Checkers

Luca Salvigni
luca.salvigni7@studio.unibo.it

Konrad Gomulka
konrad.gomulka@studio.unibo.it

Manuele Pasini
manuele.pasini@studio.unibo.it

June 13, 2022

Checkers Web App rappresenta una versione multiplayer online del gioco da tavolo dama. Le regole di gioco sono quelle della dama internazionale, in cui due giocatori muovono alternativamente i propri pezzi in base al turno. I pezzi si muovono diagonalmente, solo sulle caselle nere non occupate da altri pezzi ed hanno la possibilità di prendere(o "mangiare") quelli avversari, scavalcandoli. I pezzi catturati vengono rimossi dalla damiera ed esclusi dal gioco. Il giocatore a cui vengono catturati tutti i pezzi o che, durante il suo turno e impossibilitato a muovere, ha perso. Se, dopo aver effettuato una presa e nella casella di arrivo si presentano le condizioni per la pedina di una nuova presa e obbligatoria la presa multipla, che va effettuata nello stesso turno di gioco. Il progetto consiste nella realizzazione di una versione distribuita del gioco che permetta agli utenti di connettersi da remoto e di giocare insieme attraverso un meccanismo di stanze virtuali.

1 Goal/Requirement

L'obbiettivo del progetto è quello di realizzare un sistema distribuito basato su un'architettura client-server che permetta agli utenti di connettersi da remoto e giocare tra loro partite di dama.

1.1 Feature principali

Il sistema dovrà essere in grado di fornire le seguenti funzionalità:

- possibilità di poter svolgere una partita: tramite un meccanismo di matchmaking sarà possibile unirsi ad una lobby per iniziare una partita;
- organizzazione delle partite in lobby: il sistema deve permettere agli utenti la possibilità di creare una lobby, inserendo alcuni parametri per il settaggio di quest'ultima, dando la possibilità ad un altro giocatore di entrare per disputare una partita.
- meccanismo ad inviti: il sistema deve permettere ad un utente di invitare all'interno della propria lobby un secondo giocatore specifico.
- gestione profilo utente: ad ogni giocatore viene associato un proprio profilo di gioco che potrà a sua volta essere personalizzato in base alle preferenze del giocatore;
- classifica globale: il sistema deve tenere traccia dei risultati ottenuti da tutti i giocatori e definire una classifica sulla base di questi.

1.2 Scenari

Siccome il client non necessita di alcuna installazione sul dispositivo, i vari giocatori potranno usufruire dell'applicazione direttamente collegandosi ad Internet, rendendo di fatto il sistema cross-platform. Inoltre, va reso noto che, siccome non è parte fondamentale del progetto, l'interfaccia utente non sarà del tutto responsive, non potendo garantire la giocabilità su tutti i possibili tipi di schermo. Dando una definizione più pratica di un tipico scenario d'uso: quando un utente si connette via sito web, questo può autenticarsi tramite una fase di login o nel caso in cui non abbia un account, registarsi per la prima volta all'interno del sistema. Dopodiché potrà decidere di: creare una lobby, unirsi ad una lobby già esistente, invitare un altro giocatore, consultare la classifica globale, modificare dati relativi al proprio profilo. Una volta iniziata una partita, ogni giocatore potrà effettuare una mossa solo quando sarà effettivamente il suo turno e scambiare messaggi con il suo avversario attraverso un'apposita chat di gioco. Nel caso di un'eventuale disconnessione da parte di un giocatore, quest'ultimo avrà un tempo limite entro cui riconnettersi altrimenti verrà data sconfitta a tavolino, altrimenti la partita verrà ripresa dal punto di interruzione.

1.3 Self-assessment policy

La parte core del progetto sarà sviluppata attraverso l'uso di Node.js. Questa scelta permette di realizzare codice flessibile e multi-piattaforma, in cui ogni componente potrà, in base alle necessità, essere imitato facilmente durante le fasi di testing. La qualità del progetto verrà monitorata attraverso delle fasi di testing, verificando il corretto funzionamento del sistema attraverso un'analisi dei vari requisiti e un approfondita fase di unit testing. Oltre alla fase di testing, il corretto funzionamento del sistema nel complesso verrà assicurato mediante un approfondito utilizzo del sistema, verificandone la correttezza in base agli output forniti in Frontend .
Per quanto riguarda l'efficacia dei risultati, verrà testata simulando un incontro con l'esperto di Dominio il quale testerà il sistema fornendo un feedback.

2 Analisi dei requisiti

Il sistema nasce sulla base del requisito principale dell'essere in grado di realizzare il più semplice e funzionale ecosistema di giocatori di dama, in particolare la piattaforma deve offrire un'esperienza che sia comprensibile e fruibile da qualunque tipologia di utente; prendendo in esame il caso del gioco degli scacchi (il quale è più diffuso ed è possibile uno studio più approfondito del contesto), è possibile notare come l'età in cui i giocatori professionisti raggiungono il grado di Gran Maestro sta progressivamente diminuendo[1] stabilizzandosi al momento attorno ai vent'anni ed allo stesso tempo è possibile individuare l'età di picco dei giocatori professionisti in un range di età che va dai trentacinque anni ai quarantacinque anni.[**?**] Costruita una baseline riguardante l'età target del gioco degli scacchi si è provveduto a trasporre queste informazioni e dedurre assunzioni sul gioco della dama.
Per quanto esista una scena professionale del gioco della dama questa è certamente molto inferiore a livello di estensione e diffusione rispetto a quella degli scacchi, così come inferiore è la complessità intrinseca del gioco e la durata media delle partite; infine, tenendo in considerazione che le statistiche prese in esame fanno riferimento a giocatori professionisti che non sono il target d'utenza dell'applicazione, si sono individuate di una serie di caratteristiche che potrebbero essere presenti nell'utente di Checkers:

- **inesperienza nell'utilizzo di tecnologie informatiche**: per la semplicità e la rapidità del gioco è possibile considerare come utente target anche persone la cui età statisticamente coincide con una scarsa conoscenza informatica;
- **breve tempo a disposizione**: la rapidità del gioco è un fattore influente che deve essere consolidato attraverso una rapidità nell'utilizzo dell'applicazione, misurabile nel tempo/numero di procedure necessarie per iniziare una partita dal momento in cui l'utente accede al servizio;
- **internazionalità dei giocatori**: vista la bassa diffusione del gioco è opportuno allargare il bacino d'utenza realizzando un applicazione che abbatta, quantomeno in parte, le barriere linguistiche; sulla base di questa considerazione si è scelto di realizzare la versione internazionale di dama, che segue le seguenti regole:
  - La scacchiera è composta da 20 caselle per lato, delle quali solamente quelle nere, che corrispondono alla metà, sono utilizzabili;
  - il sistema di coordinate utilizzato è diverso da quello degli scacchi, ogni casella è numerata progressivamente all'interno del range [1,50], dove la casella numero uno è quella collocata in alto a sinistra della scacchiera, ipotizzando pedine bianche nella parte bassa della scacchiera;
  - le pedine possono muoversi diagonalmente e la cattura di una pedina avversaria è sempre obbligatoria, così come obbligatoria è la cattura del maggior numero di pedine possibili; al contrario della dama italiana le pedine possono

[1] eta media per diventare Gran Maestro

Figure 1: Sistema di coordinate

catturare in ogni direzione, persino muovendosi indietro, movimento possibile solamente in questa casistica.

– una pedina che raggiunge il bordo opposto della scacchiera diventa "re" perdendo il vincolo di movimento in avanti; un re può muoversi in qualunque direzione; in particolare se il movimento di un re comporta una cattura, tale movimento può avere lunghezza indefinita di x caselle.

Sulla base di questa breve profilazione sono stati identificati tre requisiti qualitativi dell'applicazione:

1. **intuitività**: l'applicazione deve colmare il gap di conoscenze tecnologiche presente in una fetta degli utenti dell'applicazione, le procedure devono essere rapidamente comprensibili ed attuabili anche da chi non ha particolari capacità in ambito informatico;

2. **semplicità**: corollario dell'intuitività, il sistema deve essere dotato di poche e semplici procedure, che permettano un utilizzo intuitivo del sistema; proprietà che fa riferimento anche all'aspetto grafico del sistema: questo deve agevolare l'esperienza di gioco e non appesantirla, deve permettere agli utenti di concentrarsi sulle partite senza distrazioni o grafiche esose.

3. **velocità**: ogni procedura attuabile all'interno dell'applicazione deve essere svolta con un numero di click non superiore a cinque.

2.1 Requisiti di business

Questa sezione è dedicata all'analisi e definizione dei requisiti di business che caratterizzeranno il sistema. L'approccio utilizzato per la definizione del modello è basato sulla

filosofia Domain Driven Design (DDD). Inizialmente è stata svolta una fase di **Knowledge Crunching** per esplorare il dominio ed avere una struttura visiva che possa poi essere facilmente trasferita all'implementazione effettiva.

2.1.1 Intervista agli stakeholder

Le interviste agli stakeholder rappresentano una preziosa fonte di ispirazione all'interno di un'organizzazione, perchè aiutano a scoprire aree di scostamento tra la strategia aziendale e i comportamenti o i processi decisionali attivati quotidianamente dagli stakeholder di progetto.

Quali aspetti dell'applicativo ritiene fondamentali? Beh sicuramente l'applicazione deve simulare accuratamente una partita di dama, implementando tutte le regole base come mangiare le pedine ecc.. L'interfaccia può anche essere minimale, basta che sia intuitiva, dando ad esempio la possibilità di muovere i pezzi sia trascinandoli da una casella all'altra che premendo prima una e poi l'altra e mostrando una notifica quando la partita è finita. Voglio che la mia applicazione sia competitiva, pertanto deve esserci un qualche sistema di ranking con relativa leaderboard e i giocatori devono avere la possibilità di competere contro avversari del loro stesso livello per poi salire di rango in caso di vittoria o scendere in caso di sconfitta. Ah e dato che l'applicativo verrà utilizzato anche per tornei con premi in denaro, esso deve essere sicuro cosicché l'esito di una partita non possa venire in qualche modo alterato; inoltre deve essere possibile per un giocatore riconnettersi alla partita nel caso ad esempio salti la luce o si disconnetta per qualche motivo.

Quali sono le regole base di una partita a dama? La damiera si compone di 100 caselle alternate per colore, bianche e scure, e va posizionata con l'ultima casella in basso a destra di colore bianco. Ciascun giocatore dispone all'inizio di venti pedine, di colore diverso da quelle dell'avversario, collocate sulle prime quattro righe di caselle scure poste sul proprio lato della damiera. Il nero occupa le caselle dal n. 1 al n. 20, il bianco quelle dal n. 31 al n. 50. Inizia a giocare sempre il bianco. La pedina si muove sempre in diagonale sulle caselle scure, soltanto in avanti e di una casella alla volta. Quando una pedina raggiunge e si ferma per la prima volta su una delle cinque caselle dell'ultima riga diventa dama e va contraddistinta con la sovrapposizione di un'altra pedina. Ogni pedina può mangiare sia le dame che le pedine avversarie purché si trovino su una casella diagonale accanto alla propria, sia in avanti che indietro, e che abbiano la casella diagonale successiva libera; nel caso che dalla nuova posizione di arrivo si verifica un'identica situazione di presa, si deve effettuare "una presa multipla", cioè il maggior numero possibile di pezzi avversari. Nel caso che durante una presa multipla la pedina raggiunga una casella dell'ultima riga e debba ancora continuare a mangiare all'indietro, essa transita soltanto nell'ultima casella rimanendo pedina. La dama si muove sempre in diagonale, in tutte le direzioni possibili, spostandosi di un numero di caselle a propria scelta purché non occupate da altri pezzi. La dama mangia sempre in diagonale tutti i pezzi avversari che hanno almeno una casella successiva libera a qualsiasi distanza,

incrociando con le altre diagonali, dove è possibile per la presa del maggior numero possibile di pezzi avversari; potendo passare più volte sulle stesse caselle vuote, ma mai su uno stesso pezzo. A propria scelta, la dama può fermarsi in una casella libera posta sulla diagonale che segue il termine della presa. Avendo più possibilità di presa si è obbligati a mangiare dove si catturano più pezzi avversari, indipendentemente se la presa è effettuata da una dama o da una pedina o dalla qualità dei pezzi da catturare. I pezzi catturati debbono essere tolti dalla damiera in ordine progressivo una volta ultimata la presa. Si vince la partita per abbandono dell'avversario, che si trova in palese difficoltà, o quando si catturano o si bloccano tutti i pezzi avversari; nelle gare ufficiali si vince anche quando termina il tempo di riflessione a disposizione del giocatore avversario. Si pareggia in una situazione di evidente equilibrio finale, per accordo dei giocatori o per applicazione del regolamento tecnico.

Quanto tempo pensa sia giusto mettere a disposizione per effettuare una mossa? Non ho mai sentito parlare di tempistiche relative a turni di dama, però penso che sia giusto assegnare massimo un minuto a testa per effettuare una mossa e al suo scadere ci sarà un cambio turno. In questo modo però secondo me ci saranno casi in cui un giocatore potrebbe avvantaggiarsi nel non muovere pezzi, quindi dovrà essere dato un limite massimo di turni in cui un giocatore non fa alcuna mossa. Dopodiché si assegnerà vittoria al giocatore avversario.

Parla di una situazione di evidente equilibrio finale per pareggiare, non mi è molto chiaro il concetto dato che poi si parla di applicazione del regolamento tecnico: da come ho inteso i giocatori potrebbero chiedere una patta solo nel caso in cui il sistema rileverebbe a prescindere un pareggio per il regolamento o sbaglio? Sì nel caso di una partita fisica non vi sono controlli automatizzati pertanto un giocatore potrebbe accorgersi di una imminente situazione di pareggio, nel caso dell'applicativo direi che si possa considerare pareggio solo se lo rileva il sistema.

Come dovrebbe funzionare il meccanismo di matchmaking? Il giocatore deve avere la possibilità di ricercare rapidamente una partita che in automatico trova un avversario di rango simile, a patto che sia possibile. Deve essere anche possibile sfidare un giocatore specifico tramite invito diretto usando l'Username o un conoscente da lista amici.

Vuole quindi che venga implementato anche un sistema di lista amici? Hmmm sì direi che sarebbe il caso, magari anche con la possibilità di visualizzare la cronologia delle partite e le statistiche contro gli amici. Inoltre, una chat tra i due giocatori durante la partita sarebbe utile.

Per quale motivo sarebbe utile una chat di gioco? Penso che una chat all'interno di una partita sia molto utile per quando riguarda lo scambio di idee su metodologie di gioco o strategie adottate, oppure il semplice chatting tra persone sconosciute per dare la possibilità di fare amicizia ed eventualmente svolgere ogni tanto qualche altra partita contro il giocatore avversario.

Come si aspetterebbe il sistema di ranking? Mi aspetto che il gioco possa permettere di effettuare sia partite amichevoli che classificate dando più libertà di scelta al giocatore. Il sistema di ranking dovrebbe essere composto da:

- Punteggio di un giocatore che può aumentare o diminuire in base a se ha vinto o meno;
- Utilizzo di meccanismi di classificazione in base a determinati elo;
- Meccanismo di promozione elo al raggiungimento di un certo punteggio.

Siccome ha parlato di meccanismi di ranking, secondo lei dovrebbero venire implementati dei meccanismi per aiutare i giocatori meno esperti? Effettivamente ha ragione, tali meccanismi sicuramente renderebbero il gioco più user friendly per chiunque. Alcune semplificazioni che mi vengono in mente sono:

- Illuminare le pedine solo quando è il proprio turno;
- Evidenziare unicamente le celle delle pedine che possono essere mosse;
- In caso di presa, evidenziare unicamente le celle che possano permettere di mangiare un'altra pedina;
- Evidenziare le celle in cui andrà a finire una determinata pedina quando essa viene selezionata.

Può spiegarmi più di preciso come si immagina la riconnessione ad una partita? Allora, nel caso io venga disconnesso dalla partita per qualche motivo devo potermi ricollegare al sito e in automatico l'applicazione deve chiedermi se voglio abbandonare la partita o se voglio riconnettermi. Nel frattempo l'altro giocatore deve rimanere in attesa per un periodo di tempo determinato (2 minuti), dopo il quale viene considerato vincitore in automatico.

2.1.2 Ubiquitous Language

La metodologia DDD prevede, una volta ottenuta una buona conoscenza del dominio applicativo, la necessità di definire l'**Ubiquitous Language**. L'Ubiquitous Language è l'output prodotto dalla fase di **Knowledge Crunching** e l'artefatto generato dalla comprensione condivisa del dominio. Viene utilizzato per rappresentare un linguaggio esplicito usato per la descrizione del modello del dominio e del problem domain; permette di estrarre dei termini che verranno usati nell'implementazione del codice sorgente. L'UL contiene terminologie specifiche del business e mano a mano raffinato e migliorato, andando di conseguenza a riflettere questo cambiamento sull'implementazione del codice. Questo avviene perché il knowledge crunching è un processo continuo che non viene svolto solo all'inizio ma durante l'intero ciclo di vita del progetto, in modo da rendere anche il modello stesso malleabile a futuri cambiamenti.

Termini	Significato
Abbandono	Giocatore che decide di arrendersi, concedendo di conseguenza una vittoria all'avversario
Amico	Utente del sistema presente nella propria Lista Amici ***
Attesa	Periodo di tempo in cui un giocatore dovrà aspettare all'interno di una lobby senza poter fare alcuna mossa fino alla riconnessione o all'abbandono dell'avversario
Avanti	Direzione in cui possono muoversi le pedine
Avversario	Entità che prende il ruolo di rivale da battere all'interno di una partita
Casella	Spazio all'interno del tavolo da gioco in cui vengono posizionate le pedine
Casella libera	Casella in cui non è presente alcuna pedina o dama al suo interno
Chat	Meccanismo presente all'interno di una partita tramite il quale due giocatori possono scambiare messaggi testuali
Cronologia partite	Funzionalità che permette di memorizzare dati delle partite svolte da un determinato giocatore in ordine cronologico
Dama	Pezzo particolare del gioco che si ottiene solo quando una pedina raggiunge l'ultima riga della board e diventa una Dama, potendo effettuare delle mosse su diagonali più grandi
Damiera	Consiste nel tavolo da gioco specifico per lo svolgimento di una partita di Dama
Diagonale	Movimento consentito alle pedine per muoversi all'interno della damiera
Direzioni	La parte verso cui può essere effettuata una mossa
Disconnesso	Caso particolare in cui un giocatore non riesce più a comunicare con i servizi di gioco per cui rimane momentaneamente fuori dalla partita
Elo	Indica una determinata posizione del giocatore all'interno della classifica di gioco ***
Evidenziare	Funzione utilizzata per far capire ad un giocatore dove andrà a finire una pedina facendo una specifica mossa
Giocatore	Entità che prende parte al gioco
Illuminare	Proprietà grafica utile per far capire ad un giocatore quando è il suo turno
Leaderboard	Classifica globale di tutti i giocatori
Lista Amici	Lista di tutti i propri amici
Lobby	Stanza in cui si svolge una partita tra due giocatori
Modalità	Termine che indica la possibile introduzioni di varianti al gioco
Mossa	Possibilità di spostare una pedina all'interno della damiera rispettando le regole di gioco
Notifica	Comunicazione testuale per inviare notifiche ad un giocatore
Pareggio	Risultato di una partita in cui nessun giocatore ha vinto o perso

Partita	Una partita è composta da una serie di azioni secondo determinate regole in cui due giocatori si sfidano
Partite amichevoli	Un determinato tipo di partita che permette ai giocatori di potersi allenare senza perdere eventuali punti penalizzando il proprio elo
Partite classificate	Un determinato tipo di partita il cui risultato influisce sul proprio elo
Pedina	Elemento utilizzato all'interno del gioco per compiere delle certe azioni ***
Presa	Concetto che descrive la possibilità all'interno di una partita di poter mettere fuori gioco una pedina dell'avversario, in tal caso detta presa singola ***
Presa Multipla	Possibilità da parte di una pedina di poter effettuare più prese durante una mossa, se possibile è obbligatoria
Riconnettersi	Possibilità di un giocatore di potersi unire nuovamente ad una partita da cui era stato disconnesso ***
Sicuro	Significa che devono essere garantiti dei meccanismi per la sicurezza del sistema
Statistiche	Funzionalità che permette di memorizzare delle statistiche calcolate in base alle partite di un determinato giocatore
Tempo di riflessione	Tempistica entro la quale un giocatore dovrà effettuare una mossa
Turno	Periodo di tempo in cui un determinato giocatore potrà effettuare la propria mossa mentre l'altro no
Ultima riga	Riga della damiera che, se raggiunta da un giocatore, potrà far diventare la propria pedina una dama
Username	Nome tramite il quale si è registrato un utente
User Friendly	Facilitazioni atte a rendere il gioco più intuitivo
Vittoria	Risultato di superiorità conseguito al termine di una partita
Regole	Modalità secondo cui si svolge una partita (include posizionamento delle pedine, movimenti possibili, modalità di vittoria, tempistiche)

*** Alcuni termini dell'Ubiquitous Language utilizzati dal cliente sono considerati sinonimi di altri siccome, dato lo stesso contesto, il significato è il medesimo. Di seguito una lista di termini equivalenti:

- Amico = Conoscente;
- Elo = Rango, Punteggio, Rank;
- Pedina = Pezzo;
- Presa = Mangiare, Catturare, Bloccare;
- Riconnettersi = Ricollegarsi.

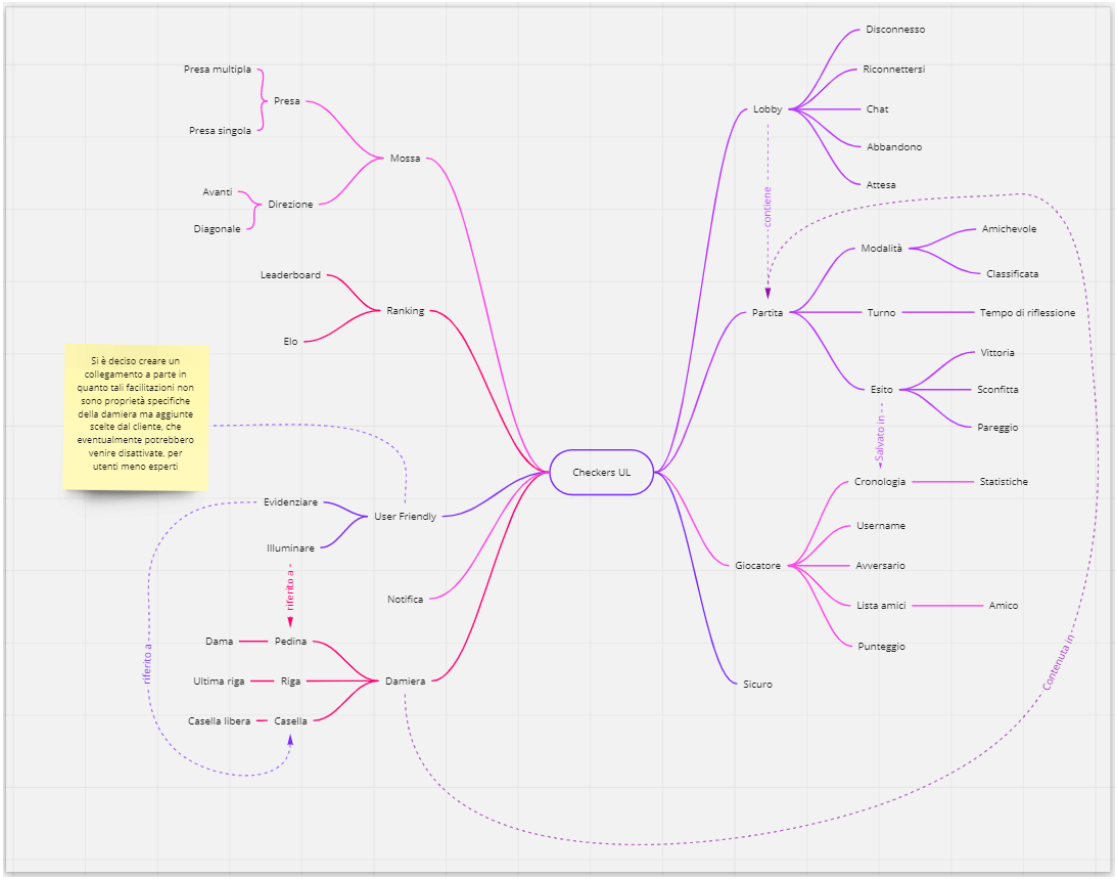


Figure 2: Mappa concettuale UL

2.2 Requisiti funzionali

Le funzionalità che dovrà fornire il sistema sono state definite tramite una stesura delle user stories e del diagramma dei casi d'uso.

2.2.1 User stories

Al termine di una prima fase di knowledge crunching si sono sviluppate user stories col fine di poter definire dettagliatamente i principali casi d'uso dell'applicazione da parte di un utente finale. Le user stories identificate sono le seguenti:

- In qualità di utente vorrei poter avere un mio account per accedere al gioco
- In qualità di giocatore vorrei poter unirmi ad una partita
- In qualità di giocatore vorrei poter creare una lobby a si potrà unire un altro giocatore

Figure 3: Fase di autenticazione

Figure 4: Modalità per unirsi ad una lobby

Figure 5: Creazione di una lobby e join da parte di un altro giocatore

- In qualità di giocatore vorrei poter invitare in lobby un giocatore specifico

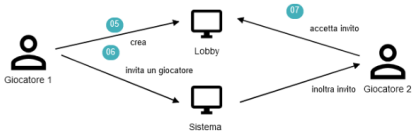


Figure 6: Invito ad una partita

- In qualità di utente vorrei poter visualizzare la cronologia delle partite che ho fatto e le mie statistiche

Figure 7: Ottenimento dati relativi ad un giocatore

- In qualità di utente vorrei poter consultare il mio profilo e modificarlo in caso di necessità
- In qualità di utente vorrei poter consultare la classifica globale, per verificare l'andamento della mia posizione al termine di una partita

Figure 8: Ottenimento profile di un utente

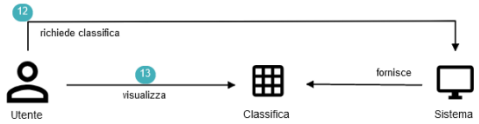


Figure 9: Consultazione classifica globale

- In qualità di giocatore vorrei poter scambiare messaggi con il mio avversario quando mi trovo in una lobby

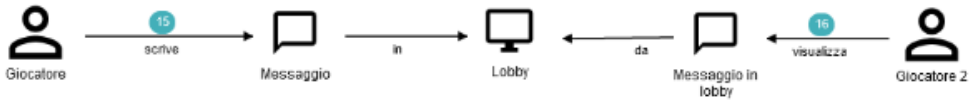


Figure 10: Scambio messaggi all'interno di una partita

- In qualità di giocatore vorrei poter effettuare una mossa quando mi trovo all'interno di una partita

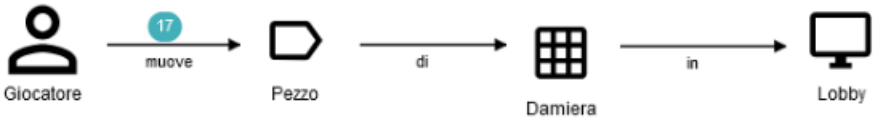


Figure 11: Effettuare una mossa

- In qualità di giocatore vorrei poter visualizzare mediante dei colori quali pedine potranno essere mosse durante il mio turno

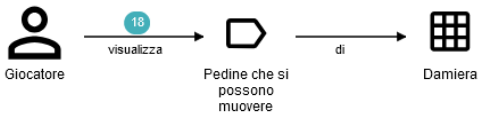


Figure 12: Visualizzare le pedine che si possono muovere

- In qualità di giocatore per ogni pedina che posso muovere voglio vedere evidenziate le possibili caselle di destinazione

Figure 13: Visualizzare le caselle di destinazione volendo muovere un determinato pezzo

- In qualità di giocatore vorrei poter abbandonare una partita

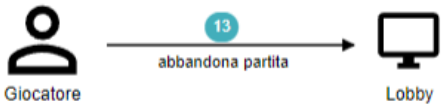


Figure 14: Abbandono partita

- In qualità di giocatore vorrei poter riconnettermi ad una partita in corso nel caso in cui mi sia disconnesso per problemi temporanei dovuti magari ad internet o assenza di corrente

Figure 15: Riconnessione partita

- In qualità di giocatore vorrei poter scegliere se avviare una partita amichevole o classificata

Figure 16: Creazione partita

2.2.2 Caso d'uso

A partire dalle user stories appena elencate è stato poi definito un seguente caso d'uso in modo da individuare i vari scenari di utlizzo del sistema. Il caso d'uso è composto dai seguenti attori:

- Utente: Attore generico all'interno del sito che svolge azioni basilari come per esempio registrazione, autenticazione, modifica profilo, ecc;
- Giocatore: Attore specifico che svolge una partita a Dama con un altro giocatore e che svolge azioni specifiche all'interno della partita.

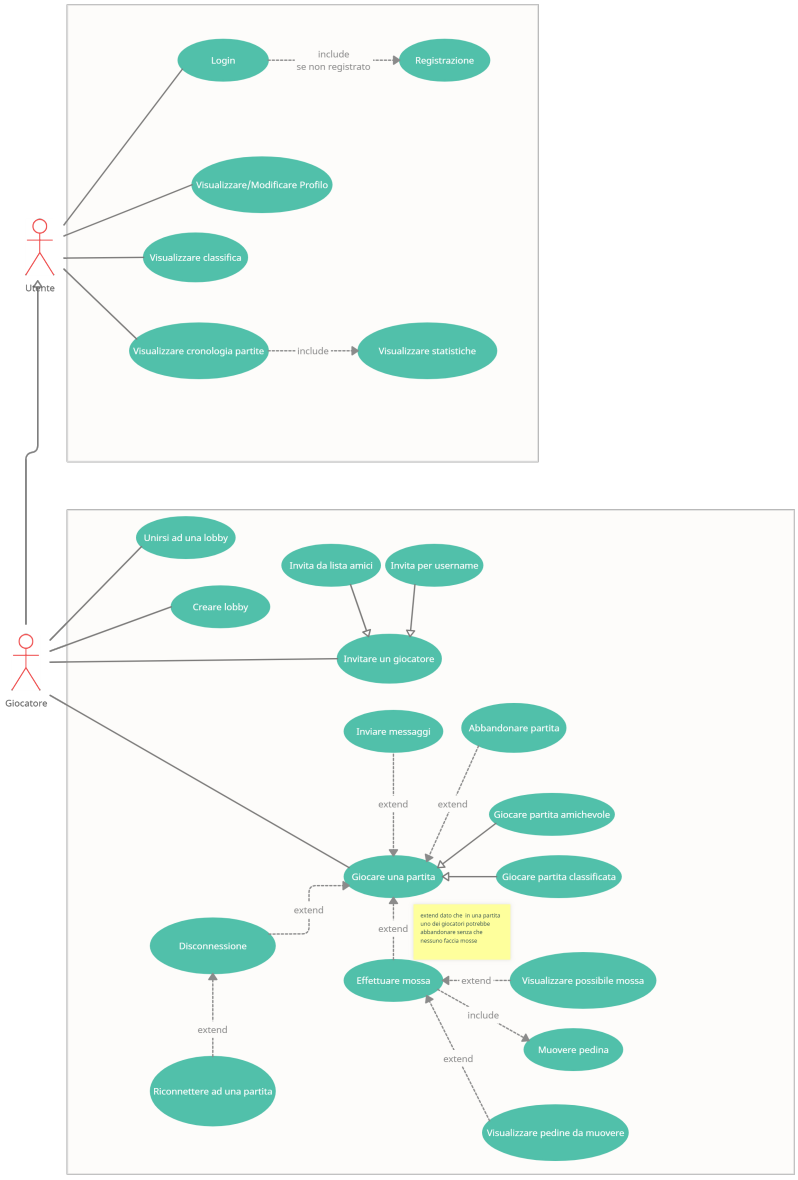


Figure 17: Caso d'uso dell'aplicazione Checkers

2.3 Requisiti non funzionali

I requisiti non funzionali che sono stati identificati durante la fase di analisi sono relativi essenzialmente all'architettura del sistema e all'organizzazione del codice stesso.

- Scalabilità: il sistema deve essere in grado di poter gestire un numero variabile di partite in contemporanea senza che abbia importanti perdite di performance;
- Disponibilità: il sistema deve cercare di fornire un alto grado di disponibilità;
- Qualità del codice: il codice prodotto deve essere comprensibile e di alta qualità;
- Disaccoppiamento: ogni parte del sistema dovrebbe interagire con le altre attraverso mezzi di comunicazione standardizzati e la re-attuazione di un componente in un diverso linguaggio di programmazione e/o framework dovrebbe avere un impatto minimo o nullo sugli altri componenti;
- Resilienza: un guasto di un'entità nel sistema non deve interromperne il funzionamento;
- Manutenibilità: le operazioni di manutenzione sull'impianto devono essere agevoli eseguire;
- Testabilità: il sistema deve essere facile da testare, nei suoi singoli componenti e nel complesso;
- Facilità di implementazione: il sistema ed i suoi componenti devono essere facili da implementare.

2.4 Tecnologie utilizzate

Basandosi sui requisiti funzionali e non funzionali citati precedentemente, sono state poi scelte le tecnologie da utilizzare per lo svolgimento del progetto.

Linguaggi

- Javascript: scelta per lo sviluppo della parte backend del sistema e anche di una parte lato client. Consiste in un linguaggio di programmazione multi paradigma orientato agli eventi, comunemente utilizzato nella programmazione Web lato client (esteso poi anche al lato server) per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso (mouse, tastiera, caricamento della pagina ecc...).
- HTML e CSS: utilizzati unicamente per la creazione dell'interfaccia grafica. HTML rappresenta un linguaggio di markup usato per il disaccoppiamento della struttura logica di una pagina web la cui formattazione viene gestita tramite CSS.

Framework

- Vue: si e optato per utilizzare la versione più recente di Vue per quanto riguarda la costruzione della parte client del sistema: l'utilizzo di Vue è stato preferito per semplicità ad altre tecnologie come Angular e React mentre si è scelta la versione più aggiornata del framework per evitare di utilizzare materiale obsoleto. Vue è un framework lato client ed è particolarmente utilizzato nello sviluppo web front-end. Ha un'associazione dati a due vie che consente uno sviluppo front-end senza interruzioni insieme a funzionalità MVC e applicazioni lato server interattive.

- Express: un framework stratificato su NodeJS, utilizzato per costruire il back-end di un sito utilizzando le funzioni e le strutture di NodeJS. Poiché NodeJS è stato sviluppato principalmente per eseguire JavaScript su una macchina invece di creare siti Web, ExpressJS è stato creato per quest'ultimo scopo.

Tools

- Gulp: Gulp è un task runner JavaScript. Gulp è un toolkit che consente di automatizzare flussi di lavoro lenti e ripetitivi per comporli in pipeline efficienti. Gulp viene usato spesso per automatizzare attività che normalmente bisognerebbe fare manualmente, come avviare un server web, compilare fogli di stile, ottimizzare risorse come immagini e JavaScript, aggiornare il browser ogni volta che viene salvato un file, eseguire unit test e analisi del codice, ecc.

- Docker: Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente.

Altre tecnologie

- MongoDB: dato che il sistema tratterà i dati organizzati in lobby indipendenti, la scelta di un DMBS non relazionale ha senso perché ovvia al problema del disadattamento di impedenza. Inoltre, MongoDB è adatto alla natura distribuita del progetto poiché è facile replicabile e scalabile.

- REST: stile architetturale che definisce le modalità di interazione tra i microservizi. REST è compatibile con qualsiasi protocollo o formato di dati, ma prevalentemente utilizza il protocollo HTTP e trasferisce i dati con JSON (JavaScript Object Notation). REST è una delle soluzioni più diffuse per ottenere dati dal web, grazie alla sua flessibilità, velocità e semplicità.

- WebSocket: scelto a favore di HTTP + REST per la comunicazione frontend-backend poiché sfrutta la messaggistica bilaterale, favorendo uno scambio di informazioni in entrambe le direzione e velocizzando di conseguenza la visualizzazione dei dati.

3 Analisi del Dominio

3.1 Bounded Context

Dopo la fase di analisi, il gruppo ha posto il focus su come modellare il sistema, in modo da realizzare una corretta progettazione dei concetti di dominio. La scelta dell'architettura ha seguito un approccio Domain Driven Design, che mette a disposizione vari modelli per realizzare un'architettura generale del sistema mediante strategical patterns. Tra le varie architetture consigliate, è stata identificata la struttura che più si addice a questo dominio, ovvero la microservice architecture. Questa scelta permette di realizzare un'applicazione distribuita basata su microservizi offrendo diversi vantaggi:

- Scalabilità
- Accessibilità
- Resilienza
- Apertura
- Interoperabilità

I Bounded Context che sono stati identificati durante questa fase sono i seguenti:

- Partita: responsabile della gestione di una partita tra due giocatori.

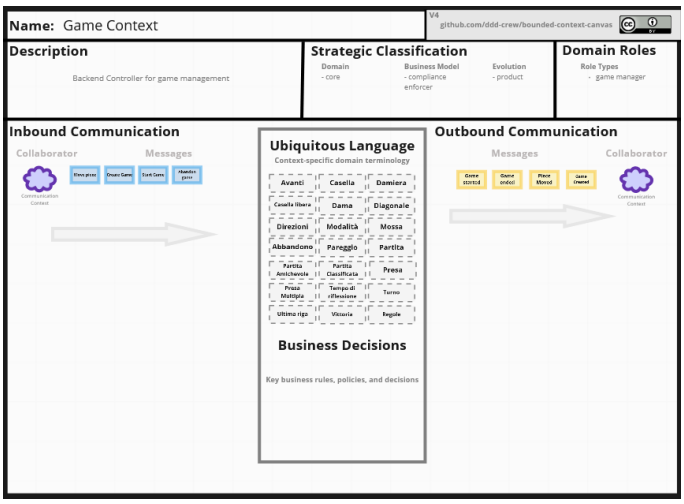


Figure 18: Bounded Context Canvas relativo alla partita

- Utente: custodisce le informazioni relative agli utenti del sistema come per esempio i dati del proprio profilo, vittorie/sconfitte e gestisce le fasi di autenticazione e registrazione.

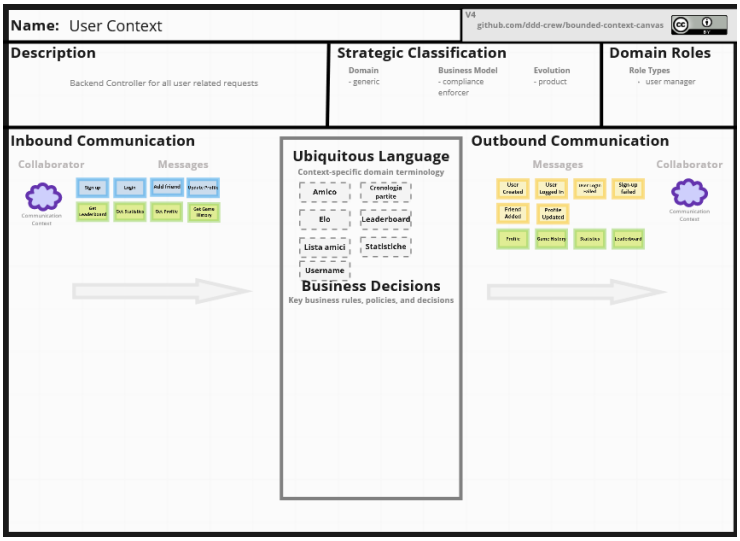


Figure 19: Bounded Context Canvas relativo all'utente

- Lobby: gestisce le lobby del sistema di gioco.

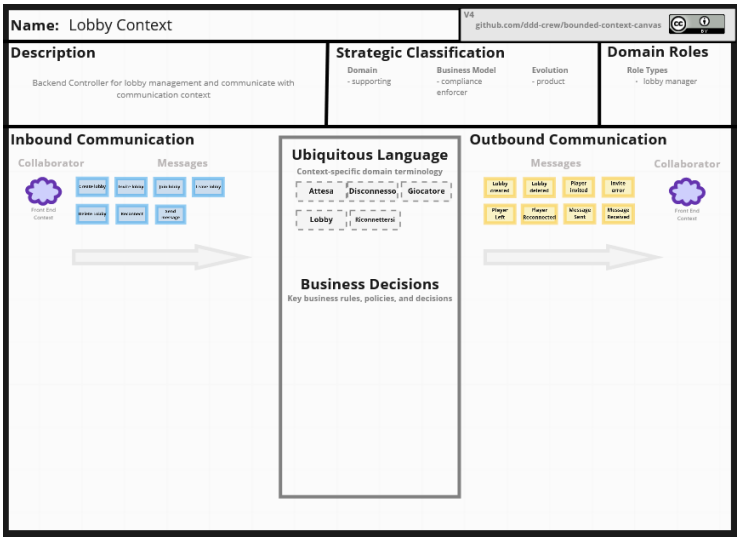


Figure 20: Bounded Context Canvas relativo alla lobby

- FrontEnd: gestisce la comunicazione del client utilizzando le API per contattare il servizio backend.

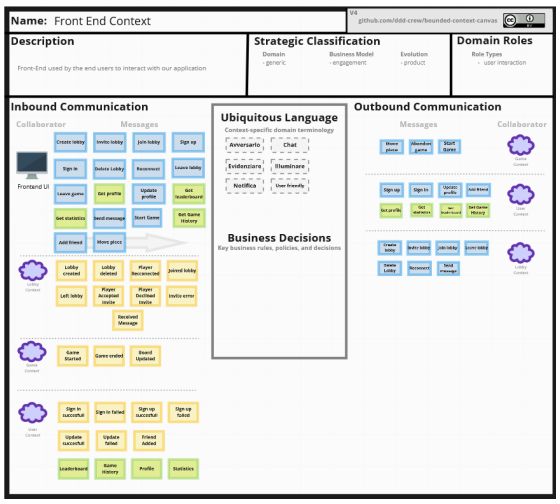


Figure 21: Bounded Context Canvas relativo al frontend

Essendo l'architettura a microservizi fortemente ispirata dal concetto di Bounded Context, la suddivisione del sistema in context ha facilitato l'identificazione dei microservizi. Da sottolineare che ad ogni Bounded Context non corrisponde necessariamente un microservizio, il quale potrebbe implementarne anche molteplici.
In seguito una rappresentazione della Context Map del sistema. Data la natura REST del sistema i context in downstream (in questo caso solamente FrontEnd) sono separati da Anti-Corruption Layer in modo da garantirne il disaccoppiamento.

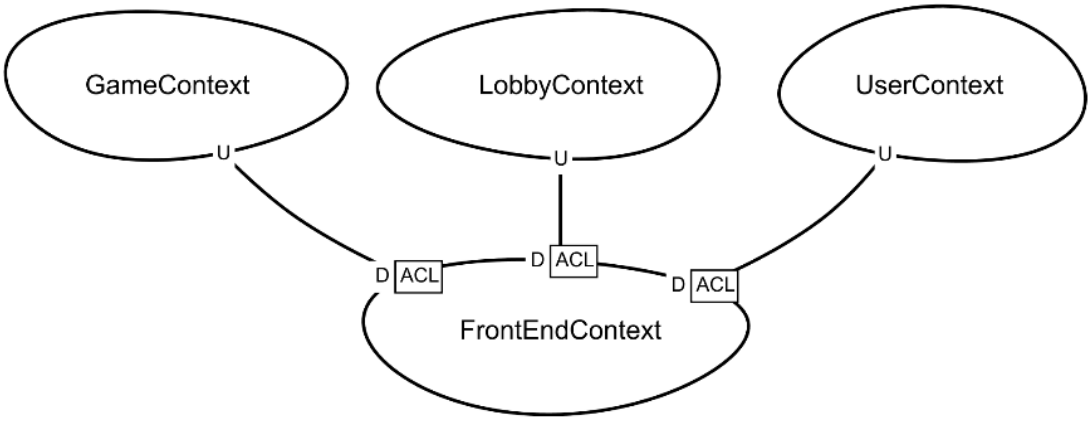


Figure 22: Context Map del sistema

3.2 Modelli di Dominio

Nella seguente sezione verranno definiti i modelli del dominio mediante tactical pattern e facendo riferimento ai bounded context identificati nella fase precedente. All'interno di ogni modellazione si individuano i seguenti concetti in comune:

- User: Rappresenta un aggregato che gestisce tutte le informazioni riguardanti uno specifico utente. A sua volta all'interno di questo aggregato si individua un'entità Player con i seguenti value objects: UserID, PalyerName, PlayerLastName e PlayerUsername.
- Lobby: Rappresenta un aggregato in grado di gestire le Lobby presenti all'interno del sistema come per esempio poterle creare o cancellare. All'interno di questo aggregato si individuano due entità che sono rispettivamente Chat e LobbyRoom contente i seguenti value object: LobbyID, Players, RoomName e Status.
- Game: Rappresenta un aggregato in grado di gestire le partite del sistema. A sua volta all'interno di questo aggregato si trova un'entità GameRoom con i seguenti value object: GameID, PossibleMoves, Status e PlayerTurn.
- Communication: Application Service responsabile della gestione delle transazioni, della ricerca degli aggregati corretti e dell'invocazione di certi metodi su di essi.
- Security: Domain Service che si occupa di fornire una corretta gestione sulla sicurezza delle comunicazioni che avvengono all'interno del sistema.

3.2.1 Game

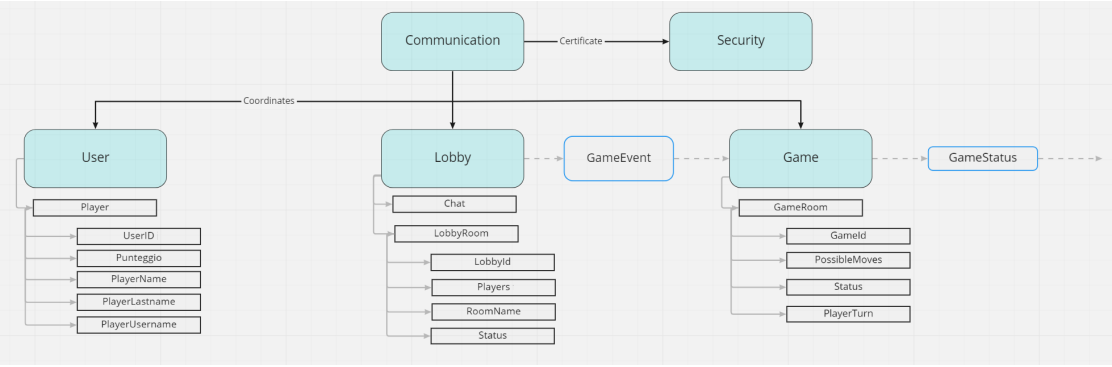


Figure 23: Tactical pattern applicato al sottodominio Game

Gli eventi di dominio presenti sono:

- GameStarted: Emesso quando due giocatori sono all'interno della stessa lobby

- GameEnded: Emesso quando la partita termina
- UdpateBoard: Emesso ogni qual volta venga effettuata una mossa da parte di un giocatore
- GameCreated: Emesso quando un giocatore decide di creare un game

3.2.2 Lobby

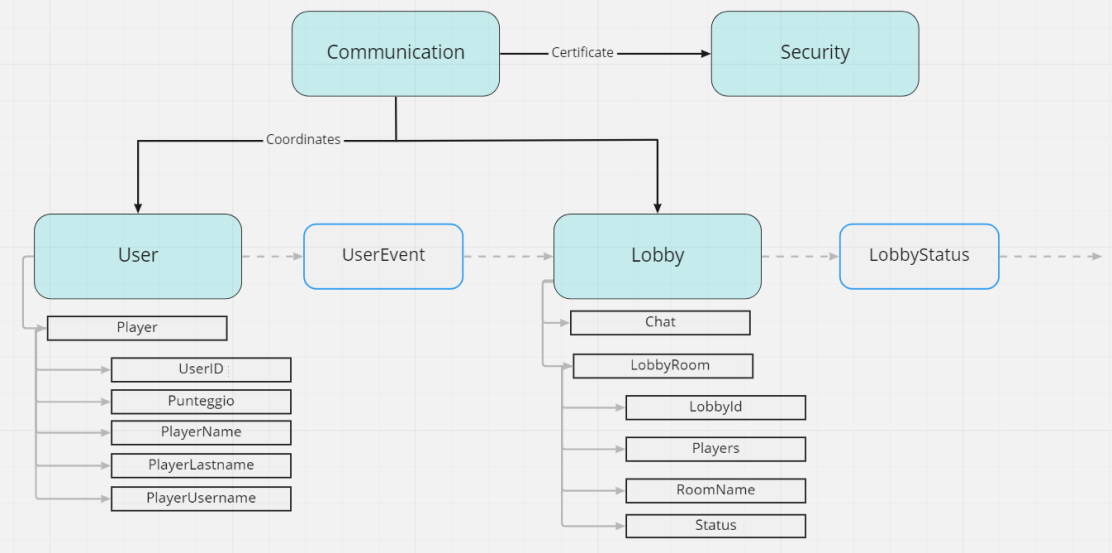


Figure 24: Tactical pattern applicato al sottodominio Lobby

Gli eventi di dominio presenti sono:

- LobbyCreated: Emesso quando un giocatore crea una propria lobby
- LobbyDeleted: Emesso quando un giocatore proprietario di una lobby decide di cancellarla
- PlayerInvited: Emesso quando un utente decide di invitare un altro giocatore in una lobby
- InviteError: Emesso quando un utente sbaglia ad inserire dei dati durante la fase di invito
- PlayerLeft: Emesso quando uno dei due giocatori decide di uscire dalla lobby
- MessaSent: Emesso quando un giocatore invia un messaggio in chat di gioco
- MessageReceived: Emesso alla ricezione di un messaggio da parte di un giocatore

3.2.3 User

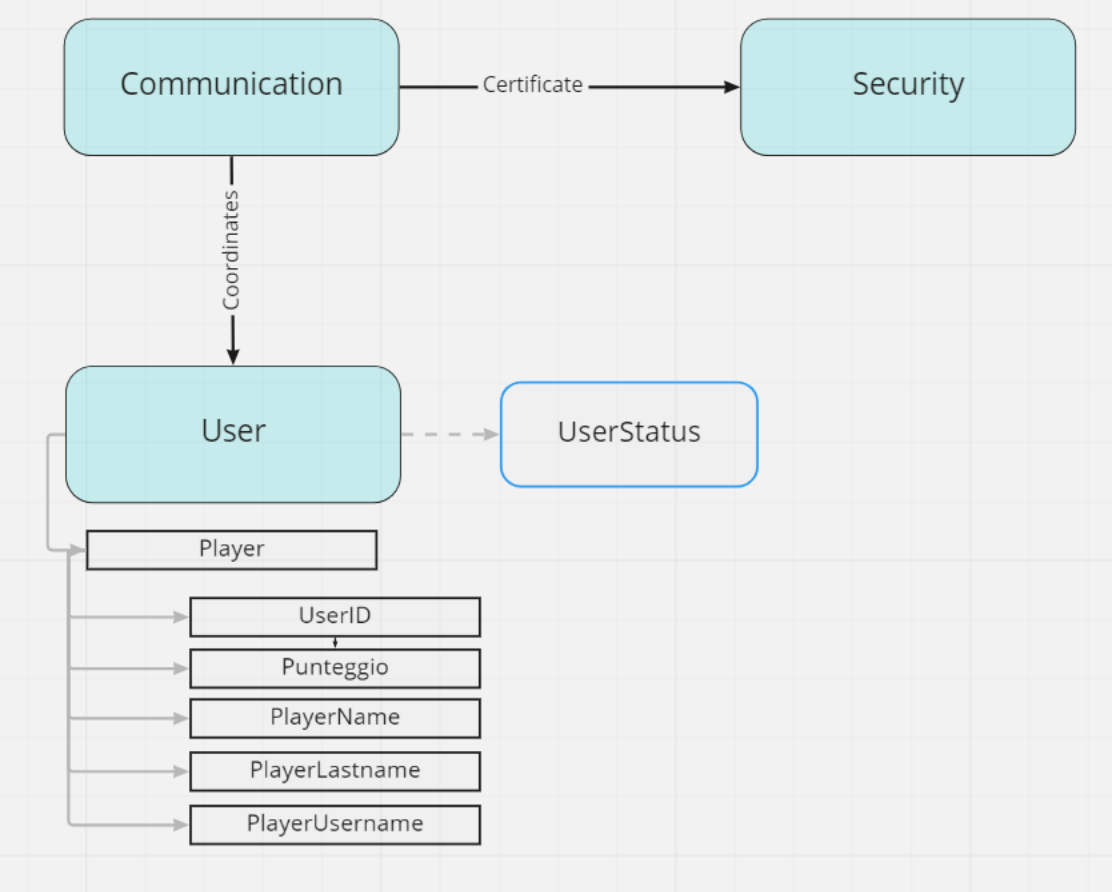


Figure 25: Tactical pattern applicato al sottodominio User

- UserCreated: Emesso quando un utente riesce a registrarsi con successo nel sistema
- UserLoggedIn: Emesso quando un utente riesce ad accedere con successo mediante username e password nel sistema
- UserLoginFailed: Emesso quando un utente ha sbagliato l'inserimento dei dati durante la fase di autenticazione
- SignUpFailed: Emesso quando un utente non ha soddisfatto alcuni parametri durante l'inserimento dati della registrazione
- ProfileUpdated: Emesso quando un utente aggiorna i propri dati profilo

4 Design

4.1 Clean Architecture

La **Clean Architecture** è un approccio che permette di scrivere applicazioni **Loosely-Coupled** ovvero con forte disaccoppiamento dell'applicazione dall'infrastruttura. Questo tipo di architettura separa UI, database, casi d'uso e dominio fornendo benefici relativi a:

- Indipendenza dal database e da agenti esterni
- Racchiude tutta la logica aziendale tramite i casi d'uso
- Facilmente testabile

Inoltre, la Clean Architecture permette di risolvere problemi tipici riguardo a: decisioni effettuate frettolosamente, difficoltà di cambiamento, troppo incentrati su framework o database, ci si concentra troppo su aspetti tecnici, difficoltà nel trovare le cose, logica aziendale sparsa. Ogni servizio del sistema segue una rigida architettura che lo suddivide in una serie di livelli, seguendo i principi della Clean Architecture. Poiché ci sono diverse opinioni e punti di vista su come dovrebbe essere questo tipo di architettura, il team ha elaborato una propria interpretazione degli strati contenuti in ciascun servizio e delle loro relazioni.

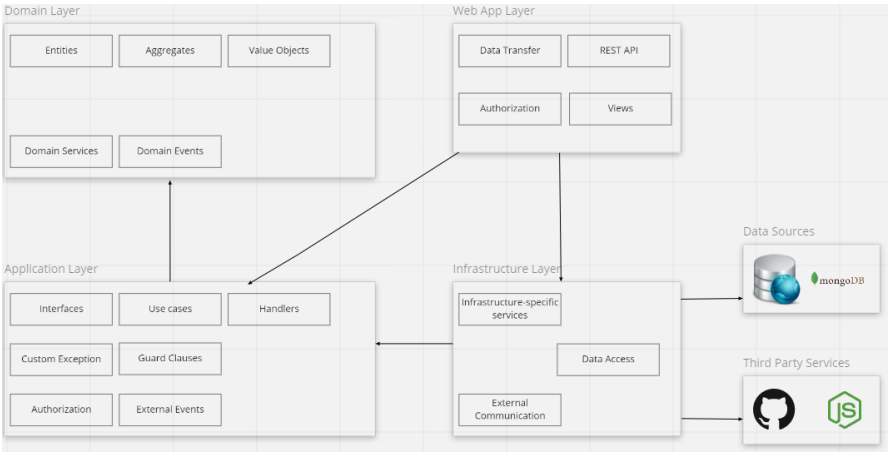


Figure 26: Clean Architecture del sistema

- Domain Layer: Rappresenta il livello centrale di ogni servizio in cui sono definite le regole di business del contesto. Contiene la definizione di entity, value objects entrambi organizzati in aggregati che permettano di definire il limite di coerenza. Questo livello definisce poi anche vincoli e processi di alto livello incapsulati all'interno dei domain services. Le entità ed i domain services possono notificare il cambiamento di stato utilizzando appositi domain events, che saranno però gestiti dall'application layer.

- Application Layer: Contiene una raccolta di tutti i casi d'uso del servizio in questione e le relative azioni per ciascuno di essi tramite un set di gestori che permettano di interagire sia con il domain layer che con l'infrastructural layer per svolgere tali azioni. I gestori possono essere utilizzati per eseguire la logica aziendale quando si verificano particolari domain events o per integrarsi con altri servizi/contesti utilizzando degli **eventi esterni** (proiezione di eventi di dominio che viene propagata ad altri contesti, utile per nascondere i dettagli del dominio interno, rendendo i contesti meno accoppiati). Inoltre, questo livello è responsabile **dell'autorizzazione**, accesso ai dati transazionali e comunicazione con servizi esterni. Per mantenere la logica dell'applicazione il più pulita possibile, le problematiche infrastrutturali come database, code di eventi o protocolli vengono mantenute lontane dal livello dell'applicazione attraverso **le interfacce dell'infrastruttura** dichiarate all'interno di questo layer, ma implementate dallo stesso livello dell'infrastruttura.

- Infrastructural Layer: Questo livello contiene l'implementazione concreta delle interfacce dichiarate dall'Application Layer o dal Domain Layer . Queste interfacce in genere modellano il comportamento relativo alla **persistenza** , alla **comunicazione esterna** e ad altri **servizi di utilità** richiesti da altri livelli.

- Web App Layer: Implementa il punto di ingresso utilizzato dai client del servizio per effettuare delle richieste utilizzando **API REST** esposte da ciascun microservizio, pertanto questo livello definisce gli endpoint REST supportati dal servizio insieme alla relativa interfaccia. A tal fine, ogni servizio contiene una serie di DTO (Data Transfer Objects, ovvero gli oggetti che modellano il formato di comunicazione con i client) per rappresentare formalmente l'interfaccia di ciascun endpoint. Infine, questo livello è responsabile della gestione del processo di autenticazione per determinare l'identità di chi effettua le richieste al servizio.

4.2 Architettura

Alla luce di quanto definito nelle fasi precedenti del progetto, sono state individuate all'interno del sistema due macro entità: **backend** e **frontend**.
Il primo comprende tutta l'architettura server-side di un'applicazione web mentre il secondo rappresenta la parte client-side. Ulteriormente il backend è stato scomposto in tre moduli realizzati sotto forma di microservizi: CommunicationService, UserService, GameService.

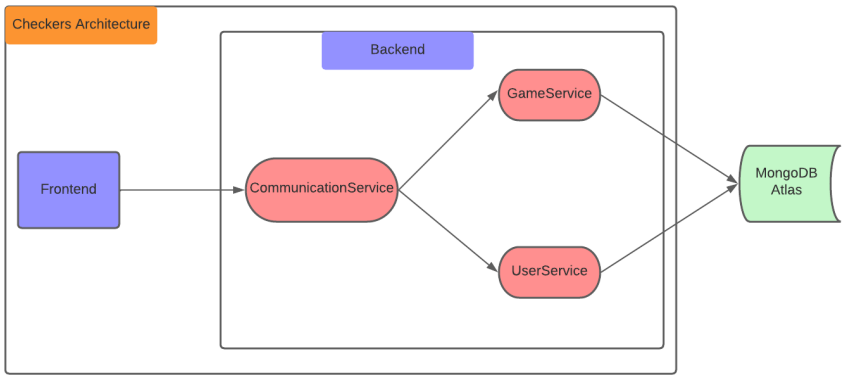


Figure 27: Visione generale architettura Checkers

In riferimento ai tre microservizi componenti il backend, l'idea alla base della scomposizione è derivante dalle fasi precedenti del progetto: la necessità di individuare e scomporre la parte server-side del sistema in componenti modulari ed indipendenti tra loro. Si è dunque giunti all'individuazione di tre funzionalità principali modellate tramite i conseguenti microservizi:

- gestione degli utenti e dei loro profili ⇒ **UserService**;
- gestione delle partite ⇒ **GameService**;
- gestione della comunicazione con i client ⇒ **CommunicationService**.

Infine, come evidenziato in figura 4.2, la proprietà di persistenza del sistema è garantita attraverso l'utilizzo di MongoDBAtlas, servizio che offre l'utilizzo di un database non relazionale tramite cloud; questa soluzione permette una forma di "outsourcing" della parte di storage dell'applicazione garantendo trasparenza rispetto al funzionamento del resto del sistema e, di conseguenza, la non necessità della gestione dello storage in termini di sicurezza, disponibilità e più in generale di gestione diretta: l'accesso al DB viene gestito attraverso le apposite API mentre la gestione ed il monitoraggio possono essere fatti attraverso l'apposita interfaccia web messa a disposizione dal servizio stesso.

4.3 Structure

Checkers è un'applicazione composta da microservizi, ciascuno dei quali uno scopo e delle determinate responsabilità. Tali microservizi sono stati realizzati utilizzando Node.js e Express ed hanno tutti una struttura:

- controllers: contiene la logica applicativa del servizio;
- models: definisce la struttura dei dati necessari al funzionamento di tale servizio: dati strutturati per la comunicazione con il database oppure strutture dati complesse utilizzate internamente al servizio;
- routes: contiene i metodi esposti dalle API REST;
- test: contiene i test unitari del microservizio;
- index: file principale utilizzato per inizializzare il singolo microservizio.

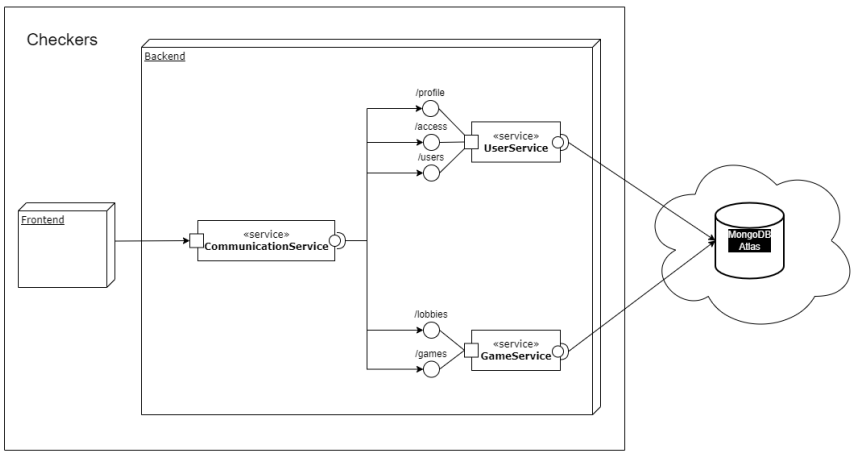


Figure 28: Diagramma dei componenti rappresentante i servizi interni al backend.

4.3.1 CommunicationService

Rappresenta il gateway del backend, la parte del sistema che si occupa di gestire le comunicazioni con i client e la logica di gestione di questi. Dal momento in cui un client si connette al sistema al momento in cui questo si disconnette è compito del CommunicationService garantire che questo possa svolgere tutte le operazioni per le

quali è autenticato.
Sono compiti del CommunicationService:

- gestire connessione e disconnessione client;
- organizzare i client in lobby;
- effettuare access control;
- gestire la logica applicativa dell'applicazione.

In particolare l'ultimo punto certifica la centralità del servizio CommunicationService: è il componente core e l'unico pro-attivo del backend, dovendo tutte le comunicazioni passare attraverso esso, non si è ritenuto conveniente spostare la logica appicativa su un altro servizio. Questo servizio si appoggia al GameService per la gestione delle singole partite di dama e al UserService per la gestione degli utenti e delle loro proprietà, oltre che per verificare l'integrità dei client e la corrispondenza tra identità fisica (socket) ed identità digitale del client (autenticazione). E' inoltre compito di questo servizio garantire la disponibilità del sistema intercettando e gestendo eventuali errori garantendo dunque trasparenza agli utilizzatori del sistema.
Come visibile in figura 4.3, è la parte del sistema che comunica con il frontend e di conseguenza con i client.

4.3.2 UserService

Rappresenta il servizio che si occupa della gestione degli utenti, è il servizio dedicato alla connessione alla parte del database che modella gli utenti dell'applicazione; è un servizio reattivo che risponde alle interrogazioni del SocketService. In particolare, è la parte del sistema che si occupa di garantire l'integrità dei client e verificarne l'autenticazione. Ogni qual volta un utente effettua un'azione per la quale è necessaria autenticazione, il SocketService contatta questo servizio che verifica l'autenticazione del client attraverso l'utilizzo di JSON Web Token[2].
E' inoltre un componente stateless, la persistenza del flusso dati che passa da questa entità è garantita da MongoDB, sul quale vengono salvate le uniche informazioni necessarie al UserService per un corretto funzionamento del sistema. Tale proprietà garantisce una maggiore scalabilità e robustezza al sistema, un'eventuale interruzione del servizio potrebbe essere gestita senza la necessità di dover recuperare il lavoro che stava svolgendo il servizio al momento dell'interruzione.
Come visibile in figura 4.3, il servizio espone tre endpoint:

- **/profile**: comprende tutte le richieste inerenti ad un singolo utente e la gestione del profilo di questo;
- **/users**: comprende tutte le richieste inerenti a più utenti, come ad esempio la classifica globale;

[2] `https://jwt.io/`

- **/access**: la parte più critica del servizio, comprende tutte le richieste inerenti il controllo degli accessi degli utenti all'interno del sistema;

4.3.3 GameService

Rappresenta il servizio che si occupa della gestione delle singole partite di dama, dalla loro creazione alla terminazione, e dell'interazione con la porzione di database che modella le partite tra utenti. Specularmente al UserService, anche questo servizio è di natura reattiva e risponde alle interrogazioni del SocketService. Il servizio espone le seguenti API: Allo stesso modo del UserService, anche il GameService è un'entità stateless.
Come visibile in figura 4.3, il servizio espone due endpoint:

- **/game**: comprende tutte le richieste inerenti ad una singola partita di dama e la gestione di questa, come ad esempio l'inizializzazione di una partita e lo spostamento di una pedina da una casella x ad una casella y;

- **/lobbies**: comprende tutte le richieste inerenti più partite, come ad esempio la richiesta di ricercare tutte le partite di un determinato utente z;

4.4 Behaviour

4.4.1 GameService/UserService

GameService e UserService sono componenti passivi e stateless del sistema; il loro comportamento è rispondere alle interrogazioni del CommunicationService ogni volta quest'ultimo necessita di interrogarli. Interagiscono unicamente con CommunicationService.
Per garantire la natura stateless di questi due servizi, si è deciso di pagare il prezzo di query frequenti sul database: prendendo il caso del GameService, ogni partita di dama è identificabile da una stringa FEN che rappresenta la posizione delle pedine sulla scacchiera: invece di mantenere in locale una lista delle partite in corso, il GameService associa ad ogni partita creata un identificativo e salva il FEN corrispondente a tale partita sul database; ogni qual volta uno dei due giocatori coinvolti in quella partita x compie una mossa, GameService si occupa di recuperare la partita x dal database ed aggiornare il FEN di quella determinata partita sulla base della mossa appena compiuta da uno dei due giocatori.

4.4.2 CommunicationService

Svolge la funzione di gateway dell'applicazione, il "middleware" tra gli utenti che utilizzano il sistema (attraverso il Frontend) ed il Backend dell'applicazione e si occupa di soddisfare le richieste dei primi; al contrario dei restanti due microservizi, il CommunicationService è un componente attivo e stateful, mantiene una lista aggiornata degli utenti connessi e si occupa della gestione della loro esperienza sul sistema, contattando a necessità il UserService ed il GameService.

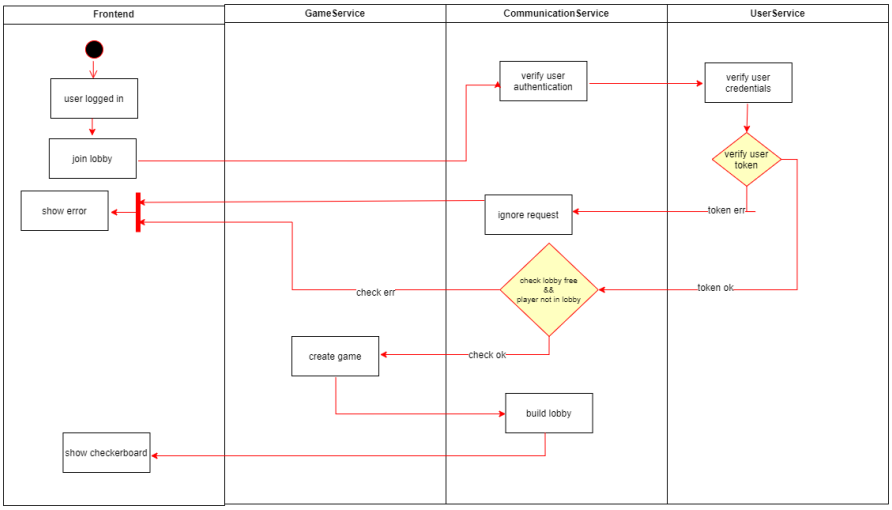


Figure 29: Diagramma di attività mostrante come i servizi dialogano tra di loro durante la procedura di login.

In figura 4.4.2 è possibile vedere, nel contesto del tentativo di un client di iniziare una partita con un secondo client, la centralità del servizio CommunicationService e la passività dei restanti due, i quali rispondono solamente alle interrogazioni del primo.

4.5 Interaction

E' possibile individuare all'interno del sistema quattro entità che comunicano tra loro: i tre servizi componenti il backend e l'entità umana, che comunica con la parte di backend del sistema attraverso un client, e la parte frontend. I flussi di comunicazione sono due:

- **client - CommunicationService**: rappresenta il flusso di comunicazione che intercorre tra il client utilizzatore del sistema ed il backend del sistema, parte dal momento in cui un utente effettua un'azione sul frontend al momento in cui tale azione, tradotta in richiesta dal frontend, raggiunge il CommunicationService. Per questo flusso di comunicazione si è scelto di utilizzare la tecnologia HTTP, in particolare la scelta è ricaduta su Socket.IO[3].

- **CommunicationService - UserService/GameService**: rappresenta il flusso di comunicazione che intercorre tra i servizi che compongono il backend, racchiude tutte le comunicazioni dal momento in cui una richiesta arriva al CommunicationService fino al momento in cui tale richiesta viene processata e generata una risposta da inviare al client.
  Per questo flusso di comunicazione si è scelto l'utilizzo della tecnologia HTTPS, implementata tramite axios [4].

[3] `https://socket.io/`
[4] `https://axios-http.com/docs/intro`

Segue l'immagine rappresentante il flusso di comunicazione tra i servizi nello stesso contesto della figura 4.4.2

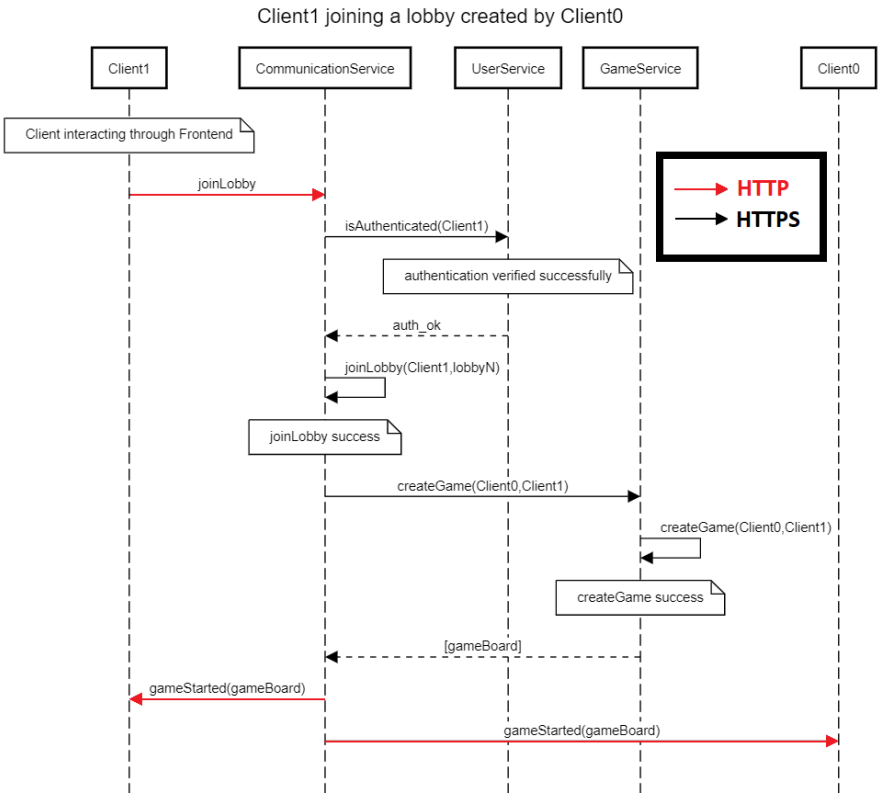


Figure 30: Caso applicativo di interazione tra client ed i tre servizi

4.5.1 HTTP/HTTPS

Axios è una libreria che mette a disposizione un client in grado di effettuare richieste HTTP ed HTTPS; per la seconda opzione è necessario l'utilizzo di un certificato valido; visto il contesto accademico di questa applicazione ci si è limitati a "costruire" un piccolo environment fittizio all'interno del quale una certification authority fittizia possiede un certificato auto-firmato con il quale sono stati firmati tre certificati, ognuno assegnato ad uno dei microservizi componenti il backend.
Ogni qual volta UserService e GameService ricevono una richiesta HTTPS, provvedono a verificare se il certificato del richiedente è valido, effettuando la procedura di verifica utilizzando la chiave pubblica della CA fittizia. In questo modo, UserService e GameService risponderanno solamente alle richieste di quelle entità che hanno un certificato valido, garantendo la non possibilità per una terza parte di bypassare il CommunicationService(unico altro possessore di un certificato valido) e tentare di comunicare direttamente con GameService ed UserService.
Sarebbe stato ideale estendere la comunicazione HTTPS anche tra i client del sistema ed

il CommunicationService ma durante la fase di progettazione si sono associati i seguenti problemi a tale realizzazione, che hanno portato il team a propendere per una connessione HTTP tra frontend e backend:

- necessità di installare il certificato fittizio in ogni browser che volesse lanciare l'applicazione;
- necessità di far riconoscere al browser un certificato auto-firmato come autorevole;

5 Dettagli Implementativi

5.1 Backend

5.1.1 Gestione utenti

Gli utenti vengono memorizzati all'interno del server attraverso una mappa che mette in correlazione il client che ha aperto la connessione col server e l'utente registrato a lui associato, nella forma

```
const online_users = new BiMap()  //{  client_id <-> user_mail  }
```

L'utilizzo di una BiMap si è reso necessario dal momento in cui non sempre il problema da risolvere è "dato il client che ha appena inviato una richiesta trovare l'utente corrispondente"; prendendo in esame la casistica in cui un utente X invita un utente Y, X conosce la mail di Y ed il server deve essere in grado di risalire dalla mail di Y al corrispondente client al quale comunicare l'invito. L'utilizzo di questa BiMap non rompe il vincolo di integrità della mappa in quanto così come l'id del socket associato ad un utente, anche le mail sono univoche all'interno del sistema. Questa scelta comporta un overhead di gestione dovuto alla BiMap che può essere vista come un wrapper su una coppia di mappe.

5.1.2 Gestione inviti

La gestione degli inviti da un giocatore ad un altro ha richiesto una modellazione particolare visto il numero di possibili casistiche in cui si può trovare il sistema; nel dettaglio si è deciso di permettere ad ogni utente X invitare più utenti contemporaneamente; il principio di accettazione dell'invito è FIFO, il primo tra gli utenti invitati che accetta l'invito sarà l'avversario di X; per tutti gli altri, nel caso in cui accettino l'invito, sarà necessario mostrare un errore. Ogni invito ha una durata temporale limitata terminata la quale l'invito deve essere considerato scaduto.

```
const invitation_timeouts = new Map() // {host_id -> {opponent_id -> timeout}}
```

In questo modo, ad ogni utente X è in grado di inviare un invito ad ogni giocatore Y ed ad ogni coppia X_i, Y_i è associato un timeout che specifica la durata dell'invito. Nel momento in cui un giocatore Y_i accetta l'invito di un giocatore X_i, la mappa soprastante viene passata e vengono resettati e successivamente rimossi tutti i timeout ed i conseguenti inviti che coinvolgono X_i ed Y_i

```
//If the player invited has invited someone else, delete those invites
if(invitation_timeouts.has(Y)){
    for(const [_,invite] of invitation_timeouts.get(Y)){
        clearTimeout(invite)
    }
}
//Delete every invite made by the inviting player X
```

```
for(const [_,invite] of invitation_timeouts.get(X)){
        clearTimeout(invite)
}
```

I giocatori che hanno ricevuto un invito da X non vengono notificati della scadenza dell'invito ricevuto, tuttavia, nel caso in cui accettino un invito scaduto il server provvede a notificare il corrispondente "invitation_expired"

5.1.3 Gestione lobby

Le lobby sono un concetto importante all'interno del sistema, un utente può creare una lobby attendendo che un giocatore esterno entri per poi iniziare una partita oppure può invitare, tramite indirizzo mail, un utente all'interno della propria lobby. Tale concetto è stato modellato mediante l'utilizzo della funzionalità "rooms" offerta dalla libreria socket.io: è possibile organizzare le connessioni in "stanze" logiche, che facilitano la comunicazione di messaggi ai socket presenti nella stanza:

```
//client joining a lobby
client.join(lobby_id)
//sending comunications to lobby
io.to(lobby_id).emit(someLobbyMessage)
```

La lobby logica è stata affiancata da un modello "fisico" di lobby, definito all'interno del SocketService:

```
class Lobby{
  constructor(stars, roomName, turn) {
    this.stars = stars;
    this.roomName = roomName;
    this.turn = turn;
    this.tie_requests = [];
    this.players = [];
    this.players.push(turn);
  }
}
```

Tale modello permette al SocketService di avere una rappresentazione fisica di una lobby attraverso la quale facilitarne la gestione.

5.1.4 Gestione disconnessione

La gestione della disconnessione di un utente necessita di particolare attenzione visto il numero di situazioni diverse all'interno del quale il client si può trovare, nel dettaglio:

```
async function handle_disconnection(player){
  //Player isn't in a lobby so just disconnect it
  if(!isInLobby(player)){
```

```
    //Disconnect user
  }else{
    //player is in a lobby, check if it's empty or he is in a game
    if(lobby.isFree()){
        // Lobby is free, Just delete the lobby and make client leave
    }else{
      //Lobby is full and a game is on, need to check if it's the host or not
        if(lobby.getPlayers(0) == player){
          //Player disconnecting is the host
        }else{
          //Player disconnecting is the guest
        }
        //Clear lobby room and assign points to winner and loser
    }
 }
}
```

5.1.5 Gestione asincronia

Per la gestione dell'asincronia del codice si è deciso di utilizzare il costrutto async/await offerta da Javascript: in questo modo è possibile avere codice più pulito rispetto all'utilizzo di catene di promise e callback.

```
let moveResult = await network.askService('put',
↪  `${gameService}/game/movePiece`, { gameId: lobbyId, from, to });

io.to(lobbyId).emit('update_board', moveResult.board);
```

5.2 Frontend

Il client frontend implementato nel progetto è un proof of concept su come è possibile interagire con il sistema stesso, e come tale ha una struttura molto semplice. La comunicazione WebSocket è gestita da una serie di API messe a disposizione per contattare il backend e ricevere determinate risposte, fra cui:

- Aprire la connessione verso il backend all'avvio;
- Intercettare e analizzare i vari messaggi in arrivo;
- Ha un elenco di callback che possono venire allegate dall'esterno ed essere eseguite una volta per ogni messaggio, in modo da poter aggiornare l'interfaccia utente;
- Possiede vari metodi per inviare nuovi messaggi al back-end.

Oltre al framework VUE, altre due librerie principali utilizzate per il frontend sono:

- vue-socket.io e socket.io-client per la gestione della comunicazione tramite WebSocket;

- Tailwind CSS e DaisyUI per lo stile dell'interfaccia.

5.3 Docker

Tutte le immagini dei service sono state realizzate a partire dal risultato del task gulp: build; avviato per ogni microservizio. Tale task produce come risultato i file *build.js*, utilizzati cadauno (assieme al package.json) dal DockerFile per realizzare l'immagine del relativo service come nell'esempio in seguito:

```
FROM node:lts-alpine

ARG ARG_PORT
ARG ARG_DB_PSW
ARG ARG_GAME_KEY
ARG ARG_GAME_CERT
ARG ARG_CERTIFICATE

ENV GameService_PORT=$ARG_PORT
ENV DB_PSW=$ARG_DB_PSW
ENV GAME_KEY=$ARG_GAME_KEY
ENV GAME_CERT=$ARG_GAME_CERT
ENV CERTIFICATE=$ARG_CERTIFICATE

RUN mkdir -p /app/
WORKDIR /app
COPY . .
RUN npm install --unsafe-perm
RUN npm run gulp build

WORKDIR /app/build
EXPOSE $ARG_PORT
CMD npm run deploy
```

Tutti i container utilizzano la distribuzione di linux ***alpine***, particolarmente leggera ed efficiente.

Il file docker-compose.yml realizza le immagini dei servizi passando i relativi ARG e taggandole con l'indirizzo del container registry di DigitalOcean.

6 OPS

Con OPS o Operations si intende tutto le pratiche atte a garantire la manutenzione, il monitoraggio e la risoluzione dei problemi negli ambienti di produzione.
Come piattaforma per la Continous Delivery e Continous Integration (CD/CI) è stato utilizzato Github Actions, permettendo di creare workflow per la build automation, testing dei push/PR e di deployment delle release.

6.1 Build Automation

Data la natura disaccopiata dei microservizi sono stati utilizzati workflow separati per ogni microservizio nonchè uno per il FrontEnd. Ciascuno di tali workflow può venire azionato nei seguenti modi:

- Push: alla ricezione di un push sul repository che effettui modifiche ad un microservizio aziona il relativo workflow
- Workflow-Dispatch: permette di azionare un workflow direttamente dall'interfaccia grafica di Github Actions
- Worflow-Call: utilizzato durante la fase di deployment, permette di azionare un workflow da un workflow differente (in modo da riutilizzare i workflow di testing)

Ciascun workflow utilizza una matrice di build per eseguire il workflow su ambienti diversi, come:

- Sistema operativo: Windows, Ubuntu o MacOS
- Node, versioni: 14, 16 e 18

Vengono inoltre impostate delle variabili ambientali nel workflow per ottenere dati sensibili come password o token direttamente dai Secret di Github Actions, evitandone la trasmissione in chiaro.

6.1.1 Dependabot

La gestione e l'aggiornamento automatico delle dipendenze è stato implementato mediante Dependabot, servizio offerto da Github Actions il quale una volta configurato controlla regolarmente la disponibilità di aggiornamenti per le librerie utilizzate e nel caso effettua pull request.
La configurazione scelta prevede un controllo con cadenza giornaliera delle dipendenze con branch target ***develop***. Nel caso vi sia una pull request generata da dependabot che supera con successo tutti i test, un'ulteriore workflow effettua il merge automatico.

6.1.2 Delivery e Deployment

La politica di versioning adottata da questo progetto è semantic versioning, si è scelto di utilizzare un tag unico ricavato mediante il plugin GitVersion per tutti i microservizi. GitVersion è stato configurato in modalità Mainline in modo da incrementare in automatico la SemVer per ogni commit su branch main ed applicarla alla release dal relativo workflow.
Dato che un cambiamento nelle API esposte da un microservizio potenzialmente causerebbe malfunzionamenti negli altri, si è scelto di utilizzare un tag unico per tutti i microservizi.

Il deployement è stato realizzato mediante un workflow azionato in caso di push sul branch main. Esso consiste nel trasferimento delle docker image dei servizi su un droplet acquistato dalla piattaforma DigitalOcean e release su Github. Nello specifico, il workflow deploy effettua le seguenti azioni:

- Vengono azionati i workflow di test relativi a ciascun microservizio
- Viene calcolata la Semantic Version della release
- Mediante il comando docker-compose vengono buildati i file e create le immagini
- Viene effettuato il login sulla piattaforma mediante un plugin apposito
- Le vecchie immagini vengono rimosse dal Container Registry di DigitalOcean
- Le nuove immagini vengono trasferite nel Container Registry di DigitalOcean direttamente mediante Docker CLI
- Per evitare ridondanza viene trasferito anche il file docker-compose.yml
- I vecchi container vengono fermati ed eliminati per azionare quelli nuovi mediante comando docker-compose
- Utilizzando la Semantic Version calcolata ed il commit message più recente viene creata la release su Github (come draft)

Dato che lo storage del Container Registry a disposizione è abbastanza limitato, è stato realizzato un workflow con schedule tutti i giorni alle ore 00:15 per effettuare il garbage collection del Container Registry.
Inoltre, per garantire l'autenticità dell'autore è stata generata una coppia di chiavi SSH ed inoltrata la chiave pubblica al droplet DigitalOcean.

6.1.3 Quality Control

Per il testing sono state utilizzate le librerie mocha e chai, oltre ad axios per effettuare richieste alle API esposte dai microservizi.
Nello specifico per ogni microservizio sono stati effettuati i seguenti test:

- GameService e UserService: per ciascuna route sono state effettuate chiamate in modo da simulare tutti i possibili scenari di esecuzione, tra cui chiamate con parametri errati o che ci si attende generino errori. In seguito, mediante le funzioni expect i risultati sono stati confrontati con quelli attesi.

- CommunicationService: per ciascun API messa a disposizione, sono state effettuate delle chiamate tramite un client simulato che in base ad un certo evento dovrà ricevere una specifica risposta. La correttezza delle informazioni ricevute è stato effettuato mediante degli assert che verificassero l'esatta ricezione di una determinata risposta. Sono stati implementati anche per quest'ultimo degli scenari che generassero degli errori.

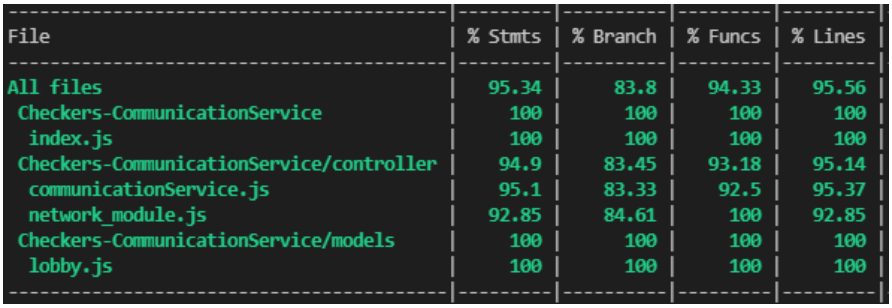
La coverage dei test è stata calcolata utilizzando la libreria Istanbul, in seguito i risultati.

File	% Stmts	% Branch	% Funcs	% Lines
All files	94.98	85.48	100	94.94
Checkers-GameService	100	100	100	100
index.js	100	100	100	100
Checkers-GameService/controllers	93.42	93.75	100	93.33
gameController.js	93.42	93.75	100	93.33
Checkers-GameService/models	100	100	100	100
gameModel.js	100	100	100	100
Checkers-GameService/routes	90.9	75	100	90.9
routes.js	90.9	75	100	90.9
Checkers-GameService/test	100	50	100	100
test.js	100	50	100	100
Checkers-GameService/test/utils	82.6	50	100	82.6
axiosRequests.js	82.6	50	100	82.6

Figure 31: Coverage di GameService

File	% Stmts	% Branch	% Funcs	% Lines
All files	95.33	88.63	100	95.31
Checkers-UserService	100	100	100	100
index.js	100	100	100	100
Checkers-UserService/controller	94.26	88.37	100	94.26
userController.js	94.26	88.37	100	94.26
Checkers-UserService/models	100	100	100	100
userModel.js	100	100	100	100
Checkers-UserService/routes	100	100	100	100
routes.js	100	100	100	100

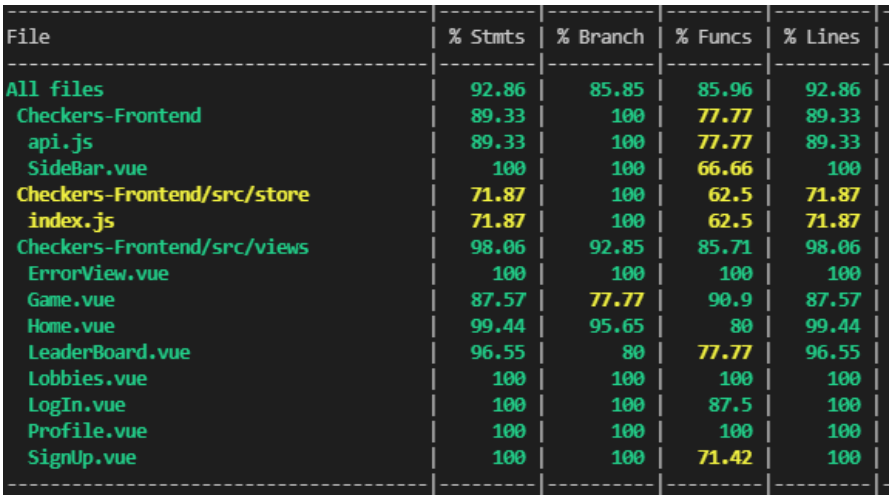
Figure 32: Coverage di UserService



File	% Stmts	% Branch	% Funcs	% Lines
All files	95.34	83.8	94.33	95.56
Checkers-CommunicationService	100	100	100	100
index.js	100	100	100	100
Checkers-CommunicationService/controller	94.9	83.45	93.18	95.14
communicationService.js	95.1	83.33	92.5	95.37
network_module.js	92.85	84.61	100	92.85
Checkers-CommunicationService/models	100	100	100	100
lobby.js	100	100	100	100

Figure 33: Coverage di CommunicationService

Mentre per quanto riguarda il Frontend sono stati effettuati dei test che permettessero di verificare il corretto rendering di tutti i componenti del sito. Ciò ha permesso quindi di verificare che le varie pagine rispondessero in modo adeguato a determinati eventi e soprattutto la reattività in base a determinate informazioni ricevute. Per la fase di testing lato Frontend sono stati utilizzati il framework Vitest ed il plugin messo a disposizione da Vue Test Utils.



File	% Stmts	% Branch	% Funcs	% Lines
All files	92.86	85.85	85.96	92.86
Checkers-Frontend	89.33	100	77.77	89.33
api.js	89.33	100	77.77	89.33
SideBar.vue	100	100	66.66	100
Checkers-Frontend/src/store	71.87	100	62.5	71.87
index.js	71.87	100	62.5	71.87
Checkers-Frontend/src/views	98.06	92.85	85.71	98.06
ErrorView.vue	100	100	100	100
Game.vue	87.57	77.77	90.9	87.57
Home.vue	99.44	95.65	80	99.44
LeaderBoard.vue	96.55	80	77.77	96.55
Lobbies.vue	100	100	100	100
LogIn.vue	100	100	87.5	100
Profile.vue	100	100	100	100
SignUp.vue	100	100	71.42	100

Figure 34: Coverage del Frontend

6.1.4 Licensing

La licenza scelta per Checkers si suddivide in due parti:

- Contenuto Web: per quanto riguarda il contenuto delle pagine la licenza scelta è **Creative Commons Attribution (CC BY)** che permette a chiunque di condividere, modificare (anche a fini commerciali) il materiale a condizione di menzionare l'autore ed eventuali modifiche effettuate. Unica eccezione a tale licenza sono i trademark, i quali non possono venire utilizzati senza permesso.

- Codice sorgente: per il codice sorgente la licenza scelta è **Apache License 2.0**, licenza non copyleft che obbliga gli utenti a preservare l'informativa di diritto d'autore e d'esclusione di responsabilità nelle versioni modificate.

6.1.5 Lint

Per garantire una maggiore qualità del codice sono state applicate delle restrizioni per la compilazione del codice mediante un plugin di linting: **eslint**. Tale plugin è stato impostato utilizzando la configurazione recommended dell'autore nonchè i seguenti plugin:

- Vue/base, vue3-essential, vue3-strongly-recommended e vue3-recommended: per garantire una corretta formattazione del codice Vue.
- prettier/recommended: Prettier è un formattatore di codice e rimuove tutto lo stile originale assicurandosi che tutto il codice emesso sia conforme ad uno stile coerente.
- babel/eslint-parser: ESLint consente l'uso di parser personalizzati. Quando si utilizza questo plug-in, il codice viene analizzato dal parser di Babel (utilizzando la configurazione specificata nel file di configurazione di Babel) e l'AST risultante viene trasformato in una struttura conforme a ESTree che ESLint può comprendere. Vengono conservate anche tutte le informazioni sulla posizione, come numeri di riga e colonne, in modo da poter rintracciare facilmente gli errori.
- airbnb-base: Plugin usato solo lato Backend che indica al file di linting di utilizzare le regole guida di Airbnb durante il controllo della formattazione.

7 Deployment Instructions

Come descritto in 6.1.2 il sistema viene deployato mediante un workflow su DigitalOcean pertanto è possibile utilizzare direttamente l'ultima versione rilasciata dall'indirizzo http://134.209.205.242:8080/.

In alternativa se si desidera eseguire il codice in locale è possibile farlo come descritto in seguito:

- Clonare la repo ed entrare nella relativa cartella mediante comandi

```
git clone https://github.com/LucaSalvigni/Checkers.git
cd Checkers
```

- Creare un file .env contenente tutti i secret
- Avviare il sistema mediante comando

```
docker compose up --build
```

- Accedere al sistema dall'indirizzo http://localhost:8080/.

8 Usage Examples

Per utilizzare il sistema è possibile accedervi dall'indirizzo http://134.209.205.242:8080/ o avviando il sistema come descritto in 7 da http://localhost:8080/. E' inoltre possibile contattare direttamente i microservizi cambiando la porta in:

- CommunicationService: 3030
- UserService: 3031
- GameService: 3032

Da notare che i servizi UserService e GameService richiedono un certificato per fornire una risposta.

8.1 Interazione col Frontend

8.1.1 Login e registrazione

Per poter accedere alle funzionalità del sistema è necessario effettuare il login o registrare un nuovo account, ciò è attuabile premendo il pulsante Sign in da una qualsiasi pagina del sistema ed inserendo le credenziali o premendo su Sign up e registrando un nuovo account.

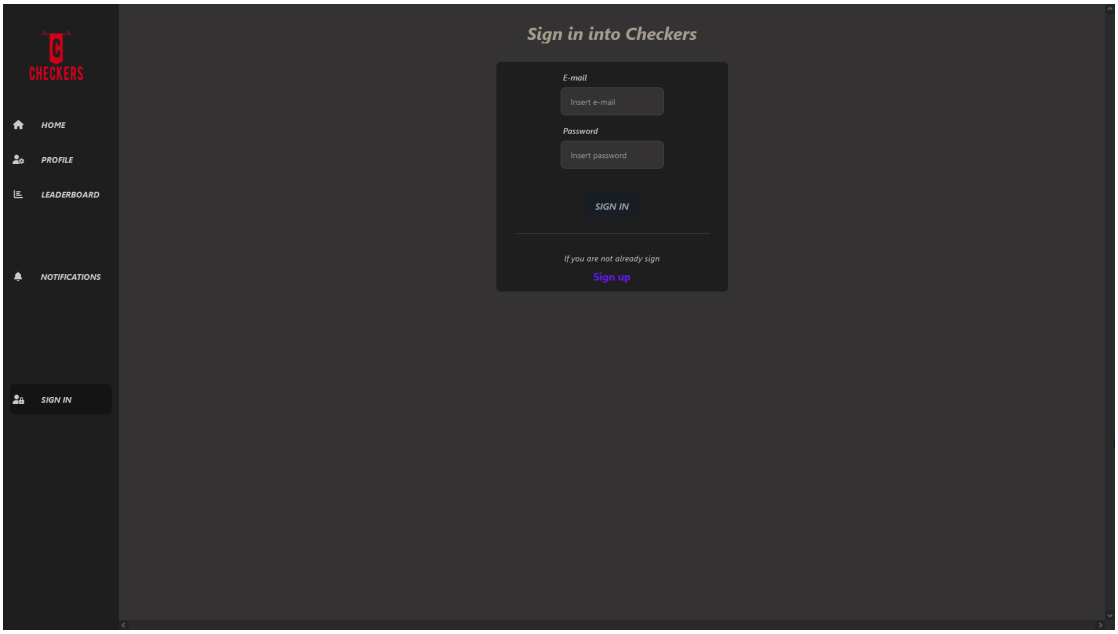


Figure 35: Sign In

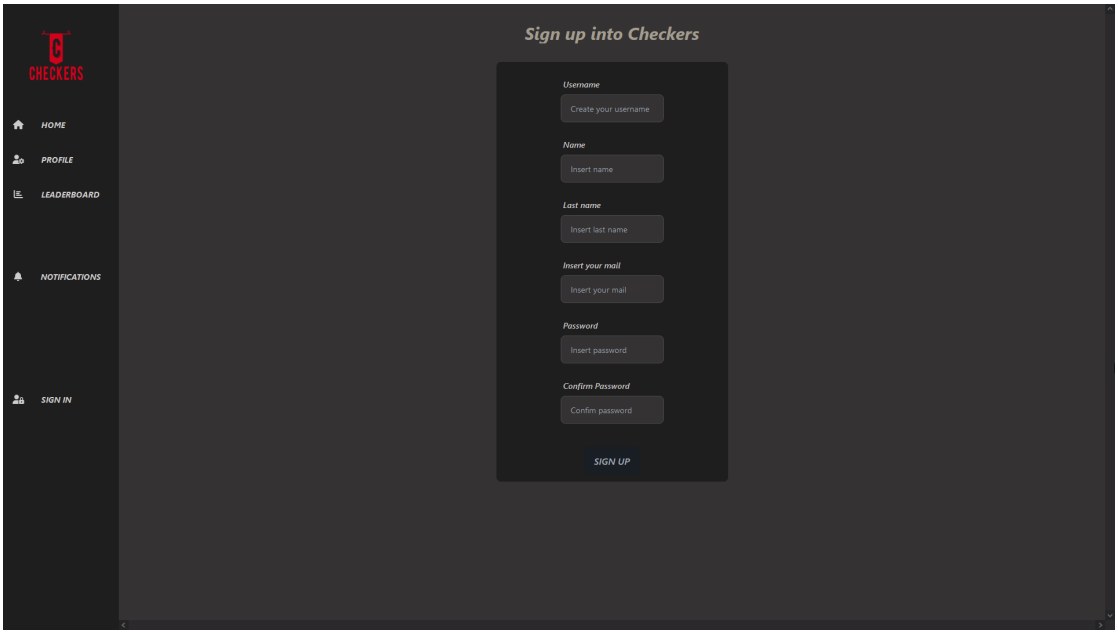


Figure 36: Sign Up

8.1.2 Profilo, cronologia e leaderboard

Una volta eseguito il login è possibile accedere alle informazioni sul proprio profilo premendo il pulsante ***Profilo*** dalla Sidebar. Da tale pagina è possibile visualizzare il proprio Elo, modificare i dati o visualizzare la cronologia delle partite.
Allo stesso modo è possibile visualizzare la leaderboard premendo sul relativo pulsante da Sidebar.

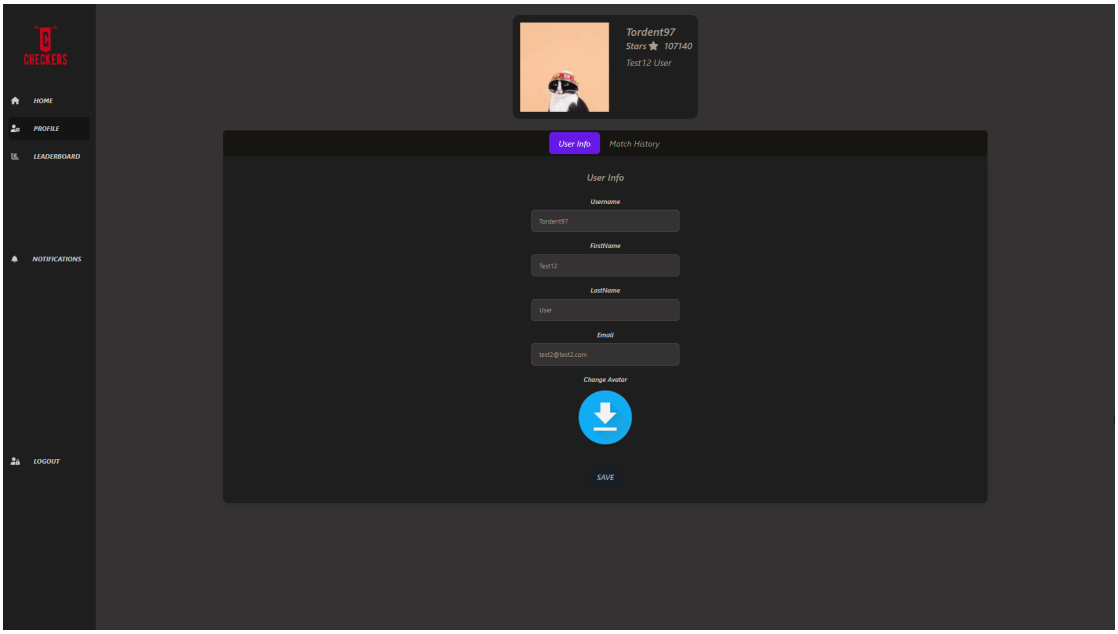


Figure 37: Profilo

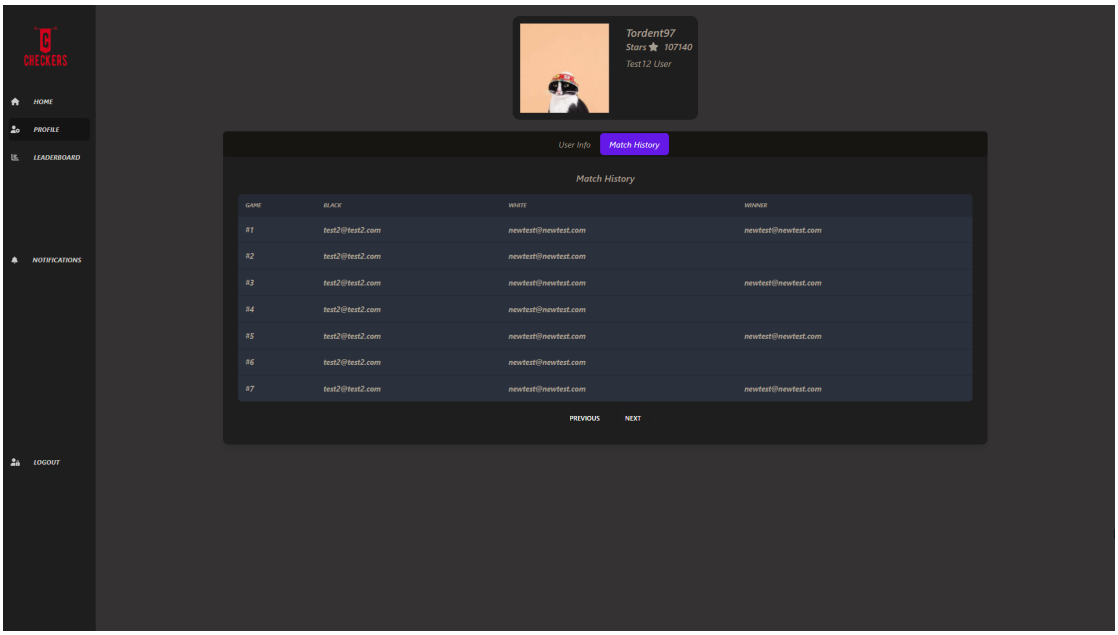


Figure 38: Cronologia partite

Figure 39: Leaderboard

8.1.3 Creazione o accesso a partita

Partendo dalla Home è possibile per un utente loggato creare una lobby (definendo l'Elo massimo), unirsi ad una lobby esistente o creare una partita privata invitando direttamente un altro utente. Se si è stati invitati in una lobby da un altro utente basterà premere sulla relativa notifica per accettare o rifiutare l'invito.

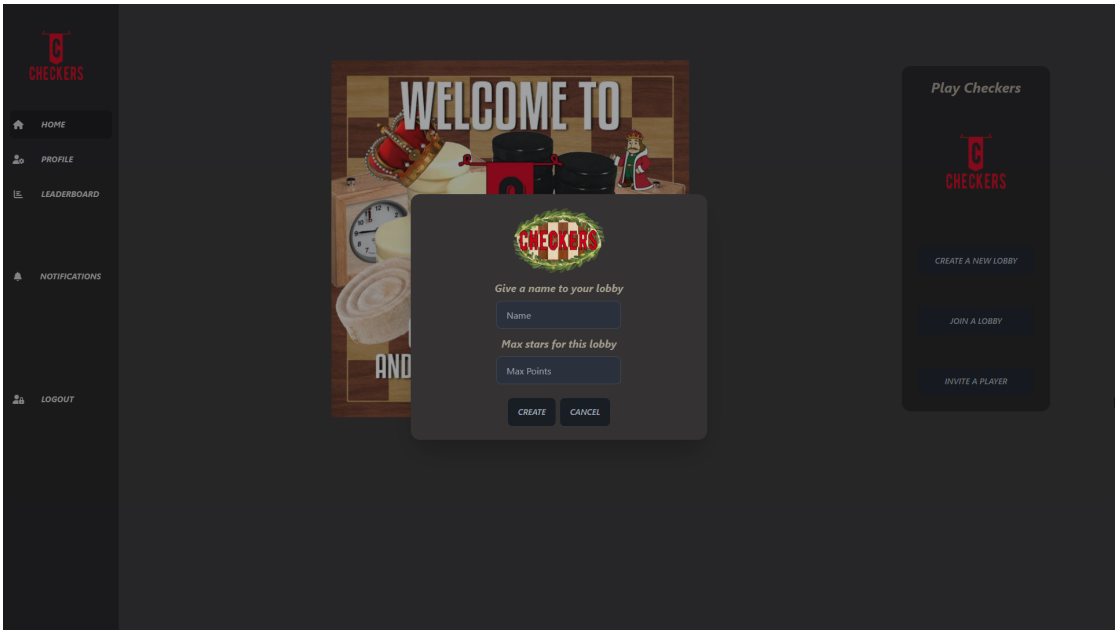


Figure 40: Creazione lobby

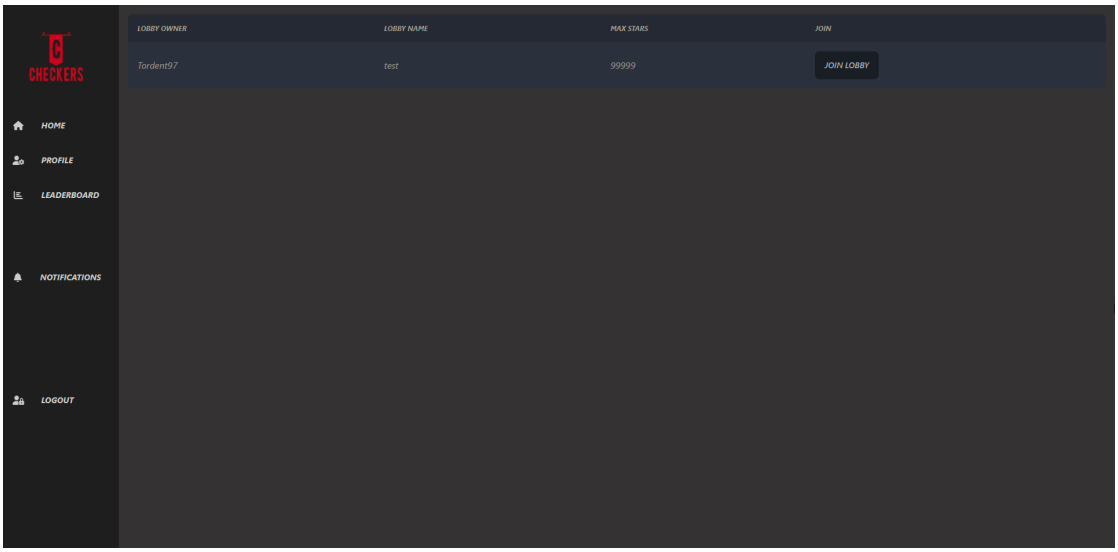


Figure 41: Join lobby

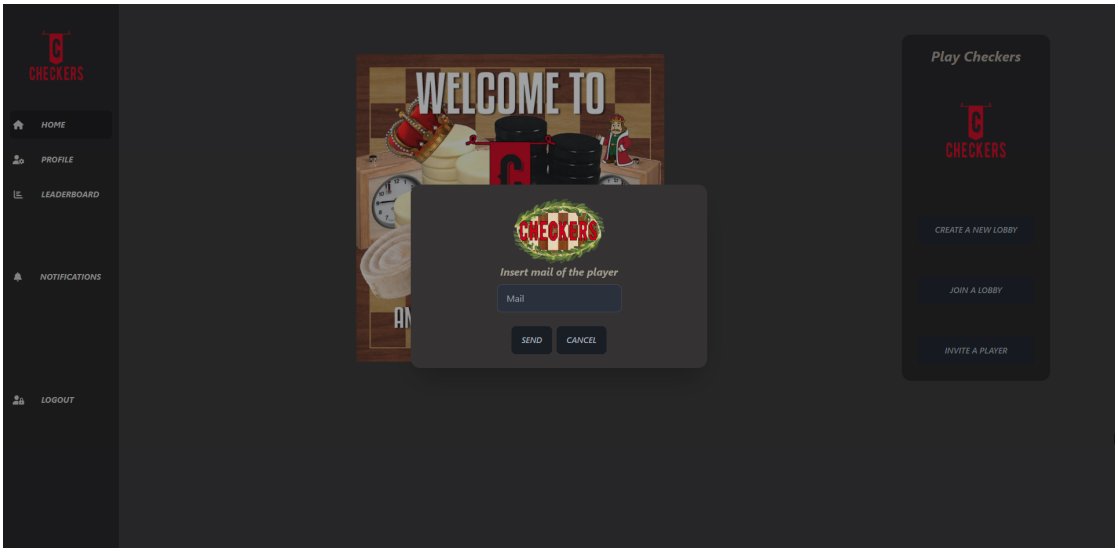


Figure 42: Creazione lobby privata con invito

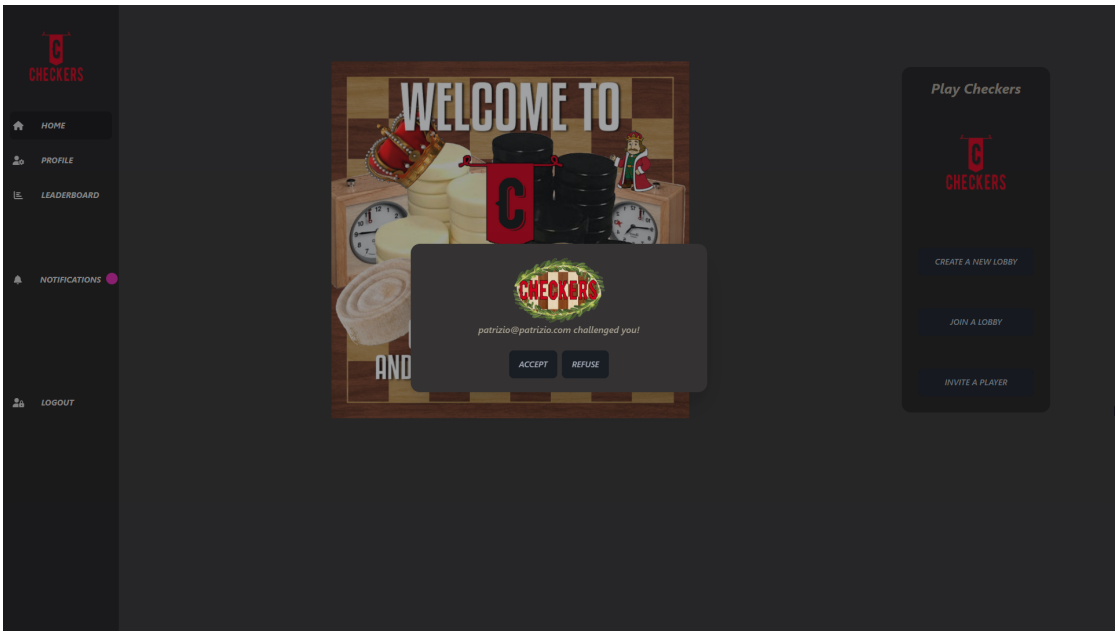


Figure 43: Notifica di invito

8.1.4 Partita

Una volta unitisi ad una lobby con un altro giocatore inizia la partita. I controlli sono relativamente semplici e consistono nel premere sul pezzo che si intende muovere per poi premere sulla nuova posizione desiderata. Vi è inoltre una chat mediante la quale conversare con il proprio avversario.

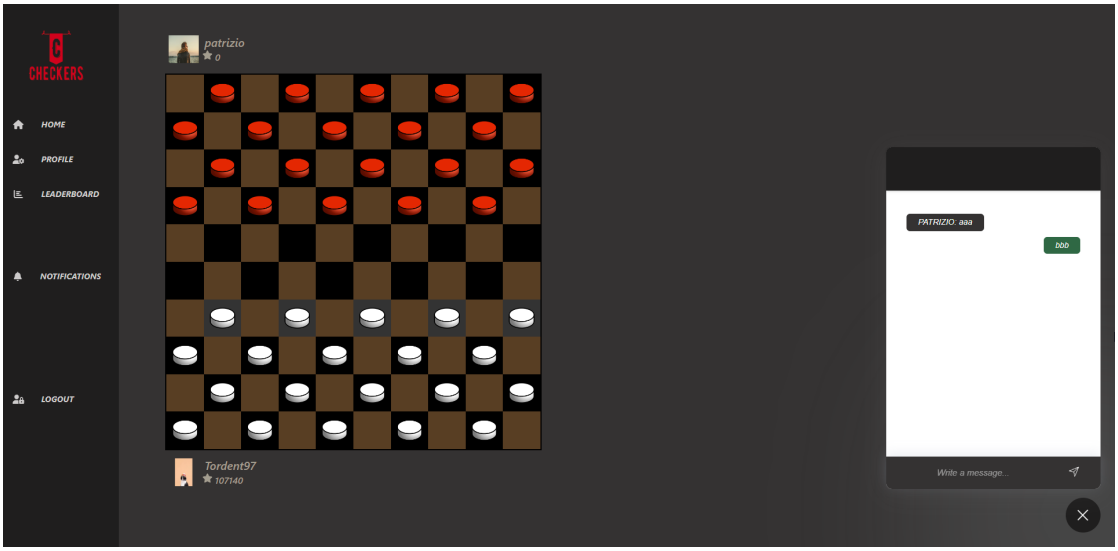


Figure 44: Partita

9 Conclusioni

Il progetto è stato realizzato in modo da poter raggiungere gli obiettivi fissati durante la fase di analisi. Arrivati alla conclusione, siamo soddisfatti del lavoro che è stato svolto riuscendo a soddisfare quasi tutti i requisiti, tranne la parte legata alla riconnessione ad una partita che non è stata realizzata per via di tempistiche. La realizzazione di un'applicazione web tramite l'utilizzo dello stack MEVN è risultato molto più intuitivo e semplice rispetto all'utilizzo di un approccio tramite XAMPP. La principale motivazione di ciò riguarda soprattutto il poter utilizzare il linguaggio Javascript sia per lo sviluppo del frontend sia per lo sviluppo del backend grazie a Node.js. Un grande pro dell'utilizzo di Node.js e del suo packet manager è la grandezza della community dietro che mette a disposizione un numero illimitato di librerie per quasi ogni esigenza, facilitando lo sviluppo e permettendo di raggiungere standard di qualità che richiederebbero molto più lavoro. Unico contro la curva di apprendimento del linguaggio e della metodologia di sviluppo che ha richiesto inizialmente una discreta quantità di tempo, recuperato successivamente tramite la fluidità dell'approccio. Infine ci sono state alcune difficoltà in fase di sviluppo dovute al fatto della scelta di utilizzare Vue 3 in quanto la documentazione ed in generale il supporto all'interno della community è risultato più limitato rispetto a Vue 2.

9.1 Sviluppi futuri

Il lavoro svolto finora ci ha permesso di attuare una trasposizione di Checkers che, dal punto di vista delle funzionalità, è identico alla versione da tavolo della dama internazionale. Pertanto, in una prospettiva futura, gran parte degli aspetti su cui si basa il progetto possono evolvere come per esempio: aggiornare l'infrastruttura, fare affidamento sul livello di personalizzazione data dall'utente, mettere a disposizione nuove funzionalità. Sarebbe inoltre importante in una versione futura andare ad implementare ulteriori meccanismi di fault-tolerance che possano rendere il più trasparenti possibili eventuali interruzioni del servizio ai clienti. Alcuni tra i possibili lavori futuri sono:

- Aggiungere la possibilità ad un giocatore disconnesso per vari motivi di poter riconnettersi ad una lobby entro un tempo limite.
- aggiungere la possibilità di recuperare le partite in corso perse durante un'interruzione del servizio (il sistema si presta già a questa funzionalità visto il salvataggio di ogni partita sul DB etichettate come "terminata" ed "in corso").
- aggiungere un meccanismo di fault-tolerance che permetta al sistema di intercettare un eventuale interruzione di uno dei servizi e mettere in stallo l'operatività del sistema (es: un meccanismo simile a quello degli heartbeat utilizzato da Hadoop)
- Garantire più libertà di personalizzazione all'utente, per esempio cambiare colore delle pedine, cambiare colore della board di gioco o cambiare il tema dell'applicazione. Modifiche che non impattano le dinamiche di gioco ma che possono garantire una user experience migliore.

- Pubblicazione del gioco online;
- Aggiunta di nuove modalità particolari di gioco legate alla Dama come per esempio `Trapdoor Checkers`

9.2 Cosa abbiamo imparato

Questo progetto ha notevolmente arricchito le nostre conoscenze in alcune aree. Abbiamo imparato a sviluppare applicazioni web utilizzando un'architettura a microservizi, ottenendo sviluppo e routine più efficienti, potendo analizzare punti di forza, criticità ed esigenze di questo approccio. La possibilità di sviluppare più microservizi in contemporanea ci ha permesso inoltre di lavorare simultaneamente, riducendo le tempistiche di sviluppo. Questa esperienza sarà sicuramente utile per dei lavori futuri in quanto questo modello sta diventando sempre più popolare tra gli sviluppatori e tra i più utilizzati per lo sviluppo di sistemi distribuiti.

In questo progetto l'utilizzo di Docker nonchè di Github Actions si è rivelato una parte fondamentale ed è stato sfruttato in un modo molto più complesso rispetto alle esperienze che abbiamo avuto. Grazie a questo, è stato possibile comprendere più in dettaglio molte funzionalità che saranno poi utili in progetti futuri.