TRACK TUNE

SW Engineering Documentation

Autori:
Mattia Rebonato
Davide Fozzato

Professore:
Prof. Carlo Combi

A.A. 2024-2025

Indice

1 Introduzione

1.1 Prefazione

Il presente documento definisce in modo esaustivo i requisiti, l'architettura e le specifiche di progettazione del software *TrackTune*, una piattaforma applicativa dedicata alla gestione, condivisione e analisi collaborativa di contenuti musicali.

La documentazione è rivolta a sviluppatori, tester, amministratori di sistema e a tutti gli stakeholder coinvolti nella realizzazione, manutenzione e gestione del progetto.

1.2 Panoramica

La piattaforma si rivolge a musicisti, compositori, insegnanti, studenti e appassionati di musica, offrendo un ambiente digitale comodo per l'archiviazione, la collaborazione e l'accesso a materiale musicale di vario genere, tra cui spartiti, testi, accordi, file audio/video e link esterni (ad esempio YouTube).
Essa si propone non solo come archivio digitale, ma anche come spazio per l'approfondimento culturale, lo scambio di interpretazioni e feedback tra utenti, favorendo la condivisione di esperienze musicali, commenti critici e annotazioni esecutive su brani e contenuti attinenti alla musica.

2 Analisi dei requisiti

2.1 Descrizione del sistema

Si vuole progettare un sistema software per gestire la collezione e la condivisione di spartiti, testi, accordi, MIDI, MP3, video, link e molto altro, relativamente a brani musicali di diversa tipologia/genere.
Gli utenti, previa autorizzazione dell'amministratore, devono poter caricare, scaricare, commentare e interagire con i vari contenuti. Per ogni risorsa il sistema deve consentire di specificare:

- autori
- genere/generi
- strumenti musicali utilizzati

Inoltre, per le risorse multimediali (audio, video, ecc.) il sistema deve consentire di specificare il luogo e la data di registrazione.
Ogni utente può aggiungere note di testo libero su segmenti specifici di esecuzione (MP3, MP4 o video YouTube). Un segmento è definito dal momento di inizio e fine, e per ciascuno è possibile includere dettagli come assoli, esecutori, strumenti, ritmi e altre caratteristiche. I commenti possono essere arricchiti con ulteriori risposte, consentendo una profondità illimitata di discussioni relativamente a quel contenuto.

Un brano può essere inserito anche da un autore o un interprete. Deve essere possibile quindi distinguere il proprio ruolo all'interno del brano e, nel caso di interpreti, gli strumenti utilizzati. I commenti sulle modalità esecutive devono apparire in maniera rilevante rispetto a quelli di altri utenti.

I video YouTube possono essere visualizzati direttamente nel software o nel browser, ma tutti i commenti relativi sono gestiti dal sistema.

Gli utenti possono aggiungere commenti su spartiti, testi e accordi, relativi alle modalità esecutive (come strumenti, ritmo, intensità, ecc.), sia su specifiche parti del brano che sull'intero brano.

Un utente può richiedere la registrazione alla piattaforma tramite un form, che l'amministratore valuterà e risponderà a seconda dell'esito. L'amministratore gestisce gli utenti, ha il compito di rimuovere coloro che pubblicano contenuti non pertinenti e di moderare i commenti. Inoltre, valuta le richieste di registrazione, il cui esito verrà comunicato all'utente interessato.

Il sistema deve permettere la ricerca delle risorse tramite i seguenti filtri:

- genere
- titolo del brano
- autore/i

Ogni risorsa per cui un utente ha lasciato un commento deve essere direttamente accessibile.
Gli strumenti musicali, i generi, i titoli dei brani e i nomi degli autori vanno gestiti attraverso dizionari aggiornabili, da usare anche per le opportune ricerche.

2.2 Requisiti funzionali

2.2.1 Gestione degli utenti

- Registrazione tramite form dedicato;
- Valutazione da parte dell'amministratore delle richieste di registrazione e relativa comunicazione dell'esito;
- Profilazione (distinzione tra utente/amministratore);
- Autorizzazione al caricamento dei file per utente;

2.2.2 Gestione dei contenuti

- Aggiunta, modifica e rimozione di contenuti multimediali
  - La rimozione può essere eseguita solo dall'amministratore o dall'utente che ha caricato il contenuto
- Visualizzazione delle risorse tramite software di terze parti o tramite l'applicativo stesso.
- Moderazione degli utenti
  - L'amministratore può rimuovere utenti;
  - L'amministratore può rimuovere commenti/risorse inappropriate.
- Gestione dei dizionari aggiornabili (solo amministatore) relativi a:
  - Autori;
  - Generi;
  - Strumenti musicali;

2.2.3 Gestione dei commenti

- Possibilità di commentare brani;
- Possibilità di rispondere ai commenti;
- Possibilità di commentare su specifici segmenti di esecuzione;
- Visualizzazione differenziata dei commenti per autori/esecutori del brano.

2.3 Casi d'uso

L'applicazione sarà disponibile esclusivamente agli utenti abilitati dall'amministratore, il quale è anch'esso un utente ma con privilegi di amministrazione sui comportamenti del sistema. L'unica operazione che può effettuare un utente non registrato è la richiesta di creazione di un account.

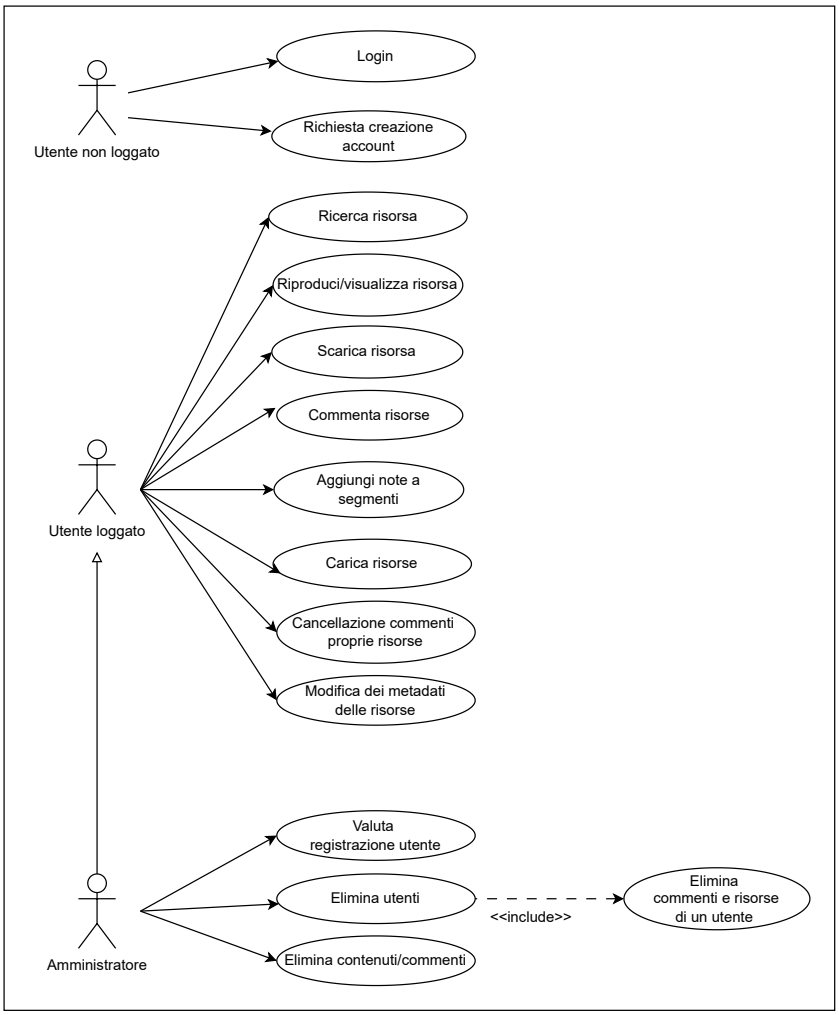

Figura 1: Use Cases

2.3.1 Utente non loggato

L'utente non autenticato può effettuare esclusivamente non operazioni:

1. Richiedere un account ad un amministratore
2. Effettuare l'accesso tramite un form di autenticazione

Caso d'uso: Login	
Id	UC0
Attori	Utente non loggato
Precondizioni	L'utente non è autenticato sulla piattaforma.
Sequenza degli eventi	1. L'utente compila i campi per effettuare l'autenticazione 2. Il sistema verifica i dati e gestisce le autorizzazioni dell'utente
Post condizioni	Utente autenticato e correttamente gestito

Tabella 1: Descrizione caso d'uso *Login*

Caso d'uso: Richiesta creazione account	
Id	UC1
Attori	Utente non loggato
Precondizioni	L'utente non è autenticato sulla piattaforma. L'utente accede alla sezione di registrazione.
Sequenza degli eventi	1. Vengono compilati i dati relativi alla richiesta 2. La richiesta viene inviata 3. Il sistema registra la richiesta e la mette in stato di approvazione 4. L'account viene salvato come account provvisorio 5. Il sistema notifica l'amministratore della nuova richiesta
Sequenza alternativa	Utente già esistente che comporta un messaggio di errore
Post condizioni	La richiesta di creazione dell'account deve essere approvata da un amministratore

Tabella 2: Descrizione caso d'uso *Richiesta creazione account*

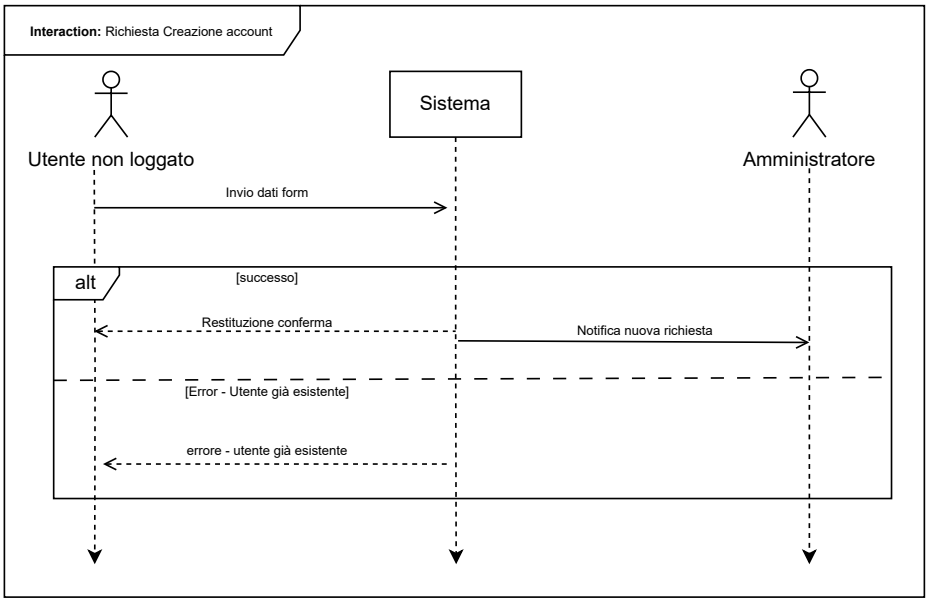

Figura 2: Sequence diagram del caso d'uso *Richiesta creazione account*

2.3.2 Utente loggato

Caso d'uso: Ricerca risorsa	
Id	UC2
Attori	Utente loggato
Precondizioni	Utente autenticato
Sequenza degli eventi	1. Il sistema invia i criteri di filtro disponibili 2. L'utente decide il filtro di ricerca 3. L'utente inserisce una parola chiave per ricercare una risorsa 4. Gli vengono restituite le risorse data quella ricerca
Sequenza alternativa 1	Errore nella ricerca: messaggio di errore
Post condizioni	L'utente ha visualizzato la risorsa correttamente

Tabella 3: Descrizione caso d'uso *Ricerca risorsa*

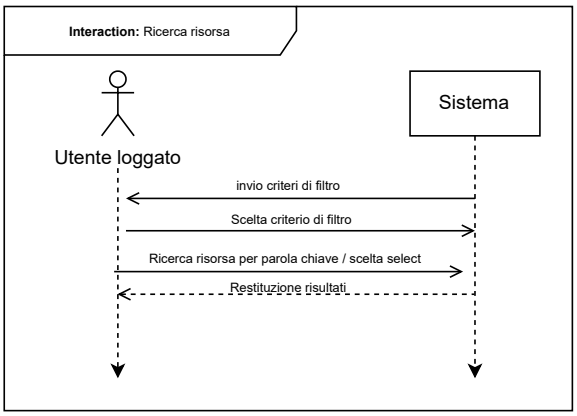

Figura 3: Sequence diagram del caso d'uso *Ricerca risorsa*

Caso d'uso: Riproduci/Visualizza risorsa	
Id	UC3
Attori	User Logged
Precondizioni	Utente autenticato L'utente ha effettuato una ricerca o ne può selezionare una
Sequenza degli eventi	1. L'utente seleziona una risorsa dal risultato delle ricerca 2. Il sistema riconosce il tipo di risorsa 3. Se multimediale, lo riproduce 4. Se testuale, lo visualizza
Sequenza alternativa 1	Errore nella riproduzione: messaggio di errore
Post condizioni	L'utente ha visualizzato la risorsa

Tabella 4: Descrizione caso d'uso *Riproduci/Visualizza risorsa*

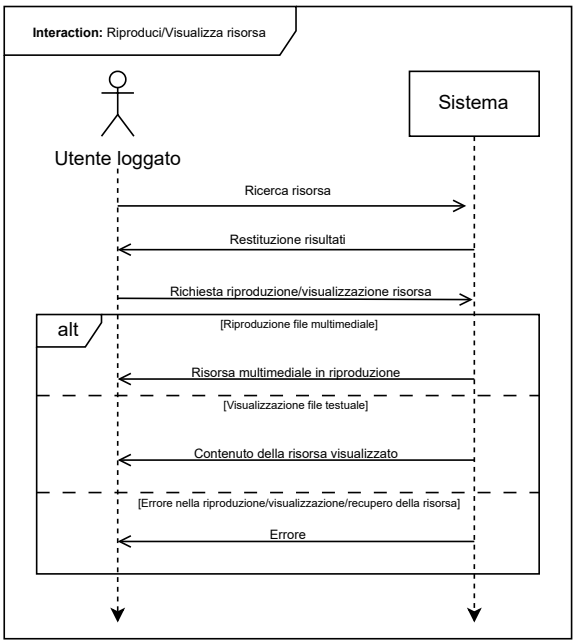

Figura 4: Sequence diagram del caso d'uso *Riproduci/Visualizza risorsa*

Caso d'uso: Scarica risorsa	
Id	UC4
Attori	User Logged
Precondizioni	Utente autenticato L'utente ha effettuato una ricerca o ne può selezionare una
Sequenza degli eventi	1. L'utente seleziona una risorsa dal risultato delle ricerca 2. L'utente richiede il download della risorsa
Sequenza alternativa 1	Errore nel download: messaggio di errore
Post condizioni	L'utente ha scaricato la risorsa correttamente

Tabella 5: Descrizione caso d'uso *Scarica risorsa*

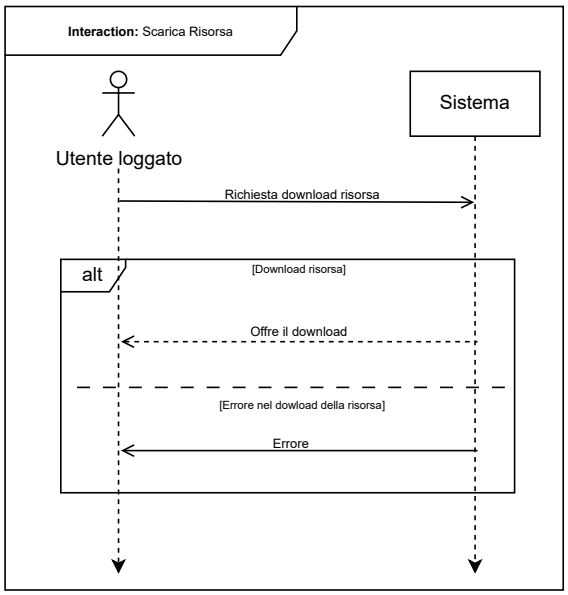

Figura 5: Sequence diagram del caso d'uso *Scarica risorsa*

Caso d'uso: Commenta risorse	
Id	UC5
Attori	User Logged
Precondizioni	Utente autenticato L'utente può visualizzare una risorsa
Sequenza degli eventi	1. L'utente seleziona una risorsa dal risultato delle ricerca 2. L'utente può scrivere un commento 3. Il commento viene salvato dal sistema
Sequenza alternativa 1	L'utente può rispondere ad un commento
Post condizioni	L'utente ha commentato la risorsa correttamente

Tabella 6: Descrizione caso d'uso *Commenta risorse*

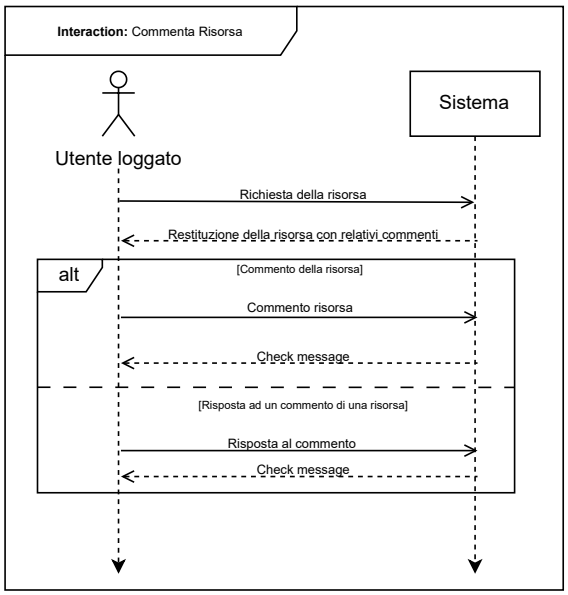

Figura 6: Sequence diagram del caso d'uso *Commenta risorse*

Caso d'uso: Aggiungi note a segmenti	
Id	UC6
Attori	User Logged
Precondizioni	Utente autenticato L'utente ha ricercato/può visualizzare una risorsa
Sequenza degli eventi	1. L'utente seleziona un intervallo di tempo per inserire una nota/commento 2. Scrive la /commento desiderata 3. Invia la nota/commento al sistema 4. Il sistema registra la nota/commento
Post condizioni	L'utente ha scritto una nota/commento su un segmento correttamente

Tabella 7: Descrizione caso d'uso *Aggiungi note a segmenti*

Caso d'uso: Carica Risorse	
Id	UC7
Attori	User Logged
Precondizioni	Utente autenticato
Sequenza degli eventi	1. L'utente visualizza la schermata di upload dei file 2. Seleziona la risorsa da caricare 3. Inserisce i dati (file + metadati obbligatori) 4. Invia i dati al sistema 5. Il sistema li memorizza
Sequenza alternativa 1	Errore di sistema: messaggio di errore
Sequenza alternativa 2	Errore nell'inserimento dei dati: messaggio di errore
Post condizioni	L'utente ha caricato la risorsa correttamente

Tabella 8: Descrizione caso d'uso *Carica Risorse*

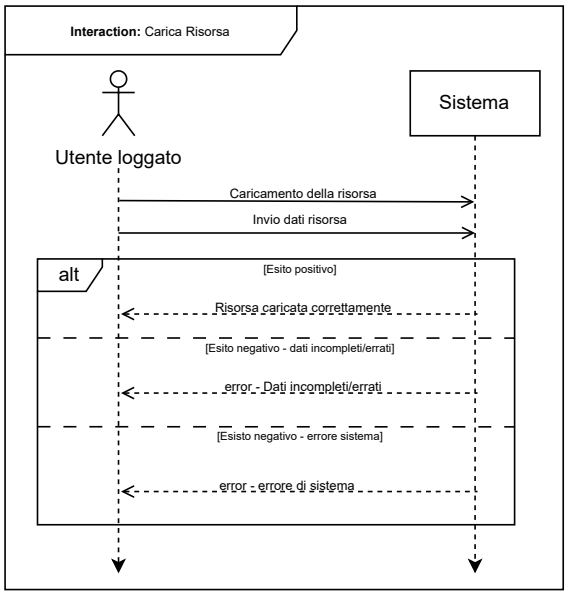

Figura 7: Sequence diagram del caso d'uso *Carica Risorse*

Caso d'uso: Cancellazione commenti delle proprie risorse	
Id	UC8
Attori	User Logged
Precondizioni	Utente autenticato e autorizzato L'utente sta visualizzando una propria risorsa caricata
Sequenza degli eventi	1. L'utente desidera eliminare un qualsiasi commento della risorsa 2. Elimina il commento 3. Il sistema controlla che abbia i permessi necessari 4. In caso positivo effettua la rimozione 5. Vengono rimossi tutte le risposte a quel commento
Sequenza alternativa 1	Errore Autorizzazioni: messaggio di errore
Post condizioni	L'utente ha eliminato il commento correttamente

Tabella 9: Descrizione caso d'uso *Cancellazione commenti delle proprie risorse*

Caso d'uso: Modifica dei metadati delle risorse	
Id	UC9
Attori	User Logged Authorized
Precondizioni	Utente autenticato L'utente è il proprietario di quella risorsa L'utente sta visualizzando i dati di una risorsa
Sequenza degli eventi	1. L'utente desidera modificare un qualsiasi metadato della risorsa 2. Modifica i metadati 3. Invia le modifiche al sistema 4. Il sistema effettua la modifica
Post condizioni	L'utente ha modificato correttamente i metadati di una risorsa

Tabella 10: Descrizione caso d'uso *Modifica dei metadati delle risorse*

2.3.3 Amministratore

Caso d'uso: Valuta registrazione utente	
Id	UC10
Attori	Administrator
Precondizioni	L'amministratore accede alla sezione di gestione delle registrazioni utente in attesa di valutazione
Sequenza degli eventi	1. L'amministratore visualizza i dettagli 2. L'amministratore approva/rifiuta la richiesta 3. Il sistema elimina la richiesta e in caso di approvazione crea l'account 4. Il sistema notifica all'utente l'esito
Post condizioni	L'utente è registrato e abilitato all'accesso al sistema (se approvato) o la sua richiesta di registrazione viene rifiutata

Tabella 11: Descrizione caso d'uso *Valuta registrazione utente*

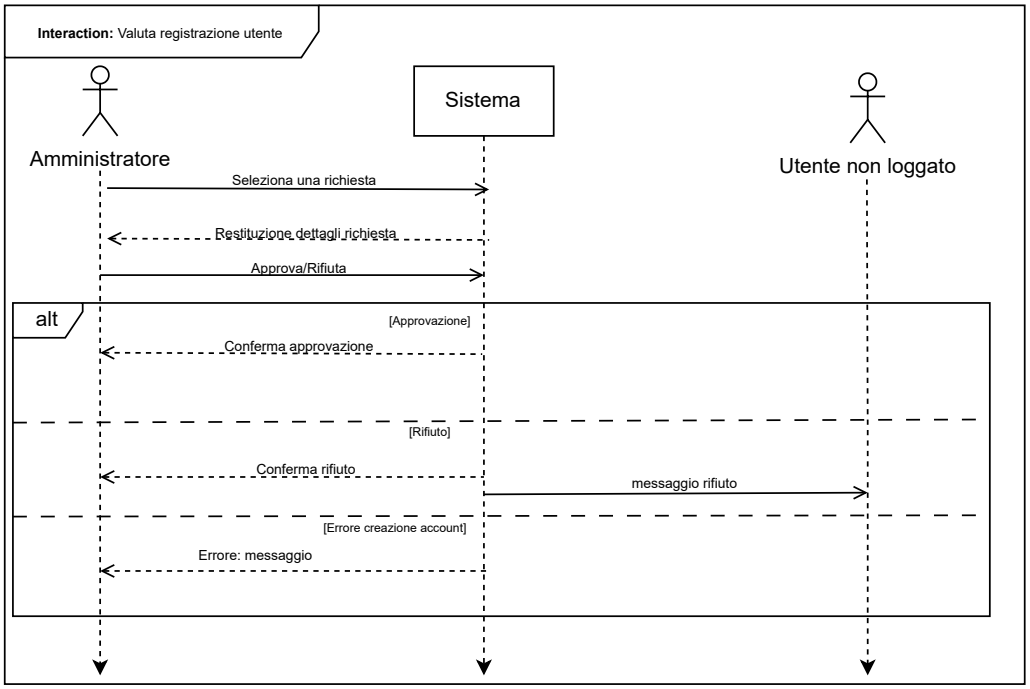

Figura 8: Sequence diagram del caso d'uso *Valuta registrazione utente*

Caso d'uso: Eliminazione utente	
Id	UC11
Attori	Administrator
Precondizioni	L'amministratore accede alla sezione di gestione degli utenti
Sequenza degli eventi	1. L'amministratore seleziona l'utente che desidera eliminare 2. L'amministratore elimina l'utente 3. Rimuove l'utente
Post condizioni	L'utente è stato cancellato correttamente

Tabella 12: Descrizione caso d'uso *Eliminazione utente*

Caso d'uso: Elimina commenti e risorse di un utente	
Id	UC12
Attori	Administrator
Precondizioni	Caso d'uso *Eliminazine Utente* (Tabella 12)
Sequenza degli eventi	1. L'amministratore comunica al server un utente 2. Le risorse e commenti di questo utente vengono eliminate
Post condizioni	Le risorse/commenti sono stati cancellati correttamente

Tabella 13: Descrizione caso d'uso *Elimina commenti e risorse di un utente*

Caso d'uso: Elimina contenuti/commenti	
Id	UC13
Attori	Administrator
Precondizioni	Visualizzazione di una risorsa/commento
Sequenza degli eventi	1. L'amministratore visualizza una risorsa 2. L'amministratore decide di eliminare la risorsa/commento 3. Invia la richiesta al sistema 4. Il sistema elimina la risorsa/commento
Post condizioni	La risorsa/commento è stata eliminata correttamente

Tabella 14: Descrizione caso d'uso *Elimina contenuti/commenti*

2.4 Glossario dei termini

GLOSSARIO DEI TERMINI	
TERMINE	**DEFINIZIONE**
Contenuto musicale / Risorsa multimediale	Qualsiasi tipo di file audio, video o immagine caricato e reso disponibile sulla piattaforma. Può includere brani musicali, podcast, lezioni, video, ecc.
Autore / Esecutore / Interprete di un brano	L'individuo o il gruppo che ha contribuito in modo significativo alla creazione, composizione, esecuzione o interpretazione originale di un'opera musicale.
Segmento di un brano	Un sottoinsieme specifico di un contenuto multimediale (es. un brano musicale o un video), definito da un intervallo di tempo di inizio e fine, permettendo di focalizzare l'attenzione su una parte specifica.
Traccia	Una traccia può contenere più risorse relative a quel contesto, ad esempio il relativo file PDF dello spartito + un'eventuale risorsa multimediale (mp4/mp3)
Brano	Un'opera musicale originale completa e autonoma, che può essere parte di un album o una singola traccia, caratterizzata da una composizione unica.
Metadato	Un insieme di dati che descrivono e forniscono informazioni su altri dati (una risorsa). Per un brano, i metadati possono includere titolo, autore, genere, durata, data di pubblicazione, ecc.
Utente autenticato / Loggato	Un utente che ha completato con successo il processo di login fornendo credenziali valide ed è riconosciuto dal sistema, avendo accesso a funzionalità riservate.
Amministratore	Un utente con privilegi speciali all'interno della piattaforma, in grado di gestire utenti, contenuti, approvare richieste e monitorare le operazioni del sistema.
Richiesta di creazione account	La procedura iniziata da un utente non autenticato per ottenere un nuovo account sulla piattaforma, soggetta ad approvazione da parte di un amministratore.
Commento	Un testo o una nota aggiunta da un utente autenticato a una risorsa (o a un altro commento) per esprimere un'opinione, fornire feedback o interagire con altri utenti.
Nota a segmento	Un tipo specifico di commento o annotazione testuale associata a un intervallo di tempo preciso all'interno di una risorsa multimediale, permettendo di evidenziare o commentare una parte specifica del contenuto.

Tabella 15: Glossario dei Termini

2.5 Activity Diagrams

Questa sezione illustra i principali flussi di lavoro del sistema attraverso gli activity diagrams.

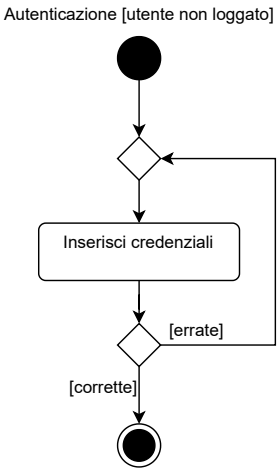

Figura 9: Activity Diagrams del caso d'uso *Login*

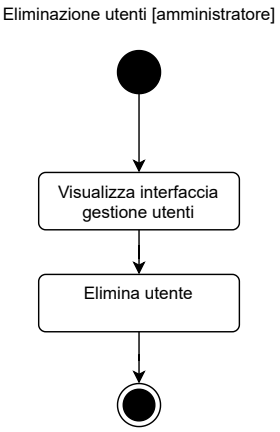

Figura 10: Activity Diagrams del caso d'uso *Elimina utenti*

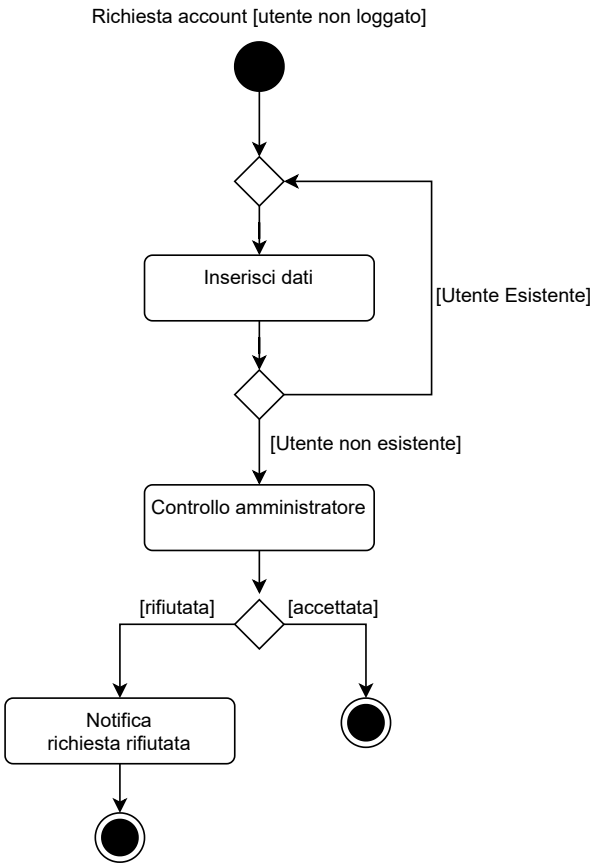

Figura 11: Activity Diagrams del caso d'uso *Richiesta creazione account*

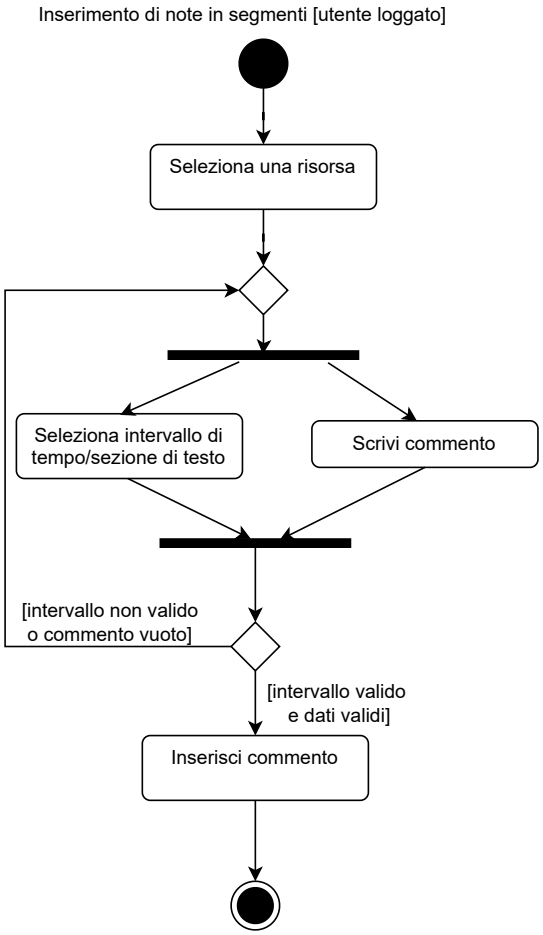

Figura 12: Activity Diagrams del caso d'uso *Aggiungi note a segmenti*

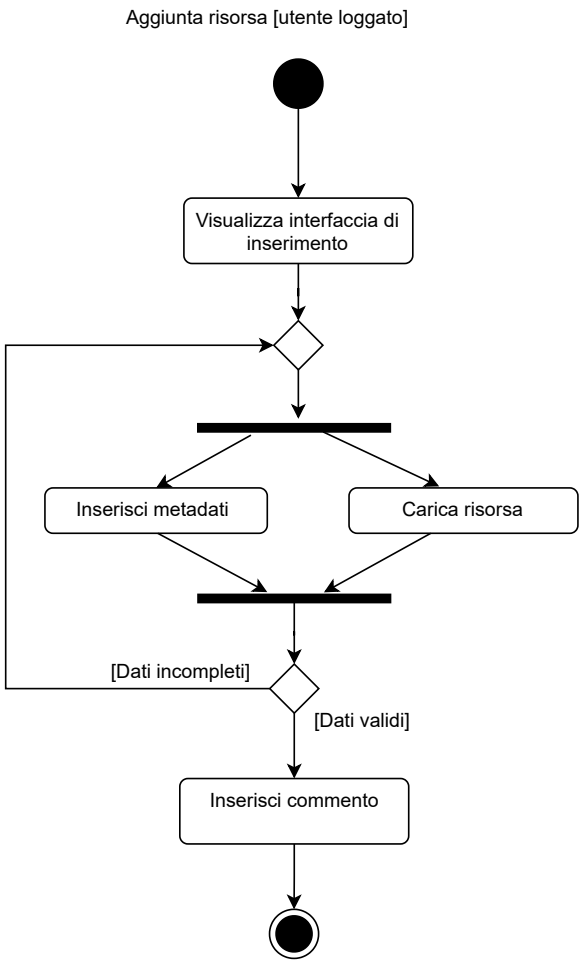

Figura 13: Activity Diagrams del caso d'uso *Carica risorse*

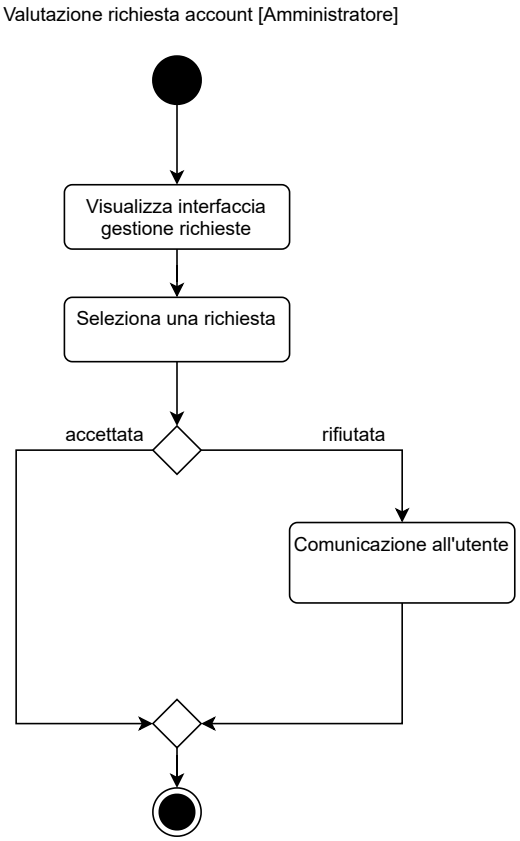

Figura 14: Activity Diagrams del caso d'uso *Valuta registrazione utenti*

3 Modello logico dei dati

La presente sezione descrive il modello logico dei dati adottato nel sistema, che rappresenta l'organizzazione e la struttura concettuale delle informazioni da gestire.

3.1 Diagramma E/R

IMMAGINE DIAGRAMMA ER PRIMA RISTRUTTURAZIONE

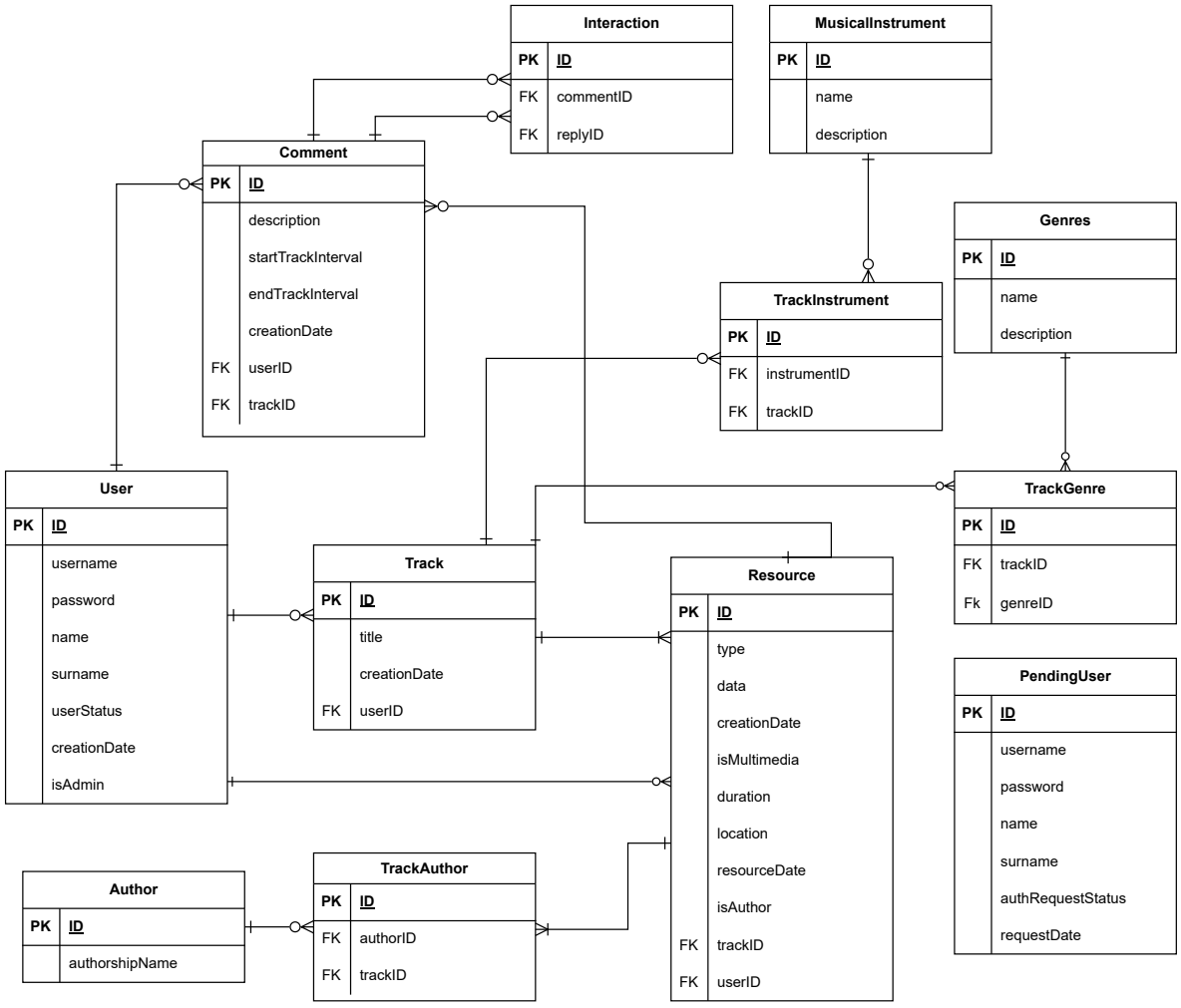

Figura 15: Diagramma E/R ristrutturato

3.2 Descrizione delle entità

Descrizione delle entità principali (possiamo rimuoverlo volendo)

4 Architettura logica

Questa sezione presenta l'architettura logica del sistema, ovvero come sono organizzati i principali componenti software e come interagiscono tra loro. L'obiettivo è offrire una panoramica chiara della struttura del progetto, utile per guidare lo sviluppo e facilitare la manutenzione nel tempo.

4.1 Diagramma UML delle classi - Model

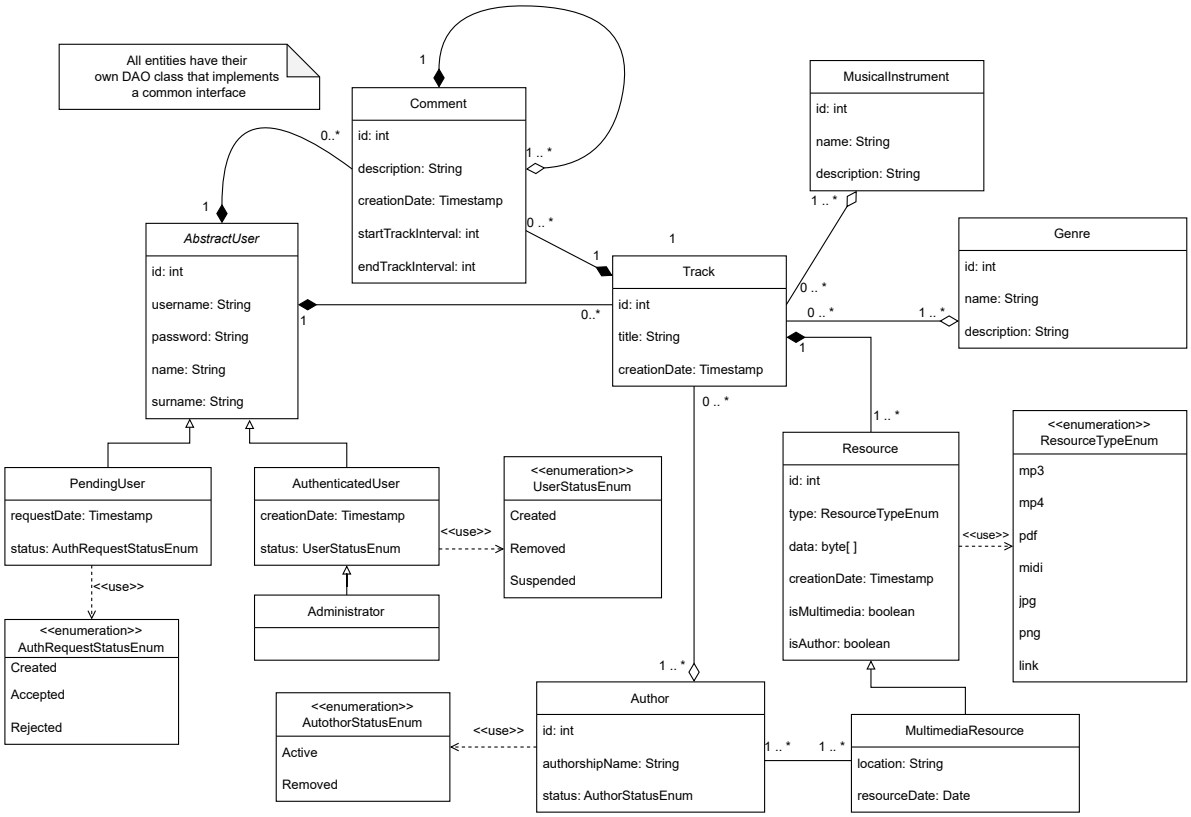

Figura 16: Class Diagrams UML relative al *Model*

4.2 Diagramma UML delle classi - Control

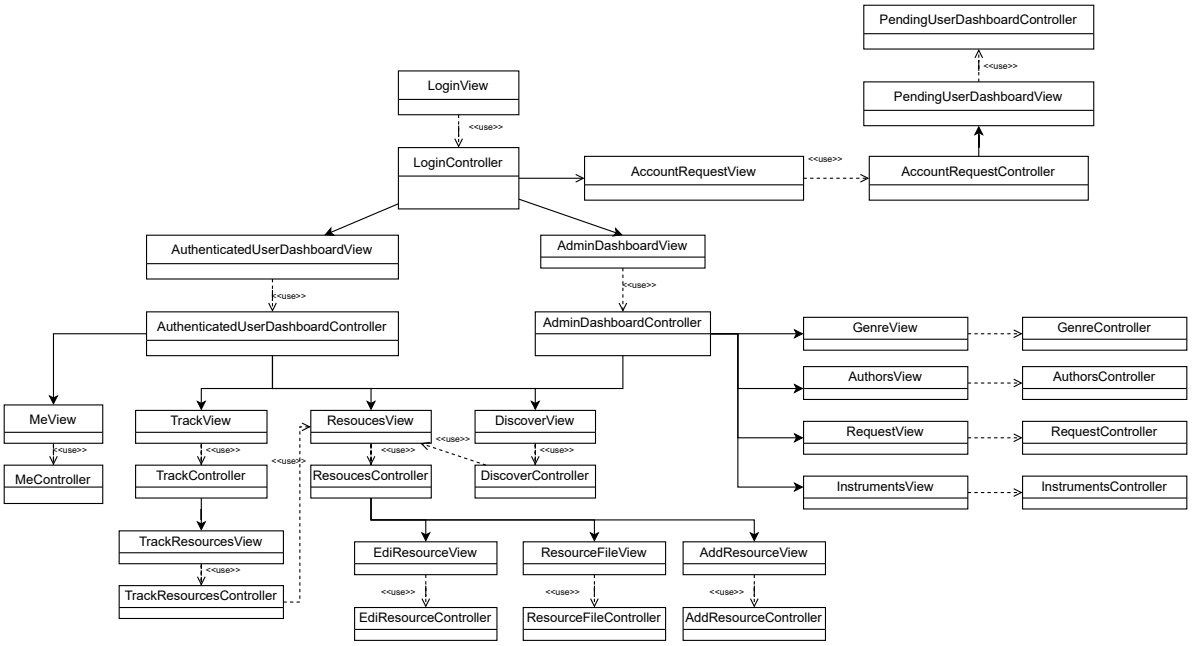

Figura 17: Class Diagrams UML relative al *Controller*

Il diagramma mostra la classe base `Controller` e tutte le sue sottoclassi organizzate nei seguenti pacchetti:

```
Controller
  common
    AddResourceController
    DiscoverController
    EditResourceController
    ResourceFileController
    ResourcesController
    TrackResourcesController
    TracksController
  admin
    AdminDashboardController
    AuthorsController
    GenresController
    InstrumentsController
    RequestsController
    UsersController
  authenticatedUser
    AuthenticatedUserDashboardController
    MeController
  authentication
    AccountRequestController
    LoginController
    SessionManager
  pendingUser
    PendingUserDashboardController
```

Tutte le classi controller estendono la classe base `Controller` e implementano l'interfaccia `Initializable` di JavaFX.

4.3 Diagramma UML delle classi - Utility ed Exception

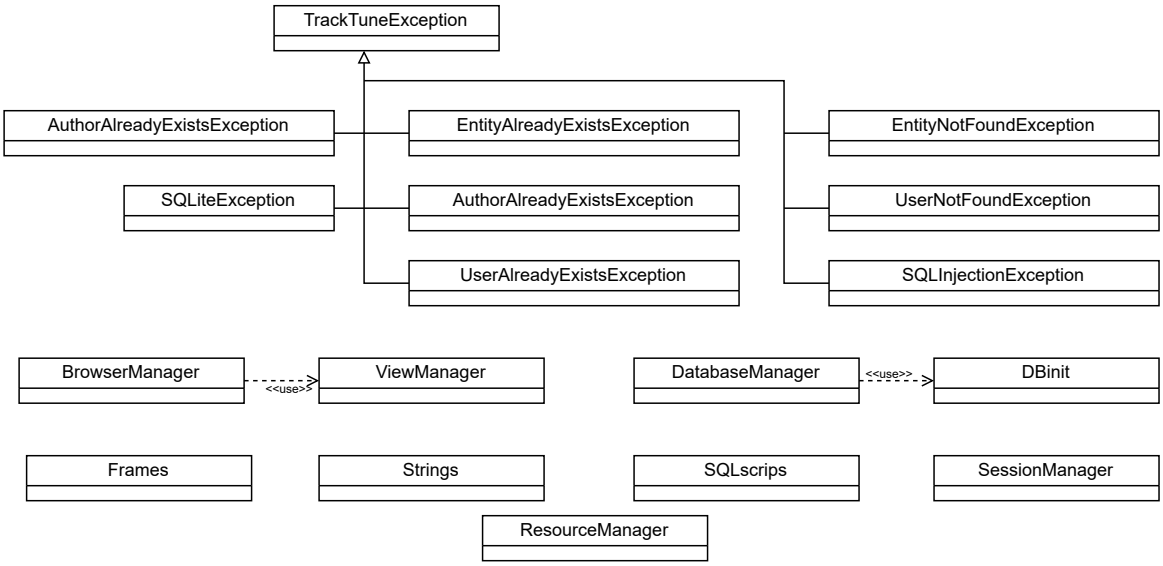

Figura 18: Class Diagrams UML relative a *Utility ed Exception*

4.4 Struttura del progetto

Di seguito è riportato l'albero dei file principali del progetto, illustrante la suddivisione in moduli, package e risorse:

```
TrackTune/
  src/
    main/
      java/                    % codice sorgente principale
        app/tracktune/
          config/
          controller/
          exception/
          interfaces/
          model/
          utils/
            DatabaseManager.java
            DBInit.java
          view/
          Main.java
      resources/                % FXML, CSS, immagini
    test/                      % test automatici (JUnit)
pom.xml                       % file di build Maven
```

4.5 Sequence Diagrams

I diagrammi di sequenza sono rappresentati esclusivamente per le classi più rilevanti e significative del sistema.

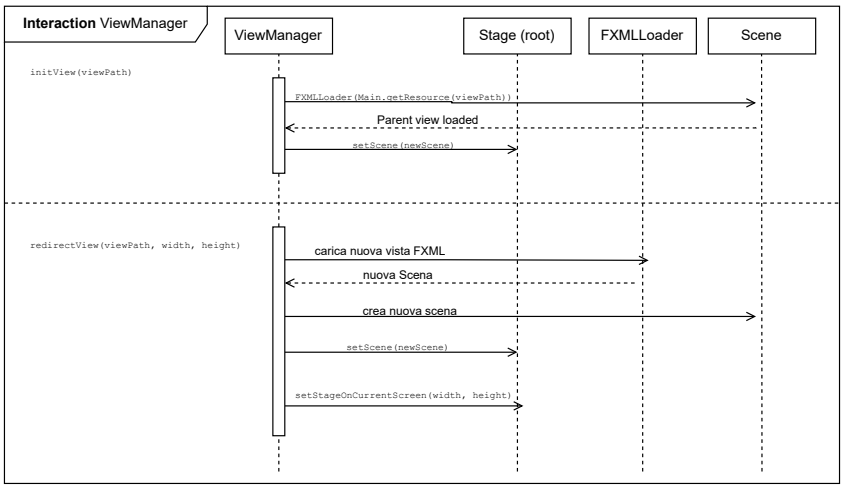

Figura 19: Sequence diagram relavito alla classe `ViewManager.java` in particolare al cambio di scena e inizializzazione root iniziale

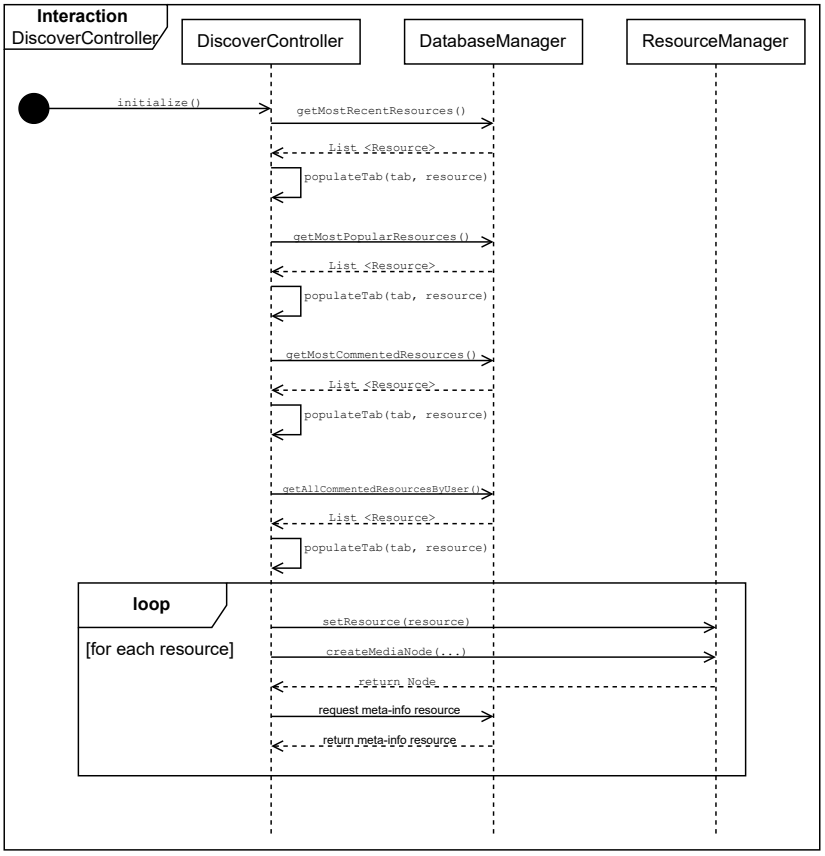

Figura 20: Sequence diagram relavito alla classe `DiscoverController.java`

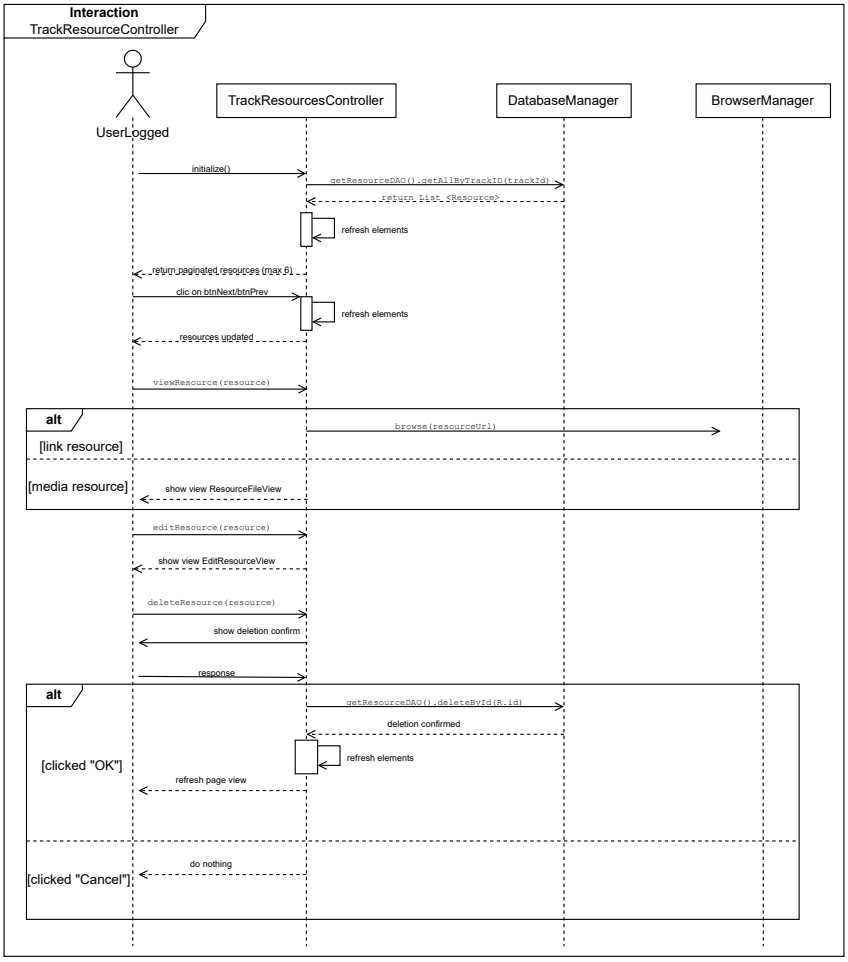

Figura 21: Sequence diagram relavito alla classe `TrackResourceController.java`

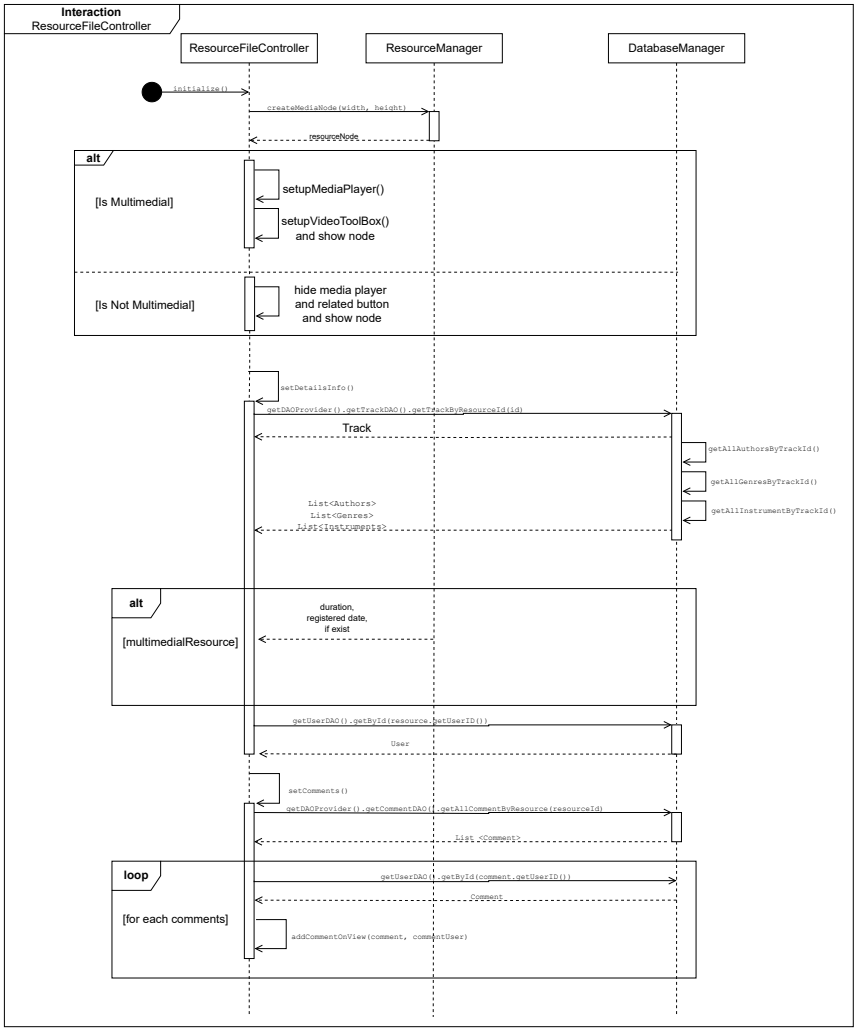

Figura 22: Sequence diagram relavito alla classe `ResourceFileController.java`

5 Progettazione e sviluppo

5.1 Pattern utilizzati

5.1.1 Pattern architetturale

Il sistema è stato progettato seguendo il pattern architetturale *Model-View-Controller* (MVC), una struttura ampiamente adottata nello sviluppo di interfacce grafiche per separare le responsabilità delle diverse componenti software.
Il pattern MVC suddivide l'applicazione in tre componenti principali:

- **Model**: rappresenta la logica e lo stato dell'applicazione. Contiene i dati, le classi di dominio e la logica di accesso ai dati (DAO), indipendente dalla logica di presentazione.
- **View**: è la parte visiva dell'applicazione, ovvero ciò che l'utente vede e con cui interagisce. In JavaFX, questa è rappresentata dai file `.fxml`, dai fogli di stile CSS e dai componenti grafici associati.
- **Controller**: gestisce l'interazione tra Model e View. Riceve gli input dell'utente dalla View, elabora le richieste e aggiorna il Model o la View di conseguenza. I controller in JavaFX implementano l'interfaccia `Initializable` per gestire l'inizializzazione della scena.

L'utilizzo del pattern MVC risulta particolarmente adatto in combinazione con JavaFX, poiché questa libreria supporta nativamente la separazione tra interfaccia grafica (FXML), logica di controllo (Controller) e dati (Model). In questo modo, il codice risulta più modulare, testabile e manutenibile.

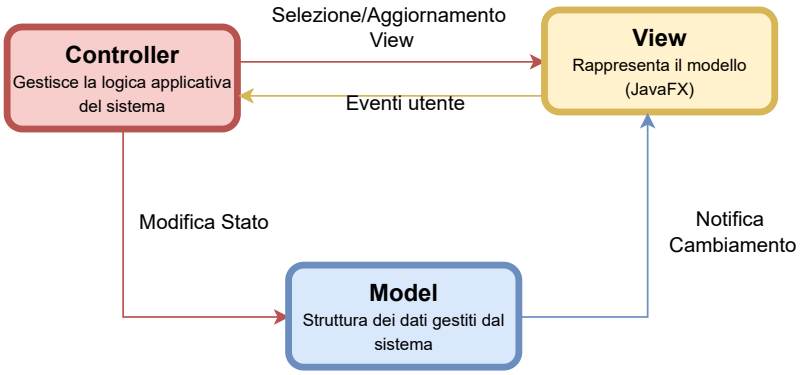

Figura 23: Pattern MVC

5.1.2 Design pattern utilizzati

Durante lo sviluppo del progetto, si è deciso di adottare alcuni design pattern per migliorare la struttura e la gestione del codice. Tra questi:

- **Singleton Pattern:** è un pattern creazionale che serve a garantire che una certa classe abbia una sola istanza. Ne sono un esempio tutte le classi del progetto denominate con `Manager` (`BrowserManager`, `ResourceManager` ...ecc).

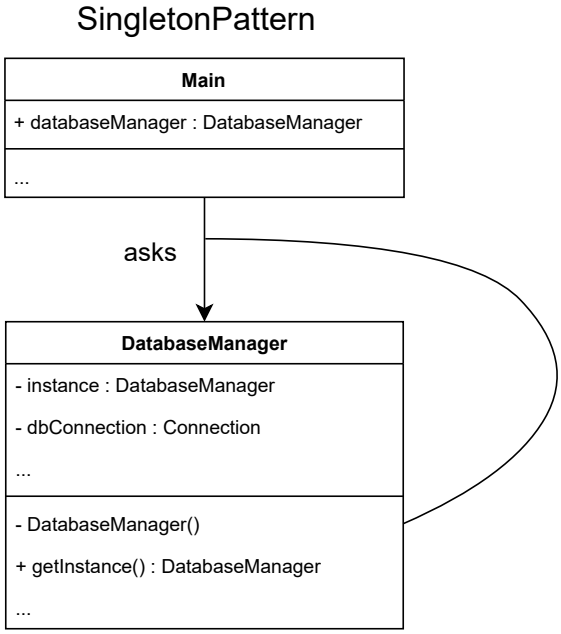

Figura 24: Esempio Singleton Pattern riguardante la classe `DatabaseManager.java`

- **Observer Pattern:** è un pattern comportamentale che si basa sul rapporto tra un oggetto osservatore e un oggetto osservato. L'osservatore si registra per essere notificato ogni volta che l'osservato cambia stato. In Java, questo pattern viene applicato implicitamente quando si lavora con strutture come `ObservableList`, molto usate per aggiornare automaticamente l'interfaccia utente in caso di modifiche. Ne sono un esempio tutti quei controlli utilizzati per filtrare, per esempio, le risorse o le tracce.

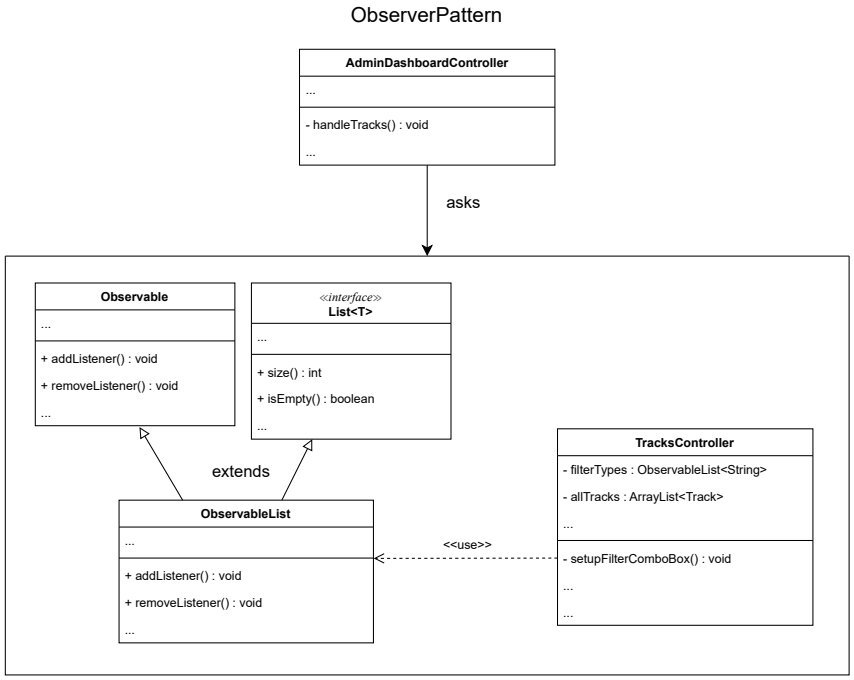

Figura 25: Esempio Observer Pattern

- **Factory Pattern:** è un pattern creazionale che permette di delegare la creazione di un oggetto a un metodo preciso, senza specificare esattamente quale classe verrà istanziata. È stato utilizzato per la comunicazione tra due controller di due viste diverse, ad esempio, quando è necessario che un controller passi dei parametri ad un controller ad esso dipendente. Un esempio di codice utilizzato è il seguente (passaggio da lista di risorse a vista di una specifica risorsa):

```
1 FXMLLoader loader = new FXMLLoader(...);
2 loader.setControllerFactory(_ -> new ResourceFileController(resource));
```

Listing 1: Esempio factory pattern

In questo caso, `FXMLLoader` gestisce internamente il factory pattern attraverso il metodo `setControllerFactory`.

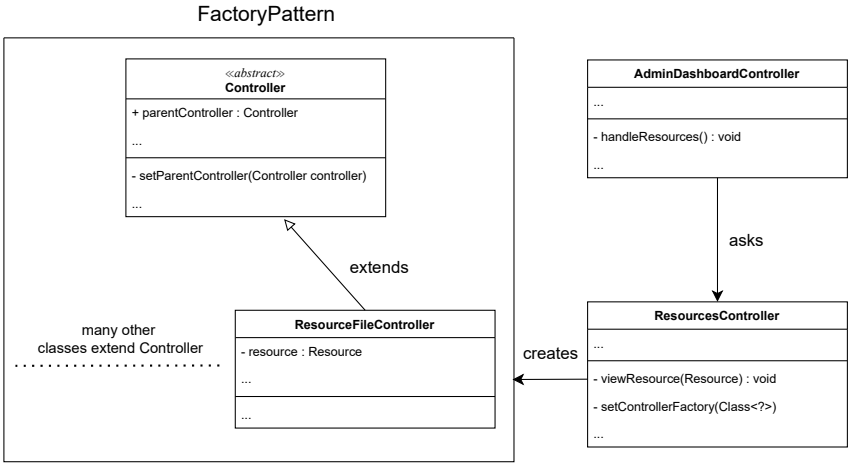

Figura 26: Esempio Observer Pattern

- **DAO (Data Access Object) Pattern:** è un pattern strutturale che separa la logica di accesso ai dati dalla logica di business, e permette di fornire un'interfaccia di programmazione comune per le transazioni su database. Nel nostro caso, è stata definita un'interfaccia generica `DAO<T>` nel package `model`, che viene implementata da tutte le classi DAO. Ogni entità del modello è rappresentata come una semplice classe Java (POJO), che viene poi gestita dalla relativa implementazione DAO.

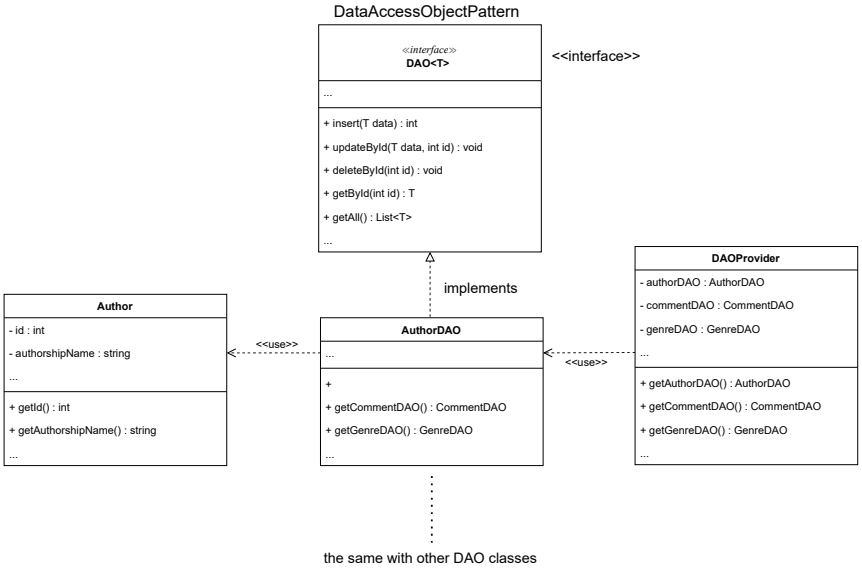

Figura 27: Esempio DAO Pattern

5.2 Gestione dell'accesso ai dati (DAO e DAOProvider)

Il sistema adotta il pattern *Data Access Object* (DAO) per gestire l'accesso ai dati in modo modulare e disaccoppiato dalla logica applicativa. Inoltre, abbiamo specificato operazioni ad hoc, in alcune classi DAO, in aggiunta a quelle dell'interfaccia comune, per facilitare alcune logiche di accesso che altrimenti sarebbero state pedanti.
L'accesso agli oggetti DAO all'interno dell'applicazione non avviene in modo diretto, ma è stato centralizzato attraverso una classe `"Wrapper"` denominata `DAOProvider`. Questo componente funge

da punto di accesso unico (simile a un *factory singleton*) per le istanze DAO, garantendo l'incapsulazione, una gestione coerente e facilitando eventuali modifiche future nel meccanismo di creazione o nella logica di accesso ai dati.

5.3 Classi di supporto e utility

Oltre alle classi principali legate al dominio e all'interfaccia utente, il sistema include alcune classi di supporto che forniscono funzionalità generiche e trasversali, riutilizzate in più parti del progetto. In questa sezione vengono descritte le principali classi di tipo *utility* utilizzate.

5.3.1 SessionManager

La classe `SessionManager` si occupa della gestione della sessione utente all'interno dell'applicazione. Permette di salvare e recuperare rapidamente informazioni legate all'utente attualmente autenticato (es. ruolo, username, ID), rendendo disponibili questi dati ad altri componenti senza doverli passare esplicitamente. La classe è implementata come **Singleton**, per garantire un'unica istanza condivisa.

5.3.2 DBInit

La classe `DBInit` è una classe utility responsabile della generazione dell'intera struttura del database SQLite. Utilizza un approccio basato su stringhe statiche, compatibile con l'inizializzazione tramite la libreria SQLite, rendendo il processo più veloce e flessibile. Include inoltre istruzioni per il controllo e l'inserimento dell'utente amministratore predefinito. Il metodo `getDBInitStatement()` restituisce concatenati tutti i comandi SQL di creazione tabelle, utili per inizializzare il database in modo completo.

5.3.3 SQLiteScripts

La classe `SQLiteScripts`, anch'essa nel package `app.tracktune.utils`, fornisce metodi statici per gestire operazioni complesse o ricorrenti sul database. Tra le principali funzionalità offerte troviamo:

- `checkForSQLInjection(...)`: controlla se stringhe di input contengono pattern pericolosi riconducibili a SQL injection.
- `deleteTrack(...)`: esegue l'eliminazione completa e sicura di un brano, rimuovendo anche tutte le dipendenze correlate (commenti, risorse, interazioni, ecc.).
- `getMostRecentResources(...)`: restituisce le 5 risorse più recenti.
- `getMostPopularResources(...)`: ottiene le risorse più popolari, in base alla frequenza di appartenenza ai brani.
- `getMostCommentedResources(...)`: seleziona le risorse con il maggior numero di commenti.

6 Fase di Test

6.1 Introduzione

La fase di test ha lo scopo di verificare che il sistema sviluppato sia corretto, affidabile e conforme ai requisiti stabiliti. Questa fase è essenziale per individuare eventuali errori o comportamenti inattesi prima della distribuzione finale del software. I test sono stati suddivisi in più livelli:

- **JUnit Test**: sono stati utilizzati per testare in modo automatico le unità di codice, in particolare metodi e classi del backend.
- **Test degli sviluppatori**: sono stati condotti test manuali durante lo sviluppo, volti a verificare il corretto funzionamento delle funzionalità implementate sia a livello logico che grafico. Questi test includono l'interazione con l'interfaccia grafica, la gestione degli errori e la validazione dei dati.
- **Test di utenti esterni**: una fase di test è stata dedicata all'interazione con utenti esterni al team di sviluppo, per raccogliere feedback sull'usabilità, sull'intuitività dell'interfaccia e sull'esperienza complessiva.

6.2 JUnit Test

Per la fase di test automatici è stato utilizzato il framework **JUnit**, che ha permesso di validare il corretto comportamento delle unità di codice in modo rapido e ripetibile. I test sono stati strutturati per coprire le principali funzionalità del sistema, con particolare attenzione alla correttezza logica dei metodi e alla gestione degli errori.
Le principali caratteristiche di JUnit impiegate sono state:

- **Annotazioni**: sono state utilizzate annotazioni come `@Test` per definire i metodi di test, `@BeforeAll` per eseguire l'inizializzazione comune prima di tutti i test, e `@TestInstance` per permettere la condivisione dello stato tra i test all'interno della stessa classe.
- **Asserzioni**: sono state usate le principali asserzioni offerte da JUnit, tra cui `assertEquals`, `assertNotNull`, `assertTrue` e `assertThrows`, per verificare i valori attesi, la corretta creazione degli oggetti, la validità di condizioni booleane e il corretto lancio delle eccezioni.

I test sono collocati nella directory `test`, separata dal codice sorgente (`main`), così da mantenere indipendente la logica applicativa dal codice di test. I test sono stati sviluppati in organizzati in package che rispecchiano la struttura del codice principale, ad esempio:

- `app.tracktune.model.author` per quanto riguarda i test degli autori
- `app.tracktune.model.user` per quanto riguarda i test sugli utenti
- ecc

I test seguono un pattern comune:

- **Setup**: Inizializzazione di un database SQLite in memoria per i test
- **Test CRUD**: Test per le operazioni Create, Read, Update, Delete
- **Test di relazioni**: Test per le relazioni tra entità (es. tracce-generi, tracce-autori)

Per i test viene utilizzato un database SQLite in memoria, in questo modo abbiamo un migliore isolamento tra software e test, impendeno qualunque effetto collaterale tra dati reali e di testing. Il database viene inizializzato tramite la classe `BBInit`

```java
// Esempio di inizializzazione del database per i test
String url = "jdbc:sqlite::memory:";
Connection connection = DriverManager.getConnection(url);
Statement stmt = connection.createStatement();
stmt.execute("PRAGMA foreign_keys = ON;");
String[] ddl = DBInit.getDBInitStatement().split(";");
for (String query : ddl) {
    if (!query.trim().isEmpty()) stmt.execute(query.trim() + ";");
}
```

Listing 2: Esempio creazione database in memoria

Ogni classe DAO ha passato correttamente tutti i test, di seguito si riporta una classe di test come esempio:

```java
import static org.junit.jupiter.api.Assertions.*;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class UserDAOTest {

    private UserDAO userDAO;

@BeforeAll
    void setup() throws Exception {
        Connection connection =
            DriverManager.getConnection("jdbc:sqlite::memory:");
        try (Statement stmt = connection.createStatement()) {
            stmt.execute("PRAGMA foreign_keys = ON;");
            String[] ddl = DBInit.getDBInitStatement().split(";");
            for (String query : ddl) {
                if (!query.trim().isEmpty()) {
                    stmt.execute(query.trim() + ";");
                }
            }
        }
        DatabaseManager.setTestConnection(connection);
        DatabaseManager db = DatabaseManager.getInstance();
        userDAO = new UserDAO(db);
    }

    /**
     * Verifies the entry and retrieval of an AuthenticatedUser by ID.
     */
  @Test
    void insertAndGetById_AuthenticatedUser() {
        Timestamp now = new Timestamp(System.currentTimeMillis());
        AuthenticatedUser user = new AuthenticatedUser(null, "testUser",
            "password123", "Test", "User", UserStatusEnum.ACTIVE, now);
        Integer id = userDAO.insert(user);
        assertNotNull(id);

        User fetched = userDAO.getById(id);
        assertInstanceOf(AuthenticatedUser.class, fetched);
        AuthenticatedUser authUser = (AuthenticatedUser) fetched;
        assertEquals("testUser", authUser.getUsername());
        assertEquals("password123", authUser.getPassword());
        assertEquals("Test", authUser.getName());
        assertEquals("User", authUser.getSurname());
        assertEquals(UserStatusEnum.ACTIVE, authUser.getStatus());
        assertFalse(fetched instanceof Administrator);
    }
```

```
45
46      /**
47       * Verifies the entry and retrieval of an Administrator by ID.
48       */
49      @Test
50      void insertAndGetById_Administrator() {
51          Timestamp now = new Timestamp(System.currentTimeMillis());
52          Administrator admin = new Administrator(null, "adminUser",
                "adminPass", "Admin", "User", UserStatusEnum.ACTIVE, now);
53          Integer id = userDAO.insert(admin);
54          assertNotNull(id);
55
56          User fetched = userDAO.getById(id);
57          assertInstanceOf(Administrator.class, fetched);
58          Administrator fetchedAdmin = (Administrator) fetched;
59          assertEquals("adminUser", fetchedAdmin.getUsername());
60          assertEquals("adminPass", fetchedAdmin.getPassword());
61          assertEquals("Admin", fetchedAdmin.getName());
62          assertEquals("User", fetchedAdmin.getSurname());
63          assertEquals(UserStatusEnum.ACTIVE, fetchedAdmin.getStatus());
64      }
65
66      /**
67       * Verify the update of an existing AuthenticatedUser.
68       */
69      @Test
70      void update_AuthenticatedUser() {
71          Timestamp now = new Timestamp(System.currentTimeMillis());
72          AuthenticatedUser user = new AuthenticatedUser(null, "updateUser",
                "initialPass", "Initial", "User", UserStatusEnum.ACTIVE, now);
73          Integer id = userDAO.insert(user);
74
75          AuthenticatedUser updated = new AuthenticatedUser(id, "updateUser",
                "updatedPass", "Updated", "User", UserStatusEnum.SUSPENDED, now);
76          userDAO.updateById(updated, id);
77
78          User result = userDAO.getById(id);
79          assertInstanceOf(AuthenticatedUser.class, result);
80          AuthenticatedUser authUser = (AuthenticatedUser) result;
81          assertEquals("updateUser", authUser.getUsername());
82          assertEquals("updatedPass", authUser.getPassword());
83          assertEquals("Updated", authUser.getName());
84          assertEquals(UserStatusEnum.SUSPENDED, authUser.getStatus());
85      }
86
87      /**
88       * Verify the update of an existing Administrator.
89       */
90      @Test
91      void update_Administrator() {
92          Timestamp now = new Timestamp(System.currentTimeMillis());
93          Administrator admin = new Administrator(null, "updateAdmin",
                "initialPass", "Initial", "Admin", UserStatusEnum.ACTIVE, now);
94          Integer id = userDAO.insert(admin);
95
96          Administrator updated = new Administrator(id, "updateAdmin",
                "updatedPass", "Updated", "Admin", UserStatusEnum.SUSPENDED, now);
97          userDAO.updateById(updated, id);
98
99          User result = userDAO.getById(id);
100         assertInstanceOf(Administrator.class, result);
```

```
101         Administrator fetchedAdmin = (Administrator) result;
102         assertEquals("updateAdmin", fetchedAdmin.getUsername());
103         assertEquals("updatedPass", fetchedAdmin.getPassword());
104         assertEquals("Updated", fetchedAdmin.getName());
105         assertEquals(UserStatusEnum.SUSPENDED, fetchedAdmin.getStatus());
106     }
107
108     /**
109      * Verifies the deletion of a user by ID and that subsequent retrieval
             throws an exception.
110      */
111     @Test
112     void deleteById_UserNotFound() {
113         Timestamp now = new Timestamp(System.currentTimeMillis());
114         AuthenticatedUser user = new AuthenticatedUser(null, "deleteUser",
                "password123", "Delete", "User", UserStatusEnum.ACTIVE, now);
115         Integer id = userDAO.insert(user);
116
117         userDAO.deleteById(id);
118
119         assertThrows(app.tracktune.exceptions.SQLiteException.class, () ->
                userDAO.getById(id));
120     }
121
122     /**
123      * Verify that the getAll method returns all users entered.
124      */
125     @Test
126     void getAllUsers() {
127         Timestamp now = new Timestamp(System.currentTimeMillis());
128         AuthenticatedUser user1 = new AuthenticatedUser(null, "user1",
                "password1", "User", "One", UserStatusEnum.ACTIVE, now);
129         Administrator admin1 = new Administrator(null, "admin1", "password2",
                "Admin", "One", UserStatusEnum.ACTIVE, now);
130         userDAO.insert(user1);
131         userDAO.insert(admin1);
132
133         List<User> all = userDAO.getAll();
134         assertTrue(all.size() >= 2);
135         assertTrue(all.stream().anyMatch(u ->
                u.getUsername().equals("user1")));
136         assertTrue(all.stream().anyMatch(u ->
                u.getUsername().equals("admin1")));
137     }
138
139     /**
140      * Check recovery of an active user by username.
141      */
142     @Test
143     void getActiveUserByUsername_ReturnsUser() {
144         String uniqueUsername = "unique" + System.currentTimeMillis();
145         Timestamp now = new Timestamp(System.currentTimeMillis());
146         AuthenticatedUser user = new AuthenticatedUser(null, uniqueUsername,
                "password123", "Unique", "User", UserStatusEnum.ACTIVE, now);
147         userDAO.insert(user);
148
149         User fetched = userDAO.getActiveUserByUsername(uniqueUsername);
150         assertNotNull(fetched);
151         assertEquals(uniqueUsername, fetched.getUsername());
152         assertEquals("Unique", fetched.getName());
153         assertEquals("User", fetched.getSurname());
```

```
154         assertInstanceOf(AuthenticatedUser.class, fetched);
155         assertEquals(UserStatusEnum.ACTIVE, ((AuthenticatedUser)
                fetched).getStatus());
156     }
157
158     /**
159      * Verifies that retrieving an inactive user by username returns null.
160      */
161   @Test
162     void getActiveUserByUsername_InactiveUserReturnsNull() {
163         String uniqueUsername = "inactive" + System.currentTimeMillis();
164         Timestamp now = new Timestamp(System.currentTimeMillis());
165         AuthenticatedUser user = new AuthenticatedUser(null, uniqueUsername,
                "password123", "Inactive", "User", UserStatusEnum.SUSPENDED, now);
166         userDAO.insert(user);
167
168         User fetched = userDAO.getActiveUserByUsername(uniqueUsername);
169         assertNull(fetched);
170     }
171 }
```

Listing 3: Esempio test per la classe `UserTest.java`

È possibile lanciare tutti i test con due metodologie:

- Se Maven risulta correttamente installato all'interno del dispositivo, è possibile lanciare i test tramite:

```
mvn test
```

Listing 4: Esecuzione test tramite Maven

- Tramite IDE (per quanto riguarda IntelliJ, recarsi nel menù laterale di destra → Maven → Lifecycle → test → run)

6.3 Test degli sviluppatori

In questa fase lo sviluppatore ha immesso nel sistema degli input (sia corretti che errati) per verificare se la reazione del software fosse quella attesa. Le funzionalità sono state testate man mano manualmente durante lo sviluppo, per assicurarsi che ogni componente rispondesse correttamente in base agli scenari previsti.

6.4 Test di utenti esterni al sistema

Per valutare la qualità, la chiarezza e l'usabilità del sistema da diversi punti di vista, sono stati coinvolti utenti esterni appartenenti a due categorie distinte:

- **Utenti con competenze avanzate**: colleghi del corso di laurea in Informatica, con conoscenze tecniche in ambito software. Questi utenti hanno fornito feedback più approfonditi riguardo l'architettura, la fluidità dell'interfaccia e la coerenza del comportamento delle funzionalità. Sebbene non siano stati rilevati errori, il loro contributo è stato utile per confermare la solidità dell'applicazione anche in scenari di utilizzo meno comuni, oltre che per proporre miglioramenti specifici all'esperienza utente e alla gestione dei casi limite.
- **Utenti con competenze non avanzate**: amici e familiari privi di una formazione tecnica. In questo caso, il focus principale è stato posto sull'intuitività dell'interfaccia e sulla comprensibilità delle funzionalità offerte. I loro feedback si sono rivelati utili per verificare che l'utilizzo dell'applicazione fosse accessibile anche a un pubblico generalista, nonché per individuare eventuali errori o ambiguità.

I test sono stati svolti manualmente e in presenza, osservando il comportamento degli utenti durante l'interazione con il sistema e raccogliendo le loro impressioni immediate. Questo ha permesso di identificare aree migliorabili sia dal punto di vista tecnico che da quello dell'usabilità.