



**UNIVERSITÀ⁹ DEGLI STUDI DI
NAPOLI FEDERICO II**

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Software Architecture Design**

Documentazione Team B5 Task R1
Profilo Utente

Anno Accademico 2024-2025

Membri del Team:

Camilla D'Andria M63001670

Gabriele Mangiacapre M63001690

Indice

1 Presentazione del Progetto	1
1.1 Descrizione Task R1	2
1.1.1 Requisiti funzionali	3
1.1.2 Requisiti NON funzionali	3
1.2 Diagramma dei casi d'uso	5
1.3 Scenari	8
1.3.1 Caso d'Uso: Gestione Informazioni Profilo . . .	8
1.3.2 Suddivisione del lavoro	12
1.3.3 Stato iniziale del progetto	12
1.3.4 Componenti e Connettori	14
2 Strumenti Utilizzati	18
2.0.1 Microsoft Teams	18
2.0.2 Github	18
2.0.3 Visual Paradigm	20
2.1 Tecnologie utilizzate	21
2.1.1 Maven	23
2.1.2 Postman	24

3 Moduli Utilizzati e le loro interazioni	26
3.0.1 T1	27
3.0.2 T4	28
3.0.3 T5	28
3.0.4 Punti di Forza dell'Architettura	35
3.0.5 T23	35
3.0.6 Diagramma ER e Gestione delle Entità con JPA	42
3.0.7 Introduzione a JPA (Java Persistence API) . .	43
3.1 Descrizione delle Entità JPA e Relazioni	45
3.1.1 Entità User	45
3.1.2 Entità Friend	46
3.1.3 Relazioni nel Diagramma ER	47
3.1.4 Benefici dell'Uso di JPA e Diagramma ER nel Progetto	47
3.2 Deployment Diagram	48
3.2.1 Descrizione del Deployment	49
3.2.2 Uso del Deployment Diagram	51
3.3 Package Diagram	51
3.3.1 Elementi del Package Diagram	52
3.3.2 Descrizione del Package Diagram	53
3.4 Interazione dei microservizi	55
3.4.1 Struttura Generale	55
3.4.2 Flusso di Comunicazione Tra Microservizi . .	59
4 Task R1 - Implementazione	62
4.1 Scelte progettuali	62

4.2	La Sezione Profilo	64
4.2.1	Modifica info personali	65
4.2.2	Aggiorna Biografia	74
4.2.3	Modifica Avatar/ Carica Foto	78
4.3	Sezione Amici	99
4.3.1	Lista Amici	101
4.3.2	Aggiungi Amico	109
4.3.3	Elimina Amico	116
4.4	Sezione Statistiche	120
4.5	Sezione Achievements	121
4.6	Testing	123

Chapter 1

Presentazione del Progetto

Man vs Automated Testing Tools Challenges è un gioco educativo progettato per consentire agli studenti di confrontarsi con strumenti di testing automatizzati, come Randoop ed Evosuite, capaci di generare casi di test JUnit. I task di gioco consistono nello sviluppo di casi di test di unità per classi Java, con l'obiettivo di raggiungere specifici traguardi, tra cui copertura di LOC, Branch, Decisioni, Eccezioni e Weak Mutation. I casi di test devono essere implementati utilizzando JUnit 4.

L'implementazione del progetto segue un approccio basato sulla Gamification, che consente agli studenti di competere con strumenti di generazione automatica, promuovendo apprendimento e divertimento.

La documentazione del progetto e le implementazioni dei gruppi

che hanno contribuito alla sua realizzazione sono disponibili sulla piattaforma GitHub, insieme alla documentazione di ogni task.

1.1 Descrizione Task R1

Il Task R1 prevede il miglioramento del profilo giocatore, introducendo nuove funzionalità e miglioramenti strutturali per rendere il profilo più interattivo e informativo. Nello specifico, il profilo deve essere organizzato in sezioni che includano:

- **Informazioni personali del giocatore:** permettendo agli utenti di modificare le informazioni del loro profilo, inclusa la biografia e l'immagine.
- **Statistiche del giocatore:** visualizzazione di dati relativi a: Partite giocate per tipologia, Numero di partite vinte.
- **Achievement:** visualizzazione di badge o premi che segnalano il raggiungimento di traguardi specifici, come il superamento di soglie di statistiche.
- **Sezione amici** consentendo l'interazione e la connessione con altri utenti, visualizzando una lista di utenti seguiti e con la possibilità di aggiungere o eliminare utenti da questa lista.

1.1.1 Requisiti funzionali

A partire dall'analisi del task assegnato, sono stati definiti i seguenti requisiti funzionali:

Visualizzazione e modifica delle informazioni personali:

Il sistema deve consentire ai giocatori di visualizzare e modificare le proprie informazioni personali in modo chiaro e organizzato.

Visualizzazione Achievement ottenuti e ottenibili: Il sistema deve consentire ai giocatori di visualizzare i traguardi raggiunti e quelli ancora da raggiungere, rappresentati attraverso badge.

Visualizzazione statistiche di gioco: Il sistema deve fornire una funzionalità per consultare le statistiche relative alle partite giocate, suddivise per tipologia.

Ricerca di altri giocatori: Il sistema deve offrire una funzionalità per cercare i giocatori presenti sulla piattaforma tramite id e permettere di seguirli.

Visualizzazione lista amici: Il sistema deve permettere ai giocatori di consultare un elenco aggiornato degli utenti che stanno seguendo.

1.1.2 Requisiti NON funzionali

A partire dall'analisi del task assegnato, sono stati individuati i seguenti requisiti non funzionali:

Performance: Il sistema deve garantire risposta per la visualizzazione delle informazioni personali, statistiche di gioco e Achievement,

anche in presenza di tanti utenti simultanei.

Scalabilità: Il sistema deve essere progettato per supportare un numero crescente di utenti attivi contemporaneamente, senza impatti significativi sulle performance.

Usabilità: L’interfaccia deve essere intuitiva e conforme agli standard di usabilità, con particolare attenzione alla chiarezza delle informazioni e alla semplicità di navigazione.

Manutenibilità:

- Il codice deve essere modulare e documentato per facilitare interventi di manutenzione e futuri aggiornamenti.
- L’architettura deve consentire l’aggiunta di nuove funzionalità senza compromettere il funzionamento del sistema esistente.

1.2 Diagramma dei casi d'uso

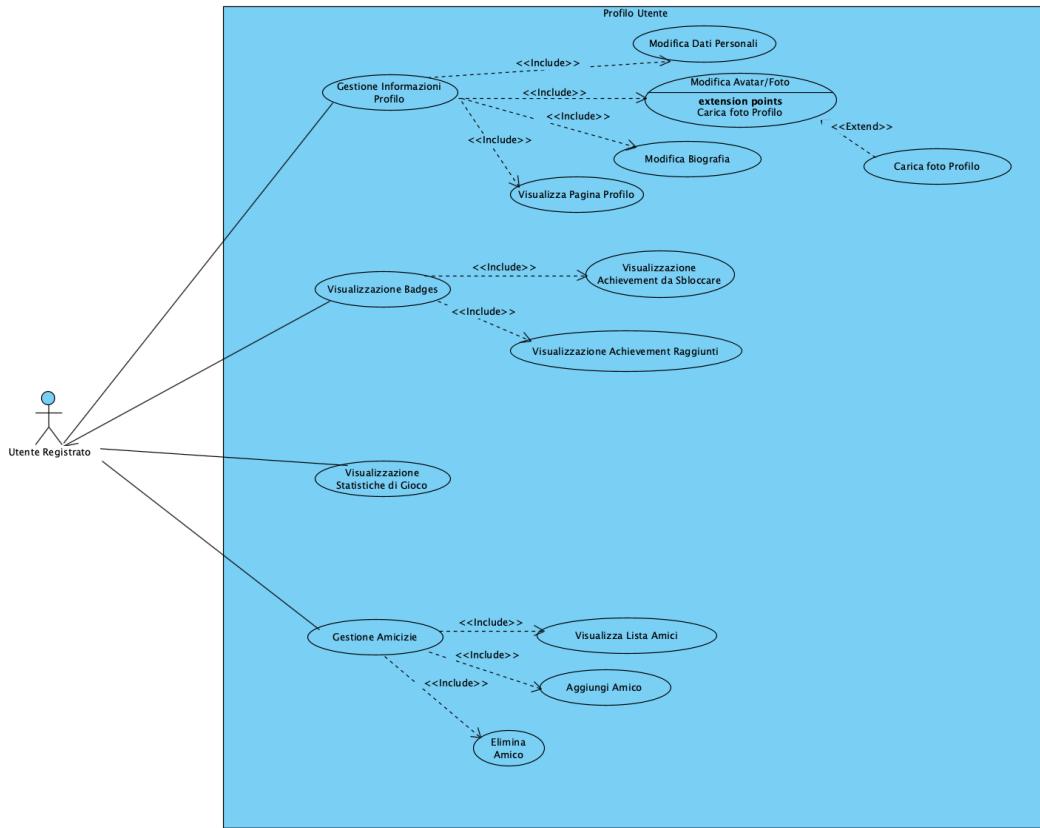


Figure 1.1: Diagramma dei Casi d'uso

L'immagine rappresenta il diagramma dei casi d'uso che descrive le funzionalità del sistema accessibili a un utente registrato nonché il giocatore (studente). L'utente registrato è l'attore principale, indicato da una figura stilizzata, e interagisce con diverse aree del sistema.

Il sistema è organizzato in diverse macro-sezioni:

Gestione del Profilo Utente:

L'utente ha la possibilità di gestire le informazioni personali del proprio profilo. Può modificare i propri dati personali e aggiornare l'avatar o

la foto associata.

Visualizzazione dei Badges:

Questa sezione permette di esplorare i traguardi raggiunti e quelli ancora da sbloccare. L'utente può vedere quali badge ha già ottenuto e quali sono i prossimi obiettivi da completare mediante una barra dei progressi.

Statistiche di Gioco:

L'utente può accedere a un'ampia gamma di statistiche sulle proprie attività nel sistema. È possibile visualizzare l'elenco delle partite giocate e analizzarle in base alla loro tipologia. Analogamente, l'utente può visualizzare il numero di partite vinte per tipo.

Gestione delle Amicizie:

Il sistema offre funzionalità per gestire le relazioni sociali. L'utente può infatti consultare la propria lista di amici, stringere nuove amicizie o eliminare amici dalla lista.

Relazioni tra le Funzionalità

Le varie funzionalità del sistema sono interconnesse attraverso relazioni specifiche:

Alcune operazioni sono indicate come "incluse" in altre, il che significa che rappresentano azioni obbligatorie o fondamentali per il funzionamento del sistema. Altre, invece, sono considerate estensioni opzionali che ampliano le funzionalità principali, come la visualizzazione dei dettagli aggiuntivi o l'interazione con elementi specifici.

Tutto il diagramma è organizzato all'interno di un'area denominata "Profilo Utente", che rappresenta la categoria generale delle funzionalità descritte. Questo layout consente di comprendere chiaramente le interazioni tra l'utente e le varie funzioni, fornendo una visione d'insieme delle possibilità offerte dal sistema.

1.3 Scenari

1.3.1 Caso d’Uso: Gestione Informazioni Profilo

Campo	Descrizione
Attore Primario	<i>Utente registrato</i>
Attore Secondario	<i>Nessuno</i>
Descrizione	Il sistema permette all’utente di modificare e visualizzare le informazioni presenti nel suo profilo.
Pre-Condizioni	L’utente deve essere registrato.
Sequenza Eventi Principale	<ol style="list-style-type: none"> 1. L’utente accede alla sezione "Profilo". 2. Il sistema visualizza una panoramica delle informazioni attuali del profilo. 3. L’utente sceglie di modificare i dati personali, cambiare avatar, foto o aggiornare la biografia. 4. Il sistema fornisce i moduli necessari per effettuare le modifiche. 5. L’utente conferma le modifiche effettuate. 6. Il sistema salva e aggiorna il profilo.
Post-Condizioni	Il database contiene i dati aggiornati dell’utente.
Casi d’Uso Correlati	<i>Modifica Dati Personalni, Modifica Avatar/Foto.</i>

Table 1.1: Gestione Informazioni Profilo

Caso d'Uso: Visualizzazione Badge

Campo	Descrizione
Attore Primario	<i>Utente registrato</i>
Attore Secondario	<i>Nessuno</i>
Descrizione	Il sistema permette all'utente di visualizzare i badge ottenuti e quelli da sbloccare.
Pre-Condizioni	<ul style="list-style-type: none"> • L'utente deve essere registrato e autenticato. • Gli achievement devono essere configurati nel sistema. • Un database degli achievement deve essere attivo.
Sequenza Eventi Principale	<ol style="list-style-type: none"> 1. L'utente accede alla sezione "Achievement". 2. Il sistema visualizza gli achievement sbloccati e quelli ancora da raggiungere.
Post-Condizioni	<ul style="list-style-type: none"> • Gli achievement sbloccati vengono aggiornati nel profilo.
Casi d'Uso Correlati	<i>Visualizzazione Achievement da Sbloccare, Visualizzazione Achievement Raggiunti</i>

Table 1.2: Visualizzazione Badge

Caso d'Uso: Visualizzazione Statistiche di Gioco

Campo	Descrizione
Attore Primario	<i>Utente registrato</i>
Attore Secondario	Nessuno
Descrizione	L'utente può visualizzare le sue statistiche di gioco, incluso il numero totale di partite giocate e le partite vinte, per tipologia.
Pre-Condizioni	<ul style="list-style-type: none"> • L'utente deve essere registrato e autenticato nel sistema. • Il database delle statistiche deve essere attivo e funzionante.
Sequenza Eventi Principale	<ol style="list-style-type: none"> 1. L'utente accede alla sezione "Statistiche di Gioco". 2. Il sistema visualizza un riepilogo che include: <ul style="list-style-type: none"> • Il numero totale di partite giocate. • Il numero totale di partite vinte. • La distribuzione delle statistiche per tipologia.
Post-Condizioni	<ul style="list-style-type: none"> • L'utente ottiene una visualizzazione aggiornata delle proprie statistiche di gioco.
Casi d'Uso Correlati	Visualizzazione Partite Giocate, Visualizzazione Partite Vinte.

Table 1.3: Visualizzazione Statistiche di Gioco

Caso d'Uso: Gestione Amicizie

Campo	Descrizione
Attore Primario	Utente registrato
Attore Secondario	Nessuno
Descrizione	L'utente può gestire le amicizie, visualizzando la lista degli amici, inviando richieste e accettando o rifiutando richieste pendenti.
Pre-Condizioni	<ul style="list-style-type: none"> • L'utente deve essere registrato e autenticato.
Sequenza Eventi Principale	<ol style="list-style-type: none"> 1. L'utente accede alla sezione "Amici". 2. Il sistema visualizza: <ul style="list-style-type: none"> • La lista degli amici attuali, con relativo tasto 'Elimina'. • La casella di ricerca di nuovi amici. 3. L'utente può: <ul style="list-style-type: none"> • stringere nuove amicizie • eliminare amicizie. 4. Il sistema aggiorna la relazione d'amicizia nel database.
Post-Condizioni	<ul style="list-style-type: none"> • La lista degli amici viene aggiornata. • Le richieste gestite cambiano stato (accettate o rifiutate).
Casi d'Uso Correlati	Visualizza Lista Amici, Elimina Amico, Aggiungi Amico

Table 1.4: Gestione Amicizie

1.3.2 Suddivisione del lavoro

Questo lavoro è stato preceduto, come già evidenziato in precedenza, da una revisione del lavoro dei nostri colleghi che hanno integrato i vari componenti dell'architettura globale e documentato tutto. Inizialmente il lavoro è stato suddiviso nei seguenti task con corrispondenti stime di persone necessarie per portare ciascuno a completamento, nelle tre settimane dateci per portare a termine il lavoro:

- Task per la prima iterazione: – Analisi dell'architettura di partenza, partendo dalla documentazione generale del progetto. – Analisi del Task specifico (R1) mediante studio della relativa documentazione.
- Task per la seconda iterazione: – Modifiche al diagramma dei casi d'uso per far fronte alle nostre esigenze. – Studio dei task precedenti, al fine di capire come implementare le funzionalità già implementate (per valutare la coerenza). – Realizzazione di diagrammi di sequenza al fine di documentare le scelte che si stessero intraprendendo.
- Task per la terza iterazione: – Realizzazione del component diagram. - Realizzazione incrementale della sezione profilo.

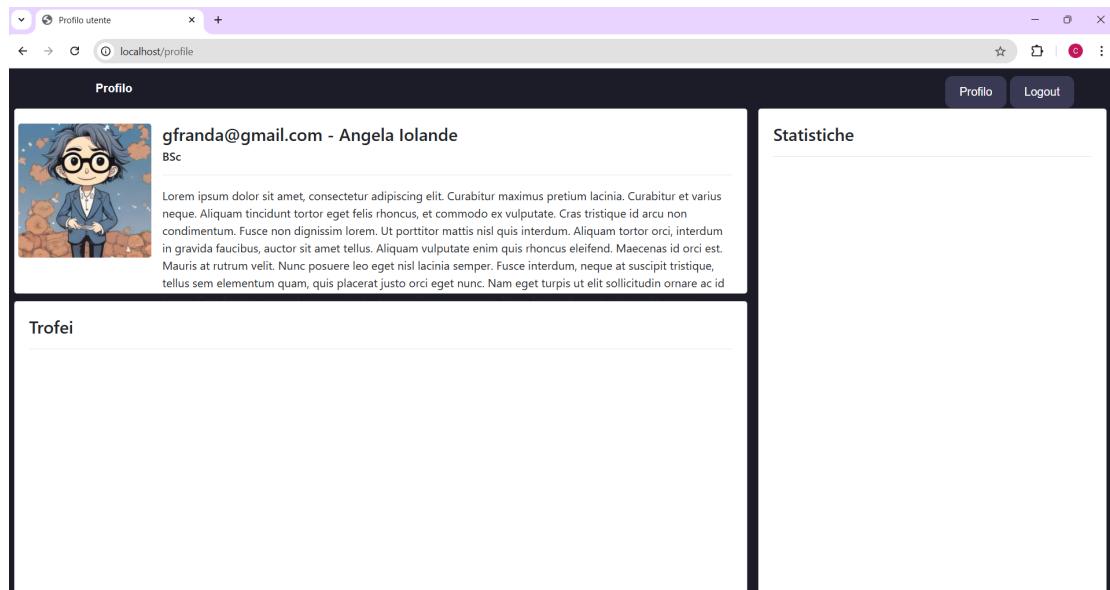
1.3.3 Stato iniziale del progetto

Per comprendere appieno il lavoro che abbiamo svolto, è utile considerare ciò che abbiamo "ereditato" dai colleghi che hanno avviato il progetto: approfondiremo questo discorso citando documentazioni utili per studenti futuri. Prima di entrare nel merito del progetto stesso, è

necessario soffermarsi su un aspetto fondamentale: l'installazione del programma. Questa fase si è rivelata piuttosto complessa. Già dalle prime fasi, ci siamo resi conto che le guide fornite non erano sufficientemente dettagliate, il che ha reso questa fase preliminare particolarmente difficile, sebbene siamo riusciti, fortunatamente, a superare tali difficoltà.

Un secondo ostacolo è stato quello di dover comprendere a pieno il modo in cui prelevare le informazioni necessarie al profilo e quindi studiare le modalità di interfaccia tra i vari microservizi. Fondamentale in questo senso è stato lo studio delle documentazioni più vicine al nostro workspace.

Il vecchio Profilo Utente



La pagina del profilo, prima delle modifiche, si presentava in modo estremamente rudimentale e poco curato. Era una struttura stat-

ica, priva di interattività e con un design datato e poco intuitivo. L’interfaccia era caratterizzata da un’estetica minimale, ma non nel senso moderno del termine: mancava di coerenza visiva, con un layout disorganizzato e poco accattivante. Inoltre, non offriva funzionalità utili per l’utente, limitandosi a mostrare informazioni basilari senza alcuna possibilità di personalizzazione o interazione. In sostanza, risultava poco funzionale e incapace di soddisfare le esigenze degli utenti moderni, trasmettendo un’impressione di trascuratezza.

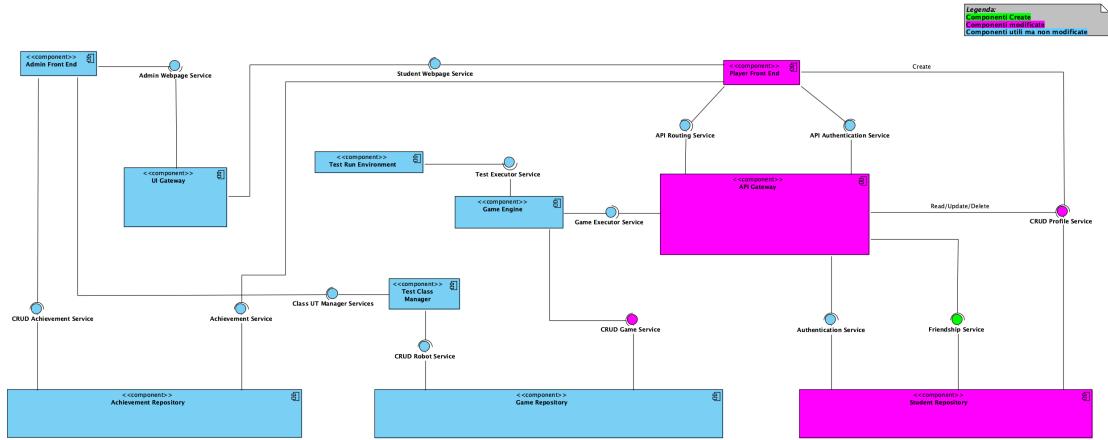
Oltre all’implementazione delle nuove funzionalità, abbiamo quindi lavorato intensamente sull’aspetto grafico e sull’interattività della pagina. L’obiettivo è stato quello di trasformare un’interfaccia statica e poco accattivante in una piattaforma moderna e dinamica, capace di rispondere alle esigenze degli utenti. Abbiamo rivisto completamente il design, puntando su un layout organizzato e coerente, e abbiamo integrato elementi interattivi che migliorano l’usabilità complessiva. Questo approccio ha permesso di creare una pagina più attraente, funzionale e coinvolgente.

1.3.4 Componenti e Connitori

Al fine di comprendere al meglio le parti del software d’interesse al nostro task, offriamo una panoramica generale del sistema attraverso il diagramma dei Componenti e dei Connettori, mostrato in Figura. Prendendo come riferimento questo diagramma, il nostro lavoro si è

concentrato sui componenti utili per la gestione del profilo.

Analiziamo tali componenti nel dettaglio:



Admin Front End: È un componente che utilizza l’interfaccia Admin Features, esposta dalla UI Gateway. Questo componente permette agli amministratori di accedere a una dashboard personalizzata con funzionalità specifiche, esclusivamente pensate per loro.

UI Gateway: Funziona come punto di ingresso per collegare il frontend con i vari servizi del backend. Fornisce supporto sia per l’interfaccia Admin Features sia per altre funzionalità. È un punto centrale per l’instradamento delle richieste agli altri componenti e per la gestione delle comunicazioni.

Test Run Environment: È il componente che permette l’esecuzione dei test, supportando i servizi di test durante le varie partite o esecuzioni simulate nel sistema.

Game Engine: È responsabile dell’implementazione delle logiche di gioco, collegandosi con i servizi di esecuzione dei test (Test Executor

Service) e di gestione dei dati di gioco (CRUD Game Service).

Test Class Manager: Questo componente gestisce l'esecuzione del testing per i diversi livelli di difficoltà. Si interfaccia con il Test Run Environment per garantire che i test siano condotti correttamente.

API Gateway: Funziona da punto di accesso centrale al sistema backend, gestendo richieste verso servizi specifici come: **API Routing Service** e **API Authentication Service**. Espone inoltre le interfacce richieste dai frontend, come Player Features per i giocatori e altre funzionalità per l'amministratore.

Player Front End: Componente dedicato all'interazione dell'utente/giocatore. Questo si connette all'API Gateway per accedere ai servizi di registrazione, autenticazione e alle funzionalità di gioco.

CRUD Profile Service: Fornisce funzionalità per la gestione del profilo degli utenti. Permette operazioni di Create, Read, Update, Delete sui dati relativi ai profili.

CRUD Achievement Service: È responsabile della gestione dei risultati e delle metriche di successo dei giocatori, come achievements o trofei.

Achievement Repository: Un repository utilizzato per salvare i risultati o i progressi degli utenti, rendendo possibile un monitoraggio dettagliato e permanente.

Friendship Service: Offre una funzionalità sociale, come la gestione di amicizie tra utenti. Supporta le interazioni tra i profili nel contesto

del gioco.

Student Repository: Memorizza i dati relativi ai progressi degli studenti, incluse le partite giocate e altre informazioni rilevanti.

CRUD Game Service: Consente operazioni CRUD sui dati dei giochi e si collega direttamente al Game Repository.

Game Repository: È il database dedicato alla memorizzazione delle informazioni sui giochi e sui risultati dei test, garantendo persistenza e accesso rapido.

Chapter 2

Strumenti Utilizzati

Di seguito gli strumenti che ci hanno consentito di svolgere i nostri compiti.

2.0.1 Microsoft Teams

Microsoft Teams è una piattaforma versatile progettata per gestire classi virtuali e facilitare la comunicazione tra utenti attraverso chiamate e messaggi. Durante lo sviluppo degli artefatti, si è rivelata uno strumento essenziale per garantire un'interazione fluida. Le riunioni in videochiamata hanno permesso di affrontare discussioni approfondite e mirate. In sintesi, la piattaforma ha svolto un ruolo fondamentale.

2.0.2 Github

GitHub è una piattaforma di hosting per lo sviluppo software che si basa su Git, un sistema distribuito per il controllo delle versioni. Oltre

alle funzionalità native di Git, GitHub offre strumenti aggiuntivi per la gestione dei progetti e la collaborazione tra sviluppatori, facilitando il monitoraggio delle modifiche al codice, la gestione delle issue e la condivisione del lavoro. È stato uno strumento essenziale per supportare lo sviluppo parallelo grazie a funzionalità come:

Repository: Contenitori utilizzati per archiviare tutti i file di un progetto, inclusi il codice sorgente, la documentazione e altri file di supporto. I repository possono essere impostati come pubblici o privati, a seconda delle esigenze del team.

Branch: Permettono di creare versioni separate del codice all'interno di un repository. Questo consente agli sviluppatori di lavorare su nuove funzionalità o correzioni senza alterare il codice principale.

Pull Request: Strumento per proporre modifiche al codice sorgente. Le pull request permettono di avviare discussioni, revisionare modifiche e decidere se integrarle nel ramo principale del progetto.

Collaborazione: GitHub offre numerosi strumenti per favorire il lavoro di squadra, come la possibilità di commentare il codice, discutere sulle issue, ricevere notifiche e utilizzare menzioni per una comunicazione più immediata.

Queste funzionalità hanno reso GitHub una risorsa chiave per la gestione del progetto e la collaborazione efficace.

2.0.3 Visual Paradigm

Visual Paradigm è stato uno strumento cruciale per il nostro progetto, poiché ha permesso creare i nuovi diagrammi.

Questa applicazione è una suite avanzata pensata per supportare le diverse fasi del ciclo di vita dello sviluppo software, dalla progettazione e modellazione fino all'implementazione e alla manutenzione. Visual Paradigm è ampiamente utilizzato nell'ingegneria del software per la sua capacità di gestire progetti complessi grazie alle sue numerose funzionalità, tra cui:

Modellazione Visuale: Offre strumenti per creare rappresentazioni visive dei sistemi software, utilizzando notazioni standard come UML (Unified Modeling Language), che consentono di descrivere concetti e relazioni all'interno del sistema.

Diagrammi UML: Supporta una vasta gamma di diagrammi, inclusi quelli per casi d'uso, classi, sequenze e attività. Questi diagrammi sono essenziali per rappresentare in modo chiaro e dettagliato diversi aspetti del sistema software.

Visual Paradigm si è dimostrato uno strumento versatile e indispensabile per documentare, progettare e gestire in maniera efficiente i processi di sviluppo del software.

2.1 Tecnologie utilizzate

Per sviluppare il nostro progetto, ci siamo concentrati su strumenti e tecnologie in grado di gestire efficacemente l'integrazione tra le diverse parti del sistema, caratterizzate da notevole eterogeneità. Questo ci ha dato l'opportunità di apprendere nuovi strumenti e tecniche, in particolare legati a framework e linguaggi di programmazione.

VS Code

Visual Studio Code è un editor di codice sorgente estremamente popolare grazie alla sua flessibilità e leggerezza. Non è solo un editor, ma offre anche numerose funzionalità avanzate che lo rendono un ambiente di sviluppo completo. Ecco alcune delle caratteristiche principali:

Prestazioni elevate: La leggerezza di VS Code lo rende veloce e adatto a qualsiasi tipo di hardware, offrendo un'esperienza utente fluida anche in contesti di sviluppo complessi.

Personalizzazione estrema: Grazie al Marketplace delle estensioni, VS Code può essere adattato a qualsiasi esigenza di sviluppo, supportando una vasta gamma di linguaggi di programmazione e framework.

Strumenti integrati: VS Code include un terminale, strumenti per il debug e un'integrazione nativa con Git, che semplifica il controllo di versione e migliora il flusso di lavoro dello sviluppatore.

Java Spring e Spring Boot

Java Spring è un framework open-source che semplifica la creazione di applicazioni Java, in particolare per l'ambito Enterprise. Questo strumento consente agli sviluppatori di concentrarsi sulla logica delle applicazioni, fornendo un'infrastruttura che gestisce molti aspetti tecnici. Le sue principali caratteristiche includono:

Supporto ai design patterns: Java Spring incoraggia l'uso di modelli architetturali standard, come il pattern Model-View-Controller (MVC), particolarmente utile nello sviluppo di applicazioni web scalabili.

Facilità di integrazione: Il framework è compatibile con tecnologie come JDBC e JMS, rendendo semplice lavorare con database e middleware.

Testing semplificato: Grazie al supporto per oggetti mock e alla struttura orientata all'interfaccia, Spring facilita la creazione di test automatizzati di qualità.

Spring Boot, una versione estesa di Spring, è progettato per accelerare lo sviluppo di applicazioni. Tra i suoi punti di forza:

Server integrati: Fornisce server preconfigurati come Tomcat, eliminando la necessità di configurazioni complesse.

Progetti preconfigurati: Include modelli per iniziare rapidamente nuovi progetti, come applicazioni web o basate sui dati.

Configurazione automatica: Riconosce le dipendenze e con-

figura automaticamente il sistema, risparmiando tempo agli sviluppatori.

2.1.1 Maven

Maven è un tool avanzato per il project management e la build automation, progettato specificamente per i progetti Java. Questo strumento fornisce un approccio standardizzato per gestire ogni fase del ciclo di vita di un progetto software, dalla scrittura del codice alla sua compilazione, esecuzione dei test, distribuzione e gestione delle dipendenze.

Ecco alcune delle sue caratteristiche principali:

Gestione automatizzata delle dipendenze: Maven utilizza un file di configurazione chiamato pom.xml (Project Object Model) per definire le dipendenze necessarie al progetto. Una volta configurato, Maven si occupa di scaricare automaticamente queste dipendenze dal Maven Central Repository o da repository specifici, eliminando la necessità di gestire manualmente i pacchetti.

Ciclo di vita standardizzato: Maven suddivide il ciclo di sviluppo in fasi ben definite come:

Clean: Pulizia dei file generati da build precedenti.

Compile: Compilazione del codice sorgente.

Test: Esecuzione dei test unitari.

Package: Creazione del pacchetto (es. file JAR o WAR).

Install: Installazione del pacchetto nel repository locale.

Deploy: Distribuzione del pacchetto in un repository remoto. Questo approccio strutturato facilita l'organizzazione del lavoro e garantisce un flusso di sviluppo coerente.

Accesso al Maven Central Repository: Maven si integra con un repository centralizzato, che contiene una vasta collezione di librerie e framework Java. Durante il processo di build, Maven scarica automaticamente le librerie necessarie, assicurandosi che siano sempre aggiornate e disponibili.

Maven rappresenta uno strumento indispensabile per i progetti Java, grazie alla sua capacità di automatizzare le operazioni ripetitive e gestire in modo efficiente le dipendenze, semplificando significativamente il processo di sviluppo e distribuzione del software.

2.1.2 Postman

Postman è uno degli strumenti più diffusi per lo sviluppo e la gestione di API, ampiamente utilizzato per testare, documentare e collaborare nello sviluppo delle interfacce di programmazione. Grazie alla sua interfaccia intuitiva e alle sue potenti funzionalità, rappresenta una risorsa essenziale per gli sviluppatori che lavorano con API. Ecco alcune delle caratteristiche principali:

Sviluppo e Debugging: Postman consente agli sviluppatori di inviare rapidamente richieste HTTP alle API, monitorare le risposte e verificare che le risorse funzionino come previsto. È particolarmente

utile per il debugging, poiché permette di individuare e correggere eventuali problemi in tempo reale.

Testing Completo delle API: Grazie a Postman, gli sviluppatori possono inviare richieste GET, POST, PUT, DELETE, e altro, ricevendo le risposte direttamente nell'applicazione. Questo consente di esaminare in dettaglio i dati di risposta, validare la correttezza delle risorse e garantire il corretto funzionamento delle API.

Documentazione delle API: Postman offre strumenti dedicati per generare documentazione chiara e dettagliata. La documentazione automatizzata aiuta sviluppatori e utenti a comprendere come interagire con le API, migliorando la comunicazione e riducendo le possibilità di errore.

Postman è quindi un tool versatile che semplifica lo sviluppo e il testing delle API, facilitando il lavoro in team e garantendo un'elevata qualità del software.

Chapter 3

Moduli Utilizzati e le loro interazioni

I microservizi si basano su un'architettura **MVC (Model-View-Controller)** che fornisce un'organizzazione modulare, scalabile e facilmente manutenibile. Questa struttura permette una separazione chiara delle responsabilità tra gestione dei dati, logica di business e presentazione all'utente, favorendo flessibilità ed estensibilità del sistema.

Il modello MVC si suddivide in tre componenti principali:

- **Model:** Gestisce i dati e la logica di business associata. Incapsula l'accesso alle informazioni e garantisce la loro integrità.
- **View:** Si occupa della presentazione grafica e dell'interazione con l'utente. Mostra i dati forniti dal Controller senza accedere direttamente al Model.

- **Controller:** Funziona come intermediario tra il Model e la View, elaborando le richieste utente e coordinando le risposte del sistema.

Questa separazione permette di modificare uno dei livelli (es. la View) senza dover alterare gli altri, garantendo così la *modularità* dell'applicazione.

Il task assegantoci richiede di intervenire sui microservizi T23,T5,T1 e T4. Di seguito presenteremo la sua struttura per una maggiore comprensione :

3.0.1 T1

- T1 per la sezione a noi di interesse si occupa della gestione e creazione degli achievement e utilizza MongoDB come database.

3.0.2 T4

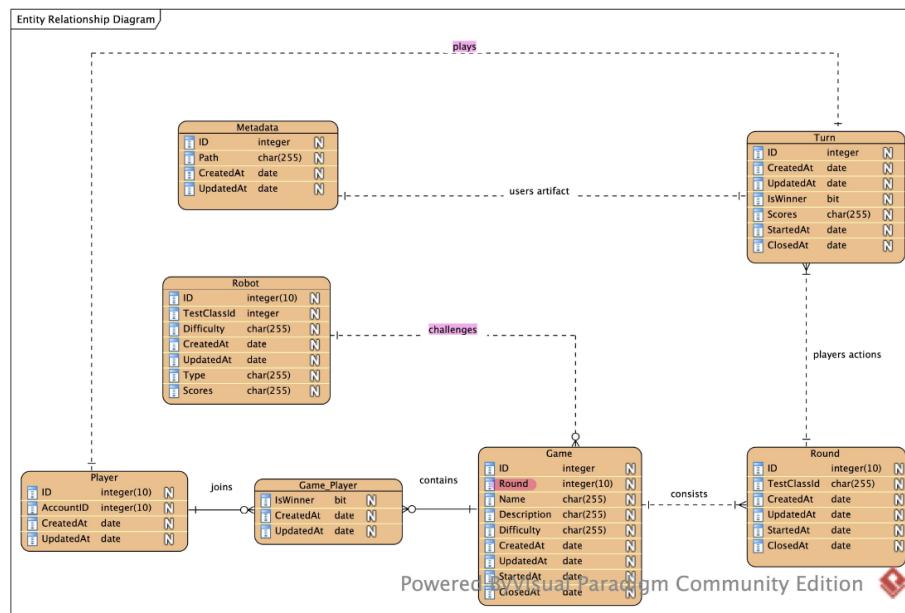


Figure 3.1: Diagramma ER T4

- T4 utilizza Postgre come database e si occupa della gestione delle partite del player.

3.0.3 T5

Ora passiamo ai moduli che in particolar modo hanno interessato il nostro lavoro e che hanno subito particolari modifiche:

- Il modulo T5 rappresenta il modulo centrale, responsabile di diverse funzionalità, tra cui la costruzione della pagina profilo di un giocatore. Per ottenere i dati necessari alla generazione di questa pagina, il modulo T5 interagisce con diversi microservizi

attraverso un sistema di servizi intermedi. Il controllo della logica applicativa è affidato al GuiController, che utilizza specifiche interfacce per comunicare con gli altri moduli del sistema.

Funzionalità del modulo T5

Per poter creare la pagina HTML del profilo del giocatore, il T5 deve compiere le seguenti azioni:

1. Otttenere i dati sugli achievement e le statistiche esistenti attraverso il microservizio T1.
1. Recuperare informazioni personali e la lista dei giocatori seguiti dal microservizio T23.
1. Accedere alle statistiche di gioco del giocatore, che vengono correlate ai dati forniti da T1, tramite il microservizio T4.

La comunicazione con questi microservizi è gestita tramite appositi servizi denominati *T1Service*, *T23Service* e *T4Service*, che fungono da intermediari tra il GuiController e i controller degli altri microservizi.

Il GuiController è il punto centrale di accesso alle funzionalità del modulo T5 e agisce come mediatore tra il frontend e i microservizi esterni. Questa scelta progettuale risponde a tre esigenze principali:

- Tutta la logica di integrazione è concentrata nel backend, riducendo la complessità del codice nel frontend e migliorando la manutenzione.

- Mantenendo il controllo delle richieste nel backend, si evita l'esposizione delle API direttamente al client. Questo approccio protegge il sistema da accessi non autorizzati e tentativi di manipolazione tramite il codice JavaScript.
- Adattabilità: Eventuali modifiche ai microservizi non richiedono interventi sui file frontend, ma solo nel backend, semplificando aggiornamenti e integrazioni future.

Modifiche strutturali Abbiamo introdotto alcune modifiche al sistema per migliorare la gestione dei profili e delle interazioni con gli altri microservizi:

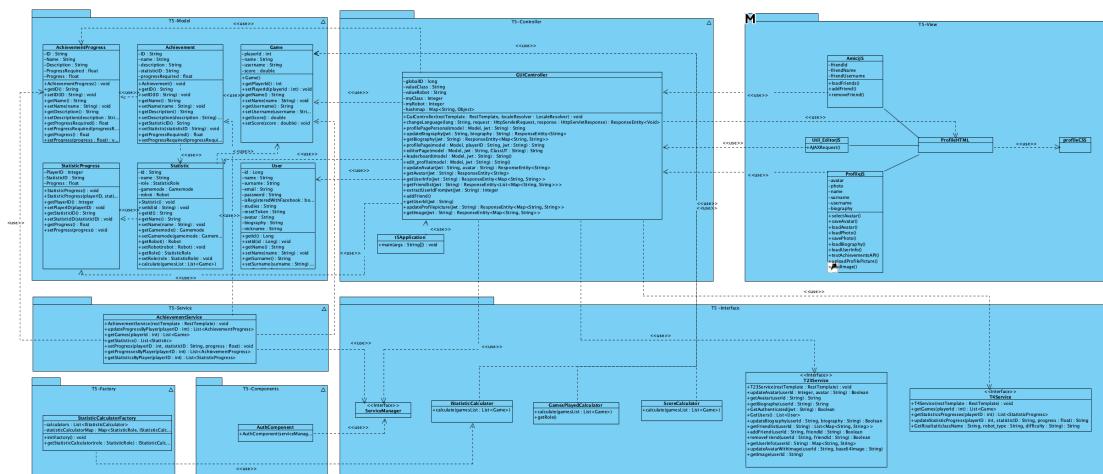
- Aggiornamento del GuiController: sono state integrate nuove funzionalità per gestire in modo più efficiente i dati relativi ai giocatori.
- Estensione del modello User: il modello è stato arricchito per supportare nuovi campi utili alla gestione dei dati personali e delle relazioni tra giocatori.

Aggiunta del file amici.js

Un'importante novità riguarda l'introduzione del file amici.js, un modulo JavaScript dedicato alla gestione delle amicizie. Questo file è stato progettato per occuparsi di:

- Visualizzazione della lista degli amici seguiti.
- Ricerca, Aggiunta e rimozione di giocatori alla lista degli amici seguiti.

Il modulo T5 si posiziona come un nodo strategico dell'architettura della Web Application. Grazie all'accenramento della logica nel GuiController, si garantisce una maggiore sicurezza e semplicità di gestione. Analizziamo il diagramma delle classi per il modulo T5 .



Struttura del Modulo T5

Il modulo T5 si suddivide in più livelli, ciascuno con un ruolo specifico.

Di seguito viene descritta l'organizzazione dettagliata:

1. Livello Model (T5-Model)

Il livello Model rappresenta il cuore logico del sistema, modellando le entità principali e le loro relazioni. Le classi più importanti sono:

- **User:** Rappresenta l'utente del sistema, con attributi come `id`, `name`, `surname`, `avatar` e `biography`.
- **Game:** Modella un gioco, con proprietà come `id`, `name`, `score` e `timePlayed`.

- **Statistic:** Gestisce le statistiche dei giochi, come punteggi e tempi di gioco.
- **Achievement:** Rappresenta obiettivi o traguardi raggiungibili dagli utenti.
- **Progress Classes** (`AchievementProgress`, `StatisticProgress`): Tengono traccia dello stato dell'utente rispetto agli obiettivi e alle statistiche.

Le relazioni tra le classi del Model permettono una gestione coerente delle entità:

- Un `User` può giocare a uno o più `Game`.
- Un `Game` è associato a diverse `Statistic`.
- Le `Statistic` possono contribuire al raggiungimento di uno specifico `Achievement`.

2. Livello View (T5-View)

Il livello View è responsabile della presentazione all'utente e dell'interazione grafica. Nel modulo T5, la View è composta da:

- **HTML:** Definisce la struttura delle pagine (es. `ProfileHTML` per la pagina del profilo).
- **CSS:** Uniforma lo stile visivo (es. `profileCSS` per la formattazione del profilo).

- **JavaScript:**
 - ProfileJS: Gestisce funzionalità dinamiche come caricamento avatar e biografia.
 - AmiciJS: Permette la gestione della lista amici, come aggiunta o rimozione.

La View si interfaccia con il Controller tramite richieste AJAX, tecnica utilizzata per aggiornare parti di una pagina web senza dover ricaricare l'intera pagina. Di fatti, permette di inviare richieste al server in modo asincrono e ottenere i dati di risposta, aggiornando lo stato del sistema garantendo una presentazione reattiva e dinamica.

3. Livello Controller (T5-Controller)

Il livello Controller, centralizzato nella classe `GUIController`, funge da punto di controllo per tutte le operazioni principali. Si occupa di:

- **Gestione delle richieste utente:** Riceve e valida le richieste HTTP, come aggiornamenti di avatar o aggiunta di amici.
- **Interazione con il livello Service:** Delega operazioni complesse, come il calcolo delle statistiche o la gestione degli obiettivi, ai servizi appropriati.
- **Restituzione dei dati alla View:** Fornisce risposte formattate alla View, spesso tramite oggetti `ResponseEntity`.

Questa centralizzazione migliora la *sicurezza* e facilita la manutenzione.

4. Livello Service (T5-Service)

Il livello Service implementa la logica di business dell'applicazione. I servizi principali includono:

- **AchievementService**: Gestisce i progressi degli utenti verso il completamento degli obiettivi.
- **T5Service**: Coordina operazioni ad alto livello, come la gestione degli amici o l'aggiornamento degli avatar.

Questo livello isola la logica di business dal Controller, rendendo il sistema più modulare e manutenibile.

5. Factory e Interfacce (T5-Factory e T5-Interface)

- **StatisticCalculatorFactory**: Centralizza la creazione di calcolatori specializzati (es. ScoreCalculator, GamesPlayedCalculator) favorendo la scalabilità del sistema.
- **Interfacce**: Definiscono contratti chiari per le funzionalità, come T5Service Interface e StatisticCalculator, garantendo che ogni implementazione soddisfi i requisiti predefiniti.

3.0.4 Punti di Forza dell'Architettura

- **Separazione delle responsabilità:** Grazie all'architettura MVC, ogni componente è indipendente e facilmente modificabile.
- **Centralizzazione:** Il `GUIController` garantisce un controllo centralizzato delle operazioni, migliorando sicurezza e coerenza.
- **Modularità ed estensibilità:** La presenza di factory e interfacce consente di aggiungere nuove funzionalità senza impattare sulle altre parti del sistema.
- **Manutenibilità:** La separazione tra Model, View e Controller facilita interventi mirati e aggiornamenti del sistema.

3.0.5 T23

- Nel modulo T23 dedicato alla gestione dei giocatori registrati, è stata introdotta una modifica significativa alla struttura del database per implementare la logica delle relazioni di amicizia. Utilizzando MySQL come database, è stato aggiornato il modello relazionale per includere una nuova funzionalità che permette agli utenti di instaurare relazioni reciproche.

Nuova struttura per le amicizie

È stata aggiunta una nuova tabella denominata `user_friends`, separata dalla tabella principale degli utenti (`students`), per gestire in modo chiaro ed efficiente le relazioni di amicizia.

La tabella user_friends contiene due colonne principali:

- user_id: identifica l'utente che ha stretto l'amicizia.
- friend_id: rappresenta l'utente amico collegato a user_id.

Entrambi i campi sono chiavi esterne che puntano alla tabella students (basandosi sul campo id).

- La relazione tra gli utenti è simmetrica: se un utente A è amico di un utente B, viene registrata una riga che associa user_id con friend_id. Questo permette di ottenere cardinalità: Ogni utente (user_id) può avere zero o più amici (friend_id), quindi la relazione è di tipo (0, N), questo consente a ogni giocatore registrato di non avere alcun amico oppure di essere connesso con un numero illimitato di altri utenti.

Benefici del modello

La scelta di utilizzare una tabella separata per le relazioni di amicizia offre diversi vantaggi:

- *Semplicità*: Le relazioni di amicizia sono modellate chiaramente, senza sovraccaricare la tabella principale degli utenti.
- *Flessibilità*: La struttura consente di espandere facilmente il sistema, aggiungendo eventuali attributi aggiuntivi alle amicizie, come lo stato della relazione (ad esempio, “in attesa di conferma”).
- *Efficienza nelle query*: Separare la logica delle amicizie rende più semplici e veloci le query che riguardano gli amici di un utente.

Funzionalità del sistema

Questa modifica consente al sistema di:

- Aggiungere nuovi amici attraverso l'inserimento di una riga nella tabella user_friends.
- Recuperare la lista di amici di un determinato utente effettuando una query su user_friends filtrata per user_id.
- Gestire eventuali richieste di amicizia future o altre dinamiche social aggiuntive, mantenendo la scalabilità del database.

L'introduzione della tabella user_friends migliora l'organizzazione del database e semplifica la gestione delle relazioni tra giocatori registrati. Separando i dati degli utenti dalla logica delle amicizie, si garantisce una maggiore modularità e una più facile manutenzione del sistema.

Il modulo T23, oltre ad aver implementato la logica per le relazioni di amicizia tramite la nuova tabella user_friends, è stato ulteriormente ampliato con l'aggiunta di due file essenziali: FriendRepository.java e Friend.java. Questi file forniscono le funzionalità necessarie per la gestione delle amicizie nel database e per l'integrazione con il sistema Spring Boot.

Dettagli dei file aggiunti

1. **FriendRepository.java** Questo file rappresenta il repository JPA responsabile dell'interazione diretta con la tabella user_friends. Contiene i metodi per gestire le operazioni CRUD (Create, Read, Up-

date, Delete) legate alle amicizie, oltre ad altre operazioni personalizzate. Di seguito i punti principali:

Recupero dettagli amici: Tramite il metodo `findFriendDetailsById`, si ottiene una lista degli amici di un utente, includendo dettagli come nickname e avatar dalla tabella `students`.

Verifica dell'esistenza di un'amicizia: Utilizzando il metodo `existsFriendship`, è possibile verificare se due utenti hanno già una relazione di amicizia.

Aggiunta di un amico: Con il metodo `addFriend`, è possibile creare una nuova relazione di amicizia tra due utenti inserendo i dati nella tabella `user_friends`.

Rimozione di un amico: Il metodo `deleteFriend` consente di eliminare una relazione di amicizia esistente.

Questo repository utilizza annotazioni Spring come `@Query`, `@Modifying`, e `@Transactional` per definire query personalizzate, garantendo la corretta gestione delle transazioni durante le operazioni che modificano i dati.

2. Friend.java Questo file è l'entità JPA che rappresenta la tabella `user_friends` nel database. Fornisce una mappatura tra i campi della tabella e le proprietà della classe.

Annotazione `@Entity`: Specifica che questa classe è un'entità JPA e che i suoi campi corrispondono ai campi della tabella `user_friends`.

Oltre all'aggiunta di questi file erano già presenti file importanti

```
1 package com.example.db_setup;
2
3 import lombok.Data;
4
5 import javax.persistence.*;
6
7 @Entity
8 @Table(name = "user_friends", schema = "studentsrepo")
9 public class Friend {
10     @EmbeddedId
11     private FriendId id;
12
13     public Friend() {}
14
15     public Friend(FriendId id) {
16         this.id = id;
17     }
18
19     // Getter e Setter
20     public FriendId getId() {
21         return id;
22     }
23
24     public void setId(FriendId id) {
25         this.id = id;
26     }
27
28 }
```

Figure 3.2: Friend.java

modificati per la modifica della biografia, delle informazioni personali e della foto profilo:

User.java

Nel progetto, l'entità User e l'interfaccia UserRepository gestiscono la tabella students all'interno del database. Questi componenti sfruttano le funzionalità di **JPA (Java Persistence API)** e **Spring Data JPA** per mappare, manipolare e interrogare i dati in

CHAPTER 3. MODULI UTILIZZATI E LE LORO INTERAZIONI

maniera efficiente e organizzata.

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    // Metodi di ricerca
    User findByEmail(String email);
    User findByName(String name);
    User findByResetToken(String resetToken);
    User findById(Integer ID);
    User findByNickname(String nickname);
    User findByIsRegisteredWithFacebook(boolean isRegisteredWithFacebook);
    User findByIsRegisteredWithGoogle(boolean isRegisteredWithGoogle);

    // Metodi di ricerca avanzata
    List<User> findBySurname(String surname);
    List<User> findByBiographyContaining(String keyword);

    // Modifiche per avatar
    @Modifying
    @Transactional
    @Query("UPDATE User u SET u.avatar = :avatar,u.profilePicture = NULL WHERE u.ID = :userId")
    void updateAvatar(@Param("userId") Integer userId, @Param("avatar") String avatar);

    //@Query("SELECT u.avatar FROM User u WHERE u.ID = :userId")
    //String findAvatarByUserId(@Param("userId") Integer userId);

    //@Query("SELECT u.profilePicture FROM User u WHERE u.ID = :userId")
    //byte[] findProfilePictureByUserId(@Param("userId") Integer userId);

    @Modifying
    @Transactional
    @Query("UPDATE User u SET u.profilePicture = :profilePicture, u.avatar = NULL WHERE u.ID = :userId")
    void updateProfilePicture(@Param("userId") Integer userId, @Param("profilePicture") byte[] profilePicture);
}
```

Figure 3.4: FriendRepositoy

User Class

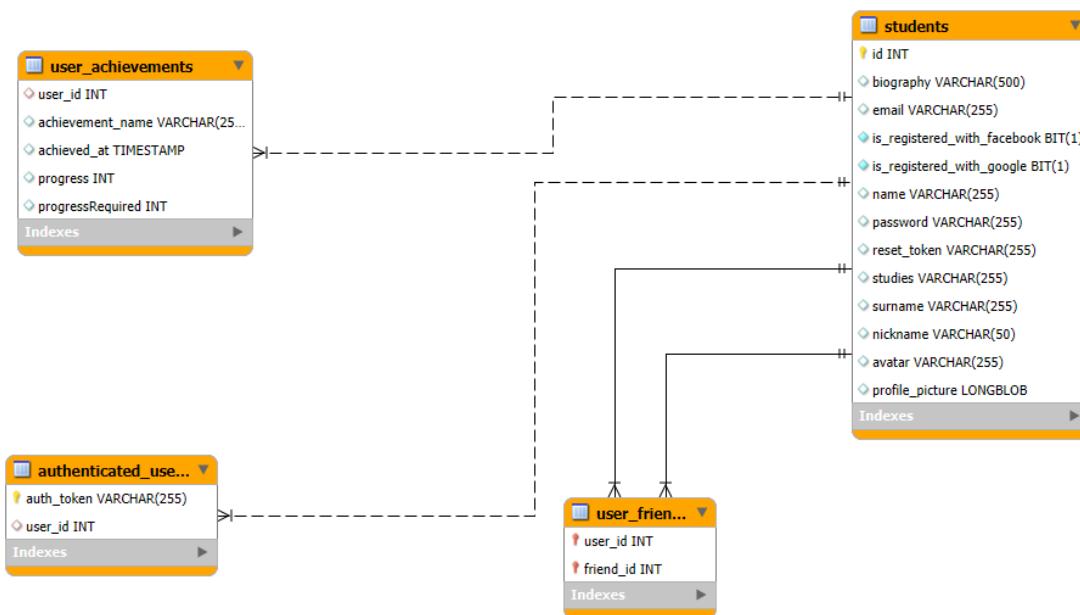
```
7  @Table(name = "students", schema = "studentsrepo")
8  @Data
9  @Entity
10 public class User {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.AUTO)
14     private Integer ID;
15
16     @Enumerated(EnumType.STRING)
17     public Studies studies;
18
19     @Column(name = "name", nullable = false, length = 100)
20     private String name;
21
22     @Column(name = "surname", nullable = false, length = 100)
23     private String surname;
24
25     @Column(name = "email", nullable = false, unique = true)
26     private String email;
27
28     @Column(name = "password", nullable = false)
29     private String password;
30
31     public boolean isRegisteredWithFacebook;
32     public boolean isRegisteredWithGoogle;
33
34     @Column(name = "nickname", unique = true, nullable = false, length = 50)
35     private String nickname;
36
37     @Column(name = "biography", length = 500)
38     private String biography;
39
40     @Column(name = "avatar", nullable = true, length = 255)
41     private String avatar;
42     You, 6 giorni fa • updateUserInfo
43     @Lob
44     @Column(name = "profile_picture")
45     private byte[] profilePicture;
```

Figure 3.5: User

La classe `User` è un’entità JPA che rappresenta la tabella `students` nel database. Ogni campo della classe è mappato a una colonna della tabella tramite annotazioni JPA. Ogni campo della classe è dotato di metodi getter e setter per l’accesso e la modifica dei dati. Questo approccio segue il principio di encapsulamento, garantendo una gestione sicura e controllata degli attributi.

`textttUserRepository.java` L’interfaccia `UserRepository` estende `JpaRepository` di Spring Data JPA. Essa fornisce un modo semplice e automatico per interagire con la tabella `students` del database.

3.0.6 Diagramma ER e Gestione delle Entità con JPA



Un diagramma ER (Entity-Relationship) è una rappresentazione visiva della struttura logica di un database relazionale. Viene utilizzato per descrivere le **entità** principali, gli **attributi** associati e le **relazioni** tra di esse. Ogni elemento del diagramma può essere tradotto in una struttura fisica del database e, nel caso di applicazioni Java, in classi mappate tramite **JPA (Java Persistence API)**.

Il diagramma ER facilita:

- La comprensione della struttura dei dati.

- La progettazione del database relazionale.
- L'implementazione del codice ORM (Object-Relational Mapping) per connettere il database con un'applicazione.

Componenti di un Diagramma ER

- **Entità:** Oggetti del mondo reale che vengono modellati nel database. Nel contesto JPA, ogni entità corrisponde a una classe annotata con `@Entity`.
- **Attributi:** Proprietà che descrivono un'entità. Corrispondono alle colonne di una tabella nel database e ai campi delle classi Java.
- **Relazioni:** Legami tra entità diverse. Possono essere:
 - **Uno-a-Uno (1:1).**
 - **Uno-a-Molti (1:N).**
 - **Molti-a-Molti (N:M).**

3.0.7 Introduzione a JPA (Java Persistence API)

La **JPA** è una specifica Java per la gestione della persistenza degli oggetti in un database relazionale. Consente di mappare classi Java e i loro attributi alle tabelle e colonne del database, eliminando la necessità di scrivere query SQL manualmente.

Caratteristiche Principali di JPA

- **Mappatura Oggetto-Relazionale (ORM):** Utilizza annotazioni come `@Entity`, `@Table`, `@Column` e `@Id` per collegare le classi Java alle tabelle.
- **Indipendenza dal Database:** JPA non dipende da un database specifico; funziona con provider ORM come **Hibernate**, **EclipseLink** o **OpenJPA**.
- **Gestione delle Relazioni:** Semplifica le relazioni tra tabelle (es. One-to-Many, Many-to-Many) utilizzando annotazioni come `@ManyToOne`, `@OneToMany`.
- **Query Object-Oriented:** Fornisce il linguaggio di query **JPQL** (Java Persistence Query Language) per scrivere query orientate agli oggetti.

Funzionamento di JPA Il processo JPA si articola in tre passaggi:

1. **Definizione delle Entità:** Si creano classi Java annotate per rappresentare le tabelle del database.
2. **Configurazione di Persistence Unit:** Tramite il file `persistence.xml` si configura l'unità di persistenza, indicando il database e il provider ORM.
3. **Interazione con il Database:** Utilizzando l'`EntityManager`, si eseguono operazioni CRUD (**Create, Read, Update, Delete**).

3.1 Descrizione delle Entità JPA e Relazioni

3.1.1 Entità **User**

L'entità **User** rappresenta la tabella `students` nel database MySQL e contiene le informazioni principali sugli utenti.

- **Chiave Primaria:**

- `ID`: Generata automaticamente tramite `@GeneratedValue`.

- **Campi principali:**

- `name`, `surname`: Campi obbligatori con lunghezza massima di 100 caratteri.
 - `email`: Campo univoco per identificare l'utente.
 - `password`: Campo per le credenziali.
 - `nickname`: Unico e obbligatorio, con lunghezza massima di 50 caratteri.
 - `avatar`, `profilePicture`: - `avatar` memorizza un percorso o URL. - `profilePicture` utilizza `@Lob` per memorizzare dati binari.

- **Campi Aggiuntivi:**

- `resetToken`: Campo opzionale per la gestione della password.

- `isRegisteredWithFacebook` e `isRegisteredWithGoogle`: Booleani che indicano la registrazione social.
- `studies`: Utilizza `@Enumerated(EnumType.STRING)` per mappare valori enumerati.

3.1.2 Entità **Friend**

L’entità **Friend** rappresenta la tabella `user_friends` utilizzata per gestire le relazioni di amicizia tra utenti.

- **Chiave Composta:**

- Definita tramite una classe `FriendId` annotata con `@Embeddable`.
- Contiene `user_id` e `friend_id`.

- **Relazione Auto-referenziata:**

- La relazione è di tipo **Molti-a-Molti**, poiché entrambi i campi puntano alla stessa tabella `students`.

- **Funzionalità principali:**

- Aggiunta di nuove amicizie.
- Recupero della lista degli amici con dettagli (nickname e avatar).
- Rimozione di una relazione di amicizia esistente.

3.1.3 Relazioni nel Diagramma ER

Le relazioni tra le entità **User** e **Friend** sono le seguenti:

- **Zero-a-Molti tra `students` e `user_friends`:**
 - Ogni utente può avere molte amicizie o nessuna, ognuna rappresentata da una riga nella tabella `user_friends`.

3.1.4 Benefici dell’Uso di JPA e Diagramma ER nel Progetto

L’adozione di JPA e la progettazione del database tramite un **diagramma ER** offrono diversi vantaggi:

- **Astrazione dal Database:** Il codice applicativo è indipendente dalla struttura del database grazie all’ORM.
- **Gestione delle Relazioni:** Relazioni complesse come Molti-a-Molti sono implementate con annotazioni semplici.
- **Manutenzione e Scalabilità:** Cambiamenti nella struttura del database possono essere riflessi facilmente nelle entità JPA.
- **Ottimizzazione delle Query:** Query JPQL migliorano la leggibilità e generano SQL ottimizzato.
- **Portabilità:** L’applicazione può utilizzare qualsiasi database relazionale supportato da JPA.

In conclusione, l'utilizzo del **diagramma ER** in combinazione con **JPA** permette di implementare in modo chiaro e robusto il database relazionale, semplificando l'integrazione con l'applicazione Java e garantendo una gestione efficiente delle entità e delle relazioni.

3.2 Deployment Diagram

Il Deployment Diagram è un diagramma UML che rappresenta la distribuzione fisica di componenti software su nodi hardware o virtuali (come server, dispositivi o container). Viene utilizzato per descrivere come le applicazioni, i moduli e i file (chiamati "artefatti") sono distribuiti nell'infrastruttura e come i nodi comunicano tra loro.

Obiettivo

- Mostrare come il software viene distribuito e eseguito su diverse piattaforme fisiche o virtuali.
- Rappresentare i dispositivi e i server coinvolti, gli artefatti distribuiti su di essi e i protocolli di comunicazione tra i nodi.

Elementi del Deployment Diagram

1. Nodi (**«device»** e **«executionEnvironment»**):

- Rappresentano entità fisiche (server, PC, dispositivi mobili) o ambienti di esecuzione virtuali (container Docker, ambienti JVM).
- Sono rappresentati come cubi o rettangoli con etichette come «device» o «executionEnvironment».

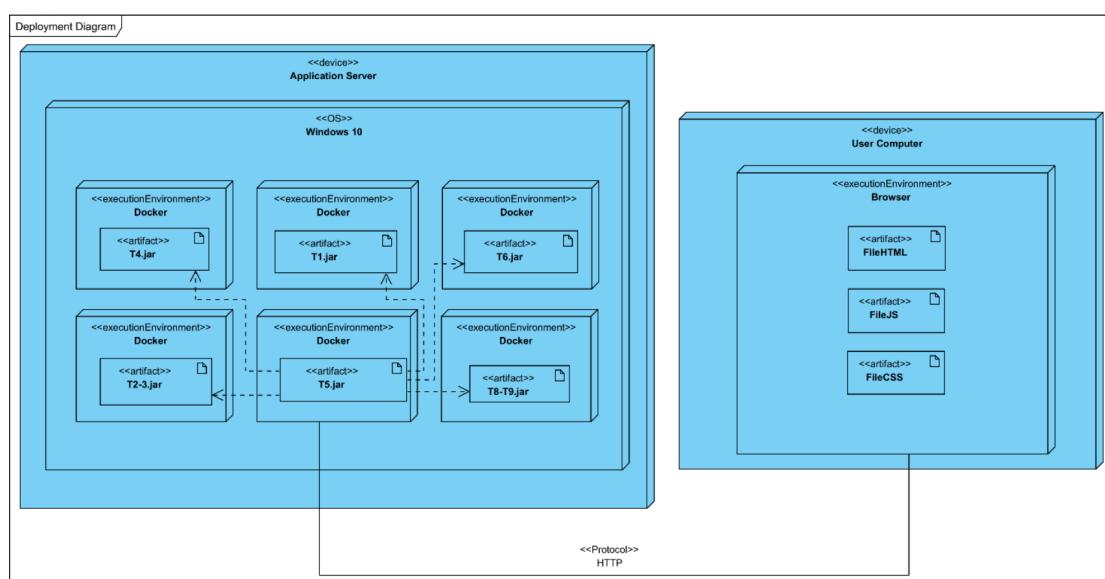
2. Artefatti («artifact»):

- Rappresentano file o componenti software distribuiti, come file .jar, .war, o script.

3. Connessioni e protocolli:

- Linee tra nodi che indicano la comunicazione o il flusso di dati, spesso annotate con il protocollo utilizzato (es. HTTP, TCP/IP).

3.2.1 Descrizione del Deployment



1. Application Server:

- Il diagramma mostra un server applicativo che utilizza Windows 10 come sistema operativo.
- Su questo server, diversi container Docker sono in esecuzione, ognuno dei quali ospita un artefatto software (file .jar):
 - T4.jar
 - T1.jar
 - T6.jar
 - T5.jar
 - T2-3.jar
 - T8-T9.jar.

2. User Computer:

- Un altro nodo è rappresentato dal computer utente, dove viene utilizzato un browser per interagire con l'applicazione.
- Gli artefatti disponibili sul computer utente sono:
 - FileHTML: Per la struttura delle pagine web.
 - FileJS: Per la logica client-side.
 - FileCSS: Per lo stile e la presentazione.

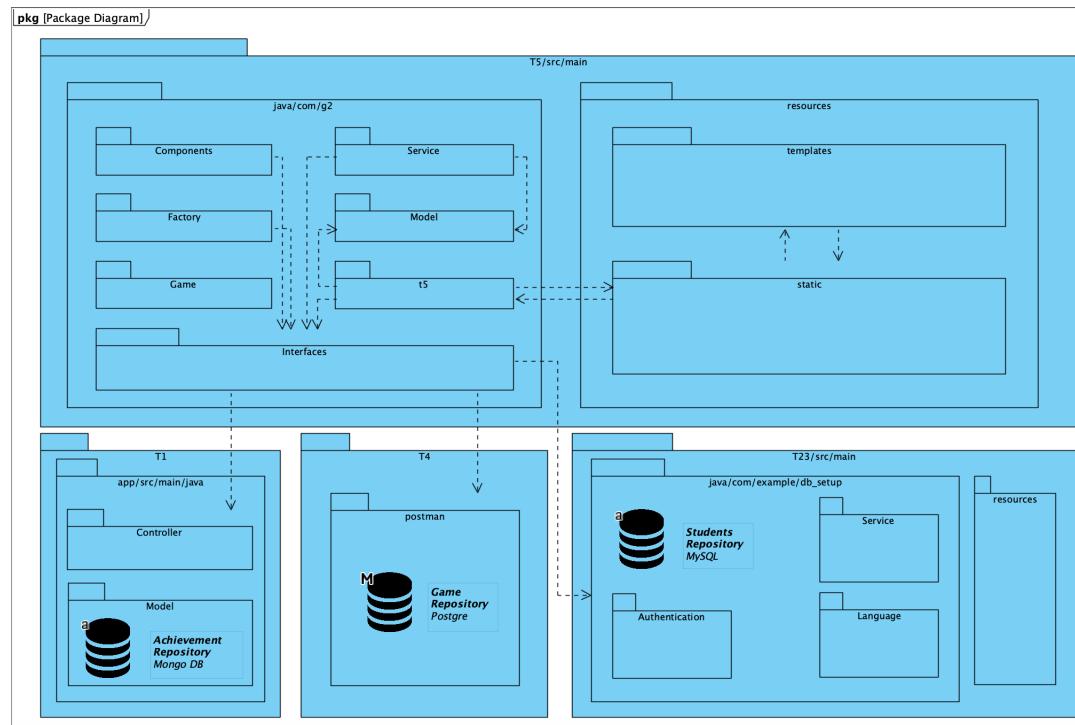
3. Connessioni:

- I nodi comunicano tra loro utilizzando il protocollo HTTP, ad esempio tra il server applicativo e il browser dell'utente.

3.2.2 Uso del Deployment Diagram

- Permette di capire come i componenti software (es. i moduli .jar) sono distribuiti nell'infrastruttura.
- Facilita la progettazione del sistema per garantire la scalabilità e l'affidabilità.
- È utile per descrivere l'interazione tra il backend e il frontend.

3.3 Package Diagram



Il Package Diagram è un diagramma UML che descrive l'organizzazione logica e le dipendenze tra i moduli o i pacchetti di un sistema software.

Ogni "package" rappresenta un'unità logica del sistema, come una libreria, un modulo o un namespace.

Obiettivo

- Mostrare la struttura logica del sistema e come i pacchetti/moduli interagiscono tra loro.
- Evidenziare le dipendenze tra i pacchetti, ad esempio quali moduli dipendono da altri per funzionare.

3.3.1 Elementi del Package Diagram

1. Pacchetti:

- Sono rappresentati come rettangoli con una linguetta in alto a sinistra, simile a una cartella.
- Ogni pacchetto rappresenta una sezione logica del sistema, come un modulo Java o un namespace.

2. Relazioni tra pacchetti:

- Le dipendenze tra i pacchetti sono rappresentate da frecce tratteggiate («import» o «use»), che indicano che un pacchetto utilizza classi o funzionalità definite in un altro.

3. Gerarchie:

- I pacchetti possono essere nidificati, rappresentando una struttura gerarchica del codice.

3.3.2 Descrizione del Package Diagram

1. Struttura Generale:

- Il diagramma rappresenta i pacchetti logici del sistema e mostra la loro organizzazione e dipendenza.
- Ogni modulo (es. T1, T2, T5, ecc.) è suddiviso in pacchetti che rappresentano la struttura del codice sorgente.

2. Pacchetti Principali:

- `java/com/g2`: Contiene pacchetti chiave come:
 - `Components`: Moduli riutilizzabili.
 - `Factory`: Per la creazione di oggetti.
 - `Service`: Per la logica di business.
 - `Model`: Contiene le entità del dominio.
 - `t5`: Modulo specifico che interagisce con altri pacchetti.
- `resources`:
 - Include cartelle come `templates` (file per il rendering delle pagine) e `static` (risorse statiche come CSS o immagini).

3. Moduli Indipendenti:

- T1:
 - Contiene un pacchetto Controller e un modello per il repository MongoDB (Achievement Repository).
- T4:
 - Gestisce il Game Repository connesso a un database PostgreSQL.
- T23:
 - Contiene pacchetti per autenticazione e il repository Students Repository basato su MySQL.

4. Relazioni tra Pacchetti:

- Il diagramma mostra le dipendenze tra i moduli:
 - Ad esempio, il modulo T5 dipende dai servizi definiti in `java/com/g2`.
 - T4 interagisce con il database PostgreSQL per il repository del game.

Uso del Package Diagram

- Permette di comprendere la struttura logica del progetto.

- Mostra come i diversi componenti collaborano tra loro a livello di codice.
- Evidenzia le dipendenze che potrebbero influenzare la modularità o la manutenibilità del sistema.

3.4 Interazione dei microservizi

I microservizi sono organizzati e distribuiti per garantire modularità, scalabilità e una chiara separazione delle responsabilità. Ogni microservizio gestisce una parte specifica del sistema e comunica con gli altri attraverso protocolli standard come HTTP o meccanismi di messaggistica.

3.4.1 Struttura Generale

1. Deployment:

- I microservizi sono distribuiti in container Docker, come mostrato nel Deployment Diagram.
- Ogni container ospita un modulo .jar (es. T1.jar, T2-3.jar, T5.jar, ecc.), che rappresenta un microservizio indipendente.

2. Comunicazione:

- I microservizi comunicano principalmente tramite HTTP.

- Alcuni microservizi sono connessi a database specifici (MongoDB, PostgreSQL, MySQL) per la persistenza dei dati.
- La comunicazione avviene secondo un'architettura RESTful, in cui ogni microservizio espone API specifiche.

Dettagli dei Microservizi

1. T23 (Students Repository)

- **Funzione principale:** Gestisce i dati degli studenti, rappresentando il cuore della gestione utente del sistema.
- **Database:**
 - T23 è collegato a un database MySQL, che contiene tutte le informazioni sugli studenti, inclusi:
 - * Nome, cognome, email e password.
 - * Informazioni sul profilo (avatar, biografia, nickname).
 - * Token per il recupero password.
 - * Registrazioni social (Google/Facebook).
- **Interazioni:**
 - **T5 (Frontend):**
 - * Fornisce i dati utente richiesti dal frontend, come dettagli del profilo, biografia, avatar e nickname.

- * Gestisce le richieste di aggiornamento dei dati degli studenti (es. modifica della biografia o reset della password).
- **T8-T9 (Authentication Service):**
 - * Valida i token di autenticazione generati da T8-T9.
 - * Consente a T8-T9 di accedere ai dati sensibili degli utenti (es. email e password hash).

2. T5 (Frontend)

- **Funzione principale:** Gestisce la comunicazione tra il frontend (browser dell'utente) e i microservizi backend.
- **Ruolo nel sistema:**
 - Serve come intermediario tra il browser dell'utente e i microservizi, come T23, T1.
 - Espone API RESTful che il frontend utilizza per inviare richieste e ricevere risposte.
 - Rende disponibili le risorse statiche per il frontend (File-HTML, FileJS, FileCSS) tramite un server web.
- **Interazioni:**
 - **T23 (Students Repository):**
 - * Inoltra richieste al T23 per gestire:
 - Visualizzazione dei dati del profilo dell'utente.

- Modifica delle informazioni personali.
- Registrazione e gestione del recupero password.
- **T8-T9 (Authentication Service):**
 - * Verifica i token di autenticazione per consentire l'accesso alle funzionalità protette.
- **T1 (Achievement Service):**
 - * Recupera gli achievement raggiunti da un utente e li visualizza nel frontend.

3. T1 (Achievement Service)

- **Funzione principale:** Gestisce gli achievement raggiunti dagli utenti nei giochi.
- **Database:**
 - Memorizza gli achievement degli utenti in un database MongoDB.
- **Interazioni:**
 - **T5 (Frontend Service):**
 - * Invia al frontend gli achievement dell'utente per la visualizzazione.

4. T4 (Game Repository)

- **Funzione principale:** Gestisce i dati statici dei giochi, come livelli e regole.

- **Database:**

- Memorizza i dati statici in un database PostgreSQL.

5. T8-T9 (Authentication Service)

- **Funzione principale:** Gestisce l'autenticazione degli utenti

- e la validazione dei token.

- **Interazioni:**

- **T23 (Students Repository):**

- * Accede ai dati degli utenti (come email e password hash) per autenticare le richieste.

- **T5 (Frontend Service):**

- * Genera e valida i token di autenticazione per consentire agli utenti di accedere ai propri account.

3.4.2 Flusso di Comunicazione Tra Microservizi

1. Accesso al Profilo Utente (Frontend → Backend):

- Il frontend (browser) invia una richiesta HTTP a T5 per visualizzare il profilo dell'utente.
- T5 inoltra la richiesta a T23 (Students Repository) per recuperare i dati del profilo.
- T23 restituisce i dati richiesti (es. nome, avatar, biografia) a T5, che li invia al frontend per la visualizzazione.

2. Progressi dell’utente:

- Il frontend invia una richiesta a T5 per registrare i progressi di un utente in un gioco.
- T5 inoltra la richiesta a T2-3 (Game Service).
- T2-3 aggiorna i progressi nel proprio database e invia un evento a T1 (Achievement Service) per verificare se l’utente ha raggiunto un nuovo achievement.
- T1 aggiorna gli achievement e notifica T5, che comunica il risultato al frontend.

3. Autenticazione:

- Quando un utente accede al sistema, il frontend invia le credenziali a T5.
- T5 inoltra la richiesta a T8-T9 (Authentication Service), che verifica le credenziali e genera un token.
- Il token viene validato a ogni richiesta inviata al backend.

Possiamo quindi affermare che l’architettura basata su microservizi garantisce una progettazione modulare, scalabile e orientata alla separazione delle responsabilità. Ogni componente è progettato per operare in modo indipendente, facilitando lo sviluppo, la manutenzione e la scalabilità del sistema. Grazie a protocolli standard e meccanismi di

CHAPTER 3. MODULI UTILIZZATI E LE LORO INTERAZIONI

comunicazione RESTful, i microservizi interagiscono efficacemente, assicurando la coerenza dei dati e una risposta rapida alle richieste degli utenti. Questa organizzazione permette di adattarsi rapidamente a nuove esigenze, di implementare modifiche senza impatti significativi sugli altri componenti e di migliorare la resilienza dell'intero sistema.

Chapter 4

Task R1 -

Implementazione

4.1 Scelte progettuali

Nel progettare l’interfaccia e la struttura del nostro profilo utente, abbiamo deciso di organizzare la pagina in **sezioni** ben definite. Questa scelta è stata guidata dall’obiettivo di migliorare l’interazione con l’utente, garantendo un’esperienza fluida, intuitiva e piacevole. Attraverso un design modulare, ogni sezione è stata pensata per rispondere a specifiche esigenze, rendendo più semplice l’accesso alle informazioni e alle funzionalità principali.

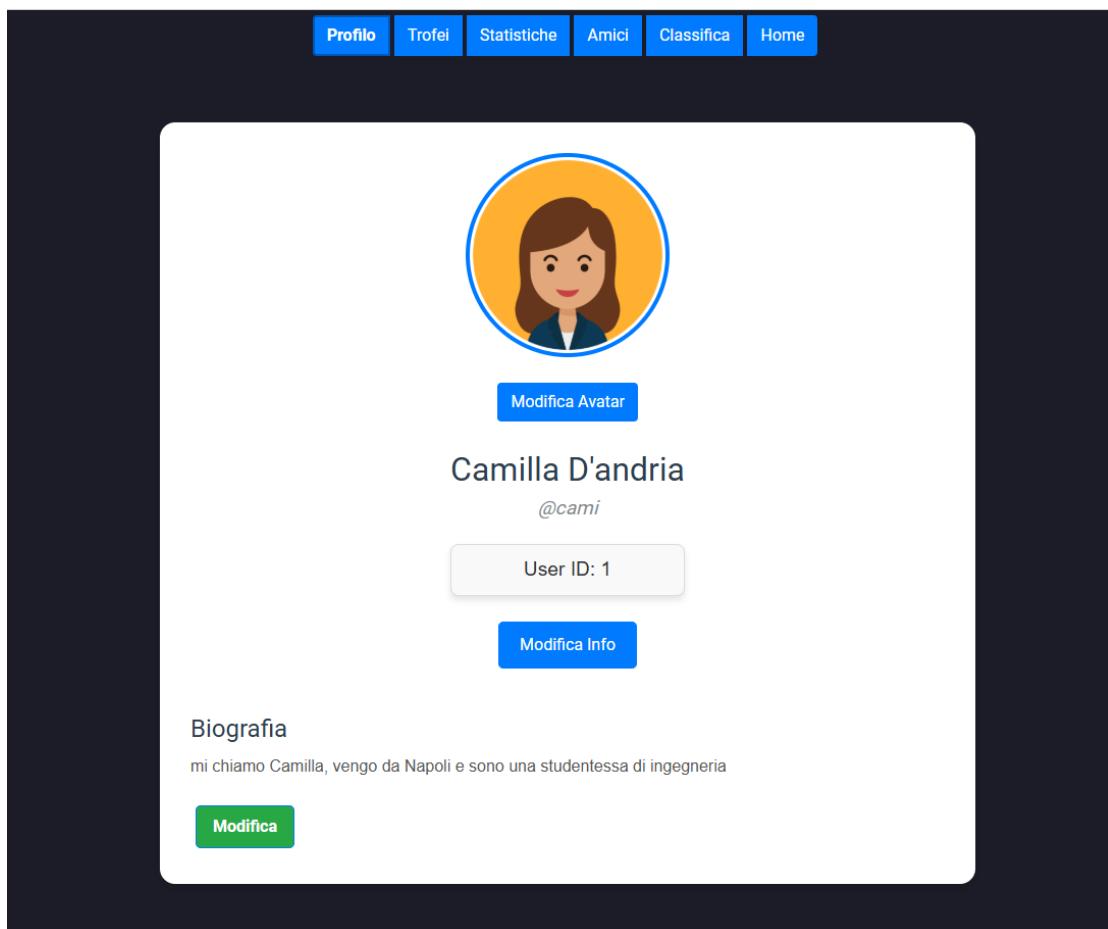
Inoltre, abbiamo posto grande attenzione all’appeal visivo dell’applicazione. Un design moderno e accattivante non solo cattura l’interesse dell’utente, ma contribuisce anche a rafforzare il coinvolgimento e la percezione di

qualità del prodotto. Questa combinazione tra funzionalità e estetica rappresenta il fulcro del nostro approccio progettuale, volto a offrire un'applicazione che sia tanto utile quanto piacevole da utilizzare.

Per accedere all'intera sezione del profilo, con tutte le modifiche da noi implementate, basta cliccare nella home in alto a destra sul nome utente. Una volta entrati nel profilo, è possibile navigare tra le diverse sezioni, facilmente accessibili tramite il menu posizionato nella parte superiore della pagina. Queste sezioni permettono all'utente di navigare in modo rapido e intuitivo tra le diverse funzionalità offerte dal profilo. Le principali sezioni disponibili sono:

- **Profilo:** Contiene le informazioni personali principali, oltre all'avatar e la biografia.
- **Trofei:** La raccolta di achievement, gli obiettivi di gioco raggiunti dall'utente.
- **Statistiche:** Una panoramica dei dati relativi ai dati di gioco dell'utente.
- **Amici:** La sezione social che gestisce le interazioni con altri utenti .
- **Classifica:** Visualizzazione delle posizioni degli utenti in una graduatoria globale o locale.
- **Home:** Permette di tornare alla pagina principale dell'applicazione.

4.2 La Sezione Profilo



La sezione **Profilo** funge da pagina principale e include i seguenti elementi:

- **Avatar:** L'immagine rappresentativa dell'utente, modificabile tramite il pulsante “Modifica Avatar”. L'utente può selezionare un avatar di default tra quelli mostrati o caricare una propria immagine da impostare come foto profilo.
- **Nome e Username:** Il nome completo dell'utente, accompagnato dall' username .

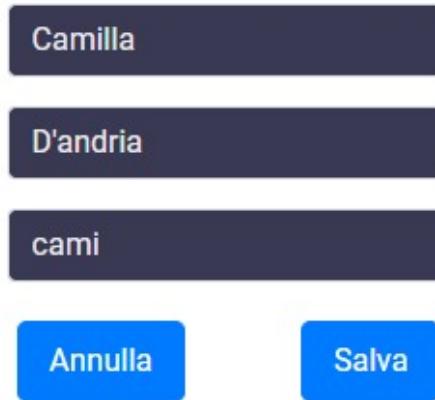
- **User ID:** Un identificativo numerico dell’utente, non modificabile.
- **Modifica Info:** Un pulsante che consente di aggiornare le informazioni personali, .
- **Biografia:** Una breve descrizione dell’utente, personalizzabile tramite il pulsante “Modifica”.

Questa struttura consente di accedere rapidamente alle informazioni essenziali e di modificarle con facilità, rendendo la sezione “Profilo” un punto di riferimento centrale per l’utente. Analizziamo ora le implementazioni pratiche di ogni funzionalità offerta da questa pagina.

4.2.1 Modifica info personali

La prima funzionalità implementata è stata quella di modifica delle informazioni personali dell’utente, quali Nome, Cognome e Nickname.

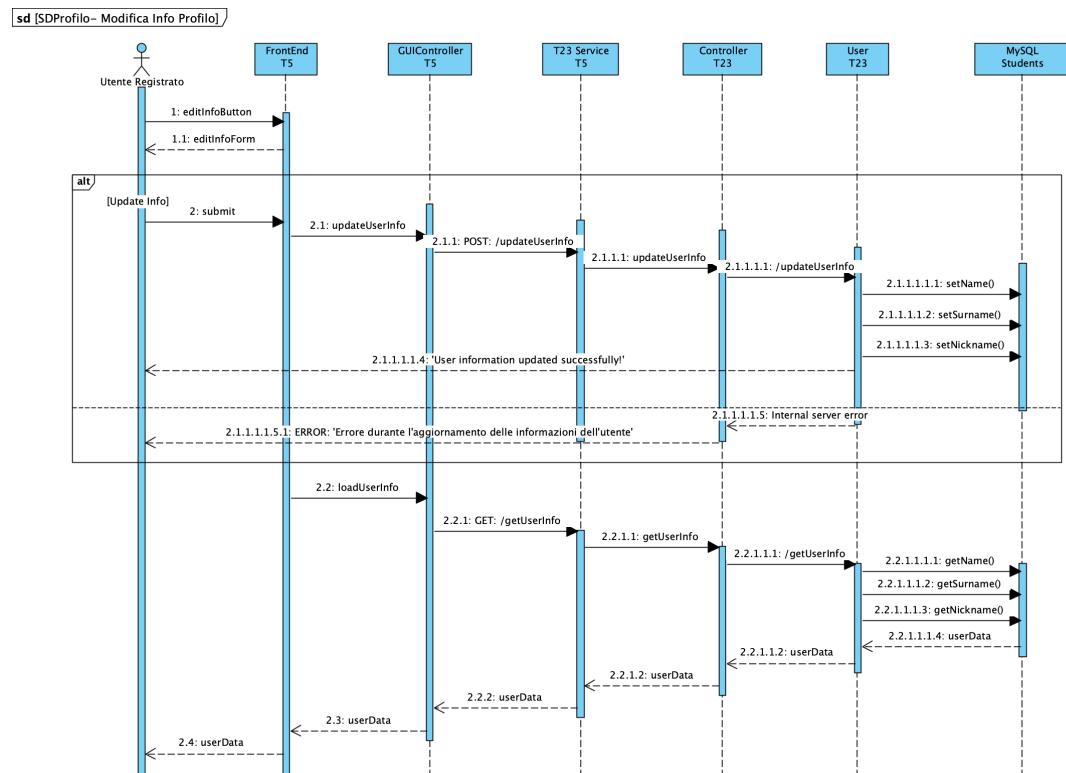
La possibilità di modifica avviene mediante un pulsante ’Modifica Info’, il quale apre un form di compilazione per i nuovi dati. Nel caso in cui si volesse modificare solo uno dei tre campi, è possibile lasciare inalterati gli altri due in quanto il form risulta già precompilato con le informazioni attuali.



Una volta compresa l’interfaccia, possiamo procedere con l’implementazione di questo requisito, analizzando il diagramma di sequenza.

Sequence Diagram - Modifica Info Profilo

La modifica delle informazioni del profilo vede l’interazione di due microservizi: T5 e T23.



Il diagramma di sequenza rappresenta il processo di modifica delle informazioni del profilo utente secondo l'architettura MVC. L'utente, tramite l'interfaccia (View), accede al form di modifica e invia i dati aggiornati. Questi vengono gestiti dal GUIController, che inoltra una richiesta POST al servizio T23. Il Controller T23 coordina l'aggiornamento delle informazioni richiamando i metodi del modello (User T23) per aggiornare nome, cognome e nickname nel database MySQL. In caso di successo, viene restituito un messaggio di conferma al frontend, mentre in caso di errore viene mostrato un messaggio di errore. Infine, la View carica nuovamente le informazioni aggiornate tramite una richiesta GET al servizio.

Per comprendere al meglio la suddivisione dei ruoli e le interazioni, di seguito sono riportati i codici relativi alla POST /updateUserInfo nei vari files.

GUIController

```

@PostMapping("/updateUserInfo")
public ResponseEntity<String> updateUserInfo(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam("name") String name,
    @RequestParam("surname") String surname,
    @RequestParam("nickname") String nickname) {
    try {
        // Decodifica il token JWT per ottenere l'ID utente
        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);

        ObjectMapper mapper = new ObjectMapper();
        @SuppressWarnings("unchecked")
        Map<String, Object> map = mapper.readValue(decodedUserJson, Map.class);
        String userId = map.get("userId").toString();

        // Chiamata al metodo del T23Service
        Boolean updateSuccess = t23Service.updateUserInfo(userId, name, surname, nickname);
        if (updateSuccess) {
            return ResponseEntity.ok("User information updated successfully!");
        } else {
            return ResponseEntity.status(400).body("Failed to update user information.");
        }
    } catch (Exception e) {
        System.out.println("Error updating user information: " + e.getMessage());
        return ResponseEntity.status(500).body("Internal server error.");
    }
}

```

Il metodo `updateUserInfo` è un endpoint POST per aggiornare nome, cognome e nickname di un utente autenticato, sfruttando l'architettura RESTful e il meccanismo di autenticazione basato su token JWT. Il token viene recuperato dai cookie e decodificato per ottenere l'ID dell'utente, che, insieme ai nuovi dati, viene passato al servizio t23Service per effettuare l'aggiornamento. Se l'operazione ha successo, restituisce un messaggio di conferma con codice HTTP 200; in caso di errore, restituisce un messaggio con codice 400 per fallimento dell'operazione o 500 per errori interni al server. Questo ID viene quindi utilizzato per chiamare il metodo `t23Service.updateUserInfo(userId, name, surname, nickname)` che si occupa di aggiornare i dati dell'utente nel backend.

T23Service

```
public Boolean updateUserInfo(String userId, String name, String surname, String nickname) {
    final String endpoint = "/updateUserInfo";
    // Creazione del payload per la richiesta POST
    MultiValueMap<String, String> formData = new LinkedMultiValueMap<>();
    formData.add("userId", userId);
    formData.add("name", name);
    formData.add("surname", surname);
    formData.add("nickname", nickname);

    try {
        // Effettua la richiesta POST al servizio T23
        return callRestPost(endpoint, formData, null, Boolean.class);
    } catch (Exception e) {
        System.err.println("Errore durante l'aggiornamento delle informazioni dell'utente: " + e.getMessage());
        return false;
    }
}
```

Per quanto riguarda la parte di interfaccia di T5 con T23, il servizio costruisce un payload per la richiesta POST tramite un oggetto `MultiValueMap<String, String>` chiamato `formData`. Il `MultiValueMap` è una struttura dati fornita da **Spring** che consente di mappare una chiave a uno o più valori, risultando ideale per costruire payload di

richieste HTTP in formato key-value. Nel caso specifico, il formData viene popolato con i dati userId, name, surname e nickname tramite il metodo add(key, value).

Questo payload viene quindi inviato all'endpoint */updateUserInfo* utilizzando il metodo callRestPost, che effettua la chiamata REST al servizio T23 specificando il tipo di dato atteso in risposta, in questo caso un valore booleano che indica il successo o il fallimento dell'operazione.

La chiamata REST è gestita all'interno di un blocco try-catch per garantire robustezza. Se l'operazione ha successo, il valore booleano restituito dal servizio viene ritornato, segnalando che l'aggiornamento è stato completato. In caso di errori, come problemi di connessione o malfunzionamenti del servizio T23, il blocco catch cattura l'eccezione, stampa un messaggio di errore nella console includendo il dettaglio dell'eccezione con e.getMessage(), e restituisce false per indicare il fallimento.

Controller (T23)

```
@PostMapping("/updateUserInfo")
public ResponseEntity<String> updateUserInfo(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam("name") String name,
    @RequestParam("surname") String surname,
    @RequestParam("nickname") String nickname) {
    try {
        // Verifica il token JWT
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Unauthorized");
        }

        // Decodifica il token JWT per ottenere l'ID utente
        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);

        ObjectMapper mapper = new ObjectMapper();
        Map<String, Object> userData = mapper.readValue(decodedUserJson, Map.class);
        String userId = userData.get("userId").toString();

        // Recupera l'utente dal database
        Optional<User> optionalUser = userRepository.findById(Integer.parseInt(userId));

        if (!optionalUser.isPresent()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User not found");
        }

        // Aggiorna i dati dell'utente
        User user = optionalUser.get();
        user.setName(name);
        user.setSurname(surname);
        user.setNickname(nickname);
        userRepository.save(user); // Salva le modifiche nel database

        return ResponseEntity.ok("User information updated successfully!");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Internal server error");
    }
}
```

il metodo *updateUserInfo* nel controller gestisce l'aggiornamento delle informazioni utente nel database. Dopo aver ricevuto la richiesta dal servizio t23Service, il controller verifica prima di tutto la validità del token JWT, recuperato dai cookie della richiesta. Se il token è assente o vuoto, viene restituita una risposta HTTP 401 ("Unauthenticated") per indicare che l'accesso non è consentito.

Il token JWT viene quindi decodificato per estrarre il payload, che contiene l'userId dell'utente autenticato. Utilizzando l'userId, il controller cerca l'utente nel database tramite il repository (*userRepository.findById*). Se l'utente non viene trovato, restituisce una risposta HTTP 404 ("User not found"), garantendo una chiara distinzione tra

errori di autenticazione e risorsa non trovata.

Nel caso in cui l'utente esista, le informazioni vengono aggiornate direttamente utilizzando i metodi setter (setName, setSurname, setNickname) sull'entità utente recuperata. Successivamente, le modifiche vengono salvate nel database tramite il metodo userRepository.save(user). Se l'operazione ha successo, il controller restituisce una risposta HTTP 200 con il messaggio "User information updated successfully!".

Tuttavia, se durante l'esecuzione si verifica un'eccezione, ad esempio un problema nel database o durante la decodifica del JWT, il controller gestisce l'errore tramite un blocco catch. In questo caso, viene stampato il messaggio di errore nella console, e la risposta HTTP 500 ("Internal server error") viene inviata al chiamante per indicare un problema interno.

User Class

```
7  @Table(name = "students", schema = "studentsrepo")
8  @Data
9  @Entity
10 public class User {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.AUTO)
14     private Integer ID;
15
16     @Enumerated(EnumType.STRING)
17     public Studies studies;
18
19     @Column(name = "name", nullable = false, length = 100)
20     private String name;
21
22     @Column(name = "surname", nullable = false, length = 100)
23     private String surname;
24
25     @Column(name = "email", nullable = false, unique = true)
26     private String email;
27
28     @Column(name = "password", nullable = false)
29     private String password;
30
31     public boolean isRegisteredWithFacebook;
32     public boolean isRegisteredWithGoogle;
33
34     @Column(name = "nickname", unique = true, nullable = false, length = 50)
35     private String nickname;
36
37     @Column(name = "biography", length = 500)
38     private String biography;
39
40     @Column(name = "avatar", nullable = true, length = 255)
41     private String avatar;
42     | You, 6 giorni fa • updateUserInfo
43     @Lob
44     @Column(name = "profile_picture")
45     private byte[] profilePicture;
```

Per quanto riguarda il collegamento tra il controller T23 e il database, la classe User gioca un ruolo fondamentale come rappresentazione dell'entità "utente" all'interno del sistema. Una volta che il controller decodifica il token JWT e recupera l'userId, utilizza il repository per accedere ai dati nel database MySQL, dove ogni riga della tabella students è mappata a un'istanza della classe User.

La classe User è annotata con @Entity, che la identifica come una tabella JPA, e specifica la sua corrispondenza con la tabella students nello schema studentsrepo. Grazie a questa mappatura, i dati dell’utente vengono recuperati utilizzando il metodo userRepository.findById(userId), che restituisce un oggetto Optional<User>. Se l’utente esiste, il controller utilizza i metodi setter della classe User, come setName, setSurname e setNickname, per aggiornare le informazioni dell’utente.

Oltre ai campi standard come name, surname, email e nickname, la classe User gestisce altri dettagli utili, come il percorso di un avatar (avatar), un’immagine in formato binario (profilePicture), e un token per il reset della password (resetToken). Questa flessibilità consente alla classe di supportare una varietà di scenari applicativi, come la registrazione tramite social network (grazie ai campi isRegisteredWithFacebook e isRegisteredWithGoogle) o l’aggiunta di informazioni personali come una biografia (biography).

Per brevità, l’immagine mostrata è tagliata e non include tutti i metodi get e set, che sono ovviamente presenti nel codice originale.

Come abbiamo detto, il controller salva le modifiche richiamando il metodo userRepository.save(user), che persiste l’istanza modificata nel database. Questo ciclo, dal recupero dell’utente all’aggiornamento e al salvataggio, è reso possibile dalla struttura chiara e modulare della classe User, che funge da intermediario tra l’applicazione e il database.

MySQL.

Grazie all'integrazione con JPA e al supporto fornito dalla classe User, il controller può concentrarsi sulla logica applicativa, lasciando la gestione delle operazioni sui dati a un'implementazione pulita e riutilizzabile.

Per quanto riguarda la composizione del database MySQL studentsrepo, contenente le informazioni personali caratterizzanti il profilo utente, si rimanda al diagramma ER presenti nel capitolo 1.

4.2.2 Aggiorna Biografia

La funzionalità "Modifica Biografia" permette agli utenti di visualizzare e aggiornare la propria biografia in modo intuitivo. All'apertura della pagina dedicata, la biografia salvata nel database viene recuperata e mostrata nella sezione principale del profilo utente. Quando quest'ultimo clicca sul pulsante "Modifica Bio", viene aperta una finestra di modifica (modal o form), che consente di aggiornare il testo della biografia. L'interfaccia offre due opzioni principali: Salva o Annulla. Se l'utente dovesse confermare le modifiche cliccando su "Salva", i dati aggiornati verrebbero inviati al backend tramite una richiesta al T23, che valida e salva le modifiche nel database.

Biografia

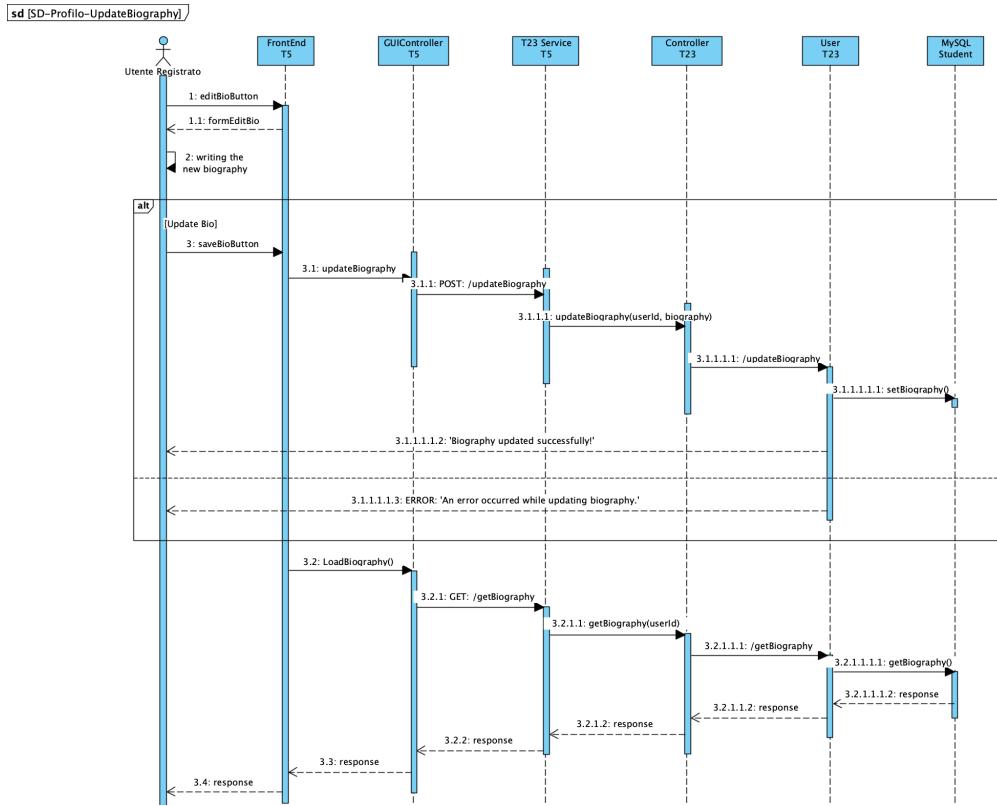
Mi chiamo Camilla, ho 24 anni e sono una studentessa di Ingegneria Informatica Magistrale. La mia passione per la tecnologia e l'innovazione mi ha guidata in questo percorso di studi impegnativo, ma stimolante. Sono determinata, curiosa e pronta ad affrontare ogni sfida per trasformare i miei sogni in realtà.

Annulla

Salva

Sequence Diagram - Aggiorna Biografia

L'implementazione della funzionalità "Aggiorna Biografia" segue un approccio molto simile a quello utilizzato per l'aggiornamento delle informazioni del profilo, sfruttando gli stessi file e componenti dell'applicazione. Questo include il controller T23, il T23Service e la classe User, che già gestiscono le operazioni CRUD per l'utente nel database. La logica generale rimane coerente: i dati forniti dall'utente vengono inviati dal frontend al backend, dove vengono elaborati e salvati nel database tramite l'entità User.



L'implementazione specifica della funzionalità "Aggiorna Biografia" si articola attraverso l'uso di due principali metodi HTTP, POST `/updateBiography` e GET `/getBiography`, che gestiscono rispettivamente l'aggiornamento e il recupero dei dati.

Il metodo POST `/updateBiography` viene attivato quando l'utente, dopo aver modificato la biografia, preme il pulsante "saveBioButton". Questo avvia una sequenza in cui i dati vengono inviati dal frontend al backend, passando attraverso il T23Service, che si interfaccia con il Controller T23. Il controller, a sua volta, inoltra la richiesta al metodo `/updateBiography`, il quale richiama la funzione `setBiography()` della classe User. Questa funzione aggiorna la biografia dell'utente nel database MySQL. Al termine dell'operazione, viene restituito un

messaggio di conferma oppure, in caso di errore, un messaggio che segnala il problema.

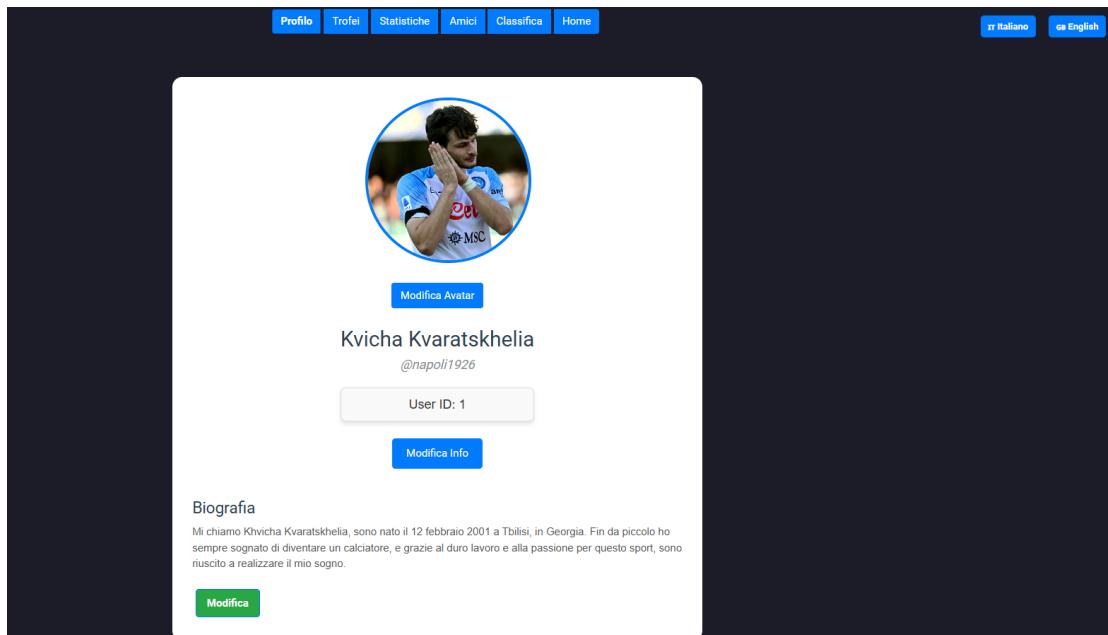
Il metodo GET `/getBiography` viene utilizzato per caricare la biografia esistente e viene invocato, ad esempio, quando l’utente apre il modulo di modifica per visualizzare o verificare il contenuto attuale. Questa operazione parte dal frontend tramite il metodo `LoadBiography()`, che chiama il servizio `T23Service`. Quest’ultimo invia una richiesta al Controller `T23`, che richiama `/getBiography` per recuperare i dati dal database tramite la funzione `getBiography()` della classe `User`. I dati della biografia vengono quindi restituiti al frontend per essere visualizzati.

Nel codice, i metodi GET e POST riguardanti la biografia richiamano molto quelli già implementati per l’aggiornamento delle informazioni del profilo, ma con alcune differenze chiave. Nel caso del metodo `updateBiography`, l’operazione è più mirata e si concentra esclusivamente sull’aggiornamento di un singolo campo, la biografia, a differenza di `updateUserInfo`, che gestisce più parametri come nome, cognome e nickname.

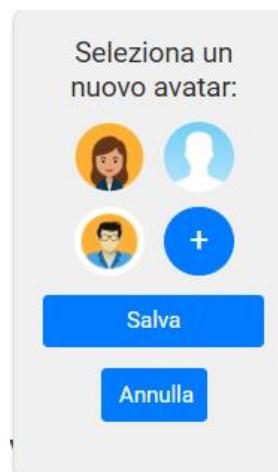
La struttura di entrambi i metodi è simile: il token JWT viene decodificato per ottenere l’ID utente, si richiama il servizio `T23Service` per eseguire l’operazione necessaria, si interfacciano con il database attraverso il repository dell’entità `User` e si gestiscono le risposte con mes-

saggi specifici per il successo o il fallimento dell'operazione. Nel caso di updateBiography, il controller utilizza il metodo findById del repository per recuperare l'utente dal database. Trovato l'utente, viene aggiornato esclusivamente il campo biography e l'entità modificata viene salvata con il metodo save.

4.2.3 Modifica Avatar/ Carica Foto



Questo caso d'uso presenta una struttura complessa, in quanto offre due funzionalità principali strettamente collegate tra loro: la possibilità di modificare l'avatar dell'utente e quella di caricare un'immagine in formato JPG dal proprio dispositivo per sostituire l'avatar esistente. In entrambi i casi, la modifica viene immediatamente riflessa nella pagina principale.



Durante l'implementazione di questo requisito, sono emerse diverse problematiche, tra cui la gestione dei conflitti tra il caricamento dell'avatar e delle immagini, la gestione del database e la conversione delle immagini per il loro trattamento e salvataggio.

Come illustrato nei capitoli precedenti, il salvataggio dell'avatar e

della foto coinvolgono entrambi il database Students in T23. Per ogni utente sono previste due colonne distinte: una dedicata al salvataggio dell'avatar e l'altra al salvataggio della foto.

Questa scelta è motivata dalla diversa natura delle due entità. Gli avatar, essendo già tutti pre-caricati in una specifica directory del progetto, richiedono solo un riferimento al percorso della loro posizione. Per questo motivo, la colonna destinata agli avatar accetta un campo di tipo **URL** che punta alla risorsa, quindi un semplice varchar. La foto (*profile picture*), invece, richiede un processo più articolato: deve essere caricata dall'utente, convertita e poi salvata direttamente nel database. Per gestire questa complessità, è stato scelto il formato **BLOB** per la colonna dedicata alle foto.

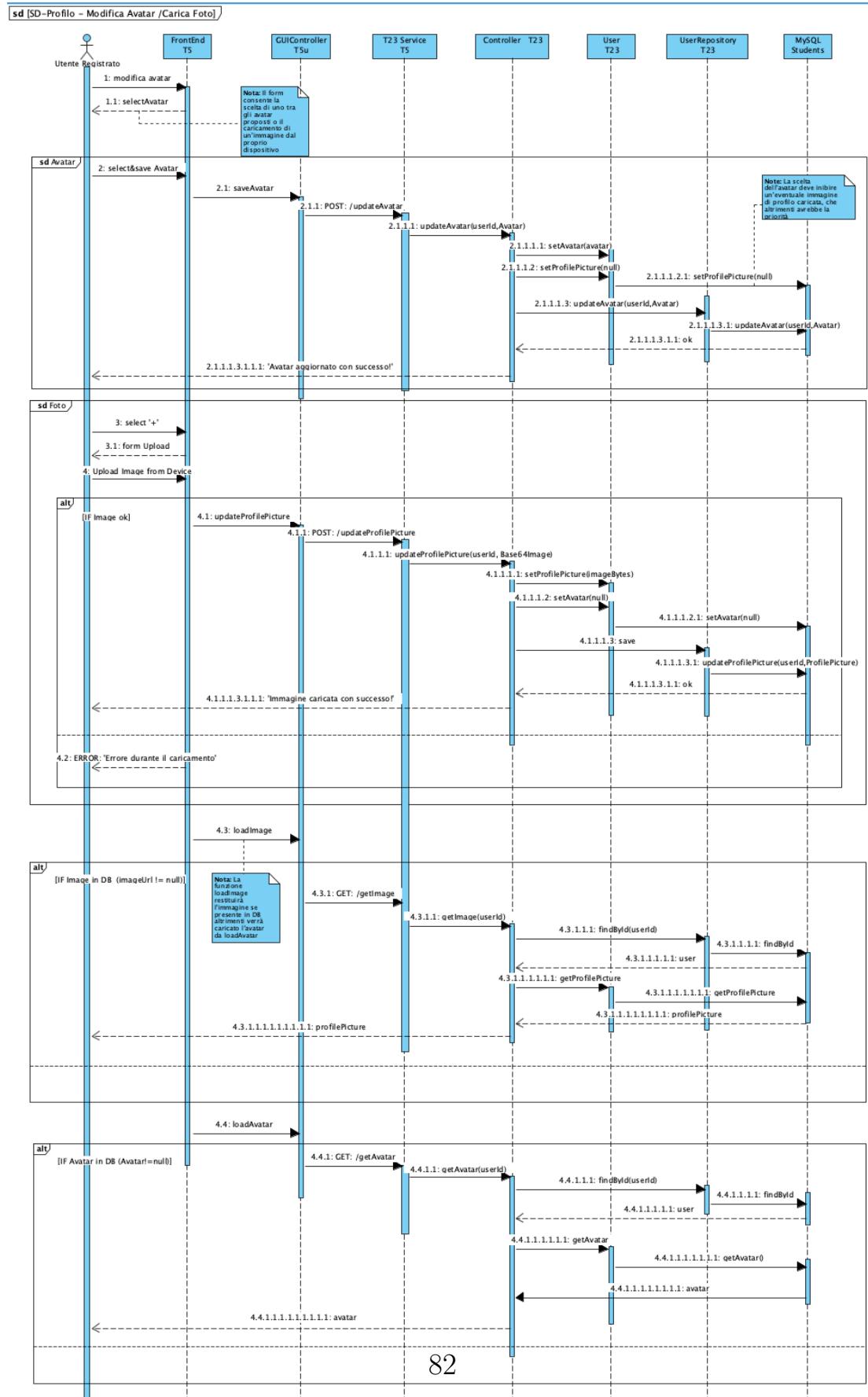
Field	Type	Null	Key	Default	Extra
<code>id</code>	<code>int</code>	<code>NO</code>	<code>PRI</code>	<code>NULL</code>	
<code>biography</code>	<code>varchar(500)</code>	<code>YES</code>		<code>NULL</code>	
<code>email</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>is_registered_with_facebook</code>	<code>bit(1)</code>	<code>NO</code>		<code>NULL</code>	
<code>is_registered_with_google</code>	<code>bit(1)</code>	<code>NO</code>		<code>NULL</code>	
<code>name</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>password</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>reset_token</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>studies</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>surname</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>nickname</code>	<code>varchar(50)</code>	<code>YES</code>	<code>UNI</code>	<code>NULL</code>	
<code>avatar</code>	<code>varchar(255)</code>	<code>YES</code>		<code>NULL</code>	
<code>profile_picture</code>	<code>longblob</code>	<code>YES</code>		<code>NULL</code>	

BLOB, acronimo di Binary Large Object, è un tipo di dato utilizzato nei database per memorizzare grandi quantità di dati binari. È particolarmente utile per salvare informazioni come immagini, file audio, video o documenti, poiché consente di archiviare direttamente i contenuti binari all'interno del database stesso, anziché doverli conservare come stringhe o numeri.

vare come file esterni. Nel nostro caso, il formato BLOB viene utilizzato per memorizzare le immagini di profilo caricate dagli utenti: ciò implica che prima di essere salvata, l'immagine deve essere convertita in un formato compatibile con il database.

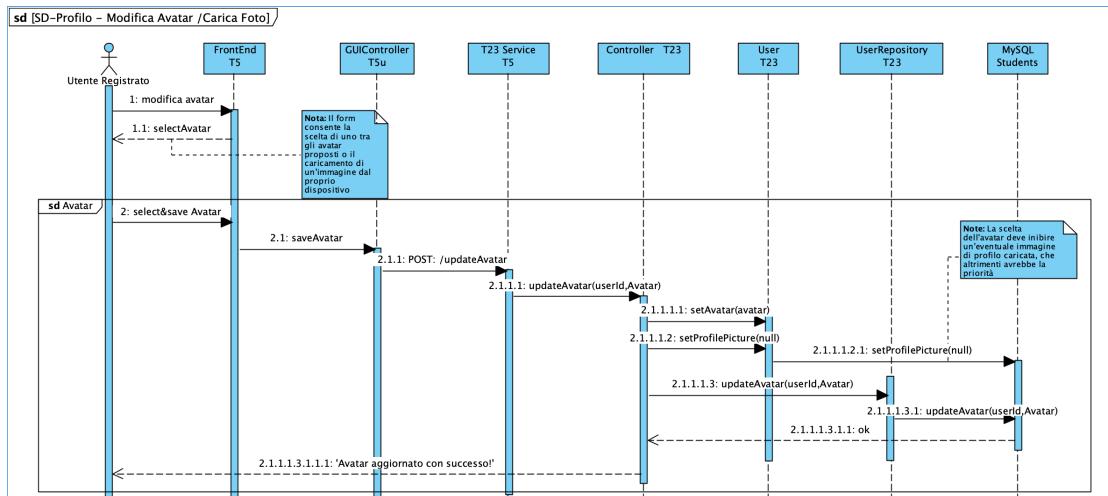
Passiamo ora all'analisi del Sequence Diagram, che mostra l'implementazione della funzionalità tramite l'interazione dei vari moduli.

Sequence Diagram - Modifica Avatar/ Carica Foto



Il diagramma si suddivide in due parti per distinguere le due possibilità di scelta da parte dell’utente.

L’aggiornamento dell’**avatar**, come detto, permette all’utente di scegliere un’immagine predefinita dalla libreria disponibile nel sistema. Questa operazione non richiede il caricamento di file personali, ma si limita alla selezione di un’immagine che viene identificata tramite il suo URL. Esse sono situate nella cartella `/T5/IMAGES/PROFILO/`, quindi gli URL inizieranno con questo percorso.

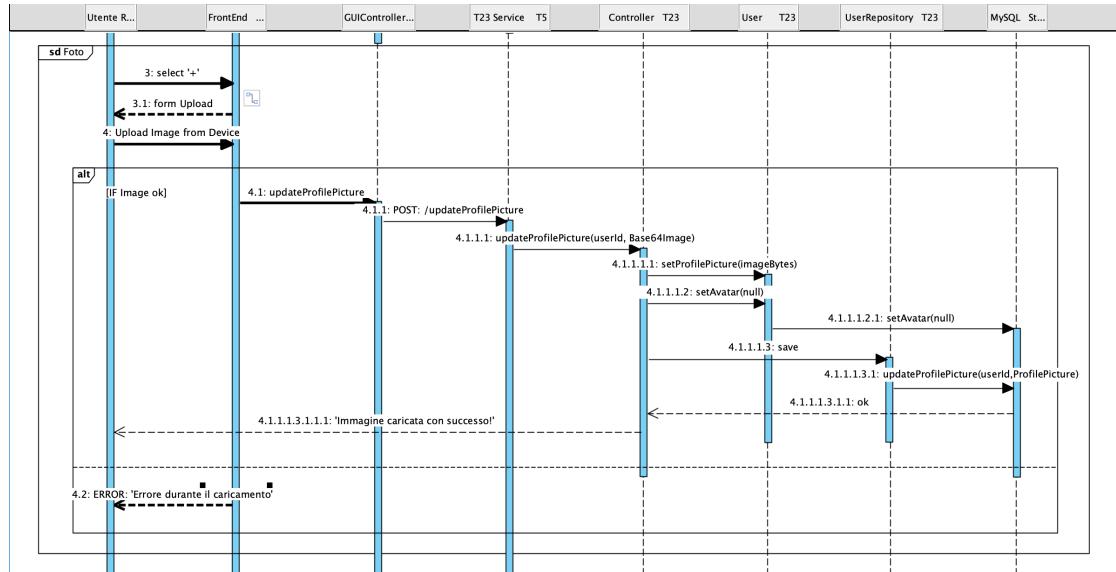


Il processo inizia quando l’utente seleziona un avatar dalla lista presente nel Front-End. A questo punto, viene inviata una richiesta al sistema tramite l’endpoint `/updateAvatar`, specificando l’ID dell’utente e l’URL dell’immagine scelta. Questa richiesta arriva al controller, che passa il compito al service del sistema (T23 Service).

Il service aggiorna il database MySQL, andando a scrivere l’URL dell’avatar nella colonna dedicata dell’utente. Se l’utente aveva precedentemente caricato una foto personale, il sistema provvede a resettare

il campo relativo alla foto, impostandolo su null. Questo accade per evitare conflitti: il profilo utente può infatti mostrare solo una delle due immagini, avatar o foto, e non entrambe.

Una volta completato l'aggiornamento, il sistema restituisce un messaggio di conferma al Front-End, notificando all'utente che l'operazione è andata a buon fine con un messaggio come: "Avatar aggiornato con successo!". In questo modo, l'utente vede immediatamente l'avatar aggiornato nella pagina principale del profilo.



Il caricamento di una **foto** offre invece una maggiore personalizzazione, permettendo all'utente di utilizzare un'immagine personale al posto dell'avatar predefinito. Questo processo, più articolato, implica il caricamento di un file dal dispositivo dell'utente e la gestione di dati binari nel database.

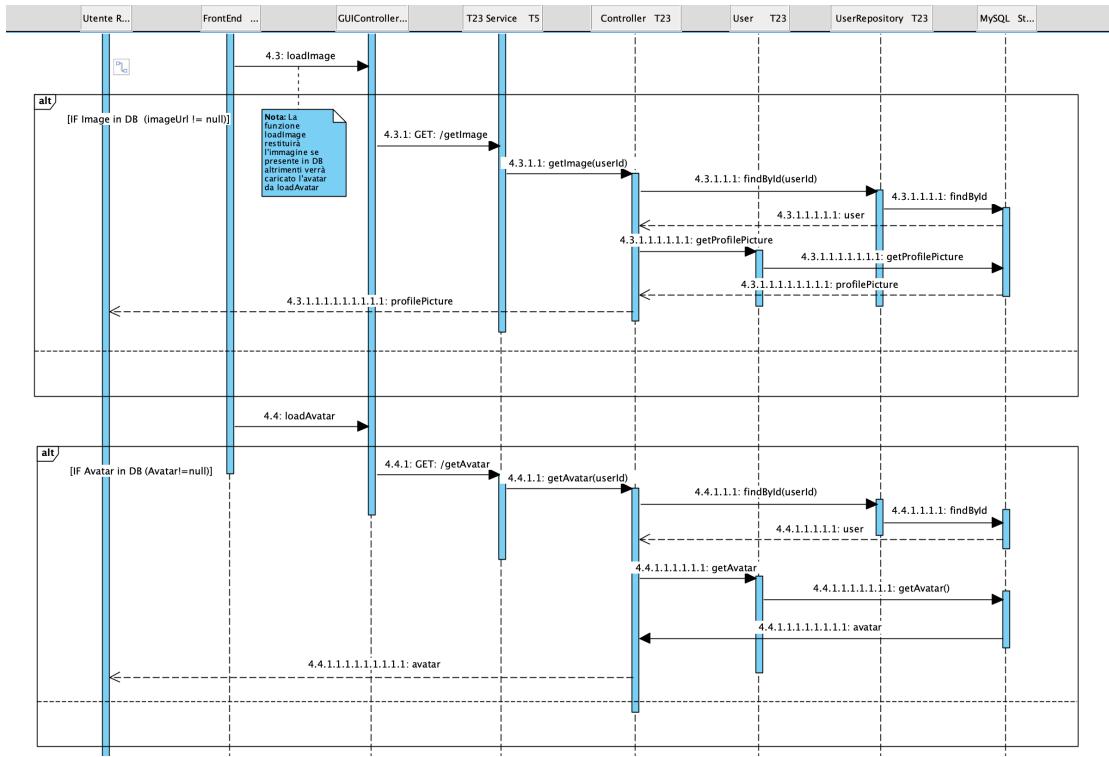
Quando l'utente decide di caricare una foto, seleziona il pulsante

"+" e accede al modulo di upload. Dopo aver scelto un'immagine dal proprio dispositivo, il sistema invia una richiesta al controller tramite l'endpoint /updateProfilePicture. Questa richiesta include l'ID dell'utente e i dati dell'immagine, che vengono convertiti in un formato compatibile con il database, come Base64 o direttamente binario.

Il controller inoltra l'operazione al service, che si occupa di elaborare l'immagine. Questa viene salvata nella colonna ProfilePicture del database. Parallelamente, il sistema resetta l'eventuale avatar preesistente, impostandolo su null. Anche in questo caso, il sistema garantisce che l'utente possa avere una sola immagine attiva associata al profilo.

Dopo il salvataggio, il sistema conferma l'operazione restituendo un messaggio al Front-End. In caso di errori, come il caricamento di un file non valido, o un formato che non sia jpg, l'utente riceve un messaggio di errore.

La parte conclusiva del diagramma illustra invece le funzioni per consentire l'effettiva visualizzazione dell'immagine nei due casi.



Quando il sistema deve visualizzare l'immagine di profilo di un utente, segue un processo ben definito per decidere se mostrare una foto personalizzata o un avatar predefinito. Questo processo è progettato per verificare prima l'esistenza di una foto caricata dall'utente e, solo in assenza di questa, utilizzare un avatar tra quelli disponibili nel sistema.

Il Front-End invoca in primis la funzione `loadImage` per recuperare l'immagine di profilo associata all'utente, inoltrando la richiesta al `GUIController`, che funge da ponte tra il Front-End e il Back-End. Il `GUIController`, a sua volta, invia una richiesta al servizio `/getImage`, gestito dal `T23 Service`, passando come parametro l'`userId` dell'utente. Questa chiamata consente al sistema di cercare l'immagine corrispondente nel database.

Il `T23 Service` delega il recupero dei dati al `UserRepository`, il

livello di accesso diretto al database, richiamando il metodo `getImage(userId)`. Il UserRepository interroga la tabella Students del database MySQL, verificando se nella colonna ProfilePicture è presente un’immagine associata all’utente. Se viene trovata un’immagine (cioè, se il valore non è null), questa viene recuperata e restituita al T23 Service.

Dopo aver ricevuto l’immagine, il T23 Service invia il dato al GUIController, che lo inoltra al Front-End. A questo punto, l’immagine viene visualizzata direttamente nell’interfaccia utente come immagine di profilo.

Se il sistema non trova una foto personalizzata nella colonna ProfilePicture (che risulta vuota), si attiva il flusso alternativo per il recupero di un avatar predefinito. In questo caso, il Front-End, tramite il GUIController, invoca la funzione `loadAvatar`, richiedendo al sistema di recuperare l’avatar associato all’utente.

Il GUIController inoltra questa richiesta al servizio `/getAvatar`, gestito dal T23 Service, passando l’`userId` dell’utente come parametro. Il T23 Service, analogamente a quanto avviene per il recupero della foto, delega l’operazione al UserRepository, il livello responsabile dell’accesso ai dati.

Questa volta, il UserRepository interroga la colonna Avatar nella tabella Students del database MySQL per verificare se è presente un URL che punta a un’immagine predefinita. Se un URL valido viene

trovato, viene recuperato e restituito al T23 Service.

L'URL dell'avatar recuperato viene quindi passato dal T23 Service al GUIController, che lo inoltra al Front-End. Una volta ricevuto, l'avatar viene visualizzato come immagine di profilo nell'interfaccia utente.

Passiamo ora all'analisi del codici che hanno consentito l'integrazione di queste funzionalità.

GUIController

```
@PostMapping("/updateAvatar")
public ResponseEntity<String> updateAvatar(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam("avatar") String avatar) {
    try {
        // Verifica se il token JWT è presente
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("User not authenticated");
        }

        // Decodifica il token JWT per ottenere l'ID utente
        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);

        ObjectMapper mapper = new ObjectMapper();
        @SuppressWarnings("unchecked")
        Map<String, Object> map = mapper.readValue(decodedUserJson, Map.class);
        String userId = map.get("userId").toString();
        Integer userIdAsInteger = Integer.parseInt(userId);
        // Chiamata al metodo del T23Service per aggiornare l'avatar
        Boolean updateSuccess = (Boolean) t23Service.updateAvatar(userIdAsInteger, avatar);

        if (updateSuccess) {
            return ResponseEntity.ok("Avatar updated successfully!");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Failed to update avatar.");
        }
    } catch (Exception e) {
        System.err.println("Error updating avatar: " + e.getMessage());
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Internal server error.");
    }
}
```

Quando un utente decide di aggiornare il proprio avatar predefinito, il sistema utilizza il metodo *updateAvatar* del GUIController per gestire l'intero processo. Tutto inizia con la richiesta proveniente dal Front-End, che invia i dati dell'avatar selezionato (sotto forma di URL) insieme al token JWT che rappresenta l'utente autenticato. Il controller,

come prima cosa, verifica la presenza del token JWT. Questo token, che viene memorizzato nei cookie, è fondamentale per identificare l’utente, come visto nei codici precedenti: se non è presente o risulta vuoto, il sistema risponde con un errore che informa l’utente che non è autenticato.

Se il token è valido, il sistema passa alla fase successiva, in cui il token viene decodificato. Il controller estrae dal token JWT le informazioni sull’utente, come l’ID, utilizzando una procedura di decodifica. Una volta ottenuto l’ID dell’utente, il controller invoca il metodo updateAvatar del T23Service. Questo servizio si occupa di aggiornare l’avatar nel database, sostituendo il precedente con quello nuovo indicato dall’utente.

A seconda del risultato, il sistema restituisce una risposta al Front-End: se l’aggiornamento va a buon fine, l’utente riceve una conferma che l’avatar è stato aggiornato con successo. In caso contrario, ad esempio se ci sono problemi con i dati inviati o con il database, il sistema restituisce un messaggio di errore, spiegando che l’aggiornamento non è riuscito. In presenza di problemi interni, come errori imprevisti durante l’elaborazione, viene restituito un messaggio che informa l’utente di un errore del server.

```
@PostMapping("/updateProfilePicture")
public ResponseEntity<String> updateProfilePicture(
    @RequestBody Map<String, String> payload,
    @CookieValue(name = "jwt", required = false) String jwt
) {
    try {
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Utente non autenticato.");
        }

        Integer userId = extractUserIdFromJwt(jwt);
        if (userId == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Token JWT non valido.");
        }

        String base64Image = payload.get("profilePicture");
        if (base64Image == null || base64Image.isEmpty()) {
            return ResponseEntity.badRequest().body("Nessuna immagine fornita.");
        }

        boolean success = t23Service.updateProfilePicture(userId, base64Image);
        return success
            ? ResponseEntity.ok("Immagine caricata con successo!")
            : ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore durante il caricamento dell'immagine.");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Si è verificato un errore.");
    }
}
```

Quando l’utente desidera caricare un’immagine personalizzata come foto di profilo, il sistema utilizza il metodo *updateProfilePicture* del GUIController. Anche in questo caso, la richiesta parte dal Front-End, che invia due elementi principali: il token JWT dell’utente autenticato e l’immagine convertita in una stringa Base64.

La scelta di utilizzare la codifica Base64 per rappresentare l’immagine non è casuale. Base64 è un sistema di codifica che converte i dati binari (come un’immagine) in una stringa di testo, utilizzando un set limitato di caratteri alfanumerici. Questo approccio è utile quando si lavora con protocolli o formati che supportano solo stringhe di testo, come JSON, utilizzato dal Front-End per inviare i dati al server.

Sebbene il database supporti nativamente i BLOB per il salvataggio di dati binari, la codifica Base64 viene utilizzata come passo intermedio per garantire la compatibilità nella trasmissione dei dati tra Front-End

e Back-End. Dopo aver ricevuto la stringa Base64, il T23Service si occupa di decodificarla, convertendola in dati binari prima di salvarla nel database.

Tornando al GUIController, una volta confermata l'autenticazione, il token viene decodificato per estrarre l'ID dell'utente. Questa fase è cruciale, poiché il sistema utilizza l'ID per associare l'immagine all'utente corretto. Successivamente, il controller si accerta che l'immagine sia effettivamente presente nella richiesta e che sia valida. Se l'immagine è mancante o non corretta, l'utente riceve un messaggio di errore che lo informa del problema.

Quando tutti i dati sono stati validati, il controller passa l'immagine e l'ID utente al T23Service, che si occupa della logica di business necessaria per elaborare l'immagine. Il servizio aggiorna il database, salvando l'immagine nella colonna corrispondente all'utente. Al termine del processo, il sistema risponde al Front-End: se tutto è andato a buon fine, l'utente riceve un messaggio di conferma che l'immagine è stata caricata con successo; in caso contrario, un messaggio lo informa che si è verificato un errore durante il caricamento.

T23Service

```
public Boolean updateAvatar(Integer userId, String avatar) {
    final String endpoint = "/updateAvatar";
    MultiValueMap<String, String> formData = new LinkedMultiValueMap<>();
    formData.add("avatar", avatar); // Parametro avatar

    try {
        // Chiamata al servizio T23
        return callRestPost(endpoint, formData, null, Boolean.class);
    } catch (Exception e) {
        System.err.println("Errore durante l'aggiornamento dell'avatar: " + e.getMessage());
        return false;
    }
}
```

Il metodo *updateAvatar* all'interno del T23Service rappresenta la logica che collega il sistema interno al servizio esterno T23 per aggiornare l'avatar di un utente. Una volta che il GUIController riceve la richiesta dal Front-End e decodifica le informazioni sull'utente, delega a questo metodo l'invio della richiesta al servizio remoto. Il T23Service si occupa di tradurre i dati ricevuti dal controller (come l'userId e l'avatar) in una chiamata REST verso l'endpoint specifico del servizio esterno, /updateAvatar.

Questa funzione è progettata per restituire un valore booleano che indica se l'operazione è andata a buon fine. In caso di errore (ad esempio, problemi di comunicazione con il servizio T23), il metodo si occupa di gestire l'eccezione internamente, evitando che il sistema interno venga interrotto o restituisca un errore non previsto.

```
public Boolean updateProfilePicture(Integer userId, String base64Image) {
    final String endpoint = "/updateProfilePicture";
    Map<String, String> payload = new HashMap<>();
    payload.put("profilePicture", base64Image);

    try {
        ResponseEntity<Boolean> response = restTemplate.postForEntity(
            endpoint,
            payload,
            Boolean.class
        );
        return response.getStatusCode() == HttpStatus.OK;
    } catch (RestClientException e) {
        System.err.println("Errore durante l'aggiornamento dell'immagine personalizzata: " + e.getMessage());
        return false;
    }
}
```

Il metodo `updateProfilePicture` rappresenta il punto centrale per l'aggiornamento della foto personalizzata di un utente. Una volta che il `GUIController` ha ricevuto dal Front-End l'immagine in formato Base64 e l'ha validata, delega al `T23Service` l'interazione con il servizio `T23` per salvare l'immagine nel database.

Questo metodo traduce la stringa Base64 e l'ID dell'utente in una richiesta REST indirizzata all'endpoint `/updateProfilePicture` del servizio `T23`. Il risultato dell'operazione viene restituito come un valore booleano, che segnala il successo o il fallimento del caricamento. In caso di problemi (ad esempio, se l'immagine non è accettata o il servizio non è disponibile), il metodo si occupa di gestire l'errore senza interrompere il flusso generale.

Controller (T23)

```
@PostMapping("/updateAvatar")
public ResponseEntity<String> updateAvatar(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam("avatar") String avatar) {
    try {
        // Verifica il token JWT
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("User not authenticated");
        }

        // Decodifica il token JWT per ottenere l'ID utente
        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);

        ObjectMapper mapper = new ObjectMapper();
        Map<String, Object> userData = mapper.readValue(decodedUserJson, Map.class);
        String userId = userData.get("userId").toString();

        // Recupera l'utente dal database
        Optional<User> optionalUser = userRepository.findById(Integer.parseInt(userId));

        if (!optionalUser.isPresent()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User not found");
        }

        // Aggiornamento avatar dell'utente
        User user = optionalUser.get();
        user.setAvatar(avatar); // Salva il nuovo avatar
        user.setProfilePicture(null);
        userRepository.updateAvatar(Integer.parseInt(userId), avatar);

        return ResponseEntity.ok("Avatar updated successfully!");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An error occurred while updating avatar.");
    }
}
```

Il metodo *updateAvatar* del Controller di T23 si occupa di aggiornare l'avatar predefinito di un utente direttamente nel database del sistema centrale. Quando viene chiamato, il metodo inizia verificando l'identità dell'utente utilizzando il token JWT fornito nella richiesta. Il token viene decodificato per ottenere l'*userId*, che rappresenta l'identificativo unico dell'utente. A questo punto, il metodo interroga il *UserRepository*. Questo repository esegue una query per cercare un record corrispondente all'ID fornito e restituisce un oggetto opzionale contenente l'utente, qualora venga trovato. Questa operazione consente al Controller di gestire

eventuali casi in cui l’utente non esista più nel database, ad esempio restituendo un messaggio di errore.

Una volta confermata l’esistenza dell’utente, il Controller accede alla classe User. Il campo avatar di questa entità viene aggiornato con il nuovo avatar selezionato dall’utente. Contemporaneamente, per garantire la coerenza dei dati, il Controller azzera il campo profilePicture, assicurandosi che solo un’immagine (l’avatar o la foto personalizzata) sia attiva per l’utente. Questi aggiornamenti all’entità User vengono persistiti nel database tramite il metodo updateAvatar del UserRepository, che esegue una query SQL per applicare le modifiche al record dell’utente.

Se l’operazione va a buon fine, il Controller restituisce una risposta che conferma l’aggiornamento dell’avatar. Questo risultato positivo consente al sistema di comunicare al Front-End che la modifica è stata completata correttamente. Al contrario, se l’utente non viene trovato nel database o si verifica un errore durante l’elaborazione, il metodo restituisce un messaggio di errore adeguato.

```

@PostMapping("/updateProfilePicture")
public ResponseEntity<String> updateProfilePicture(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestBody Map<String, String> payload) {
    try {
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("User not authenticated");
        }

        // Decodifica del token JWT
        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);
        ObjectMapper mapper = new ObjectMapper();
        Map<String, Object> userData = mapper.readValue(decodedUserJson, Map.class);
        String userId = userData.get("userId").toString();

        // Validazione immagine
        String base64Image = payload.get("profilePicture");
        if (base64Image == null || base64Image.isEmpty()) {
            return ResponseEntity.badRequest().body("No image provided");
        }

        // Decodifica immagine Base64
        byte[] imageBytes = Base64.getDecoder().decode(base64Image);

        // Aggiornamento colonna profile_picture (BLOB)
        Optional<User> optionalUser = userRepository.findById(Integer.parseInt(userId));
        if (!optionalUser.isPresent()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User not found");
        }

        User user = optionalUser.get();
        user.setProfilePicture(imageBytes);
        user.setAvatar(null); // Reset del campo avatar
        userRepository.save(user);

        return ResponseEntity.ok("Profile picture updated successfully!");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An error occurred while updating");
    }
}
    
```

Il metodo *updateProfilePicture* del Controller di T23 è responsabile del caricamento e dell’aggiornamento della foto personalizzata di un utente nel database. Come per l’avatar, il metodo inizia verificando l’identità dell’utente tramite il token JWT fornito nella richiesta. Una volta decodificato il token e ottenuto l’*userId*, il Controller utilizza il UserRepository per accertarsi che l’utente esista. Questo passaggio consente di validare l’operazione e di evitare errori in caso di dati non coerenti.

Dopo aver confermato l’esistenza dell’utente, il metodo si concentra

sull’elaborazione dell’immagine ricevuta. La foto viene fornita come stringa Base64. Il Controller decodifica questa stringa in un array di byte, che rappresenta il formato nativo dell’immagine. Questo array è compatibile con il tipo di dato BLOB utilizzato nel database per la colonna dedicata alla foto personalizzata (profilePicture).

Il Controller quindi accede all’entità User per aggiornare il campo profilePicture con i dati binari dell’immagine. Per garantire la coerenza dei dati, il Controller resetta il campo avatar, disattivando eventuali avatar precedentemente associati all’utente. A differenza dell’aggiornamento dell’avatar, che utilizza un metodo specifico del repository per applicare i cambiamenti, l’aggiornamento della foto personalizzata viene effettuato utilizzando il metodo save del UserRepository, che salva direttamente l’entità User con i nuovi dati nel database.

Una volta completata l’operazione, il metodo restituisce una risposta che conferma il successo del caricamento della foto personalizzata. In caso di errori, come un’immagine non valida o un problema di accesso al database, il Controller restituisce un messaggio di errore adeguato.

Si rimanda alle pagine precedenti per il funzionamento della classe User.

UserRepository

```
@Modifying
@Transactional
@Query("UPDATE User u SET u.avatar = :avatar,u.profilePicture = NULL WHERE u.ID = :userId")
void updateAvatar(@Param("userId") Integer userId, @Param("avatar") String avatar);

@Modifying
@Transactional
@Query("UPDATE User u SET u.profilePicture = :profilePicture, u.avatar = NULL WHERE u.ID = :userId")
void updateProfilePicture(@Param("userId") Integer userId, @Param("profilePicture") byte[] profilePicture);
```

La UserRepository estende l’interfaccia **JpaRepository**, ereditando così metodi standard per operazioni CRUD (Create, Read, Update, Delete) come il salvataggio e la ricerca di record. Oltre ai metodi di base, questa repository definisce diverse query personalizzate per soddisfare esigenze specifiche, come ricerche avanzate e aggiornamenti mirati. I metodi personalizzati sfruttano la JPA Query Language (JPQL) e annotazioni come @Query, @Modifying e @Transactional per modificare direttamente i record nel database.

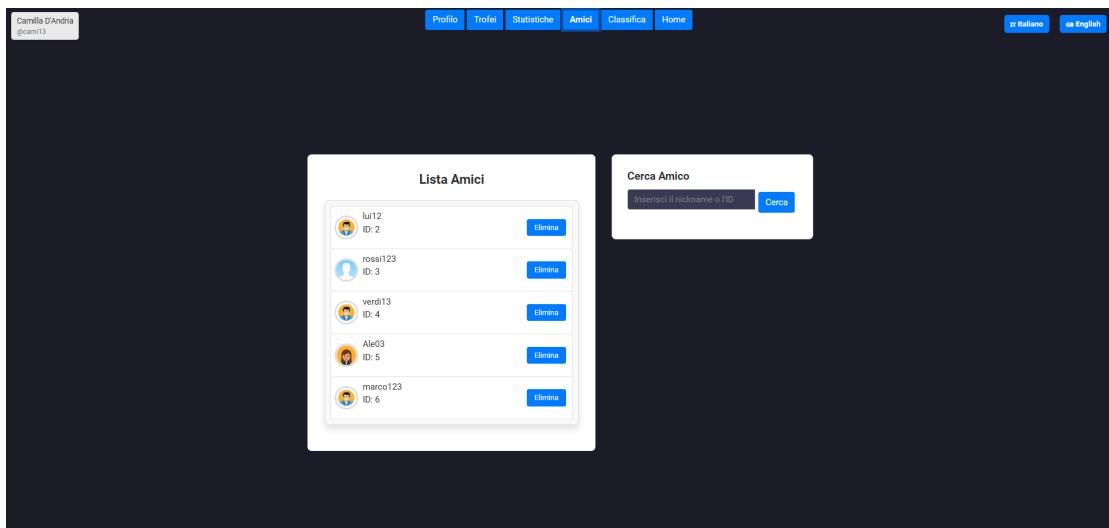
Il metodo *updateAvatar* è responsabile dell’aggiornamento del campo *avatar* di un utente nel database. Ricevendo come parametri l’*userId* e l’URL dell’avatar, il metodo esegue direttamente una query JPQL che aggiorna il campo *avatar* e azzera il campo *profilePicture* per garantire la coerenza dei dati. L’annotazione @Modifying indica che si tratta di un’operazione di aggiornamento, mentre @Transactional assicura che l’operazione venga eseguita in una transazione.

Il metodo *UpdateProfilePicture* aggiorna il campo *profilePicture*, che memorizza l’immagine in formato BLOB come array di byte. Similmente al metodo precedente, la query personalizzata aggiorna il campo

profilePicture e azzera avatar per mantenere una sola immagine attiva. Anche qui, @Transactional garantisce la consistenza del database durante l'operazione.

Si rimanda comunque al capitolo precedente per comprendere al meglio le interazioni con il database.

4.3 Sezione Amici



La seconda sezione implementata per il profilo utente consente a quest'ultimo di accedere ad un'area social, che apre porte anche futuri aggiornamenti e nuove funzionalità per il software.

Sempre nell'orbita di rendere il gioco accattivante e aumentare la gamification, un'area social può rappresentare un punto di svolta per migliorare l'esperienza dell'utente e favorire l'interazione tra i giocatori. Tra le funzionalità già implementate, spicca la possibilità di

gestire una lista amici, che consente agli utenti di ampliare le loro connessioni all'interno del gioco in modo semplice e intuitivo.

Nello specifico, il sistema permette di:

- **Aggiungere nuovi amici:** Attraverso una modalità simile al "follow" utilizzata nei social network, gli utenti possono scegliere di seguire altri giocatori, creando una rete di connessioni che promuove interazioni e competizioni amichevoli.
- **Visualizzare la lista amici:** Una sezione dedicata consente agli utenti di avere una panoramica immediata delle loro connessioni, rendendo più semplice monitorare attività, progressi o interazioni recenti.
- **Rimuovere amici dalla lista:** Per mantenere la lista sempre aggiornata e rilevante, gli utenti possono eliminare connessioni che non ritengono più utili o pertinenti.

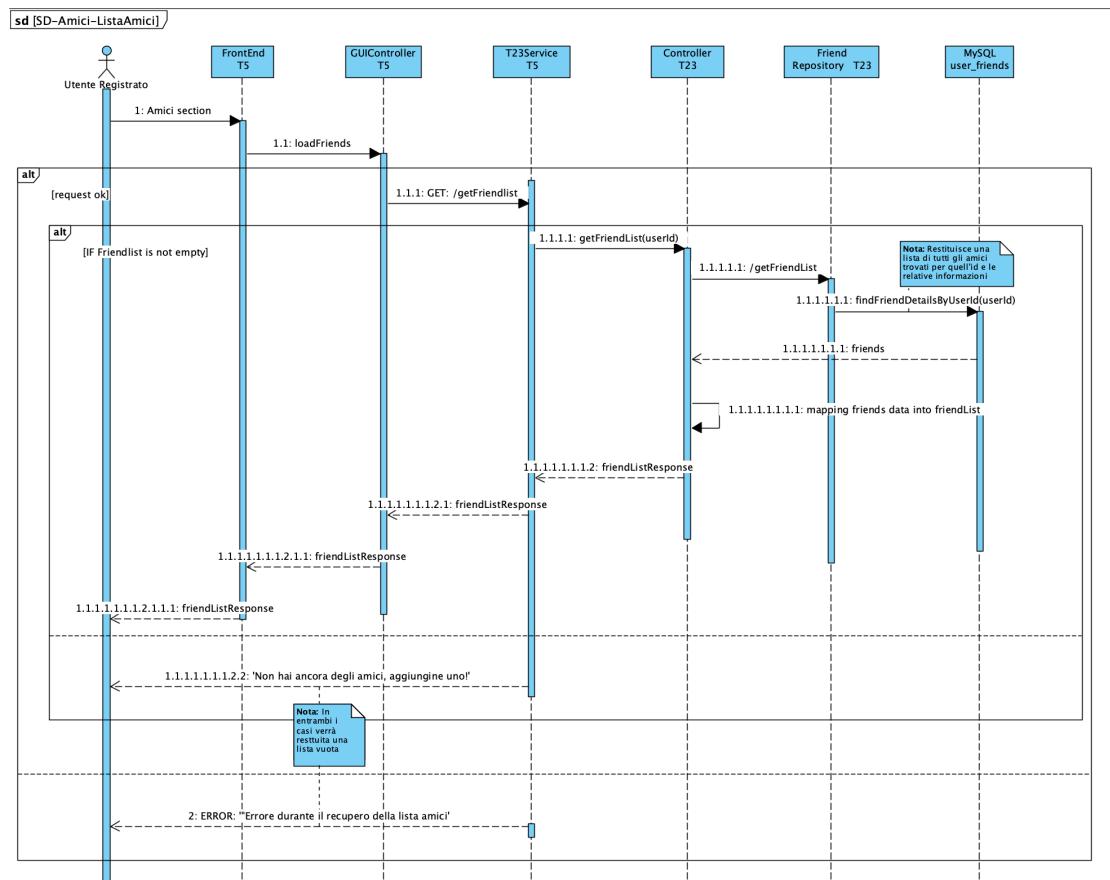
Queste funzionalità rappresentano un primo passo significativo verso la creazione di un ambiente sociale all'interno del software. L'idea di utilizzare un sistema di "follow" invece di una tradizionale richiesta di amicizia rende il processo più fluido e meno invasivo, favorendo una maggiore libertà nell'ampliare la rete di contatti.

4.3.1 Lista Amici

La funzionalità della lista amici consente di visualizzare, in automatico appena si entra nella sezione 'Amici', la lista dei giocatori che l'utente ha aggiunto fino a quel momento.

Attraverso un diagramma di sequenza, addentriamoci nella parte implementativa.

Sequence Diagram - Lista Amici



Il sequence diagram rappresenta il processo attraverso cui un utente registrato accede alla propria lista amici all'interno del sistema. Il flusso inizia quando l'utente seleziona la sezione "Amici" dall'interfaccia,

avviando una richiesta dal FrontEnd al GUIController tramite il metodo loadFriends. Il GUIController, a sua volta, inoltra una chiamata GET al servizio getFriendList, che funge da intermediario tra il FrontEnd e il livello di gestione dei dati. Questo servizio inoltra la richiesta al Controller, il quale comunica con FriendRepository per interrogare la base di dati MySQL userfriends.

Se la query alla base dati va a buon fine, il sistema entra in un ciclo in cui elabora ogni amico presente nella lista recuperata. Per ogni amico vengono raccolti i dettagli necessari, tra cui l'ID univoco, il nickname e l'avatar associato. Questi dati vengono aggregati e inviati come risposta attraverso il Controller, fino a raggiungere nuovamente il FrontEnd, che si occupa di mostrare la lista degli amici all'utente.

Il illustra inoltre i due scenari alternativi. Nel caso in cui la lista degli amici sia vuota, viene restituita una risposta priva di dati accompagnata da un messaggio informativo all'utente: "Non hai ancora degli amici, aggiungine uno!". Se invece si verifica un errore durante il recupero della lista, il sistema invia un messaggio di errore: "Errore durante il recupero della lista amici", informando l'utente del problema. In entrambi i casi, per coerenza, il sistema restituisce comunque una lista vuota.

GUIController

```
@GetMapping("/getFriendlist")
public ResponseEntity<List<Map<String, String>>> getFriendlist(
    @CookieValue(name = "jwt", required = false) String jwt) {
try {
    // 1. Verifica il JWT
    if (jwt == null || jwt.isEmpty()) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(null); // Utente non autenticato
    }

    // 2. Decodifica del JWT
    Integer userId = extractUserIdFromJwt(jwt);
    if (userId == null) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null); // JWT non valido
    }

    // 3. Chiamata al T23Service per ottenere la lista amici
    List<Map<String, String>> friendlist = t23Service.getFriendlist(userId.toString());
    if (friendlist == null || friendlist.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null); // Nessun amico trovato
    }

    return ResponseEntity.ok(friendlist); // Restituisci la lista amici
} catch (Exception e) {
    // 4. Log degli errori per debugging
    System.err.println("Errore durante il recupero della lista amici: " + e.getMessage());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
}
}
```

Approfondiamo il funzionamento del metodo `getFriendlist` del `GUIController`, che rappresenta uno degli snodi principali nel processo di recupero della lista amici. Questo metodo è mappato sull'endpoint `/get-Friendlist` ed è progettato per gestire la richiesta del FrontEnd, garantendo un flusso controllato e sicuro per restituire la lista degli amici associati all'utente.

La prima operazione svolta riguarda la verifica del JWT. Come sempre, se il token è assente o vuoto, il sistema interrompe l'esecuzione restituendo uno status HTTP 401 (UNAUTHORIZED), informando il FrontEnd che l'utente non ha effettuato l'autenticazione.

Una volta ottenuto un ID utente valido, il metodo si interfaccia con il `T23Service`, delegando a quest'ultimo la responsabilità di recuperare

la lista amici dal database. Questa chiamata al servizio rappresenta la comunicazione con i livelli sottostanti descritta nel sequence diagram, in cui il servizio interroga il Controller e successivamente il modulo Friend per elaborare i dati richiesti. Nel caso in cui il servizio restituisca una lista vuota o nulla, il metodo invia una risposta con uno status HTTP 404 (NOT FOUND), segnalando che l'utente non ha amici registrati.

Se invece il servizio ritorna una lista valida, questa viene restituita al FrontEnd come una risposta di successo con status HTTP 200 (OK), formattata come una lista di mappe contenenti i dettagli degli amici (come ID, nickname e avatar), proprio come visto nel ciclo di elaborazione del sequence diagram.

Infine, il metodo prevede una gestione delle eccezioni attraverso un blocco try-catch, utile per affrontare eventuali errori imprevisti che possono verificarsi durante l'elaborazione. In tal caso, il sistema registra un messaggio di errore nel log per facilitare il debugging e invia una risposta con status HTTP 500 (INTERNAL SERVER ERROR), segnalando un malfunzionamento interno.

T23Service

```
public List<Map<String, String>> getFriendlist(String userId) {
    final String endpoint = "/getFriendlist";

    MultiValueMap<String, String> queryParams = new LinkedMultiValueMap<>();
    queryParams.add("userId", userId);

    try {
        System.out.println("Calling endpoint: " + BASE_URL + endpoint);
        return callRestGet(endpoint, queryParams, new ParameterizedTypeReference<List<Map<String, String>>>() {});
    } catch (Exception e) {
        System.err.println("Errore durante la chiamata a " + endpoint + ": " + e.getMessage());
        return new ArrayList<>(); // Ritorna una lista vuota come fallback
    }
}
```

All'interno del metodo `getFriendlist`, il servizio prende come input l'ID utente e si occupa di inoltrare la richiesta GET verso l'endpoint remoto `/getFriendlist`. Il primo passo consiste nella costruzione dei parametri della query, utilizzando un oggetto `MultiValueMap` che associa la chiave `"userId"` al valore ricevuto. Questa configurazione consente di inviare in maniera corretta i dati richiesti dal Controller del T23 per elaborare la lista degli amici.

Una volta preparata la richiesta, il metodo stampa a console un messaggio di log che indica quale endpoint verrà contattato. Questo passaggio ci è stato utile sia per monitorare l'operazione in corso che per facilitare l'identificazione di eventuali problemi. La vera e propria chiamata REST viene eseguita tramite il metodo `callRestGet`, che sfrutta la classe `ParameterizedTypeReference` per deserializzare automaticamente la risposta JSON ricevuta in una struttura dati `List<Map<String, String>>`. Ogni map rappresenta i dettagli di un singolo amico, come l'ID, il nickname e l'avatar.

Nel caso in cui la chiamata all'endpoint fallisca per qualsiasi motivo (ad esempio, problemi di rete, errori nel servizio remoto o risposta malformata), il metodo cattura l'eccezione all'interno di un blocco try-catch. Viene stampato un messaggio di errore a console, indicante l'endpoint e la natura del problema. Per garantire robustezza al sistema e una gestione controllata dell'errore, il metodo restituisce una lista vuota come fallback, evitando così blocchi o risposte incomplete al chiamante.

Controller (T23)

```
@GetMapping("/getFriendlist")
public ResponseEntity<List<Map<String, String>>> getFriendlist(@CookieValue(name = "jwt", required = false) String jwt) {
    try {
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(null);
        }

        byte[] decodedUserObj = Base64.getDecoder().decode(jwt.split("\\.")[1]);
        String decodedUserJson = new String(decodedUserObj, StandardCharsets.UTF_8);

        ObjectMapper mapper = new ObjectMapper();
        @SuppressWarnings("unchecked")
        Map<String, Object> map = mapper.readValue(decodedUserJson, Map.class);
        int userId = Integer.parseInt(map.get("userId").toString());

        // Recupera la lista degli amici con i dettagli (nickname e avatar) dal repository
        List<Map<String, Object>> friends = friendRepository.findFriendDetailsByUserId(userId);

        // Trasforma i dati degli amici in una lista di mappe da restituire come JSON
        List<Map<String, String>> friendListResponse = new ArrayList<>();
        for (Map<String, Object> friend : friends) {
            Map<String, String> friendData = new HashMap<>();
            friendData.put("nickname", friend.get("nickname").toString());
            friendData.put("avatar", friend.get("avatar").toString()); // Recuperato dalla tabella students
            friendData.put("friendId", friend.get("friendId").toString());
            friendListResponse.add(friendData);
        }
    }

    return ResponseEntity.ok(friendListResponse);
} catch (Exception e) {
    e.printStackTrace();
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
}
}
```

Il metodo getFriendlist nel Controller inizia ricevendo il JWT dal cookie della richiesta HTTP. Dopo aver verificato che il token non

sia nullo o vuoto, il metodo procede con la sua decodifica. La parte payload del JWT, che contiene le informazioni sull’utente, viene estratta e decodificata utilizzando Base64. Una volta ottenuta la stringa JSON decodificata, questa viene convertita in una mappa tramite l’ObjectMapper di Jackson, che consente di ottenere l’ID utente.

Con l’ID utente in mano, il Controller si occupa di interrogare il repository (friendRepository) per ottenere i dettagli della lista amici associata a quell’ID. Il repository restituisce una lista di risultati grezzi, dove ciascun elemento contiene le informazioni necessarie, come il nickname, l’avatar e l’ID dell’amico. La variabile friends infatti, è una lista di mappe che arriva dal metodo findFriendDetailsById del friendRepository, quindi non utilizziamo i classici metodi get e set forniti dalla classe Friend.

Il Controller elabora questi risultati trasformandoli in una lista di mappe di tipo `<String, String>`, dove vengono inserite solo le informazioni rilevanti per il client: il nickname, l’avatar e l’ID dell’amico. Questa trasformazione garantisce che la risposta sia chiara e pronta per essere restituita al FrontEnd.

Infine, se tutto il processo si conclude correttamente, il metodo restituisce una risposta HTTP 200 OK, contenente la lista amici formattata. In caso di errore durante l’intero flusso, come problemi nella decodifica del JWT o nell’accesso ai dati, l’eccezione viene catturata, stampata per il debugging e viene restituita una risposta HTTP 500

INTERNAL SERVER ERROR.

FriendRepository

La novità della sezione amici sta nel fatto che essa interagisce con una nuova sezione del dataset MySQL, userfriends, da noi aggiunta. Per supporo a tale modifica, è stato quindi creata FriendRepository, che assieme a Friend.java e FriendId.java consente l'accesso al database per le varie funzionalità.

In generale, la FriendRepository è un'interfaccia di tipo Spring Data JPA Repository. Questo tipo di file è progettato per gestire l'interazione con il database, fornendo metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) e, optionalmente, query personalizzate, come quella utilizzata nel nostro caso per accedere alle informazioni degli amici dato un id, mostrata in foto.

```
@Repository
public interface FriendRepository extends JpaRepository<Friend, FriendId> {

    // Metodo per ottenere la lista degli amici con i dettagli (nickname e avatar) dalla tabella students
    @Query(value = "SELECT uf.friend_id AS friendId, s.nickname AS nickname, s.avatar AS avatar " +
        "FROM user_friends uf " +
        "JOIN students s ON uf.friend_id = s.ID " +
        "WHERE uf.user_id = :userId", nativeQuery = true)
    List<Map<String, Object>> findFriendDetailsByUserId(@Param("userId") int userId);
```

4.3.2 Aggiungi Amico

Cerca Amico

Cerca

Amico trovato: verdi13 (ID: 4)

Aggiungi Amico

Cerca Amico

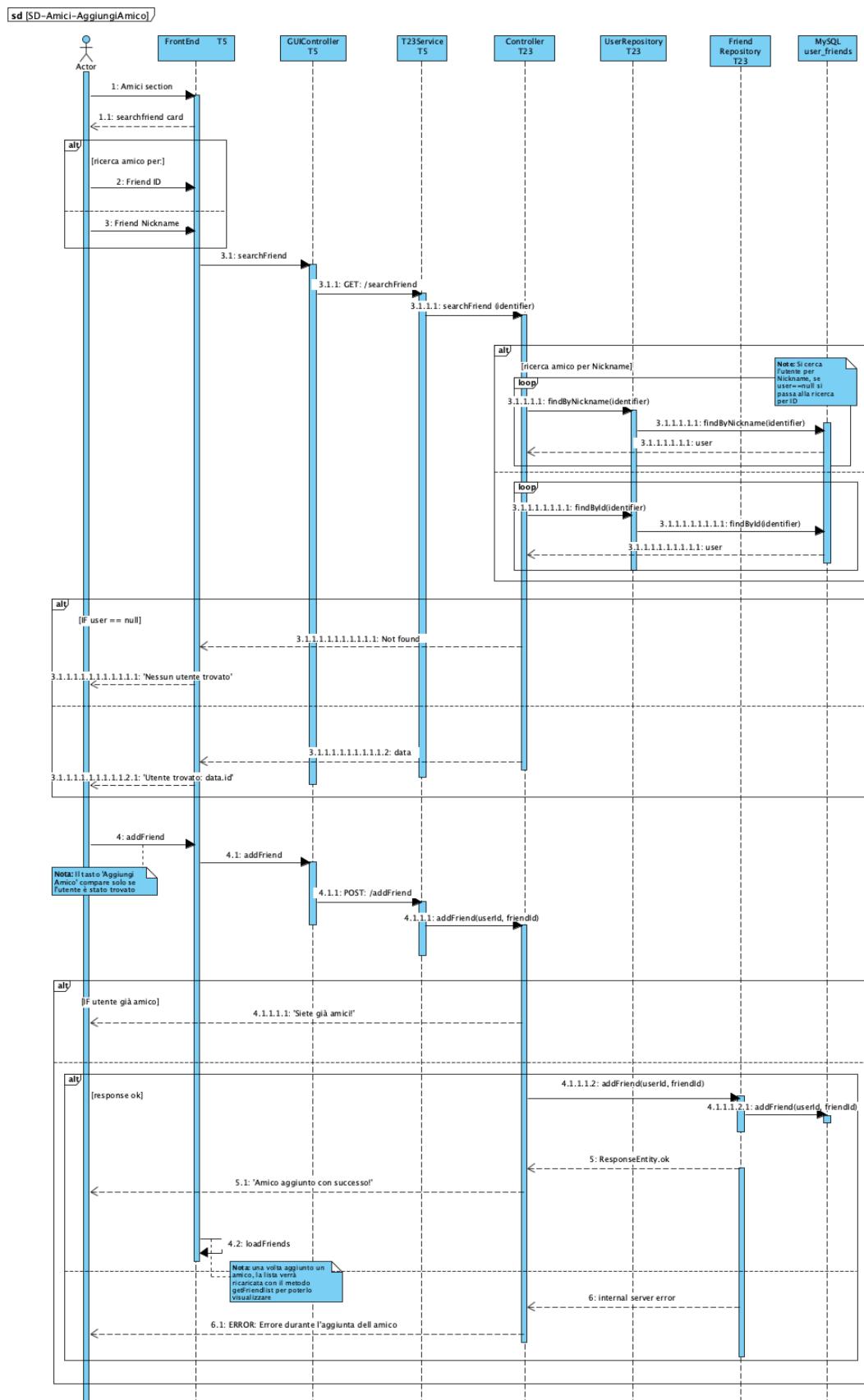
Cerca

Amico trovato: verdi13 (ID: 4)

Aggiungi Amico

Questa funzionalità come detto consente di aggiungere un amico tramite l'apposita box di ricerca: l'utente puo cercare altri giocatori tramite ID o Nickname e una volta trovato cliccare su aggiungi per far comparire il nome nella propria lista di amici.

Sequence Diagram - Aggiungi Amico



Il flusso inizia con l'utente che accede alla sezione amici nell'applicazione e seleziona la funzione di ricerca per aggiungere un nuovo amico. La "searchFriend card" avvia una richiesta da parte del frontend per trovare un amico utilizzando un identificativo specifico. Tale identificativo può essere un nickname o un ID amico. Questa richiesta viene inoltrata al controller di T5 attraverso un metodo di ricerca search-Friend.

Successivamente, il controller trasmette la richiesta ancora una volta all'interfaccia per il T23, dove avviene l'interfaccia al repository di gestione utenti (UserRepository T23). A questo livello, il sistema esegue due tipi di ricerca in sequenza: prima cerca un utente tramite il nickname, e se non trova risultati, procede con una ricerca basata sull'ID dell'utente. Entrambe le ricerche avvengono attraverso query specifiche al repository amici (Friend Repository T23) che interagisce direttamente con il database MySQL (MySQL user friends).

Se l'utente non viene trovato, viene restituito un messaggio "Nessun utente trovato" al frontend, chiudendo il processo. Al contrario, se l'utente viene trovato, il sistema procede con un'altra verifica per determinare se l'utente selezionato è già amico dell'utente corrente. Questa fase è implementata dal repository, che controlla la relazione tramite query mirate.

Se l'utente non è ancora amico, il flusso avanza alla fase di aggiunta dell'amico. Il controller esegue una chiamata POST al servizio

per aggiungere l'amico, inoltrando il userId e il friendId. Il repository amici aggiorna il database aggiungendo il nuovo amico. Se l'operazione ha successo, viene inviato un messaggio di conferma "Amico aggiunto con successo" al frontend. A questo punto, il frontend esegue un caricamento aggiornato della lista amici, assicurando che il nuovo amico venga visualizzato all'utente.

Il diagramma prevede anche casi di errore: se l'utente selezionato è già presente nella lista amici, il sistema restituisce il messaggio "Siete già amici", impedendo di aggiungere un duplicato. Inoltre, in caso di errori interni durante l'aggiornamento del database, viene restituito un errore generico "Errore durante l'aggiunta dell'amico".

Senza aggiungere dettagli, essendo ormai noto il procedimento, vengono riportati i codici relativi alla ricerca dell'utente e all'aggiunta di quest'ultimo nella lista.

GUIController

```

@GetMapping("/searchFriend")
public ResponseEntity<Map<String, String>> searchFriend(
    @RequestParam String identifier,
    @CookieValue(name = "jwt", required = false) String jwt
) {
    try {
        // Verifica autenticazione
        Integer userId = extractUserIdFromJwt(jwt);
        if (userId == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }

        // Cerca l'utente tramite T23Service
        Map<String, String> user = t23Service.searchFriend(identifier);
        if (user == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }

        return ResponseEntity.ok(user);
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

```

```

@PostMapping("/addFriend")
public ResponseEntity<String> addFriend(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam Integer friendId) {
    try {
        Integer userId = extractUserIdFromJwt(jwt);
        if (userId == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Token non valido.");
        }

        String response = t23Service.addFriend(userId.toString(), friendId.toString());
        if (response == null) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore durante l'aggiunta dell'amico.");
        }

        return ResponseEntity.ok(response);
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore interno del server.");
    }
}

```

T23Service

```
public Map<String, String> searchFriend(String identifier) {
    final String endpoint = "/searchFriend"; // Endpoint in T23

    MultiValueMap<String, String> queryParams = new LinkedMultiValueMap<>();
    queryParams.add("identifier", identifier);

    try {
        return callRestGet(endpoint, queryParams, new ParameterizedTypeReference<Map<String, String>>() {});
    } catch (Exception e) {
        System.err.println("Errore durante la ricerca dell'amico: " + e.getMessage());
        return null;
    }
}

public String addFriend(String userId, String friendId) {
    final String endpoint = "/addFriend";
    MultiValueMap<String, String> payload = new LinkedMultiValueMap<>();
    payload.add("userId", userId);
    payload.add("friendId", friendId);

    try {
        return callRestPost(endpoint, payload, null, String.class);
    } catch (Exception e) {
        System.err.println("Errore durante l'aggiunta dell'amico: " + e.getMessage());
        return null;
    }
}
```

Controller (T23)

```

@GetMapping("/searchFriend")
public ResponseEntity<Map<String, String>> searchFriend(@RequestParam String identifier) {
    try {
        // Cerca l'utente tramite nickname o ID
        User user = userRepository.findByNickname(identifier);
        if (user == null) {
            try {
                user = userRepository.findById(Integer.parseInt(identifier));
            } catch (NumberFormatException e) {
                return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
            }
        }

        if (user == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }

        // Restituisci i dettagli dell'utente trovato
        Map<String, String> userDetails = new HashMap<>();
        userDetails.put("id", user.getId().toString());
        userDetails.put("nickname", user.getNickname());
        return ResponseEntity.ok(userDetails);
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

```

```

@PostMapping("/addFriend")
public ResponseEntity<String> addFriend(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam Integer friendId) {
    try {
        // Estrai l'ID dell'utente dal JWT
        Integer userId = extractUserIdFromJwt(jwt);
        if (userId == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Token non valido.");
        }

        // Controllo esistenza amicizia
        boolean exists = friendRepository.customExistsById(userId, friendId);
        if (exists) {
            return ResponseEntity.badRequest().body("Amicizia già esistente.");
        }

        // Inserisci amicizia
        friendRepository.addFriend(userId, friendId);

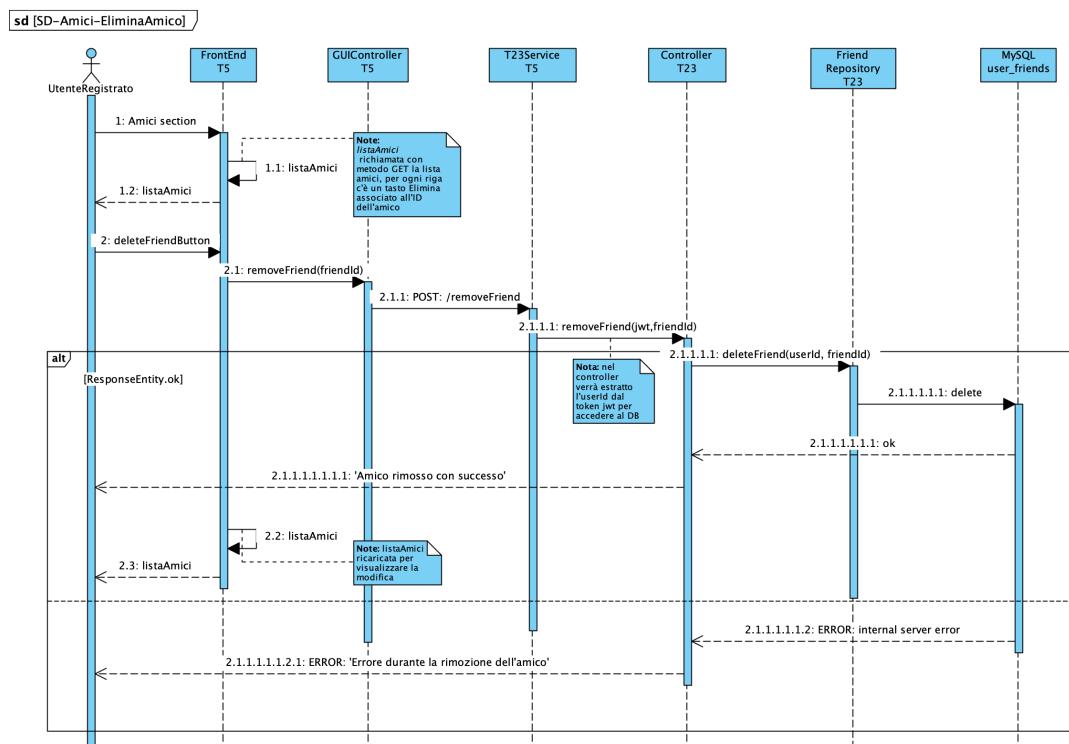
        return ResponseEntity.ok("Amico aggiunto con successo.");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore durante l'aggiunta dell'amico: " +
    }
}

```

4.3.3 Elimina Amico

Il servizio Elimina Amico consente di rimuovere dalla lista un amico precedentemente aggiunto, grazie all'apposito tasto presente nella lista amici accanto al preciso utente che si vuole eliminare.

Sequence Diagram - Elimina Amico



Il flusso si avvia con la visualizzazione della lista amici nel frontend. Questa lista viene ottenuta tramite una richiesta GET (si rimanda alla documentazione nelle pagine precedenti) e include un pulsante di eliminazione associato all'ID di ciascun amico. Quando l'utente seleziona il pulsante deleteFriendButton, viene generata una richiesta al GUIController T5 per rimuovere l'amico specificato tramite il metodo

removeFriend(friendId).

Il GUIController di T5, a sua volta, inoltra una richiesta POST al servizio (T23Service T5) utilizzando il metodo */removeFriend*. Qui, il servizio passa la richiesta al controller (Controller T23) includendo il jwt per l'autenticazione e l'ID dell'amico da rimuovere. A questo livello, il controller estrae lo userId dal token JWT per identificare correttamente l'utente corrente e interagire con il database.

Successivamente, il Controller T23 chiama il metodo deleteFriend(userId, friendId) del repository amici (Friend T23), che si occupa di comunicare con il database MySQL (userfriends). Viene eseguita l'operazione delete per rimuovere l'amico corrispondente all'ID specificato.

A questo punto, il diagramma prevede due possibili esiti:

Se l'eliminazione dal database avviene con successo, viene restituita una conferma al controller (ok), che a sua volta invia una risposta positiva al servizio e al frontend con il messaggio "Amico rimosso con successo". Il frontend procede quindi con un aggiornamento della lista amici per riflettere la modifica effettuata. Questo aggiornamento avviene tramite una nuova richiesta GET per ricaricare la lista amici e mostrarla all'utente.

Se si verifica un problema durante l'eliminazione dell'amico nel database (ad esempio, un errore interno del server), viene restituita una risposta di errore al controller. Il controller trasmette l'errore al servizio, che lo inoltra al frontend con il messaggio "Errore durante

la rimozione dell'amico". A questo punto, il frontend è in grado di notificare l'utente dell'impossibilità di completare l'operazione.

GUIController

```
@PostMapping("/removeFriend")
public ResponseEntity<String> removeFriend(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam Integer friendId) {
    try {
        // Verifica che il token JWT esista
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Utente non autenticato.");
        }

        // Chiama il T23Service per rimuovere l'amico
        Boolean result = t23Service.removeFriend(jwt, friendId);

        if (result != null && result) {
            return ResponseEntity.ok("Amico rimosso con successo.");
        } else {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore durante la rimozione dell'amico.");
        }
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore interno del server.");
    }
}
```

T23Service

```
public Boolean removeFriend(String jwt, Integer friendId) {
    final String endpoint = "/removeFriend";

    // Prepara i parametri per la richiesta
    MultiValueMap<String, String> formData = new LinkedMultiValueMap<>();
    formData.add("friendId", friendId.toString());

    Map<String, String> headers = new HashMap<>();
    headers.put("Authorization", "Bearer " + jwt);

    try {
        // Effettua la chiamata POST al backend (T23)
        return callRestPost(endpoint, formData, headers, Boolean.class);
    } catch (Exception e) {
        System.err.println("Errore durante la rimozione dell'amico: " + e.getMessage());
        return false;
    }
}
```

Controller (T23)

```
@PostMapping("/removeFriend")
public ResponseEntity<String> removeFriend(
    @CookieValue(name = "jwt", required = false) String jwt,
    @RequestParam Integer friendId) {
    try {
        if (jwt == null || jwt.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Token non valido o mancante.");
        }

        // Estrai l'ID utente dal token JWT
        Integer userId = extractUserIdFromJwt(jwt);
        if (userId == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("ID utente non valido.");
        }

        // Crea un oggetto FriendId
        FriendId friendIdObj = new FriendId(userId, friendId);

        if (!friendRepository.customExistsById(userId, friendId)) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Amicizia non trovata.");
        }

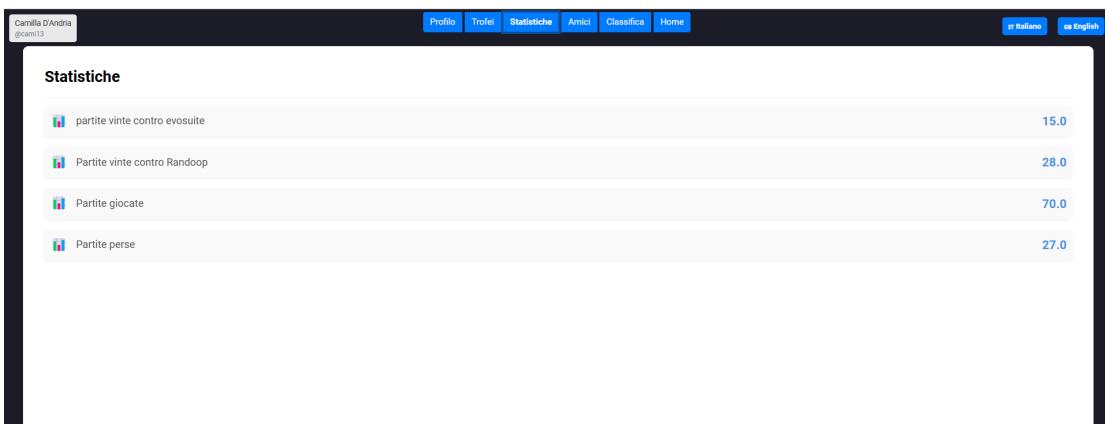
        // Rimuovi l'amicizia dal database
        friendRepository.deleteFriend(userId, friendId);

        return ResponseEntity.ok("Amico rimosso con successo.");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore interno del server.");
    }
}
```

FriendRepository

```
@Modifying
@Transactional
@Query("DELETE FROM Friend f WHERE f.id.userId = :userId AND f.id.friendId = :friendId")
void deleteFriend(@Param("userId") Integer userId, @Param("friendId") Integer friendId);
```

4.4 Sezione Statistiche



La sezione statistiche consente di visualizzare tutti i dati di gioco dell’utente.

Il salvataggio dati relativi alle statistiche e quindi il prelievo delle informazioni nella sezione dedicata al profilo (T5) da quella riservata al Game (T4), contenente la Game Repository, sono state un requisito fondamentale per poter poi fornire l’interfaccia grafica dei dati all’utente. In questo lo scopo del nostro gruppo è stato quello di aggiornare ciò che ci è stato lasciato in eredità dai nostri colleghi, in quanto queste funzionalità erano già presenti, aggiungendo la parte dell’interfaccia ai dati nel nuovo profilo utente creato. In questo senso, il nostro lavoro è stato decommentare alcune righe di codice che non permettevano il salvataggio delle partite e quindi di conseguenza delle statistiche di gioco, e creare l’interfaccia.

Al fine di comprendere la struttura e il funzionamento del caso d’uso, ci è stato utile lo studio delle documentazioni A3-T4 relativa al salvataggio dei dati e al loro aggiornamento, e T4-G18 per il calcolo e

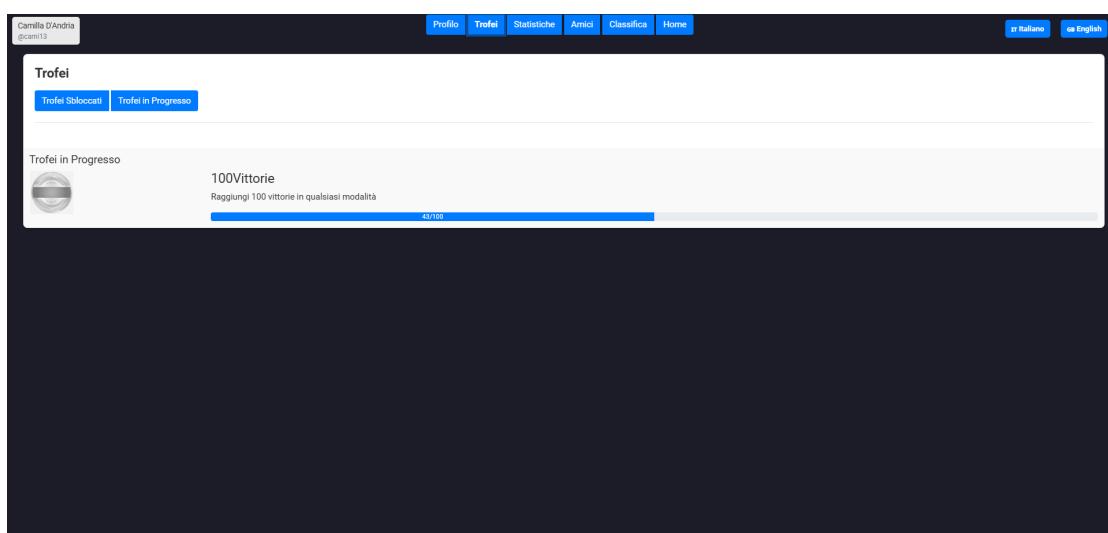
il prelievo delle statistiche. A tale proposito si rimanda allo studio di queste ultime nel caso di necessità di future modifiche.

Nel nostro caso quindi, i dati delle statistiche erano già disponibili a livello T5, raccolte mediante l'interfaccia T4Service e altri files specifici riportati in figura, tutti disponibili al percorso

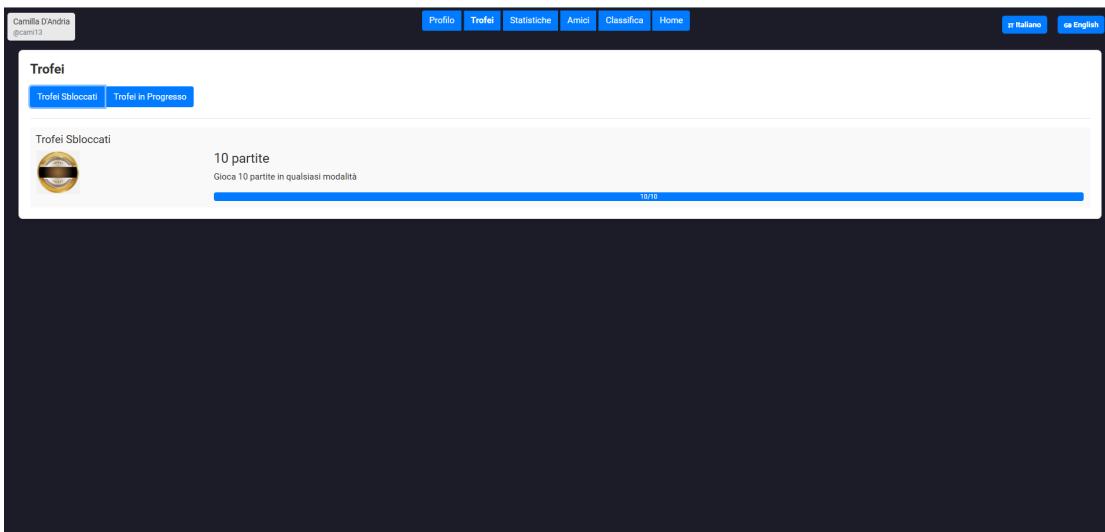
A13/T5-G2/t5/src/main/java/com/g2/Interfaces



4.5 Sezione Achievements



CHAPTER 4. TASK R1 - IMPLEMENTAZIONE



La sezione Achievements è pensata per consentire agli utenti di visualizzare i propri progressi di gioco tramite un'interfaccia moderna e dinamica. Attraverso questa sezione, l'utente può verificare quali trofei ha sbloccato, legati a specifici obiettivi di gioco, e quali restano ancora da raggiungere. La sezione è suddivisa in due pagine, accessibili tramite appositi pulsanti: la prima consente di visualizzare gli obiettivi raggiunti, caratterizzati da un badge colorato e una barra dei progressi al 100/100; la seconda, invece, mostra gli achievements ancora da sbloccare, visibili in grigio e con una barra di avanzamento che riflette il progresso compiuto fino a quel momento.

Per quanto riguarda gli achievements, il codice ereditato prevedeva già l'implementazione di questa funzionalità. Tuttavia, il nostro intervento principale è stato quello di riallocare la funzione all'interno del profilo, sfruttando sezioni dinamiche, e di rimodernarla aggiungendo un elemento interattivo che permette di navigare tra le due sottosezioni (trofei sbloccati e non).

Per l'aggiunta di nuovi achievements, così come per le statistiche, è stato effettuato un accesso al profilo amministratore, dal quale è stato possibile gestire l'inserimento di nuovi obiettivi e monitorare le relative statistiche.

4.6 Testing

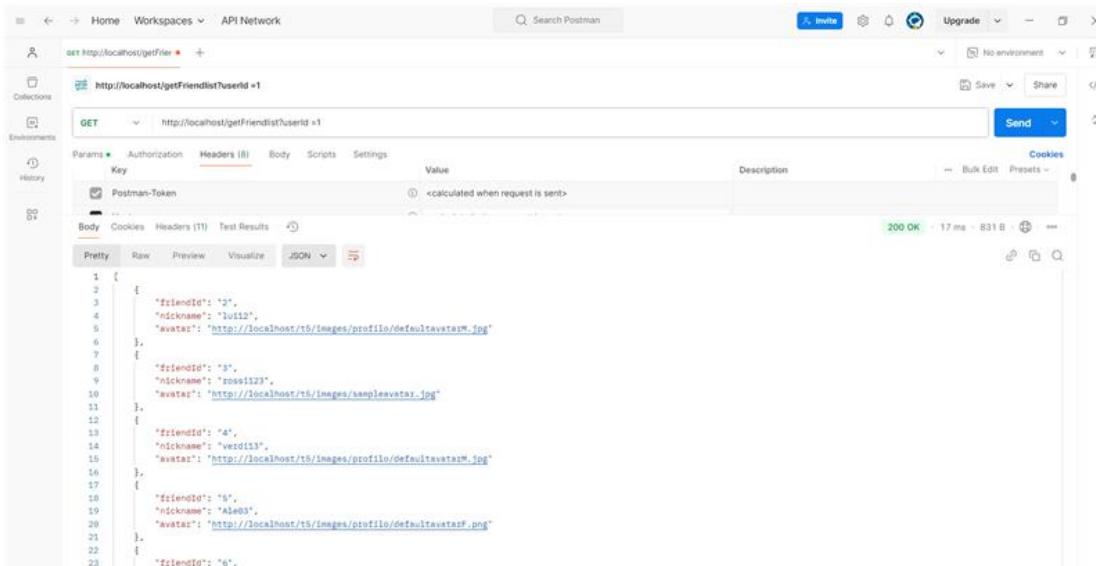
Per il processo di testing delle API, è stato utilizzato **Postman** per verificare il corretto funzionamento dei metodi sviluppati. Le richieste sono state inviate per testare vari endpoint con differenti parametri e payload, assicurandosi che le risposte fossero conformi alle aspettative.

Durante la fase di testing, sono stati controllati i seguenti aspetti:

- **Lo stato delle risposte HTTP**, per garantire l'affidabilità delle chiamate.
- **La struttura e il contenuto dei dati restituiti**, validando la coerenza delle risposte rispetto ai requisiti.
- **L'integrazione tra i metodi**, verificando la corretta interazione tra componenti e il rispetto dei flussi previsti.

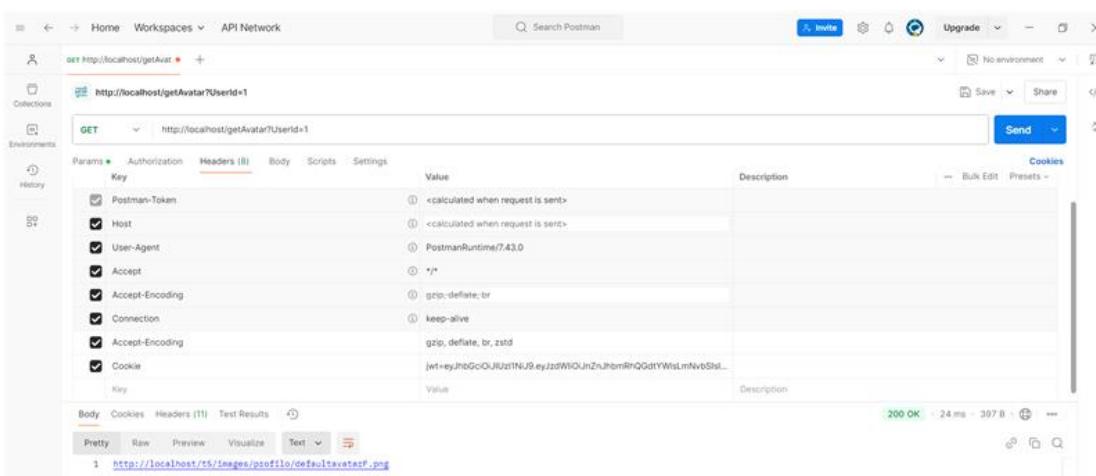
Questa attività ha permesso di individuare e risolvere eventuali anomalie, assicurando la stabilità e la robustezza dei metodi implementati.

CHAPTER 4. TASK R1 - IMPLEMENTAZIONE



The screenshot shows a Postman interface with a successful API call. The URL is `http://localhost/getFriendlist?userId=1`. The response body is a JSON array of friend objects:

```
1 [
2   {
3     "friendId": "3",
4     "nickname": "luul2",
5     "avatar": "http://localhost/t5/images/profilo/defaultavatarm.jpg"
6   },
7   {
8     "friendId": "3",
9     "nickname": "rossi123",
10    "avatar": "http://localhost/t5/images/sampleavatar.jpg"
11 },
12   {
13     "friendId": "4",
14     "nickname": "verdi11",
15     "avatar": "http://localhost/t5/images/profilo/defaultavatarm.jpg"
16 },
17   {
18     "friendId": "5",
19     "nickname": "Alez03",
20     "avatar": "http://localhost/t5/images/profilo/defaultavatraf.png"
21 },
22   {
23     "friendId": "6",
24 }
```



The screenshot shows a Postman interface with a successful API call. The URL is `http://localhost/getAvatar?userId=1`. The response body is a single URL:

```
1 http://localhost/t5/images/profilo/defaultavatraf.png
```