Final Report

Emanuele Dall'Ara, Nicholas Ricci

Abstract

Si vuole sviluppare una versione digitale e distribuita del gioco **“Cards Against Humanity”**. **L’obiettivo** del gioco è quello di creare delle frasi bizzarre e strampalate.

Ciascun utente, tramite un applicativo web, potrà creare o partecipare ad una stanza di gioco nella quale giocare con altri utenti. L’utente creatore della stanza potrà avviare il gioco appena il numero di partecipanti presenti in essa sarà maggiore o uguale a 3. Il gioco si compone di due mazzi di carte, uno nero e uno bianco. Ogni carta contiene delle frasi o parole, in particolare quelle nere presentano delle frasi incomplete o con delle parole mancanti. **All’inizio della partita**, un giocatore casuale verrà selezionato per essere lo **“Czar”** delle carte. Ciascun giocatore riceverà 10 carte bianche mentre lo Czar riceverà una carta nera. **Da adesso in poi il gioco si svolgerà a turni.**

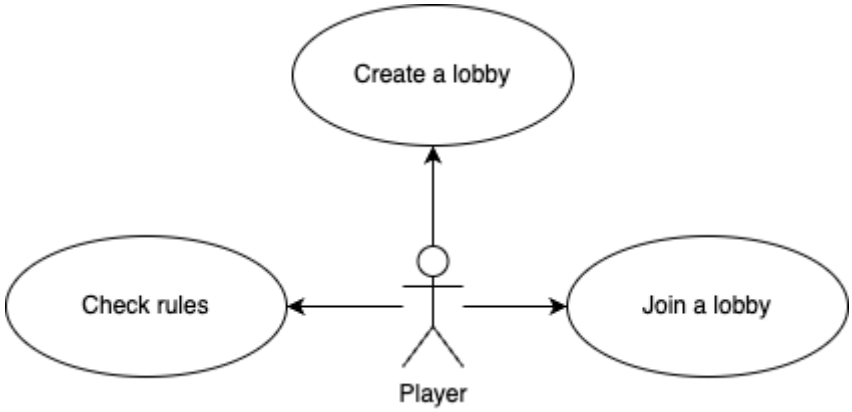
Ad ogni turno, lo Czar rivelerà la sua carta nera agli altri giocatori e ognuno di essi presenterà in risposta una carta bianca cercando di completare la frase nel modo più esilarante e inappropriato possibile. Lo Czar ora visualizzerà tutte le risposte e selezionerà quella migliore per lui. A questo punto il **vincitore** diventerà il nuovo Czar e guadagnerà un punto. Ora gli altri giocatori pescheranno tante carte bianche quante ne mancano per arrivare a 10 e il nuovo Czar perderà le sue carte bianche e pescherà una nuova carta nera per iniziare il nuovo turno ed inizierà il nuovo turno. Alla fine della partita vince il giocatore che ha più punti. Il punteggio massimo a cui arrivare è 10.

Goal/Requirements

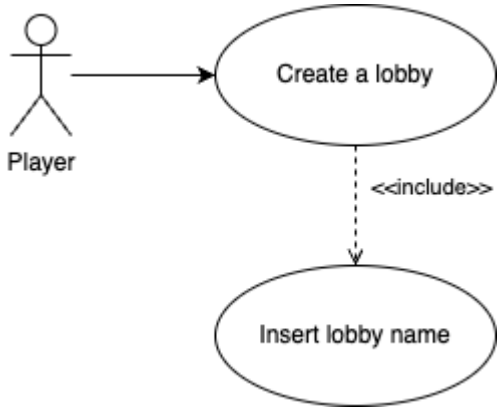
L'obiettivo del progetto è quello di sviluppare una applicazione web per giocare al gioco di carte Cards Against Humanity. Gli obiettivi personali che si sono posti riguardano l'approfondimento delle architetture dei sistemi distribuiti, in particolare si intende ottenere un applicativo in cui distinti client interagiscono tra loro attraverso la comunicazione con un server. In questo caso il server svolge la funzione di gestore di messaggi mentre i client svolgeranno tutta la logica applicativa.

Scenarios

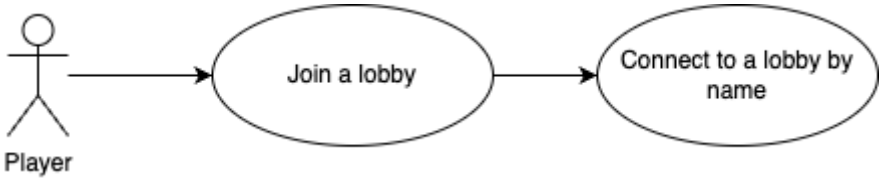
All'avvio dell'applicazione, l'utente potrà scegliere se creare una stanza di gioco, entrare in una esistente oppure leggere le regole di Cards Against Humanity.



Nel primo caso, l'utente configurerà la propria lobby scegliendone il nome. Al termine della configurazione della lobby e alla pressione del pulsante create lobby, il client invierà tramite socket una richiesta di creazione della stanza al server che (nel caso in cui non esista una lobby con lo stesso nome) creerà una nuova stanza con l'utente host all'interno.



In alternativa un utente potrà scegliere di entrare in una lobby esistene premendo il pulsante join lobby, che mostrerà tutte le stanze disponibili per una sessione di gioco in cui sarà possibile connettersi tramite il pulsante connect. In questo caso, verrà inviata dal client verso il server, sempre tramite socket, una richiesta di partecipazione alla stanza esistente.



Self-assessment policy

Il sistema deve garantire la giocabilità della partita:

- si devono seguire le regole di Cards Against Humanity per completare una sessione di gioco;
- se un client non host si disconnette e la stanza ha ancora il numero minimo di giocatori, si deve garantire il proseguo del gioco;
- se l'host si disconnette la partita deve terminare e i vari partecipanti si devono disconnettere dalla stanza in automatico;
- se troppi host si disconnettono da una stanza e si scende sotto il numero minimo di giocatori, la partita deve terminare e la stanza deve essere eliminata;

Requirements Analysis

User

- Creazione di una lobby
- Avvio della partita, se si ha creato la lobby;
- Entrare/Uscire da una lobby esistente;
- Selezionare una soluzione se si è czar;
- Inviare soluzioni all'attuale czar;
- Visualizzare il punteggio attuale;
- Abbandonare una partita in corso.

Functional

- Premendo il pulsante *Create Game* l'applicazione crea una stanza se è stato inserito un nome per la stanza e nessuna stanza attualmente ha quel nome;
- Premendo il pulsante *Join Game* vengono visualizzate tutte le stanze disponibili;
- Premendo sul pulsante *Rules* vengono visualizzate le regole del gioco;
- Premendo sul pulsante *Connect* si accede alla stanza selezionata;
- Il gioco non può avviarsi se ci sono meno di tre giocatori nella stanza;
- La partita deve terminare se ci sono meno di tre giocatori;
- La lobby deve chiudersi se il creatore esce;
- Il primo czar viene scelto casualmente, il successivo deve essere il vincitore del turno concluso;
- Durante il gioco se il czar abbandona, bisogna assegnarne uno nuovo;
- Un utente può inviare una sola carta risposta per turno;
- La partita si deve concludere quando un giocatore guadagna 10 punti.

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.

Technologies

L'applicazione è stata sviluppata utilizzando lo stack MERN, che comprende MongoDB come database, Express.js come framework di backend, React come libreria di frontend e Node.js come ambiente di runtime. In particolare, il frontend è stato creato utilizzando il framework Vite, che si basa su tecniche innovative per garantire prestazioni elevate e un'esperienza di sviluppo fluida.

Per quanto riguarda la scrittura del codice, è stata utilizzata la versione TypeScript di JavaScript, che aggiunge tipi, classi, interfacce e moduli opzionali per migliorare la qualità del codice e semplificare la manutenzione.

La connessione tra client e server è stata implementata utilizzando il protocollo WebSocket, modellato attraverso la libreria npm "Socket.io". Questa libreria semplifica notevolmente l'implementazione di una logica di scambio di messaggi in tempo reale tra il client e il server, consentendo anche la diffusione di messaggi a diversi client associati.

La gestione delle librerie utilizzate è stata effettuata tramite il manager npm e per la parte dei test è stata utilizzata la libreria Vitest, che ha permesso di analizzare la coverage del progetto e implementare tutti i vari test, con la possibilità di eseguirli su browser tramite un'interfaccia grafica.

Il framework Vite si basa su diversi principi fondamentali per garantire prestazioni elevate e un'esperienza di sviluppo fluida. In particolare, utilizza i moduli ES nativi e le moderne API del browser per compilare il codice "on the fly" e garantire tempi di build rapidi e aggiornamenti istantanei nel browser. Inoltre, il server di sviluppo integrato in Vite è ottimizzato per il reloading rapido e la sostituzione dei moduli senza fermare l'esecuzione, consentendo agli sviluppatori di vedere in tempo reale le modifiche apportate al codice senza dover aggiornare l'intera pagina.

Vite implementa anche il caricamento "lazy" dei moduli, consentendo di migliorare le prestazioni dell'applicazione, soprattutto per quelle di grandi dimensioni, e di garantire un caricamento iniziale più rapido per gli utenti. Inoltre, utilizza tecniche come il "tree-shaking" e il "code splitting" per rimuovere il codice inutilizzato dall'applicazione e dividere il codice in sezioni più piccole, migliorando ulteriormente le prestazioni.

Infine, la libreria Vitest ha permesso di implementare tutti i vari test necessari per garantire la qualità del codice e analizzare la coverage del progetto, con la possibilità di eseguire i test su browser tramite un'interfaccia grafica.

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 pattern di progettazione molto utile per la condivisione di dati globali tra diversi componenti in un'applicazione React. In pratica, viene creato un componente Provider che detiene i dati e li condivide con gli altri componenti attraverso un Consumer o degli Hook personalizzati. Questo permette di semplificare la logica di gestione dei dati e garantire una maggiore efficienza nell'interazione tra i componenti.

  Il provider pattern è flessibile e può essere utilizzato in diversi contesti, come ad esempio per lo scambio di dati tra componenti di livello diverso o per la gestione di dati comuni a più pagine dell'applicazione. In questo modo, è possibile evitare di passare manualmente i dati da un componente all'altro attraverso le props, semplificando notevolmente il codice e rendendo l'applicazione più facile da leggere e mantenere.
  Un esempio di utilizzo del provider pattern potrebbe essere quello di gestire lo stato globale dell'applicazione, ad esempio la lingua selezionata dall'utente o il tema dell'applicazione. In questo caso, il Provider detiene lo stato globale dell'applicazione e lo condivide con i componenti figli attraverso un Consumer o un Hook personalizzato. In questo modo, i componenti figli possono accedere allo stato globale senza doverlo passare manualmente come props attraverso ogni livello di gerarchia dei componenti.
  Utilizzo del pattern nell'applicazione:

```
const AppRoute = (props: any) =>
    <Provider store={store}>
        <ConfigProvider
            theme={{
                token: {
                    colorPrimary: '#000',
                },
            }}
        >
            <SocketContextComponent>
                <Routes>
                    <Route path="/" element={<Home />} />
                    <Route path="/lobby" element={<Lobby />} />
                    <Route path="/rules" element={<Rules />} />
                    <Route path="/create" element={<CreateLobby />} />
                    <Route path="/waiting" element={<WaitingLobby {...props} />} />
                    <Route path="/game" element={<Game {...props} />} />
                    <Route path="*" element={<ErrorRoute />} />
                </Routes>
            </SocketContextComponent>
        </ConfigProvider>
    </Provider>

export default AppRoute
```

- **Hooks**: gli hooks sono stati una rivoluzione per la progettazione dei componenti React, poiché hanno introdotto un modo semplice e diretto per i componenti di accedere a funzionalità come props, stato e contesto. Questo ha permesso di separare la logica di gestione dello stato e della logica di rendering dei componenti, semplificando il codice e rendendolo più leggibile.
  Gli Hooks sono funzioni speciali che consentono di utilizzare funzionalità di React in componenti funzionali, che in precedenza erano disponibili solo nei componenti di classe. Ciò ha permesso ai componenti funzionali di gestire lo stato interno, utilizzare il ciclo di vita del componente e accedere al contesto senza doversi convertire in classi.
  Inoltre, l'utilizzo degli hooks ha permesso di semplificare la gestione delle interazioni tra i componenti, poiché condividono la stessa logica di stato e di gestione degli eventi. Ciò ha reso più facile la creazione di componenti altamente modulari e riutilizzabili.
  Infine, gli Hooks hanno fornito una maggiore flessibilità nella gestione del rendering, poiché è possibile utilizzare gli Hooks per controllare quando e come un componente viene renderizzato. Questo ha permesso di migliorare le prestazioni dell'applicazione, evitando il rendering di componenti inutili o evitando di eseguire il rendering di componenti troppo spesso.

  Esempio di Hooks:

```
const SpinningCard = () =>
  <Container>
    <Card>
      <Front>
        <Title style={{ color: "#000000" }}>Cards Against Humanity</Title>
        <Paragraph>Trying to _____</Paragraph>
      </Front>
      <Back>
        <Title style={{ color: "#ffffff" }}>Loading...</Title>
        <Paragraph>_____ the server</Paragraph>
      </Back>
    </Card>
  </Container>

export default SpinningCard;
```

- **Compound Pattern**: il compound pattern è un pattern avanzato che consente a più componenti di condividere stato e logica in modo semplice ed efficiente. Questo pattern si basa sull'utilizzo di un componente principale, noto come "contenitore" o "wrapper", che gestisce lo stato e la logica comune a tutti i componenti figli.
  In pratica, il componente principale detiene lo stato e la logica comune a tutti i componenti figli e passa queste informazioni ai componenti figli come props. In questo modo, i componenti figli possono accedere allo stato e alla logica comune senza doverla gestire autonomamente.
  Il compound pattern è molto utile quando si hanno componenti che condividono una grande quantità di stato e logica comune. In questo caso, il compound pattern semplifica notevolmente la gestione dei dati e garantisce una maggiore efficienza nella gestione delle interazioni tra i componenti.
  Ad esempio, si potrebbe utilizzare il compound pattern per creare un componente che gestisca una lista di elementi e un componente che gestisca la singola visualizzazione di un elemento. In questo caso, il componente principale detiene lo stato della lista di elementi e la logica di gestione degli elementi, mentre il componente figlio gestisce solo la visualizzazione di un singolo elemento. Il componente principale passa le informazioni relative alla lista di elementi al componente figlio come props, semplificando notevolmente la gestione dei dati e garantendo una maggiore efficienza nella gestione delle interazioni tra i componenti.
  Inoltre, il compound pattern è molto flessibile e può essere utilizzato in diversi contesti, come ad esempio per la gestione di modali, tooltip, form e widget complessi. In ogni caso, il compound pattern semplifica la gestione dei dati e della logica comune, garantendo una maggiore modularità e riutilizzabilità dei componenti.
  Esempio di Compound Pattern:

```tsx
const WhiteCard = ({ cardStyle = {}, frontStyle = {}, title = "Cards Against Humanity", children, ...props }: WhiteProps) =>
    <Card style={cardStyle}>
        <Front {...props}>
            <Title style={{ color: "#000000", ...frontStyle }}>{title}</Title>
            {children}
        </Front>
    </Card>

export default WhiteCard;
```

- **Render props pattern**: il pattern render props è un pattern di React che consente a un componente di ricevere una funzione come prop, che a sua volta ritorna un elemento React. In pratica, invece di implementare la propria logica di rendering, il componente utilizza la funzione passata come prop per renderizzare il componente figlio.
  Ciò consente di controllare in modo più preciso il rendering di un componente figlio, poiché la funzione passata come prop può contenere la logica di rendering e avere accesso alle props e allo stato del componente genitore.
  Ad esempio, si potrebbe utilizzare il pattern render props per creare un componente che gestisce la visualizzazione di un tooltip. In questo caso, il componente genitore passerebbe una funzione come prop al componente figlio che gestisce la visualizzazione del tooltip. La funzione passata come prop sarebbe responsabile della logica di rendering del tooltip e avrebbe accesso alle props e allo stato del componente genitore.
  Esempio di Render Props (a sinistra il passaggio dei children e a destra il recupero del valore tramite props):

```tsx
const BlackCard = ({ cardStyle = {}, title = "Cards Against Humanity? More like _____.", children }: BlackProps) =>
  <Card cardStyle={cardStyle}>
    <Front>
      <Title style={{ color: "#fff" }}>{title}</Title>
      {children}
    </Front>
  </Card>

export default BlackCard;
```

```tsx
const Card = styled.div<any>`
height: ${cardHeight};
width: ${cardWidth};
position: relative;
font-family: "Poppins", sans-serif;
border-radius: 0.6em;
border: 1px solid #fff;
${props => props.cardStyle}
```

- **State reducer pattern**: il pattern state reducers di React consente di gestire in modo pulito l'aggiornamento dello stato di un componente in base a diversi criteri. In pratica, il componente genitore passa una funzione come prop al componente figlio che gestisce lo stato. Questa funzione viene utilizzata per aggiornare lo stato del componente figlio, in base a diversi criteri specificati dal componente genitore.
  Il pattern state reducers è utile quando si ha la necessità di gestire lo stato di un componente in modo complesso e personalizzato. La funzione passata come prop può contenere la logica per aggiornare lo stato in modo preciso, in base a criteri specifici, come ad esempio la validazione dei dati inseriti dall'utente o la gestione di eventi asincroni.
  Inoltre, il pattern State Reducers consente di separare la logica di gestione dello stato dal componente figlio, semplificando il codice e rendendolo più leggibile. Ciò consente di avere un maggiore controllo sulla gestione dello stato e di garantire una maggiore flessibilità nella gestione degli aggiornamenti dello stato.
  Il pattern state reducers è spesso utilizzato in combinazione con il pattern render props. In questo caso, la funzione passata come prop alcomponente figlio utilizza il pattern state reducers per gestire l'aggiornamento dello stato del componente figlio e la funzione Render Props per controllare il rendering del componente figlio.
  Esempio di State Reducer:

```
import { configureStore } from '@reduxjs/toolkit'
import { blackCards, userName, whiteCards } from '../reducers'
import { testBlackCards, testUserName, testWhiteCards } from '../testReducer'

export const store = configureStore({
    reducer: {
        whiteCards: whiteCards.reducer,
        blackCards: blackCards.reducer,
        user: userName.reducer,
    },
})


export const testStore = configureStore({
    reducer: {
        whiteCards: testWhiteCards.reducer,
        blackCards: testBlackCards.reducer,
        user: testUserName.reducer,
    },
})

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

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. Inoltre, l'utilizzo di Hooks e di Render props ha permesso di semplificare la gestione del ciclo di vita dei componenti e di fornire una maggiore flessibilità nella gestione del rendering. Infine, lo State reducer pattern ha permesso di gestire in modo efficace l'aggiornamento dello stato e di garantire una maggiore coerenza e pulizia del codice.

Structure

Il progetto si concentra sulla modellazione di due entità principali:

- Carta
- Giocatore

Il mazzo viene creato a runtime, scaricando i dati dal database e dividendo le carte in bianco e nero.

La struttura del progetto si divide in due moduli principali: il Client e il Server.

Client

Il modulo Client contiene tutta la logica dell'applicazione e permette all'utente di interagire con il gioco.

Le sue componenti sono diverse e includono:

- asset: contiene tutte le immagini e le icone utilizzate dal client;
- components: contiene al suo interno tutti i componenti da renderizzare;
- context: ha la definizione di tutte le funzioni delle socket e della socket stessa che viene utilizzata per la comunicazione in tempo reale tra client e server;
- hooks: contiene tutte le funzioni utilizzate dal client;
- pages: contiene il rendering di tutte le diverse pagine del gioco e la loro logica;
- route: descrive le diverse route per visualizzare una pagina specifica;
- store: permette di salvare variabili localmente;
- types: descrive i tipi personalizzati usati nel progetto;
- test: contiene tutti i file di test utilizzati per verificare la corretta implementazione del codice.

Server

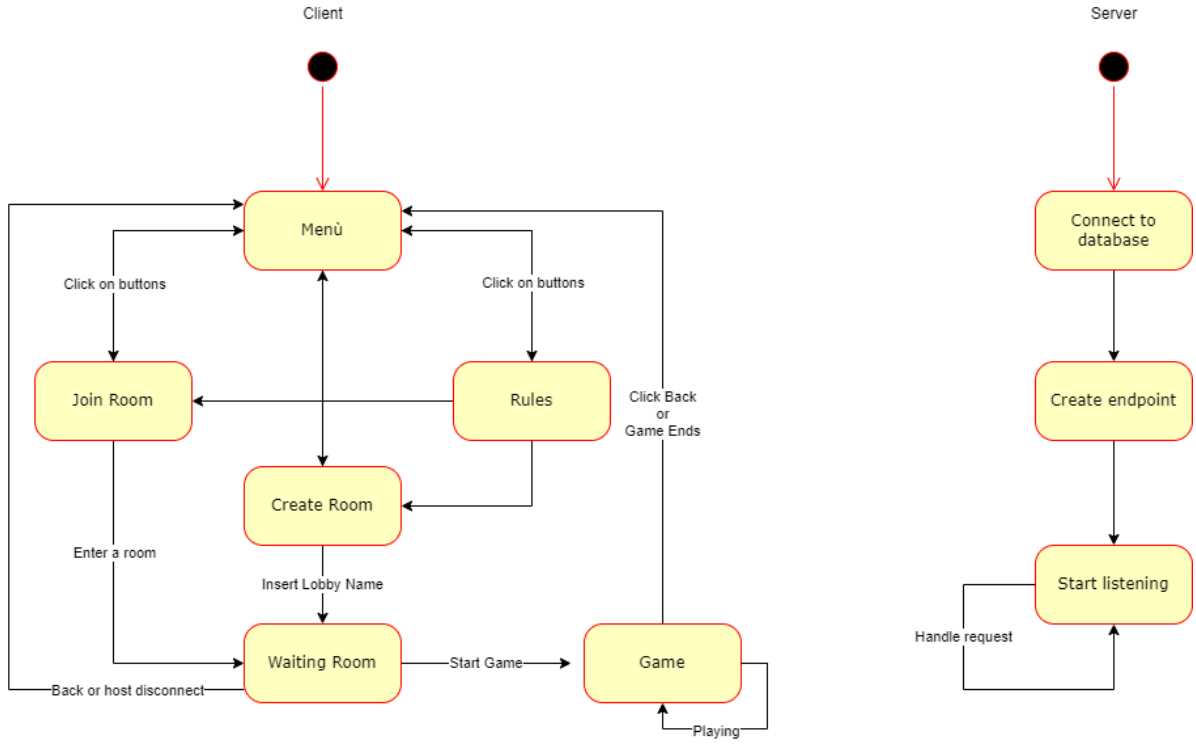
Il modulo Server, invece, implementa la logica relativa alla gestione dei client, delle lobby e dei diversi eventi che si possono verificare durante lo sviluppo di una partita. Le sue componenti principali sono:

- db: si occupa del modello e del collegamento al database;
- utils: gestisce tutti i messaggi che il client può mandare verso il server;
- index: si connette al database, crea il server e lo avvia;
- test: contiene tutti i file di test utilizzati per verificare la corretta implementazione del codice.

In sintesi, la struttura del progetto è stata organizzata in modo logico e modulare, con il modulo Client che si occupa della logica dell'applicazione e il modulo Server che gestisce la logica della partita. La suddivisione in componenti dei moduli ha permesso di semplificare lo sviluppo e la manutenzione del codice, garantendo la qualità del progetto e la sua scalabilità. L'inclusione di file di test per entrambi i moduli ha permesso di verificare la corretta implementazione del codice e di individuare eventuali problemi in modo tempestivo.

Behaviour

Di seguito è riportato il diagramma degli stati del sistema.



Le principali richieste che i client inviano al server possono essere:

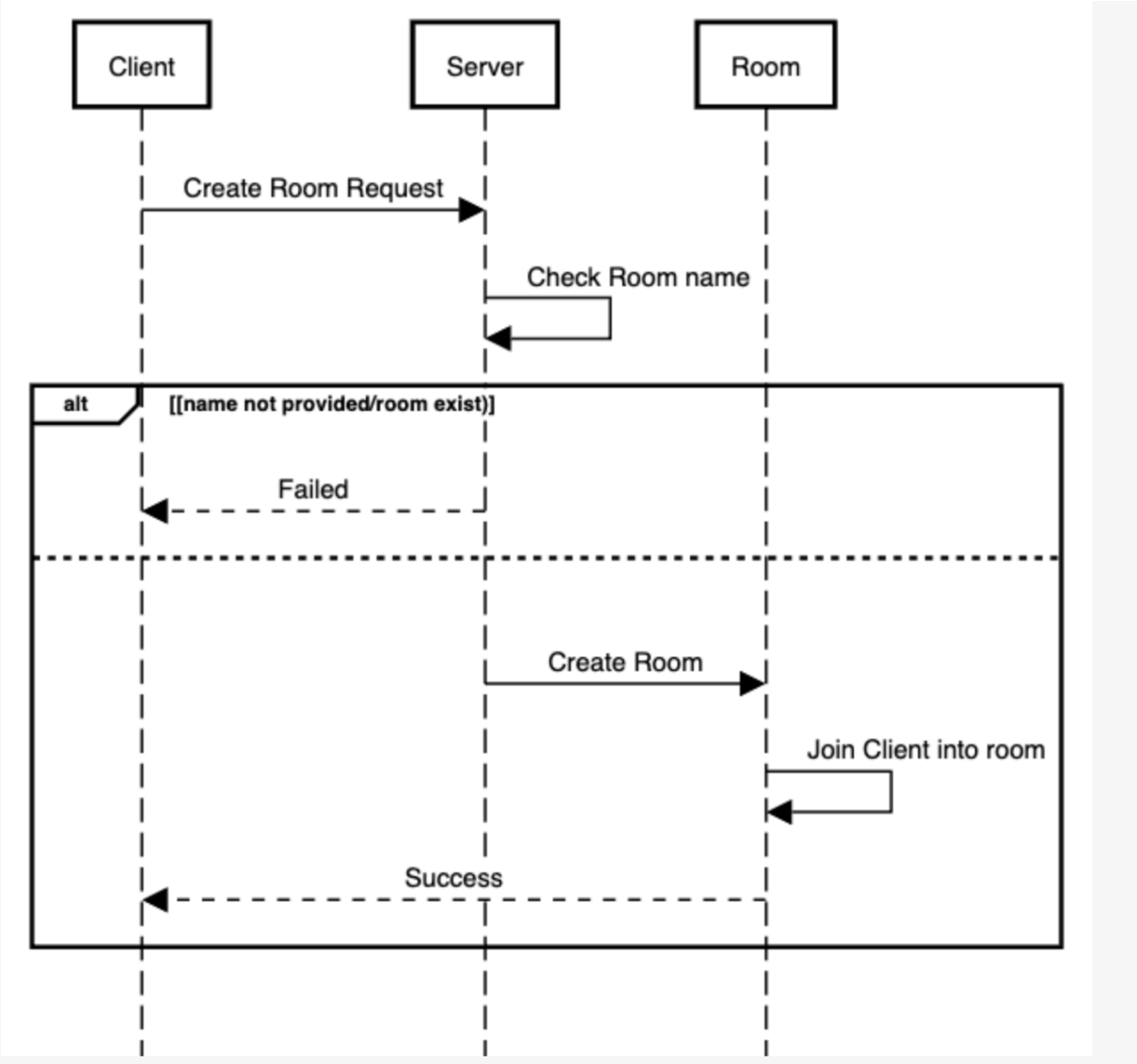
- Ricevere i due mazzi di carte
- Entrare/uscire da una stanza
- Avviare una partita
- Scegliere una carta da giocare (player)
- Scegliere la carta migliore (czar)
- Cambiare lo czar
- Aggiornare il punteggio
- Fine della partita

Interaction

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

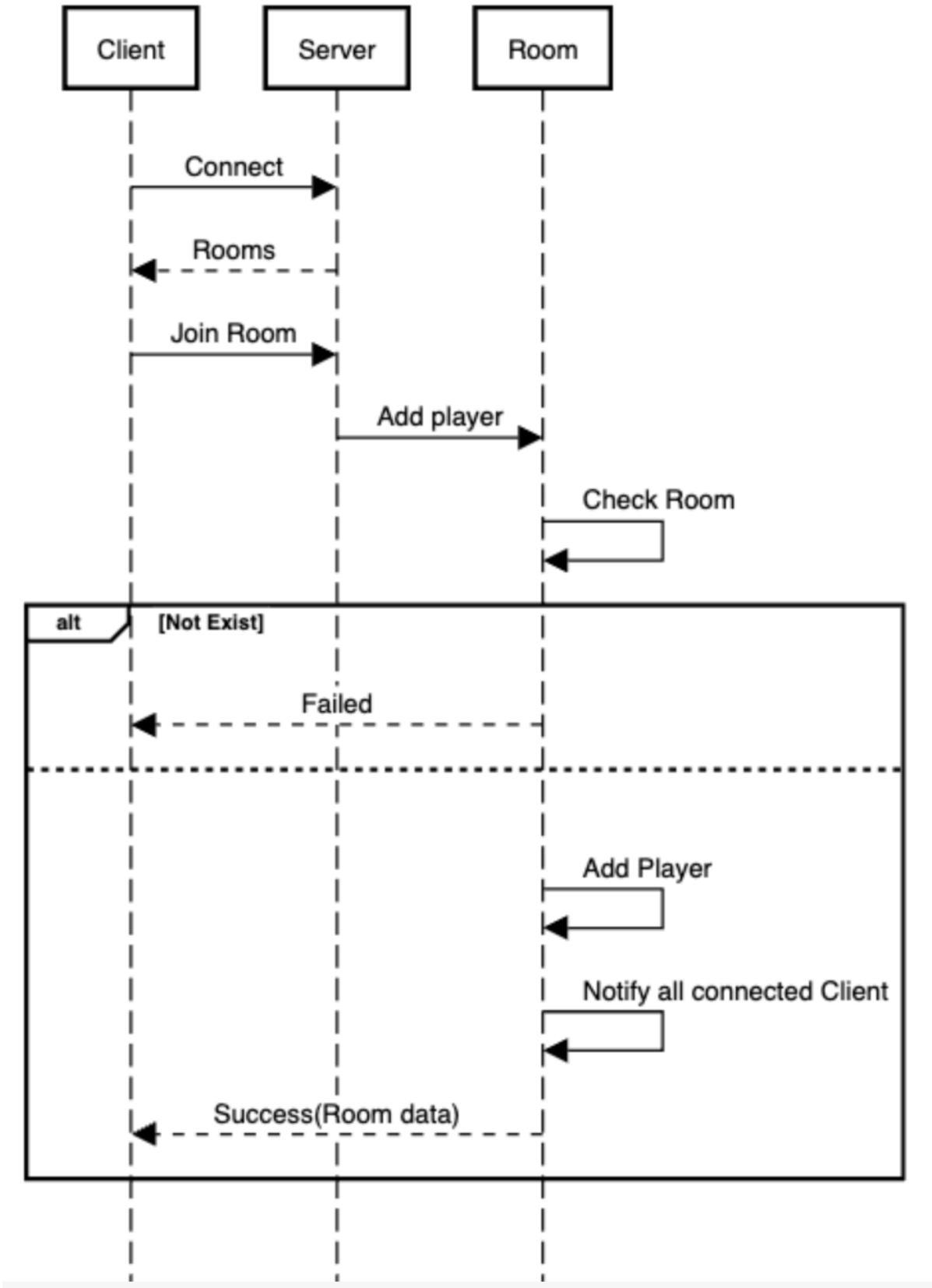
Room Creation

Premendo sul pulsante "Create Lobby" verrà chiesto all'utente di inserire un nome e il server si occuperà di verificare che una room con lo stesso nome non esista, in caso positivo verrà creata.



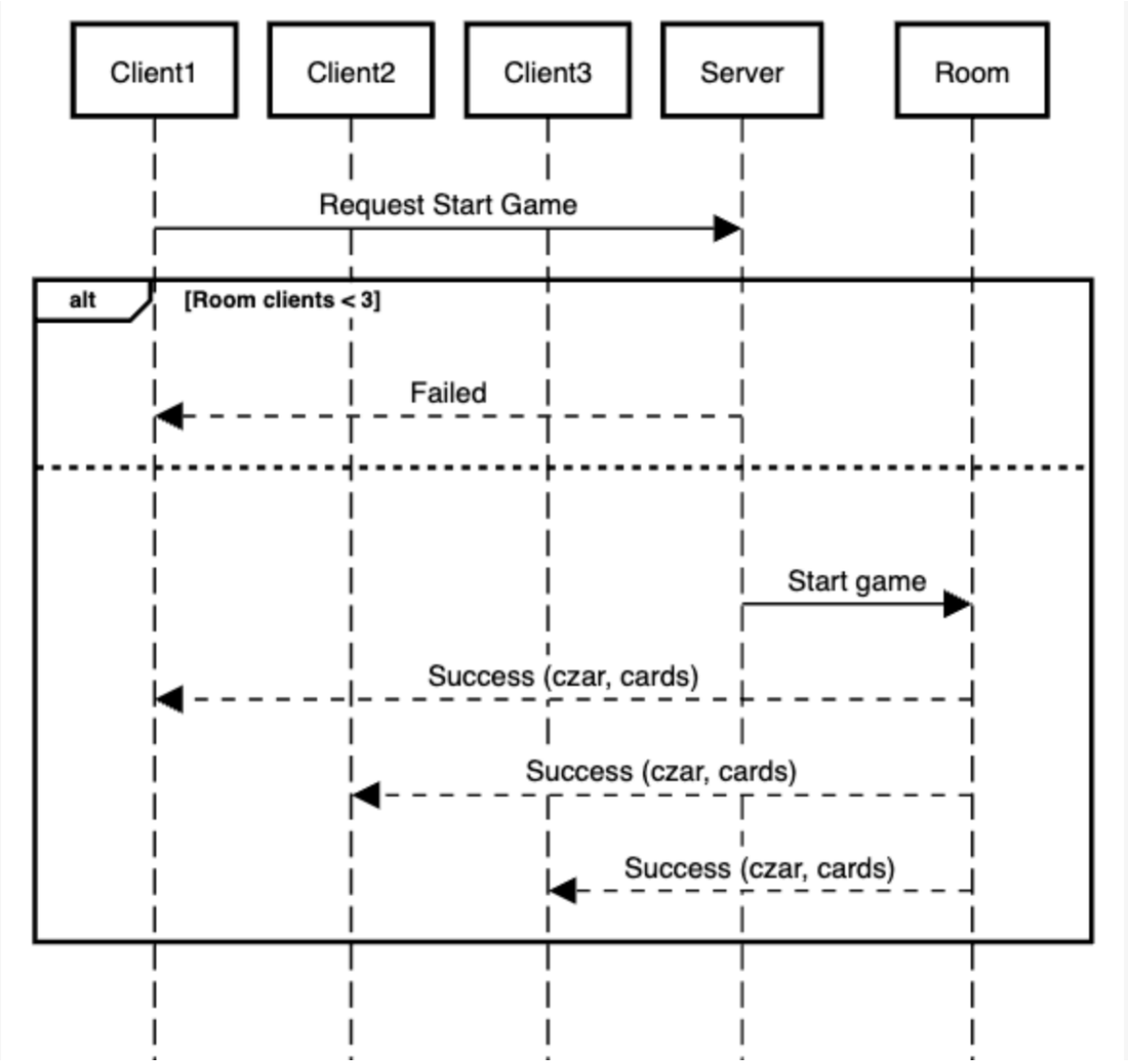
Join Room

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



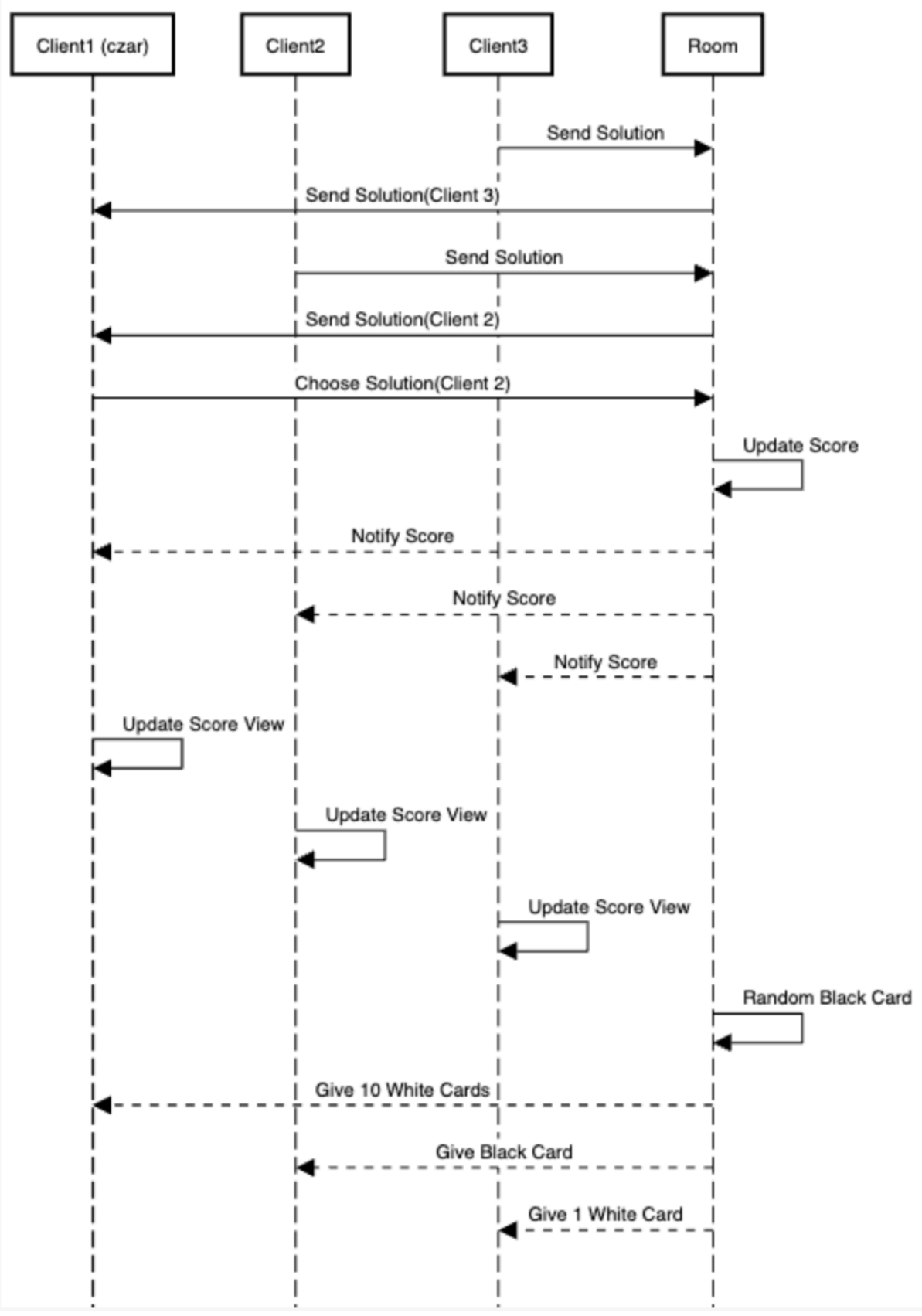
Start the Game

La partita inizia quando l'host della stanza preme il pulsante "Start Game" a patto che la condizione di avere almeno 3 giocatori sia soddisfatta. A quel punto vengono distribuite le carte ai player tranne al giocatore scelto come czar.



Round

Ogni giocatore (non czar) leggerà la carta nera e sceglierà una delle risposte bianche che ha in mano, per poi inviare la carta allo czar. Dopo averla inviata non potrà più cambiare la risposta o scegliere altre carte fino al turno successivo. Lo czar aspetta tutte le carte dei giocatori e infine sceglie la risposta migliore, a questo punto il nuovo czar diventa il proprietario della carta vincente e inizia un nuovo turno.



Validation

Per garantire la qualità del codice e la stabilità del sistema, si è deciso di sviluppare il sistema in parallelo con i test, creando una classe di test corrispondente per ogni modulo al fine di controllare ogni singolo componente.

Dato che l'architettura del progetto era basata su Vite, la scelta di utilizzare la libreria di test Vitest è stata quella più logica. Vitest ha dimostrato di essere una soluzione efficiente e affidabile per implementare i test, grazie alla sua capacità di integrarsi facilmente con il framework Jest utilizzato per la gestione dei test.

In particolare, il vantaggio di Vitest rispetto ad altre librerie di test è il fatto di richiedere una configurazione minima, risparmiando tempo e risorse nel processo di sviluppo dei test. A differenza di altre librerie come Babel o Jest, Vitest si integra perfettamente con la struttura del progetto, consentendo di effettuare test in modo semplice e preciso.

Vitest è una libreria che permette di implementare i test per le applicazioni web, utilizzando il framework Jest. Grazie a Vitest, è possibile eseguire test integrati sia lato client che lato server, valutare la coverage del codice e analizzare i risultati dei test in modo dettagliato.

In particolare, Vitest ha permesso di implementare tutti i vari test necessari per garantire la qualità del codice e analizzare la coverage del progetto. Grazie alla funzionalità di esecuzione dei test su browser tramite un'interfaccia grafica, è stato possibile effettuare test in modo rapido e preciso, visualizzando in tempo reale i risultati e individuando eventuali problemi.

Per i test lato server, Vitest ha permesso di effettuare test sulle API REST implementate con Express.js, verificando che i vari endpoint rispondessero correttamente alle richieste e restituissero i dati attesi. Inoltre, è stato possibile testare la gestione degli errori e la validazione dei dati in input.

Per i test lato client, Vitest ha permesso di testare 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.

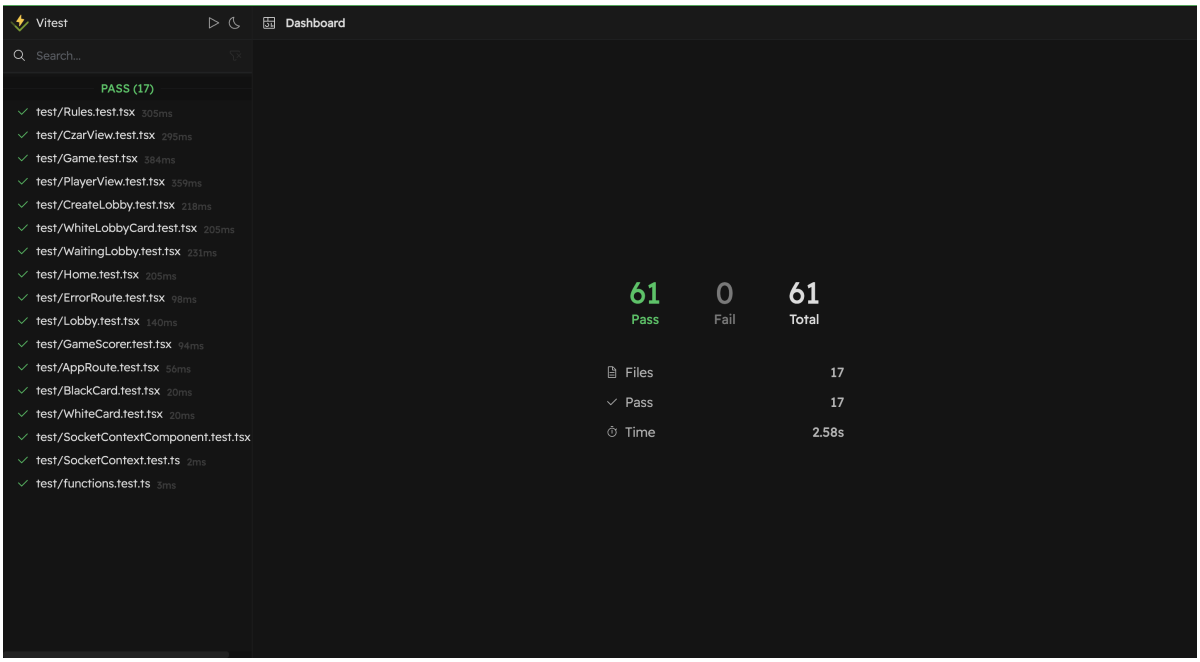
Grazie alla funzionalità di analisi della coverage del codice, Vitest ha permesso di valutare la percentuale di codice coperta dai test e di individuare eventuali aree del progetto che necessitavano di ulteriori test.

In sintesi, Vitest ha permesso di implementare un sistema di test completo e affidabile per l'applicazione, garantendo la qualità del codice e la stabilità del sistema. La possibilità di eseguire i test su browser tramite un'interfaccia grafica ha semplificato notevolmente il processo di testing, consentendo di individuare rapidamente eventuali problemi e di risolverli in modo tempestivo.

Per eseguire i test è necessario seguire i seguenti passi:

1. Posizionarsi nella cartella `server` o `client` del progetto.
2. Eseguire il comando `npm install` per installare tutte le dipendenze necessarie.
3. Eseguire uno dei seguenti comandi:
    - `npm run test` per eseguire i test su console.
    - `npm run test:ui` per visualizzare i test sul proprio web browser.
    - `npm run coverage` per visualizzare la coverage.

```
 Test Files  17 passed (17)
      Tests  61 passed (61)
   Start at  14:44:59
   Duration  9.79s (transform 385ms, setup 1.54s, collect 17.14s, tests 2.79s, environment 5.28s, prepa
re 1.16s)

 % Coverage report from c8
----------------------------|---------|----------|---------|---------|-------------------------------
File                        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------------|---------|----------|---------|---------|-------------------------------
All files                   |   89.14 |    86.82 |   79.09 |   89.14 |
 components/Cards           |     100 |      100 |     100 |     100 |
  BlackCard.tsx             |     100 |      100 |     100 |     100 |
  SpinningCard.tsx          |     100 |      100 |     100 |     100 |
  WhiteCard.tsx             |     100 |      100 |     100 |     100 |
  WhiteLobbyCard.tsx        |     100 |      100 |     100 |     100 |
 context                    |   87.12 |     87.5 |   77.77 |   87.12 |
  SocketContext.ts          |   99.03 |    89.47 |   66.66 |   99.03 | 87
  SocketContextComponent.tsx |   74.48 |       80 |     100 |   74.48 | ...-57,62-65,69-70,84-87,93-96
 hooks                      |   70.19 |     88.6 |   73.07 |   70.19 |
  functions.ts              |   64.48 |    76.92 |   62.85 |   64.48 | ...148,153-169,172-178,208-209
  hooks.ts                  |     100 |      100 |      50 |     100 |
  style.utils.tsx           |     100 |      100 |     100 |     100 |
  useSocket.ts              |     100 |      100 |     100 |     100 |
 pages/Game                 |   84.87 |    78.04 |   68.75 |   84.87 |
  CzarView.tsx              |   90.14 |    58.33 |   66.66 |   90.14 | 26,32-34,36-38
  Game.tsx                  |    69.6 |       80 |   57.14 |    69.6 | 41-71
  GameScorer.tsx            |    87.5 |      100 |      50 |    87.5 | 16-18
  PlayerView.tsx            |     100 |     92.3 |     100 |     100 | 18
 pages/Home                 |     100 |     87.5 |     100 |     100 |
  Home.tsx                  |     100 |     87.5 |     100 |     100 | 44
 pages/Lobby                |    99.2 |    81.48 |   83.33 |    99.2 |
  CreateLobby.tsx           |     100 |      100 |      75 |     100 |
  Lobby.tsx                 |    97.5 |    71.42 |     100 |    97.5 | 34
  WaitingLobby.tsx          |     100 |    82.35 |      75 |     100 | 20-22,39
 pages/Rules                |     100 |      100 |     100 |     100 |
  Rules.tsx                 |     100 |      100 |     100 |     100 |
 route                      |     100 |      100 |     100 |     100 |
  AppRoute.tsx              |     100 |      100 |     100 |     100 |
  ErrorRoute.tsx            |     100 |      100 |     100 |     100 |
 store                      |     100 |      100 |     100 |     100 |
  store.ts                  |     100 |      100 |     100 |     100 |
----------------------------|---------|----------|---------|---------|-------------------------------

 PASS  Waiting for file changes...
       press h to show help, press q to quit

 ✓ test/socket.test.ts (7)
 ✓ test/utils.test.ts (18)
 ✓ test/index.test.ts (3)

 Test Files  3 passed (3)
      Tests  28 passed (28)
   Start at  14:45:04
   Duration  1.03s (transform 392ms, setup 0ms, collect 1.39s, tests 108ms)

 % Coverage report from c8
--------------|---------|----------|---------|---------|-------------------------------------------
File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------|---------|----------|---------|---------|-------------------------------------------
All files     |   82.07 |    74.57 |     100 |   82.07 |
 server       |    96.9 |     90.9 |     100 |    96.9 |
  index.ts    |   95.83 |    88.88 |     100 |   95.83 | 44-46
  socket.ts   |     100 |      100 |     100 |     100 |
 server/utils |   75.56 |    70.83 |     100 |   75.56 |
  utils.ts    |   75.56 |    70.83 |     100 |   75.56 | ...87,191,195-199,203-205,209-211,215-218
--------------|---------|----------|---------|---------|-------------------------------------------

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```


Vitest
Dashboard
Search...
PASS (17)
test/Rules.test.tsx 305ms
test/CzarView.test.tsx 295ms
test/Game.test.tsx 384ms
test/PlayerView.test.tsx 359ms
test/CreateLobby.test.tsx 218ms
test/WhiteLobbyCard.test.tsx 205ms
test/WaitingLobby.test.tsx 231ms
test/Home.test.tsx 205ms
test/ErrorRoute.test.tsx 98ms
test/Lobby.test.tsx 140ms
test/GameScorer.test.tsx 94ms
test/AppRoute.test.tsx 56ms
test/BlackCard.test.tsx 20ms
test/WhiteCard.test.tsx 20ms
test/SocketContextComponent.test.tsx
test/SocketContext.test.ts 2ms
test/functions.test.ts 3ms
61
Pass
0
Fail
61
Total
Files 17
Pass 17
Time 2.58s

Deployment Instructions

Per effettuare il deployment del server e del client, è necessario avere una versione di node installata sul proprio sistema, preferibilmente la `v18.15.0`.

Server

Per effettuare il deployment in locale del server, è necessario seguire i seguenti passi:

1. Posizionarsi nella cartella `server` del progetto.
2. Eseguire il comando `npm install` per installare tutte le dipendenze necessarie.
3. Eseguire il comando `npm run dev` per avviare il server.
4. Il server sarà in ascolto all'indirizzo http://localhost:3000.

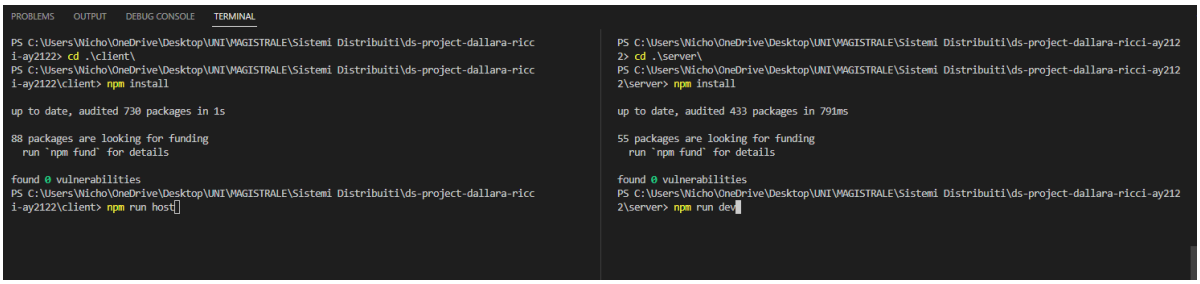
Il comando `npm run dev` utilizza il package `nodemon` per avviare il server in modalità sviluppo. Questo significa che il server si ricaricherà automaticamente ad ogni modifica del codice, senza la necessità di riavviarlo manualmente.

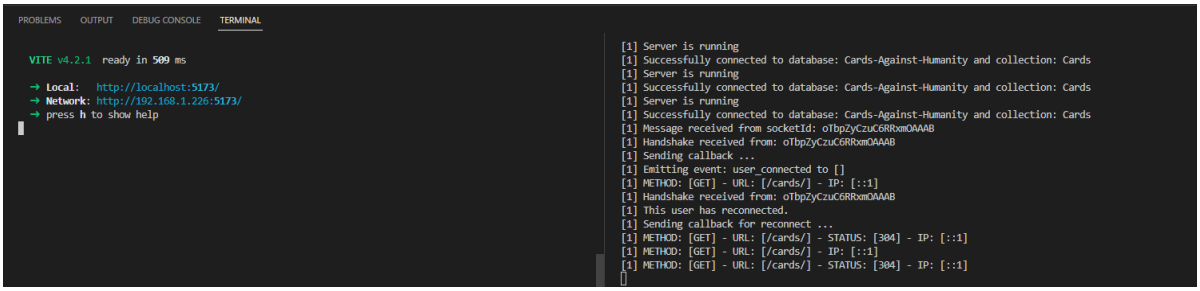
Client

Per effettuare il deployment in locale del client, è necessario seguire i seguenti passi:

1. Posizionarsi nella cartella `client` del progetto.
2. Eseguire il comando `npm install` per installare tutte le dipendenze necessarie.
3. Eseguire il comando `npm run dev` o `npm run host` per avviare il client.
4. Il client sarà in ascolto all'indirizzo http://localhost:5173.

Il comando `npm run dev/npm run host` utilizza il package `vite` per avviare il client in modalità sviluppo. Questo significa che il client si ricaricherà automaticamente ad ogni modifica del codice, senza la necessità di riavviarlo manualmente.





Docker

Per dockerizzare l'applicazione, è necessario aver installato sul proprio dispositivo Docker. Posizionarsi nella cartella `client` del progetto ed eseguire:

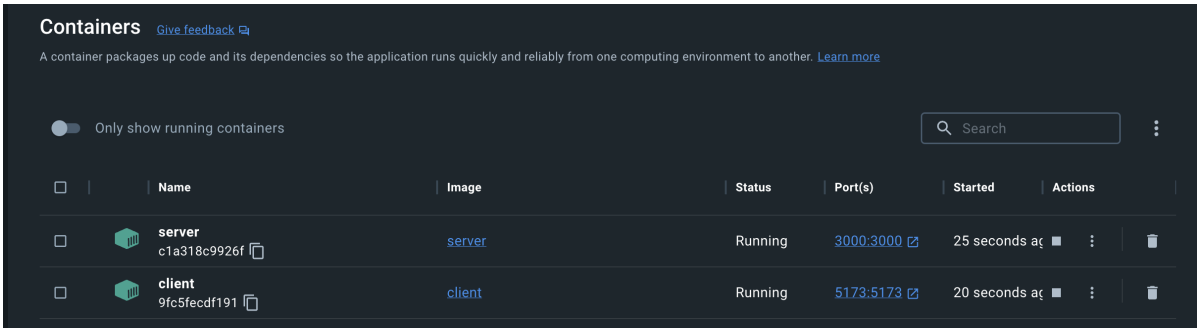
```
docker build -t client .
```

Posizionarsi nella cartella `server` del progetto ed eseguire:

```
docker build -t server .
```

Per eseguire l'applicazione dal suo container eseguire i seguenti comandi:

```
docker run -d --rm -p 3000:3000 --name server server

docker run -d --rm -p 5173:5173 --name client client
```

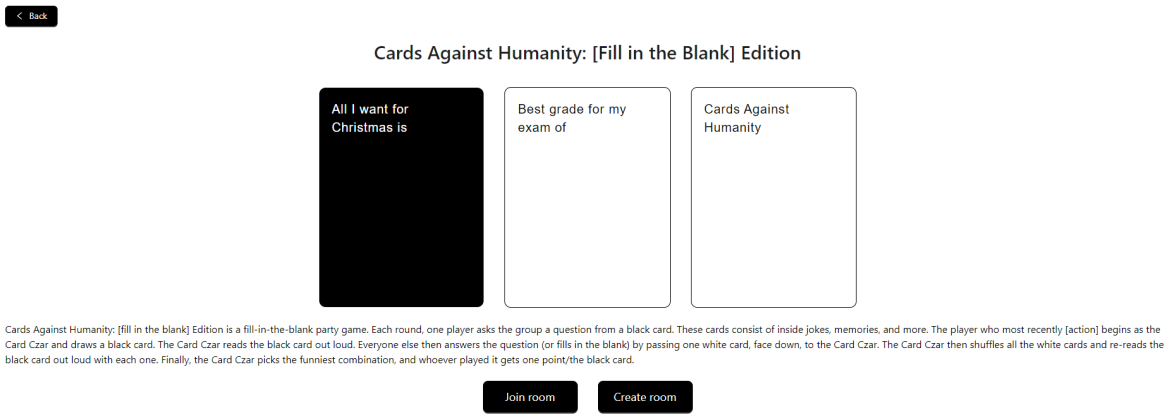


Usage Examples

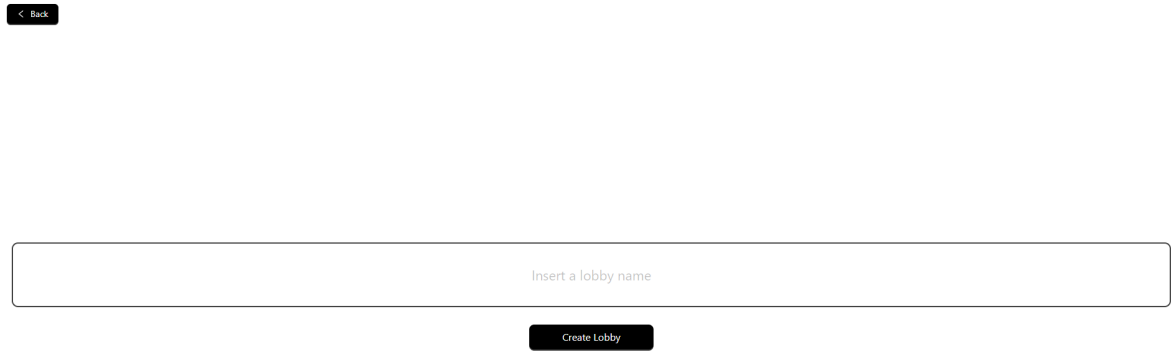
Di seguito vengono descritte tutte le operazioni che un client può effettuare dopo aver seguito le istruzioni di Deployment.

- Aprendo un client qualsiasi si apre la schermata di Home, dalla quale abbiamo tre scelte: unirsi a una partita, creare una stanza o leggere le regole.

- Nella schermata delle regole è possibile leggere il regolamento e vengono mantenuti i due pulsanti precedenti per non dover uscire per giocare.

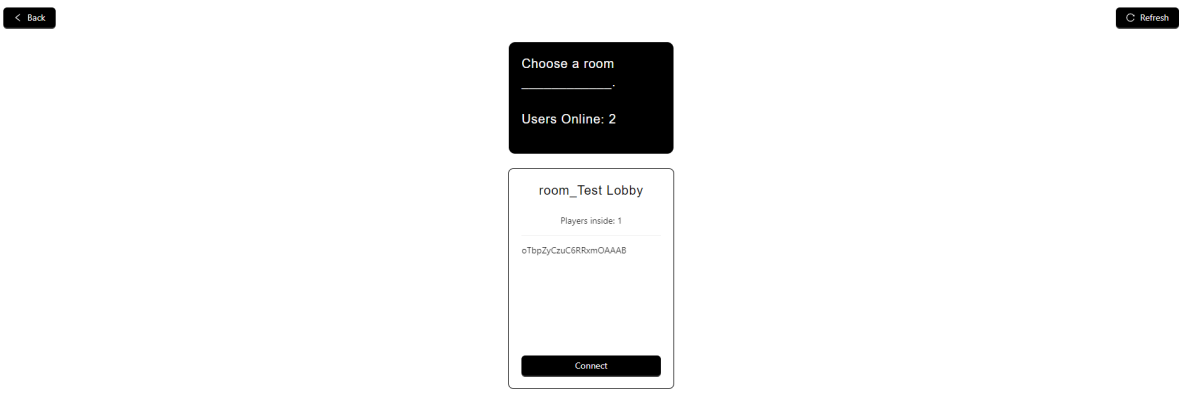


- Per creare una lobby si preme sul pulsante "Create Lobby" e ci viene chiesto di inserire il nome della nostra stanza.



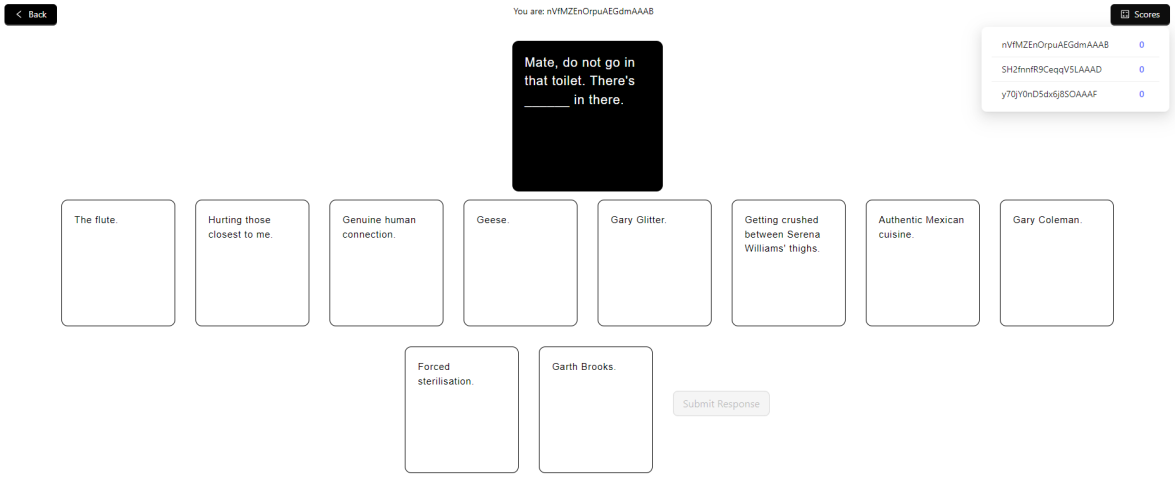
- Una volta inserito la stanza viene creata ed è visibile agli altri giocatori.

- Per potersi unire ad una stanza esistente si preme il pulsante "Join Lobby", che permette di visualizzare tutte le stanze aperte in quel momento.



- Entrando un una stanza qualsiasi si finisce nella sala d'attesa, in cui si vedono tutti i player nella stessa stanza. Solo l'host può iniziare la partita, infatti i giocatori che non lo sono non hanno il pulsante di start.

- Quando almeno tre giocatori sono presenti nella stanza l'host può far iniziare la partita. Il primo turno lo czar viene scelto casualmente mentre successivamente diventa colui che vince il turno precedente. I giocatori (non czar) vedono 10 carte risposta e la carta domanda al centro.



- Quando tutti i giocatori hanno inviato la loro risposta lo czar deve decidere la soluzione migliore.

- Dopo aver scelto il nuovo czar diventa il vincitore e guadagna un punto. I turni si ripetono fino ad arrivare a 10 punti, dopodiché la partita si conclude, la lobby viene eliminata e tutti i giocatori tornano alla Home.

Conclusion

Si ritiene che il progetto sviluppato abbia rispettato tutti i requisiti proposti nella definizione dell'elaborato.

Il team si è concentrato sulla definizione delle specifiche funzionali del gioco, che sono state trasformate in task più piccoli. In particolare, sono state definite le regole del gioco, le carte del gioco e le modalità di scelta dello czar.

Una volta definite le specifiche funzionali, il team si è concentrato sulla realizzazione delle varie funzionalità del gioco. Sono stati creati i mockup per la visualizzazione delle carte e delle lobby, e sono stati implementati i metodi per la gestione delle socket sia lato client che server.

In seguito, il team ha lavorato sulla gestione della logica di gioco, implementando la scelta delle carte e il calcolo del punteggio. Sono stati inoltre sviluppati i metodi per la gestione delle lobby e la connessione dei giocatori alla stanza di gioco.

Durante tutto il processo di sviluppo, il team ha utilizzato GitLab per la gestione del codice e delle issues, in modo da tenere traccia dei progressi e delle modifiche apportate al progetto. Il lavoro è stato suddiviso in task specifici e assegnati ai membri del team, in modo da ottimizzare il tempo e le risorse disponibili.

In conclusione, grazie alla suddivisione di task e alla gestione rigorosa del progetto, il team è riuscito a rispettare i requisiti e a sviluppare un gioco funzionale e divertente per gli utenti.

Future works

In futuro, uno degli sviluppi più importanti che potrebbero essere effettuati riguarda la gestione dei nomi dei giocatori. Attualmente, i nomi sono associati all'ID della socket, ma sarebbe utile implementare un sistema di generazione di nomi casuali o consentire agli utenti di scegliere il proprio nome visualizzato durante il gioco.

Un'altra possibile evoluzione riguarda la gestione delle carte del gioco. Attualmente, sono utilizzate solo carte domanda con un singolo spazio vuoto da riempire. Tuttavia, il gioco originale prevede anche carte con 2 o 3 spazi, che richiedono agli utenti di scegliere più carte in un ordine specifico. Sarebbe utile aggiornare il gioco per gestire anche questi casi, in modo da offrire una maggiore varietà di opzioni ai giocatori.

Infine, un'ulteriore evoluzione potrebbe riguardare la personalizzazione delle lobby. Ad esempio, si potrebbe consentire agli utenti di scegliere il numero massimo di giocatori per la stanza, il numero massimo di punteggio raggiungibile per vincere, oppure di decidere se lo czar viene assegnato casualmente o in senso orario/antiorario. Inoltre, potrebbe essere interessante implementare un sistema che permetta ai giocatori di pescare un nuovo mazzo di carte "spendendo" uno dei propri punti. Questi sviluppi potrebbero rendere il gioco ancora più coinvolgente e divertente per gli utenti.

What did we learned

La conclusione del progetto ha permesso di acquisire una maggiore comprensione e padronanza nell'utilizzo delle socket, nonché delle relazioni tra client e server nella programmazione distribuita. Durante lo sviluppo del progetto, siamo stati in grado di utilizzare efficacemente git per il controllo della versione del codice, lavorando con diverse tecniche per migliorare la qualità del codice e cercando di risolvere ogni issue nel minor tempo possibile. Questo ha permesso di consolidare le nostre competenze e di apprendere nuove tecniche per la gestione di progetti complessi.