Documentazione Belmopoly

# Glossario

1 Glossario 2

2 Introduzione 3

2.1 Informazioni sul progetto 3

2.2 Abstract 3

2.3 Scopo 3

3 Analisi 4

3.1 Analisi del dominio 4

3.2 Analisi e specifica dei requisiti 4

3.2.1 Spiegazione elementi tabella dei requisiti: 10

3.3 Use Case 11

3.3.1 Use Case Main 11

3.3.2 Use Case Gioco 12

3.4 Pianificazione 13

3.4.1 Progettazione 14

3.4.2 Implementazione 14

3.5 Analisi dei mezzi 15

3.5.1 Software 15

3.5.2 Hardware 15

4 Progettazione 16

4.1 Design dell’architettura del sistema 16

4.2 Design delle interfacce 16

4.2.1 Interfaccia Login/Sign Up 16

4.2.2 Interfaccia Main 17

4.2.3 Interfaccia Room 18

4.2.4 Interfaccia Friends 18

4.2.5 Interfaccia User 19

4.2.6 Interfaccia Notifications 19

4.2.7 Interface Character 20

4.2.8 Interfaccia Game 20

4.3 Design procedurale 21

4.3.1 Diagrammi di flusso 21

4.3.2 Diagramma ER 21

5 Implementazione 22

6 Test 22

6.1 Protocollo di test 22

6.2 Risultati test 23

6.3 Mancanze/limitazioni conosciute 23

7 Consuntivo 23

8 Conclusioni 23

8.1 Sviluppi futuri 23

8.2 Considerazioni personali 23

9 Glossario 23

10 Bibliografia 24

10.1 Bibliografia per articoli di riviste: 24

10.2 Bibliografia per libri 24

10.3 Sitografia 24

11 Allegati 24

# Introduzione

## Informazioni sul progetto

* **Allievo:** Alex Gilardi
* **Allievo:** Alan Gregorio
* **Allievo:** Mark Iljins
* **Docente:** Michel Palucci
* **Scuola e classe:** SAMT Sezione informatica, I3AA
* **Materia:** Modulo 306
* **Data inizio consegna:** 29.01.2025 – 28.05.2025

## Abstract

## Scopo

Lo scopo principale del progetto è quello di realizzare un’applicazione WEB che permette agli utenti che lo utilizzano di divertirsi e giocare al famoso gioco Monopoly. Ovviamente per rendere il nostro applicativo migliore oltre ad avere le funzionalità base del gioco, che tutti conoscono, abbiamo anche aggiunto delle funzionalità extra.

Un ulteriore scopo del progetto è sicuramente quello di migliorare le nostre competenze nei vari linguaggi che andremo a utilizzare. Infatti, questo progetto ci permetterà di andare a esplorare ulteriormente la creazione e la struttura di DataBase tramite il linguaggio SQL. Ci permetterà anche di migliorare le nostre abilità nei linguaggi WEB come PHP, Javascript, HTML.

Infine, essendo il nostro primo progetto a gruppi, ci permetterà di applicare e migliorare quello che abbiamo acquisito durante lo svolgimento del primo progetto, svolto nel primo semestre.

# Analisi

## Analisi del dominio

Belmopoly è un applicativo WEB user-friendly molto facile da utilizzare, infatti possiede un design molto semplice e intuitivo, e non necessita di nessun requisito per essere utilizzato. L’applicativo è pensato per tutti gli utenti che vogliono divertirsi in compagnia giocando a un gioco da tavola famosissimo, cioè Monopoly, ma rimanendo online.

Attualmente ci sono già abbastanza prototipi di Monopoly Online in circolazione, però di quelli presenti quasi tutti hanno dei difetti abbastanza significativi, che rovinano l’esperienza di gioco all’utente.

Oltre alle funzionalità base, gli utenti avranno anche la possibilità di, personalizzare la propria pedina (Testa, Busto, Gambe), di chattare con gli altri giocatori durante partita e infine la possibilità di mettere all’asta una proprietà, dove tutti i giocatori possono offrire la propria somma di denaro.

Grazie a queste funzionalità aggiuntive, rende il nostro applicativo unico, e migliore da giocare in quanto tutti quelli già presenti posseggono solo le funzionalità base.

## Analisi e specifica dei requisiti

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-01 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Login Utente | | | | |
| **Note** | L’utente potrà effettuare l’accesso al suo account dell’applicativo. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Textbox per Password | | | | |
| **002** | Textbox per Email | | | | |
| **003** | Tasto Sign Up | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-02 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Sign Up Utente | | | | |
| **Note** | L’utente se non possiede già un account potrà crearne uno da questa interfaccia. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Textbox per Username | | | | |
| **002** | Textbox per Email | | | | |
| **003** | Textbox per Password | | | | |
| **004** | Tasto Login | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-03 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Main | | | | |
| **Note** | L’utente potrà visualizzare una sezione per creare una room, continuare una partita, e visualizzare una lista di amici + gli inviti | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Tasto per creare la Room | | | | |
| **002** | Tasto per continuare una partita | | | | |
| **003** | Tasto dove visualizzare amici e inviti | | | | |
| **004** | Tasto per logout | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-04 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Creazione Room | | | | |
| **Note** | L’utente potrà visualizzare una pagina dove poter creare una nuova room privata, dove poter invitare a giocare i vari amici. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | TextBox per il nome della room | | | | |
| **002** | Tasto per avviare la partita | | | | |
| **003** | Tasto per tornare alla MainPage4 | | | | |
| **004** | 4 Tasti per aggiungere i vari amici (basta anche solo 2 giocatori) | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-05 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia con le Room | | | | |
| **Note** | L’utente potrà visualizzare una pagina dove poter continuare la partita che vuole. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Zona con tutte le partite da continuare | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-06 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Gioco | | | | |
| **Note** | L’utente potrà visualizzare tutti gli elementi necessari per giocare al Monopoli | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Zona a destra riguardante i vari player con il loro patrimonio e proprietà | | | | |
| **002** | Zona al centro dell’interfaccia la tavola di gioco con tutte le caselle. | | | | |
| **003** | Pulsante per il lancio dei dadi. | | | | |
| **004** | Pulsante per aprire la chat. | | | | |
| **005** | Pulsante per richiedere un trade. | | | | |
| **006** | Pulsante per richiedere di uscire dalla partita. | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-07 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Interfaccia Popup Casella | | | | |
| **Note** | L’utente quando finisce su una casella, essa verrà mostrata all’utente in mezzo all’interfaccia di gioco modi popup. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Pulsante Compra | | | | |
| **002** | Pulsante Asta | | | | |
| **003** | Zona con le informazioni riguardanti la proprietà | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-08 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Login | | | | |
| **Note** | L’utente può effettuare il login | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione dell’utente. | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-09 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Logout | | | | |
| **Note** | L’utente può effettuare il logout | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione dell’utente. | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-10 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Creazione Account | | | | |
| **Note** | L’utente può creare un nuovo utente se già non ne possiede uno | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione nome utente | | | | |
| **002** | Gestione email | | | | |
| **003** | Gestione password | | | | |
| **004** | Gestione DB | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-11 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Eliminazione Account | | | | |
| **Note** | L’utente può eliminare il proprio account | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione DB | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-12 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Soldi | | | | |
| **Note** | L’utente possiede un conto con un certo quantitativo di soldi con cui può comprare le varie proprietà o pagare gli altri in caso cada su una casella non sua. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione soldi sufficienti | | | | |
| **002** | Gestione soldi no negativi | | | | |
| **003** | Gestione + 200 ogni giro | | | | |
| **004** | Gestione Bancarotta | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-13 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Tiro Dadi | | | | |
| **Note** | L’utente quando è il suo turno può tirare i dadi | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione lancio | | | | |
| **002** | Zona dove ci sono i dadi | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-14 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Proprietà | | | | |
| **Note** | L’utente può comprare nuove proprietà, possederle e scambiare | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione dell’acquisto della proprietà | | | | |
| **002** | Gestione del possesso della proprietà | | | | |
| **003** | Gestione per lo scambio di proprietà | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-15 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Prigione | | | | |
| **Note** | L’utente quando cade sulla cella “Go To Jail”, viene automaticamente portato alla cella della prigione senza passare per il via. Se possiede una carta “Jail Free” può uscire di prigione gratis. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione cella “Go To Jail” | | | | |
| **002** | Gestione turni prigione | | | | |
| **003** | Gestione uscire di prigione | | | | |
| **004** | Gestione carta “Jail Free” | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-16 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Carte Eventi | | | | |
| **Note** | L’utente se cade su Imprevisti o Probabilità, gli verrà mostrata a schermo una carta di quel tipo. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione carte imprevisti | | | | |
| **002** | Gestione carte probabilità | | | | |
| **003** | Gestione evento da eseguire | | | | |
| **004** | Gestione carte inventario | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-17 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Lista Amici | | | | |
| **Note** | L’utente può mandare una richiesta di amicizia a un altro utente e può anche rifiutare eventuali richieste. Può anche rimuovere dalla lista un amico. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione invio richiesta amicizia | | | | |
| **002** | Gestione rifiuto richiesta amicizia | | | | |
| **003** | Gestione rimuovi amico | | | | |
| **004** | Gestione apri lista | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-18 | **Priorità** | 1 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Invita Amico | | | | |
| **Note** | L’utente mentre crea una partita può invitare a giocare i propri amici che ha nella lista amici. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione invito a giocare | | | | |
| **002** | Gestione accetta invito a giocare | | | | |
| **003** | Gestione rifiuto invito a giocare | | | | |
| **004** | Zona nella pagina di creazione room dove poter mandare l’invito | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-19 | **Priorità** | 2 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Asta | | | | |
| **Note** | L’utente quando cade su una nuova proprietà, se decide di non comprarla, essa verrà messa all’asta, dove tutti i giocatori possono offrire la propria somma per comprarla. | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione blocca partita | | | | |
| **002** | Gestione offerte player | | | | |
| **003** | Gestione timer offerta maggiore | | | | |
| **004** | Gestione aumento offerta | | | | |
| **005** | Gestione vincita asta | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-20 | **Priorità** | 3 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Chat Istantanea | | | | |
| **Note** | L’utente durante la partita può scriversi con gli altri giocatori in modo istantaneo nella chat situata in basso a destra | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione invio messaggio | | | | |
| **002** | Gestione ricezioni messaggi | | | | |
| **003** | TextBox per scrivere il messaggio | | | | |
| **004** | Gestione apri chat | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requisito** | REQ-21 | **Priorità** | 3 | **Versione** | 1.0 |
| **Nome** | Algoritmo Gestione Pedina Personalizzata | | | | |
| **Note** | L’utente può personalizzare la propria pedina prima di iniziare a giocare (Testa, Busto, Gambe) | | | | |
| **Sotto requisiti** | | | | | |
| **001** | Gestione selezione testa | | | | |
| **002** | Gestione selezione busto | | | | |
| **003** | Gestione selezione gambe | | | | |
| **004** | Salvataggio personaggio | | | | |

### Spiegazione elementi tabella dei requisiti:

**ID**: Identificativo univoco del requisito

**Nome**: Breve descrizione del requisito

**Priorità**: Indica l’importanza di un requisito nell’insieme del progetto, definita assieme al

committente. Ad esempio, poter regolare la qualità audio ha priorità minore rispetto al fatto di poter convertire il formato del file audio. Solitamente si definiscono al massimo di 2-3 livelli di priorità.

**Versione**: Indica la versione del requisito. Ogni modifica del requisito avrà una versione aggiornata.

**Note**: Eventuali osservazioni importanti o riferimenti ad altri requisiti.

**Sotto requisiti**: Elementi che compongono il requisito.

## Use Case

### Use Case Main

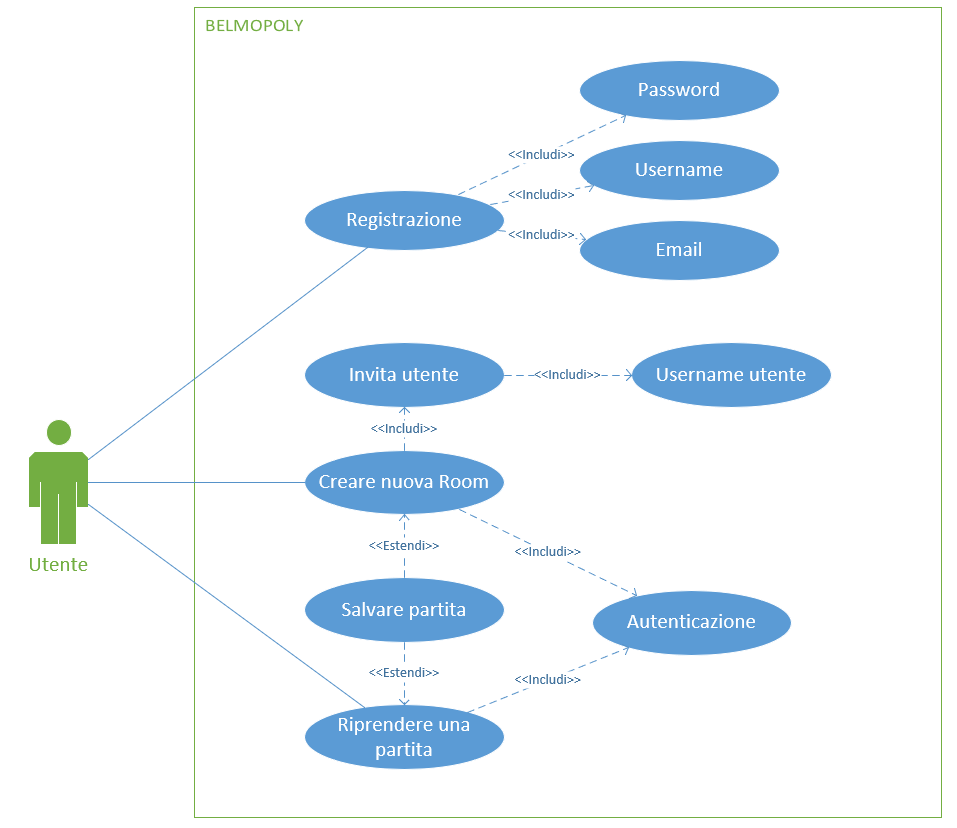


Figura 1 - Use Case Main

Questo è lo Use Case Main dell’applicativo WEB Belmpoly. Esso descrive il funzionante dell’applicazione senza la parte di gioco. Come si vede c’è solo un attore che utilizza l’applicativo WEB:

1. **Utente**: L’utente potrà svolgere principalmente tre azioni. La prima di tutte che andrà a fare se non possiede già un account nell’applicazione, è la registrazione. Ovviamente questa operazione non può essere completata se non si inserisce password, username e email.

Mentre la seconda azione che può svolgere, è quella di poter creare una nuova Room (partita). Ovviamente la Room non può essere creata se prima non si fa l’autenticazione, e inoltre bisogna per forza invitare almeno un giocatore, altrimenti non si può procedere con la creazione. L’invito del giocatore sarà svolto tramite username. In più l’utente ha anche la possibilità di salare la partita, esso non è obbligatorio.

L’ultima azione che può svolgere è quella di riprendere una partita già cominciata, come per la creazione, prima bisogna svolgere l’autenticazione, altrimenti non si può riprendere. Anche qui c’è la possibilità di salvare la partita, non obbligatorio.

### Use Case Gioco

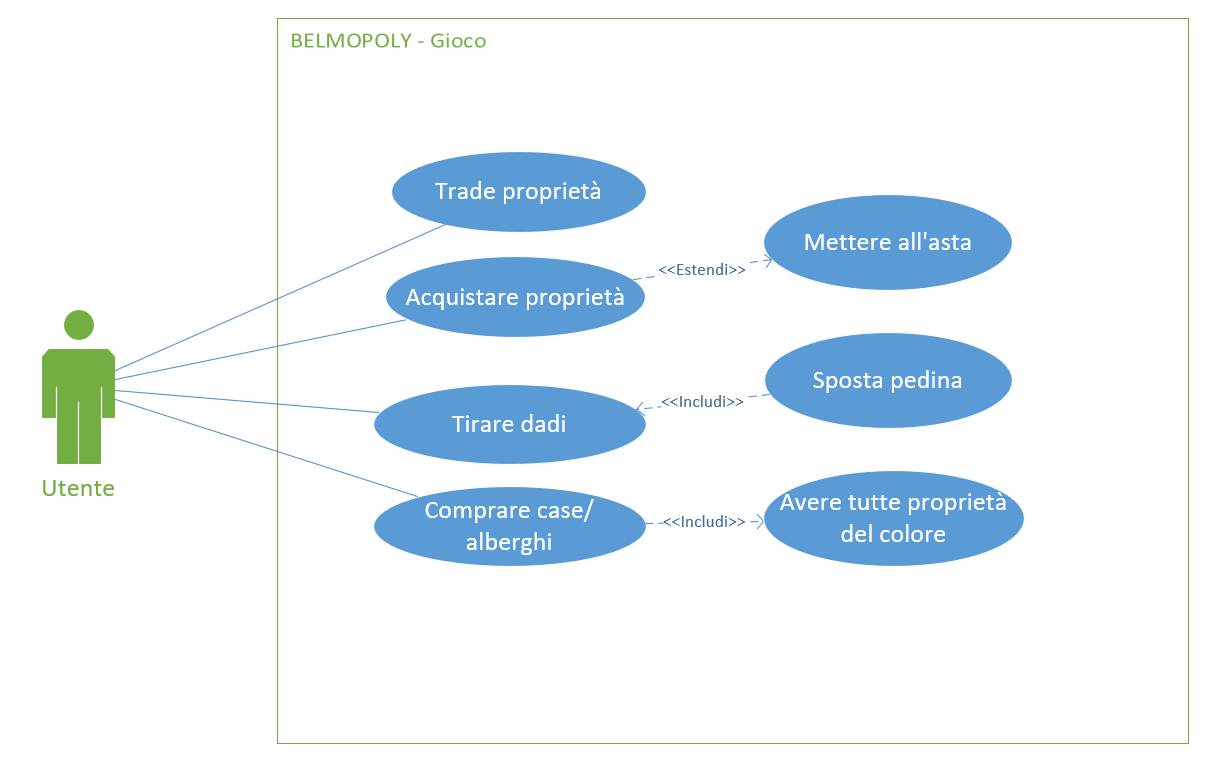


Figura 2 - Use Case Gioco

Questo è lo Use Case Gioco dell’applicativo WEB Belmpoly. Esso descrive la parte di gioco dell’applicazione. Come prima c’è solo un attore che utilizza l’applicativo WEB:

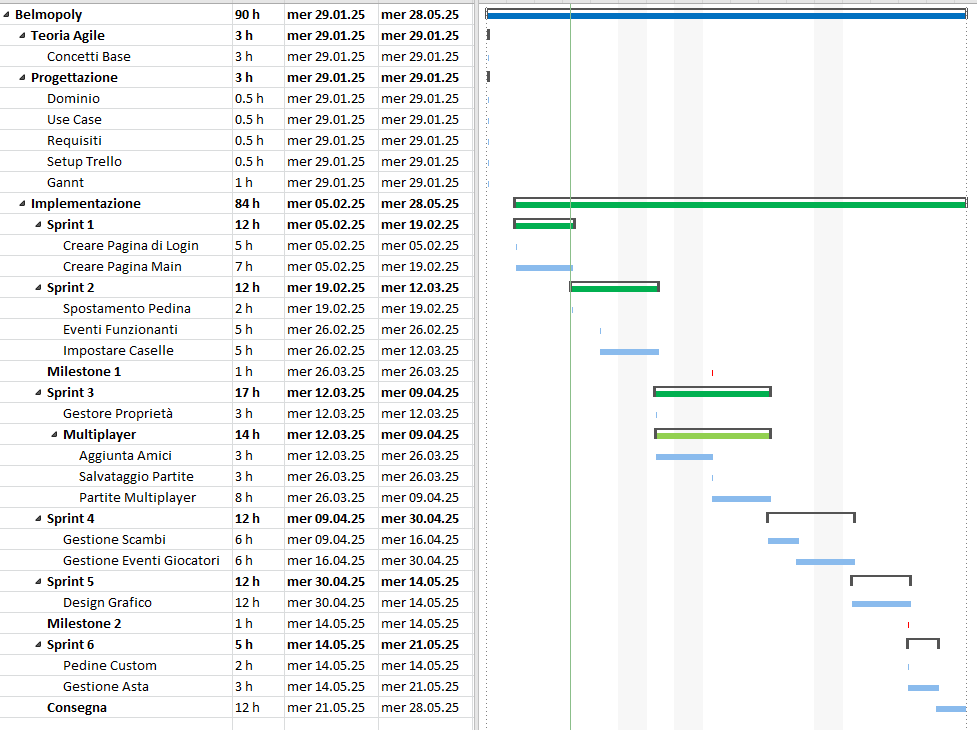
1. **Utente**: L’utente in questo caso può svolgere 4 azioni principali. La prima è quella di poter tirare i dadi, che automaticamente innesca l’attività secondaria cioè quella del movimento della pedina.

La seconda azione che può fare è quella di comprare nuove proprietà, questa azione viene estesa dal poter mettere all’asta la proprietà se si decide di non comprarla.

La terza attività che può andare a svolgere è il tiro dei due dadi, essa ovviamente include l’azione dello spostamento della pedina.

Mentre l’ultima azione che può andare a fare è quella di comprare case o alberghi per la proprietà in questione, ovviamente questa azione deve essere inclusa dal possedere tutte le proprietà dello stesso colore, altrimenti non può comprare nessuna delle due.

## Pianificazione



### Progettazione

La fase di progettazione è la fase più importante a inizio progetto, essa comprende tutte le attività necessarie per poi avviare la fase di implementazione. All’interno di questa fase abbiamo inserito 5 attività principali, e abbiamo previsto che ci porteranno via in totale 3 ore, cioè mezza giornata.   
In questa fase andremo a definire tutti i requisiti principali con la loro priorità, inoltre andremo a settare il setup di Trello[[1]](#footnote-1) con tutte le varie task da svolgere. Altre due attività che realizzeremo sono il Gannt stesso e lo Use Case dell’applicativo WEB. In fine andremo a realizzare tutti i diagrammi ER e di flusso.

### Implementazione

La fase di implementazione è la fase principale, infatti i porterà via all’incirca 84 ore, è quella che contiene tutte le attività per la realizzazione effettiva dell’applicativo WEB. Questa fase è suddivisa in 6 Sprint, che a loro volta al loro interno hanno attività differenti. In questa fase troviamo anche 2 Milestone cioè due punti importanti e fondamentali per il progetto.

#### Sprint 1

Lo sprint 1 sarà della durata di 12 ore, in questa fase andremo a creare la pagina di autenticazione e login, con già tutti i vari controlli e il salvataggio dei dati su DB. Inoltre andremo a creare la pagina principale dell’applicativo più la tavola di gioco.

#### Sprint 2

Lo sprint 2 è anche lui della durata di circa 12 ore, in questa fase andremo a completare le attività che non abbiamo finito nello sprint1, e che reputiamo necessarie finirle. In più andremo a svolgere le prime attività importanti per il funzionamento dell’applicativo, cioè lo spostamento della pedina all’interno della tavola, un piccolo evento per quando si capita su una cella, e in fine completare definitivamente la tavola da gioco con tutti i nomi e prezzi.

#### Sprint 3

Lo sprint 3 è leggermente più lungo degl’altri tre, dalla durata di circa 17 ore. Qui andremo a sviluppare l’algoritmo per la gestione delle proprietà, cioè tutte quelle funzionalità che permetto di comprare le proprietà, e di far pagare gli altri giocatori qualvolta ci finiscano sopra. In più andremo a sviluppare il multiplayer, cioè l’aggiunta di amici all’interno della partita e in fine il salvataggio della partita stessa su DB.

#### Sprint 4

Lo sprint 4 durerà all’incirca 12 ore, in questo tempo andremo a svolgere due attività molto importanti. La prima attività sarà quella di realizzare tutti gli algoritmi che permetteranno gli scambi di proprietà tra giocatori. Mentre la seconda cosa che andremo a fare saranno tutti gli algoritmi riguardante i vari eventi che può svolgere il giocatore, come il lancio dei dadi, la visualizzazione delle proprie proprietà ecc.

#### Sprint 5

Lo sprint 5 è quello che contiene meno attività, esso durerà sempre circa 12 ore. In questa fase andremo a migliorare e finire definitivamente il design dell’applicativo WEB. Esso verrà sviluppato basandosi sullo stile “Cyberpunk”, infatti ogni nome proprietà ha un nome riguardante questo tema.

#### Sprint 6

Lo sprint 6 è l’ultimo che andremo a svolgere, infatti sarò quello meno importante, in quanto contiene due attività di proprietà 3. In questa fase, se riusciremo con le tempistiche, andremo a implementare gli algoritmi per la personalizzazione della pedina e gli algoritmi per la gestione delle aste.

## Analisi dei mezzi

Per la realizzazione dell’applicativo WEB Belmopoly abbiamo utilizzato i PC fornito dalla Scuola Arti e Mestieri di Trevano con i software già installati al suo interno.

### Software

Durante lo sviluppo sono stati impiegati i seguenti strumenti software e librerie:

* **Node.js**: ambiente di esecuzione JavaScript lato server, utilizzato per la gestione del backend e degli script di automazione.
* **PHP**: linguaggio server-side utilizzato per la logica di business e l’interazione con il database.
* **JavaScript**: linguaggio principale per la gestione della parte frontend, con codice eseguito nei browser.
* **MySQL:** sistema di gestione di database relazionali per la memorizzazione e gestione dei dati persistenti.

### Hardware

Il prodotto Belmopoly è progettato per essere eseguito su una piattaforma virtuale (VM) host, configurata su hardware fornito dalla scuola. Questa soluzione consente una gestione flessibile dell’ambiente di sviluppo e produzione, isolando l’applicazione e facilitandone la distribuzione.

* **Piattaforma**: macchina virtuale host Linux.
* **Particolarità e limitazioni**: l’utilizzo di una VM permette di replicare ambienti di produzione, ma può introdurre limitazioni legate alle risorse assegnate (CPU, RAM, storage).

# Progettazione

## Design dell’architettura del sistema

Descrive:

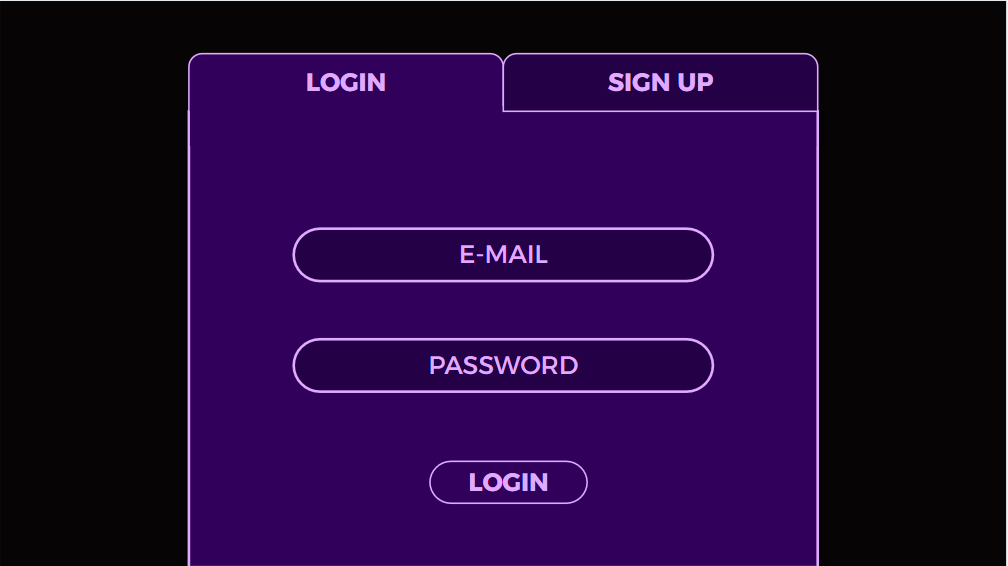
* La struttura del programma/sistema lo schema di rete...
* Gli oggetti/moduli/componenti che lo compongono.
* I flussi di informazione in ingresso ed in uscita e le relative elaborazioni. Può utilizzare *diagrammi di flusso dei dati* (DFD).
* Eventuale sitemap

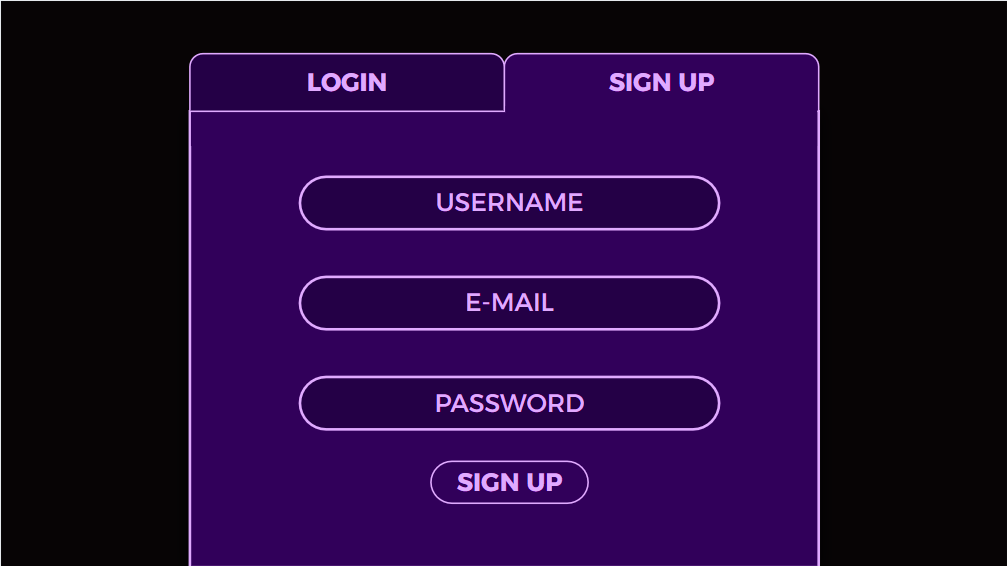
## Design delle interfacce

La progettazione delle interfacce è stata realizzata basandosi sulle informazioni ricavate durante la fase di analisi e le abbiamo progettate tramite il sito moqups[[2]](#footnote-2) .

Tutte le interfacce sono state pensate per essere utilizzate su Browser Web.

### Interfaccia Login/Sign Up

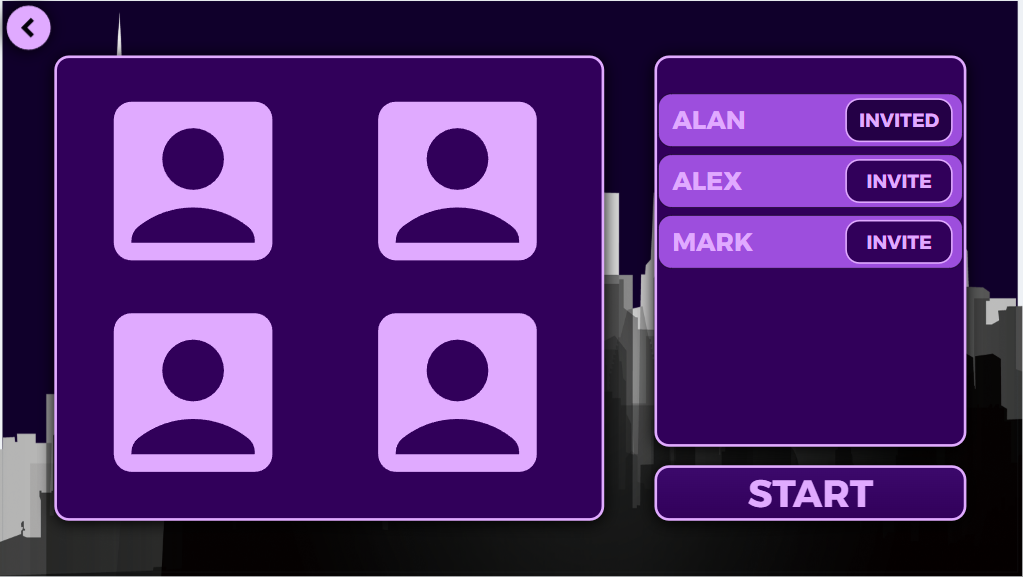




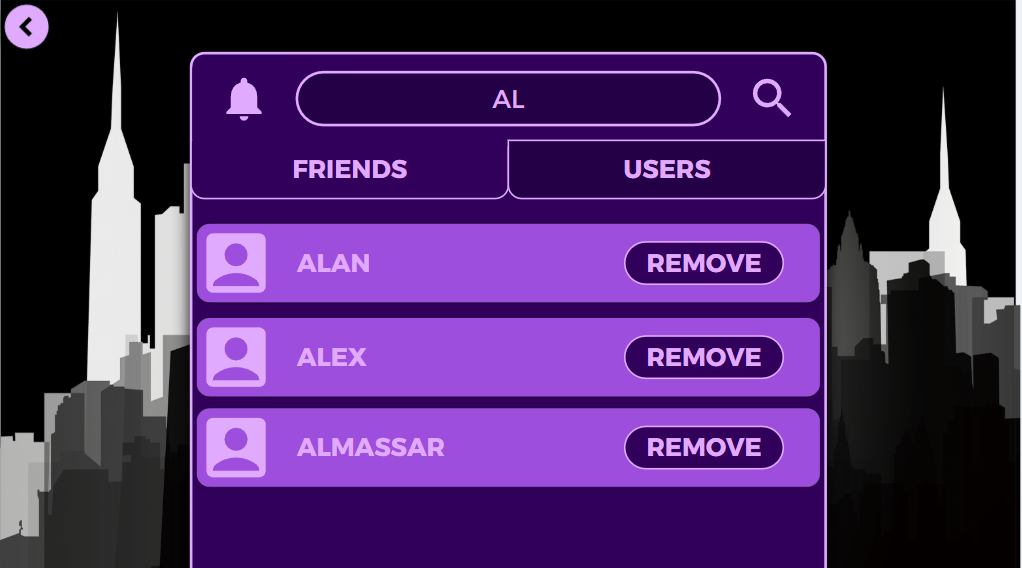
### Interfaccia Main



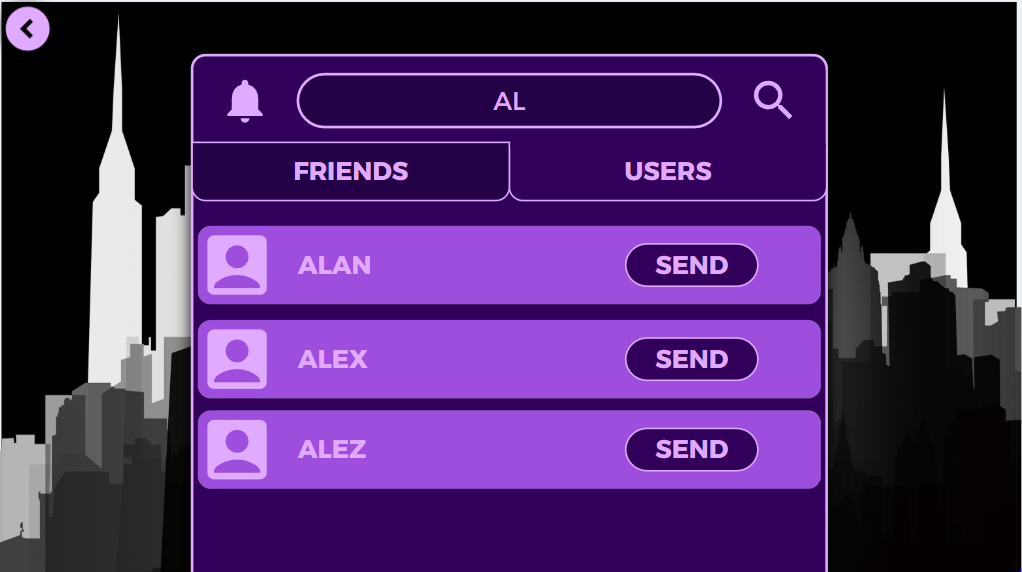
### Interfaccia Room



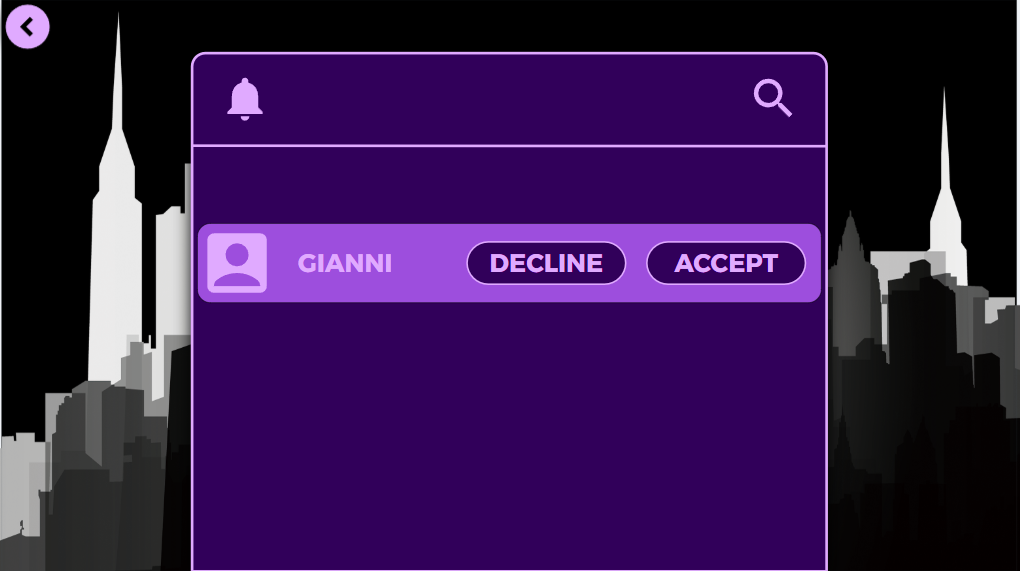
### Interfaccia Friends



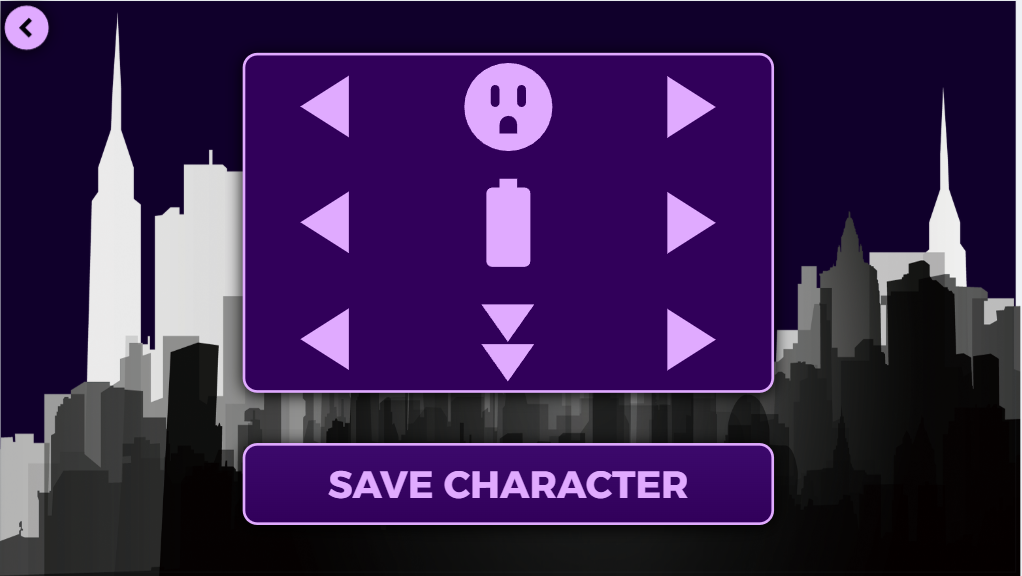
### Interfaccia User



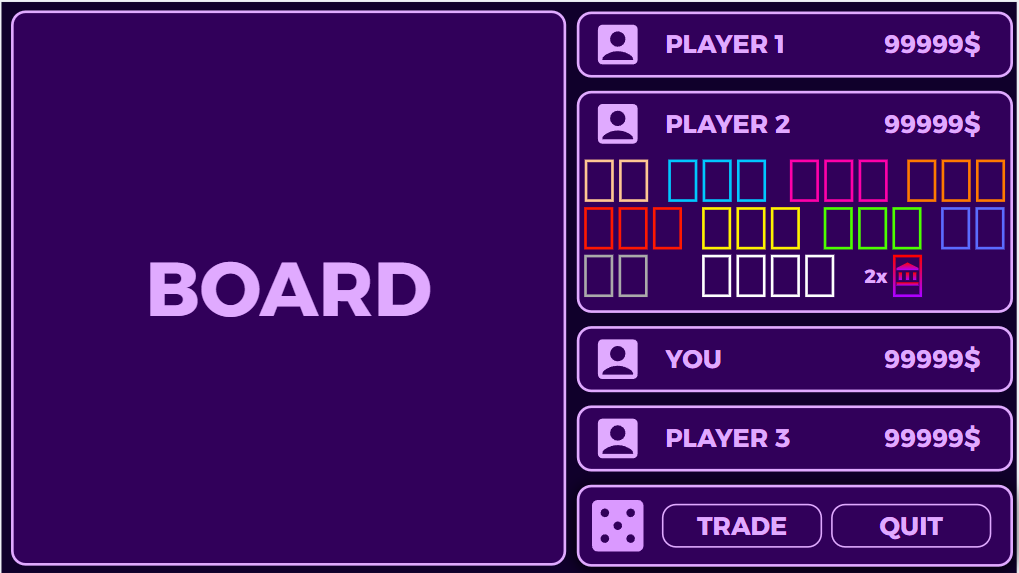
### Interfaccia Notifications



### Interface Character



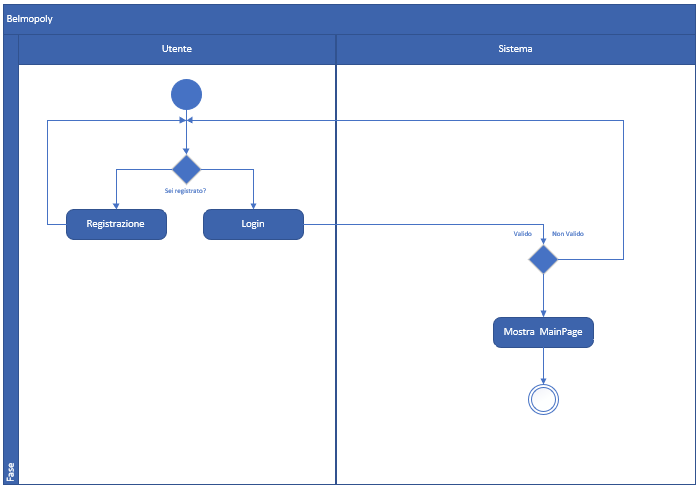
### Interfaccia Game



## Design procedurale

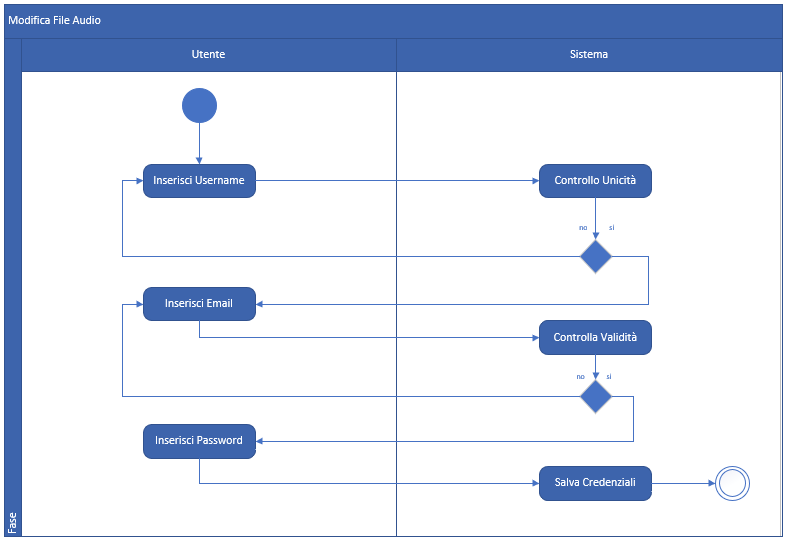
### Activity Diagram

#### Login



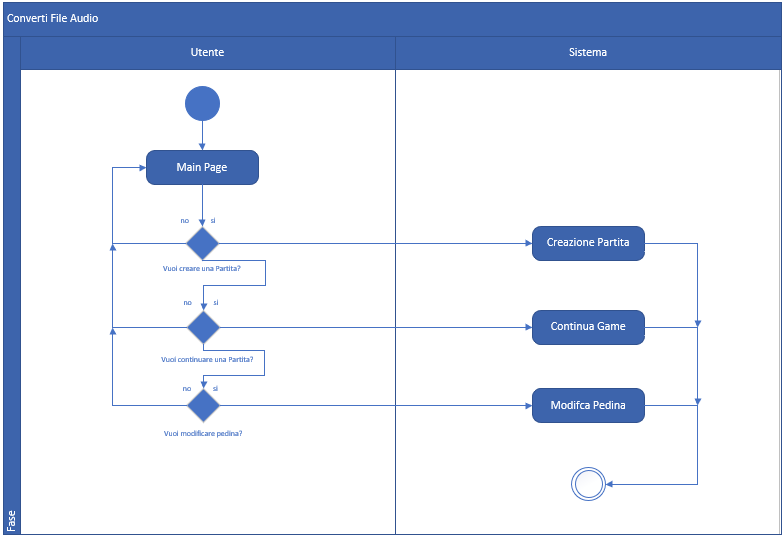
Questo è l’activity diagram per il login dell’applicazione WEB Belmopoly. Come si può vedere ha una struttura estremamente semplice, infatti inizialmente si chiede all’utente se è già registrato, se si va verso il login, altrimenti va verso la registrazione che poi riporterà al punto iniziale. Una volta passato il login il sistema controllerà se i dati inseriti sono validi.

#### Registrazione



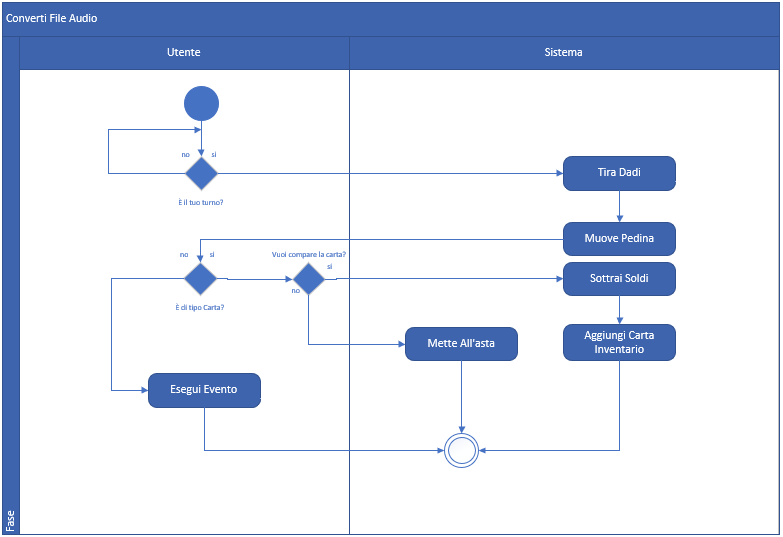
Questo è l’activity diagram per registrarsi sull’applicazione. Rispetto al diagramma di login esso ha una struttura leggermente più complessa, ma il concetto di funzionamento rimane semplice e intuitivo. Infatti vengono eseguite semplicemente due tipi di azione. Inserire il dato, che viene effettuata dall’utente, e controllare il dato inserito, che viene svolto dal sistema.

#### Globale



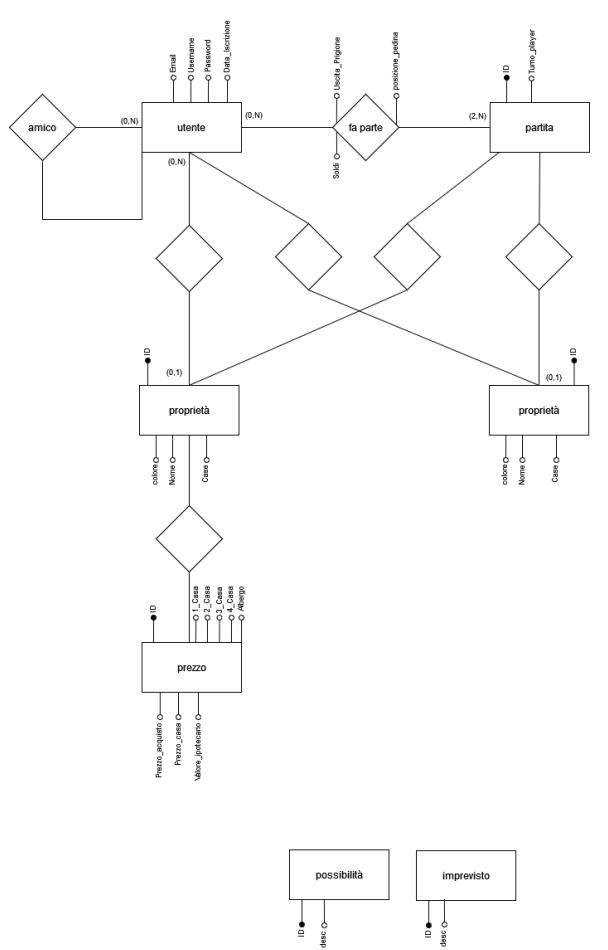
Questo è l’activity diagram che rappresenta tutte le azioni che si possono eseguire prima di iniziare una partita all’interno dell’applicazione. Come si può vedere è una struttura abbastanza complessa, però il concetto rimane semplice e intuitivo. L’utente una volta che si trova nella Main Page può decidere se creare una partita, continuare una partita, oppure modificare la propria pedina.

#### Gioco Effettivo



Questo è l’activity diagram che rappresenta il funzionamento del gioco effettivo, cioè quando l’utente si trova in partita. Esso è il diagramma con la struttura più complessa e anche a livello di concetto è più impegnativo rispetto agli altri. Come si può vedere c’è un controllo iniziale che “chiede” all’utente se il suo turno. Dopodiché ci sono tutte le azioni che permettono di giocare al classico Monopoli.

### Diagramma ER



Questo è il diagramma ER, esso rappresenta la struttura del DataBase che ho lo scopo di contenere tutti i dati riguardante l’applicazione WEB Belmopoly. Come si vede il DataBase è formato da 5 tabelle principali.

La tabella **utente** serve per salvare al suo interno le informazioni degli utenti registrati, perciò il nome utente, l’email e la password crittografata.

La tabella **partita** server per salvare tutti i dati che riguardano una specifica partita che è stata slavata per poi continuarla in seguito.

La tabella **proprietà** a sinistra serve per salvare tutte le proprietà libere quando si salava una partita per poi continuarla in seguito. A essa è pure collega la tabella che definisce il prezzo della proprietà e i prezzi delle varie case e dell’albergo.

# Implementazione

## Libs

La cartella **libs** invece contiene strumenti di supporto, come ad esempio la classe per connettersi al database o per scrivere nei log. Queste librerie vengono usate dai modelli e dai controller per far funzionare correttamente il progetto.

### Classe Database

namespace libs;  
class Database{  
 private static *$conn*;  
 public static function getConnection() {  
  
 if (isset(self::*$conn*)) {  
 return self::*$conn*;  
 } else {  
 try{  
 self::*$conn* = new \PDO('mysql: = '.*HOST*.'; dbname='.*DATABASE*.'; port='.*PORT*,  
 *USER*, *PASSWORD*);  
 self::*$conn*->setAttribute(\PDO::*ATTR\_ERRMODE*, \PDO::*ERRMODE\_EXCEPTION*);  
 }  
 catch (\PDOException $e){  
 require\_once './application/controller/MappaturaErrori.php';  
  
 Logger::*log*("ERROR -> Database errore di connessione: " . $e->getMessage());  
  
 $MappaturaErrori = new \MappaturaErrori();  
 $MappaturaErrori->ConnectionErrorDatabase();  
 die();  
  
 }  
 return self::*$conn*;  
 }  
 }  
}

**Funzionamento:**

* Mantiene una singola connessione al DB (Singleton) per evitare connessioni multiple.
* Se la connessione esiste, la restituisce subito.
* Se no, crea una nuova connessione PDO con parametri configurati.
* In caso di errore, lo registra e mostra una pagina di errore personalizzata.
* Usa die() per fermare lo script se la connessione fallisce.

## Models

La cartella **Models** contiene i file che rappresentano i dati usati nell’applicazione, come ad esempio il modello dell’utente. Questi modelli servono per gestire e organizzare le informazioni in modo chiaro.

### Classe AutenticazioneUtenti

La classe AutenticazioneUtenti gestisce tutte le operazioni di autenticazione e registrazione degli utenti. Si occupa di verificare le credenziali di accesso, controllare la validità dei dati di registrazione, gestire la sicurezza delle password e mantenere le informazioni di sessione necessarie per l’autenticazione all’interno dell’applicazione.

#### \_\_construct()

public function \_\_construct()  
{  
 $this->conn = \libs\Database::*getConnection*();  
}

Inizializza la connessione al database usando la classe Database. Questo permette di usare il database in tutte le altre funzioni.

#### verificaLogin($email, $password)

public function verificaLogin($email, $password)  
{  
 if (session\_status() == *PHP\_SESSION\_NONE*) {  
 session\_start();  
 }  
  
 try {  
 $sth = $this->conn->prepare('SELECT id, password, username FROM utente WHERE email = :email');  
 $sth->bindValue(':email', $email);  
 $sth->execute();  
  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
 if (!$row) {  
 $\_SESSION["ControlloLogin"] = "Email o password errata";  
 \libs\Logger::*log*("Tentativo di login fallito per email: $email");  
 return false;  
 }  
  
 $username = $row["username"];  
 $passwordHash = $row["password"];  
 $\_SESSION["id"] = $row["id"];  
 setcookie('id', $row["id"], time() + 10000, '/');  
  
 if (password\_verify($password, $passwordHash)) {  
 \libs\Logger::*log*("INFO -> Utente $email autenticato con successo");  
  
 $\_SESSION['email'] = $email;  
 $\_SESSION['username'] = $username;  
 $\_SESSION["isAuthenticated"] = true;  
 return true;  
 } else {  
 $\_SESSION["ControlloLogin"] = "Email o password errata";  
 \libs\Logger::*log*("Tentativo di login fallito per email: $email");  
 return false;  
 }  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("ERROR -> Errore verificaLogin: " . $e->getMessage());  
 return false;  
 }  
}

Controlla se le credenziali di accesso (email e password) corrispondono a un utente registrato.

* Cerca nel database l’utente con l’email fornita.
* Se l’utente non esiste, salva un messaggio di errore nella sessione e torna false.
* Se esiste, verifica che la password inserita corrisponda a quella memorizzata (hashata).
* Se la password è corretta, salva dati importanti nella sessione e un cookie per tenere traccia dell’accesso e torna true.
* Se la password è sbagliata, registra l’errore e torna false.

#### registraUtente($email, $username, $password)

public function registraUtente($email, $username, $password) {  
 require\_once './application/libs/logger.php';  
  
 $regexEmail = '/^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*@[a-z0-9-]+(\.[a-z0-9-]+)\*(\.[a-z]{2,3})$/';  
 $regexPassword = '/^(?=.\*[A-Z])(?=.\*\d)(?=.\*[\W\_]).{8,}$/';  
 $regexUsername = '/^[a-zA-Z0-9]{1,20}$/';  
  
  
 if (!$this->controllaUniicitaUsername($username)) {  
 if (session\_status() == *PHP\_SESSION\_NONE*) {  
 session\_start();  
 }  
  
 $\_SESSION["ControlloRegister"] = "Username è già presente";  
 return false;  
 }  
  
(if di controlli)  
  
 try{  
 $password = password\_hash($password, *PASSWORD\_BCRYPT*);  
  
 $sth = $this->conn->prepare("INSERT INTO utente (email, username, password) VALUES (:email, :username, :password)");  
  
 $sth->bindValue(':email', $email);  
 $sth->bindValue(':username', $username);  
 $sth->bindValue(':password', $password);  
 $sth->execute();  
  
 return true;  
 }catch(\PDOException $e){  
  
 $\_SESSION["ControlloRegister"] = "Registrazione utente non riuscita contatta un amministratore";  
 return false;  
  
 }  
  
}

Registra un nuovo utente controllando la validità e la sicurezza dei dati.

* Controlla che l’username sia unico.
* Controlla che email, username e password rispettino regole precise (es. formato email valido, password sicura).
* Se uno di questi controlli fallisce, salva un messaggio di errore nella sessione e ritorna false.
* Se tutto è corretto, cripta la password e inserisce i dati nel database.
* Registra l’evento e ritorna true se l’inserimento va a buon fine, altrimenti registra l’errore e ritorna false.

#### controllaUniicitaUsername($username)

private function controllaUniicitaUsername($username){  
  
 $controllo = true;  
  
 $sth = $this->conn->prepare("SELECT username from utente");  
 $sth->execute();  
 while ($row = $sth->fetch(\PDO::*FETCH\_ASSOC*)) {  
 if($row["username"] == $username){  
 $controllo = false;  
 }  
 }  
  
 return $controllo;  
}

Controlla nel database che l’username passato non sia già stato registrato da un altro utente.  
Ritorna true se l’username è disponibile, false se è già presente.

### GestioneUtenti

La classe GestioneUtenti si occupa della gestione delle amicizie tra utenti. Al suo interno troviamo le funzioni che permettono di inviare, accettare o rifiutare richieste di amicizia, oltre a mostrare gli amici o cercare nuovi utenti da aggiungere.

#### AggiungiAmico($usernameUtente, $UsernameAmico)

public function AggiungiAmico($usernameUtente, $UsernameAmico)  
{  
 $sth = $this->conn->prepare('SELECT id, username FROM utente WHERE username IN (:utente, :amico)');  
 $sth->bindValue(':utente', $usernameUtente);  
 $sth->bindValue(':amico', $UsernameAmico);  
 $sth->execute();  
  
 $ids = [];  
 while ($row = $sth->fetch( \PDO::*FETCH\_ASSOC*)) {  
 $ids[$row['username']] = $row['id'];  
 }  
  
 if (!isset($ids[$usernameUtente]) || !isset($ids[$UsernameAmico])) {  
 \libs\Logger::*log*("WARN -> Tentativo di amicizia fallito: uno o entrambi gli username non esistono ({$usernameUtente}, {$UsernameAmico})");  
 }  
  
  
 $idUtente = $ids[$usernameUtente];  
 $idAmico = $ids[$UsernameAmico];  
 try{  
 $sth = $this->conn->prepare('INSERT INTO amico (mandante, ricevente,richiesta) VALUES (:mandante, :ricevente,:richiesta)');  
 $sth->bindValue(':mandante', $idUtente);  
 $sth->bindValue(':ricevente', $idAmico);  
 $sth->bindValue(':richiesta', false);  
 $sth->execute();  
 \libs\Logger::*log*("INFO -> Richiesta di amicizia inviata correttamente da {$usernameUtente} a {$UsernameAmico}");  
 }catch (\PDOException $e){  
 \libs\Logger::*log*("WARNING -> Richiesta di amicizia fallita da {$usernameUtente} a {$UsernameAmico} ->".$e->getMessage());  
 }  
  
  
}

Questa funzione serve per inviare una richiesta di amicizia. Recupera gli ID dei due utenti dal database e li inserisce nella tabella amico, segnando che la richiesta non è ancora stata accettata (richiesta = false).

#### MostraRichiesteAmicizia($username)

public function MostraRichiesteAmicizia($username)  
{  
 $amici = array();  
  
 try {  
  
 *// Ottieni l'id dell'utente* $sth = $this->conn->prepare('SELECT id FROM utente WHERE username = :username');  
 $sth->bindValue(':username', $username);  
 $sth->execute();  
 while ($row = $sth->fetch(\PDO::*FETCH\_ASSOC*)) {  
 $idUtente = $row['id'];  
 }  
  
 *// Ottieni gli username delle richieste ricevute* $sth = $this->conn->prepare('  
 SELECT u1.username   
 FROM amico a  
 JOIN utente u1 ON u1.id = a.mandante  
 WHERE a.richiesta = :richiesta  
 AND a.mandante != :mandante  
 AND a.ricevente = :ricevente;  
 ');  
 $sth->bindValue(':richiesta', 0); *// Stato della richiesta* $sth->bindValue(':mandante', $idUtente); *// Non vogliamo che l'utente sia mandante* $sth->bindValue(':ricevente', $idUtente); *// L'utente è il ricevente* $sth->execute();  
  
 } catch (\PDOException $e) {  
 \libs\Logger::*log*("ERROR -> Tentativo di mostrare le richieste di amicizia fallito: {$e->getMessage()}");  
 die();  
 }  
  
 *// Popola l'array $amici con gli oggetti Utente* while ($row = $sth->fetch()) {  
 $utente = new Utente();  
 $utente->setUsername($row['username']); *// Usa username invece di id* $amici[] = $utente;  
 }  
  
 return $amici;  
}

Restituisce l’elenco delle richieste di amicizia ricevute da un utente. Cerca tutte le righe nella tabella amico dove l’utente è il ricevente e la richiesta è ancora in sospeso.

#### accettaRichiestaAmicizia($usernameUtente, $usernameAmico)

public function accettaRichiestaAmicizia($usernameUtente, $usernameAmico)  
{  
 try{  
 $sth = $this->conn->prepare('UPDATE amico a  
 JOIN utente u1 ON u1.id = a.mandante  
 JOIN utente u2 ON u2.id = a.ricevente  
 SET a.richiesta = TRUE  
 WHERE u1.username = :usernameAmico  
 AND u2.username = :usernameUtente');  
  
 $sth->bindValue(':usernameUtente', $usernameUtente);  
 $sth->bindValue(':usernameAmico', $usernameAmico);  
 $sth->execute();  
 \libs\Logger::*log*("INFO -> Amicizia accettata correttamente da {$usernameUtente} a {$usernameAmico}");  
 }catch (\PDOException $e){  
 \libs\Logger::*log*("WARN -> Tentativo di accettare l'amicizia fallito: {$e->getMessage()}");  
 }  
}

Imposta a true il campo richiesta nella tabella amico, indicando che l’utente ha accettato la richiesta di amicizia ricevuta.

#### rifiutaRichiestaAmicizia($usernameUtente, $usernameAmico)

public function rifiutaRichiestaAmicizia($usernameUtente, $usernameAmico)  
{  
 try{  
 $sth = $this->conn->prepare('DELETE a  
 FROM amico a  
 JOIN utente u1 ON u1.id = a.mandante  
 JOIN utente u2 ON u2.id = a.ricevente  
 WHERE u1.username = :usernameAmico  
 AND u2.username = :usernameUtente  
 AND a.richiesta = FALSE');  
  
 $sth->bindValue(':usernameUtente', $usernameUtente);  
 $sth->bindValue(':usernameAmico', $usernameAmico);  
 $sth->execute();  
 \libs\Logger::*log*("INFO -> Rifiuto di amicizia inviata correttamente da {$usernameUtente} a {$usernameAmico}");  
 }catch (\PDOException $e){  
 \libs\Logger::*log*("WARN -> Tentativo di rifiuto amicizia fallito: {$e->getMessage()}");  
 }  
  
}

Elimina dal database una richiesta di amicizia non ancora accettata, se l’utente decide di rifiutarla.

#### MostraAmicizia($param, $username)

public function MostraAmicizia($param, $username)  
{  
 $amici = array();  
  
 if (!isset($param)) {  
 $param = "%";  
 } else {  
 $param = $param . "%"; *// Aggiungi il % per fare la ricerca che inizia con $param* }  
  
 try {  
 $sth = $this->conn->prepare('  
 SELECT   
 CASE   
 WHEN u1.username = :username THEN u2.username  
 ELSE u1.username  
 END AS amico\_username  
 FROM amico a  
 JOIN utente u1 ON u1.id = a.mandante  
 JOIN utente u2 ON u2.id = a.ricevente  
 WHERE (u1.username = :username OR u2.username = :username)  
 AND a.richiesta = 1  
 AND (u1.username LIKE :param OR u2.username LIKE :param)'); *// Aggiunta la condizione LIKE per il parametro* $sth->bindValue(':username', $username);  
 $sth->bindValue(':param', $param);  
 $sth->execute();  
 } catch (\PDOException $e) {  
 \libs\Logger::*log*("ERROR -> Tentativo di mostrare le amicizie fallito: {$e->getMessage()}");  
 die();  
 }  
  
 while ($row = $sth->fetch()) {  
 $utente = new Utente();  
 $utente->setUsername($row['amico\_username']);  
 $amici[] = $utente;  
 }  
  
 return $amici;  
}

Restituisce l’elenco degli amici di un utente. È possibile anche filtrare i risultati in base a un parametro ($param) che cerca tra i nomi degli amici.

#### eliminaAmicizia($usernameUtente, $usernameAmico)

public function eliminaAmicizia($usernameUtente, $usernameAmico)  
{  
  
 try {  
  
 $sth = $this->conn->prepare('  
 DELETE a FROM amico a  
 JOIN utente u1 ON a.mandante = u1.id  
 JOIN utente u2 ON a.ricevente = u2.id  
 WHERE (u1.username = :user1 AND u2.username = :user2)  
 OR (u1.username = :user2 AND u2.username = :user1)  
 ');  
  
 $sth->execute([  
 ':user1' => $usernameUtente,  
 ':user2' => $usernameAmico  
 ]);  
  
 *// Verifica se è stata effettivamente eliminata qualche riga* if($sth->rowCount() > 0) {  
 \libs\Logger::*log*("INFO -> Amicizia tra {$usernameUtente} e {$usernameAmico} eliminata correttamente");  
 return true;  
 } else {  
 \libs\Logger::*log*("WARNING -> Nessuna amicizia trovata tra {$usernameUtente} e {$usernameAmico}");  
 return false;  
 }  
  
 } catch (\PDOException $e) {  
 \libs\Logger::*log*("ERROR -> Eliminazione amicizia fallita: {$e->getMessage()}");  
 }  
}

Rimuove l’amicizia tra due utenti, cancellando la riga corrispondente dalla tabella amico, indipendentemente da chi l’ha inviata.

#### mostraUtenti($param, $username)

public function mostraUtenti($param, $username)  
{  
  
 $utenti = array();  
 $amici = $this->MostraAmicizia('%',$username);  
  
  
 if(!isset($param)){  
 $param = "%";  
 }  
 try{  
 $sth = $this->conn->prepare('SELECT username FROM utente WHERE username LIKE :param AND username != :usernameUtente');  
 $search = $param."%";  
 $sth->bindValue(':param', $search);  
 $sth->bindValue(':usernameUtente',$username);  
 $sth->execute();  
  
 while ($row = $sth->fetch()) {  
 $utente = new Utente();  
 $utente->setUsername($row['username']);  
 $utenti[] = $utente;  
 }  
  
 $amiciUsernames = array\_map(function($a) {  
 return $a->getUsername();  
 }, $amici);  
  
 $utentiUsernames = array\_map(function($u) {  
 return $u->getUsername();  
 }, $utenti);  
  
  
 $utentiUsernames = array\_diff($utentiUsernames, $amiciUsernames);  
  
 $finalUtenti = array\_filter($utenti, function($u) use ($utentiUsernames) {  
 return in\_array($u->getUsername(), $utentiUsernames);  
 });  
  
  
 return array\_values($finalUtenti);  
 }  
 catch (\PDOException $e){  
 \libs\Logger::*log*("WARNING -> Errore nella ricerca dei utenti");  
 }  
}

Restituisce l’elenco degli utenti non ancora amici dell’utente attuale. Filtra i risultati in base al nome inserito e rimuove quelli che sono già presenti nella lista amici.

### GestioneRoom

Questa classe si occupa della gestione delle "room" (stanze di gioco) e delle partite nel sistema. In particolare, gestisce la creazione, eliminazione e gestione degli inviti per i giocatori nelle stanze di gioco, oltre a gestire lo stato della partita (avvio, controllo stato, posizioni dei giocatori, ecc).

#### creaRoom($username)

public function creaRoom($username)  
{  
 try {  
  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->bindValue(":username", $username);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 $id = $row['id'];  
 $\_SESSION['uuid'] = uniqid('', true);  
 setcookie('uuid', $\_SESSION['uuid'], time() + 3600, '/');  
 $sth = $this->conn->prepare("INSERT INTO partita (turno\_player, unique\_key) VALUES (:turno\_player, :unique\_key)");  
 $sth->bindValue(':turno\_player', $id);  
 $sth->bindValue(':unique\_key', $\_SESSION['uuid']);  
 $sth->execute();  
 $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :unique\_key");  
 $sth->bindValue(':unique\_key', $\_SESSION['uuid']);  
 $sth->execute();  
 $partita\_id = $sth->fetchColumn();  
 $sth = $this->conn->prepare("INSERT INTO fa\_parte (utente\_id, partita\_id, capo\_partita, richiesta, utente\_prigione, posizione\_pedina)   
 VALUES (:utente\_id, :partita\_id, :capo\_partita, :richiesta, :utente\_prigione, :posizione\_pedina)");  
 $sth->bindValue(':utente\_id', $id);  
 $sth->bindValue(':partita\_id', $partita\_id);  
 $sth->bindValue(':capo\_partita', true, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':richiesta', false, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':utente\_prigione', false, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':posizione\_pedina', 0, \PDO::*PARAM\_INT*);  
 $sth->execute();  
 \libs\Logger::*log*("INFO -> Room creata con successo, ID: " . $partita\_id);  
  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("WARN -> Creazione room fallita: " . $e->getMessage());  
 }  
  
  
}

Crea una nuova room di gioco associata all’utente indicato. Recupera l’ID utente dal database, genera un UUID univoco per la partita, inserisce la nuova partita nella tabella partita e aggiunge il giocatore come capo partita nella tabella fa\_parte.

#### elliminaRoom($username)

public function elliminaRoom($username)  
{  
 try {  
 $this->conn->beginTransaction();  
  
 *// Ottieni l'id della partita in base alla unique\_key* $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :unique\_key");  
 $sth->execute(['unique\_key' => $\_SESSION['uuid']]);  
 $partita = $sth->fetch();  
  
 if ($partita) {  
 $partitaId = $partita['id'];  
  
 *// Elimina prima i riferimenti in fa\_parte* $sth = $this->conn->prepare("DELETE FROM fa\_parte WHERE partita\_id = :partita\_id");  
 $sth->execute(['partita\_id' => $partitaId]);  
  
 *// Ora elimina la partita* $sth = $this->conn->prepare("DELETE FROM partita WHERE id = :partita\_id");  
 $sth->execute(['partita\_id' => $partitaId]);  
  
 $this->conn->commit();  
 \libs\Logger::*log*("INFO -> Room eliminata con successo: " . $\_SESSION['uuid']);  
  
 $\_SESSION['uuid'] = null;  
 } else {  
 \libs\Logger::*log*("WARN -> Nessuna room trovata con uuid: " . $\_SESSION['uuid']);  
 $this->conn->rollBack();  
 }  
  
 } catch (PDOException $e) {  
 $this->conn->rollBack();  
 \libs\Logger::*log*("ERROR -> Tentativo di eliminare la room fallito: " . $e->getMessage());  
 }  
}

Elimina la room corrispondente all’UUID memorizzato in sessione. Prima elimina tutte le associazioni dei giocatori alla partita (fa\_parte), poi elimina la partita stessa dalla tabella partita, gestendo il tutto tramite transazione per mantenere l’integrità dei dati.

#### invitaAmicoRoom($username)

public function invitaAmicoRoom($username)  
{  
  
 try {  
 echo "ciao";  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->bindValue(":username", $username);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 $id = $row['id'];  
  
 $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :unique\_key");  
 $sth->bindValue(':unique\_key', $\_SESSION['uuid']);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
 $partita\_id = $row['id'];  
  
  
 $sth = $this->conn->prepare("INSERT INTO fa\_parte (utente\_id, partita\_id, capo\_partita, richiesta, utente\_prigione, posizione\_pedina)   
 VALUES (:utente\_id, :partita\_id, :capo\_partita, :richiesta, :utente\_prigione, :posizione\_pedina)");  
 $sth->bindValue(':utente\_id', $id);  
 $sth->bindValue(':partita\_id', $partita\_id);  
 $sth->bindValue(':capo\_partita', false, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':richiesta', true, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':utente\_prigione', false, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':posizione\_pedina', 0, \PDO::*PARAM\_INT*);  
 $sth->execute();  
  
  
 \libs\Logger::*log*("INFO -> Invito room effettuato con successo, ID room: " . $partita\_id . " ID player: " . $id);  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("INFO -> Invito room fallito, ID room: " . $partita\_id . " ID player: " . $id . " error: " . $e->getMessage());  
 }  
}

Invia un invito ad un altro utente per partecipare alla room corrente. Recupera l’ID dell’utente invitato e della partita tramite UUID, quindi inserisce una nuova riga in fa\_parte con la richiesta di partecipazione attiva.

#### getInvitiConUsernameCapo($username)

public function getInvitiConUsernameCapo($username)  
{  
 $richieste = [];  
  
 try {  
 *// 1. Prendo l'id dell'utente* $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->bindValue(":username", $username);  
 $sth->execute();  
 $userRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$userRow) {  
 throw new \Exception("Utente non trovato.");  
 }  
  
 $userId = $userRow['id'];  
  
 $sth = $this->conn->prepare("  
 SELECT u.username AS capo\_username  
 FROM fa\_parte f\_mio  
 INNER JOIN fa\_parte f\_capo ON f\_mio.partita\_id = f\_capo.partita\_id AND f\_capo.capo\_partita = TRUE  
 INNER JOIN utente u ON f\_capo.utente\_id = u.id  
 WHERE f\_mio.utente\_id = :utente\_id AND f\_mio.richiesta = TRUE  
 ");  
 $sth->bindValue(":utente\_id", $userId, \PDO::*PARAM\_INT*);  
 $sth->execute();  
  
 while ($row = $sth->fetch(\PDO::*FETCH\_ASSOC*)) {  
 $richieste[] = $row['capo\_username'];  
 }  
  
 return $richieste;  
  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("WARN -> Errore nel recupero inviti: " . $e->getMessage());  
 return [];  
 }  
}

Recupera una lista di username dei capi partita che hanno inviato inviti di partecipazione all’utente specificato. Utilizza join tra le tabelle fa\_parte e utente per ottenere i dati correlati.

#### accettaInvito($username)

public function accettaInvito($username)  
{  
 try {  
  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->bindValue(":username", $username);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
 $id = $row['id'];  
 $sth = $this->conn->prepare("  
 SELECT DISTINCT f\_mio.partita\_id as id  
 FROM fa\_parte f\_mio  
 INNER JOIN fa\_parte f\_capo   
 ON f\_mio.partita\_id = f\_capo.partita\_id   
 AND f\_capo.capo\_partita = TRUE  
 WHERE f\_mio.utente\_id = :utente\_id   
 AND f\_mio.richiesta = TRUE  
 ");  
 $sth->bindValue(":utente\_id", $id, \PDO::*PARAM\_INT*);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
 $partita\_id = $row['id'];  
 $sth = $this->conn->prepare("SELECT unique\_key FROM partita WHERE id = :id");  
 $sth->bindValue(":id", $partita\_id);  
 $sth->execute();  
 $row = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
 $uuid = $row['unique\_key'];  
  
 $\_SESSION['uuid'] = $uuid;  
 setcookie('uuid', $uuid, time() + 3600, '/');  
  
  
 $sth = $this->conn->prepare("UPDATE fa\_parte   
 SET capo\_partita = :capo\_partita   
 WHERE utente\_id = :utente\_id AND partita\_id = :partita\_id");  
  
 $sth->bindValue(':capo\_partita', false, \PDO::*PARAM\_BOOL*);  
 $sth->bindValue(':utente\_id', $id, \PDO::*PARAM\_INT*);  
 $sth->bindValue(':partita\_id', $partita\_id, \PDO::*PARAM\_INT*);  
 $sth->execute();  
  
  
 \libs\Logger::*log*("INFO -> Invito room effettuato con successo, ID room: " . $partita\_id . " ID player: " . $id);  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("INFO -> Invito room fallito, ID room: " . $partita\_id . " ID player: " . $id . " error: " . $e->getMessage());  
 }  
}

Permette all’utente di accettare un invito ricevuto per una partita. Imposta la sessione e il cookie con l’UUID della partita, aggiorna lo stato nella tabella fa\_parte per indicare che la richiesta non è più in sospeso.

#### eliminaInvitoGiocatore($capoUsername, $giocatoreUsername)

public function eliminaInvitoGiocatore($capoUsername, $giocatoreUsername)  
{  
 try {  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->execute(['username' => $capoUsername]);  
 $capoRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$capoRow) {  
 throw new \Exception("Capo non trovato.");  
 }  
 $capoId = $capoRow['id'];  
  
 $sth = $this->conn->prepare("  
 SELECT partita\_id   
 FROM fa\_parte   
 WHERE utente\_id = :capo\_id AND capo\_partita = TRUE  
 ");  
 $sth->execute(['capo\_id' => $capoId]);  
 $partitaRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$partitaRow) {  
 throw new \Exception("Partita non trovata per il capo.");  
 }  
 $partitaId = $partitaRow['partita\_id'];  
  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->execute(['username' => $giocatoreUsername]);  
 $utenteRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$utenteRow) {  
 throw new \Exception("Giocatore non trovato.");  
 }  
 $utenteId = $utenteRow['id'];  
  
 $sth = $this->conn->prepare("  
 DELETE FROM fa\_parte   
 WHERE partita\_id = :partita\_id AND utente\_id = :utente\_id  
 ");  
 $sth->execute([  
 'partita\_id' => $partitaId,  
 'utente\_id' => $utenteId  
 ]);  
  
 \libs\Logger::*log*("INFO -> Invito eliminato per '$giocatoreUsername' nella partita del capo '$capoUsername'.");  
  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("ERROR -> Errore nell'eliminazione invito: " . $e->getMessage());  
 }  
}

Rimuove un invito o la partecipazione di un giocatore specifico dalla partita il cui capo è l’utente indicato. Verifica che il capo e la partita esistano prima di eseguire la cancellazione.

#### startGame()

public function startGame()  
{  
 try {  
 $uuidPartita = $\_SESSION['uuid'];  
  
 *// 1. Recupera l'id della partita tramite uuid* $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :uuid");  
 $sth->execute(['uuid' => $uuidPartita]);  
 $partitaRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$partitaRow) {  
 throw new \Exception("Partita non trovata con uuid: $uuidPartita");  
 }  
  
 $partitaId = $partitaRow['id'];  
  
 *// 2. Aggiorna la colonna `room` a false per tutti i partecipanti* $sth = $this->conn->prepare("  
 UPDATE fa\_parte   
 SET room = FALSE   
 WHERE partita\_id = :partita\_id  
 ");  
 $sth->execute(['partita\_id' => $partitaId]);  
  
 \libs\Logger::*log*("INFO -> Partita con UUID '$uuidPartita' avviata. Room impostata su FALSE per tutti.");  
  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("ERROR -> Errore nell'avvio della partita con uuid '$uuidPartita': " . $e->getMessage());  
 }  
}

Avvia la partita associata all’UUID di sessione impostando il campo room a false per tutti i partecipanti, indicando che la room è chiusa e la partita è iniziata.

#### isStartGame($username)

public function isStartGame($username)  
{  
  
 try {  
  
 $sth = $this->conn->prepare("SELECT id FROM utente WHERE username = :username");  
 $sth->execute(['username' => $username]);  
 $userRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$userRow) {  
 throw new \Exception("Utente non trovato.");  
 }  
  
 $userId = $userRow['id'];  
  
  
 $sth = $this->conn->prepare("  
 SELECT room   
 FROM fa\_parte   
 WHERE utente\_id = :utente\_id  
 LIMIT 1  
 ");  
 $sth->execute(['utente\_id' => $userId]);  
 $roomRow = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
  
 if (!$roomRow) {  
 return true; *// Non è in nessuna room => considerata "chiusa"* }  
  
  
 return !$roomRow['room'];  
  
 } catch (\Exception $e) {  
 \libs\Logger::*log*("ERROR -> Errore nel controllo room: " . $e->getMessage());  
 return true; *// Per sicurezza ritorna true (come se fosse chiusa)* }  
  
  
}

Controlla se la partita alla quale partecipa l’utente è già iniziata controllando il campo room. Ritorna true se la partita è iniziata (room chiusa), false altrimenti.

#### numeroGiocatori($uuid)

public function numeroGiocatori($uuid)  
{  
 *// Ottieni l'id della partita tramite UUID* $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :uuid");  
 $sth->execute(['uuid' => $uuid]);  
 $idPartita = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$idPartita || !isset($idPartita['id'])) {  
 return []; *// Nessuna partita trovata* }  
  
 $idPartita = $idPartita['id'];  
  
 *// Ottieni tutti gli username dei giocatori che fanno parte della partita, ordinati per utente\_id* $sth = $this->conn->prepare("  
 SELECT u.username  
 FROM fa\_parte fp  
 JOIN utente u ON fp.utente\_id = u.id  
 WHERE fp.partita\_id = :id  
 ORDER BY u.id ASC  
");  
 $sth->execute(['id' => $idPartita]);  
 $usernames = $sth->fetchAll(\PDO::*FETCH\_COLUMN*); *// Prende solo la colonna 'username'* return $usernames;  
}

Restituisce un array contenente gli username di tutti i giocatori che partecipano alla partita identificata dall’UUID, ordinati per ID utente.

#### salvaPosizionePedina($posizione)

public function salvaPosizionePedina($posizione)  
{  
  
 \libs\Logger::*log*("INFO -> Posizione: " . $posizione);  
 *// Ottieni l'id della partita tramite UUID* $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :uuid");  
 $sth->execute(['uuid' => $\_SESSION['uuid']]);  
 $idPartita = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$idPartita || !isset($idPartita['id'])) {  
 return []; *// Nessuna partita trovata* }  
  
 $idPartita = $idPartita['id'];  
 \libs\Logger::*log*("INFO -> idPartita: " . $idPartita);  
  
 *// Ottieni tutti gli username dei giocatori che fanno parte della partita, ordinati per utente\_id* $sth = $this->conn->prepare("  
 UPDATE fa\_parte  
 SET posizione\_pedina = :posizione  
 WHERE partita\_id = :idPartita AND utente\_id = :utenteid;  
  
");  
 $sth->execute([  
 'idPartita' => $idPartita,  
 'utenteid' => $\_COOKIE['id'],  
 'posizione' => $posizione  
 ]);  
 \libs\Logger::*log*("INFO -> id: " . $\_COOKIE['id']);  
  
 $usernames = $sth->fetchAll(\PDO::*FETCH\_COLUMN*); }

Aggiorna la posizione della pedina del giocatore nella partita corrente. Usa l’UUID di sessione per identificare la partita e l’ID utente presente nel cookie per aggiornare la colonna posizione\_pedina nella tabella fa\_parte.

#### prendiPosizionePedina()

public function prendiPosizionePedina()  
{  
 *// Ottieni l'id della partita tramite UUID* $sth = $this->conn->prepare("SELECT id FROM partita WHERE unique\_key = :uuid");  
 $sth->execute(['uuid' => $\_SESSION['uuid']]);  
 $idPartita = $sth->fetch(\PDO::*FETCH\_ASSOC*);  
  
 if (!$idPartita || !isset($idPartita['id'])) {  
 return []; *// Nessuna partita trovata* }  
  
 $idPartita = $idPartita['id'];  
  
 *// Ottieni tutte le posizioni dei giocatori nella partita, unendo con utente per avere username* $sth = $this->conn->prepare("  
 SELECT u.username, fp.posizione\_pedina  
 FROM fa\_parte fp  
 JOIN utente u ON u.id = fp.utente\_id  
 WHERE fp.partita\_id = :idPartita  
");  
 $sth->execute(['idPartita' => $idPartita]);  
  
 $risultati = $sth->fetchAll(\PDO::*FETCH\_ASSOC*);  
  
 *// Costruisci array associativo username => posizione\_pedina* $posizioni = [];  
 foreach ($risultati as $riga) {  
 $posizioni[$riga['username']] = $riga['posizione\_pedina'];  
 \libs\Logger::*log*("INFO -> prova: " . $riga['username'] . " " . $riga['posizione\_pedina']);  
 }  
  
 return $posizioni;  
}

Recupera la posizione delle pedine di tutti i giocatori nella partita corrente. Restituisce un array associativo dove la chiave è lo username e il valore è la posizione della pedina.

## Controller

La cartella **Controller** gestisce le richieste degli utenti. Qui si controlla la logica del gioco, si chiamano i modelli per lavorare con i dati e si decide cosa mostrare nelle pagine o risposte.

### Classe autenticazione

Gestisce il processo di login, logout e registrazione degli utenti. Controlla se l’utente è autenticato, mostra le pagine di login e registrazione, verifica le credenziali e registra nuovi utenti tramite il modello corrispondente.

#### controlloLogin()

public function controlloLogin(){  
  
  
 if (session\_status() == *PHP\_SESSION\_NONE*) {  
 session\_start();  
 }  
 if(!isset($\_SESSION["isAuthenticated"])){  
  
 $this->login();  
  
 return false;  
 }else{  
 return true;  
 }  
}

Controlla se l’utente ha già una sessione attiva e risulta autenticato. Se non lo è, mostra la pagina di login e ritorna false; se è autenticato, ritorna true.

#### logout()

public function logout(){  
  
 if (session\_status() == *PHP\_SESSION\_NONE*) {  
 session\_start();  
 }  
 session\_destroy();  
  
 $this->login();  
}

Chiude la sessione corrente dell’utente e poi mostra la pagina di login per permettere un nuovo accesso.

#### login()

public function login(){  
 require 'application/views/login/index.php';  
}

Carica e mostra la pagina di login, dove l’utente può inserire le proprie credenziali.

#### verificaLogin()

public function verificaLogin(){  
  
  
 $\_SESSION["ControlloLogin"] = "";  
  
 $email = $\_POST['email'];  
 $password = $\_POST['password'];  
  
  
  
 if(empty($email) or empty($password)){  
 $\_SESSION["ControlloLogin"] = "Inserisci tutti i campi";  
 $this->login();  
 }else{  
  
 $AutenticazioneUtenti = new \models\AutenticazioneUtenti();  
  
 if($AutenticazioneUtenti->verificaLogin($email,$password)){  
  
 require\_once './application/controller/Home.php';  
 $home = new Home();  
 $home->index();  
 }else{  
  
 $this->login();  
 }  
  
 }  
  
}

Prende i dati inseriti nel form di login (email e password), verifica che non siano vuoti, e controlla le credenziali tramite il modello. Se la verifica ha successo, carica la pagina principale (home); altrimenti rimanda alla pagina di login.

#### RegistraUtenteView()

public function RegistraUtenteView()  
{  
 require 'application/views/register/index.php';  
  
}

Carica e mostra la pagina per registrare un nuovo utente, con i campi necessari.

#### RegistraUtente()

public function RegistraUtente()  
{  
  
 $email = $\_POST['email'];  
 $username = $\_POST['username'];  
 $password = $\_POST['password'];  
  
  
 if(empty($email) or empty($username) or empty($password)){  
  
 if (session\_status() == *PHP\_SESSION\_NONE*) {  
 session\_start();  
 }  
  
 $\_SESSION["ControlloRegister"] = "Inserisci tutti i campi";  
  
 $this->registraUtenteView();  
 }else{  
 $AutenticazioneUtenti = new \models\AutenticazioneUtenti();  
 if($AutenticazioneUtenti->registraUtente($email,$username,$password)){  
 $this->login();  
 }else{  
 $this->registraUtenteView();  
 }  
 }  
  
}

Riceve i dati di registrazione dal form, verifica che tutti i campi siano compilati, e prova a registrare l’utente tramite il modello. Se la registrazione ha successo, mostra la pagina di login; altrimenti torna alla pagina di registrazione.

### Classe GestioneAccount

Gestisce le funzionalità legate agli amici e alle amicizie degli utenti. Controlla se l’utente è autenticato prima di eseguire azioni come aggiungere un amico, mostrare richieste di amicizia, accettare o rifiutare richieste, eliminare amicizie e visualizzare la lista di amici o utenti. La classe usa il modello GestioneUtenti per operazioni sul database e carica le relative viste per mostrare i dati.

#### aggiungiAmico($username = null)

public function aggiungiAmico($username = null){  
 require\_once "./application/controller/Autenticazione.php";  
 require\_once "./application/controller/home.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $regexUsername = '/^[a-zA-Z0-9]{1,20}$/';  
 $usernameAmico = $username;  
  
 if(preg\_match($regexUsername, $usernameAmico)){  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $GesioneUtenti->AggiungiAmico($\_SESSION['username'], $usernameAmico);  
  
  
 $this->mostraUtenti();  
 }else{  
 $\_SESSION["VerificaAmico"] = "Username non valido";  
 }  
  
 }  
}

Verifica che l’utente sia loggato e che lo username passato sia valido. Se sì, aggiunge l’amico specificato alla lista amici dell’utente, poi mostra la lista aggiornata degli utenti.

#### mostraRichiesteAmicizia()

public function mostraRichiesteAmicizia(){  
 require\_once "./application/controller/Autenticazione.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $amici = $GesioneUtenti->MostraRichiesteAmicizia($\_SESSION['username']);  
  
  
 require 'application/views/notifhce/requests.php';  
  
  
 }  
}

Controlla se l’utente è autenticato, quindi recupera tutte le richieste di amicizia ricevute e carica la vista per mostrarle.

#### accettaRichiestaAmicizia($username = null)

public function accettaRichiestaAmicizia($username = null)  
{  
 require\_once "./application/controller/Autenticazione.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
 $regexUsername = '/^[a-zA-Z0-9]{1,20}$/';  
 $usernameAmico = $username;  
 if(preg\_match($regexUsername, $usernameAmico)) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $GesioneUtenti->accettaRichiestaAmicizia($\_SESSION['username'], $usernameAmico);  
  
 $this->mostraRichiesteAmicizia();  
  
 }else{  
 $\_SESSION["VerificaAmico"] = "Username non valido";  
 }  
  
 }  
}

Se l’utente è loggato e lo username è valido, accetta la richiesta di amicizia da quell’utente e aggiorna la lista delle richieste.

#### rifiutaRichiestaAmicizia($username)

public function rifiutaRichiestaAmicizia($username)  
{  
 require\_once "./application/controller/Autenticazione.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()){  
 $regexUsername = '/^[a-zA-Z0-9]{1,20}$/';  
 $usernameAmico = $username;  
 if(preg\_match($regexUsername,$usernameAmico)){  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $GesioneUtenti->rifiutaRichiestaAmicizia($\_SESSION['username'], $usernameAmico);  
 $this->mostraRichiesteAmicizia();  
 }else{  
 $\_SESSION["VerificaAmico"] = "Username non valido";  
 }  
  
 }  
  
}

Controlla login e validità dello username, poi rifiuta la richiesta di amicizia proveniente da quell’utente e aggiorna la lista delle richieste.

#### mostraAmicizie($param = '%')

public function mostraAmicizie($param = '%'){  
 require\_once "./application/controller/Autenticazione.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
  
 $amici = $GesioneUtenti->MostraAmicizia($param, $\_SESSION['username']);  
  
 require 'application/views/amici/friends.php';  
  
 }  
}

Verifica il login e mostra la lista degli amici filtrata con il parametro dato (o tutti se non specificato), caricando la vista dedicata.

#### elliminaAmicizia($username = null)

public function elliminaAmicizia($username = null)  
{  
 require\_once "./application/controller/Autenticazione.php";  
 require\_once "./application/controller/home.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {

$regexUsername = '/^[a-zA-Z0-9]{1,20}$/';  
 $usernameAmico = $username;  
  
  
 if(preg\_match($regexUsername, $usernameAmico)) { $GesioneUtenti = new \models\GestioneUtenti();  
 $GesioneUtenti->eliminaAmicizia($\_SESSION['username'], $usernameAmico);  
 } else {  
 $\_SESSION["VerificaAmico"] = "Username non valido";  
 }  
 }  
  
 $this->mostraAmicizie();  
}

Controlla login e validità dello username, elimina l’amicizia con quell’utente e mostra la lista aggiornata degli amici.

#### mostraUtenti($param = null)

public function mostraUtenti($param = null)  
{  
 require\_once "./application/controller/Autenticazione.php";  
 require\_once "./application/controller/home.php";  
  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
 $utenti = $GesioneUtenti->mostraUtenti($param, $\_SESSION['username']);  
  
  
 require\_once "./application/views/amici/users.php";  
  
 }else{  
 $home = new home();  
 $home->index();  
 }  
   
}

Se l’utente è autenticato, recupera e mostra una lista di utenti filtrata (escluso l’utente stesso), caricando la vista corrispondente. Altrimenti, mostra la pagina home.

### Classe home

Gestisce le azioni principali legate alla home dell’utente, alle stanze di gioco (room), agli inviti e all’avvio delle partite. Controlla l’autenticazione e coordina la visualizzazione delle pagine e le operazioni sulle stanze.

#### index()

public function index(){  
require\_once "./application/controller/autenticazione.php";  
require\_once "./application/controller/GestioneAccount.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 require\_once "./application/views/home/index.php";  
  
  
 }  
}

Controlla se l’utente è loggato e mostra la pagina principale (home).

#### creaRoom()

public function creaRoom()  
{  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $amici = $GesioneUtenti->MostraAmicizia('%',$\_SESSION['username']);  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $creaRoom->creaRoom($\_SESSION['username']);  
  
 header("Location:" . *URL* . "Home/creaRoomView");  
 }  
}

Se l’utente è autenticato, crea una nuova stanza di gioco per lui e reindirizza alla pagina di creazione stanza.

#### creaRoomView()

public function creaRoomView()  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $GesioneUtenti = new \models\GestioneUtenti();  
  
 $amici = $GesioneUtenti->MostraAmicizia("%",$\_SESSION['username']);  
  
 require\_once "./application/views/creazioneRoom/index.php";  
  
  
 }  
  
}

Mostra la pagina per la creazione della stanza, caricando la lista amici dell’utente.

#### esciRoom()

public function esciRoom(){  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $creaRoom->elliminaRoom($\_SESSION['username']);  
  
 $this->index();  
  
 }  
  
  
}

Se l’utente è loggato, elimina la stanza di gioco dell’utente e ritorna alla home.

#### invitaRoom($username = null)

public function invitaRoom($username = null)  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $creaRoom->invitaAmicoRoom($username);  
  
 header("Location:" . *URL* . "home/creaRoomView");  
  
 }  
  
}

Invita un amico specificato a unirsi alla stanza di gioco dell’utente, poi ricarica la pagina di creazione stanza.

#### mostraRichiesteRoom()

public function mostraRichiesteRoom()  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $users = $creaRoom->getInvitiConUsernameCapo($\_SESSION['username']);  
  
  
  
 require\_once "./application/views/notifhce/invites.php";  
  
 }  
  
}

Mostra le richieste di invito alle stanze ricevute dall’utente.

#### accettaRichiesteRoom()

public function accettaRichiesteRoom()  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $creaRoom->accettaInvito($\_SESSION['username']);  
  
 $this->creaRoomView();  
  
 }  
  
}

Accetta un invito alla stanza di gioco e mostra la pagina di creazione stanza.

#### elliminaInvitoRoom($capoPartita = null)

public function elliminaInvitoRoom($capoPartita = null)  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
  
 $creaRoom->eliminaInvitoGiocatore($capoPartita,$\_SESSION['username']);  
  
 $this->mostraRichiesteRoom();  
  
 }  
  
}

Elimina un invito ricevuto da una stanza specifica e aggiorna la lista degli inviti.

#### isStartGame()

public function isStartGame()  
{  
  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 header('Content-Type: application/json');  
 $gestioneRoom = new \models\GestioneRoom();  
 $response = $gestioneRoom->isStartGame($\_SESSION['username']);  
  
 }  
   
}

Controlla se il gioco è stato avviato nella stanza dell’utente e restituisce la risposta in formato JSON.

#### startGame()

public function startGame()  
{  
 require\_once "./application/controller/autenticazione.php";  
 $autenticazione = new autenticazione();  
  
 if($autenticazione->controlloLogin()) {  
  
 $creaRoom = new \models\GestioneRoom();  
 $creaRoom->startGame(); *// aggiorna il DB* $uuid = $\_SESSION['uuid'];  
  
$client = new Client("ws://localhost:3000");  
$client->send(json\_encode([  
 'startGame' => true,  
 'room' => $uuid,  
 'startingIndex' => 0  
 ]));  
  
 $client->close();  
header("Location:" . *URL* . "board/index");  
 }  
}

Avvia il gioco aggiornando il database e invia un messaggio WebSocket per notificare il server Node.js. Poi reindirizza l’utente alla pagina del gioco.

## WebSocket

Gestisce la comunicazione in tempo reale tra i giocatori. Permette di entrare in una stanza, passare il turno, ricevere aggiornamenti istantanei e sincronizzare lo stato del gioco. Essenziale per il funzionamento live delle partite.

### Gestione turni

Questo modulo implementa un server WebSocket che gestisce il sistema di turni tra giocatori nelle stanze di gioco.

#### Inizializzazione del server WebSocket

const WebSocket = require('ws');  
const wss = new WebSocket.Server({ port: 4000 });

Avvia un server WebSocket in ascolto sulla porta 4000, pronto a ricevere connessioni dai client.

#### Gestione connessioni WebSocket

if (data.joinRoom && data.playerId) {  
 const roomId = data.joinRoom;  
 const playerId = data.playerId;  
  
 if (!rooms[roomId]) {  
 rooms[roomId] = {  
 players: {},  
 playerOrder: [],  
 currentTurnPlayerId: playerId   
 };  
 console.log(` Stanza ${roomId} creata, turno a ${playerId}`);  
 }  
  
 if (rooms[roomId].players[playerId]?.closeTimeout) {  
 *clearTimeout*(rooms[roomId].players[playerId].closeTimeout);  
 console.log(` Timeout chiusura annullato per ${playerId}`);  
 }  
  
 rooms[roomId].players[playerId] = { ws };  
 ws.roomId = roomId;  
 ws.playerId = playerId;  
  
 if (!rooms[roomId].playerOrder.includes(playerId)) {  
 rooms[roomId].playerOrder.push(playerId);  
 }  
  
 console.log(` Giocatore ${playerId} aggiunto/aggiornato nella stanza ${roomId}`);  
  
 sendRoomStateToPlayer(roomId, playerId);  
}

Quando un giocatore si connette ed entra in una stanza, viene:

* creata la stanza (se non esiste),
* assegnato il turno iniziale,
* salvato il WebSocket,
* aggiornato l’ordine dei giocatori.

Viene poi inviato lo stato corrente al giocatore.

#### Fine turno e cambio giocatore (turnEnd)

if (data.turnEnd && data.room && data.playerId) {  
 const roomId = data.room;  
 const playerId = data.playerId;  
  
 const room = rooms[roomId];  
 if (room && room.currentTurnPlayerId === playerId) {  
 const order = room.playerOrder;  
 const currentIndex = order.indexOf(playerId);  
 const nextIndex = (currentIndex + 1) % order.length;  
 const nextPlayerId = order[nextIndex];  
  
 room.currentTurnPlayerId = nextPlayerId;  
 console.log(` Turno passato a ${nextPlayerId} nella stanza ${roomId}`);  
  
 broadcastRoomState(roomId);  
 }  
}

Quando un giocatore termina il turno, il server:

* verifica che sia davvero il suo turno,
* passa il turno al prossimo giocatore in ordine,
* invia a tutti lo stato aggiornato.

#### Gestione disconnessione del client

ws.on('close', () => {  
 const roomId = ws.roomId;  
 const playerId = ws.playerId;  
  
 console.log(` Connessione chiusa per ${playerId}`);  
  
 if (roomId && rooms[roomId] && rooms[roomId].players[playerId]) {  
 rooms[roomId].players[playerId].closeTimeout = *setTimeout*(() => {  
 console.log(` Timeout scaduto: rimuovo ${playerId} dalla stanza ${roomId}`);  
  
 delete rooms[roomId].players[playerId];  
 rooms[roomId].playerOrder = rooms[roomId].playerOrder.filter(id => id !== playerId);  
  
 const remainingPlayers = rooms[roomId].playerOrder;  
  
 if (remainingPlayers.length === 0) {  
 delete rooms[roomId];  
 console.log(` Stanza ${roomId} eliminata (vuota)`);  
 } else {  
  
 if (rooms[roomId].currentTurnPlayerId === playerId) {  
 rooms[roomId].currentTurnPlayerId = remainingPlayers[0];  
 console.log(` Turno riassegnato a ${remainingPlayers[0]}`);  
  
 broadcastRoomState(roomId);  
 }  
 }  
 }, 3000);   
 }  
});

Quando un giocatore si disconnette:

* viene avviato un timeout di 3 secondi,
* se non rientra, viene rimosso dalla stanza,
* se era il suo turno, passa al prossimo,
* se la stanza resta vuota, viene eliminata.

#### broadcastRoomState(roomId)

function broadcastRoomState(roomId) {  
 const room = rooms[roomId];  
 if (!room) return;  
  
 Object.entries(room.players).forEach(([id, player]) => {  
 const client = player.ws;  
 if (client.readyState === WebSocket.OPEN) {  
 client.send(JSON.stringify({  
 turn: id === room.currentTurnPlayerId  
 }));  
 }  
 });  
}

Invia a **tutti i giocatori nella stanza** un messaggio che indica chi ha il turno attivo.

#### sendRoomStateToPlayer(roomId, playerId)

function sendRoomStateToPlayer(roomId, playerId) {  
 const room = rooms[roomId];  
 const client = room?.players[playerId]?.ws;  
  
 if (client && client.readyState === WebSocket.OPEN) {  
 client.send(JSON.stringify({  
 turn: playerId === room.currentTurnPlayerId  
 }));  
 }  
}

Invia al singolo giocatore specificato lo stato della stanza, indicando se è il suo turno.

### StartGame

Questo modulo WebSocket si occupa della **gestione della creazione delle stanze e dell'avvio sincronizzato del gioco**.

#### Inizializzazione del WebSocket Server

const WebSocket = *require*('ws');  
const wss = new WebSocket.Server({ port: 3000 });  
  
let rooms = {};

Importa la libreria WebSocket e avvia il server sulla porta 3000.  
Inoltre, crea un oggetto rooms per mantenere la mappatura tra nomi delle stanze e i client connessi.

#### Connessione iniziale del client

wss.on('connection', ws => {  
 console.log("Un client si è connesso a WebSocket");  
  
 const clientId = Math.random().toString(36).substring(7); *// Genera un ID temporaneo per il client* console.log(`ID Client: ${clientId} - Connessione WebSocket avvenuta`);

Quando un nuovo client si connette, gli viene assegnato un clientId temporaneo a scopo di logging e tracciamento delle attività.

#### Gestione dei messaggi ricevuti

ws.on('message', message => {  
 console.log(`ID Client: ${clientId} - Messaggio ricevuto: ${message}`);  
 const data = JSON.parse(message);

Ogni messaggio ricevuto viene registrato e interpretato in formato JSON. Da qui partono le logiche di join e start.

#### Join a una stanza (joinRoom)

if (data.joinRoom) {  
 const room = data.joinRoom;  
 if (!rooms[room]) {  
 rooms[room] = [];  
 console.log(`ID Client: ${clientId} - Stanza ${room} creata.`);  
 }  
 rooms[room].push(ws);  
 console.log(`ID Client: ${clientId} - Client aggiunto alla stanza ${room}. Client tot: ${rooms[room].length}`);  
}

Se il client invia un messaggio con joinRoom, viene aggiunto alla stanza corrispondente.  
Se la stanza non esiste, viene creata

#### Avvio del gioco (startGame)

if (data.startGame && data.room) {  
 console.log(`ID Client: ${clientId} - Tentativo di avviare il gioco nella stanza ${data.room}`);  
  
 if (rooms[data.room] && rooms[data.room].length > 0) {  
 console.log(`ID Client: ${clientId} - Stanza trovata, avvio del gioco per ${rooms[data.room].length} client`);  
  
 rooms[data.room].forEach(client => {  
 client.send(JSON.stringify({  
 startGame: true,  
 room: data.room,  
 startingIndex: data.startingIndex  
 }));  
 });  
 } else {  
 console.log(`ID Client: ${clientId} - Stanza ${data.room} non trovata o vuota. Impossibile avviare il gioco.`);  
 }  
}

Se un client richiede di avviare il gioco (startGame), il server notifica tutti i client nella stanza.  
Il messaggio inviato include: startGame, room, e startingIndex.

#### Gestione della disconnessione del client

ws.on('close', () => {  
 console.log(`ID Client: ${clientId} - Connessione chiusa`);  
  
  
 for (let room in rooms) {  
 rooms[room] = rooms[room].filter(client => client !== ws);  
 }  
 console.log(`ID Client: ${clientId} - Client disconnesso e rimosso dalle stanze`);  
});

Quando un client si disconnette, viene rimosso da tutte le stanze in cui era presente.

# Test

## Protocollo di test

Definire in modo accurato tutti i test che devono essere realizzati per garantire l’adempimento delle richieste formulate nei requisiti. I test fungono da garanzia di qualità del prodotto. Ogni test deve essere ripetibile alle stesse condizioni.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case:**  **Riferimento**: | TC-001  REQ-012 | **Nome:** | Import a card with KIC, KID and KIK keys, but not shown with the GUI |
| **Descrizione:** | Import a card with KIC, KID and KIK keys with no obfuscation, but not shown with the GUI | | |
| **Prerequisiti:** | Store on local PC: Profile\_1.2.001.xml (appendix n\_n) and Cards\_1.2.001.txt (appendix n\_n).  PIN (OTA\_VIEW\_PIN\_PUK\_KEY) and ADM (OTA\_VIEW\_ADM\_KEY) user right not set. | | |
| **Procedura:** | 1. Go to “Cards manager” menu,  in main page click “Import Profiles” link, Select the “1.2.001.xml” file, Import the Profile 2. Go to “Cards manager” menu,  in main page click “Import Cards” link, Select the “1.2.001.txt” file, Delete the cards,  Select the “1.2.001.txt” file, Import the cards 3. Research the “41795924770” Card, Click the imsi card link Check the card details 4. Execute the SQL: SELECT imsi, dir, keyset, cntr, rawtohex(kickey), rawtohex(kidkey), rawtohex(kikkey), rawtohex(chv), rawtohex(dap)FROM otacardkey a where imsi='340041795924770' ORDER BY keyset; | | |
| **Risultati attesi:** | Keys visible in the DB (OtaCardKey) but not visible in the GUI (Card details) | | |

## Risultati test

Tabella riassuntiva in cui si inseriscono i test riusciti e non del prodotto finale. Se un test non riesce e viene corretto l’errore, questo dovrà risultare nel documento finale come riuscito (la procedura della correzione apparirà nel diario), altrimenti dovrà essere descritto l’errore con eventuali ipotesi di correzione.

## Mancanze/limitazioni conosciute

Descrizione con motivazione di eventuali elementi mancanti o non completamente implementati, al di fuori dei test case. Non devono essere riportati gli errori e i problemi riscontrati e poi risolti durante il progetto.

# Consuntivo

Consuntivo del tempo di lavoro effettivo e considerazioni riguardo le differenze rispetto alla pianificazione (cap. 1.7) (ad esempio Gantt consuntivo).

# Conclusioni

Quali sono le implicazioni della mia soluzione? Che impatto avrà? Cambierà il mondo? È un successo importante? È solo un’aggiunta marginale o è semplicemente servita per scoprire che questo percorso è stato una perdita di tempo? I risultati ottenuti sono generali, facilmente generalizzabili o sono specifici di un caso particolare? ecc.

## Sviluppi futuri

Migliorie o estensioni che possono essere sviluppate sul prodotto.

## Considerazioni personali

Cosa ho imparato in questo progetto? ecc.

# Glossario

Inserite una semplice tabella con due colonne che spieghi i termini specifici del progetto (lista dei termini in ordine alfabetico A-Z)

Esempio:

|  |  |
| --- | --- |
| **Termine** | **Descrizione** |
| AJAX | **Asynchronous JavaScript And XML**: una tecnica che permette di eseguire richieste ed ottenere dati da una pagina web in modo asincrono. |
| CSS | **Cascading Style Sheets**: linguaggio che permette di definire il layout e la grafica di una pagina web. |

# Bibliografia

## Bibliografia per articoli di riviste:

1. Cognome e nome (o iniziali) dell’autore o degli autori, o nome dell’organizzazione,
2. Titolo dell’articolo (tra virgolette),
3. Titolo della rivista (in italico),
4. Anno e numero
5. Pagina iniziale dell’articolo.

## Bibliografia per libri

1. Cognome e nome (o iniziali) dell’autore o degli autori, o nome dell’organizzazione,
2. Titolo del libro (in italico),
3. ev. Numero di edizione,
4. Nome dell’editore,
5. Anno di pubblicazione,
6. ISBN.

## Sitografia

1. URL del sito (se troppo lungo solo dominio, evt completo nel diario),
2. Eventuale titolo della pagina (in italico),
3. Data di consultazione (GG-MM-AAAA).

**Esempio:**

* http://standards.ieee.org/guides/style/section7.html, *IEEE Standards Style Manual*, 07-06-2008.

# Allegati

Elenco degli allegati, esempio:

* Diari di lavoro
* Codici sorgente/documentazione macchine virtuali
* Istruzioni di installazione del prodotto (con credenziali di accesso) e/o di eventuali prodotti terzi
* Documentazione di prodotti di terzi
* Eventuali guide utente / Manuali di utilizzo
* Mandato e/o QdC
* Prodotto
* …

1. <https://trello.com/> [↑](#footnote-ref-1)
2. <https://app.moqups.com/> [↑](#footnote-ref-2)