

Uno Final Report

Giacomo Totaro

`giacomo.totaro2@studio.unibo.it`

Serena Bertaccini

`serena.bertaccini@studio.unibo.it`

Settembre 2024

Questo progetto nasce con l'obiettivo di sviluppare una versione digitale del gioco da tavolo Uno, un gioco conosciuto ormai da tutti, molto popolare. Ciascun utente, tramite un applicativo web, potrà creare o partecipare ad una stanza di gioco nella quale giocare con altri utenti. La partita si avvierà appena il numero di partecipanti presenti nella stanza sarà uguale a 4. Il gioco si compone di un mazzo di carte suddivise in quattro colori principali: rosso, giallo, verde e blu. Ogni colore presenta carte numerate da 0 a 9, oltre a una serie di carte speciali. Queste includono "Salta", che consente di saltare il turno del giocatore successivo; "Inverti", che cambia la direzione del gioco; "Pesca Due", che obbliga il giocatore successivo a pescare due carte; "Jolly", che permette al giocatore di cambiare il colore in gioco; e "Pesca Quattro", una versione potenziata del "Jolly" che consente anche di far pescare quattro carte al giocatore successivo. A inizio partita, ogni giocatore riceve 7 carte, mentre una viene scoperta al centro per formare la pila degli scarti. Ciascun giocatore, al proprio turno, deve scartare una carta che corrisponda per numero o colore a quella in cima alla pila degli scarti. In alternativa, il giocatore può giocare una carta speciale, indipendentemente dal colore della carta in cima alla pila. Se un giocatore non può giocare una carta valida, deve pescare una carta dal mazzo e, se ancora non può giocare, deve passare il turno. L'obiettivo del gioco è rimanere senza carte in mano prima degli altri giocatori. Quando un giocatore rimane con una sola carta in mano, deve dichiarare "Uno" per informare gli altri. Se un giocatore non dichiara "Uno" prima che il turno passi, deve pescare due carte come penalità. Il gioco si conclude quando rimane un giocatore con ancora carte in mano.

1 Goal/Requirements

Nella seguente sezione verranno riportate una serie di Q/A per spiegare più dettagliatamente i diversi aspetti del progetto, in particolare obiettivi da raggiungere, requisiti e risultati attesi.

1.1 Questions/Answers

- *Cosa succede una volta avviato il gioco ?*
 - All'avvio dell'applicativo, l'utente è invitato a inserire un nickname, il quale sarà memorizzato nel "local storage" del browser.
- *Quali opzioni sono disponibili dopo il login?*
 - L'utente ha la possibilità di creare una nuova lobby, unirsi a una esistente o consultare le regole del gioco.
- *Come viene gestita la creazione delle lobby?*
 - Una volta creata una lobby, essa sarà visibile nella lista delle lobby aperte, in attesa dell'arrivo degli altri giocatori. Al raggiungimento del quarto giocatore, la partita inizierà automaticamente.
- *Quanti utenti possono partecipare contemporaneamente?*
 - Il sistema è progettato per supportare partite simultanee, con un massimo di 4 utenti per ciascuna partita.
- *Quali azioni può compiere un utente durante la partita?*
 - Durante il proprio turno, l'utente deve selezionare una carta valida dalla propria mano o, in alternativa, pescare una carta. Se il timer di 30 secondi scade senza che l'utente abbia effettuato una mossa, una carta verrà pescata automaticamente e il turno passerà al giocatore successivo.
- *Cosa accade al termine della partita?*
 - Quando un giocatore esaurisce le carte in mano, viene dichiarato vincitore e attende la conclusione della partita. La partita si considera terminata quando resta un solo utente con carte in mano.

1.2 Scenarios

In uno scenario di utilizzo, un utente, per accedere al gioco, dovrà connettersi al sistema inserendo un nickname. Successivamente, verrà indirizzato alla Home Page, dove avrà la possibilità di creare una nuova lobby, unirsi a una già esistente o consultare le regole del gioco.

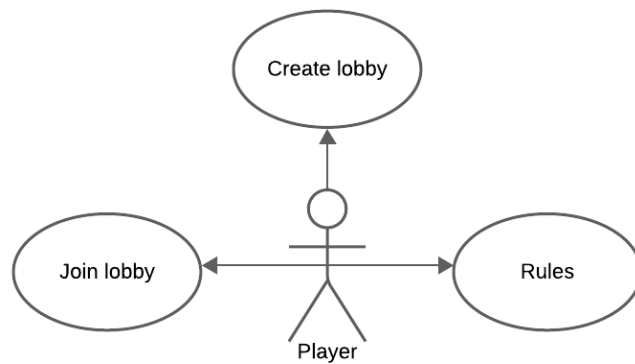


Figure 1: Home Page

Qualora l'utente opti per la creazione di una nuova lobby, sarà richiesto di inserire un nome per la stanza; una volta confermata l'operazione, il client invierà una richiesta al server tramite socket, il quale provvederà alla creazione della stanza con l'utente come primo partecipante.

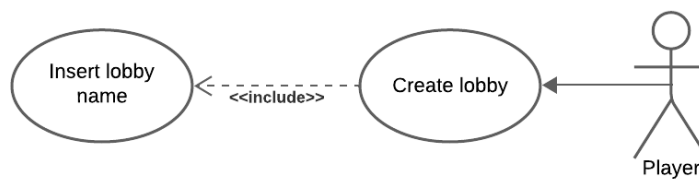


Figure 2: Create Lobby

In alternativa, l'utente potrà scegliere di entrare in una lobby esistente premendo il pulsante "Join lobby", che mostrerà tutte le stanze disponibili in cui sarà possibile connettersi tramite il pulsante "Join Game". In questo caso, il client invierà al server, sempre tramite socket, una richiesta di join per la stanza scelta.

Quando nella lobby saranno presenti quattro giocatori, la partita inizierà automaticamente. Ogni utente avrà a disposizione un turno della durata di 30 secondi, durante il quale dovrà decidere se giocare una carta o pescare. Se al termine del proprio turno l'utente non ha effettuato alcuna mossa, pescherà una carta e il turno passerà automaticamente al giocatore successivo. Una volta terminata la partita, il creatore della lobby potrà decidere se avviare un "re-match" o uscire e tornare alla Home Page.

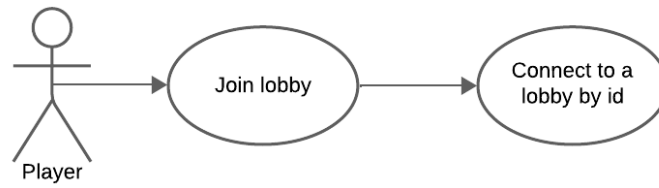


Figure 3: Join Lobby

1.3 Self-assessment policy

Al fine di garantire il controllo della qualità e l'efficacia del progetto, sono stati realizzati dei test automatici (CI) sfruttando il framework Jest. I test, hanno come obiettivo verificare tutte le funzionalità principali del sistema, a partire da test più semplici fino alla realizzazione di test più complessi.

2 Requirements Analysis

2.1 User

- Creazione di una lobby;
- Avvio della partita, se si ha creato la lobby;
- Entrare/Uscire da una lobby esistente;
- Selezionare una carta valida o pescare in partita;
- Dichiarare *Uno* in partita;
- Visualizzare ranking finale;
- Abbandonare una partita in corso.

2.2 Functional

- Premendo il pulsante *Create lobby*, l'applicazione crea una stanza se è stato inserito un nome;
- Premendo il pulsante *Join lobby* vengono visualizzate tutte le stanze disponibili;
- Premendo il pulsante *Rules* vengono visualizzate le regole del gioco;
- Premendo sul pulsante *Join Game* si accede alla stanza selezionata;
- Il gioco non può avviarsi se ci sono meno di quattro giocatori nella stanza;

- La partita deve terminare se ci sono meno di quattro giocatori;
- Se un giocatore non dichiara *Uno* prima che il giocatore successivo inizi il proprio turno, allora deve pescare due carte;
- Un giocatore ha 30 secondi per completare il turno; in caso contrario, deve pescare una carta e cedere il turno al successivo;
- Un giocatore può giocare una sola carta per turno;
- La partita si conclude quando rimane un solo giocatore con carte in mano.

2.3 Non-Functional

- Interfaccia grafica semplice e intuitiva;
- Consistenza delle informazioni;
- Il sistema deve essere compatibile con i principali browser web e con i sistemi operativi più diffusi, in modo da essere accessibile a un vasto pubblico;
- Il gioco deve essere stabile e non deve presentare malfunzionamenti o interruzioni durante la partita;
- Il gioco deve essere veloce e reattivo, in modo da garantire un'esperienza di gioco fluida.

2.4 Technologies

L'applicazione è stata sviluppata utilizzando lo stack MERN, che comprende MongoDB come database, Node.js ed Express.js come framework di backend, e React come libreria per il frontend. Per la gestione del progetto, è stato scelto il framework Vite, noto per le sue tecniche innovative che garantiscono prestazioni elevate e un'esperienza di sviluppo fluida. Inizialmente avevamo considerato l'utilizzo di Vue, già studiato durante l'anno scolastico, ma abbiamo optato per React, al fine di ampliare le nostre competenze e familiarizzare con una libreria ampiamente adottata nel settore dello sviluppo web.

La struttura del codice è stata implementata in TypeScript, una versione tipizzata di JavaScript che introduce tipi, classi, interfacce e moduli opzionali, migliorando così la qualità del codice e facilitando la manutenzione.

La connessione tra client e server è stata realizzata tramite il protocollo WebSocket, utilizzando la libreria *Socket.io* che semplifica l'implementazione della logica di comunicazione in tempo reale tra client e server, consentendo anche la diffusione di messaggi a più client.

La gestione delle dipendenze è stata affidata a *npm*, mentre per i test è stata impiegata la libreria *Jest*, che ha permesso di analizzare la coverage del progetto e di implementare vari test.

Vite, utilizzato come build tool, si distingue per l'uso di moduli ES nativi e moderne API

del browser, che consentono la compilazione “on the fly” del codice, garantendo tempi di build rapidi e aggiornamenti istantanei nel browser. Inoltre, il server di sviluppo integrato in Vite è ottimizzato per un reloading rapido e per la sostituzione dei moduli senza interrompere l’esecuzione, permettendo agli sviluppatori di vedere in tempo reale le modifiche al codice senza dover ricaricare l’intera pagina.

In conclusione, l’utilizzo dello stack MERN combinato con tecnologie avanzate come TypeScript, Vite e Socket.io, ha permesso di realizzare un’applicazione moderna, performante e facilmente manutenibile.

3 Design

Durante lo sviluppo del progetto, sono stati scelti diversi pattern React per la loro efficacia e la loro flessibilità. In particolare sono stati utilizzati i seguenti pattern:

- **Provider Pattern:** Il provider pattern è un modello di progettazione molto efficace per la condivisione di dati globali tra i vari componenti di un’applicazione React. Esso prevede la creazione di un componente Provider che gestisce e distribuisce i dati a tutti i componenti figli tramite un Consumer o hook personalizzati. Questo approccio semplifica la gestione dei dati, riducendo la necessità di passare manualmente le props tra componenti e migliorando l’efficienza e la leggibilità del codice. Il pattern è particolarmente utile in contesti come la gestione dello stato globale dell’applicazione, ad esempio nel nostro caso è stato utilizzato per avvolgere l’intero albero dei componenti. Questo consente a tutti i componenti figli di accedere allo store Redux.

```
const AppRoute = (props: any) => {
  const [loadingAssets, setLoadingAssets] = useState(true);

  const onLoaded = () => {
    setLoadingAssets(false);
  };

  const location = useLocation();

  if (loadingAssets) return <Loading onLoaded={onLoaded} />;

  return (
    <Root>
      <Provider store={store}>
        <AnimatePresence mode='wait'>
          <Routes location={location} key={location.key}>
            <Route key="/" path="/" element={<StartPage />} />
            <Route key="/create-user" path="/create-user" element={<CreateUser />} />
            <Route key="/home" path="/home" element={<Home />} />
            <Route key="/lobby" path="/lobby" element={<Lobby />} />
            <Route key="/create" path="/create" element={<CreateLobby />} />
            <Route key="/waiting" path="/waiting" element={<WaitingLobby {...props} />} />
            <Route key="/game" path="/game" element={<Game {...props} />} />
            <Route key="/rules" path="/rules" element={<Rules />} />
            <Route key="*" path="*" element={<ErrorRoute />} />
          </Routes>
        </AnimatePresence>
      </Provider>
    </Root>
  )
}
export default AppRoute
```

Figure 4: Esempio Provider Pattern

- **Hooks:** Gli hooks hanno rappresentato una svolta significativa nella progettazione dei componenti React, offrendo un modo semplice per accedere a funzionalità come props, stato e contesto. Introdotti per l'uso nei componenti funzionali, hanno permesso di gestire lo stato interno e il ciclo di vita del componente, superando i limiti dei componenti di classe. Grazie agli hooks, il codice è diventato più modulare, leggibile e riutilizzabile, facilitando la gestione delle interazioni tra componenti e ottimizzando le prestazioni del rendering, evitando operazioni superflue.

```
// hooks.ts
import {
  TypedUseSelectorHook,
  useDispatch as useReduxDispatch,
  useSelector as useReduxSelector,
} from "react-redux";
import { AppDispatch, RootState } from "../store/store";

export const useAppDispatch = () => useReduxDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useReduxSelector;

export default function DrawingStack() {
  const { drawingStack, currentPlayer } = useAppSelector(drawingStackAndCurrentPlayerSelector);
  const colorSelection = useAppSelector((state) => state.game.colorSelection);
  const alreadyDrawn = useAppSelector((state) => state.game.alreadyDrawn);

  const dispatch = useAppDispatch();

  const [gameStarted, setGameStarted] = useState(false);

  const handleClick = () => {
    if (currentPlayer === 0 && !colorSelection && !alreadyDrawn) API.move(true, "");
  };

  useEffect(() => {
    setTimeout(() => {
      dispatch(ready());
      setGameStarted(true);
    }, 2000);
  }, [dispatch]);
}
```

Figure 5: Esempio Hooks

- **State Reducer Pattern:** Il pattern state reducers di React permette una gestione ordinata e flessibile dello stato di un componente, basata su criteri specifici. In questo pattern, il componente genitore fornisce al componente figlio una funzione come prop, incaricata di aggiornare lo stato del figlio secondo la logica definita dal genitore. Questo approccio è particolarmente utile per gestire stati complessi, come la validazione dei dati o la gestione di eventi asincroni, separando la logica di stato dal componente figlio e rendendo il codice più leggibile. Di seguito un esempio di implementazione 6.

L'utilizzo di questi pattern ha consentito di sviluppare il progetto in modo modulare e flessibile, semplificando la gestione dei dati globali e la condivisione di logica e stato tra i componenti.

```

export const gameSlice = createSlice({
  name: 'game',
  initialState: initialState,
  reducers: {
    setPlayerId(state, action) { ...
    },
    init: (state, action) => { ...
    },
    ready(state) { ...
    },
    stopGame(state) { ...
    },
    initGame(state) { ...
    },
    setInLobby(state, action) { ...
    },
    moveCard(state, action: PayloadAction<{ ...
    },
    finalMovePlayer(state, action: PayloadAction<{ ...
    },
    setFirstCard(state, action: PayloadAction<{ ...
    },
    moveFirstCard(state, action: PayloadAction<{ ...
    },
    setColorSelection(state, action: PayloadAction<{ ...
    },
    draw2Cards(state, action: PayloadAction<{ ...
    },
    finalPlayerPass(state, action: PayloadAction<{ ...
    },
  },
});

```

Figure 6: Esempio State Reducer Pattern

3.1 Structure

Il progetto si concentra sulla modellazione di due entità principali:

- Giocatore
- Lobby

Il mazzo di carte viene generato dinamicamente durante l'esecuzione, caricando i dati dal database.

Nella figura 7 viene riportato il diagramma delle classi che illustra le entità principali del dominio e, per ognuna di essa, vengono riportati i suoi attributi e i metodi principali.

La struttura del progetto è suddivisa in due moduli principali: Client e Server.

3.1.1 Client

Il modulo Client gestisce l'intera logica dell'applicazione, consentendo agli utenti di interagire con il gioco. Le sue componenti principali includono:

- **asset**: contiene tutte le immagini e le icone utilizzate nell'interfaccia;

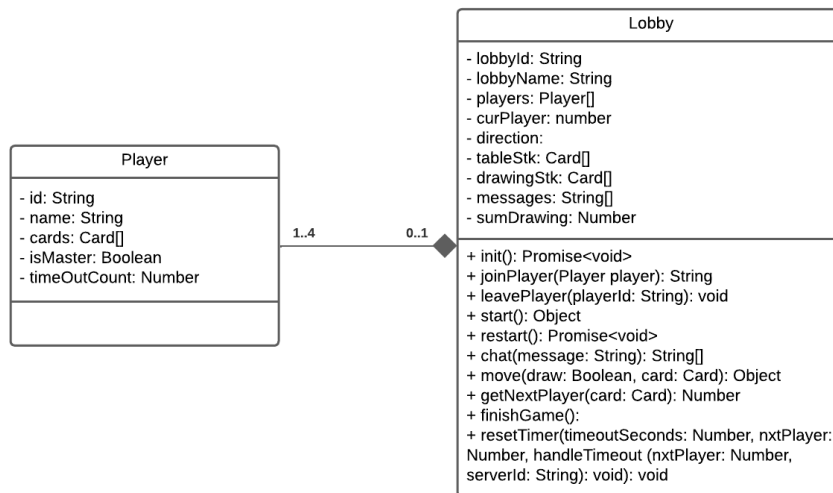


Figure 7: Class Diagram

- **api**: definisce le funzioni per la gestione delle socket e della comunicazione in tempo reale tra client e server;
- **components**: contiene tutti i componenti dell'interfaccia da renderizzare;
- **hooks**: include le funzioni utilizzate per la gestione dello stato e delle interazioni nel client;
- **pages**: gestisce il rendering e la logica delle diverse pagine del gioco;
- **routes**: definisce le rotte per la navigazione tra le varie pagine dell'applicazione;
- **store**: consente di memorizzare variabili localmente;
- **style**: contiene tutti i file CSS per la gestione dell'aspetto grafico;
- **utils**: include tutte le interfacce e le funzioni comuni utilizzate in più file;
- **test**: raccoglie i file di test per verificare la corretta implementazione del codice.

3.1.2 Server

Il modulo Server gestisce l'amministrazione dei client, delle lobby e degli eventi durante una partita. Le sue componenti principali sono:

- **index**: gestisce la connessione al database, crea e avvia il server;
- **socket**: si occupa della gestione degli eventi lato server;
- **utils**: racchiude e gestisce la logica di gioco;

- **test:** contiene i file di test per la verifica dell'implementazione del codice.

In sintesi, il progetto è stato strutturato in modo logico e modulare, con il modulo Client dedicato alla gestione della logica dell'applicazione e il modulo Server responsabile della logica di gioco. Questa suddivisione in componenti ha facilitato lo sviluppo e la manutenzione del codice, assicurando al contempo la qualità e la scalabilità del progetto. L'inclusione di file di test per entrambi i moduli ha consentito di verificare l'implementazione del codice e di rilevare tempestivamente eventuali anomalie.

3.2 Behaviour

Nella figura 8 viene riportato il diagramma degli stati che descrive il comportamento della logica iniziale del sistema ovvero, dal momento in cui l'utente avvia l'applicazione fino all'inizio della partita.

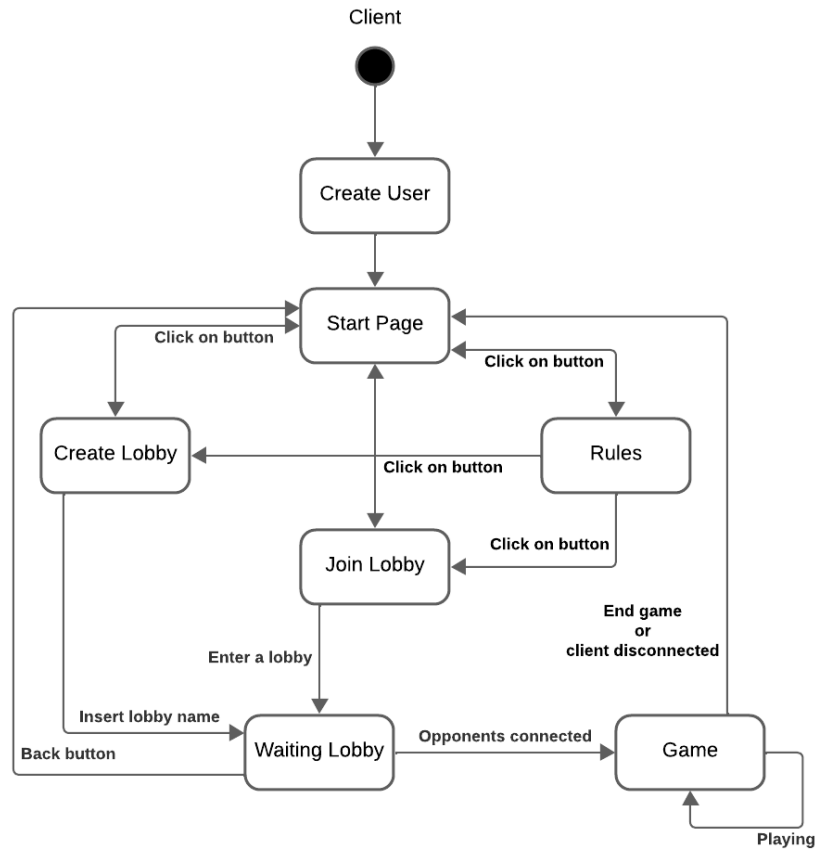


Figure 8: State Diagram - Initial Logic

All'avvio dell'applicazione, l'utente è invitato a inserire un nickname, dopodiché viene

indirizzato alla Home Page. In questa sezione, l'utente può visualizzare le stanze già create da altri giocatori e scegliere se unirsi a una di esse, crearne una nuova o consultare le regole del gioco. Se l'utente decide di creare una nuova stanza, dovrà attendere la connessione degli altri partecipanti prima di avviare la partita. In alternativa, se l'utente desidera unirsi a una stanza esistente, può semplicemente selezionare una delle lobby disponibili nella lista, e la partita avrà inizio.

3.3 Interaction

In questa sezione verranno mostrati diversi diagrammi di sequenza per illustrare le principali interazioni del sistema.

All'avvio dell'applicazione l'utente si trova nel menù principale e può scegliere di creare una lobby o unirsi a una esistente (la pagine delle regole non viene modellata perchè non ci sono interazioni particolari con il server).

3.3.1 Lobby creation

Premendo sul pulsante "Create Lobby" verrà chiesto all'utente di inserire un nome e il server si occuperà di verificare che il nome non sia troppo corto, in caso positivo verrà creata.

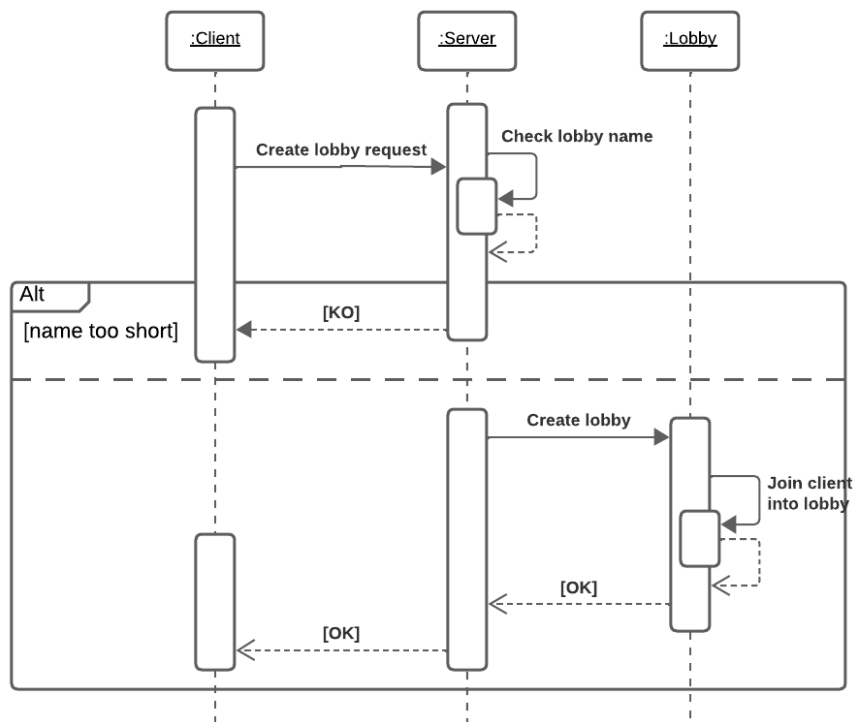


Figure 9: Sequence Diagram - Create Lobby

3.3.2 Join lobby

Premendo sul pulsante “Join Lobby” viene visualizzata la lista di tutte le lobby esistenti e si può scegliere di unirsi a una di esse.

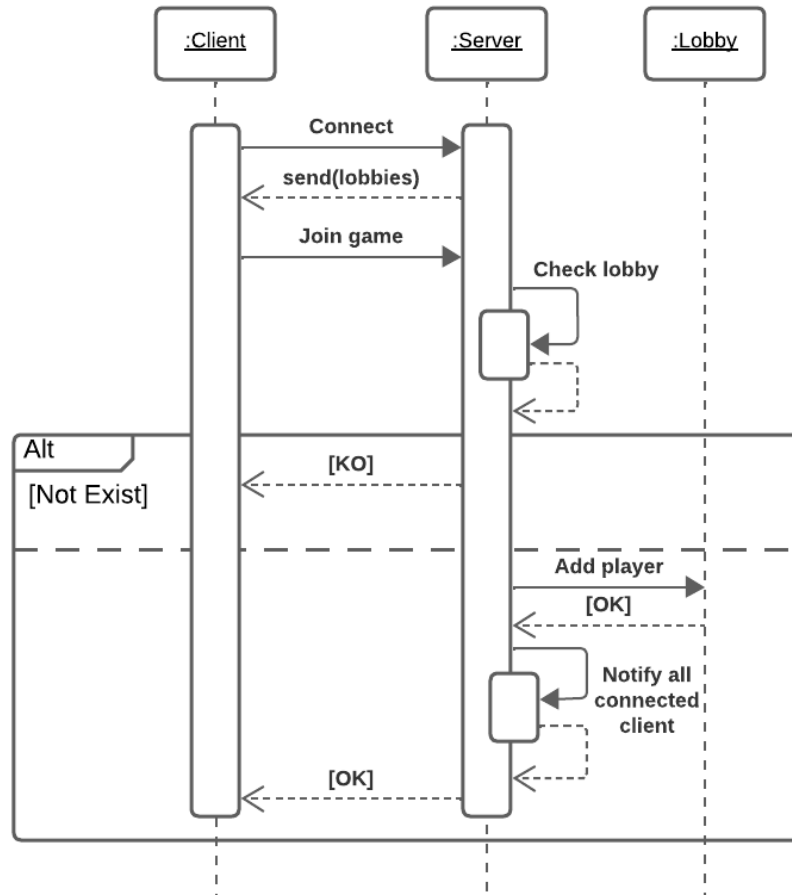


Figure 10: Sequence Diagram - Join Lobby

3.3.3 Game

La partita inizia automaticamente quando la lobby è composta da quattro giocatori. Ognuno di essi avrà sette carte in mano e, al proprio turno, dovrà scegliere di scartare una carta valida oppure pescarne una.

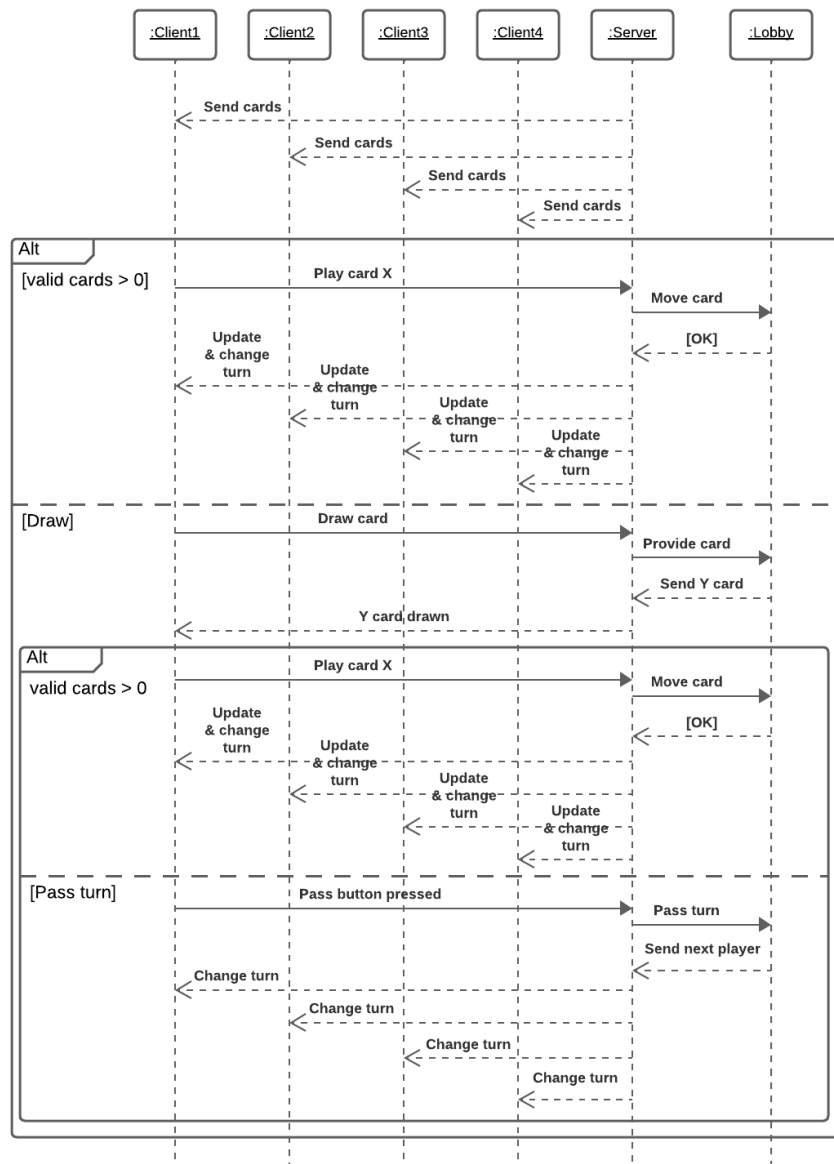


Figure 11: Sequence Diagram - Game Logic

4 Implementation Details

In questa sezione verranno riassunte le strategie di implementazione utilizzate. Il sistema è composto principalmente da due parti: Server e Client.

4.1 Server

Il server è stato sviluppato utilizzando TypeScript, un superset di JavaScript, ed eseguito su Node.js [3]. Per la comunicazione tra client e server è stato adottato il protocollo WebSocket tramite l'utilizzo di Socket.io [4]. Questa scelta è stata motivata dall'esigenza di garantire aggiornamenti in tempo reale ai client connessi, in seguito alle interazioni di altri utenti con il sistema.

Le WebSocket consentono una comunicazione bidirezionale in tempo reale tra client e server, rendendole particolarmente adatte per la realizzazione di giochi multiplayer online, dove è essenziale che il server possa inviare dati ai client anche senza una richiesta esplicita da parte loro, a differenza del protocollo HTTP che è unidirezionale.

La server socket sviluppata per questo progetto gestisce tutte le interazioni tra i client e il sistema, dalla creazione e connessione a una stanza fino alla gestione completa del flusso di una partita. Il server comunica con i client attraverso due principali metodi:

- *emit*: Questo metodo permette di inviare messaggi specifici a un singolo client oppure a tutti i client connessi. In quest'ultimo caso, il server invia lo stesso messaggio a tutti i partecipanti senza distinzione, favorendo una comunicazione sincronizzata tra tutti i client.
- *broadcast*: Permette di inviare un messaggio a tutti i client connessi, eccetto quello che ha originato l'evento. Questo garantisce che le informazioni siano distribuite in modo uniforme tra tutti i partecipanti, mantenendo la coerenza del gioco senza interferire con l'utente che ha generato l'azione.

Il WebSocket server è definito nel file *index.ts* ed è configurato per ascoltare le richieste sulla porta 3000. Tutte le operazioni legate alla logica del sistema sono rigorosamente controllate per garantire l'integrità e la sicurezza, prevenendo operazioni non autorizzate o potenzialmente dannose.

4.2 Client

Il client è stato sviluppato in TypeScript e React. L'utilizzo di Socket.io lato client è reso possibile grazie a specifiche API che permettono di gestire le comunicazioni in tempo reale. Queste API facilitano l'invio e la ricezione di dati tra client e server. Tra i metodi principali utilizzati ci sono:

- *socket.emit(eventName, data)*: consente di inviare un messaggio specifico al server, accompagnato da eventuali dati pertinenti. Questo metodo è fondamentale per attivare eventi specifici lato server e trasferire informazioni cruciali, come le azioni del giocatore o le scelte nel gioco;
- *socket.on(eventName, callback)*: permette di definire una funzione di callback che verrà eseguita ogni volta che il client riceve un evento con il nome specificato. Questo metodo è essenziale per gestire aggiornamenti in tempo reale, come il cambiamento di stato di una partita o le azioni compiute dagli altri giocatori;

- *socket.off(eventName, callback)*: disconnette un determinato listener associato a un evento specifico. Questo metodo è utile per gestire correttamente le risorse del client e prevenire la duplicazione di listener che potrebbero portare a comportamenti imprevisti o a problemi di memoria.
- *socket.removeAllListeners(eventName?)*: rimuove tutti i listener associati a un evento specifico, o se nessun evento è specificato, rimuove tutti i listener registrati per il socket. Questo metodo è particolarmente utile quando si desidera ripulire il client da eventuali listener rimasti attivi prima di chiudere una connessione o di passare a una nuova fase del gioco.

I percorsi che consentono all'utente di visualizzare le interfacce grafiche sono:

- `/`: pagina iniziale;
- `/create-user`: schermata per la creazione dell'utente, in cui è possibile inserire il nome;
- `/home`: pagina principale, dove l'utente può decidere se creare una stanza, unirsi a una stanza esistente o consultare le regole del gioco;
- `/lobby`: schermata che elenca le lobby create dagli altri utenti;
- `/create`: pagina per la creazione di una nuova lobby;
- `/waiting`: schermata di attesa, dove l'utente può vedere gli altri giocatori nella lobby in attesa che questa si completi;
- `/game`: pagina in cui si svolge la partita;
- `/rules`: schermata dedicata alla visualizzazione delle regole di gioco;

4.3 Ensure reliability

Per evitare il blocco del sistema sono state adottate le seguenti precauzioni:

- Timer di gioco: Durante la partita, ogni giocatore dispone di un massimo di 30 secondi per effettuare la propria mossa. Se il tempo scade senza che sia stata eseguita un'azione, il turno passa automaticamente al giocatore successivo e viene pescata una carta. Nel caso in cui un giocatore lasci scadere il timer per tre volte consecutive, verrà disconnesso dalla partita.
- Gestione delle disconnessioni: In caso di disconnessione di un giocatore durante una partita in corso, una notifica viene inviata agli altri partecipanti e la partita viene interrotta immediatamente.

5 Self-assessment / Validation

Il testing è stato realizzato tramite test automatici utilizzando il framework **Jest** [2] insieme a **Babel** [1]. In particolare, abbiamo implementato tutti i vari test necessari per garantire la qualità del codice e analizzare la coverage del progetto. Per i test lato client, abbiamo testato l'interfaccia utente implementata con React, verificando che i componenti funzionassero correttamente e che gli eventi scatenati dagli utenti venissero gestiti in modo appropriato. Inoltre, è stato possibile testare la gestione dello stato dell'applicazione e la navigazione tra le diverse pagine.

Per quanto riguarda invece i test lato server, ci siamo impegnati a implementare un sistema di test completo e affidabile per l'applicazione, verificando che i vari endpoint rispondessero correttamente alle richieste e restituissero i dati attesi. Inoltre, è stato possibile testare la gestione degli errori.

Per eseguire i test è necessario seguire i seguenti passi:

1. Posizionarsi nella cartella **mongo** del progetto e eseguire il comando **docker compose up**;
2. Dopodichè posizionarsi nella cartella **server** o **client** del progetto;
3. Eseguire il comando **npm install** per installare tutte le dipendenze necessarie;
4. Eseguire il comando **npm test** per eseguire i test.

6 Deployment Instructions

Per eseguire il deployment di database, server e client, è necessario avere Node.js e Docker installati sul proprio sistema.

6.1 Database

Per avviare un'istanza locale di MongoDB, procedere come segue:

1. Navigare nella directory **mongo** del progetto;
2. Aprire Docker ed eseguire il comando **docker compose up**.

Il database sarà accessibile all'indirizzo <http://localhost:8081>.

6.2 Server

Per effettuare il deployment locale del server, seguire i passaggi riportati di seguito:

1. Spostarsi nella directory **server** del progetto;
2. Eseguire il comando **npm install** per installare le dipendenze necessarie;
3. Eseguire il comando **npm run build** per compilare il codice Typescript;

4. Eseguire il comando `npm start` per avviare il server.

Il server sarà attivo all'indirizzo `http://localhost:3000`.

6.3 Client

Per avviare il client in locale, procedere come segue:

1. Accedere alla directory `client` del progetto;
2. Eseguire il comando `npm install` per installare le dipendenze necessarie;
3. Eseguire il comando `npm run dev` per avviare il client.

Ogni istanza del client sarà disponibile all'indirizzo URL fornito dalla console.

7 Usage Examples

Di seguito vengono illustrate le principali operazioni che un utente può eseguire all'interno dell'applicazione.

- All'avvio iniziale, l'utente è accolto da una schermata in cui deve inserire il proprio nome utente.

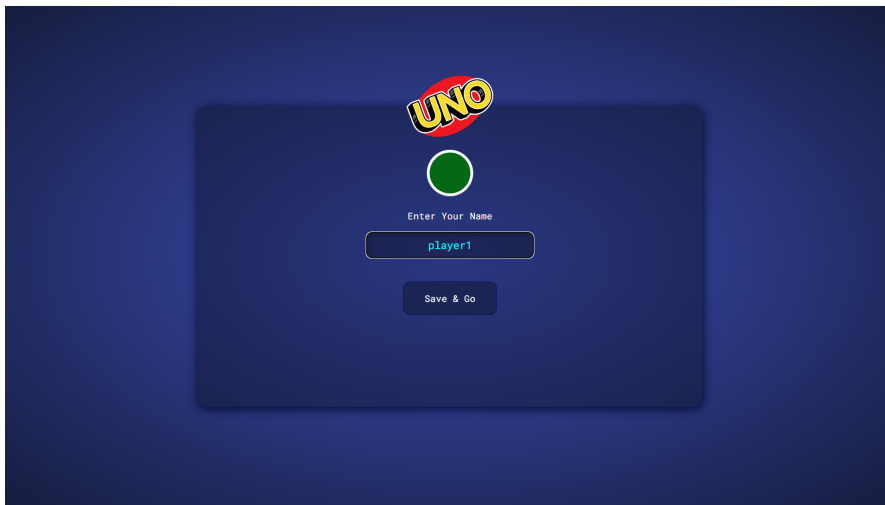


Figure 12: Login

- Successivamente, viene presentata la schermata Home, che offre tre opzioni: connettersi a una stanza già esistente, crearne una nuova, oppure consultare le regole del gioco.

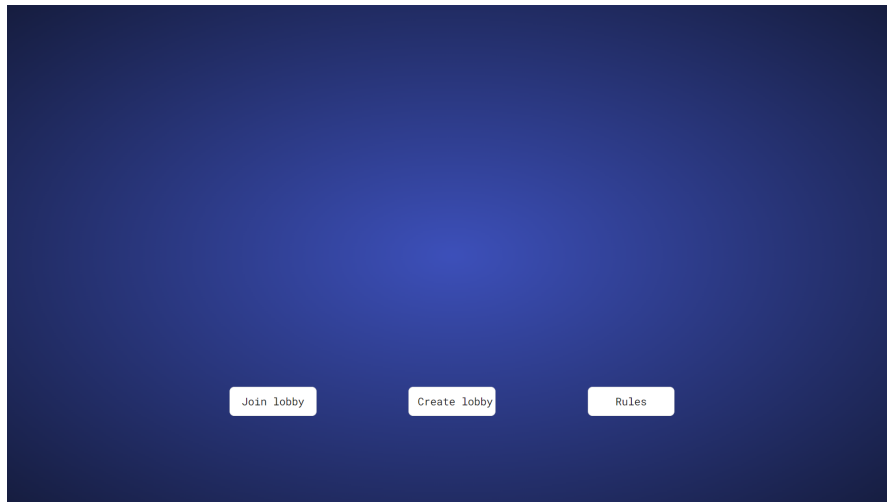


Figure 13: Home

- Se l'utente sceglie di creare una nuova stanza, viene richiesto di inserire il nome della stanza e confermare la scelta.

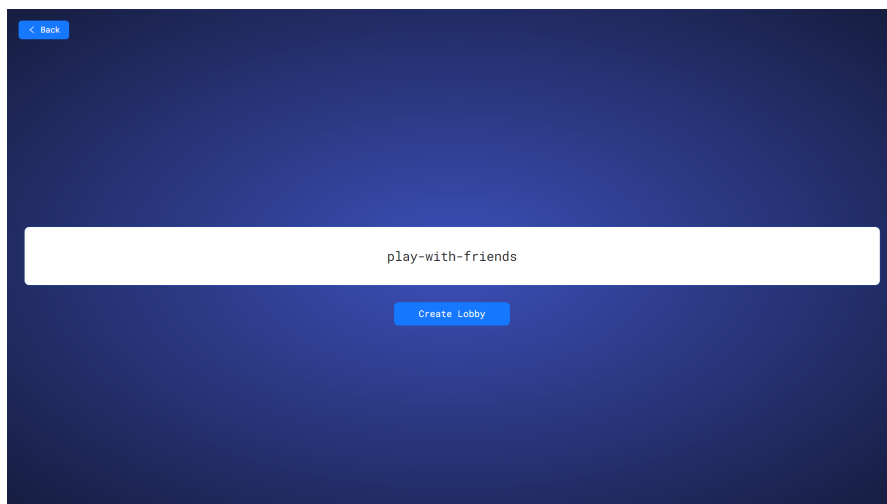


Figure 14: Creazione Lobby

- Dopo la creazione della lobby, l'utente viene reindirizzato a una pagina di attesa, dove può visualizzare i giocatori che si sono uniti fino a quel momento.



Figure 15: Attesa giocatori

- Se l'utente desidera consultare le stanze già esistenti, tramite il pulsante "Join Lobby" accede a una pagina che mostra quelle disponibili e non ancora complete. Selezionando una lobby, è possibile unirsi ad essa.

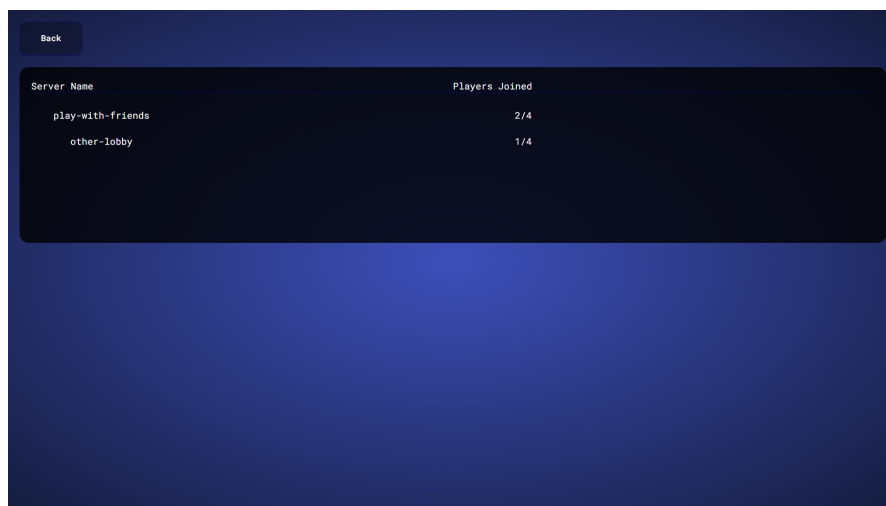


Figure 16: Selezione lobby

- Una volta che quattro giocatori si sono uniti alla lobby, la partita inizia automaticamente. Durante la partita, il giocatore può selezionare una carta dalla propria mano (se giocabile) o pescare una carta dal mazzo.



Figure 17: Partita

- Sia nella fase di attesa che durante la partita, i giocatori possono comunicare tra loro tramite il pulsante *Chat*, che apre una finestra di messaggistica sovrapposta alla pagina corrente.

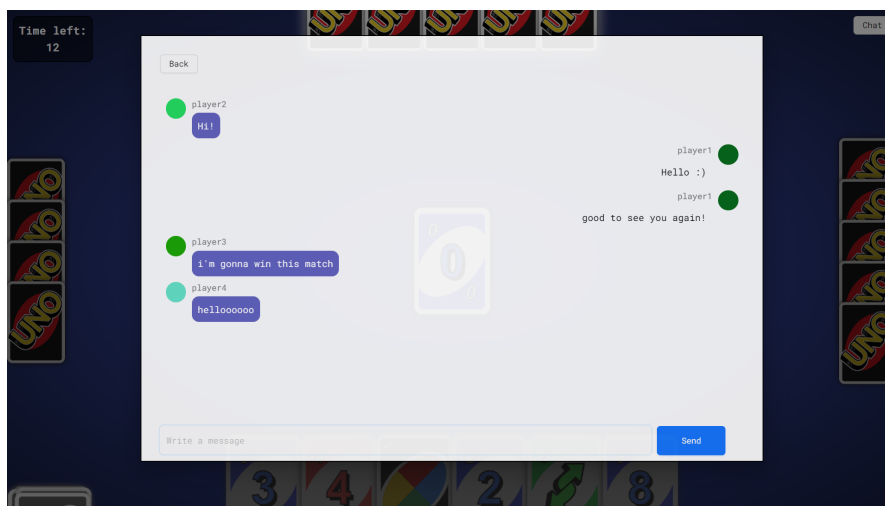


Figure 18: Chat durante la partita

- Al termine della partita, viene visualizzata una classifica con i risultati dei giocatori. L'utente che ha creato la lobby ha la facoltà di decidere se rigiocare con gli stessi partecipanti.

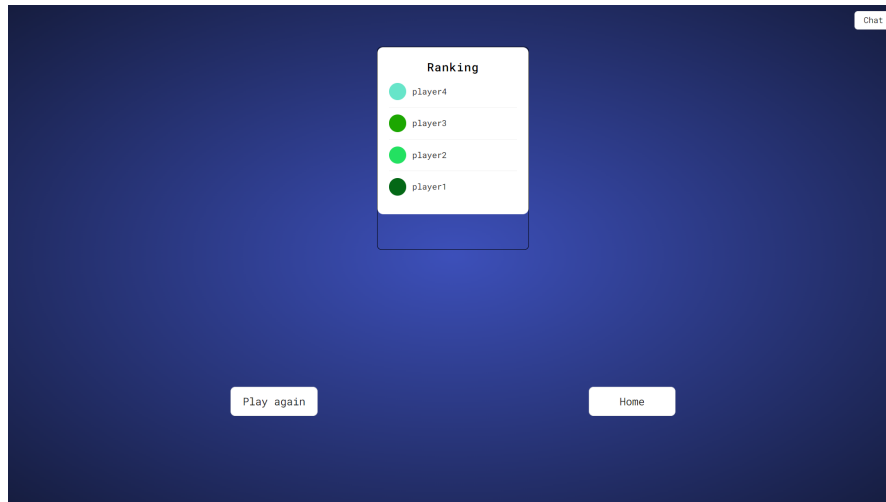


Figure 19: Classifica

8 Conclusions

Nelle conclusioni di questo progetto, possiamo affermare che gli obiettivi prefissati sono stati raggiunti con successo. La realizzazione di un sistema di gioco multiplayer online ha richiesto una combinazione efficace di tecnologie avanzate, tra cui TypeScript, React, Node.js e WebSocket, che hanno consentito di implementare un'infrastruttura robusta e scalabile. Il sistema sviluppato offre un'esperienza utente fluida, con un'interfaccia intuitiva e funzionalità in tempo reale, essenziali per garantire l'interattività tra i partecipanti.

8.1 Future Works

Nelle versioni future del gioco è possibile introdurre le seguenti migliorie:

- Gestire in maniera meno brutale la disconnessione di un giocatore, permettendo agli utenti rimasti di finire la loro partita;
- Consentire a un client che si è disconnesso in modo accidentale di potersi riconnettere alla partita;
- Salvare le informazioni degli utenti nel database, in modo da consentire a un giocatore di poter effettuare il login con le proprie credenziali e rendere persistenti i dati sulle partite giocate e poter creare una classifica globale in base alle partite vinte;
- Migliorare l'interfaccia grafica in modo tale da garantire una migliore User Experience.

8.2 What did we learned

Questo progetto ha offerto l'opportunità di applicare in modo pratico diverse competenze acquisite durante il corso, in particolare nel design e nell'implementazione di sistemi distribuiti. Inoltre, ci ha consentito di estendere le nostre conoscenze attraverso l'esplorazione e l'impiego del protocollo di comunicazione WebSocket, che non avevamo precedentemente utilizzato. Il progetto ha anche rappresentato un'esperienza formativa significativa, poiché ha permesso di approfondire i principi fondamentali legati al ciclo di vita di un progetto, dall'analisi iniziale dei requisiti, passando per la fase di implementazione, fino alla fase conclusiva di testing. Complessivamente, riteniamo che questa esperienza sia stata estremamente positiva, poiché ha contribuito a consolidare e ampliare le nostre competenze tecniche e progettuali.

References

- [1] Babel. <https://babeljs.io/docs/>.
- [2] Jest. <https://jestjs.io/>.
- [3] Node.js. <https://nodejs.org/en>.
- [4] Socket.io. <https://socket.io/docs/v4/>.