



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria
Corso di Laurea in Ingegneria Informatica

Progetto di Ingegneria del Software

Applicazione per la gestione di campionati Fanta-sports

RIALTI JACOPO, TESCARO ROCCO

Anno Accademico 2023-2024

CONTENTS

1	Introduzione	3
1.1	Obiettivi del progetto	3
1.2	Elementi del progetto	3
1.3	Estendibilità del progetto	5
2	Analisi dei requisiti	7
2.1	Use Case Diagram	7
2.2	Use Case Templates	8
2.2.1	Casi d'uso principali di uno User	8
2.2.2	Casi d'uso principali di un Creator	10
2.3	Mockups	11
2.3.1	Create league	12
2.3.2	Join league	12
2.3.3	Add Players to a team	13
2.3.4	Set Players votes	13
2.3.5	Evaluate current Day	13
2.3.6	Set team Formation	14
2.3.7	League details	14
2.4	Class Diagram	15
2.5	DAO Pattern	16
2.6	MVC Pattern	17
2.7	Entity Relationship Diagram	18
3	Implementazioni delle classi	19
3.1	entities	20
3.1.1	User	20
3.1.2	Team	20
3.1.3	Player	21
3.1.4	League	21
3.2	gui	22
3.2.1	LoginPage	22
3.2.2	MainPage	22
3.2.3	CreateNewLeaguePage	23
3.2.4	LeagueDetailsPage	24
3.3	daosystem	25
3.3.1	UserDao	25
3.3.2	TeamDao	26

3.3.3	PlayerDao	27
3.3.4	LeagueDao	28
3.4	repositories	28
3.5	resources	29
3.5.1	static	29
3.5.2	templates	29
4	Test (JUnit)	31
4.1	UserTest	31
4.2	TeamTest	32
4.3	PlayerTest	33
4.4	LeagueTest	34

INTRODUZIONE

1.1 OBIETTIVI DEL PROGETTO

Il progetto corrente nasce con l'obiettivo di realizzare un'applicazione per la gestione di campionati Fanta-sports, in particolare di calcio. E' ormai diffusissimo il fenomeno dei campionati fanta-sports, dove i giocatori hanno la possibilità di amministrare le proprie squadre favorite e sfidarsi con altri giocatori. L'applicazione nasce appunto con l'obiettivo di fornire un servizio/strumento per l'organizzazione di campionati privati e per estensione anche pubblici.

L'applicazione permette di creare leghe private di fanta-soccer, ovvero un campionato in cui gli utenti ad invito possono creare una squadra virtuale, composta da giocatori reali, e sfidarsi, con anche la possibilità di estendere le regole del campionato personalizzandole o possibilmente collegare un proprio database all'applicazione rendendo possibile la creazione di campionati di altri sport.

Le principali funzionalità dell'applicazione sono: la possibilità di creare un campionato, di invitare altri utenti a partecipare, di avviare un'asta e comporre una squadra, di definire modalità di punteggio e regole del campionato, di visualizzare la classifica e i risultati delle partite.

1.2 ELEMENTI DEL PROGETTO

Il progetto è composto da:

- **Utente giocatore (User):** è l'utente che partecipa al campionato, partecipare all'asta, comporre e gestire la propria squadra, visualizzare la classifica e i risultati delle partite. Non può modificare creare un campionato o modificare le regole di uno esistente (nemmeno se partecipante dello stesso).

- **Utente amministratore (Creator):** è l'utente che crea un campionato, invita altri utenti a partecipare, avvia l'asta, modifica le regole. Scelte le regole, l'utente amministratore non può più modificarle a campionato iniziato. Ha anche il compito di fornire il database dei giocatori (reali) e il database delle valutazioni per il calcolo del punteggio.
- **Squadra (Team):** Rappresenta la squadra di un utente in un determinato campionato. Ogni squadra è composta da un insieme di giocatori reali selezionati dall'utente durante l'asta. Ogni squadra ha un proprietario (l'utente che l'ha creata) e appartiene a una lega specifica. Le squadre accumulano punti in base alle prestazioni dei loro giocatori nelle formazioni nelle partite reali, secondo le regole a punti. Ogni squadra ha anche una formazione, che è l'insieme di giocatori selezionati per giocare in una determinata giornata del campionato. La formazione può essere modificata dall'utente proprietario prima dell'inizio di ogni giornata.
- **Giocatore sportivo (Player):** Rappresenta un giocatore reale nel contesto del campionato. Ogni giocatore ha un punteggio che determina le sue prestazioni nelle partite reali. Queste prestazioni influenzano i punti accumulati dalle squadre nelle quali il giocatore è stato selezionato. Ogni giocatore può appartenere a più squadre e leghe, a seconda delle scelte degli utenti durante l'asta. Inoltre, un giocatore può essere parte della formazione di una squadra, che è l'insieme di giocatori selezionati per giocare in una determinata giornata del campionato.
- **Lega (League):** Rappresenta un campionato nel contesto del gioco. Ogni lega è creata da un utente amministratore e può avere un numero specifico di partecipanti. Le leghe hanno regole specifiche, come il numero di giocatori in una formazione e il numero di utenti giocatori partecipanti. Ogni lega ha anche una data di inizio e che indica quando sarà la prossima giornata. Inoltre, ogni lega ha un insieme completo di giocatori reali che possono essere selezionati dagli utenti durante l'asta. Infine, ogni lega ha un insieme di squadre, ciascuna delle quali appartiene a un utente specifico che partecipa alla lega.

1.3 ESTENDIBILITÀ DEL PROGETTO

Il progetto è stato pensato per essere estendibile, in particolare per quanto riguarda la possibilità di creare campionati di altri sport. Per fare ciò è necessario fornire un database di giocatori e un database di valutazioni per il calcolo del punteggio.

Un'altra possibilità di estensione è la possibilità di avviare leghe pubbliche, ovvero leghe il cui accesso è disponibile a tutti gli utenti dell'applicazione, senza quindi necessità di invito da parte dell'amministratore. A questa estensione è legata anche la possibilità di visualizzare una classifica globale, ovvero una classifica che tiene conto di tutti i campionati pubblici attivi.

In ultima istanza è possibile estendere l'applicazione affinché siano gestite altre regole o tipologie di campionati, idealmente non vincolare dalla regola dell'immutabilità delle regole a campionato iniziato.

ANALISI DEI REQUISITI

2.1 USE CASE DIAGRAM

Il software presenta 2 diversi tipi di attori: lo User e il Creator. Lo user può partecipare ad una lega creata da un altro Creator, gestendo il suo team all'interno della lega. Lo user per diventare un Creator, deve creare una nuova lega, nella quale avrà poi la possibilità di gestire e modificare tutte le varie opzioni, squadre e giocatori all'interno di essa. Lo schema dei casi d'uso sottostante è stato realizzato mediante StarUML secondo lo standard UML.

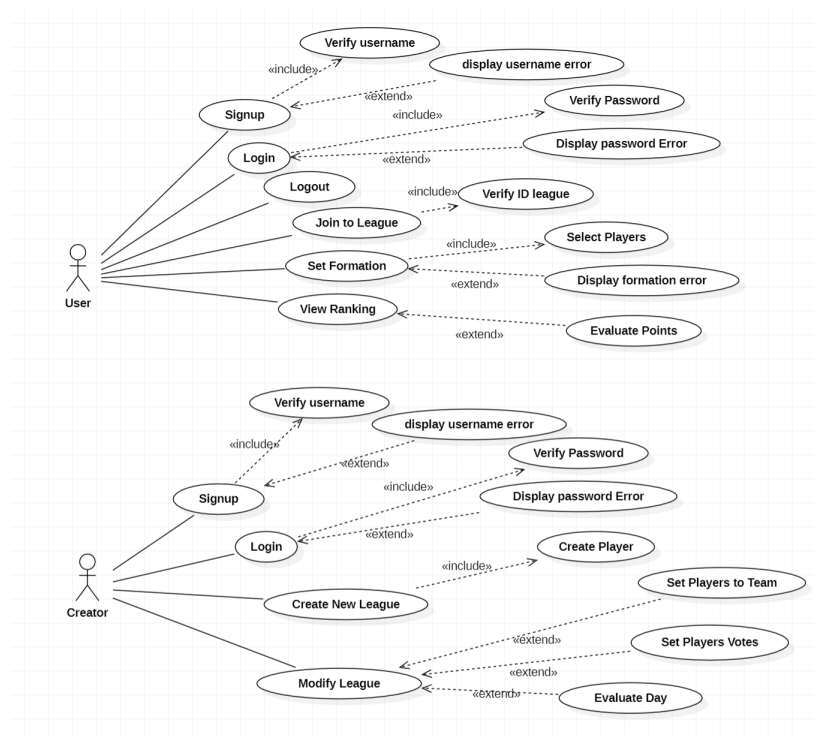


Figure 1: Use Case Diagram

2.2 USE CASE TEMPLATES

2.2.1 *Casi d'uso principali di uno User*

UC-1	Join to League
Descrizione	Lo user effettua l'accesso alla lega, inserendo l'ID di quest'ultima nell'apposito campo e diventando un partecipante.
Livello	User goal
Attore principale	User
Azioni	<ol style="list-style-type: none"> 1. Lo user, dalla schermata home, inserisce l'ID della lega nel apposito form "Join to League". 2. Se l'ID è corretto e la capienza della lega non è piena, effettuerà l'accesso. 3. L'utente è diventato un partecipante della lega e potrà vedere tutte le informazioni di quest'ultima.
Casi straordinari	2a. Nel caso l'ID della lega non corrisponde ad una lega esistente oppure la lega ha già raggiunto il numero dei partecipanti, lo User non potrà partecipare.

Table 1: Partecipazione dello User ad una lega già creata

UC-2	Set Formation
Descrizione	Lo user dalla pagina della lega, potrà selezionare i giocatori presenti nel suo team, inserendoli nella rosa per la prossima giornata.
Livello	User goal
Attore principale	User
Azioni	<ol style="list-style-type: none"> 1. Lo user clicca sulla relativa lega dalla schermata home. 2. Lo user seleziona i giocatori che vuole inserire nella rosa, attraverso un multiple choice. 3. Lo user, una volta selezionati, clicca su Set Formation, salvando così i giocatori selezionati nella rosa.
Casi straordinari	<ol style="list-style-type: none"> 2a. Nel caso in cui il numero di giocatori selezionati non corrisponde al numero di giocatori che devono essere inseriti nella rosa, non potrà cliccare sul pulsante per salvare quest'ultima. 2b. Se lo user si scorda di inserire la rosa per una determinata giornata, il calcolo della giornata verrà effettuato sull'ultima rosa salvata.

Table 2: Inserimento della formazione titolare

2.2.2 *Casi d'uso principali di un Creator*

UC-3	Create New League
Descrizione	Uno user, cliccando sul pulsante nella home page "Create New League", può creare una nuova lega, inserendo tutte le informazioni relative a questo campionato. Lo user dopodiché diventerà Creator di quest'ultima.
Livello	User goal
Attore principale	Creator
Azioni	<ol style="list-style-type: none"> 1. Il creator dalla schermata home clicca su "Create League". 2. Inserisce le informazioni richieste (nome, numero di partecipanti, pool dei players, start date, etc.). 3. Clicca su "Create" per creare la lega.
Casi straordinari	2a. Nel caso in cui il Creator imposta dei dati non validi riguardanti il set-up della Lega e clicca su Save, verrà reindirizzato nuovamente sulla stessa pagina ma con tutti i campi azzerati.

Table 3: Creazione di una nuova lega

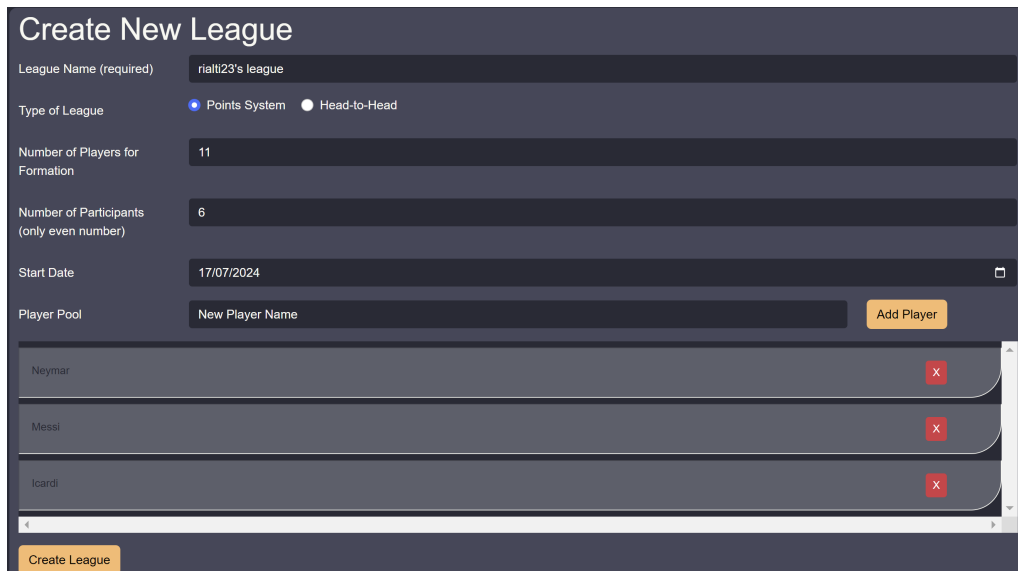
UC-4	Modify League
Descrizione	Il creator può modificare una lega esistente, calcolare il punteggio della giornata, inserire la data della prossima giornata, assegnare giocatori ai vari partecipanti e rimuovere partecipanti.
Livello	User goal
Attore principale	Creator
Azioni	<ol style="list-style-type: none"> 1. Il creator seleziona la lega da modificare dalla schermata home. 2. Inserisce la data della prossima giornata. 3. Calcola il punteggio della giornata. 4. Assegna giocatori ai vari partecipanti. 5. Rimuove partecipanti, se necessario.
Casi straordinari	<ol style="list-style-type: none"> 2a. Nel caso in cui il creator inserisca una data non valida, viene mostrato un messaggio di errore. 4a. Se l'assegnazione dei giocatori non è valida, viene mostrato un messaggio di errore e il creator deve correggere l'assegnazione.

Table 4: Modifica di una lega

2.3 MOCKUPS

Ecco alcuni Mockups che sono veri e propri screenshot della GUI realizzata a questo proposito.

2.3.1 *Create league*

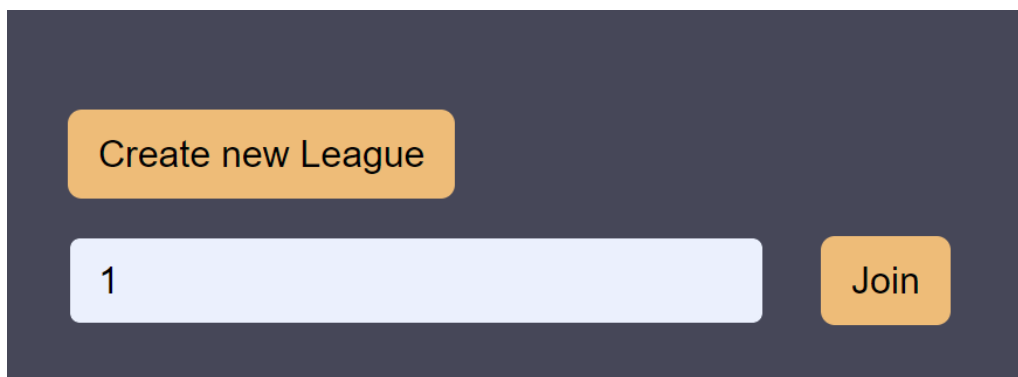


The screenshot shows a 'Create New League' form with the following fields and options:

- League Name (required):** riali23's league
- Type of League:** ☒ Points System ☐ Head-to-Head
- Number of Players for Formation:** 11
- Number of Participants (only even number):** 6
- Start Date:** 17/07/2024
- Player Pool:** A list of players with a search bar and an 'Add Player' button. The list includes Neymar, Messi, and Icardi, each with a red 'x' icon to its right.
- Create League:** A yellow button at the bottom left.

Figure 2: Crea una nuova lega

2.3.2 *Join league*

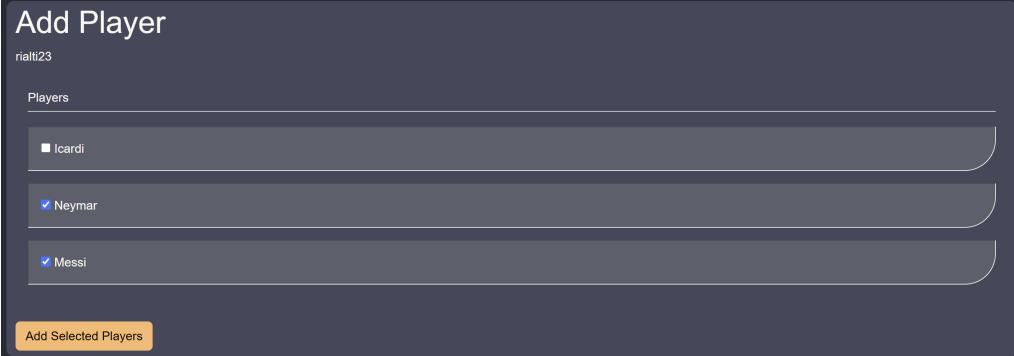


The screenshot shows a 'Join league' form with the following elements:

- Create new League:** A yellow button at the top left.
- 1:** A light blue input field containing the number 1.
- Join:** A yellow button at the bottom right.

Figure 3: Partecipa ad una lega già creata

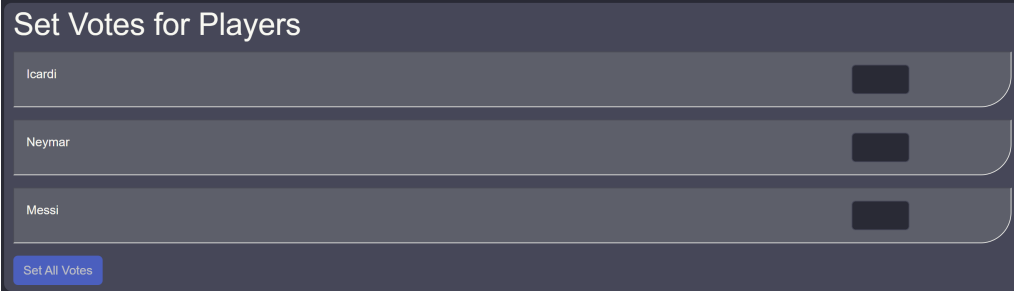
2.3.3 *Add Players to a team*



The mockup shows a dark-themed form titled "Add Player" with a subtitle "rialti23". Below the title is a section labeled "Players" containing three rows. Each row has a checkbox and a player name: the first row has an unchecked checkbox and "Icardi"; the second row has a checked checkbox and "Neymar"; the third row has a checked checkbox and "Messi". At the bottom of the form is an orange button labeled "Add Selected Players".

Figure 4: Aggiungi giocatori ad un Team

2.3.4 *Set Players votes*



The mockup shows a dark-themed form titled "Set Votes for Players". It contains three rows, each with a player name and a rating input field: "Icardi" with a value of 1, "Neymar" with a value of 2, and "Messi" with a value of 3. At the bottom of the form is a blue button labeled "Set All Votes".

Figure 5: Assegna le valutazioni a tutti i giocatori della lega

2.3.5 *Evaluate current Day*



The mockup shows a dark-themed form titled "Evaluate this Day". It has a label "Set Next Day Date" followed by a date input field containing "25/07/2024" and a calendar icon. At the bottom of the form is an orange button labeled "Evaluate".

Figure 6: Calcola punteggio giornata e assegna nuova data della prossima

2.3.6 *Set team Formation*

The screenshot shows a dark-themed interface for setting a team formation. At the top, there's a header 'My Formation'. Below it, a section titled 'Set Formation' contains two horizontal bars, each with a small square icon and a player's name: 'Icardi' and 'Neymar'. At the bottom of this section is a blue button labeled 'Add Selected Players to Formation'.

Figure 7: Seleziona i giocatori da inserire nella formazione titolare

2.3.7 *League details*

The screenshot displays a dark-themed interface for league details. At the top left is an orange 'Modify' button. Below it, league information is presented in a grid-like format:

League Name: rialti23's league	Number of Participants: 6	Number of Formations: 11
Type: points	Status:	Start Date: 2024-07-26

Below this information is a section titled 'Ranking' containing a table with two rows:

1	Points: 0
2	Points: 0

The names 'rialti23' and 'rialti22' are visible below the first and second rows respectively.

Figure 8: Visualizza informazioni riguardo la lega

2.4 CLASS DIAGRAM

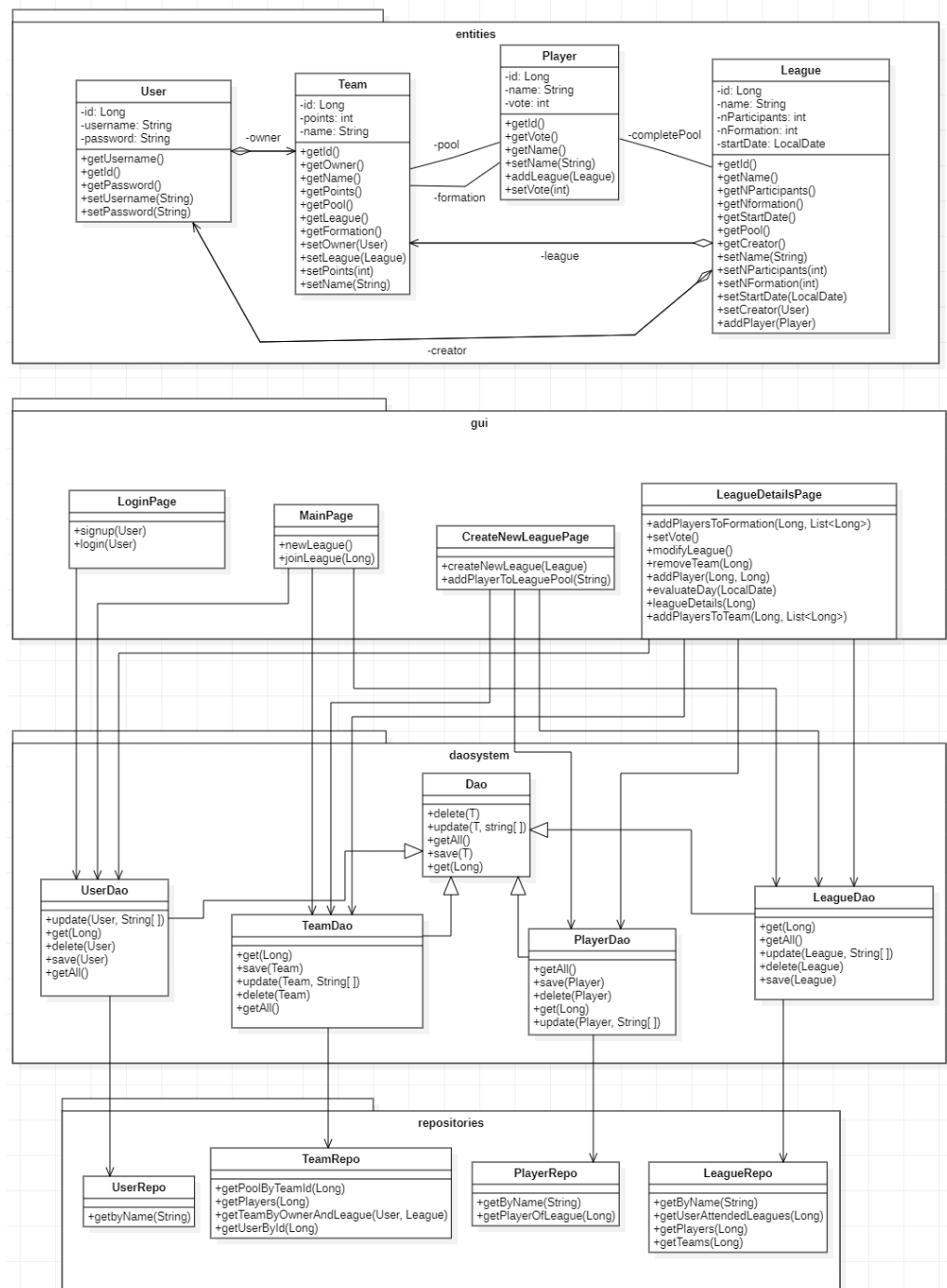


Figure 9: Class Diagram

Il codice contiene all'interno tali packages:

- **entities**: rappresenta le varie entità all'interno del software. Contiene tutte le classi tramite le quali vengono rappresentati tutti gli oggetti nel software. All'interno delle classi sono definiti gli attributi delle varie entità.
- **gui**: contiene le classi che si occupano di modificare i dati, comunicando esclusivamente con dei servizi per poter accedere al database senza farli direttamente. In questo package sono presenti, dunque, tutti i controller per i modelli mappati nel Database.
- **dao**: contiene le interfacce che definiscono i metodi per accedere ai dati. Poiché il DAO è un pattern architetturale, è possibile creare diverse implementazioni di queste interfacce per accedere a diversi tipi di database.
- **repositories**: contiene le classi che implementano le interfacce DAO. Queste classi sono responsabili di accedere ai dati e di eseguire le operazioni CRUD (Create, Read, Update, Delete) sui dati e query personalizzate.

È importante puntualizzare che le classi nella repository **dao** non sono strettamente necessarie per il funzionamento del software, SpringBoot fornisce già un'astrazione per l'accesso al database che è poi stata effettivamente sfruttata nel package **repositories**. Tuttavia, l'implementazione del DAO pattern è stata effettuata per una maggiore chiarezza e separazione delle responsabilità all'interno del software, per dimostrare una padronanza del pattern e per facilitare eventuali modifiche future.

2.5 DAO PATTERN

Abbiamo utilizzato il Design Pattern DAO in modo che il client non possa intergere direttamente con il Database.

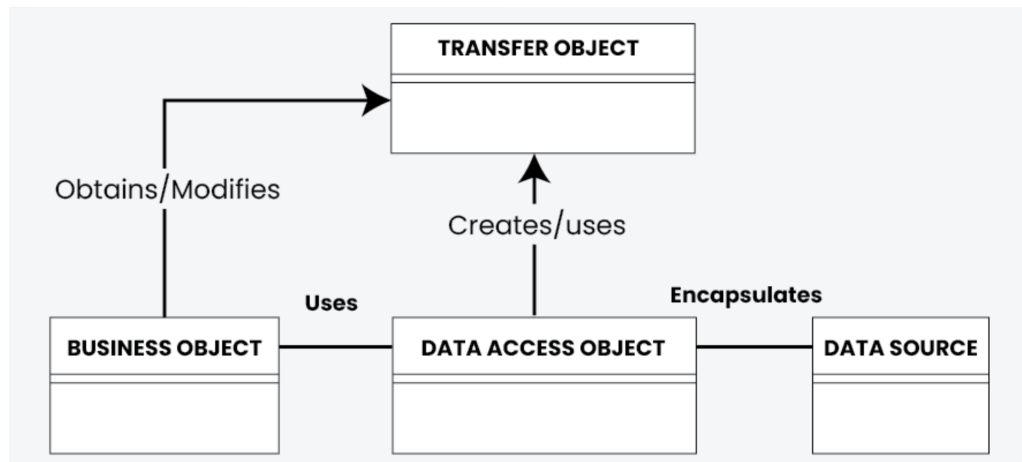


Figure 10: General DAO Diagram

Il pattern architetturale DAO (Data Access Object) viene utilizzato per gestire la persistenza dei dati registrati in un database. La persistenza si riferisce alla capacità di un dato di "sopravvivere" all'esecuzione di un programma, ovvero di continuare a esistere e rimanere coerente anche dopo la fine del programma stesso. Questo pattern ha lo scopo di separare la logica di accesso ai dati dal resto dell'applicazione. È progettato per isolare il codice che si occupa dell'accesso e della manipolazione dei dati da quello che implementa la logica di business del nostro software. La classe DAO definisce e offre metodi CRUD (Create, Read, Update, Delete) per i dati presenti nel database.

Le classi che sfruttano il DAO e che sono presenti nel DB sono le seguenti:

- User;
- Team;
- League;
- Player.

2.6 MVC PATTERN

Sebbene, al contrario del pattern DAO, il pattern MVC non sia stato implementato in modo esplicito, è possibile identificare le varie componenti del pattern all'interno del progetto. Il pattern Model-View-Controller (MVC) è un pattern architetturale che separa i dati (Model), l'interfaccia

utente (View) e la logica di controllo (Controller) di un'applicazione. Questo pattern permette di separare le responsabilità all'interno del software, facilitando la manutenzione e l'estensione del codice.

Come per il Dao, SpringBoot (starter-web) fornisce già un'implementazione del pattern MVC, che in questo caso non è stato ri-implementato, specificando ulteriormente la separazione fra le varie componenti.

Nel caso del progetto corrente (ma, essendo un impiego comune, anche in altri progetti), il **Model** coincide con le classi presenti nel package entities, vale a dire i dati del database; la **View** è rappresentata dalle pagine HTML che vengono visualizzate dall'utente; il **Controller** è rappresentato dalle classi presenti nel package gui, che si occupano di gestire le richieste dell'utente e di aggiornare il modello di dati (sfruttando il pattern DAO) in base alle azioni dell'utente.

Le classi che sfruttano il MVC sono le seguenti:

- LoginPage;
- MainPage;
- CreateNewLeaguePage;
- LeagueDetailsPage.

2.7 ENTITY RELATIONSHIP DIAGRAM

Questo è il diagramma ER del progetto, con le varie cardinalità delle relazioni.

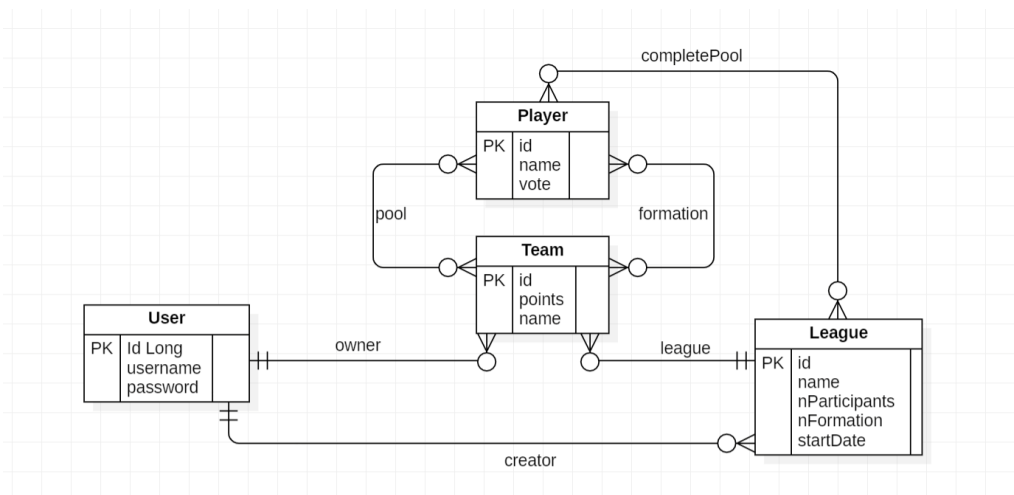


Figure 11: Entity Relationship Diagram

IMPLEMENTAZIONI DELLE CLASSI

Abbiamo suddiviso il progetto in package Java: il modello dati nel package entities, la logica di controllo nel package gui ed infine la gestione del Database nel package dao e repositories. Il codice è suddiviso in tale modo:

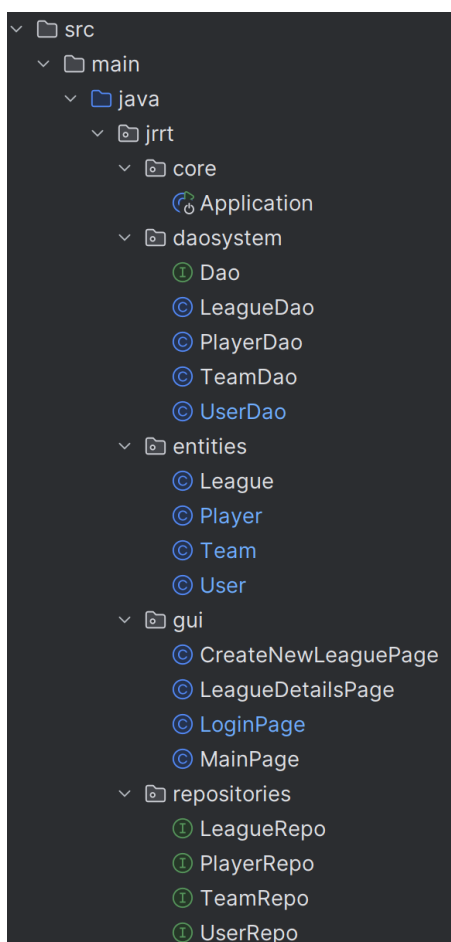


Figure 12: Suddivisione dei package

3.1 ENTITIES

Il package entities si occupa di definire un modello di composizione di classi su cui è possibile eseguire i casi d'uso espressi nello Use Case Diagram. Questo package si articola nelle seguenti classi:

3.1.1 *User*

Lo User è un utente, che registrandosi alla webApp, può creare nuove leghe o partecipare ad altre create da altri User. Gli attributi sono:

- id: un identificativo univoco dell'utente.
- username: una stringa che contiene il nickname con il quale l'utente vuole chiamarsi nel programma.
- password: una stringa relazionata allo username per permettere il login.

3.1.2 *Team*

Il Team è una squadra, che viene creata quando un utente partecipa o crea una lega. Gli attributi sono:

- id: un identificativo univoco della squadra.
- name: una stringa che contiene il nome della squadra.
- points: un intero che rappresenta i punti accumulati dalla squadra durante tutto il campionato.
- owner: un oggetto User che rappresenta l'utente proprietario della squadra.
- league: un oggetto League che rappresenta la lega a cui appartiene la squadra.
- pool: un insieme di oggetti Player che rappresentano i giocatori disponibili per la squadra.
- formation: un insieme di oggetti Player che rappresentano la formazione attuale della squadra per la prossima giornata.

3.1.3 *Player*

Il Player è un giocatore, che può essere aggiunto a una squadra da il creatore della lega. Gli attributi sono:

- id: un identificativo univoco del giocatore.
- name: una stringa che contiene il nome del giocatore.
- vote: un intero che rappresenta il voto del giocatore nell'ultima partita.
- leagues: un insieme di oggetti League che rappresentano le leghe a cui il giocatore appartiene.
- teams: un insieme di oggetti Team che rappresentano le squadre a cui il giocatore appartiene come membro del pool.
- teamsFormation: un insieme di oggetti Team che rappresentano le squadre in cui il giocatore è attualmente in formazione.

3.1.4 *League*

La League è una lega, che può essere creata da un utente. Gli attributi sono:

- id: un identificativo univoco della lega.
- name: una stringa che contiene il nome della lega.
- nParticipants: un intero che rappresenta il numero di partecipanti alla lega.
- nFormation: un intero che rappresenta il numero di formazioni per squadra nella lega.
- startDate: una data che rappresenta la data in cui viene calcolata la prossima giornata.
- creator: un oggetto User che rappresenta l'utente che ha creato la lega.
- teams: un insieme di oggetti Team che rappresentano le squadre che partecipano alla lega.
- completePool: un insieme di oggetti Player che rappresentano i giocatori disponibili per la lega.

3.2 GUI

Il package gui (Graphical User Interface) si occupa della gestione dell'interfaccia utente dell'applicazione. Questo package contiene le classi che definiscono come l'utente interagisce con l'applicazione, creando, modificando o elementi dell'entities, ma mai accedendo direttamente al DataBase.

3.2.1 *LoginPage*

La classe LoginPage è un controller Spring MVC che gestisce le operazioni di login e registrazione degli utenti. Ecco i principali metodi implementati nella classe:

- **loginPage:** Questo metodo gestisce le richieste GET alla radice dell'applicazione ("/"). Prepara un nuovo oggetto User e lo aggiunge al modello, quindi restituisce la vista "loginPage" per essere visualizzata.
- **signup:** Questo metodo gestisce le richieste POST a "/signup". Prende un oggetto User come parametro, controlla se esiste già un utente con lo stesso nome utente nel database. Se non esiste, salva il nuovo utente nel database e restituisce la vista "loginPage" con un messaggio informativo. Se esiste, restituisce la vista "loginPage" con un messaggio di errore.
- **login:** Questo metodo gestisce le richieste POST a "/login". Prende un oggetto User come parametro, cerca un utente con lo stesso nome utente nel database. Se l'utente esiste e la password corrisponde, imposta l'utente come l'utente attualmente loggato nella sessione e reindirizza alla pagina principale. Se l'utente non esiste o la password non corrisponde, restituisce la vista "loginPage" con un messaggio di errore.

3.2.2 *MainPage*

La classe MainPage è un controller Spring MVC che gestisce le operazioni principali dell'applicazione. Ecco i principali metodi implementati nella classe:

- **mainPage:** Questo metodo gestisce le richieste GET a "/main". Rimuove l'attributo "league" dalla sessione, recupera l'utente attuale dalla sessione e, se l'utente non è null, recupera le leghe a cui

l'utente partecipa, le ordina per data di inizio e le aggiunge al modello. Infine, restituisce la vista "mainPage" per essere visualizzata.

- **newLeague:** Questo metodo gestisce le richieste GET a "/newLeague". Recupera l'utente e la lega dalla sessione. Se la lega è null, crea una nuova lega con valori predefiniti e la aggiunge alla sessione. Recupera i giocatori della lega e li aggiunge al modello. Infine, restituisce la vista "createNewLeaguePage" per essere visualizzata.
- **joinLeague:** Questo metodo gestisce le richieste POST a "/joinLeague". Prende un parametro "leagueId" che rappresenta l'ID della lega a cui l'utente vuole unirsi. Recupera l'utente dalla sessione e la lega dal database. Se la lega esiste e non è piena, crea una nuova squadra con l'utente come proprietario e la lega come lega, salva la squadra nel database e reindirizza alla pagina principale. Se la lega non esiste o è piena, restituisce un messaggio di errore e reindirizza alla pagina principale.

3.2.3 *CreateNewLeaguePage*

La classe *CreateNewLeaguePage* è un controller Spring MVC che gestisce la creazione di nuove leghe. Ecco i principali metodi implementati nella classe:

- **createNewLeague:** Questo metodo gestisce le richieste POST a "/createNewLeague". Prende un oggetto *League* come parametro, controlla se esiste già una lega con lo stesso nome nel database. Se non esiste, salva la nuova lega nel database, crea una nuova squadra con l'utente come proprietario e la lega come lega, salva la squadra nel database e rimuove l'attributo "league" dalla sessione. Infine, reindirizza alla pagina principale.
- **addPlayerToLeaguePool:** Questo metodo gestisce le richieste POST a "/addPlayer/playerName". Prende una stringa *playerName* come parametro, recupera l'utente e la lega dalla sessione. Se l'utente o la lega sono null, reindirizza alla pagina principale. Recupera il pool di giocatori della lega e controlla se esiste già un giocatore con lo stesso nome nel pool. Se esiste, restituisce un messaggio di errore e reindirizza alla pagina "newLeague". Se non esiste, crea un nuovo giocatore con il nome fornito, aggiunge il giocatore alla lega, salva il giocatore nel database e reindirizza alla pagina "newLeague".

3.2.4 *LeagueDetailsPage*

La classe *LeagueDetailsPage* è una Spring MVC controller che funziona mostrando e gestendo tutti i dettagli specifici di una lega. Ecco i principali metodi implementati nella classe:

- **leagueDetails:** Questo metodo gestisce le richieste GET a `"/leagueDetails/id"`. Recupera l'utente e la lega dalla sessione. Se l'utente o la lega sono null, reindirizza alla pagina principale. Altrimenti, aggiunge l'utente, la lega, le squadre della lega e i giocatori della lega al modello. Infine, restituisce la vista `"leagueDetailsPage"` per essere visualizzata.
- **modifyLeague:** Questo metodo gestisce le richieste GET a `"/modifyLeague"`. Recupera l'utente e la lega dalla sessione. Se l'utente o la lega sono null, reindirizza alla pagina principale. Altrimenti, aggiunge l'utente, la lega, le squadre della lega e i giocatori della lega al modello. Infine, restituisce la vista `"modifyLeaguePage"` per essere visualizzata.
- **removeTeam:** Questo metodo gestisce le richieste GET a `"/removeTeam/id"`. Prende un parametro `"id"` che rappresenta l'ID della squadra da rimuovere. Recupera la squadra dal database e la rimuove. Se la lega non ha più squadre, rimuove anche la lega. Infine, reindirizza alla pagina `"modifyLeague"`.
- **addPlayer:** Questo metodo gestisce le richieste GET a `"/addPlayer/userId/teamId"`. Prende due parametri `"userId"` e `"teamId"` che rappresentano l'ID dell'utente e l'ID della squadra rispettivamente. Recupera l'utente, la squadra e la lega dalla sessione. Se l'utente, la squadra o la lega sono null, reindirizza alla pagina principale. Altrimenti, aggiunge l'utente, la squadra e i giocatori della lega al modello. Infine, restituisce la vista `"addPlayersPage"` per essere visualizzata.
- **addPlayersToTeam:** Questo metodo gestisce le richieste POST a `"/addPlayersToTeam/teamId"`. Prende un parametro `"teamId"` che rappresenta l'ID della squadra e una lista di ID di giocatori. Aggiunge i giocatori alla squadra e salva la squadra nel database. Infine, reindirizza alla pagina `"modifyLeague"`.
- **addPlayersToFormation:** Questo metodo gestisce le richieste POST a `"/addPlayersToFormation/teamId"`. Prende un parametro `"teamId"`

che rappresenta l'ID della squadra e una lista di ID di giocatori. Aggiunge i giocatori alla formazione della squadra e salva la squadra nel database. Infine, reindirizza alla pagina "leagueDetails/teamId".

- `setVote`: Questo metodo gestisce le richieste POST a `"/setVote"`. Prende una serie di parametri che rappresentano i voti dei giocatori. Imposta il voto di ogni giocatore e salva il giocatore nel database. Infine, reindirizza alla pagina "modifyLeague".
- `evaluateDay`: Questo metodo gestisce le richieste POST a `"/EvaluateDay"`. Prende un parametro "newStartDate" che rappresenta la nuova data della prossima giornata. Aggiorna la data, calcola i punti di ogni squadra in base ai voti dei giocatori nelle varie formazioni e salva la lega nel database. Infine, reindirizza alla pagina "modifyLeague".

3.3 DAOSYSTEM

Il package `daosystem` gestisce le connessioni e le operazioni sul database utilizzando il pattern DAO (Data Access Object). Questo pattern fornisce un'interfaccia astratta che consente di manipolare i dati dell'applicazione, indipendentemente dal sistema di persistenza dei dati sottostante. In questo caso, il sistema di persistenza dei dati è un database MySQL. MySQL è un sistema di gestione di database relazionale (RDBMS) open source che utilizza SQL (Structured Query Language) per l'accesso ai dati. Il package `daosystem` utilizza anche il framework Spring Boot JPA (Java Persistence API) per semplificare l'accesso ai dati nel database. JPA è una specifica standard di Java che astrae i dettagli dell'accesso ai dati, permettendo agli sviluppatori di interagire con il database in modo più intuitivo e orientato agli oggetti.

3.3.1 *UserDao*

La classe `UserDao` si interfaccia con il database per la gestione della tabella degli utenti, separando l'oggetto `User` Java dall'entità corrispondente salvata nel database. Questa classe contiene la JPA Repository `UserRepo` e si occupa di inizializzarla. `UserDao` avvia il collegamento che mette in comunicazione il programma e il database, attraverso il passaggio del parametro `ApplicationContext` ottenuto una volta avviata

l'applicazione. I principali metodi che vengono forniti da UserDao sono i seguenti:

- `'save(User user)'`: salva, attraverso la repository `'UserRepo'`, l'oggetto `'User'` nel database e notifica il successo dell'operazione.
- `'delete(User user)'`: cancella, attraverso la repository `'UserRepo'`, l'oggetto `'User'` nel database e notifica il successo dell'operazione.
- `'getAll()'`: fornisce un insieme contenente tutti gli oggetti `'User'` attualmente esistenti nel database.
- `'get(Long id)'`: restituisce un'occorrenza di un oggetto `'User'` data la sua chiave primaria.
- `'getByName(String name)'`: restituisce un'occorrenza di un oggetto `'User'` dato il suo nome.
- `'update(User user, String[] params)'`: aggiorna, attraverso la repository `'UserRepo'`, l'oggetto `'User'` nel database e notifica il successo dell'operazione. </pre>

3.3.2 TeamDao

La classe TeamDao si interfaccia con il database per la gestione della tabella delle squadre, separando l'oggetto Team Java dall'entità corrispondente salvata nel database. Questa classe contiene la JPA Repository TeamRepo e si occupa di inizializzarla. TeamDao avvia il collegamento che mette in comunicazione il programma e il database, attraverso il passaggio del parametro ApplicationContext ottenuto una volta avviata l'applicazione. I principali metodi che vengono forniti da TeamDao sono i seguenti:

- `'save(Team team)'`: salva, attraverso la repository `'TeamRepo'`, l'oggetto `'Team'` nel database e notifica il successo dell'operazione.
- `'delete(Team team)'`: cancella, attraverso la repository `'TeamRepo'`, l'oggetto `'Team'` nel database e notifica il successo dell'operazione.
- `'getAll()'`: fornisce un insieme contenente tutti gli oggetti `'Team'` attualmente esistenti nel database.
- `'get(Long id)'`: restituisce un'occorrenza di un oggetto `'Team'` data la sua chiave primaria.

- `'getPlayers(Long Id)'`: restituisce un insieme di oggetti `'Player'` che appartengono alla squadra con l'ID specificato.
- `'getByUserId(Long userId)'`: restituisce un oggetto `'Team'` che appartiene all'utente con l'ID specificato.
- `'getTeamByOwnerAndLeague(User owner, League league)'`: restituisce un oggetto `'Team'` che appartiene all'utente specificato e alla lega specificata.
- `'getPoolByTeamId(Long teamId)'`: restituisce un insieme di oggetti `'Player'` che appartengono alla pool della squadra con l'ID specificato.

3.3.3 *PlayerDao*

La classe `PlayerDao` si interfaccia con il database per la gestione della tabella dei giocatori, separando l'oggetto `Player` Java dall'entità corrispondente salvata nel database. Questa classe contiene la JPA Repository `PlayerRepo` e si occupa di inizializzarla. `PlayerDao` avvia il collegamento che mette in comunicazione il programma e il database, attraverso il passaggio del parametro `ApplicationContext` ottenuto una volta avviata l'applicazione. I principali metodi che vengono forniti da `PlayerDao` sono i seguenti:

- `'save(Player player)'`: salva, attraverso la repository `'PlayerRepo'`, l'oggetto `'Player'` nel database e notifica il successo dell'operazione.
- `'delete(Player player)'`: cancella, attraverso la repository `'PlayerRepo'`, l'oggetto `'Player'` nel database e notifica il successo dell'operazione.
- `'getAll()'`: fornisce un insieme contenente tutti gli oggetti `'Player'` attualmente esistenti nel database.
- `'get(Long id)'`: restituisce un'occorrenza di un oggetto `'Player'` data la sua chiave primaria.
- `'getByName(String name)'`: restituisce un'occorrenza di un oggetto `'Player'` dato il suo nome.
- `'getPlayersOfLeague(Long id)'`: restituisce un insieme di oggetti `'Player'` che appartengono alla lega con l'ID specificato.

3.3.4 *LeagueDao*

La classe *LeagueDao* si interfaccia con il database per la gestione della tabella delle leghe, separando l'oggetto *League* Java dall'entità corrispondente salvata nel database. Questa classe contiene la JPA Repository *LeagueRepo* e si occupa di inicializzarla. *LeagueDao* avvia il collegamento che mette in comunicazione il programma e il database, attraverso il passaggio del parametro *ApplicationContext* ottenuto una volta avviata l'applicazione. I principali metodi che vengono forniti da *LeagueDao* sono i seguenti:

- *'save(League league)'*: salva, attraverso la repository *'LeagueRepo'*, l'oggetto *'League'* nel database e notifica il successo dell'operazione.
- *'delete(League league)'*: cancella, attraverso la repository *'LeagueRepo'*, l'oggetto *'League'* nel database e notifica il successo dell'operazione.
- *'getAll()'*: fornisce un insieme contenente tutti gli oggetti *'League'* attualmente esistenti nel database.
- *'get(Long id)'*: restituisce un'occorrenza di un oggetto *'League'* data la sua chiave primaria.
- *'getByName(String name)'*: restituisce un insieme di oggetti *'League'* dato il loro nome.
- *'getPlayers(Long id)'*: restituisce un insieme di oggetti *'Player'* che appartengono alla lega con l'ID specificato.
- *'getTeams(Long id)'*: restituisce un insieme di oggetti *'Team'* che appartengono alla lega con l'ID specificato.
- *'getUserAttendedLeagues(Long id)'*: restituisce un insieme di oggetti *'League'* a cui l'utente con l'ID specificato ha partecipato.

3.4 REPOSITORIES

Il package *repositories* è un componente fondamentale nell'architettura di un'applicazione Spring Boot. Questo package contiene le interfacce che estendono *CrudRepository* o *JpaRepository*, che forniscono metodi CRUD (Create, Read, Update, Delete) per l'accesso ai dati. Ogni interfaccia nel package *repositories* corrisponde a un'entità nel database. Ad esempio, *UserRepo* corrisponde all'entità *User*, *LeagueRepo* corrisponde all'entità

League, e così via. Queste interfacce definiscono metodi per eseguire operazioni sul database relative alle loro entità corrispondenti. Questi metodi includono operazioni standard come `save`, `delete`, `findAll`, e `findById`. Inoltre, è possibile definire metodi personalizzati per eseguire query specifiche sul database. Ad esempio, `UserRepo` ha un metodo `getByName` che restituisce un `User` in base al suo nome. Spring Data JPA implementa automaticamente queste interfacce quando l'applicazione viene eseguita. Ciò significa che non è necessario scrivere l'implementazione di questi metodi; Spring Data JPA genera l'implementazione basandosi sul nome del metodo.

3.5 RESOURCES

Il folder `resources` è una directory speciale in un progetto Spring Boot che viene utilizzata per contenere file non-Java, come configurazioni, file di proprietà, file HTML, CSS, JavaScript, e immagini. Questi file sono utilizzati per vari scopi come la configurazione dell'applicazione, l'interfaccia utente, ecc.

3.5.1 *static*

Il package `static` è una directory speciale all'interno di `resources` che viene utilizzata per servire contenuti statici come CSS, JavaScript, immagini e altri file multimediali. Spring Boot configura automaticamente Spring MVC per servire i file statici da questa directory. Ad esempio, il file `style.css` all'interno del package `static/css` contiene le definizioni di stile CSS utilizzate per formattare l'interfaccia utente dell'applicazione.

3.5.2 *templates*

Il package `templates` è un'altra directory speciale all'interno di `resources` che viene utilizzata per contenere i file di template. Questi file di template sono utilizzati per generare dinamicamente il codice HTML che viene inviato al browser. Spring Boot utilizza un motore di templating, come Thymeleaf o FreeMarker, per processare questi file di template. Ad esempio, i file `.html` all'interno del package `templates` contengono il codice HTML che definisce la struttura delle pagine web dell'applicazione.

TEST (JUNIT)

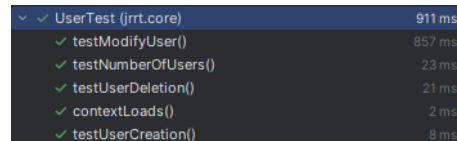
JUnit è un framework di testing per il linguaggio di programmazione Java. È utilizzato per scrivere e eseguire test automatizzati per verificare che il codice si comporti come previsto. Ogni test è indipendente dagli altri. Prima di ogni test, il database è ripulito e preparato per il test successivo. Questo garantisce che i test non influenzino l'uno l'altro e che ogni test sia ripetibile.

4.1 USERTEST

Nel contesto del UserTest fornito, JUnit viene utilizzato per testare il comportamento della classe UserDao. Ogni metodo annotato con `@Test` è un caso di test indipendente. Ecco una breve descrizione di come funzionano i test in UserTest:

- `contextLoads()`: Questo test verifica che l'istanza di UserDao sia stata correttamente iniettata nel test. Se UserDao è null, il test fallisce.
- `testNumberOfUsers()`: Questo test verifica che il metodo `getAll()` di UserDao restituisca il numero corretto di utenti. Crea un certo numero di utenti, li salva nel database, quindi verifica che il numero di utenti restituiti da `getAll()` sia uguale al numero di utenti creati.
- `testUserCreation()`: Questo test verifica che un utente possa essere correttamente salvato nel database. Crea un nuovo utente, lo salva nel database, quindi cerca l'utente nel database per nome e verifica che i dettagli dell'utente corrispondano.
- `testUserDeletion()`: Questo test verifica che un utente possa essere correttamente eliminato dal database. Crea un nuovo utente, lo salva nel database, lo elimina, quindi cerca l'utente nel database per nome e verifica che l'utente non esista più.

- `testModifyUser()`: Questo test verifica che i dettagli di un utente possano essere correttamente modificati. Crea un nuovo utente, lo salva nel database, modifica i dettagli dell'utente, lo salva di nuovo, quindi cerca l'utente nel database per nome e verifica che i dettagli dell'utente siano stati aggiornati.



✓ UserTest (jrrt.core)	911 ms
✓ testModifyUser()	857 ms
✓ testNumberOfUsers()	23 ms
✓ testUserDeletion()	21 ms
✓ contextLoads()	2 ms
✓ testUserCreation()	8 ms

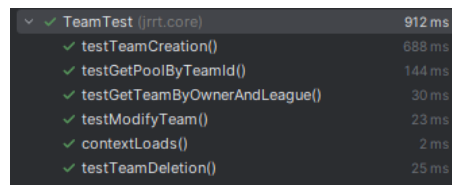
Figure 13: User test eseguiti con esito positivo

4.2 TEAMTEST

Nel contesto del TeamTest fornito, JUnit viene utilizzato per testare il comportamento della classe TeamDao. Ogni metodo annotato con `@Test` è un caso di test indipendente. Ecco una breve descrizione di come funzionano i test in TeamTest:

- `contextLoads()`: Questo test verifica che l'istanza di TeamDao, UserDao, LeagueDao e PlayerDao siano state correttamente iniettate nel test. Se una di queste è null, il test fallisce.
- `testTeamCreation()`: Questo test verifica che un team possa essere correttamente salvato nel database. Crea un nuovo utente, una nuova lega e un nuovo team, li salva nel database, quindi cerca il team nel database per ID e verifica che i dettagli del team corrispondano.
- `testTeamDeletion()`: Questo test verifica che un team possa essere correttamente eliminato dal database. Crea un nuovo utente, una nuova lega e un nuovo team, li salva nel database, elimina il team, quindi cerca il team nel database per ID e verifica che il team non esista più.
- `testModifyTeam()`: Questo test verifica che i dettagli di un team possano essere correttamente modificati. Crea un nuovo utente, una nuova lega e un nuovo team, li salva nel database, modifica i dettagli del team, lo salva di nuovo, quindi cerca il team nel database per ID e verifica che i dettagli del team siano stati aggiornati.

- `testGetTeamByOwnerAndLeague()`: Questo test verifica che il metodo `getTeamByOwnerAndLeague` di `TeamDao` restituisca il team corretto. Crea un nuovo utente, una nuova lega e un nuovo team, li salva nel database, quindi chiama il metodo `getTeamByOwnerAndLeague` e verifica che il team restituito corrisponda al team creato.
- `testGetPoolByTeamId()`: Questo test verifica che il metodo `getPoolByTeamId` di `TeamDao` restituisca l'insieme corretto di giocatori. Crea un nuovo utente, una nuova lega, un nuovo team e due nuovi giocatori, li salva nel database, aggiunge i giocatori al team, quindi chiama il metodo `getPoolByTeamId` e verifica che l'insieme di giocatori restituito corrisponda ai giocatori aggiunti al team.



✓ TeamTest (jrrt.core)	912 ms
✓ testTeamCreation()	688 ms
✓ testGetPoolByTeamId()	144 ms
✓ testGetTeamByOwnerAndLeague()	30 ms
✓ testModifyTeam()	23 ms
✓ contextLoads()	2 ms
✓ testTeamDeletion()	25 ms

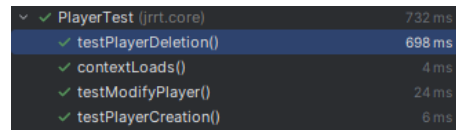
Figure 14: Team test eseguiti con esito positivo

4.3 PLAYERTEST

Nel contesto del `PlayerTest` fornito, JUnit viene utilizzato per testare il comportamento della classe `PlayerDao`. Ogni metodo annotato con `@Test` è un caso di test indipendente. Ecco una breve descrizione di come funzionano i test in `PlayerTest`:

- `contextLoads()`: Questo test verifica che l'istanza di `PlayerDao` sia stata correttamente iniettata nel test. Se `PlayerDao` è null, il test fallisce.
- `testPlayerCreation()`: Questo test verifica che un giocatore possa essere correttamente salvato nel database. Crea un nuovo giocatore, lo salva nel database, quindi cerca il giocatore nel database per ID e verifica che i dettagli del giocatore corrispondano.
- `testPlayerDeletion()`: Questo test verifica che un giocatore possa essere correttamente eliminato dal database. Crea un nuovo giocatore, lo salva nel database, lo elimina, quindi cerca il giocatore nel database per ID e verifica che il giocatore non esista più.

- `testModifyPlayer()`: Questo test verifica che i dettagli di un giocatore possano essere correttamente modificati. Crea un nuovo giocatore, lo salva nel database, modifica i dettagli del giocatore, lo salva di nuovo, quindi cerca il giocatore nel database per ID e verifica che i dettagli del giocatore siano stati aggiornati.



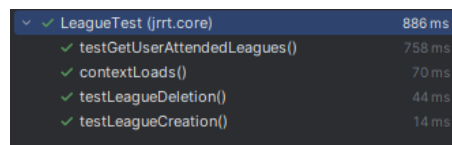
✓ PlayerTest (jrrt.core)	732 ms
✓ testPlayerDeletion()	698 ms
✓ contextLoads()	4 ms
✓ testModifyPlayer()	24 ms
✓ testPlayerCreation()	6 ms

Figure 15: Player test eseguiti con esito positivo

4.4 LEAGUETEST

Nel contesto del `LeagueTest` fornito, JUnit viene utilizzato per testare il comportamento della classe `LeagueDao`. Ogni metodo annotato con `@Test` è un caso di test indipendente. Ecco una breve descrizione di come funzionano i test in `LeagueTest`:

- `contextLoads()`: Questo test verifica che l'istanza di `LeagueDao`, `UserDao` e `PlayerDao` siano state correttamente iniettate nel test. Se una di queste è null, il test fallisce.
- `testLeagueCreation()`: Questo test verifica che una lega possa essere correttamente salvata nel database. Crea un nuovo utente e una nuova lega, li salva nel database, quindi cerca la lega nel database per ID e verifica che i dettagli della lega corrispondano.
- `testLeagueDeletion()`: Questo test verifica che una lega possa essere correttamente eliminata dal database. Crea un nuovo utente e una nuova lega, li salva nel database, elimina la lega, quindi cerca la lega nel database per ID e verifica che la lega non esista più.
- `testGetUserAttendedLeagues()`: Questo test verifica che il metodo `getUserAttendedLeagues` di `LeagueDao` restituisca le leghe corrette. Crea un nuovo utente e due nuove leghe, li salva nel database, quindi chiama il metodo `getUserAttendedLeagues` e verifica che le leghe restituite corrispondano alle leghe create.

A screenshot of a JUnit test runner interface showing the results of a test suite. The suite is 'LeagueTest (jrrt.core)' and it passed. Below the suite name, four individual tests are listed, each with a green checkmark indicating success and a duration in milliseconds. The tests are 'testGetUserAttendedLeagues()', 'contextLoads()', 'testLeagueDeletion()', and 'testLeagueCreation()'.

✓ LeagueTest (jrrt.core)	886 ms
✓ testGetUserAttendedLeagues()	758 ms
✓ contextLoads()	70 ms
✓ testLeagueDeletion()	44 ms
✓ testLeagueCreation()	14 ms

Figure 16: League test eseguiti con esito positivo