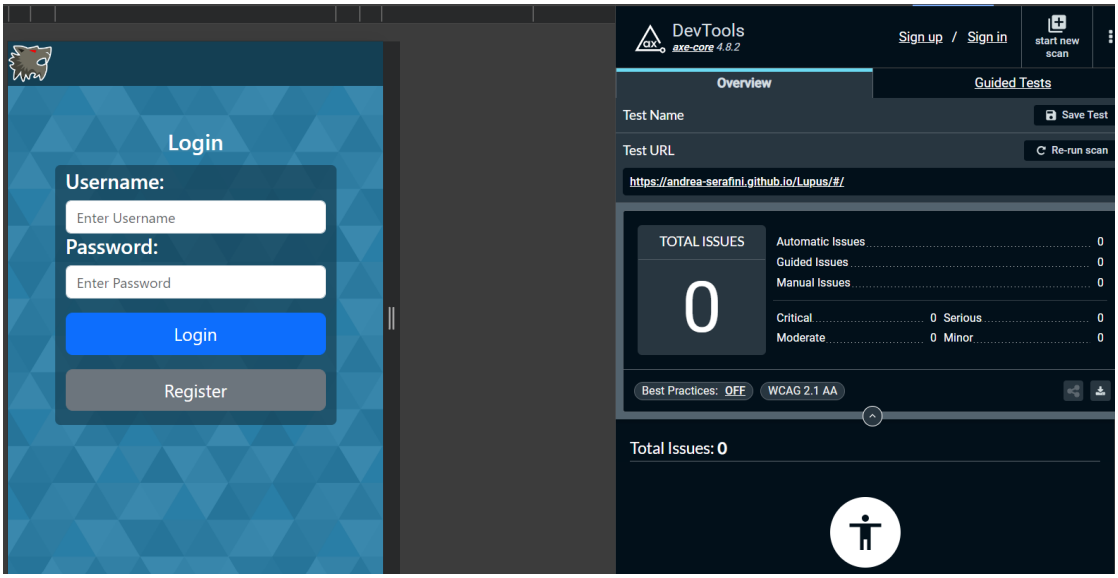

Figura 29: Axe DevTools in uso

Il set di tool di cui ci siamo serviti per monitorare ed eventualmente correggere questioni riguardanti l'accessibilità nel presente applicativo. Axe DevTools è un insieme di strumenti leader del settore, che permette tramite un'estensione installabile direttamente su browser di ispezionare una pagina web o sottoparti di essa allo scopo di individuare problemi di accessibilità. Il tool poi procede con il suddividere i problemi in varie categorie a seconda della gravità dei problemi eventualmente riscontrati:

Figura 30: Axe DevTools logo

5.5 Lighthouse

Lighthouse è uno strumento automatizzato completamente open-source pensato per migliorare le prestazioni, la qualità e la correttezza delle applicazioni Web [20].

Durante il controllo di una pagina, questo tool esegue una serie di test automatizzati e genera quindi un rapporto sul rendimento di quest'ultima, visibile nella figura 31. Da qui è possibile utilizzare eventuali test falliti come indicatori di quali azioni si possano intraprendere per migliorare la propria applicazione.

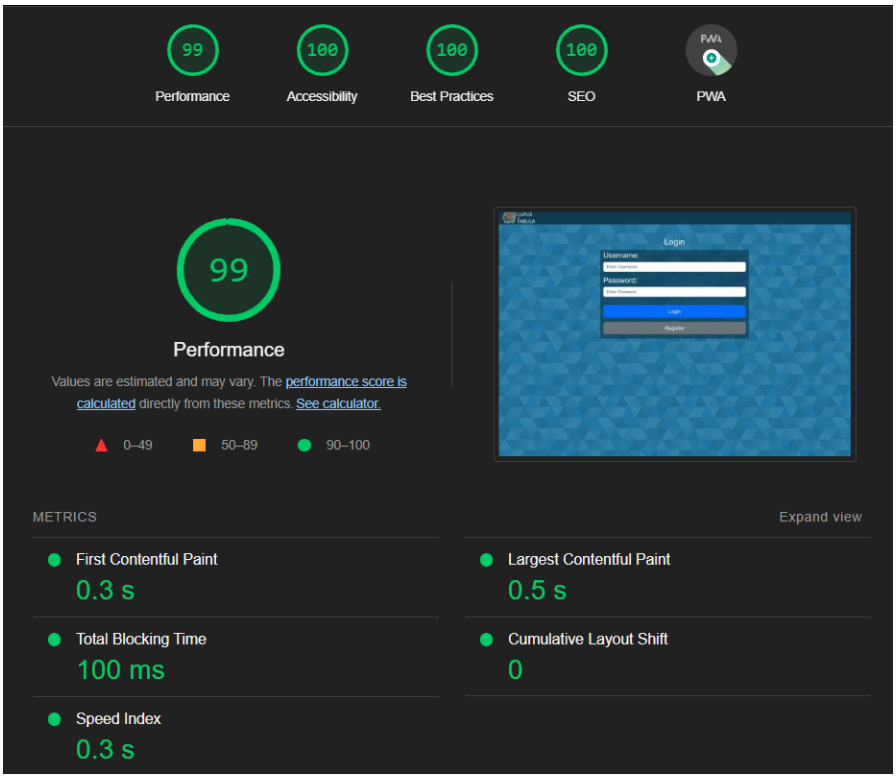

Figura 31: Lighthouse in uso

Figura 32: Lighthouse logo

6 Istruzioni per il deployment

In questo capitolo verranno illustrate le modalità e gli strumenti utilizzati per effettuare il deploy dell'applicazione sviluppata.

6.1 Node-NPM

La prima modalità utilizzata per effettuare il deploy del servizio è sta quella basata su *Node* ed *npm*. Per poter quindi eseguire questi step sarà necessario installare o verificare l'installazione dei seguenti componenti.

- MongoDB
- Node
- npm

Per verificare l'installazione di MongoDB sarà sufficiente lanciare il comando `mongod -version`, nel caso non fosse presente bisognerà procedere a installarlo. Analogamente con i comandi `node -v` e `npm -v` si verificherà la versione installata degli altri due.

Una volta che l'ambiente è stato correttamente impostato sarà sufficiente utilizzare *git* per importare il progetto, oppure scaricarlo manualmente direttamente dalla pagina web del repository.

6.1.1 Local

Per le fasi di sviluppo il sistema è stato utilizzato in modalità locale, con l'obiettivo di verificarne il funzionamento in itinere. Per fare ciò è stato necessario avviare localmente le due componenti, server e client.

Per avviare il server dopo essersi spostati all'interno dell'omonima cartella con il terminale bisognerà eseguire i seguenti comandi:

```
npm install
set LUPUS_SERVER_PORT=8081
node server.js
```

Il primo si occuperà di installare tutte le dipendenze necessarie, mentre l'ultimo avvierà effettivamente il server. Per quanto riguarda invece il secondo comando mostrato, il suo utilizzo è opzionale in quanto all'interno del progetto la variabile d'ambiente riportata viene impostata di default al valore *8080*, perciò l'utilizzo dell'istruzione si riduce al solo caso in cui sia necessario utilizzare una porta diversa.

Ora per avviare il client bisognerà analogamente da un nuovo terminale spostarsi nella cartella corretta, poi eseguire i seguenti comandi:

```
npm install
set REACT_APP_SERVER_PORT=8081
npm run build
serve -s build
```

Dopo aver installato tutte le dipendenze sarà necessario eseguire il secondo comando per modificare la porta alla quale contattare il server solo nel caso si fosse modificata rispetto a quella di default *8080*. I seguenti comandi si occuperanno di creare la build e rendere l'applicazione accessibile online come mostrato nella figura 33.

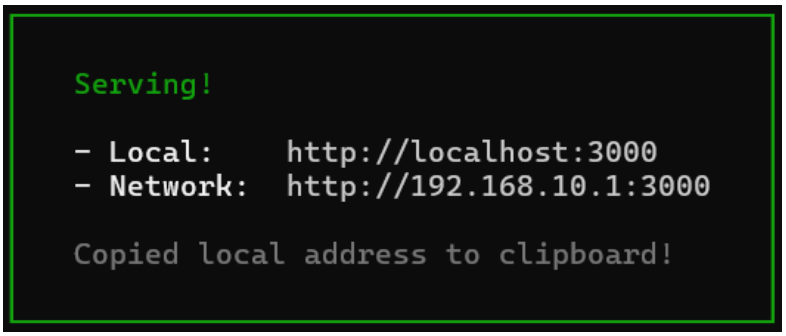

Figura 33: Output del comando serve

6.1.2 Distributed

Per la fase di test eseguita in maniera distribuita è stato necessario introdurre alcuni strumenti ulteriori. Per raggiungere questo obiettivo sono stati quindi utilizzati anche:

- Ngrok[21]
- GitHub Pages[22]

Il primo è un reverse proxy server cross-platform con cui è possibile esporre un server locale, collocato dietro NAT, e firewall alla rete Internet tramite secure tunnel. Dopo aver creato un dominio dalla dashboard messa a disposizione sul loro sito utilizzando il comando

```
ngrok tunnel --label edge=edghts_*** http://localhost:8080
```

sarà possibile connettersi al server sulla macchina locale anche dalla rete esterna all'indirizzo `wise-resolved-wasp.ngrok-free.app`.

Il codice del client dovrà poi essere modificato per sostituire l'indirizzo *localhost* con quello pubblico. Effettuata la sostituzione è stato sfruttato il servizio offerto da GitHub, ovvero GitHub Pages.

Pages permette di fare hosting di un sito web direttamente dal repository del progetto. Avendo aggiunto anche `"deploy": "gh-pages -d build"` agli script sarà possibile eseguire il comando

```
npm run deploy -- -m "Deploy message"
```

per effettuare la build del client e pushare la nuova versione dell'applicazione.

Dopo aver svolto tutti gli step sarà possibile accedere al sito all'indirizzo

```
    https://andrea-serafini.github.io/Lupus/
```

per poter utilizzare il sistema collegandosi da remoto al server locale.

6.2 Docker

Per effettuare un deployment semplificato e indipendente dall'architettura della quale si dispone sono stati impostati i file necessari per l'utilizzo di Docker. Una volta installata e avviata l'applicazione di Docker i comandi da eseguire da linea di comando saranno:

```
#Per avviare
    docker compose up --build

#Per terminare
    docker compose down
```

La struttura del progetto e dei conseguenti file *docker-compose* permette sia di avviare tutti i container insieme, che di avviare in maniera separata il client e la coppia server-database.

7 Esempio d'uso

Qui di seguito verrà illustrato un caso d'uso del sistema per il quale sono riportati screenshot dal punto di vista di due utenti. L'utente *Andrea* partecipa alla partita dal cellulare, per cui l'interfaccia mostrata sarà quella mobile, mentre l'utente *Piero* accede al client attraverso un browser da un PC desktop.

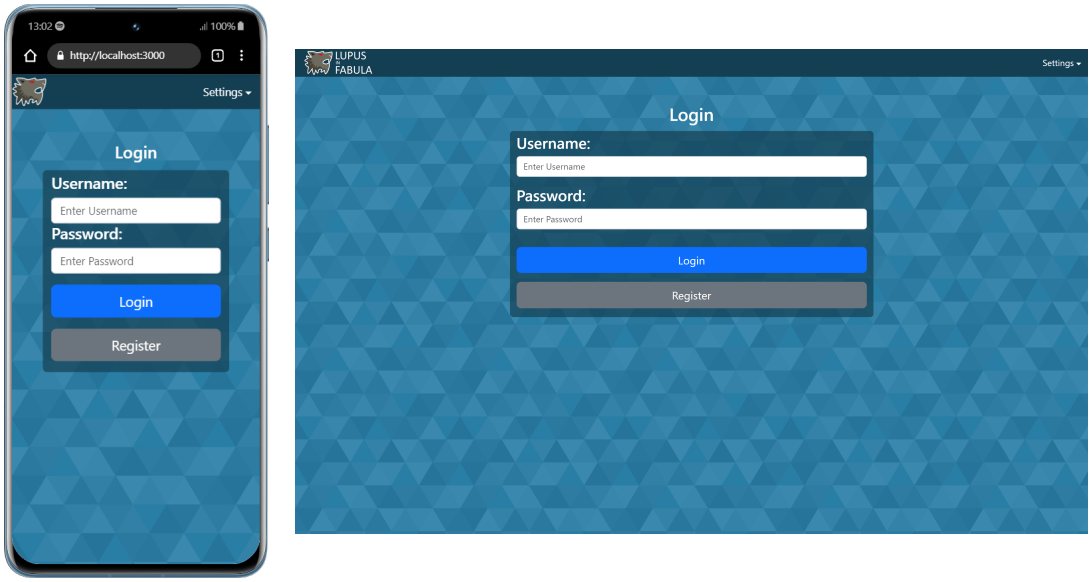

Figura 34: Schermata di autenticazione

Nella figura 34 l'interfaccia proposta all'utente richiede l'inserimento delle credenziali per il login. Nel caso fosse la prima volta che si accede al servizio sarà necessario cliccare sul bottone *Register* per registrarsi. Dal menù in alto a destra sarà possibile, come mostrato nella figura 35, selezionare la lingua nella quale utilizzare l'applicazione.

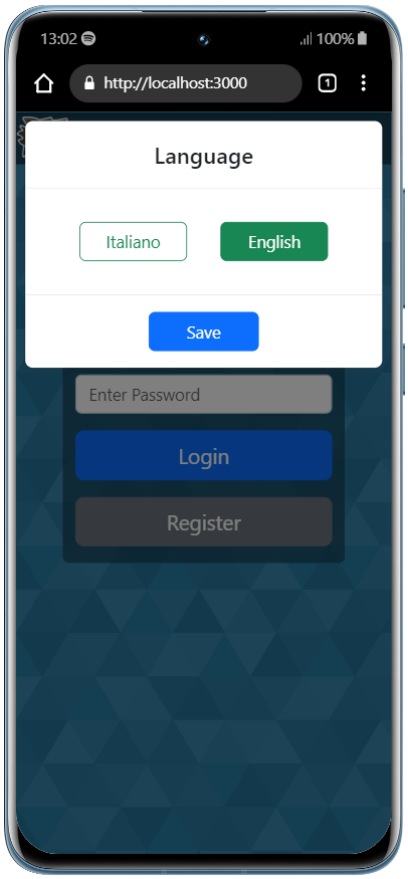

Figura 35: Schermata di settaggio lingua

Una volta effettuato l'accesso si avrà la possibilità nell'interfaccia 36 di creare una stanza, o di unirsi a una già creata in precedenza, inserendo nell'apposito campo il codice univoco del *party*.

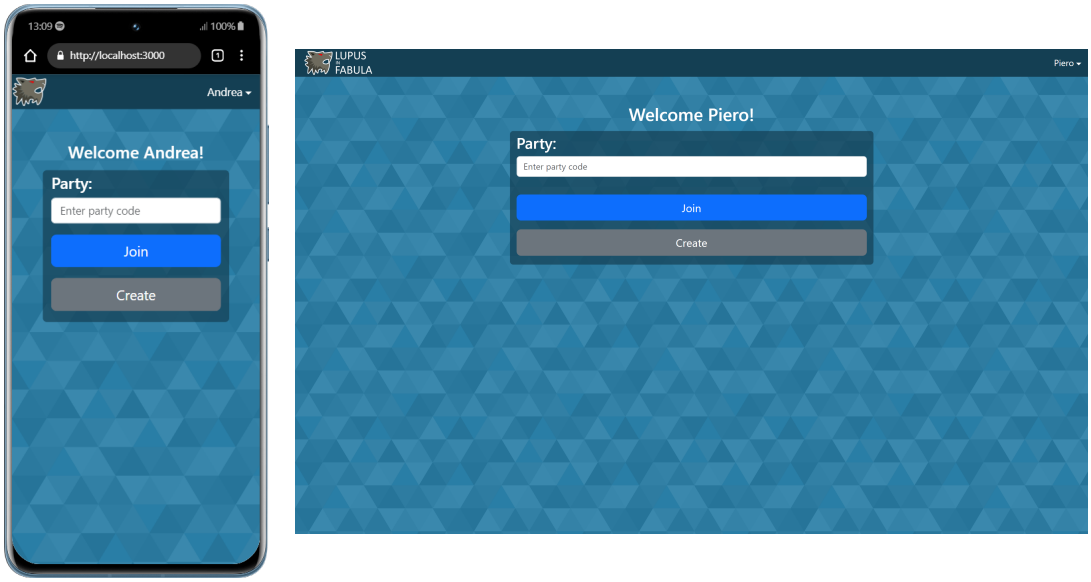

Figura 36: Schermata di selezione party

La schermata successiva che viene mostrata è una semplice *lobby* all'interno della quale è possibile attendere tutti i partecipanti. I pulsanti nella parte bassa dello schermo danno la possibilità all'utente di lasciare la lobby e tornare alla schermata precedente, oppure di chiuderla proseguendo così a quella successiva.

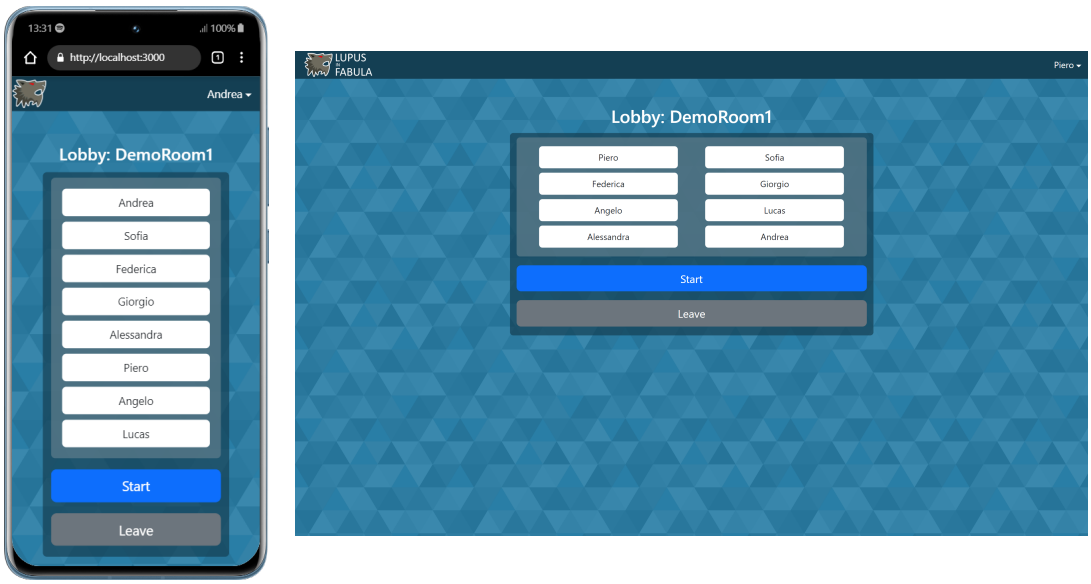

Figura 37: Schermata di lobby

Dopo aver chiuso la lobby l'interfaccia mostrata aggiungerà la possibilità di modificare

le impostazioni della partita prima di cominciare a giocare. La schermata di personalizzazione è mostrata nella figura 38 e permette all'utente di modificare il numero di lupi presenti, oltre che di aggiungere o rimuovere i ruoli extra.

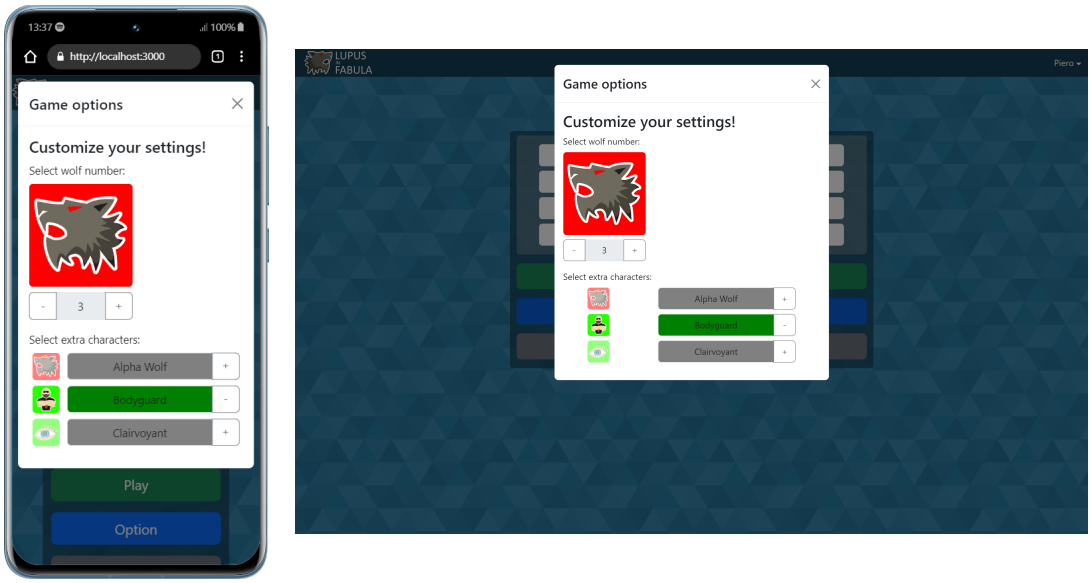

Figura 38: Schermata delle opzioni

Decisa la personalizzazione si può procedere ad avviare la partita, che per prima cosa mostrerà il ruolo assegnato in maniera casuale a ogni giocatore, come mostrato nella figura 39.

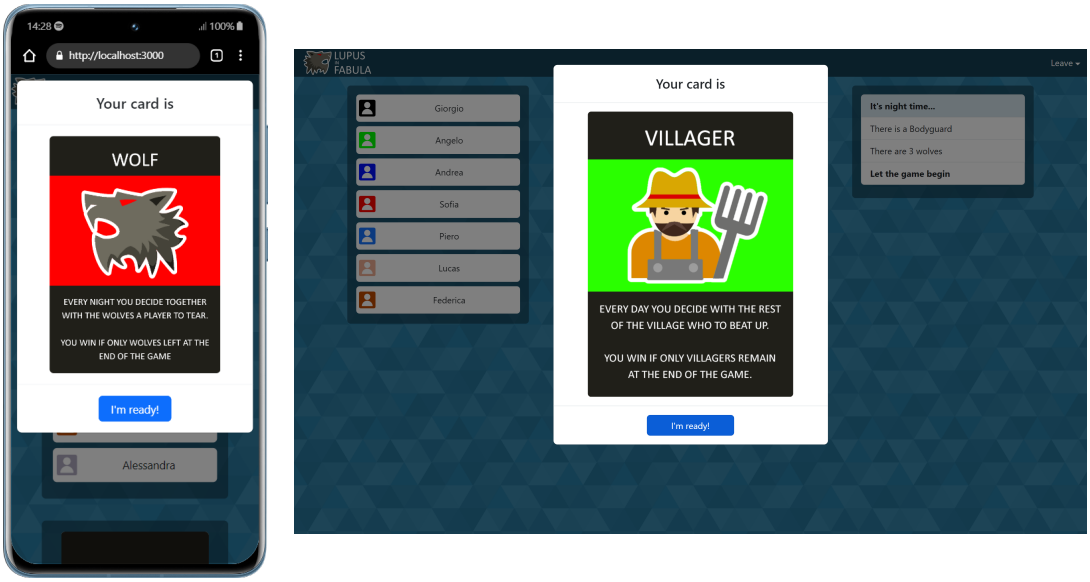

Figura 39: Schermata di inizio partita

Durante i turni di gioco ai partecipanti verrà richiesto di votare per portare a termine un'azione, uccidere, proteggere, svelare, o come nel caso riportato nella figura 40, accusare uno degli altri giocatori.

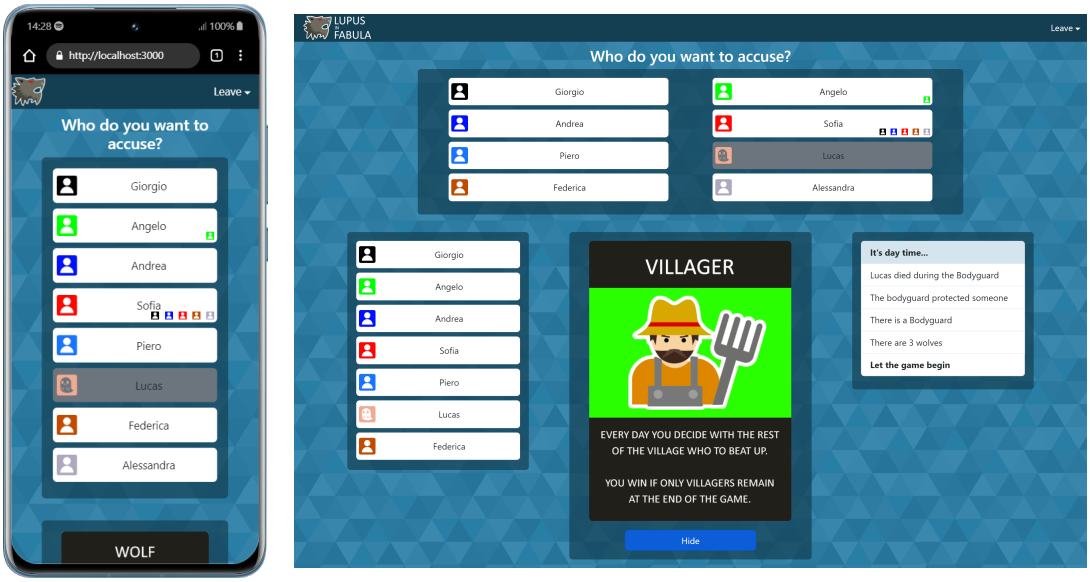

Figura 40: Schermata di turno di accusa

Nella figura 41 si può vedere la schermata che viene mostrata ai giocatori che muoiono durante la partita. In quest'ultima sarà possibile continuare a seguire gli avvenimenti

della partita, ricoprendo un ruolo onnisciente per quanto riguarda le carte distribuite agli altri giocatori.

Figura 41: Schermata del giocatore morto

Al termine della partita verrà infine mostrata una schermata riassuntiva delle due squadre, indicando come visibile nella figura 42 quale delle due abbia vinto e quali membri siano stati uccisi durante il susseguirsi delle fasi. Da qui sarà possibile salvare l'esito della partita sul database oppure procedere direttamente a iniziarne una nuova.

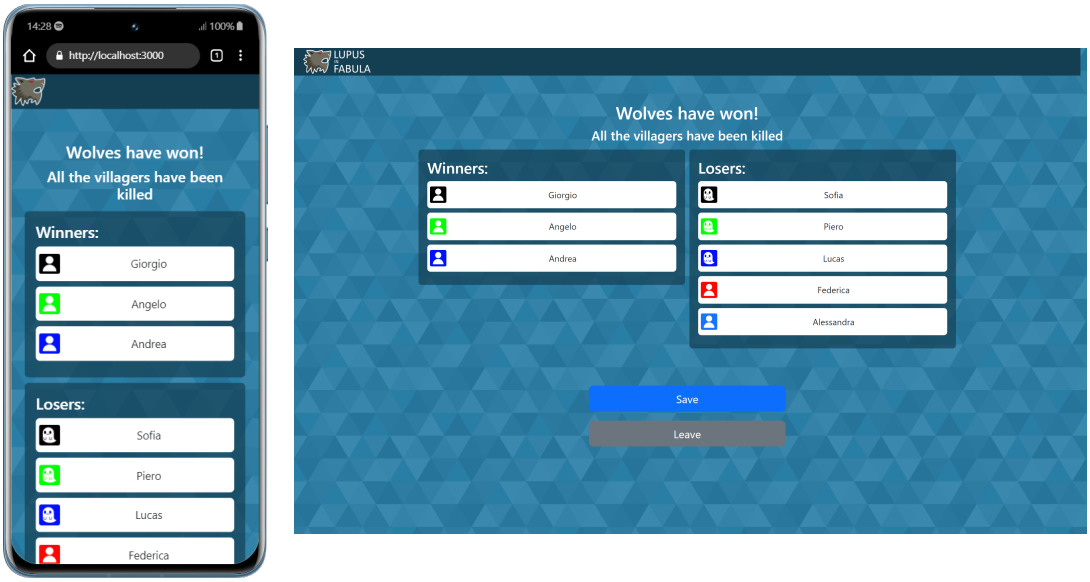

Figura 42: Schermata di fine partita

Come per la schermata di selezione della lingua, una volta effettuato il login sarà possibile per l'utente, sempre utilizzando il menù a tendina in alto a destra, visualizzare le proprie statistiche di gioco.

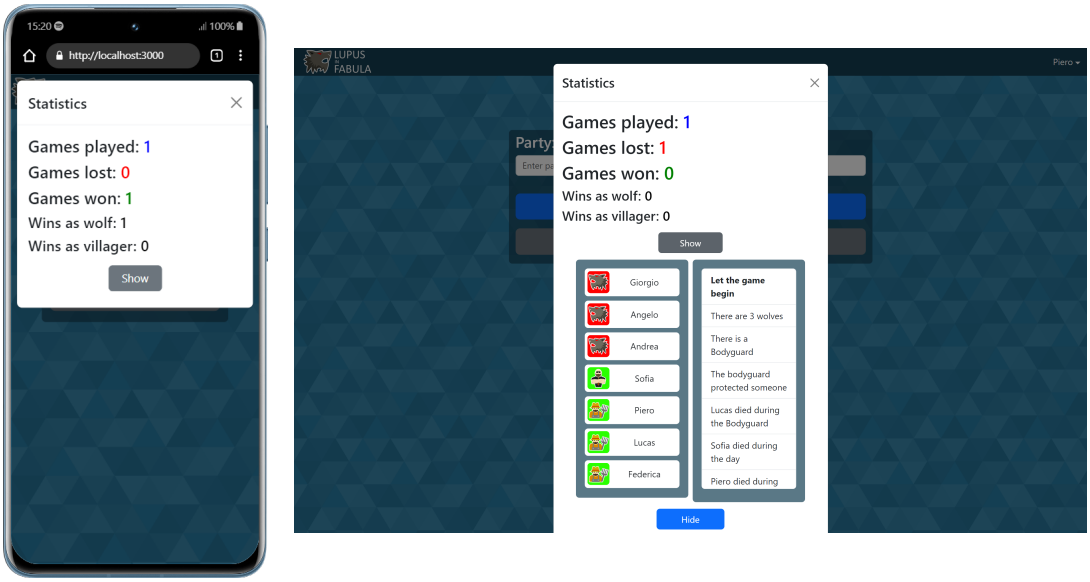

Figura 43: Schermata delle statistiche del giocatore

Nel caso un utente infine provasse a navigare al di fuori dei percorsi definiti all'interno del sistema l'interfaccia che gli verrà presentata sarà quella visibile nella figura 44,

attraverso la quale sarà possibile ritornare alla pagina iniziale.

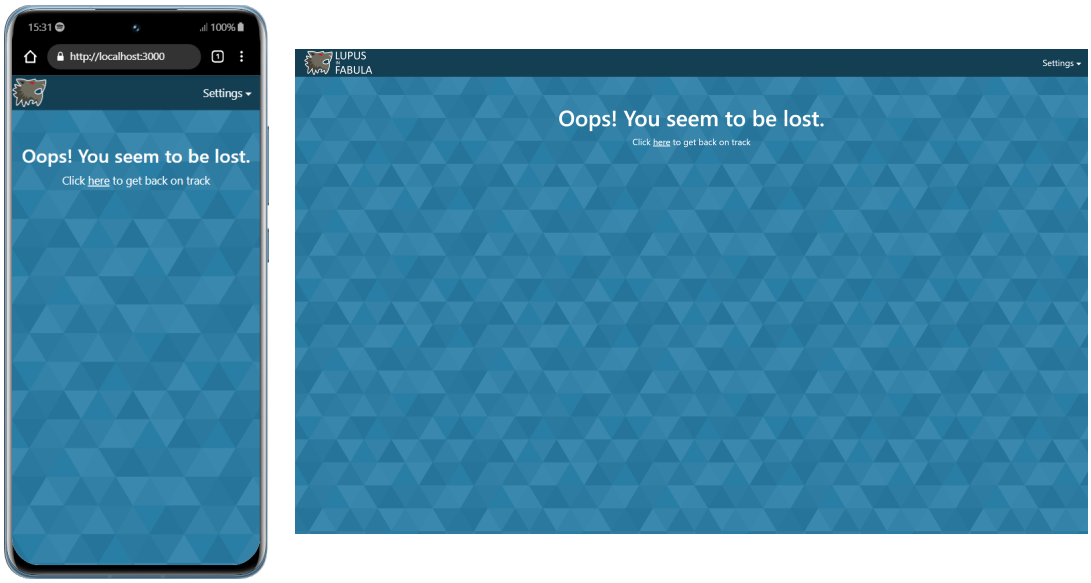

Figura 44: Schermata 404 indirizzo non trovato

8 Conclusioni

8.1 Sviluppi futuri

In futuro ampliando e approfondendo il progetto i punti salienti che presentano possibilità di ulteriori sviluppi sono risultati:

- **Applicazione mobile:** come era stato ipotizzato in fase di definizione delle specifiche di progetto sarebbe interessante, oltre che utile per migliorarne la fruibilità, sviluppare un'applicazione mobile che faccia uso di librerie specifiche.
- **Comunicazione interna:** attualmente il sistema prevede che per apprezzare al meglio il gioco implementato gli utenti si ritrovino nello stesso luogo, o che utilizzino sistemi alternativi di comunicazione, come ad esempio Discord. In futuro sfruttando le capacità di PeerJS sarebbe interessante aggiungere la possibilità di comunicare all'interno della lobby, testualmente, con un canale audio, o addirittura video.

8.2 Nozioni apprese

Lo sviluppo di questo progetto ha permesso di approfondire, e familiarizzare con, i framework e i tool utilizzati, oltre che apprendere attraverso tentativi ed errori quali fossero le metodologie migliori per impostare un lavoro di questo tipo.

Il risultato finale ottenuto rispetta tutti i requisiti fondamentali definiti in fase di analisi e proposta di progetto, oltre che tutte le aspettative dal punto di vista di usabilità e funzionalità del sistema. Dopo aver effettuato test con amici e colleghi non solo il livello di implementazione è stato ritenuto più che soddisfacente, ma anche l'idea alla base della digitalizzazione del gioco ha riscosso molto successo, con numerose richieste per una futura release così da poter continuare a utilizzarlo.

I problemi riscontrati durante lo sviluppo hanno a volte rappresentato sfide importanti, soprattutto non avendo la possibilità di confrontarsi con un gruppo, ma per ognuno è stata trovata una soluzione o una via alternativa che nel complesso ho sempre ritenuto all'altezza delle personali aspettative.

Riferimenti bibliografici

[1] Mafia (gioco) - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Mafia_(gioco).

[2] Party (role-playing games) - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Party_(role-playing_games).

[3] Matchmaking (video games) - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Matchmaking_(video_games).

[4] MEAN (solution stack) - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/MEAN_(solution_stack).

[5] Node.js - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Node.js.

[6] npm Inc. npm — npm Docs — docs.npmjs.com. https://docs.npmjs.com/cli/v6/commands/npm.

[7] Facebook Inc. Introduzione a JSX – React — it.reactjs.org. https://it.reactjs.org/docs/introducing-jsx.html.

[8] Facebook Inc. Componenti e Props – React — it.reactjs.org. https://it.reactjs.org/docs/components-and-props.html.

[9] facebook Inc. State e Lifecycle – React — it.reactjs.org. https://it.reactjs.org/docs/state-and-lifecycle.html.

[10] Dan Abramov and the Redux documentation authors. Redux - A predictable state container for JavaScript apps. — Redux — redux.js.org. https://redux.js.org/.

[11] Dan Abramov and the Redux documentation authors. Redux Fundamentals, Part 4: Store — Redux — redux.js.org. https://redux.js.org/tutorials/fundamentals/part-4-store.

[12] Dan Abramov and the Redux documentation authors. Redux Fundamentals, Part 3: State, Actions, and Reducers — Redux — redux.js.org. https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers.

[13] Dan Abramov and the Redux documentation authors. Redux Fundamentals, Part 2: Concepts and Data Flow — Redux — redux.js.org. https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow.

[14] Rooms — Socket.IO — socket.io. https://socket.io/docs/v4/rooms/.

[15] bcrypt - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Bcrypt.

[16] react-notifications — npmjs.com. https://www.npmjs.com/package/react-notifications.

[17] bcrypt — npmjs.com. https://www.npmjs.com/package/bcrypt.

[18] GitHub - reduxjs/redux-devtools: DevTools for Redux with hot reloading, action replay, and customizable UI — github.com. https://github.com/reduxjs/redux-devtools.

[19] axe DevTools — Developer Tools for Accessibility Testing — deque.com. https://www.deque.com/axe/devtools/.

[20] GitHub - GoogleChrome/lighthouse: Automated auditing, performance metrics, and best practices for the web. — github.com. https://github.com/GoogleChrome/lighthouse.

[21] Overview — ngrok documentation — ngrok.com. https://ngrok.com/docs/.

[22] GitHub - gitname/react-gh-pages: Deploying a React App (created using create-react-app) to GitHub Pages — github.com. https://github.com/gitname/react-gh-pages.

[23] Peer-to-peer - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Peer-to-peer.

[24] How To Use MERN Stack: A Complete Guide — mongodb.com. https://www.mongodb.com/languages/mern-stack-tutorial.

[25] OpenJS Foundation. Node.js — nodejs.org. https://nodejs.org/en/.

[26] OpenJS Foundation. Express - Node.js web application framework — expressjs.com. https://expressjs.com/.

[27] Facebook Inc. Primi Passi – React — it.reactjs.org. https://it.reactjs.org/docs/getting-started.html.

[28] React (web framework) - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/React_(web_framework).

[29] Mark Otto, Jacob Thornton, and Bootstrap contributors. Bootstrap — getbootstrap.com. https://getbootstrap.com/.

[30] Mark Otto, Jacob Thornton, and Bootstrap contributors. Get started with Bootstrap — getbootstrap.com. https://getbootstrap.com/docs/5.3/getting-started/introduction/.

[31] PeerJS - Simple peer-to-peer with WebRTC — peerjs.com. https://peerjs.com/.

[32] Mongoose v8.1.2: Schemas — mongoosejs.com. https://mongoosejs.com/docs/guide.html.

[33] Introduction — Socket.IO — socket.io. https://socket.io/docs/v4/.

[34] Middlewares — Socket.IO — socket.io. https://socket.io/docs/v4/middlewares/.

[35] Introduction — react.i18next.com. https://react.i18next.com/.

[36] Internationalization and localization - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Internationalization_and_localization.