Nim Multiplayer

Applicazioni e Servizi Web

Bondi Davide - 0000885217 {davide.bondi3@studio.unibo.it}

11 Ottobre 2021

1 Introduzione

Nim Multiplayer è un'applicazione web che permette a 2 o più utenti di giocare a Nim (https://it.wikipedia.org/wiki/Nim). Nim è un gioco matematico che coinvolge 2 giocatori. Nim Multiplayer supporta le partite 1 vs 1, ed aggiunge regole per gestire le partite multi giocatore.

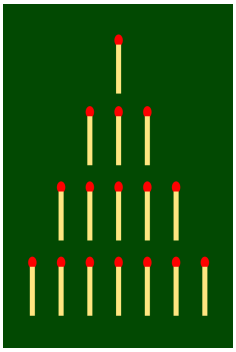

Figura 1: una partita Nim[3].

Per giocare a Nim, i 2 giocatori, a turno, rimuovono gli "*oggetti* o *elementi*" da "*cumuli* o *pile*" distinti.

Per maggiore chiarezza, in questo progetto si è scelto di utilizzare termini più specifici (e ritenuti più appropriati) per esprimere il gergo del gioco: gli "*oggetti* o *elementi*", data la loro forma, assumono il nome di *bastoncini*; mentre i "*cumuli* o *pile*" vengono definiti semplicemente come *righe* o *righe di bastoncini*.

Durante il proprio turno, la mossa consiste nel rimuovere un numero qualsiasi di bastoncini posti su una qualsiasi e unica riga, ma tali bastoncini devono essere adiacenti e tra loro non vi devono essere altri bastoncini già rimossi (non è possibile saltare la mossa nel proprio turno).

A seconda della versione, l'obiettivo del gioco è evitare di rimuovere l'ultimo bastoncino (versione *Marienbad* o *misère*) o rimuovere l'ultimo bastoncino (versione *standard*).

Il numero di bastoncini e di righe sono concordati a piacere tra i giocatori all'inizio della partita.

Il Nim è divenuto piuttosto famoso perché ha una strategia di vittoria semplice (ha classe di complessità L), facilmente utilizzabile come esempio in teoria dei giochi, in particolare questa si basa sul calcolo binario[3].

In breve, Nim Multiplayer modella: il gioco Nim tradizionale con vittoria *standard* e *Marienbad*; aggiunge una variante riguardo la rotazione dei turni (turni chaos); aggiunge regole per la gestione della vittoria *standard* e *Marienbad* in partite multi giocatore.

2 Requisiti

2.1 Funzionali

- Registrazione di un nuovo utente e servizio di login.
- Creazione di una stanza di gioco e relativi inviti.
- Configurare le opzioni di gioco, durante la creazione di una nuova partita.
  - Scegliere il numero di giocatori.
  - Scegliere il numero di righe di bastoncini.
  - Scegliere la rotazione dei turni: rotazione oppure chaos.
  - Scegliere la modalità di vittoria: Standard oppure Marienbad.
- Gestione di una partita Nim tradizionale (1 vs 1): svolgimento del turno; rotazione dei turni (2 tipologie); modalità di vittoria (2 tipologie).
  - Svolgimento del turno: durante il proprio turno, la mossa consiste nel rimuovere un numero qualsiasi di bastoncini posti su una qualsiasi e unica riga, ma tali bastoncini devono essere adiacenti e tra loro non vi devono essere altri bastoncini già rimossi (non è possibile saltare la mossa nel proprio turno).
  - Rotazione dei turni (2 tipologie).
    - La tipica rotazione: i giocatori si alternano in un ordine prestabilito per tutta la durata della partita, generato casualmente ad inizio partita.
    - Turni *chaos*: il gioco procede a turni casuali, ovvero il gioco sceglierà di far passare il turno casualmente ad un giocatore fra quelli che sono "indietro di un turno".
  - Modalità di vittoria (2 tipologie).
    - Standard: il giocatore che rimuove l'ultimo bastoncino vince.
    - Marienbad: il giocatore che rimuove l'ultimo bastoncino perde.
- Gestione di una partita multiplayer (fino a 6 giocatori).
  - Svolgimento del turno e rotazione dei turni: sono identici ai concetti descritti nei punti precedenti.
  - Modalità di vittoria (2 tipologie).
    - Standard: identica a 1 vs 1 (il giocatore che rimuove l'ultimo bastoncino vince, e la partita finisce).
    - Marienbad: il giocatore che rimuove l'ultimo bastoncino viene eliminato dalla partita, ed il gioco prosegue nel seguente modo finché non rimarrà un solo giocatore (vincitore). Dopo l'eliminazione di un giocatore, i bastoncini nelle righe vengono tutti ripristinati, ad eccezione dei bastoncini rimossi dai giocatori eliminati.

  - Gestione delle disconnessioni dei giocatori. Le disconnessioni vengono gestite quando si presenta il turno del giocatore disconnesso.
    * Vittoria Standard: quando si verifica la disconnessione di un giocatore, quest'ultimo viene eliminato dalla partita senza alterare i bastoncini in gioco.
    * Vittoria Marienbad: la disconnessione di un giocatore è considerata come una eliminazione, per cui verranno ripristinati i bastoncini opportuni.

2.2 Non Funzionali

- Connessione sicura tramite https (implementato).
- Memorizzazione cifrata delle password nel database (implementato).
- Recupero delle credenziali utente, tramite invio di una mail (non implementato).
- Giocare una partita come guest non registrato (non implementato).
- Osservare una partita (non implementato).
- Aggiungere nel server un task dedicato per identificare periodicamente le partite finite e le partite mai iniziate, per poi cancellarle e ridurre ragionevolmente lo spazio di archiviazione del database (non implementato).

2.3 Requisiti Tecnologici

L'applicazione verrà realizzata con il solution stack MEVN.

Le comunicazioni real time riguardo la creazione e la gestione della stanza di gioco verranno implementate con la libreria Socket.IO.

2.4 Requisiti degli Utenti

Il metodo di design scelto è *user centered design* (UCD), il quale pone il focus sui bisogni degli utenti in modo iterativo, al fine di massimizzare il grado di soddisfazione della *user experience* (UX).

Per questo è stata svolta un'analisi dei *target user*, e sono state simulate alcune *Personas* come modelli "virtuali" di utenti, correlate ad uno *scenario d'uso* specifico. Ecco la lista di *Personas* considerate.

- Simone conosce la versione tradizionale di Nim e conosce il "segreto" celato dietro la "complessità" del gioco.
  Simone gioca partite 1 vs 1, con modalità di vittoria standard, e turni svolti rigorosamente in rotazione.

- Eraldo ha giocato sempre e solo la versione tradizionale di Nim con modalità di vittoria Marienbad. Ripudia la vittoria standard perché secondo lui non è la versione originale, pertanto non deve essere giocata.
  Eraldo gioca partite 1 vs 1, con modalità di vittoria Marienbad, e turni svolti in rotazione.

- Ilaria è completamente estranea a giochi matematici e combinatori, tuttavia vorrebbe comunque essere partecipe e giocare insieme ai suoi amici. Talvolta vorrebbe anche poter vincere senza necessariamente dover capire la classe di complessità del gioco.
  Ilaria gioca partite multiplayer con gli amici, con qualsiasi modalità di vittoria, e turni svolti in modalità chaos.

- Lorenzo è il classico nerd che conosce a fondo i giochi matematici e giochi strategici che includono il calcolo combinatorio, a volte si cimenta in tornei di scacchi piuttosto impegnativi. Per lui il Nim è un gioco estremamente semplice e noioso.
  Lorenzo gioca partite multiplayer con il numero massimo di giocatori consentito, con modalità di vittoria Marienbad (perché le partite sono più lunghe e richiedono un piccolo sforzo mnemonico), e turni svolti rigorosamente in rotazione.

3 Design

3.1 Design delle Interfacce Utente

In questa sezione si conclude l'analisi della *Human Computer Interaction* (HCI), iniziata durante l'identificazione delle Personas precedenti.

Nelle applicazioni web è consono seguire un approccio di design *mobile first* per progettare le interfacce utente, e così è stato fatto. In particolare, il *progressive enhancement* dell'applicazione si è fermato proprio a livello mobile, cioè non sono stati aggiunti ulteriori raffinamenti significativi per i dispositivi desktop, poiché l'applicazione ed il gioco in sè, sono due elementi talmente minimali, in cui tutto quello da disporre nell'interfaccia è esposto in maniera ottimale anche nei dispositivi mobile.

Questo fattore non esclude il fatto che l'applicazione dispone di *responsive design* e *layout liquido*, ovvero, nonostante l'interfaccia dell'applicazione risulta praticamente identica in tutti i dispositivi, in realtà essa è più fruibile e più "confortevole" nei dispositivi desktop, in quanto gli elementi dell'interfaccia sono disposti su un viewport più largo e si adattano ad esso, migliorando la *user experience* e l'*usabilità* dell'applicazione. Un esempio: nei dispositivi mobile per leggere le regole del gioco bisogna scrollare a fondo, mentre nei dispositivi desktop è sufficiente scrollare meno. Un altro esempio: nei dispositivi desktop, i bastoncini sono renderizzati con una qualità di immagine migliore.

Qui di seguito sono rappresentati i *mockup* prodotti durante la fase di design delle interfacce utente. Per contestualizzarli in relazione all'*usabilità* dell'applicazione, sono stati raggruppati in due principali *storyboard*.

I *mockup* sono stati realizzati tramite Pencil, il quale dispone di feature per creare mockup interattivi (inter-page linking[1]), ma non dispone di feature per creare *storyboard*, per cui quelle inserite qui di seguito sono state prodotte con un editor generico per immagini.

La *storyboard* in figura 2 rappresenta l'insieme di task inerenti alla gestione dei dati dell'utente. Mentre la *stroryboard* in figura 3 rappresenta il task che svolge un utente per creare e giocare una partita.

Un approccio piuttosto informale, ma allo stesso tempo molto pratico per valutare l'*usabilità* del design delle interfacce, consiste nell'osservare l'applicazione nell'ottica delle *euristiche di Nielsen*. Svolgendo questa analisi, si ritiene che tutte le euristiche siano pienamente soddisfatte, soprattutto grazie al fatto della semplicità del sistema modellato.

Osservando le due *storyboard* in figura 2 e 3, si intuisce che le rotte dell'applicazione appartengono a due categorie principali: al dominio dei dati utente, oppure al dominio di gioco. Queste due categorie sono molto importanti, perché sono state sfruttate per modularizzare il design e il codice in molteplici punti.

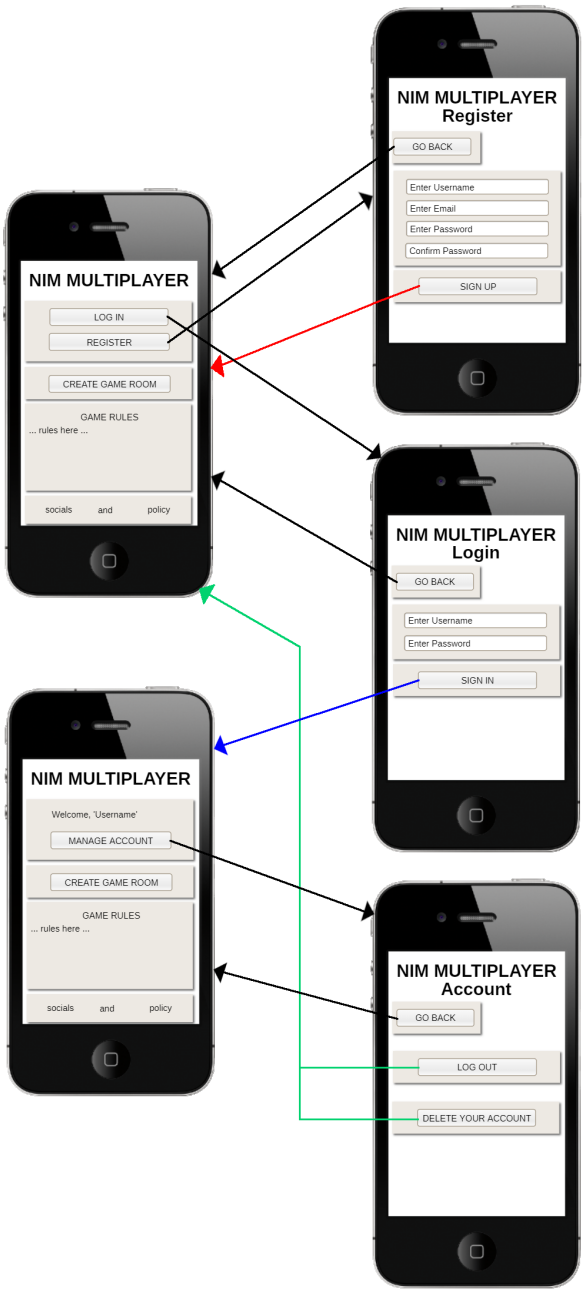

Figura 2: storyboard con i mockup per gestire il dominio dei dati utente.
Linee nere: navigazione che non causa alcuna modifica al sistema.
Linea rossa: registrazione di un nuovo account utente.
Linea blu: login di un utente.
Linee verdi: logout e/o eliminazione dell'account.

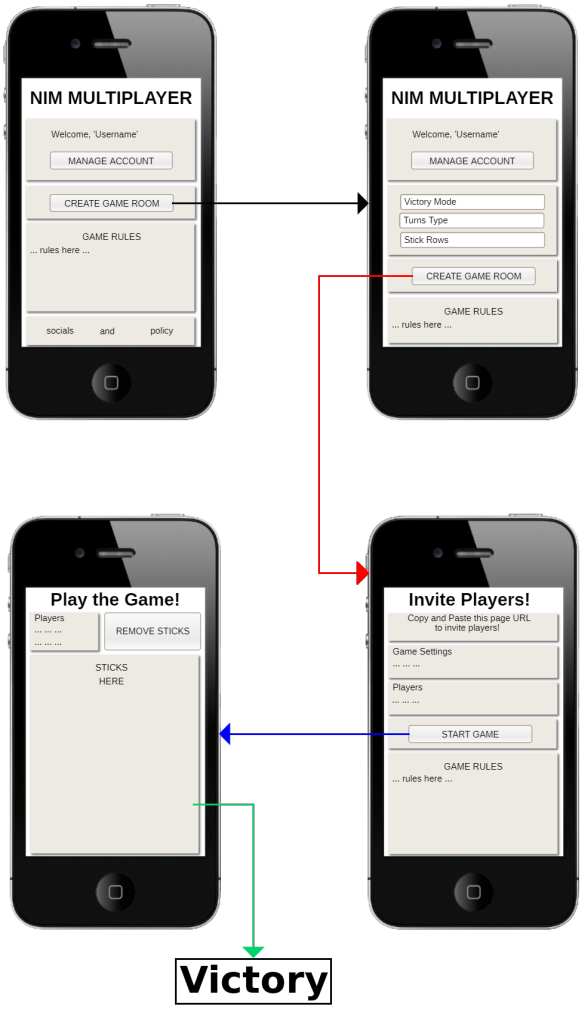

Figura 3: storyboard con i mockup per gestire la creazione e lo svolgimento di una partita (dominio di gioco).
Linea nera: navigazione che non causa alcuna modifica al sistema.
Linea rossa: creazione di una partita.
Linea blu: dopo aver invitato i giocatori desiderati, la partita comincia.
Linea verde: dopo aver giocato la partita, i giocatori sono indirizzati verso la pagina che indica il vincitore.

Nella figura 3 si può osservare che il numero di giocatori non deve essere specificato prima di creare una nuova partita, bensì viene dedotto automaticamente in base a quanti utenti visitano la pagina della stanza di gioco. In questo modo, l'utente non deve scegliere il numero di giocatori, ma deve solo copiare l'URL della stanza di gioco ed inviarlo ai giocatori. Naturalmente, nella fase in cui vengono invitati i giocatori, in pagina deve essere riportato il numero massimo di giocatori che possono essere contenuti in una stanza di gioco.

3.2 Design dell'Architettura di Sistema

Il sistema da realizzare non richiede la gestione di situazioni particolarmente vincolanti, per questo si è scelto di adottare una soluzione di tipo *Single Page Application* (SPA), al fine di ottimizzare la reattività dell'applicazione.

Di conseguenza, il layer della struttura HTML e il layer di presentazione CSS (quindi tutto ciò che riguarda la view di sistema) sono collocati totalmente a client side, mentre invece il layer di *application logic* (o *business logic*) è distribuito tra client e server.

3.2.1 Front-end

Cercando di astrarre quanto più possibile dalle tecnologie utilizzate, la struttura della SPA deve essere organizzata nel seguente modo.

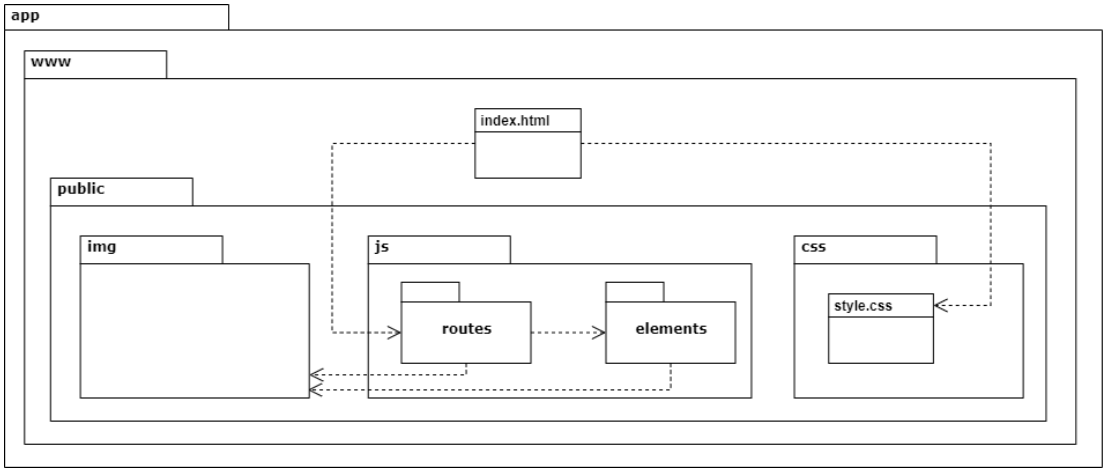

Figura 4: package diagram della SPA (la view situata a client side).
Per applicare maggiore chiarezza, il formalismo del package diagram è stato violato e sono stati inseriti i due file `index.html` e `style.css`, ma concettualmente il diagramma rappresenta comunque l'organizzazione e la dipendenza dei package e dei due file rappresentati.

1. Il file `index.html` è la *Single Page*, la quale contiene al suo interno tutti gli import e la logica di *Application*, quindi contiene tutti i framework o librerie e i file utilizzati a client side.

2. Il package `public` contiene tutti gli elementi importati ed utilizzati in `index.html`. Quindi `public` contiene l'implementazione dell'applicazione client side, e deve essere strutturato in questo modo.

   - Il package `img` contiene le immagini.
   - Il package `js` (Javascript) contiene gli elementi del framework che implementa la SPA, e deve avere a sua volta 2 sub-package.
     (a) Il package `routes` che contiene gli elementi che rappresentano le rotte (pagine) dell'applicazione.
     (b) Il package `elements` che contiene gli elementi che rappresentano porzioni di HTML incapsulate e riutilizzate fra le rotte.
   - Il package `css` contiene solo il file `style.css`, che rappresenta il layer di presentazione da importare in `index.html`.

Considerando ciò che è emerso dalla sezione 3.1, il package `public/js/routes` deve avere due sub-package per racchiudere i due domini presenti nel sistema: `routes/user` contiene le rotte del dominio dei dati utente; `routes/game` contiene le rotte del dominio di gioco.

3.2.2 Back-end

Cercando di astrarre quanto più possibile dalle tecnologie utilizzate, la struttura del codice a server side deve essere organizzata nel seguente modo.

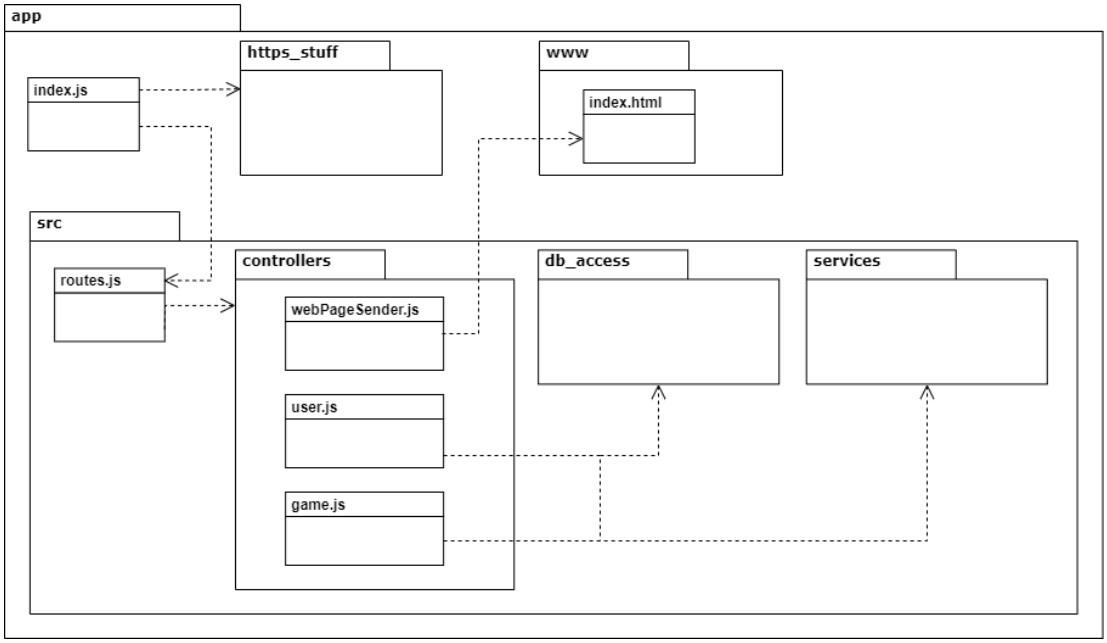

Figura 5: package diagram dell'organizzazione server side.
Per applicare maggiore chiarezza, il formalismo del package diagram è stato violato e sono stati inseriti alcuni file, ma concettualmente il diagramma rappresenta comunque l'organizzazione e la dipendenza dei package e dei file rappresentati (i file rappresentati hanno estensione `js`, perché il progetto Nim Multiplayer è realizzato con la piattaforma Node e quindi è scritto in Javascript, ma a prescindere dal linguaggio usato, l'organizzazione di progetto deve essere questa).

1. Il file `index.js` è l'entry point dell'applicazione (il file eseguibile principale): connette l'applicazione al database; inizializza librerie e servizi dell'applicazione web; avvia il server.

2. Questo progetto adotta il pattern MVC nel web, per cui i controller agiscono tramite le rotte dell'applicazione, quindi il file `routes.js` definisce le funzionalità di sistema ed utilizza i controller per soddisfare le richieste.

3. Considerando quanto espresso fino ad ora, sono state individuate le responsabilità sottostanti, e ciascuna di esse è stata mappata da un corrispettivo controller rappresentato in figura 5.

- Inviare la SPA verso il client che contatta il server (`controllers/webPageSender.js`).
- Gestire il dominio dei dati utente, tramite chiamate Axios (`controllers/user.js`).
- Gestire il dominio del gioco, tramite chiamate Axios e Socket.IO (`controllers/game.js`).

(In figura 5 sono rappresentati tutti i controller di sistema.)

4. Infine, in figura 5 sono rappresentati i package sottostanti.

   - `https_stuff`: contiene il certificato e la chiave https.
   - `db_access`: considerando il pattern MVC, contiene gli elementi che rappresentano i model, cioè i file che accedono al database.
   - `services`: contiene i file che forniscono ulteriori servizi, ad esempio: `services/user/passwordEncryption.js`.

   Come già ribadito più volte, il sistema modella 2 domini distinti, gli utenti e il gioco, per cui `db_access` e `services` contengono a loro volta i 2 sub-package `user` e `game` per migliorare l'organizzazione del codice.

Accesso alla Persistenza: directory `db_access` e directory `db_access/models`

Un'importante scelta di design coinvolge la directory `app/src/db_access` e la sua sub-directory `db_access/models`. La directory `db_access/models` deve contenere i modelli che rappresentano le entità salvate nel database, mentre la directory `db_access` deve contenere gli oggetti che dispongono di metodi per manipolare tali entità ed operare sul database.
In questo modo, il codice risulta notevolmente leggibile e flessibile, ovvero le dipendenze della libreria utilizzata per interfacciarsi al database, sono totalmente incapsulate dentro i file di `db_access`, per cui si può cambiare tecnologia di interfacciamento al database, semplicemente modificando i file di `db_access` e di `db_access/models`.
Un esempio di codice per comprendere questi vantaggi è riportato in 5.2.1.

Questa organizzazione di codice appena descritta, è una sorta di pattern adapter[1], in cui l'adaptee è la libreria scelta per interfacciarsi al database e gli adapter sono gli oggetti di `db_access`.

Un altro esempio strutturalmente identico a questo tipo di pattern adapter, è osservabile nel file `app/src/services/user/passwordEncryption.js`, quest'ultimo adatta (seppur in maniera estremamente minimale) la libreria scelta per cifrare e decifrare la password nel database.

[1] Adapter design pattern, descritto in GoF[2].

Persistenza: Dati degli Utenti

Per memorizzare i dati degli utenti, è sufficiente disporre uno spazio di archiviazione che ammette solamente una tipologia di dato omogenea. I dati degli utenti memorizzati sono sempre gli stessi per ogni utente: l'identificativo (cioè l'username); la password; l'email (che è un secondo identificativo).

Persistenza: Stato delle Partite

Per memorizzare lo stato delle partite è assolutamente consigliato l'impiego di una tecnologia di persistenza NoSQL, perché il numero dei bastoncini e il numero di giocatori sono valori variabili per ogni partita (ma non solo, vi sono anche altri valori illustrati qui di seguito che sono variabili), pertanto NoSQL permette di gestire queste caratteristiche dinamiche con facilità.

Un dettaglio interessante inerente alla memorizzazione delle partite, è che indipendentemente dalla loro configurazione, il loro schema è sempre lo stesso, sia che una partita abbia modalità di vittoria Standard o Marienbad, sia che una partita abbia i turni in rotazione oppure chaos.

I dati memorizzati all'interno di uno schema di una partita sono riportati nel seguente elenco, e per ognuno di essi viene descritto come il loro valore influenza il comportamento di gioco.

- `sticks`: rappresenta i bastoncini, è un dato variabile per ogni partita, ed è rappresentato come una matrice, cioè un array di array in cui quest'ultimi sono le righe che contengono i valori dei bastoncini.
  I valori dei bastoncini sono impostati inizialmente a false, e questo rappresenta lo stato di "bastoncino rimosso = false". Per gestire tutte le casistiche di configurazione di una partita, quando un bastoncino viene rimosso, nel valore del bastoncino viene inserito il nome del giocatore che lo ha rimosso. In questo modo, nelle partite Marienbad multiplayer, quando un giocatore viene eliminato, si possono facilmente ripristinare tutti i bastoncini rimossi dai giocatori presenti in gioco (come richiesto in 2.1).
  (Nel caso di una partita con modalità di vittoria Standard multiplayer, o nel caso di una partita 1 vs 1, sarebbe sufficiente inserire true come valore nei bastoncini rimossi, ma per semplicità si è scelto di adottare la soluzione appena descritta per soddisfare i vincoli di tutte le tipologie di partite.)

- `standardVictory`: rappresenta la modalità di vittoria, è un valore booleano, `true` significa che la partita ha vittoria Standard, `false` significa che la partita ha vittoria Marienbad.

- `turnRotation`: rappresenta la rotazione dei turni, è un valore booleano, `true` significa che i turni sono in rotazione, `false` significa che i turni sono chaos.

- `activePlayer`: è una stringa che contiene l'username del giocatore di turno.

- `players`: è un array con dimensione variabile che contiene gli username degli utenti in partita.
  Il suo significato cambia in base alla modalità di rotazione dei turni.
  - se `turnRotation` è uguale a `true`, allora i turni dei giocatori sono alternati per tutta la partita, seguendo l'ordine di `players`, generato casualmente ad inizio partita.
  - se `turnRotation` è uguale a `false` (turni chaos), allora `players` contiene i giocatori che devono svolgere il turno. Quando un giocatore svolge il suo turno, viene inserito in `playersWithTurnDone`. Quando `players` è vuoto e `playersWithTurnDone` contiene tutti i giocatori, allora tutti i giocatori vengono rimossi da `playersWithTurnDone` per essere inseriti dentro `players`.
- `playersWithTurnDone`: è un array con dimensione variabile che contiene gli username degli utenti che hanno svolto il proprio turno.
  Questo dato viene utilizzato solo nelle partite con turni chaos.
- `eliminatedPlayers`: è un array con dimensione variabile che contiene gli username degli utenti eliminati dalla partita.
- `disconnectedPlayers`: è un array con dimensione variabile che contiene gli username degli utenti disconnessi, e che devono essere eliminati dalla partita nel momento in cui si presenta il loro turno.

3.2.3 Front-end vs Back-end

In tutte le applicazioni web, si deve cercare di spostare quanto più possibile l'application logic a client side, in modo da ridurre il carico di lavoro del server.

Rotte del Dominio dei Dati Utente

Per il motivo appena descritto, nella pagina di registrazione, è la SPA che deve avere la responsabilità di eseguire i controlli riguardo l'inserimento degli input field. In seguito, il server si deve occupare di verificare che: l'username e la mail non sono dati già registrati; l'username non supera una certa lunghezza, perché altrimenti questo causa problemi di renderizzazione dell'username durante il gioco (questo controllo è stato inserito a server side, perché è stato ritenuto più "sicuro" svolgerlo lì).

Allo stesso modo, nella pagina di login, è la SPA che deve avere la responsabilità di eseguire i controlli riguardo l'inserimento degli input field. In seguito, il server si deve occupare di svolgere i controlli sul database.

Rotta della Stanza di Gioco

Sempre per lo stesso motivo, nella pagina di gioco, è la SPA che deve avere la responsabilità di validare la mossa prima di inviarla al server. In seguito, il server si deve occupare di applicare la mossa alla partita, e gestire le disconnessioni dei giocatori in caso si verifichino.

4 Tecnologie

L'applicazione è stata realizzata con il solution stack MEVN: MongoDB; Express.js; Vue.js; Node.js.

All'inizio e durante lo svolgimento del progetto, si è considerato se utilizzare anche un game framework, come ad esempio Phaser, tuttavia questa opzione è stata scartata perché il funzionamento del gioco è estremamente semplice, e per modellarlo è stato sufficiente il solution stack proposto.

Inserire un game framework avrebbe forse reso l'applicazione più "pesante" senza trarre alcun significativo vantaggio, anche se, non conoscendo alcun game framework di Node, mi azzardo a supporre che forse avrebbe potuto rendere il codice del "game room" più modulare e meglio organizzato.

4.1 Back-end

4.1.1 MongoDB

Come già scritto nel design, la flessibilità di NoSQL e del formato documentale di MongoDB, hanno permesso di svolgere una significativa economia concettuale (le partite si modellano con un solo schema), la quale non sarebbe stata applicabile sfruttando il modello SQL relazionale.

Questo vantaggio è osservabile nello schema che rappresenta una partita: `app/src/db_access/models/game.js`.

4.1.2 Node.js

Node è una piattaforma software che impiega un modello I/O non bloccante e ad eventi. Per questo, Node è una piattaforma dotata di elevata scalabilità, altamente indicata per *data-intensive real-time* (DIRT) application (tra cui videogiochi online) che gestiscono un elevato numero di connessioni simultanee.

Node possiede il più grande ecosistema di librerie open source al mondo, e in questo progetto sono state utilizzate le seguenti.

- `mongoose` per interfacciarsi al database.
- `express` per usare il framework Express.js.
- `cors` per impostare le politiche CORS.
- `express-session` per gestire la sessione (cookie) di Express.js.
- `socket.io` per utilizzare la libreria Socket.IO a server side.
- `bcrypt` per cifrare le password nel database.

4.1.3 Express.js

Express è un framework server side per applicazioni web, minimale e flessibile. Facilita l'implementazione di API REST in Node.

4.2 Front-end

4.2.1 Vue.js

Vue permette di sviluppare SPA reattive sfruttando il dual-binding tra modello dati e vista. Vue implementa il paradigma *Model-View-ViewModel* (MVVM), in particolare si focalizza sul ViewModel layer che connette la View con il Model attraverso un two way data binding. Vue rende possibile implementare un'applicazione ragionando in termini di dati, variabili e oggetti, astraendosi rispetto al DOM della pagina, seppur usando comunque le viste HTML.

4.2.2 Bootstrap, CSS e SCSS

In questo progetto è stato utilizzato SCSS, ovvero un'estensione del linguaggio CSS, eseguito da preprocessor (o transpiler) SASS.

Il linguaggio SCSS viene accompagnato dal framework Bootstrap, il quale offre responsive design e layout liquido.

Bootstrap fornisce davvero tantissime funzionalità e ciò lo rende un framework "pesante". In questo progetto si è sperimentato un'alternativa per alleggerire Bootstrap, ovvero invece di importare l'intera libreria a client-side tramite content delivery network (CDN), Bootstrap è stato inserito a server side tramite il suo corrispondente node module in versione SCSS. Dopodiché nel file di presentazione SCSS (`app/www/public/scss/style.scss`) sono stati importati ed utilizzati solo gli elementi di Bootstrap necessari per lo stile del progetto, e infine il file SCSS è stato tradotto nel file CSS (`app/www/public/css/style.css`) da inviare al client.

Innestare Bootstrap nel SCSS in questo modo, comporta alcuni trade-off rispetto ad importarlo tramite CDN.

- Il server di Nim Multiplayer invia un file CSS più grande rispetto al file che dovrebbe inviare tramite la soluzione CDN.

  Il file CSS inviato dal server di Nim Multiplayer contiene: gli elementi di Bootstrap inclusi nel file SCSS e gli elementi specifici di progetto.

  Nella soluzione CDN: gli elementi di Bootstrap sono reperibili tutti via CDN e il file CSS inviato dal server di Nim Multiplayer contiene solamente gli elementi specifici di progetto.

  Il file CSS inviato dal server di Nim Multiplayer, è più piccolo della libreria Bootstrap reperibile tramite CDN.

  Quindi in questo modo Bootstrap è più "leggero" a client side.

- Nel file SCSS è possibile attribuire ai type selector (`body`, `section`, ...) direttamente le classi degli elementi di Bootstrap (il type selector esegue la `@extend` delle classi di Bootstrap), in questo modo si evita, in maniera minima, ma rilevante, di inserire le classi di Bootstrap nella parte strutturale di HTML.

4.3 Back-end e Front-end

4.3.1 Socket.IO

Per le comunicazioni real-time si è scelto di impiegare la libreria Socket.IO.

L'unica feature utilizzata è quella più semplice, e coinvolge solamente il concetto di evento emesso ed ascoltato da un sottoinsieme di client.

Socket.IO fornisce anche il concetto di room (server side), tuttavia non è stato utilizzato questo tipo di astrazione, perché semplicemente non si ha avuto l'esigenza di farlo, ovvero una partita può essere modellata semplicemente come un singolo evento, in cui giunge la mossa svolta dal giocatore attivo oppure la disconnessione di un giocatore.

5 Codice

5.1 Front-end

5.1.1 Organizzazione (Design) del Codice SCSS

Il package `public/scss` rispecchia la struttura del package `public/js` illustrata nella sezione 3.2.1. Il package `public/scss` contiene gli elementi sottostanti.

1. Il file `style.scss` che viene eseguito con preprocessor SASS per generare il suo file corrispettivo: `public/css/style.css`. Quest'ultimo (come già spiegato) è il layer di presentazione da importare in `www/index.html`.
   Il file `style.scss` deve contenere solo i selettori SCSS di elementi e classi utilizzate fra due o più rotte.

2. Il package `routes`: ogni file contenuto in questo package contiene i selettori SCSS di elementi contenuti in una rotta dell'applicazione; per ogni file contenuto in `public/js/routes` può esistere 1 o 0 file di presentazione collocato in questo package.

3. Il package `elements`: ogni file contenuto in questo package contiene i selettori SCSS di una porzione di html incapsulata e riutilizzata fra le rotte dell'applicazione; per ogni file contenuto in `public/js/elements` può esistere 1 o 0 file di presentazione collocato in questo package.

5.1.2 Il "Pattern": `errorMessage`

`errorMessage` è un "pattern" che è stato improvvisato ed inserito in questo progetto per migliorare la leggibilità e la qualità del codice. "Pattern" è posto tra virgolette perché chiaramente non è pattern vero e proprio, e non è neppure una porzione di codice modularizzata, perché purtroppo non è stato trovato alcun modo per farlo. Tuttavia è una forma di codice che è stata ripetuta in più punti a client side, perché si ritiene che sia particolarmente efficace per rendere maggiormente comprensibile il codice.

Molti Vue component (contenuti nei package `js/routes` o in `js/elements`) contengono nel proprio insieme di dati, il dato `errorMessage`. Come suggerisce il nome, questo dato rappresenta il messaggio di errore comunicato all'utente che sta interagendo con l'applicazione. Questo dato viene utilizzato da alcune operazioni come una sorta di stato, per capire se continuare a procedere con l'esecuzione in corso: se `errorMessage` possiede contenuto, allora si è già verificato un errore e l'operazione non deve procedere, oppure l'operazione deve continuare a verificare se ci sono altri errori da comunicare all'utente, concatenandoli al contenuto di `errorMessage` generato precedentemente.

Un'altra forma di codice ritenuta molto comprensibile e che spesso accompagna `errorMessage`, sono le funzioni `gatherClientSideErrorsFrom(vueComponent)` e `gatherServerSideErrorsFrom(response)`, i quali si occupano entrambi di "raccogliere" gli errori, elaborare un messaggio di errore, e restituirlo come valore di ritorno con cui impostare il valore di `errorMessage`.

In alcuni punti si potrebbero avere alcune varianti di queste funzioni, ma l'idea di fondo è piuttosto intuibile e rimane sempre la stessa.

Nel codice sottostante, ecco un esempio di utilizzo di questo "pattern", nel Vue component che rappresenta la pagina di registrazione di un nuovo utente: `app/www/public/js/routes/user/register.js`.

```
...
  <p v-html="errorMessage" class="error-message"></p>
...
data() {
  return {
    ...
    errorMessage: '',
  }
},
...
methods: {
  register: function(event) {
    event.preventDefault()
    this.errorMessage = gatherClientSideErrorsFrom(this)
    if (this.errorMessage === '') {
      const credential = {
        username: this.username,
        email: this.email,
        password: this.password
      }
      axios.post(serverAddress + 'register', credential)
      .then((response) => {
        this.errorMessage = gatherServerSideErrorsFrom(response)
        if (this.errorMessage === '') {
          registrationSuccessful(this)
        }
      })
      .catch((error) => { this.errorMessage = error })
    }

    function gatherClientSideErrorsFrom(vueComponent) { ... }

    function gatherServerSideErrorsFrom(response) { ... }
  }
...
```

5.1.3 Elementi di Design aggiunti durante l'Implementazione

Il package `public/js/routes/utilities` contiene 2 file di utility.

- `two-page-routing.js` gestisce la navigazione rappresentata dalla storyboard in figura 2. Questa utility è stata modularizzata in questo modo, perché l'utente può svolgere quella storyboard anche partendo dalla rotta rappresentata dal file `public/js/routes/create-game-room.js`, di conseguenza sfruttando `two-page-routing.js` non si ha codice duplicato.
- `session-utilities.js` racchiude varie utility inerenti alla sessione di login dell'utente, applicabili ad un'istanza di Vue. Anche questa scelta di design serve ad applicare il principio DRY, oltre ad incapsulare un insieme di responsabilità ben precise.

5.2 Back-end

5.2.1 Accesso alla Persistenza: directory `db_access` e directory `db_access/models`

Nel file `controller/user.js` si utilizza: `db_access/user.js`.

```
1  const usersCollection = require('../db_access/user')
2
3  ...
4
5  exports.logIn = function(req, res) {
6    ...
7    usersCollection.findUserByUsername(username)
8    ...
```

Mentre nel file `db_access/user.js` si utilizzano: la libreria `mongoose` per interfacciarsi al database e il suo modello `db_access/models/user.js`.

```
1  const mongoose = require('mongoose')
2  const UserModel = require('./models/user')
3  const User = UserModel(mongoose)
4
5  ...
6
7  exports.findUserByUsername = function(username) {
8    return User.findById(username).lean()
9  }
10 ...
```

5.2.2 Un Raro Errore di Interazione

Se il giocatore di turno (giocatore attivo) invia la mossa nell'esatto momento in cui un altro giocatore si disconnette, entrambi gli eventi potrebbero modificare nello stesso momento l'istanza di gioco salvata nel database, causando comportamenti imprevedibili. È un problema molto raro, ma può comunque verificarsi. Ad ogni modo, esso è facilmente risolvibile inserendo un monitor per regolare l'accesso al database di quella specifica partita, tra i due handler degli eventi.

Non ho implementato questo monitor, sia per mancanza di tempo, sia perché comunque è un problema molto raro, e anche perché non sapevo esattamente come testarlo, poiché dato che è letteralmente impossibile inviare contemporaneamente una mossa e una disconnessione "manualmente" utilizzando due schede diverse del browser, avrei dovuto utilizzare un sistema di test più complesso e non ne sarebbe valsa la pena in termini di tempo speso per realizzarlo e guadagno effettivo nel testare il caso in questione.

6 Test

6.1 Acceptance Test

Il design riportato in 3.1 può essere considerato una sorta di acceptance test.

6.2 Acceptance e Functional Test degli Errori Utente

In questa sezione vengono descritti i test riguardo l'utilizzo di ogni pagina con l'obiettivo di generare tutti gli errori possibili (functional test degli errori utente), e verificare la correttezza del messaggio di errore generato, valutandone la qualità in termini di comunicazione con l'utente (acceptance test degli errori utente).

6.2.1 Network Error

Un test che può essere eseguito in molteplici punti dell'applicazione, riguarda la visualizzazione dell'assenza di connessione da parte del client o del server durante l'esecuzione di una Axios request, e si può svolgere nel seguente modo.

Prima di fare eseguire all'applicazione una request tramite Axios, si deve arrestare l'esecuzione del server, dopodiché si fa eseguire all'applicazione tale Axios request. In questo modo, dopo qualche secondo Axios genera (a client side) una exception avente come messaggio di errore "Error: Network Error", visibile dall'utente nel HTML paragraph dedicato agli errori (`errorMessage`).

Dopo questa constatazione, se la request inviata non richiede una situazione particolare per avere successo, si può avviare il server e (senza refreshare la pagina) rieseguire la Axios request fallita precedentemente, la quale questa volta non genera più il Network Error, perché ovviamente ora il server può rispondere.

Si riporta ora un'osservazione riguardo le exception generate da Axios e un test "irrilevante" che è stato eseguito giusto per curiosità. Axios genera le exception basandosi sulla response ricevuta (o non ricevuta, nel caso del Network Error), ad esempio, se Axios riceve HTTP status code 500, allora genera una exception avente come messaggio di errore "Error: Request failed with status code 500". Il test "irrilevante" riguarda proprio questa exception. Viene definito "irrilevante" perché in condizioni normali non dovrebbe mai verificarsi, ad ogni modo, il test eseguito consisteva nel modificare il codice a server side, in modo che il server rispondesse con il codice di errore HTTP 500, ed il client visualizzava l'errore in pagina.

In conclusione, si vuole affermare che, il test "Network Error" equivale al verificare la visualizzazione di un qualsiasi altro errore generato da Axios (il codice che gestisce la visualizzazione del Network Error è lo stesso utilizzato per gestire tutte le eccezioni di Axios), pertanto l'utente è sempre più o meno consapevole della situazione in cui si trova.

6.2.2 Registration Page

Test da eseguire per innescare tutti gli errori del client side.

- Lasciare tutti gli input field vuoti e cliccare il bottone **Sign up**, dopodiché leggere il messaggio di errore generato, e inserire i valori negli input field in modo da verificare tutte le combinazioni del messaggio di errore.

Test da eseguire per innescare tutti gli errori del server side.

- Inserire un username già esistente nel database.
- Inserire una email già esistente nel database.
- Inserire un username troppo lungo.
- Eseguire il login e visitare questa pagina inserendo nella barra degli indirizzi il suo path (`/register`), a questo punto si verrà reindirizzati alla home page.

6.2.3 Login Page

Test da eseguire per innescare tutti gli errori del client side.

- Lasciare entrambi o 1 degli input field vuoti e cliccare il bottone **Sign in**.

Test da eseguire per innescare tutti gli errori del server side.

- Inserire username e password non esistenti, oppure username e password sbagliati.
- Eseguire il login e visitare questa pagina inserendo nella barra degli indirizzi il suo path (`/login`), a questo punto si verrà reindirizzati alla home page.

6.2.4 Account Page

Test da eseguire per innescare tutti gli errori del server side.

- Senza eseguire il login, visitare questa pagina inserendo nella barra degli indirizzi il suo path (`/account`), a questo punto si verrà reindirizzati alla home page.
- Eseguire il login, e aprire questa pagina in 2 schede dello stesso browser, dopodiché effettuare il log out (o cancellare definitivamente l'account) in una pagina e poi cancellare definitivamente l'account (o effettuare il log out) nell'altra.

6.2.5 Create Game Room Page

Test da eseguire per innescare tutti gli errori del server side.

- Senza eseguire il login, cliccare sul bottone **CREATE GAME ROOM**.

6.2.6 Game Room Page - Invite other Players!

Test da eseguire per innescare tutti gli errori del client side.

- Cliccare sul bottone **Start Game!** mentre nel game room c'è 1 utente.
- Cliccare sul bottone **Start Game!** mentre nel game room c'è un numero di utenti superiore al numero massimo consentito.

Test da eseguire per innescare tutti gli errori del server side.

- Senza eseguire il login, visitare un game room in attesa di giocatori.
- Senza eseguire il login (oppure anche dopo aver eseguito il login), visitare la pagina di una partita non esistente (può avere un semplice _id inventato, ad esempio "abc123", oppure può avere un _id appartenente a MongoDB, il messaggio di errore generato è lo stesso in entrambi i casi)

6.2.7 Game Room Page - Time to play the game!

Test da eseguire per innescare tutti gli errori del client side.

- Durante la partita, rimuovere 0 bastoncini.
- Durante la partita, selezionare e rimuovere bastoncini che non sono nella stessa riga.
- Durante la partita, selezionare e rimuovere bastoncini nella stessa riga, ma non adiacenti.

Test da eseguire per innescare tutti gli errori del server side.

- Eseguire il login, e visitare una pagina di un gioco già iniziato.

6.2.8 Game End Page

Test da eseguire per innescare tutti gli errori del server side.

- Senza inviare parametri via POST, visitare questa pagina inserendo nella barra degli indirizzi il suo path (`/game-end`), a questo punto si verrà reindirizzati alla home page.

6.3 Functional Test del Gioco

Test da eseguire per verificare il corretto funzionamento del gioco, in particolare per accertarsi che i requisiti siano soddisfatti.
(Lo svolgimento delle partite 1 vs 1, aventi qualsiasi configurazione, è considerato triviale e quindi non viene testato, anche perché il sottoinsieme di questo tipo di partite, viene verificato nei seguenti test riguardo le partite multiplayer.)

- Creare e giocare 2 partite simultaneamente (possono avere gli stessi utenti oppure utenti diversi).
- Controllare lo svolgimento dei turni, sia con vittoria Standard, sia con vittoria Marienbad.
  - I turni in rotazione sono osservabili dall'interfaccia di gioco.
  - I turni chaos, per testarli è consigliato svolgere una partita 1 vs 1 oppure svolgere una partita multiplayer con 3 giocatori, e notare che, prima o poi (in modo totalmente non deterministico), un giocatore invece di svolgere 1 turno alternandosi con gli altri giocatori, svolgerà (al massimo) 2 turni consecutivi.
- Verificare lo svolgimento di una partita multiplayer senza disconnessioni. Si ritiene che una partita con 4 giocatori sia ragionevolmente complessa.
  - Una partita con turni in rotazione e vittoria Standard.
  - Una partita con turni in rotazione e vittoria Marienbad.
  - Una partita con turni chaos e vittoria Standard.
  - Una partita con turni chaos e vittoria Marienbad.

6.3.1 Gestione delle Disconnessioni

Eseguire i seguenti scenari per osservare il comportamento corretto della gestione delle disconnessioni. Si ricorda che le disconnessioni vengono gestite quando si presenta il turno del giocatore disconnesso.
Anche in questo caso, si considera una partita con 4 giocatori, con i turni in rotazione, e gli scenari sottostanti devono essere svolti sia con vittoria Standard, sia con vittoria Marienbad.

- Disconnettere il giocatore di turno successivo rispetto al giocatore attivo (1 disconnessione), ed eseguire la mossa del giocatore attivo.

  Applicare questo scenario in 2 casi.

  1. Dopo la disconnessione, deve rimanere in gioco un solo giocatore, e quindi vince.
  2. Dopo la disconnessione, devono rimanere in gioco 2 o più giocatori, e continuare a giocare la partita fino alla fine.

- Disconnettere i 2 giocatori di turno successivo rispetto al giocatore attivo (2 disconnessioni consecutive), ed eseguire la mossa del giocatore attivo.

  Applicare questo scenario in 2 casi.

  1. Dopo le 2 disconnessioni, deve rimanere in gioco un solo giocatore, e quindi vince.
  2. Dopo le 2 disconnessioni, devono rimanere in gioco 2 o più giocatori, e continuare a giocare la partita fino alla fine.

- Disconnettere il giocatore di turno (giocatore attivo).

  Applicare questo scenario in 2 casi.

  1. Dopo la disconnessione, deve rimanere in gioco un solo giocatore, e quindi vince.
  2. Dopo la disconnessione, devono rimanere in gioco 2 o più giocatori, e continuare a giocare la partita fino alla fine.

- Disconnettere il giocatore di turno successivo rispetto al giocatore attivo, e poi disconnettere il giocatore attivo, causando 2 disconnessioni consecutive.

  Applicare questo scenario in 2 casi.

  1. Dopo le 2 disconnessioni, deve rimanere in gioco un solo giocatore, e quindi vince.
  2. Dopo le 2 disconnessioni, devono rimanere in gioco 2 o più giocatori, e continuare a giocare la partita fino alla fine.

- Disconnettere i 2 giocatori di turno successivo rispetto al giocatore attivo, e poi disconnettere il giocatore attivo, causando 3 disconnessioni consecutive.

  Applicare questo scenario in 2 casi.

  1. Dopo le 3 disconnessioni, deve rimanere in gioco un solo giocatore, e quindi vince.
  2. (Questo è l'unico caso in cui è necessaria una partita con 5 giocatori.) Dopo le 3 disconnessioni, devono rimanere in gioco 2 o più giocatori, e continuare a giocare la partita fino alla fine.

Eseguire questi scenari in una partita (sia con vittoria Standard, sia con vittoria Marienbad) con i turni chaos, è piuttosto difficile perché la rotazione dei turni è casuale (anche se, in una situazione specifica si può prevedere quale sarà il giocatore attivo successivo, cioè si consideri il caso in cui ci sono 4 giocatori in partita e 2 di essi hanno già svolto il turno, allora si conosce il giocatore attivo ed il prossimo giocatore attivo). Ad ogni modo sono state eseguite svariate prove con i turni chaos e sembra funzionare tutto a dovere.

7 Deployment

Per installare ed eseguire l'applicazione si devono svolgere le seguenti istruzioni.

1. Installare `MongoDB`. Creare il database `nim_multiplayer` e le collezioni `users` e `games`. Nella collezione `users` si devono impostare `username` e `mail` come identificativi primari.

2. Installare `Node.js` e `git`.

3. Clonare il repository:

   ```
   git clone https://github.com/deividvd/Nim-Multiplayer/
   ```

4. Tramite la command line, posizionarsi all'interno di `Nim-Multiplayer/app/`, e risolvere le dipendenze di `Node.js`:

   ```
   npm install
   ```

5. Eseguire l'applicazione:

   ```
   node index.js
   ```

6. Utilizzare l'applicazione tramite un browser all'indirizzo (la connessione sarà non sicura perché il certificato https è auto firmato):

   ```
   https://localhost:3000
   ```

8 Conclusioni

Mi ritengo soddisfatto dell'applicazione realizzata, ho implementato un gioco vero e proprio, seppur semplice, inoltre penso che il codice è scritto e organizzato in modo comprensibile ed efficiente per garantirne il riuso e la modifica, nonostante fosse la mia prima esperienza di programmazione full stack con Javascript.

Per migliorare ulteriormente il gioco, può essere aggiunto un limite di tempo per svolgere la mossa. Eventualmente si può aggiungere anche una chat dentro le stanze di gioco, tuttavia questa chat non è particolarmente necessaria perché ovviamente si presume che i giocatori si conoscano e quindi se lo desiderano possono usufruire di chat di terze parti come ad esempio Discord.

Riferimenti bibliografici

[1] Evolus. Pencil website (http://pencil.evolus.vn).

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[3] Wikipedia contributors. Nim — Wikipedia, the free encyclopedia, 2021. [Online; accessed 16-September-2021].