



UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

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

Elaborato di Software Architecture Design

Documentazione Progetto: task R2

Anno Accademico 2024-2025

Prof. Anna Rita Fasolino

Membri del gruppo

Esposito Andrea M63001650 andrea.esposito66@studenti.unina.it

Papale Livio M63001824 li.papale@studenti.unina.it

Sorrentino Carmine M63001819 carmine.sorrentino4@studenti.unina.it

Vallefuro Roberto M63001443 roberto.vallefuro@studenti.unina.it

[Aggiungere link alla repository consegnata]

Indice

1 Punto di Partenza	1
1.1 Descrizione del task	1
1.1.1 Stato Iniziale del Progetto ENACTEST	1
1.1.2 Processo di sviluppo applicato	5
1.1.3 Strumenti utilizzati	6
2 Analisi dei Requisiti	8
2.0.1 User Stories	8
2.1 Analisi dei requisiti	10
2.1.1 Requisiti funzionali	10
2.1.2 Requisiti sui Dati e Non Funzionali	12
2.1.3 Analisi d'impatto	12
2.2 Scenari e diagrammi di attività	13
2.2.1 Visualizzazione della classifica	14
2.2.2 Scelta di un ordinamento della classifica	14
2.3 Iterazioni	15
2.3.1 Iterazione 1	15
2.3.2 Iterazione 2	16
2.3.3 Iterazione 3	17

3 Analisi componente T56	19
3.0.1 Diagramma di contesto	19
3.1 Stato iniziale componente T56	21
4 Modifiche strutturali al Task 56	23
4.0.1 Diagramma dei componenti	23
4.0.2 Interfaccia PlayerRepository	24
4.0.3 Diagramma delle Classi	25
4.0.4 Descrizione del Nuovo Database	27
4.0.5 Issues Rilevate	28
4.0.6 Diagramma dei Package	29
4.0.7 Sequence Diagram	31
5 Implementazione della soluzione	34
5.1 Implementazione Database	34
5.1.1 Aggiunta del Container	34
5.1.2 Collegamento del Database con il Server Spring	35
5.1.3 Modifiche al Front-End	40
5.1.4 Integrazione Front-End Classifica con Back-End	43
5.1.5 Modifiche al Back-End	51
6 Deployment diagram finale complessivo dell'architettura	53
7 Testing	54
7.1 Testing End to End	54
7.1.1 BaseTest	55
7.1.2 NavigationTest	56
7.1.3 LeaderboardScoreOrderTest	56

7.1.4	LeaderboardWinsOrderTest	57
7.1.5	LeaderboardTotalMatchesOrderTest	58
8	Sviluppi futuri	60

Chapter 1

Punto di Partenza

1.1 Descrizione del task

Gestire la creazione e visualizzazione di **classifiche** dei vari giochi, verosimilmente da visualizzare attraverso la *homepage*. Esempi di classifiche: classifica delle partite giocate, partite vinte, classi testate....

1.1.1 Stato Iniziale del Progetto ENACTEST

Panoramica

Il progetto ENACTEST rappresenta una piattaforma avanzata per il testing di codice online, destinata a facilitare l'interazione tra amministratori e utenti attraverso una serie di funzionalità dinamiche e interattive. Gli amministratori sono dotati di strumenti per:

- Creare e gestire classi di test;
- Definire scalate di difficoltà;

- Assegnare achievements;
- Monitorare una leaderboard che raccoglie i dati di tutti gli utenti registrati.

Gli **utenti registrati**, d'altro canto, hanno la possibilità di:

- Visualizzare il proprio profilo, attualmente in una forma preliminare;
- Interagire con due robot di testing, Randoop ed Evosuite, per affrontare sfide basate su classi fornite dagli amministratori.

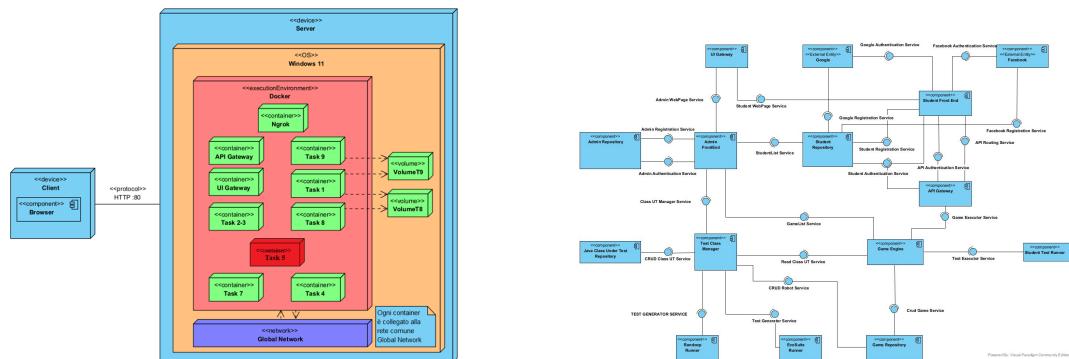
Architettura Tecnica

Il backend (fig. 1.1) del sistema è organizzato secondo il modello dei microservizi. Questa scelta architettonica promuove l'efficienza nello sviluppo delle applicazioni web, garantendo a ciascun sottosistema l'accesso ai dati necessari per eseguire le proprie funzioni e offrire capacità di calcolo agli altri sistemi, minimizzando l'accoppiamento tra i diversi componenti.

Di seguito, una tabella descrive i sottosistemi e le loro funzioni:

Stato originario del sottosistema T56

Dopo l'ultima iterazione di aggiornamento, denominata A13_Refactoring_T56, le funzionalità dei singoli container T5 e T6 sono state consolidate in un unico nuovo container. Sfortunatamente, nella fase di refactoring



(a) Diagramma di deployment del sistema di partenza.

(b) Diagramma dei componenti.

Figure 1.1: Diagrammi del sistema ENACTEST.

Sottosistema	Funzione
T1	Gestisce l'interfaccia amministrativa e mantiene un database MongoDB con i dati degli amministratori.
(T23)	Si occupa dell'autenticazione degli utenti, facilitando la registrazione, il cambio di password e l'accesso.
T4	È incaricato della registrazione delle sessioni di gioco, ma attualmente soffre di limitazioni dovute a una documentazione incompleta e a sfide legate al linguaggio di programmazione utilizzato (Go).
(T56)	Gestisce l'interfaccia utente e la logica di gioco.
T7	Supervisiona la compilazione e l'esecuzione del codice generato dagli utenti.
T8	È responsabile del controllo del robot Evosuite.
T9	Amministra le operazioni del robot Randoop.

non sono stati prodotti i diagrammi dei casi d'uso del nuovo sottosistema. Poiché il nostro progetto è focalizzato principalmente sulla creazione di una pagina web, viene presentato il diagramma dei casi d'uso del sottosistema T5 (fig. 1.2).

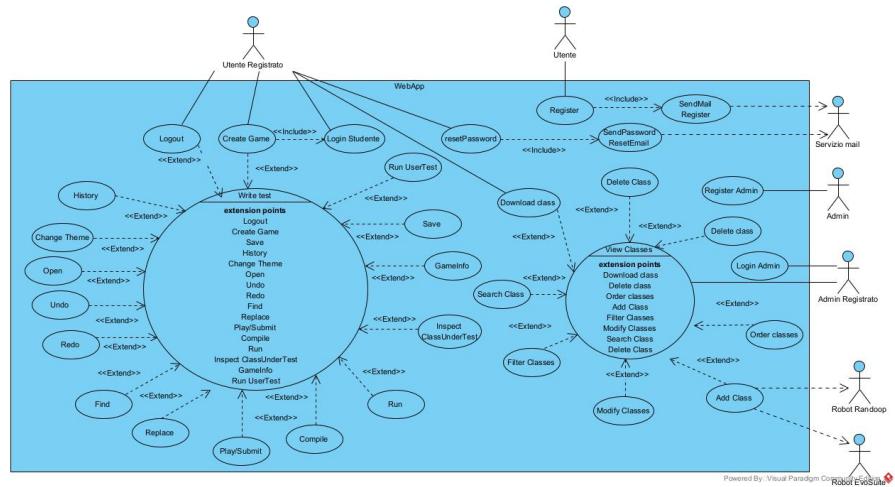


Figure 1.2: Diagramma dei casi d'uso di alto livello.

Nonostante i casi d'uso *write test* e *view classes* non siano più presenti nell'attuale configurazione del progetto, il diagramma fornisce comunque una visione chiara delle operazioni che possono essere eseguite da un utente generico e quelle riservate a un utente registrato. Questo diagramma può quindi essere considerato come un riferimento ad alto livello per le azioni eseguibili dagli utenti sul sistema.

Diagramma di Contesto

Viene presentato il diagramma di contesto che, considerando il server come un elemento "black box", rappresenta tutte le interazioni possibili che ogni entità esterna può effettuare con esso. Le azioni modificate a seguito delle nostre revisioni sono indicate in blu, mentre le nuove funzionalità sono evidenziate in verde.

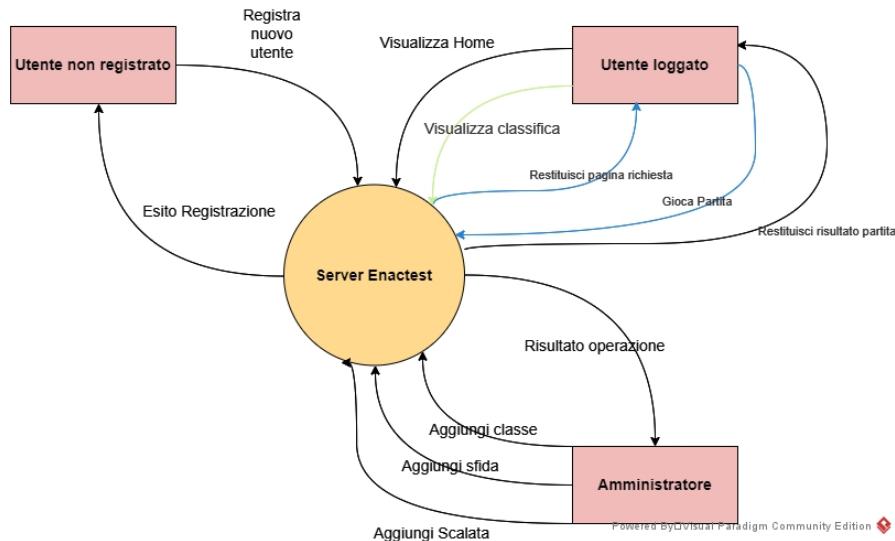


Figure 1.3: Diagramma di contesto.

Nome Azione	Modifica/Aggiunta
Gioca Partita	Adesso il client web dell'utente passa all'interno del FormData la propria mail, così da poter legare i risultati della partita al giocatore.
Restituisci pagina richiesta	È possibile restituire la pagina /leaderboard.
Visualizza classifica	L'utente loggato può ora richiedere di visualizzare una pagina di classifica.

Table 1.1: Modifiche e aggiunte alle azioni nel sistema ENACTEST.

1.1.2 Processo di sviluppo applicato

È stata adottata la **metodologia agile**, utilizzando **Sprint Backlog** e riunioni giornaliere **SCRUM**. Questo approccio consente di concentrarsi sullo sviluppo pratico del codice, rispondendo rapidamente ai cambiamenti e rilasciando software funzionante in tempi brevi.

SCRUM è un *metodo* che gestisce il ciclo di sviluppo in modo iterativo, con obiettivi definiti per ogni sprint. Lo Sprint Backlog è il piano di lavoro per ogni fase, mentre il Daily Scrum è un incontro quotidiano breve per sincronizzare le attività e pianificare i prossimi passi. Le riunioni, sia in presenza che a distanza, sono state cruciali per l’analisi del problema e la pianificazione delle attività, con assegnazione dei compiti e definizione delle scadenze.

1.1.3 Strumenti utilizzati

Strumenti che abbiamo scelto di usare

Per quanto riguarda gli strumenti principali che abbiamo scelto di utilizzare, essi sono i seguenti:

- **Discord:** utilizzato per agevolare la comunicazione a distanza tra i membri del team, sia tramite videochiamate per discutere le fasi del progetto, sia per scambi rapidi di messaggi.
- **GitHub:** impiegato per il controllo delle versioni del codice e per facilitare la collaborazione all’interno del gruppo. La piattaforma consente di utilizzare un repository per archiviare i file e monitorare le revisioni del progetto.
- **Visual Paradigm:** utilizzato per la creazione di diagrammi e modelli UML, strumenti essenziali per la documentazione e per una collaborazione agile all’interno del team.

- **Ambiente di sviluppo:** per la scrittura e compilazione del codice è stato scelto **Visual Studio Code**, un editor leggero, veloce e altamente versatile. Supporta diverse estensioni e consente di lavorare con più linguaggi di programmazione e di markup in modo semplice (nel nostro caso, principalmente Java, JavaScript, CSS, HTML, XML).
- **Docker:** impiegato come ambiente di sviluppo integrato per creare, testare e distribuire l'applicativo all'interno di container.
- **Thymeleaf:** motore di template utilizzato per generare HTML in modo efficiente, garantendo che le pagine siano ottimizzate per l'interazione con i dati provenienti dai servizi.
- **Wireframe.cc:** utilizzato durante l'iterazione 1 per la creazione di un mockup della classifica.
- **Miro.com:** utilizzato per la creazione delle immagini per rappresentare gli sprint backlog relativi a ciascuna iterazione incrementale.
- **Selenium:** utilizzato per il *testing* automatizzato dell'applicazione web.
- **JUnit:** scelto come *framework* di *testing*.
- **Google Chrome:** usato come browser per accedere e testare l'applicazione.

Chapter 2

Analisi dei Requisiti

2.0.1 User Stories

Per delineare con precisione i requisiti del progetto, abbiamo adottato un approccio incentrato sulle User Stories, utilizzando la struttura "as an (attore) I want to (azione) so that (risultatobeneficio)". Questo metodo ci ha permesso di catturare le esigenze e le aspettative degli utenti finali in termini di funzionalità del sistema, offrendo una visione orientata agli obiettivi degli attori coinvolti.

Le User Stories identificate sono state:

1. "*Come giocatore, Voglio poter visualizzare una classifica dei giocatori, in modo da potermi confrontare con gli altri giocatori.*"
2. "*Come giocatore, voglio poter selezionare il criterio di ordinamento della classifica, in modo da poter vedere le mie prestazioni in diverse statistiche.*"

3. "*come giocatore, voglio poter ricercare per nome un altro giocatore, in modo da potermi confrontare con chi desidero.*"



Figure 2.1: Storie utente.

La scomposizione delle User Stories in requisiti ha fornito una base dettagliata per la progettazione dei casi d'uso, illustrando in modo preciso le interazioni tra gli utenti e il sistema. Questo approccio ha assicurato una comprensione approfondita delle funzionalità necessarie e delle aspettative degli utenti, stabilendo una solida fondazione per le successive fasi di sviluppo e progettazione del sistema.

Criteri di Accettazione I criteri di accettazione usano il modello “DATO [stato iniziale], QUANDO [azione o evento che si verifica], ALLORA [risultato atteso o comportamento desiderato]”, in modo da conoscere cosa serve implementare al fine di soddisfare la *user story*

con successo.

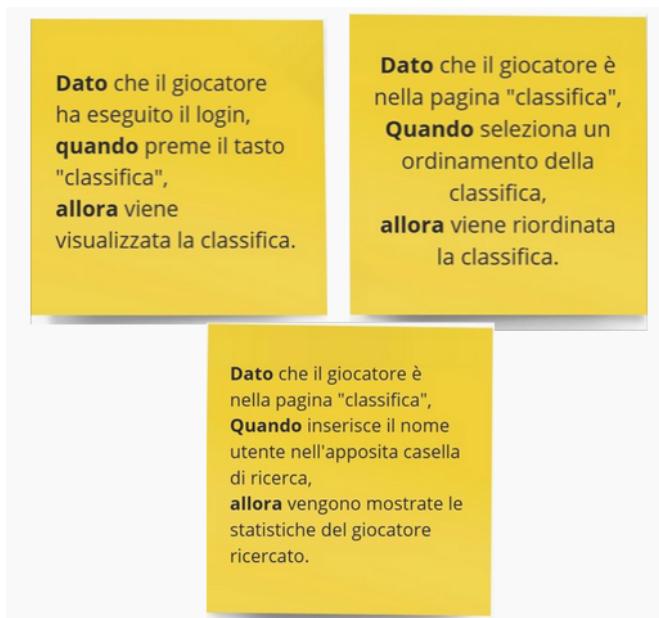


Figure 2.2: Criteri di accettazione.

2.1 Analisi dei requisiti

2.1.1 Requisiti funzionali

I requisiti funzionali rappresentano le funzionalità specifiche del sistema che sono necessarie per soddisfare le esigenze degli utenti. I dettagli di questi requisiti sono elencati nella Tabella 2.1 e includono tre tipi principali di azioni:

- Selezionare e filtrare le classifiche.
- Salvare i dati di gioco dei giocatori per consentire la ricostruzione delle classifiche su richiesta.
- Mantenere una sezione dedicata alle classifiche.

Nr.	Descrizione del Requisito
1	Il sistema deve prevedere una sezione dedicata alle classifiche.
2	Il sistema deve permettere la visualizzazione delle classifiche ai giocatori.
3	Il sistema deve offrire classifiche basate sul punteggio totale delle partite.
4	Il sistema deve offrire classifiche basate sul numero totale di partite giocate.
5	Il sistema deve offrire classifiche basate sul numero di partite vinte.
6	Il sistema deve aggiornare tutte le classifiche ogni volta che una nuova partita viene registrata.
7	Il giocatore deve poter visualizzare tutte le posizioni in classifica.
8	Il giocatore deve poter cercare altri giocatori nella classifica.
9	Il giocatore deve poter selezionare il criterio di ordinamento della classifica.
10	Il giocatore deve poter visualizzare la classifica aggiornata in qualsiasi momento.

Table 2.1: Requisiti funzionali del sistema.

2.1.2 Requisiti sui Dati e Non Funzionali

Relativamente ai requisiti sui dati e non funzionali, sono state identificate necessità specifiche riguardanti i dati da salvare per ogni giocatore e l'assicurazione che il backend sia scalabile. I dettagli sono presentati nella Tabella 2.2, che include i seguenti punti:

Nr.	Descrizione del Requisito
1	La classifica deve includere: posizione, nome del giocatore, numero totale di partite giocate, numero totale di vittorie, e punteggio totale.
2	Il sistema deve essere robusto in risposta a numerose richieste di aggiornamento delle classifiche.
3	Il sistema deve essere robusto in risposta a numerose richieste di visualizzazione delle classifiche.
4	Il sistema deve includere un database per il salvataggio delle classifiche.

Table 2.2: Requisiti sui dati e non funzionali del sistema.

2.1.3 Analisi d'impatto

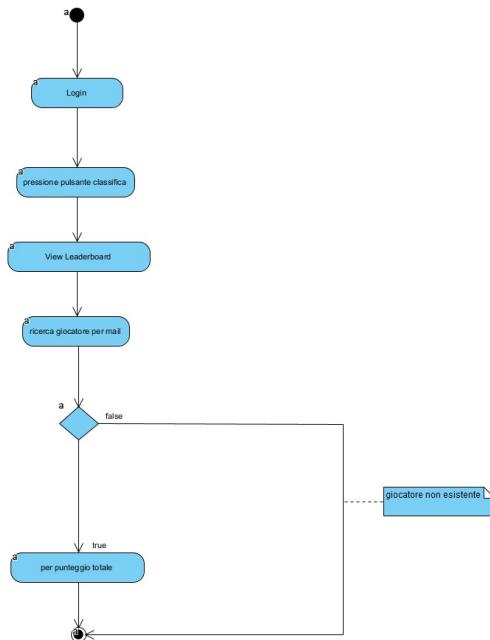
Una volta definiti i requisiti, si è proceduto con un'analisi d'impatto che ha evidenziato le componenti del sistema necessarie a essere approfondite o modificate. L'analisi dei requisiti funzionali ha rivelato che questi si concentrano principalmente su due funzionalità: il salvataggio dei dati e la visualizzazione di classifiche nel front-end. La funzionalità di visualizzazione delle classifiche è stata prontamente assegnata al modulo T56, data la sua diretta correlazione con le esigenze del progetto.

La decisione su come implementare il salvataggio dei dati è stata più complessa. Inizialmente, si è considerato l'uso dei dati già presenti nel sottosistema T4, che funge da repository per i dati delle partite. Tuttavia, a causa del malfunzionamento attuale di T4 e delle difficoltà legate alla comprensione del linguaggio di programmazione utilizzato, si è optato per una diversa strategia di implementazione. Si propone quindi l'avvio di una migrazione incrementale della gestione dei dati dal sottosistema T4 al T56. Questa scelta appare naturale, poiché consente di centralizzare la gestione dei dati delle partite nel modulo in cui queste vengono giocate e le statistiche vengono visualizzate, riducendo così l'accoppiamento precedentemente esistente tra T56 e T4. In sintesi, si può affermare che tutte le modifiche progettuali impatteranno significativamente il sottosistema T5.

2.2 Scenari e diagrammi di attività

Dalle storie utente sono starti sviluppati i seguenti scenari di funzionamento, ognuno con portata, livello, attore principale, descrizione, precondizioni, scenario principale e flusso alternativo.

2.2.1 Visualizzazione della classifica

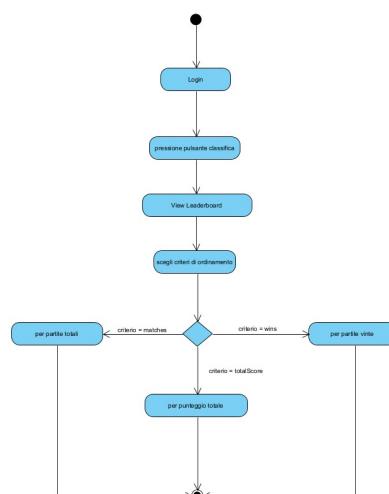


Caso d'uso:	Visualizza Classifica
Portata	Applicazione gioco
Livello	Obiettivo utente
Attore primario	Studente
Descrizione	Lo studente vuole visualizzare la classifica.
Pre-Condizioni	Lo studente è autenticato e si trova nella home.
Sequenza di eventi principale	1. Lo studente preme sul bottone classifica presente nella barra di navigazione in alto; 2. Lo studente sceglie il criterio di ordinamento;
Post-Condizioni	Il sistema mostra la <u>leaderboard</u> ordinata allo studente

Figure 2.4: Caso d'uso visualizza classifica

Figure 2.3: Activity Diagram
Ricerca

2.2.2 Scelta di un ordinamento della classifica



Caso d'uso:	ricerca giocatore
Portata	Applicazione gioco
Livello	Obiettivo utente
Attore primario	Studente
Descrizione	Lo studente vuole cercare i dati di un altro giocatore in classifica
Pre-Condizioni	Lo studente si trova nella classifica
Sequenza di eventi principale	1. Lo studente inserisce il nome del giocatore nella barra di ricerca dei giocatori.
Post-Condizioni	Il sistema mostra una classifica contenente i dati del giocatore ricercato (solo se presenti).

Figure 2.6: Caso d'uso ricerca giocatore

Figure 2.5: Activity Diagram Ordinamento

2.3 Iterazioni

2.3.1 Iterazione 1

Iterazione 1: Panoramica generale sul progetto, Studio della documentazione relativa al microservizio T5 (su cui stimiamo che ci sarà il principale impatto del nostro task, vista la presenza in esso del Page Builder con cui dovremo gestire la pagina della classifica che realizzeremo), Definizione e specifica dei requisiti funzionali e non, definizione delle principali storie utente con approccio SCRUM e dei relativi criteri di accettazione, diagramma dei casi d'uso (creato aggiungendo i casi d'uso dal diagramma preesistente in Documentazione A7-2024, relativa a task che impattano su T5 e sul non più esistente T6) e creazione di un primo *mockup* della pagina della classifica con Wireframe.cc.

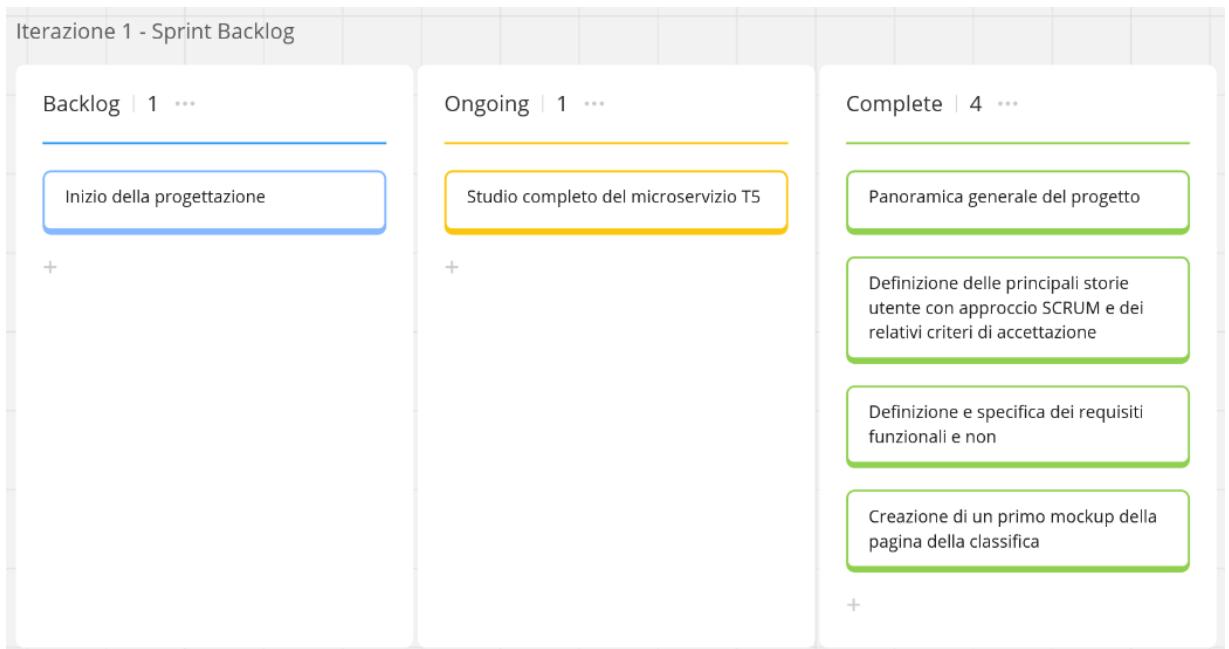


Figure 2.7: Backlog relativo all’iterazione 1 con le attività svolte, in corso e programmate per il futuro alla prima iterazione.

2.3.2 Iterazione 2

Iterazione 2: Approfondimento della documentazione relativa al microservizio T4 (del quale inizialmente ritenevamo di utilizzare i dati su giocatori e partite giocate per realizzare la classifica), approfondimento sulle API riguardanti i database presenti nel sistema, implementazione di una pagina di test della classifica utilizzando HTML, CSS e JavaScript, in modo tale da poter testare le funzionalità che si è scelto di implementare.

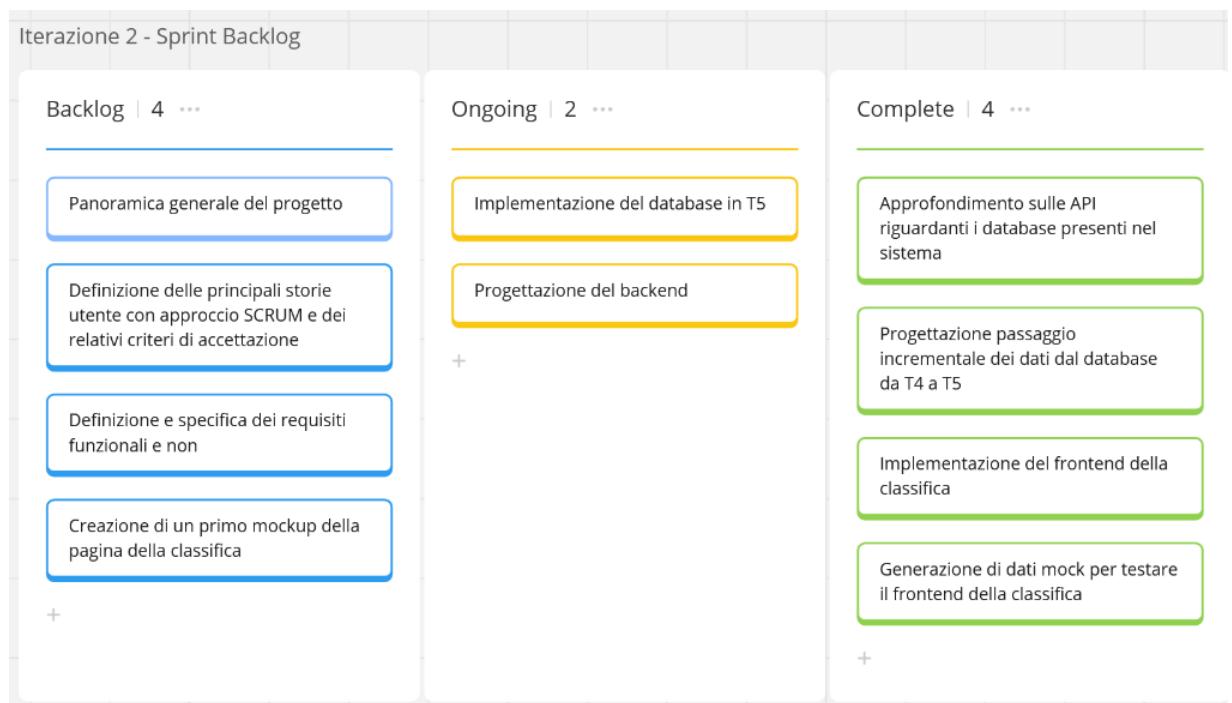


Figure 2.8: Backlog relativo all'iterazione 2 con le attività svolte, in corso e programmate per il futuro alla seconda iterazione.

2.3.3 Iterazione 3

Iterazione 3: Creazione ed implementazione del database in T5, creazione di nuovi servizi contenenti la business logic in T5, miglioramento della documentazione con creazione di numerosi diagrammi UML, come quelli di sequenza.

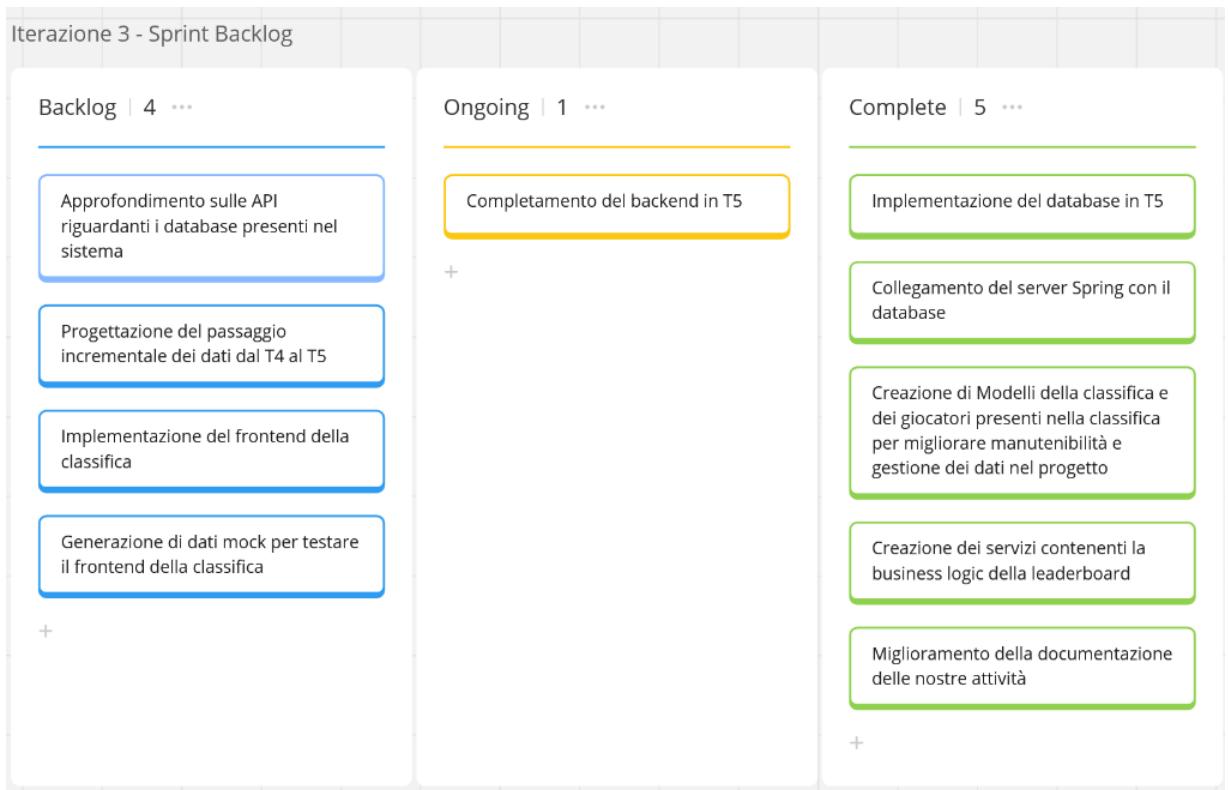


Figure 2.9: Backlog relativo all'iterazione 3 con le attività svolte, in corso ed eventuali sviluppi futuri alla terza iterazione.

Chapter 3

Analisi componente T56

3.0.1 Diagramma di contesto

Dopo aver analizzato la documentazione pre-esistente sull'ultima versione del componente T56 si è passati alla definizione di un nuovo *diagramma di contesto* (fig. 3.1).

Un *diagramma di contesto* è uno strumento utilizzato nella fase di analisi di un progetto software per fornire una visione generale e comprensiva dell'applicativo. Esso rappresenta graficamente gli attori coinvolti e tutti gli elementi chiave che interagiscono con il sistema, facilitando la comprensione del flusso di lavoro e delle interazioni tra le varie componenti.

Il nostro punto di partenza per l'analisi è stata proprio la creazione di tale diagramma, che mira a delineare una panoramica generale dell'esecuzione dell'applicativo. Questo include la rappresentazione degli attori coinvolti e di tutti gli elementi essenziali per il funziona-

mento del servizio implementato.

Nello specifico, abbiamo identificato due attori principali: *Utente* e *Studente*, con quest'ultimo che rappresenta una specializzazione del primo. Entrambi gli attori interagiscono con l'applicativo tramite un browser, attraverso il quale accedono all'interfaccia utente fornita dal sistema software per sfruttare le varie funzionalità.

L'attore *Utente* ha accesso solamente alla pagina di login, in quanto solo gli utenti registrati e autenticati possono partecipare al gioco. Una volta autenticato, l'*Utente* diventa uno *Studente* e può utilizzare l'interfaccia utente per selezionare tra diverse opzioni: *Visualizza Classifica*, *Nuova Partita* e *Visualizza Profilo*. Dopo aver effettuato la scelta, lo studente verrà indirizzato alla visualizzazione della classifica (o del profilo) o all'interfaccia che permette di scegliere la modalità di gioco desiderata.

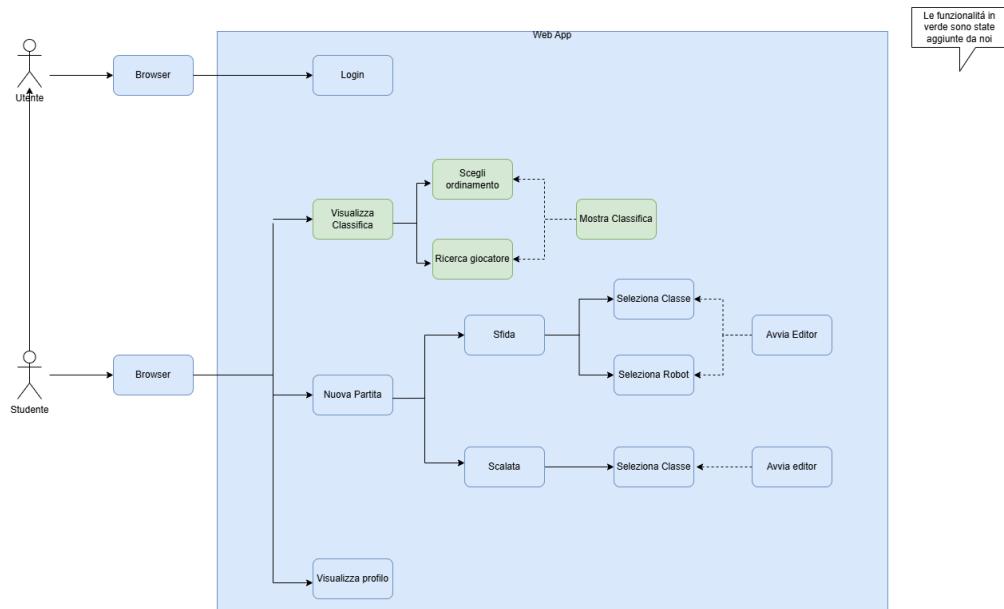


Figure 3.1: Diagramma di contesto del sottosistema T56.

3.1 Stato iniziale componente T56

Il sottosistema T56 è stato significativamente modificato nell'ultimo aggiornamento del progetto, come documentato nel *refactoring A13_Refactoring_T56*. Originariamente basato su un server Spring MVC che seguiva il pattern Model-View-Controller, al sistema è stato aggiunto un nuovo layer di astrazione per facilitare la creazione e gestione delle richieste ai controllers.

Il nuovo layer di astrazione è suddiviso in tre principali pacchetti:

- **Component:** (fig.3.3) Questo pacchetto semplifica la creazione di pagine dinamiche utilizzando il pagebuilder, i models e gli ObjectComponents.
- **Interface:** (fig.3.2) Facilita le interazioni con gli altri sottosistemi attraverso il ServiceManager e vari servizi forniti dal pacchetto.
- **Game:** (fig.3.4) Gestisce le richieste per l'editor di gioco e le interazioni necessarie per il calcolo delle statistiche e il salvataggio dei dati, grazie ai componenti GameLogic e GameController.

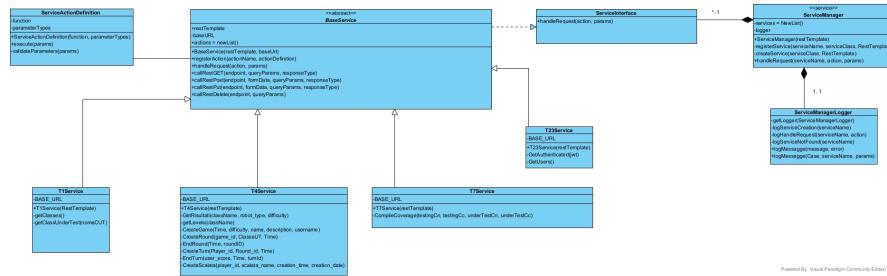


Figure 3.2: Diagramma delle classi del package Interface.

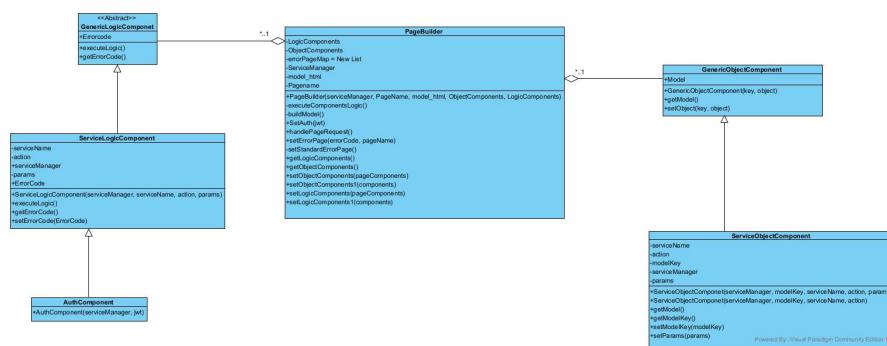


Figure 3.3: Diagramma delle classi del package Component.



Figure 3.4: Diagramma delle classi del package Game.

Nonostante i diagrammi delle classi preesistenti non rappresentino completamente tutti i componenti del sistema, è stato possibile ottenere una comprensione approfondita della struttura del server Spring all'interno del sottosistema tramite questi diagrammi e le informazioni disponibili sulla relativa pagina GitHub.

Chapter 4

Modifiche strutturali al Task 56

4.0.1 Diagramma dei componenti

La necessità di integrare un nuovo database nel sistema ha comportato significative modifiche al diagramma dei componenti, come illustrato in Figura 4.1. In particolare, è stato introdotto un nuovo componente, il *Leaderboard Database*, e un connettore tra il *Student Front-End*, che richiede il servizio, e il *Leaderboard Database*, che offre il servizio di gestione dei dati. Queste modifiche, come evidenziato nel diagramma, influenzano direttamente il Task 56, riflettendo l'espansione delle capacità del sistema e il miglioramento nella gestione delle classifiche.

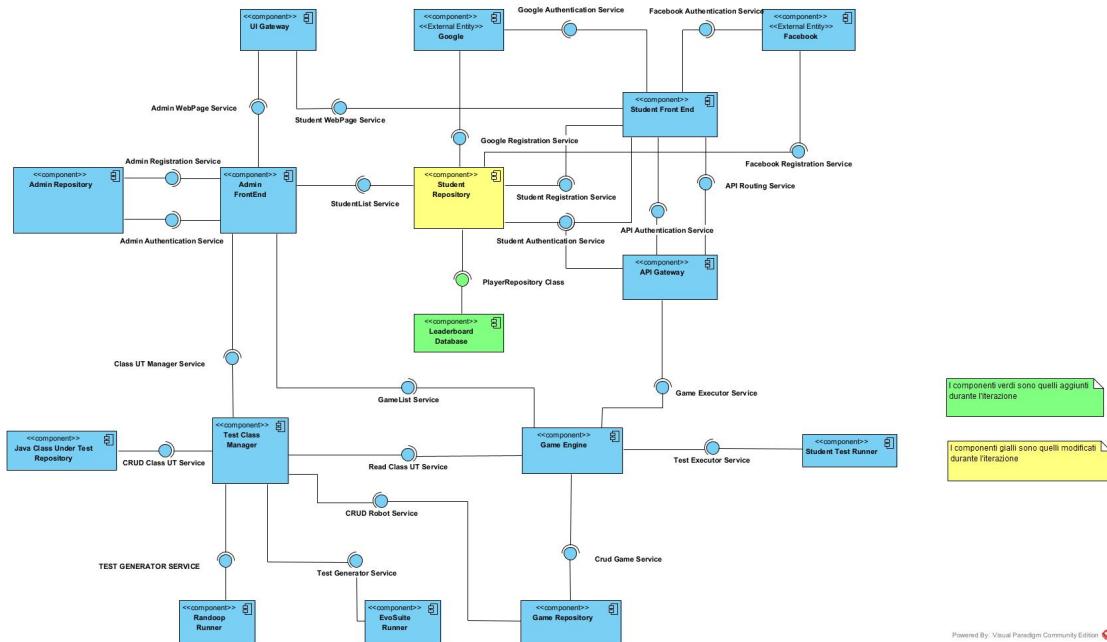


Figure 4.1: Diagramma dei componenti con modifiche.

4.0.2 Interfaccia PlayerRepository

PlayerRepository è un'interfaccia in Spring Data MongoDB che facilita l'interazione con la collezione di dati del database associata ai giocatori, in questo caso specifico, nella collezione LeaderboardStats. È stato implementato per semplificare le operazioni **CRUD** (Create, Read, Update, Delete) sui dati dei giocatori, sfruttando i metodi forniti da MongoRepository, che è una superinterfaccia che fornisce funzionalità comuni per lavorare con MongoDB. La scelta di questa interfaccia è stata dettata dal fatto che implementasse in automatico tutta la parte di comunicazione e di query con il database, evitando la definizione di *queries* specifiche e la gestione di *database clients* all'interno del front-end.

4.0.3 Diagramma delle Classi

Il diagramma delle classi rappresenta la struttura statica del sistema. Esso evidenzia le principali classi sviluppate per la creazione della pagina web della *Leaderboard* e per l'invio dei dati relativi alle partite al nuovo database. Considerando la complessità che un singolo diagramma comprensivo di tutte le classi avrebbe introdotto, abbiamo optato per la realizzazione di diagrammi focalizzati esclusivamente sugli aggiornamenti implementati e sulle interazioni con il codice preesistente.

Aggiornamenti al Diagramma dei Modelli Prima di procedere alla codifica, sono stati definiti due modelli fondamentali per la gestione dei dati:

- **Player:** Memorizza informazioni individuali sui giocatori, quali nome, punteggio e dettagli del profilo, essenziali per la gestione dei dati dei partecipanti.
- **Classifica:** Gestisce l'ordinamento e la visualizzazione dei giocatori basandosi sui loro punteggi o altre metriche, facilitando così la visualizzazione delle classifiche e il confronto delle prestazioni.

Questi modelli sono impiegati dal `GUIController` e dal `GameController`, supportati dal `LeaderboardService`, per gestire l'intera logica di business.

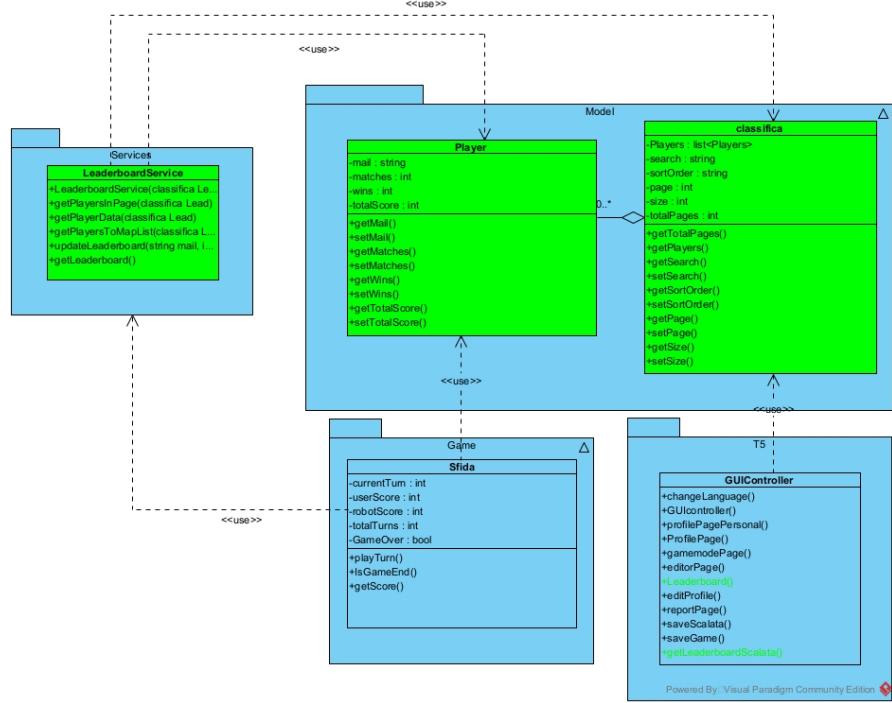


Figure 4.2: Diagramma delle classi per gli aggiornamenti ai model.

Aggiornamenti al Diagramma dei Servizi Successivamente alla definizione dei modelli, si è proceduto alla stesura dei servizi necessari per il corretto funzionamento del sistema e per mantenere il codice organizzato secondo il pattern architetturale **MVC** di Spring, con il supporto del *layer* di servizio e del *repository*. Durante la fase di progettazione, sono stati identificati tre servizi cruciali per l’implementazione delle nuove funzionalità:

- **LeaderboardService:** Gestisce le operazioni di supporto per la visualizzazione della pagina `leaderboard.html` e le operazioni di gestione dei dati con il database MongoDB.
- **PlayerComparator:** Definisce il criterio di ordinamento per la

comparazione tra i giocatori all'interno di una lista.

- **LeaderboardDatabaseService:** Amministra tutte le richieste al database, includendo operazioni CRUD e richieste per ottenere l'intera tabella.

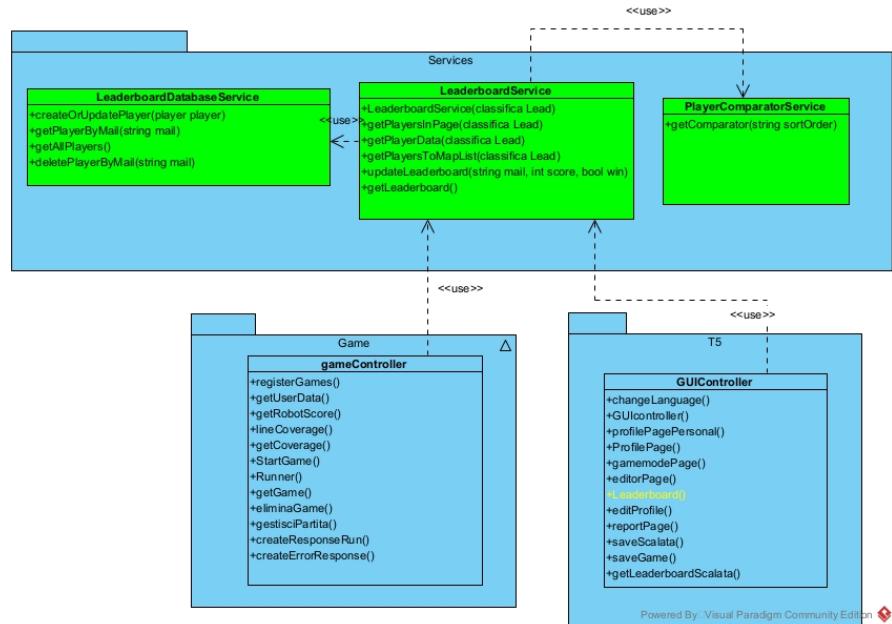


Figure 4.3: Diagramma delle classi per gli aggiornamenti ai servizi.

4.0.4 Descrizione del Nuovo Database

MongoDB, un database **NoSQL** orientato ai documenti, è stato scelto per la sua capacità di gestire grandi quantità di dati in formati non strutturati e semi-strutturati con flessibilità e alte prestazioni. A differenza dei database relazionali tradizionali, che utilizzano tabelle e righe, MongoDB opera con collezioni e documenti, offrendo una struttura dati più dinamica che si adatta facilmente alle esigenze dell'applicazione. Nel nostro progetto, il database è impiegato per gestire una singola

collezione, `LeaderboardStats`, che include i seguenti dati degli utenti:

Campo	Descrizione
<code>mail</code>	Usata come username
<code>matches</code>	Registra il numero totale di partite giocate
<code>wins</code>	Registra il numero di partite vinte
<code>totalScore</code>	Registra il punteggio totale del giocatore

Table 4.1: Descrizione dei campi della collezione `LeaderboardStats`

Leaderboard			
mail	char(256)		
matches	integer(10)	N	
wins	integer(10)	N	
totalscore	integer(10)	N	

Figure 4.4: Esempio di tabella della collezione `LeaderboardStatis`.

4.0.5 Issues Rilevate

Durante il progetto, sono emerse due problematiche principali legate all’infrastruttura del database T4. La prima *issue* riguardava la mancata creazione di un turno durante le partite, mentre la seconda era relativa al fallimento nel salvataggio dei dati delle partite. Questi problemi (insieme alla difficoltà nel modificare il sistema T4) hanno evidenziato le limitazioni dell’implementazione attuale e la necessità di iniziare una migrazione dei dati dal database T4 a quello implementato in questa documentazione.

4.0.6 Diagramma dei Package

Un diagramma dei package è un tipo di diagramma UML utilizzato per organizzare e rappresentare gli elementi di un sistema software in "package". Ogni *package* raggruppa elementi correlati come classi, interfacce e altri *package*, in base alla loro funzionalità, livelli di accesso o utilizzo all'interno del sistema. Questo diagramma facilita la visualizzazione dell'architettura del software, delineando una struttura modulare che aiuta a comprendere le dipendenze e le relazioni tra i componenti. Durante la fase progettuale, sono stati realizzati aggiornamenti architettonici significativi per migliorare la gestione dei dati e aumentarne la scalabilità, con un focus particolare sui moduli Model, Repository e Service.

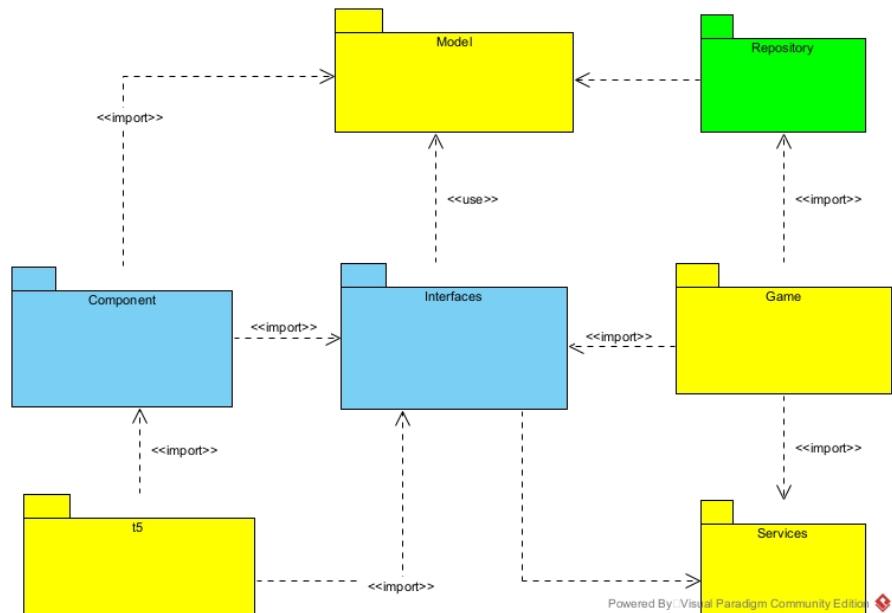


Figure 4.5: Diagramma dei packages di T56.

Modulo Model

Il modulo Model è stato arricchito con entità chiave quali la classe `Player`, progettata per modellare le entità principali dell'applicazione relative al dominio dei dati utente, e la classe `Classifica`, utilizzata per gestire i dati della richiesta di classifica con maggiore facilità. Le entità di questo modulo fungono da rappresentazione fondamentale degli oggetti di business e sono ottimizzate per interagire con il layer di persistenza dei dati.

Introduzione del Modulo Repository

È stato creato un nuovo modulo Repository per astrarre l'interfaccia tra la logica di business e le operazioni di persistenza dei dati. Questo modulo serve da collegamento tra i modelli di dati e il database, e utilizza l'interfaccia `PlayerRepository` per gestire le operazioni CRUD (Creazione, Lettura, Aggiornamento, Cancellazione). Questo approccio promuove il principio di separazione delle preoccupazioni, isolando la logica di business dalla logica di accesso ai dati.

Modulo Service

Nel modulo Service sono stati implementati tre nuovi servizi: `PlayerComparator` utilizzato per definire il criterio di comparazione tra giocatori, `LeaderboardService` per la gestione di tutte le operazioni relative agli oggetti di tipo classifica, e `LeaderboardDatabaseService`, per semplificare le inter-

azioni con il database.

Moduli t5 e Game

I moduli T5 e Game risultano solo leggermente cambiati, infatti è stata aggiunta solo la nuova logica per salvare i dati nel database in GameController e la logica per la generazione della pagina classifica in GuiController.

4.0.7 Sequence Diagram

Un diagramma di sequenza è un diagramma UML utilizzato per visualizzare l'ordine cronologico delle interazioni tra oggetti in un processo specifico. Mostra come gli oggetti scambiano messaggi e collaborano per svolgere una funzione. I diagrammi di sequenza sono importanti perché chiariscono la complessità delle interazioni software, aiutano a individuare problemi di progettazione e facilitano la comunicazione tra sviluppatori e *stakeholder*. Sono essenziali per analizzare, progettare e documentare i flussi operativi all'interno dei sistemi software.

Visualizza Classifica

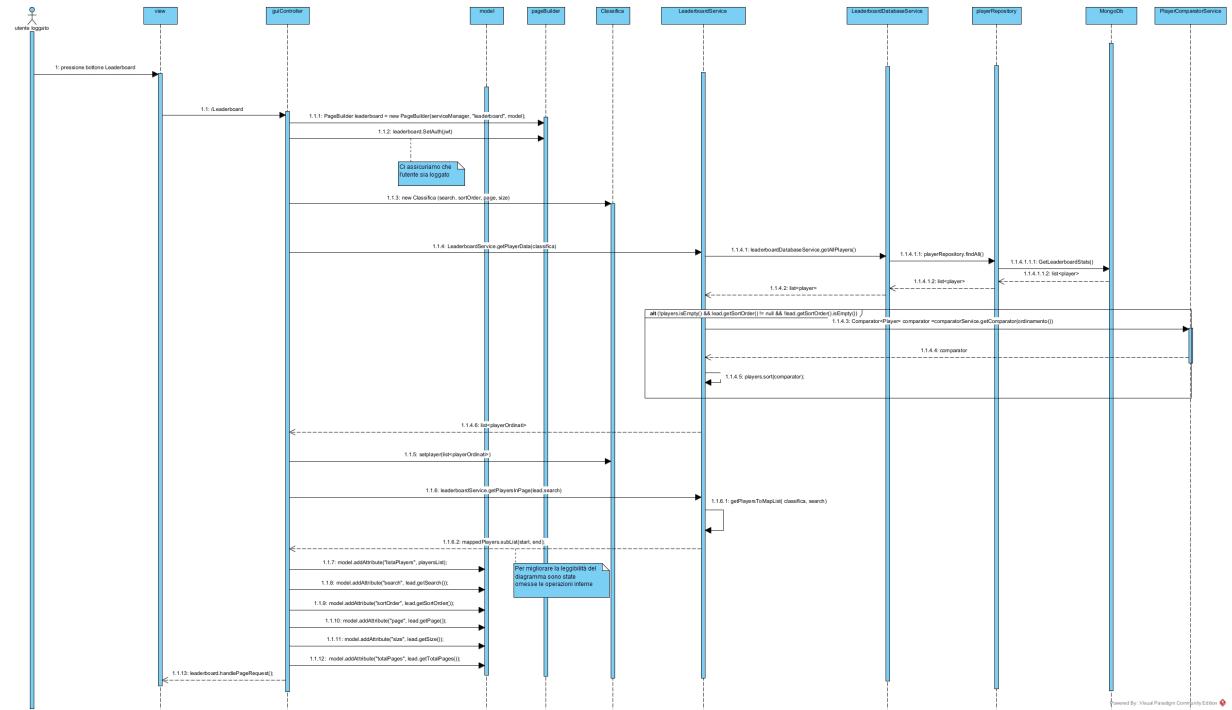


Figure 4.6: Diagramma dei packages di T56.

Il diagramma di sequenza dettaglia le operazioni eseguite dall'applicazione quando uno studente richiede di visualizzare la classifica:

1. L'interazione inizia quando lo studente preme il pulsante della classifica e viene reindirizzato all'indirizzo /leaderboard.
2. Il GuiController crea un'istanza del pagebuilder e un'istanza della classifica con tutti i dati ricevuti nella richiesta (numero di pagina, numero di righe per pagina, tipo di ordinamento selezionato ed eventuale filtro di ricerca).
3. Il GuiController verifica che l'utente sia autenticato e richiede al LeaderboardService la lista degli utenti.

4. Il LeaderboardService inoltra la richiesta al LeaderboardDatabase e, una volta ricevuta la lista degli utenti, la riordina secondo l'ordinamento scelto dallo studente.
5. Il GuiController riceve la lista degli utenti ordinati e li memorizza nella classifica.
6. Il GuiController incorpora i dati nel modello della pagina.
7. Infine, viene restituita la risposta all'utente tramite l'*handle* della richiesta di pagina.

Chapter 5

Implementazione della soluzione

5.1 Implementazione Database

La scelta di MongoDB, un database non relazionale (**NoSQL** orientato ai documenti), è stata guidata dalla sua scalabilità, dalla flessibilità del modello dei dati e dalla semplicità di utilizzo. La nostra necessità si limitava a una singola tabella per gestire la *leaderboard*, eliminando così la necessità dei tipici meccanismi con chiavi esterne dei database SQL.

5.1.1 Aggiunta del Container

Per implementare il database, il primo passo è stato modificare il *docker-compose* del Task 56. È stato definito un nuovo container, *db-leaderboard*, mappando la porta 27017 sulla 27018 (poiché la 27017 è già utilizzata dal database MongoDB del Task 1). Il database è stato collegato alla rete *global-network*, per facilitare l'accessibilità futura

da altri task per *testing*, e viene popolato tramite lo script *initLeaderboard.js* (Figura 5.1).

```
1 db = db.getSiblingDB('LeaderboardStats'); // Connessione al database 'LeaderboardStats'  
2  
3 // Crea la collezione 'LeaderboardStats' se non esiste  
4 db.createCollection('LeaderboardStats');
```

Figure 5.1: File InitLeaderboard.js.

5.1.2 Collegamento del Database con il Server Spring

Per integrare il database con il server Spring, sono state necessarie quattro operazioni principali: l'aggiunta delle dipendenze del database nel file di configurazione del server, la configurazione delle dipendenze aggiunte, la creazione di un modello per i dati da inserire nel database e la creazione di una classe di estensione a *MongoTemplate*, creare un servizio che implementi tutte le operazioni CRUD effettuabili sulla *repository*.

Configurazione del Server È stato necessario includere *spring-boot-starter-data-mongodb* (Figura 5.2) tra le dipendenze nel file *pom.xml*, facilitando l'integrazione tra le applicazioni Spring Boot e MongoDB. Questa è una delle *starter dependencies* che Spring offre per semplificare la configurazione e l'implementazione dei progetti.

```
91      |     <!-- Dependency MongoDB -->
92      |     <dependency>
93      |       <groupId>org.springframework.boot</groupId>
94      |       <artifactId>spring-boot-starter-data-mongodb</artifactId>
95      |     </dependency>
96   </dependencies>
97
98   <build>
99     <plugins>
100       <plugin>
101         <groupId>org.springframework.boot</groupId>
102         <artifactId>spring-boot-maven-plugin</artifactId>
103         <configuration>
104           <mainClass>com.g2.t5.T5Application</mainClass>
105         </configuration>
106       </plugin>
107     </plugins>
108   </build>
109 </project>
```

Figure 5.2: Modifiche al pom.xml.

Configurazione delle Dipendenze Aggiunte Per connettere il server Spring al server MongoDB, sono stati impostati i seguenti parametri: hostname del database, porta d’ascolto e il nome del database al quale l’applicazione si connette per eseguire operazioni di lettura e scrittura dei dati.

```
21  # MongoDB
22  spring.data.mongodb.host=db-leaderboard
23  spring.data.mongodb.port=27017
24  spring.data.mongodb.database=LeaderboardStats
```

Figure 5.3: Modifiche ad application.properties.

Classe Player.java La classe Player rappresenta un giocatore nel database MongoDB e viene utilizzata per salvare e recuperare le statis-

tiche relative ai giocatori nella collezione `LeaderboardStats`. Ecco i campi che vengono creati nel database per ogni istanza di `Player`:

- `mail`: un campo identificativo che funge da ID del documento.
È una stringa che rappresenta l'email del giocatore.
- `matches`: un intero che indica il numero totale di partite giocate dal giocatore.
- `wins`: un intero che rappresenta il numero totale di vittorie del giocatore.
- `totalScore`: un intero che tiene traccia del punteggio complessivo accumulato dal giocatore.

Questi campi permettono di tenere traccia delle prestazioni del giocatore all'interno di un'applicazione o di un gioco, facilitando l'analisi delle statistiche e il confronto con altri giocatori nella classifica.

```
6  @Document(collection = "LeaderboardStats")
7  public class Player {
8      @Id
9      private String mail;
10     private int matches;
11     private int wins;
12     private int totalScore;
13
14     public Player(String mail, int matches, int wins, int totalScore) {
15         this.mail = mail;
16         this.matches = matches;
17         this.wins = wins;
18         this.totalScore = totalScore;
19     }
20
21     // Getter e setter
22     public String getMail() {
23         return this.mail;
24     }
25
26     public void setMail(String mail) {
27         this.mail = mail;
28     }
29
30     public int getMatches() {
31         return this.matches;
32     }
33
34     public void setMatches(int matches) {
35         this.matches = matches;
36     }
37
38     public int getWins() {
39         return this.wins;
40     }
41
42     public void setWins(int wins) {
43         this.wins = wins;
44     }
45
46     public int getTotalScore() {
47         return this.totalScore;
48     }
49
50     public void setTotalScore(int totalScore) {
51         this.totalScore = totalScore;
52     }
53 }
```

Figure 5.4: Classe Player.java.

Interfaccia PlayerRepository PlayerRepository è un’interfaccia in Spring Data MongoDB che facilita l’interazione con la collezione di dati del database associata ai giocatori, in questo caso specifico, nella collezione LeaderboardStats. È stato implementato per semplificare le operazioni CRUD (Create, Read, Update, Delete) sui dati dei giocatori, sfruttando i metodi forniti da MongoRepository, che è una superinterfaccia che fornisce funzionalità comuni per lavorare con MongoDB. Oltre ai metodi preesistenti di un’interfaccia

MongoTemplate, è stato aggiunto anche il metodo:

- `findByMail(String mail)`: Ricerca un giocatore basandosi sull'email, che funge da chiave primaria (ID del documento). Questo metodo restituisce un oggetto Player corrispondente all'email specificata, permettendo operazioni specifiche come la visualizzazione o l'aggiornamento delle statistiche del giocatore.

```
7  public interface PlayerRepository extends MongoRepository<Player, String> {  
8      Player findByMail(String mail);  
9  }  
10
```

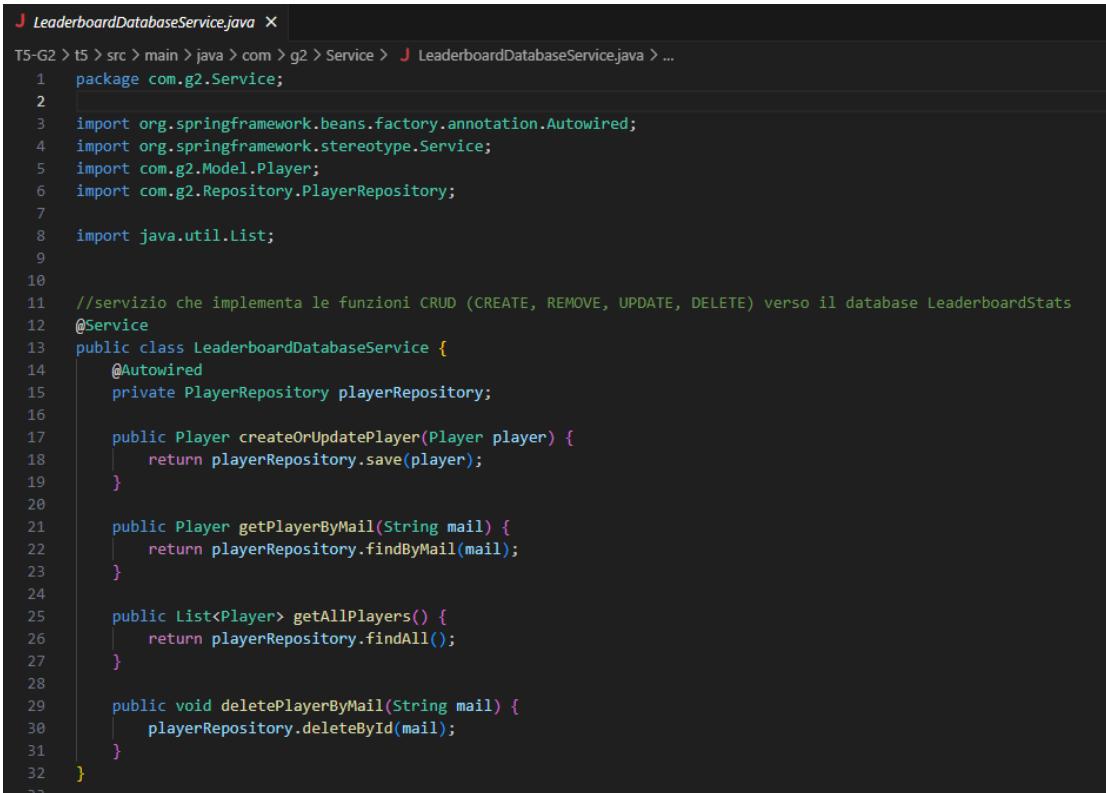
Figure 5.5: Interfaccia PlayerRepository.java.

Servizio LeaderboardDatabaseService Il servizio LeaderboardDatabaseService è stato implementato per gestire tutte le operazioni CRUD relative al PlayerRepository. Questo servizio è fondamentale per la manutenibilità del progetto, in quanto centralizza la logica di accesso ai dati. In caso di modifiche future relative all'architettura del database, sarà necessario aggiornare solamente i metodi di questo servizio, senza intervenire sul resto dell'applicazione.

I principali metodi offerti includono:

- `createOrUpdatePlayer(Player player)`: Permette la creazione o l'aggiornamento di un giocatore nel database.
- `getPlayerByMail(String mail)`: Recupera le informazioni di un giocatore tramite la sua email.

- `getAllPlayers()`: Ottiene i dati di tutti i giocatori presenti nel database.
- `deletePlayerByMail(String mail)`: Elimina le informazioni di un giocatore dal database tramite la sua email.



```

J LeaderboardDatabaseService.java ×
T5-G2 > t5 > src > main > java > com > g2 > Service > J LeaderboardDatabaseService.java > ...
1 package com.g2.Service;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5 import com.g2.Model.Player;
6 import com.g2.Repository.PlayerRepository;
7
8 import java.util.List;
9
10
11 //servizio che implementa le funzioni CRUD (CREATE, REMOVE, UPDATE, DELETE) verso il database LeaderboardStats
12 @Service
13 public class LeaderboardDatabaseService {
14     @Autowired
15     private PlayerRepository playerRepository;
16
17     public Player createOrUpdatePlayer(Player player) {
18         return playerRepository.save(player);
19     }
20
21     public Player getPlayerByMail(String mail) {
22         return playerRepository.findByMail(mail);
23     }
24
25     public List<Player> getAllPlayers() {
26         return playerRepository.findAll();
27     }
28
29     public void deletePlayerByMail(String mail) {
30         playerRepository.deleteById(mail);
31     }
32 }

```

Figure 5.6: Servizio LeaderboardDatabaseService.

5.1.3 Modifiche al Front-End

Per l’implementazione della classifica dal punto di vista del front-end, sono state necessarie diverse operazioni chiave per integrare efficacemente le funzionalità nel sito esistente. Queste includono la modifica di un file HTML esistente, l’uso di Thymeleaf per facilitare il passaggio

dei dati tra il back-end e il front-end, e l'aggiornamento della barra di navigazione per migliorare l'accessibilità alla classifica.

Modifica della Navbar

Il primo passo è stato quello di modificare la Navbar per permettere agli utenti di accedere facilmente alla pagina della classifica. Questa modifica è stata realizzata intervenendo sul file situato in /com/g2/resources/fragments. È stato aggiunto un bottone che reindirizza direttamente a /leaderboard, semplificando così l'interfaccia utente e migliorando l'esperienza di navigazione.

```
1 <nav th:fragment="navbar" class="navbar bg-body-tertiary">
2   <div class="container-fluid">
3     <a class="navbar-brand" href="/main"> Home </a>
4     <a class="navbar-brand" href="/leaderboard">Classifica</a> [Red box]
5     <div class="collapse navbar-collapse" id="navbarNav">
6       <ul class="navbar-nav">
7         <li class="nav-item">
8           <a class="nav-link active" aria-current="page" href="/main">Home</a>
9         </li>
10        <li class="nav-item">
11          <a class="nav-link" href="/leaderboard">LeaderBoard</a>
12        </li>
13      </ul>
14    </div>
```

Figure 5.7: Modifica al codice di Navbar.html.



Figure 5.8: Effetto Visivo Navbar.

mMdifiche a leaderboard.html

Il file `leaderboard.html` sfrutta Thymeleaf per integrare i dati del backend in una visualizzazione strutturata della classifica dei giocatori.

Le caratteristiche principali comprendono:

- **Intestazione e Navigazione:** Utilizzo di frammenti Thymeleaf per incorporare elementi ricorrenti quali l'header e la navbar.
- **Visualizzazione Classifica:** Una tabella che mostra la classifica dei giocatori, con dettagli come nome, numero totale di partite giocate, vittorie e punteggio complessivo.
- **Funzionalità di Ricerca e Ordinamento:** Moduli per la ricerca di giocatori e per l'ordinamento della classifica secondo diverse metriche.
- **Paginazione:** Controlli di navigazione che permettono di muoversi tra le varie pagine di dati.
- **Bootstrap e JavaScript:** Uso di Bootstrap per un design responsivo e di script JavaScript per funzionalità aggiuntive relative alla gestione della tabella.

Questa struttura è pensata per facilitare l'accesso e l'interazione dell'utente, garantendo contemporaneamente la sicurezza nella gestione dei form.

Posizione	Nome	Partite totali	Partite vinte	Punteggio totale
1	bobbone@gmail.com	2	2	68
2	izio.5555@gmail.com	1	0	0

Figure 5.9: pagina classifica.

5.1.4 Integrazione Front-End Classifica con Back-End

Per garantire la funzionalità della pagina `leaderboard.html`, sono state necessarie diverse modifiche al back-end, tra cui l'aggiunta del modello della classifica nel sistema, l'introduzione di servizi per la gestione dei giocatori e della classifica stessa, e la modifica della gestione delle chiamate a `/leaderboard` nel `GuiController`.

Modifica della Chiamata al GuiController La richiesta GET `/leaderboard` gestita dal `GuiController` è cruciale per la visualizzazione della classifica dei giocatori. Questa richiesta supporta funzionalità come la ricerca, l'ordinamento dei dati dei giocatori e la paginazione dei risultati. I parametri passati dal back-end includono:

- **search**: Un parametro opzionale utilizzato per filtrare i giocatori basandosi su criteri di ricerca specifici.
- **sortOrder**: Determina l'ordinamento dei dati visualizzati, con un valore predefinito di "`totalScore`".
- **page**: Indica la pagina corrente per la paginazione dei risultati, con un valore predefinito di 0.
- **size**: Specifica il numero di risultati visualizzati per pagina, con un valore predefinito di 30.
- **jwt**: Un cookie utilizzato per verificare l'autenticazione dell'utente.

Al ricevimento della richiesta GET /leaderboard, il controller registra l'accesso tramite un messaggio di log e inizializza il PageBuilder per configurare il modello di pagina. Viene poi creata un'istanza dell'oggetto Classifica con i parametri specificati per gestire la raccolta e l'organizzazione dei dati dei giocatori. Questo oggetto invoca i servizi necessari per filtrare e ordinare i giocatori secondo i criteri forniti. La funzione popola il modello con i dati dei giocatori, la query di ricerca, i dettagli dell'ordinamento e della paginazione, e il numero totale di pagine disponibili. Prima di processare i dati, il sistema verifica l'autenticazione dell'utente tramite il cookie JWT per assicurare che solo gli utenti autenticati possano accedere alle informazioni. Infine, restituisce la risposta adeguata all'utente attraverso il PageBuilder, completando così la gestione della richiesta.

```
//CAMBIAMENTI A /LEADERBOARD EFFETTUATI IL 13/12/2025
@GetMapping("/leaderboard")
public String leaderboard(@RequestParam(value = "search", required = false) String search,
    @RequestParam(value = "sortOrder", defaultValue = "totalScore") String sortOrder,
    @RequestParam(value = "page", defaultValue = "0") int page,
    @RequestParam(value = "size", defaultValue = "30") int size,
    Model model,
    @CookieValue(name = "jwt", required = false) String jwt) {
    // instanzio il modello base della pagina

    logger.info("[GUICONROLLER]: ricevuta richiesta pagina classifica");
    PageBuilder leaderboard = new PageBuilder(serviceManager, PageName:"leaderboard", model);
    leaderboard.setAuth(jwt);
    // instanzio l'oggetto leaderboard che farà da wrapper dei dati della richiesta e che contiene i dati dei giocatori nella classifica
    Classifica lead = new Classifica (search, sortOrder, page, size);
    // mi assicuro che l'utente che richiede la pagina sia autenticato

    // Carica i dati riguardanti tutti i giocatori
    lead.setPlayers(leaderboardService.getPlayerData(lead));
    // listaplayers utilizza il casting di List<Map<String, Object>> perché la
    List<Map<String, Object>> playersList = leaderboardService.getPlayersInPage(lead,search);

    // Passo i parametri della leaderboard al front end
    logger.info("[GUICONROLLER]: Caricamento dei dati della classifica nella pagina");
    model.addAttribute("listaPlayers", playersList);
    model.addAttribute("search", lead.getSearch());
    model.addAttribute("sortOrder", lead.getSortOrder());
    model.addAttribute("page", lead.getPage());
    model.addAttribute("size", lead.getSize());
    model.addAttribute("totalPages", lead.getTotalPages());
    logger.info("[GUICONROLLER]: restituzione pagina classifica");

    return leaderboard.handlePageRequest();
}
```

Figure 5.10: Codice della chiamata REST /leaderboard.

Model Classifica La classe `Classifica` gestisce la logica per il recupero e l’organizzazione dei dati di classifica dei giocatori. La classe tiene traccia di vari parametri di ricerca e ordinamento, nonché di paginazione attraverso i campi `search`, `sortOrder`, `page`, `size` e `totalPages`.

Il costruttore della classe accetta parametri per la ricerca (`search`), l’ordinamento (`sortOrder`), il numero di pagina (`page`) e la dimensione della pagina (`size`), impostando inizialmente la lista dei giocatori come vuota e il numero totale di pagine a zero. Questo permette un’istanziazione iniziale della classifica senza dati concreti, pronta per essere popolata successivamente.

I metodi getter e setter permettono la manipolazione e il recu-

pero dei dati dei giocatori. Il metodo `setPlayers(List<Player> players)` oltre a impostare la lista dei giocatori, calcola automaticamente il numero totale di pagine basato sulla dimensione della pagina e sulla quantità di giocatori, utilizzando il numero totale di giocatori e la dimensione della pagina per determinare il numero di pagine necessarie per visualizzare tutti i giocatori.

Inoltre, la classe `Classifica` include metodi per aggiornare i parametri di ricerca e ordinamento, permettendo di modificare dinamicamente i criteri secondo cui i giocatori sono visualizzati e ordinati.

```

17 public class Classifica {
18
19
20     private List<Player> players;
21     private String search;
22     private String sortOrder;
23     private int page;
24     private int size;
25     private int totalPages; //variabile strettamente legata a players, infatti la sua modifica è legata al setplayer.
26
27     public Classifica(String search, String sortOrder, int page, int size) {
28         this.search = search;
29         this.sortOrder = sortOrder;
30         this.page = page;
31         this.size = size;
32         this.totalPages=0;
33         this.players=Collections.emptyList();
34     }
35
36     public int getTotalPages() {
37         return this.totalPages;
38     }
39
40
41     // Getter e Setter
42

```

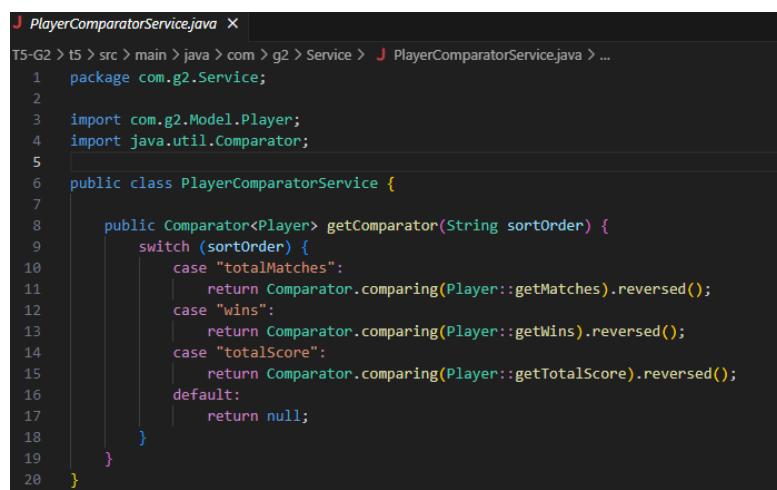
Figure 5.11: Codice della definizione di `Classifica`.

servizio PlayerComparatorService La classe `PlayerComparatorService` fornisce un servizio per ottenere un comparatore per gli oggetti `Player` basato su diversi criteri di ordinamento. Questo servizio è essenziale per supportare operazioni di ordinamento personalizzate nelle liste di giocatori, come necessario nell'applicazione.

Metodo getComparator: Il metodo `getComparator` accetta una stringa `sortOrder` che determina il criterio di ordinamento per i giocatori. I criteri supportati includono:

- **totalMatches:** Ordina i giocatori in base al numero totale di partite giocate, in ordine decrescente.
- **wins:** Ordina i giocatori in base al numero totale di vittorie, in ordine decrescente.
- **totalScore:** Ordina i giocatori in base al punteggio totale accumulato, in ordine decrescente.

Il comparatore viene generato utilizzando il metodo `comparing` della classe `Comparator` di Java, applicato ai metodi getter appropriati della classe `Player`. Se il valore di `sortOrder` non corrisponde a uno dei criteri specificati, il metodo restituirà `null`, indicando l'assenza di un criterio di ordinamento valido.



```

1 package com.g2.Service;
2
3 import com.g2.Model.Player;
4 import java.util.Comparator;
5
6 public class PlayerComparatorService {
7
8     public Comparator<Player> getComparator(String sortOrder) {
9         switch (sortOrder) {
10             case "totalMatches":
11                 return Comparator.comparing(Player::getMatches).reversed();
12             case "wins":
13                 return Comparator.comparing(Player::getWins).reversed();
14             case "totalScore":
15                 return Comparator.comparing(Player::getTotalScore).reversed();
16             default:
17                 return null;
18         }
19     }
20 }
```

Figure 5.12: Codice di PlayerComparatorService.

Classe LeaderboardService La classe LeaderboardService si occupa di gestire la logica di business per la visualizzazione e manipolazione dei dati della classifica nel sistema. Utilizza servizi ausiliari per interfacciarsi con il database e per fornire operazioni specifiche come l'ordinamento e la paginazione dei giocatori.

Metodi Principali:

- **getPlayerData(Classifica lead):** Questo metodo recupera i dati dei giocatori dal database tramite il servizio LeaderboardDatabaseService, applicando poi l'ordinamento specificato nell'oggetto Classifica. Utilizza PlayerComparatorService per ottenere il comparatore basato sul criterio di ordinamento desiderato.
- **getPlayersToMapList(Classifica lead, String SearchFilter):** Converte l'elenco dei giocatori in una lista di mappe, utili per il passaggio di dati complessi a Thymeleaf. Includendo funzioni di filtro e posizionamento dei giocatori nella classifica.
- **getPlayersInPage(Classifica lead, String SearchFilter):** Filtra e ritaglia l'elenco dei giocatori per la visualizzazione in una pagina specifica, gestendo così la paginazione dei dati della classifica.
- **updateLeaderboard(String mail, int additionalScore, boolean isWinner):** Aggiorna i dati di un giocatore nel database in base

al risultato di una partita, incrementando il numero di partite giocate, le vittorie e aggiornando il punteggio totale. Gestisce anche la creazione di nuovi giocatori se non presenti.

- **getLeaderboard()**: Recupera l'elenco completo dei giocatori dal database senza applicare filtri o ordinamenti.

Dipendenze e Componenti: La classe è annotata con `@Service` per indicare che si tratta di un componente del servizio nella logica di business di Spring. Dipende da `LeaderboardDatabaseService` per tutte le operazioni di database, evidenziando un'architettura pulita e una chiara separazione delle responsabilità. Utilizza inoltre il componente `PlayerComparatorService` per delegare l'ordinamento dei dati, mostrando un uso efficace del principio di singola responsabilità e dell'inversione di controllo.

```

25  @Service
26  public class LeaderboardService {
27
28      @Autowired
29      private LeaderboardDatabaseService leaderboardDatabaseService;
30
31      public List<Player> getPlayerData(Classifica lead) {
32          // Supponendo che "Player" abbia un costruttore che accetti questi parametri
33          PlayerComparatorService comparatorService = new PlayerComparatorService();
34          List<Player> players = getLeaderboard();
35
36          if (!players.isEmpty() && lead.getSortOrder() != null && !lead.getSortOrder().isEmpty()) {
37              Comparator<Player> comparator = comparatorService.getComparator(lead.getSortOrder());
38              if (comparator != null) {
39                  players.sort(comparator);
40              }
41          }
42
43          return players;
44      }
45
46
47      // Metodo per convertire la lista di Player in una lista di Mappe <string, object> necessario al passaggio dei dati con thymeleaf
48      public List<Map<String, Object>> getPlayersToMapList(Classifica lead, String SearchFilter) {
49          List<Map<String, Object>> playersMapList = new ArrayList<>();
50          int position = 1; // Inizia da 1 per la posizione del primo giocatore
51          for (Player player : lead.getPlayers()) {
52              Map<String, Object> playerMap = new HashMap<>();
53              playerMap.put("Mail", player.getMail());
54              playerMap.put("totalMatches", player.getMatches());
55              playerMap.put("wins", player.getWins());
56              playerMap.put("totalscore", player.getTotalScore());
57              playerMap.put("position", position); // Aggiunge la posizione attuale del giocatore alla mappa
58              playersMapList.add(playerMap);
59              position++; // Incrementa la posizione per il prossimo giocatore
60          }
61          // Filtra i giocatori prima di aggiungerli alla lista di mappe
62          if (Searchfilter != null) {
63              playersMapList = playersMapList.stream()
64                      .filter(playerMap -> playerMap.get("Mail").equals(Searchfilter))
65                      .collect(Collectors.toList());
66          }
67          return playersMapList;
68      }

```

Figure 5.13: Metodi getPlayerData e getPlayersToMapList.

```

70      //metodo utile per ottenere la lista dei giocatori che devono essere mostrati nella pagina
71      public List<Map<String, Object>> getPlayersInPage(Classifica lead, String Searchfilter) {
72          //Trasformo la lista di tutti i giocatori in una lista di mappe, operazione necessaria per poter lavorare con thymeleaf
73          List<Map<String, Object>> mappedPlayers = getPlayersToMapList(lead, Searchfilter);
74          int start = lead.getPage() * lead.getSize();
75          int end = Math.min(start + lead.getSize(), mappedPlayers.size());
76          return mappedPlayers.subList(start, end); //Ritorna solo la porzione di lista richiesta
77      }
78
79
80
81
82      public String updateLeaderboard(String mail, int additionalScore, boolean isWinner) {
83          try {
84              Player entry = leaderboardDatabaseService.getPlayerByMail(mail);
85
86              if (entry == null) {
87                  entry = new Player(mail, matches:0, wins:0, totalScore:0);
88              }
89
90              entry.setMatches(entry.getMatches() + 1);
91              if (isWinner) {
92                  entry.setWins(entry.getWins() + 1);
93              }
94              entry.setTotalScore(entry.getTotalScore() + additionalScore);
95
96              Player updatedEntry = leaderboardDatabaseService.createOrUpdatePlayer(entry);
97              if (updatedEntry != null && updatedEntry.getTotalScore() == entry.getTotalScore()) {
98                  System.out.println("Leaderboard aggiornata correttamente per " + mail);
99                  return "Leaderboard aggiornata correttamente per " + mail;
100             } else {
101                 throw new RuntimeException("Errore di validazione dopo l'aggiornamento della leaderboard per " + mail);
102             }
103         } catch (Exception e) {
104             System.out.println("Errore durante l'aggiornamento della leaderboard: " + e.getMessage());
105             throw new RuntimeException("Errore durante l'aggiornamento della leaderboard per " + mail, e);
106         }
107     }
108
109     public List<Player> getLeaderboard(){
110         return leaderboardDatabaseService.getAllPlayers();
111     }

```

Figure 5.14: Metodi getPlayersInPage e updateLeaderboard.

5.1.5 Modifiche al Back-End

In seguito all'introduzione di un nuovo database e adottando una strategia di migrazione incrementale dei dati, abbiamo deciso di non apportare modifiche radicali al `GameController`. Abbiamo identificato il punto di elaborazione dei dati di gioco e integrato la funzionalità di salvataggio nel nostro database attraverso il `LeaderboardService`. Le modifiche apportate includono l'aggiunta di un parametro al form inviato dall'interfaccia utente del front-end e l'implementazione del meccanismo di salvataggio nel back-end.

Modifica al Front-End Per permettere il salvataggio dei dati salvati nel database, è stato essenziale modificare il comportamento del front-end per trasmettere l'identificativo dell'utente. Pertanto, è stata adeguata la funzione `getFormData` nel file JavaScript `/resources/static/t5/js/Util_Editor.js`, in cui viene incluso il passaggio di un nuovo parametro `mail`.

```

321 // Funzione per ottenere i dati del form da inviare
322 function getFormData() {
323     const formData = new FormData();
324     const className = localStorage.getItem("underTestClass");
325
326     //formData.append("testingClassName", "Test"+className+".java");
327     formData.append("testingClassName", editor_utente.getValue());
328     formData.append("underTestClass", `${className}.java`);
329     formData.append("underTestClassCode", editor_robot.getValue());
330     formData.append("className", className);
331     formData.append("playerId", String(parseJwt(getCookie("jwt"))).userId);
332     formData.append("turnId", localStorage.getItem("turnId"));
333     formData.append("difficulty", localStorage.getItem("difficulty"));
334     formData.append("type", localStorage.getItem("robot"));
335     formData.append("order", orderTurno);
336     formData.append("username", localStorage.getItem("username"));
337     formData.append("testClassId", className);
338     formData.append("eliminaGame", false);
339
340     //AGGIUNTO IL 13/12/2020 MOMENTO IN CUI L'USERNAME EQUIVALE ALLA MAIL E NON È SALVATO NEL LOCALSTORAGE
341     //QUESTO APPEND È STATO EFFETTUATO IN MODO CHE NEL MOMENTO IN CUI VIENE EFFETTUATA LA CHIAMATA REST A: /RUN SIA POSSIBILE SALVARE LA MAIL DEL RICHIEDENTE NEL DATABASE DI TS
342     formData.append("mail", username);
343
344     return formData;
345 }

```

Figure 5.15: Nuova Funzione `getFormData`.

Modifica al Back-End Dopo un'analisi dettagliata del flusso delle chiamate a /run, si è optato per salvare i dati della partita solo al termine della stessa, con la condizione che la vittoria venga assegnata se il punteggio dell'utente (`userscore`) supera quello del robot (`robotScore`).

```

320     private ResponseEntity<String> gestisciPartita(Map<String, String> userData, GameLogic gameLogic, Boolean isGameEnd, int robotScore, String playerId, String mail) {
321         if (userData.get("coverage") != null && !userData.get("coverage").isEmpty()) {
322             // Calcolo copertura e punteggio utente
323             // Il primo è covered e il secondo è missed
324             boolean isWinner = false;
325
326             int[] lineCoverage = getCoverage(userData.get("coverage"), coverageType:"LINE");
327             int[] branchCoverage = getCoverage(userData.get("coverage"), coverageType:"BRANCH");
328             int[] instructionCoverage = getCoverage(userData.get("coverage"), coverageType:"INSTRUCTION");
329
330             int lineCov = LineCoverage(userData.get("coverage"));
331             logger.info("[GAMECONTROLLER] /run: LineCov {}", lineCov);
332
333             int userScore = gameLogic.GetScore(lineCov);
334             logger.info("[GAMECONTROLLER] /run: user_score {}", userScore);
335
336             // Salvo i dati del turno
337             gameLogic.playTurn(userScore, robotScore);
338
339             // Controllo fine partita
340             if (isGameEnd || gameLogic.isGameEnd()) {
341                 activeGames.remove(playerId);
342                 logger.info("[GAMECONTROLLER] /run: risposta inviata con GameEnd true");
343                 // Modifica aggiunta nel 14/12/2024
344                 if (userScore > robotScore) {
345                     isWinner = true;
346                 }
347                 leaderboardService.updateLeaderboard(mail, userScore, isWinner);
348                 return createResponseRun(userData, robotScore, userScore, gameOver:true, lineCoverage, branchCoverage, instructionCoverage);
349             } else {
350                 logger.info("[GAMECONTROLLER] /run: risposta inviata con GameEnd false");
351                 return createResponseRun(userData, robotScore, userScore, gameOver:false, lineCoverage, branchCoverage, instructionCoverage);
352             }
353         } else {
354             // Errori di compilazione
355             logger.info("[GAMECONTROLLER] /run: risposta inviata errori di compilazione");
356             return createResponseRun(userData, robotScore:0, userScore:0, gameOver:false, lineCoverageV>null, null, null);
357         }
358     }

```

Figure 5.16: Nuovo metodo GestisciPartita.

Chapter 6

Deployment diagram finale complessivo dell'architettura

Nel **Diagramma di deployment** non risultano modificati i vari *container* presenti, ma abbiamo deciso di reinserirlo e di mostrare meglio in quali *container* sono presenti dei database (ed in particolare abbiamo inserito anche il tipo di database).

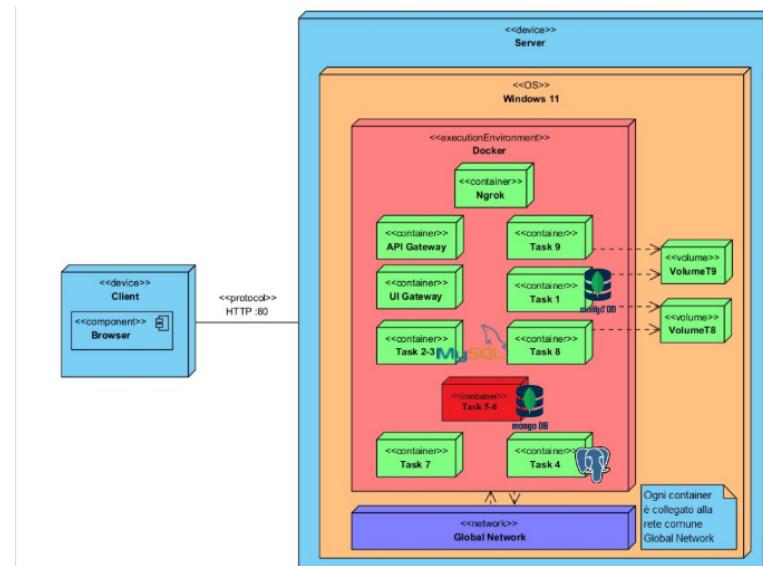


Figure 6.1: Diagramma di deployment.

Chapter 7

Testing

7.1 Testing End to End

L’obiettivo principale del **Test End-to-End** (E2E), noto anche come Test di Sistema, è quello di integrare efficacemente i nuovi componenti con quelli già esistenti all’interno del sistema, assicurandosi che l’applicazione funzioni correttamente nel suo insieme. Questo tipo di test valuta l’intera applicazione simulando scenari reali d’uso, al fine di verificare il corretto funzionamento di tutte le sue parti, che includono l’interfaccia utente, i servizi di *backend*, i database e le comunicazioni di rete.

L’obiettivo del test E2E è quello di convalidare il comportamento complessivo dell’applicazione, tenendo conto di aspetti fondamentali come le funzionalità, l’affidabilità, le prestazioni e la sicurezza. Lo scopo finale è identificare eventuali difetti o problemi che potrebbero sorgere durante l’interazione tra le diverse componenti dell’applicazione.

Per eseguire il test E2E, sono stati impiegati strumenti specifici. Tra questi, **Selenium** è stato utilizzato per il *testing* automatizzato dell'applicazione web, **JUnit** è stato scelto come *framework* di *testing* e **Google Chrome** è stato usato come browser per accedere e testare l'applicazione.

7.1.1 BaseTest

La classe `BaseTest` è responsabile della configurazione iniziale e della chiusura di ogni test. Le operazioni principali eseguite includono:

- Configurazione del WebDriver.
- Apertura della pagina di login.
- Esecuzione del processo di login.
- Verifica della reindirizzamento alla pagina /leaderboard.

```

12  public abstract class BaseTest {
13
14     protected WebDriver driver;
15
16     @BeforeEach
17     public void setup() {
18         WebDriverManager.chromedriver().setup();
19         // Configura le opzioni di Chrome
20         ChromeOptions options = new ChromeOptions();
21         options.addArguments("--no-sandbox"); // Disabilita la sandbox per evitare errori se eseguito come root (non raccomandato in produzione)
22         options.addArguments("--disable-dev-shm-usage"); // Risolve problemi di memoria in ambienti come Docker
23         options.addArguments("--headless"); // Esegue test senza aprire una finestra del browser (utile per ambienti di CI/CD)
24
25         // Crea un'istanza di ChromeDriver con le opzioni configurate
26         driver = new ChromeDriver(options);
27
28         driver.get("http://localhost/login");
29
30         // Login automatico
31         performLogin("email:izio.555@gmail.com", password: "Napolino@");
32     }
33
34     private void performLogin(String email, String password) {
35         driver.findElement(By.id("email")).sendKeys(email);
36         driver.findElement(By.id("password")).sendKeys(password);
37         driver.findElement(By.cssSelector("input[type='submit']")).click();
38
39         // Aggiungi un breve ritardo per attendere il completamento del reindirizzamento
40         try {
41             Thread.sleep(3000); // Aspetta 3 secondi (3000 millisecondi)
42         } catch (InterruptedException e) {
43             e.printStackTrace();
44         }
45         // Verifica che la pagina corrente sia quella corretta
46         Assertions.assertEquals("http://localhost/main", driver.getCurrentUrl(), "L'URL non corrisponde a /main dopo il login.");
47     }
48
49     @AfterEach
50     public void tearDown() {
51         if (driver != null) {
52             driver.quit();
53         }
54     }
55 }
```

Figure 7.1: Codice di BaseTest.

7.1.2 NavigationTest

La classe `NavigationTest` verifica che il pulsante “Classifica”, situato in alto nella pagina, reindirizzi correttamente all’URL desiderato.

```
public class NavigationTest extends BaseTest {  
    @Test  
    void testLeaderboardNavigation() {  
        // Assicurarsi che il login sia stato completato correttamente  
        Assertions.assertEquals("http://localhost/main", driver.getCurrentUrl(), "Il reindirizzamento dopo il login non è corretto.");  
  
        // Clicca sul link della classifica nella navbar  
        WebElement leaderboardLink = driver.findElement(By.linkText("Classifica"));  
        leaderboardLink.click();  
  
        // Aspetta che la navigazione sia completa e verifica l'URL  
        new WebDriverWait(driver, Duration.ofSeconds(10)).until(ExpectedConditions.urlToBe("http://localhost/leaderboard"));  
  
        // Verifica che l'URL sia corretto  
        Assertions.assertEquals("http://localhost/leaderboard", driver.getCurrentUrl(), "La navigazione alla pagina della classifica non è avvenuta correttamente.");  
    }  
}
```

Figure 7.2: Codice di `NavigationTest`.

7.1.3 LeaderboardScoreOrderTest

La classe `LeaderboardScoreOrderTest` valuta la corretta implementazione dell’ordinamento per “totalscores” nella pagina della classifica. Le operazioni includono:

- Reindirizzamento a `/leaderboard`.
- Selezione dell’opzione di ordinamento.
- Verifica dell’ordinamento corretto dei dati nella pagina.

```
public class LeaderboardScoreOrderTest extends BaseTest {
    @Test
    public void testScoresAreOrderedCorrectly() {
        // Assicurarsi di essere nella pagina di leaderboard
        driver.get("http://localhost/leaderboard");

        // Selezionare l'ordine per 'totalScore' se non già selezionato
        WebElement sortOrderSelect = new WebDriverWait(driver, Duration.ofSeconds(10))
            .until(ExpectedConditions.elementToBeClickable(By.id("sortOrder")));
        sortOrderSelect.click();

        WebElement totalScoreOption = sortOrderSelect.findElement(By.xpath("//option[@value='totalScore']"));
        if (!totalScoreOption.isSelected())
            totalScoreOption.click();

        // Ricarica la pagina con l'ordinamento corretto
        driver.navigate().refresh();

        List<WebElement> voidListtest = driver.findElements(By.xpath("//tbody/tr/td[5]")); // Assumendo che totalScore sia la quinta colonna
        if (voidListtest.isEmpty())
            System.out.println("Nessun giocatore presente nella leaderboard.");
        return; // Esce dal test se non ci sono giocatori
    }

    // Raccogliere i punteggi dal sito e trasformarli in una lista di Integer
    List<Integer> scores = driver.findElements(By.xpath("//tbody/tr/td[5]")) // Assumendo che totalScore sia la quinta colonna
        .stream()
        .map(WebElement::getText)
        .map(Integer::parseInt)
        .collect(Collectors.toList());

    // Verificare che i punteggi siano ordinati in modo decrescente
    for (int i = 0; i < scores.size() - 1; i++) {
        Assertions.assertThat(scores.get(i) >= scores.get(i + 1), "I punteggi non sono ordinati correttamente.");
    }
}
}
```

Figure 7.3: Codice di LeaderboardScoreOrderTest.

7.1.4 LeaderboardWinsOrderTest

La classe `LeaderboardWinsOrderTest` esamina l'efficacia dell'ordinamento per numero di vittorie nella pagina della classifica. Le operazioni compiute sono:

- Reindirizzamento a `/leaderboard`.
- Selezione dell'opzione di ordinamento per vittorie.
- Controllo dell'accuratezza dell'ordinamento dei dati nella pagina.

```

public class LeaderboardWinsOrderTest extends BaseTest {
    @Test
    public void testScoresAreOrderedCorrectly() {
        // Assicurarsi di essere nella pagina di leaderboard
        driver.get("http://localhost/leaderboard");

        // Selezionare l'ordine per 'totalscore' se non già selezionato
        WebElement sortOrderSelect = new WebDriverWait(driver, Duration.ofSeconds(10))
            .until(ExpectedConditions.elementToBeClickable(By.id("sortOrder")));
        sortOrderSelect.click();
        WebElement totalScoreOption = sortOrderSelect.findElement(By.xpath("//option[@value='wins']"));
        if (!totalScoreOption.isSelected()) {
            totalScoreOption.click();
        }

        // Ricarica la pagina con l'ordinamento corretto
        driver.navigate().refresh();

        List<WebElement> voidListTest = driver.findElements(By.xpath("//tbody/tr/td[5]")); // Assumendo che totalscore sia la quinta colonna
        if (voidListTest.isEmpty()) {
            System.out.println("Nessun giocatore presente nella leaderboard.");
            return; // Esce dal test se non ci sono giocatori
        }

        // Raccogliere i punteggi dal sito e trasformarli in una lista di Integer
        List<Integer> scores = driver.findElements(By.xpath("//tbody/tr/td[4]")) // Assumendo che totalScore sia la quinta colonna
            .stream()
            .map(WebElement::getText)
            .map(Integer::parseInt)
            .collect(Collectors.toList());

        // Verificare che i punteggi siano ordinati in modo decrescente
        for (int i = 0; i < scores.size() - 1; i++) {
            Assertions.assertTrue(scores.get(i) >= scores.get(i + 1), "I punteggi non sono ordinati correttamente.");
        }
    }
}

```

Figure 7.4: Codice di LeaderboardWinsOrderTest.

7.1.5 LeaderboardTotalMatchesOrderTest

La classe `LeaderboardTotalMatchesOrderTest` controlla se l'ordinamento per numero di partite giocate è stato implementato correttamente. Le operazioni svolte includono:

- Reindirizzamento a `/leaderboard`.
- Selezione dell'opzione di ordinamento per partite giocate.
- Verifica dell'ordinamento corretto dei dati nella pagina.

```
public class LeaderboardTotalMatchesOrderTest extends BaseTest {  
    @Test  
    public void testScoresAreOrderedCorrectly() {  
        // Assicurarsi di essere nella pagina di leaderboard  
        driver.get("http://localhost/leaderboard");  
  
        // Selezionare l'ordine per 'totalScore' se non già selezionato  
        WebElement sortOrderSelect = new WebDriverWait(driver, Duration.ofSeconds(10))  
            .until(ExpectedConditions.elementToBeClickable(By.id("sortOrder")));  
        sortOrderSelect.click();  
        WebElement totalScoreoption = sortOrderSelect.findElement(By.xpath("//option[@value='totalMatches']"));  
        if (!totalScoreoption.isSelected()) {  
            totalScoreoption.click();  
        }  
  
        // Ricarica la pagina con l'ordinamento corretto  
        driver.navigate().refresh();  
  
        List<WebElement> voidlisttest = driver.findElements(By.xpath("//tbody/tr/td[5]")); // Assumendo che totalscore sia la quinta colonna  
        if (voidlisttest.isEmpty()) {  
            System.out.println("Nessun giocatore presente nella leaderboard.");  
            return; // Esce dal test se non ci sono giocatori  
        }  
  
        // Raccogliere i punteggi dal sito e trasformarli in una lista di Integer  
        List<Integer> scores = driver.findElements(By.xpath("//tbody/tr/td[3]")) // Assumendo che totalScore sia la quinta colonna  
            .stream()  
            .map(WebElement::getText)  
            .map(Integer::parseInt)  
            .collect(Collectors.toList());  
  
        // Verificare che i punteggi siano ordinati in modo decrescente  
        for (int i = 0; i < scores.size() - 1; i++) {  
            Assertions.assertTrue(scores.get(i) >= scores.get(i + 1), "I punteggi non sono ordinati correttamente.");  
        }  
    }  
}
```

Figure 7.5: Codice di LeaderboardTotalMatchesOrderTest.

Chapter 8

Sviluppi futuri

Per quanto concerne i possibili **sviluppi futuri** con cui continuare il lavoro che abbiamo fatto, ampliando le possibilità, da parte dell'utente, di operare con la classifica, essi, a nostro parere potrebbero essere:

- **Creazione classifiche per modalità di gioco:** aggiungere nuove classifiche filtrate per ciascuna modalità di gioco presente;
- **Creazione classifiche per classi testate:** aggiungere nuove classifiche per numero e per tipo delle classi testate;
- **Creazione classifiche per amici:** possibilità di visualizzare, da parte dell'utente, una classifica dove sono presenti solo gli utenti amici;
- **Visualizzazione profilo utente da classifica:** possibilità di cliccare su un utente presente nella classifica e visualizzarne un'anteprima delle informazioni del profilo utente.