



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli studi di Firenze

Dipartimento di Ingegneria dell'Informazione

Swetify: un'applicazione a linea di comando per lo streaming musicale

Autori:

A. Delli Colli
N. Malgeri
F. Viti

Corso:

B024253
Basi di dati - Ingegneria del software

Anno accademico:

2023-2024

Docente del corso:

E. Vicario

Indice

1	Introduzione	3
1.1	Motivazioni	3
1.2	Architettura dell'applicazione e pratiche utilizzate	3
2	Requisiti	3
2.1	Funzionalità proposte per il cliente	3
2.2	Funzionalità proposte per l'artista	3
3	Progettazione	4
3.1	Diagramma dei casi d'uso	4
3.2	Template dei casi d'uso	4
3.3	Mockups	7
3.3.1	Pagina di accesso	7
3.3.2	Pagina di ricerca	7
3.3.3	Suggerimenti	8
3.3.4	Coda di riproduzione	8
3.4	Struttura delle classi	9
3.4.1	Domain model	9
3.4.2	Business logic	10
3.4.3	DAO	10
3.5	Design patterns	11
3.5.1	DAO	11
3.5.2	State	12
3.6	Diagramma di sequenza	13
4	Implementazione	13
4.1	Domain model	13
4.1.1	BaseEntity	13
4.1.2	BaseUser	13
4.1.3	Customer	13
4.1.4	Artist	13
4.1.5	Track	14
4.1.6	DurationConverter	14
4.1.7	Song	14
4.1.8	Podcast	14
4.1.9	Playlist	14
4.1.10	SongPlaylist	14
4.1.11	PodcastPlaylist	14
4.1.12	Album	14
4.1.13	PlaybackQueue	14
4.1.14	TrackPlaysCount	14
4.1.15	SongPlaysCount	14
4.1.16	PodcastPlaysCount	14
4.1.17	TrackPlaysCountListener	14
4.2	Business logic	15
4.2.1	Handler	15
4.2.2	Session	15
4.2.3	NavigationManager	15
4.2.4	AlbumViewHandler	15
4.2.5	ArtistInfoHandler	15
4.2.6	HomeHandler	15
4.2.7	LoginHandler	16
4.2.8	PlaybackHandler	16
4.2.9	PlaylistHandler	16

4.2.10	RegistrationHandler	16
4.2.11	SearchHandler	16
4.2.12	SuggestionsHandler	16
4.2.13	UserPlaylistsHandler	16
4.3	DAO	17
4.3.1	BaseDAO	17
4.3.2	CustomerDAO	17
4.3.3	ArtistDAO	17
4.3.4	SongDAO	17
4.3.5	PodcastDAO	17
4.3.6	SongPlaylistDAO	17
4.3.7	PodcastPlaylistDAO	17
4.3.8	AlbumDAO	17
4.3.9	SongPlaysCountDAO	17
4.3.10	PodcastPlaysCountDAO	17
4.3.11	SuggestionDAO	18
5	Test	18
5.1	Tipologie di test effettuati e organizzazione	18
5.2	Test di integrazione	19
5.2.1	SongDAOTest	19
5.2.2	ArtistDAOTest	20
5.2.3	CustomerDAOTest	20
5.2.4	SuggestionDAOTest	20
5.2.5	SongPlaylistDAOTest	20
5.2.6	SongPlaysCountDAOTest	20
5.3	Test funzionali	21
5.3.1	HomeHandlerTest	21
5.3.2	PlaybackHandlerTest	21
5.3.3	RegistrationLoginHandlersTest	21
5.3.4	AlbumLoadHandlerTest	22

1 Introduzione

1.1 Motivazioni

Il nostro intensivo utilizzo di piattaforme di streaming musicali ha suscitato in noi un interesse riguardo la loro struttura e il desiderio di replicarne il funzionamento.

Abbiamo deciso quindi di realizzare un'applicazione che simuli le loro funzionalità da noi denominata Swetify.

1.2 Architettura dell'applicazione e pratiche utilizzate

Per la progettazione del software è stato seguito lo standard UML, realizzando i diagrammi con StarUML. Il software è stato realizzato in Java. Il modello dei dati è rappresentato dal package **domainModel**, mentre la logica e la visualizzazione dell'interfaccia utente si trova nel package **businessLogic**. La **businessLogic** si interfaccia con dei DAO che si occupano di rendere i dati persistenti attraverso **JPA** (***Jakarta Persistence API***). In particolare è stato scelto **Hibernate** come implementazione della **JPA**. Come RDBMS è stato usato **H2**, integrato di default in **Hibernate**. L'architettura generale dell'applicazione è illustrata in Figura 1. Il testing è stato effettuato con il framework **JUnit 5**. Per semplicità, l'interfaccia effettivamente realizzata è a linea di comando. Sono stati realizzati dei mockup utilizzando **Inkscape**, un programma per grafica vettoriale, che rappresentano l'aspetto che avrebbe l'applicazione se fosse stata realizzata un'interfaccia grafica.

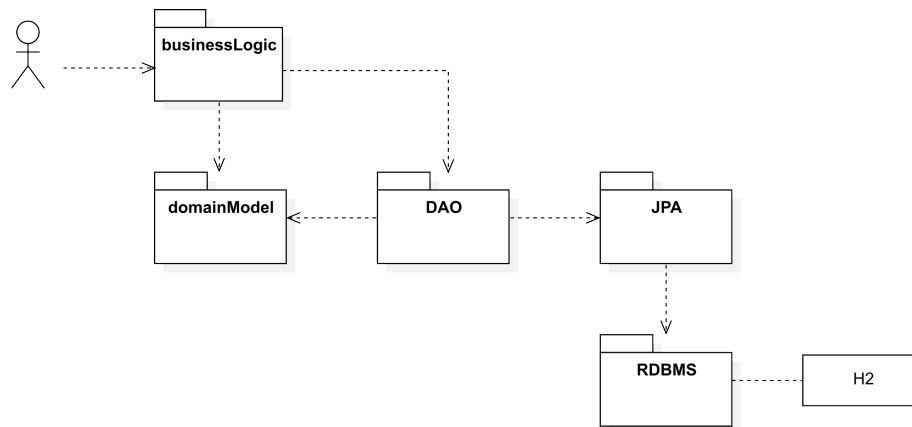


Figura 1: Architettura dell'applicazione

2 Requisiti

Swetify prevede la partecipazione di due tipologie di utenti: cliente e artista.

2.1 Funzionalità proposte per il cliente

- visualizzare un catalogo musicale che consenta agli utenti di cercare brani, album e artisti tramite una barra di ricerca.
- visualizzare le informazioni dettagliate di un brano, inclusi titolo, artista, album e durata.
- riprodurre, mettere in pausa e saltare i brani.
- creare, modificare ed eliminare le proprie playlist, aggiungere e rimuovere brani da queste playlist.
- ricevere raccomandazioni di brani basate sulla cronologia di ascolto dell'utente
- seguire gli artisti per ricevere aggiornamenti sui nuovi rilasci.

2.2 Funzionalità proposte per l'artista

- caricare album contenenti canzoni o podcast.

3 Progettazione

3.1 Diagramma dei casi d'uso

Come accennato nella sezione 2, ci sono due protagonisti, l'utente (*Customer*) e l'artista (*Artist*). La figura 2 rappresenta il diagramma dei casi d'uso, questi ultimi corrispondenti ai requisiti funzionali descritti precedentemente.

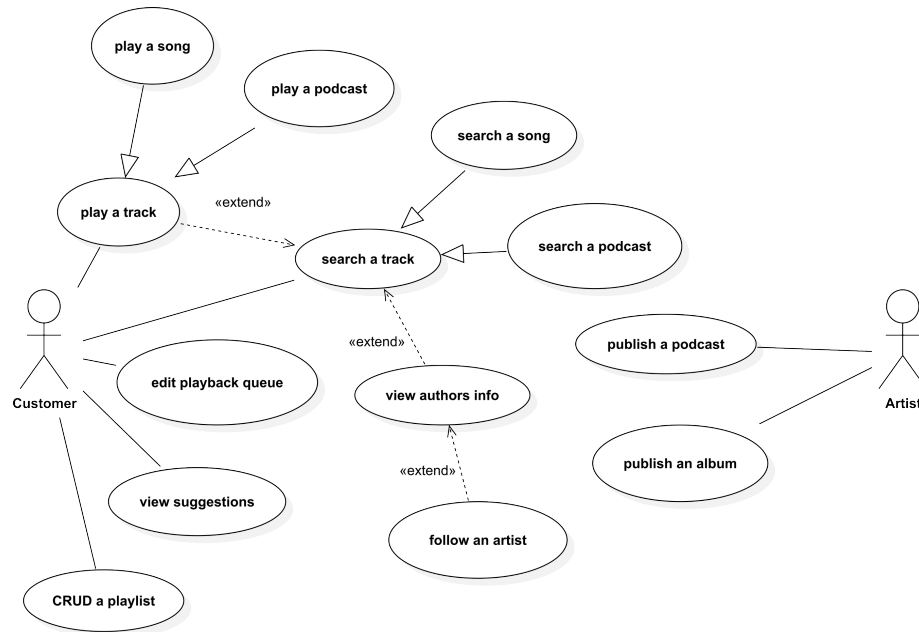


Figura 2: diagramma dei casi d'uso

3.2 Template dei casi d'uso

UC1	riproduzione canzone
livello	user goal
descrizione	l'utente cerca e riproduce una canzone
attori	cliente
pre-condizioni	l'utente deve avere le credenziali per effettuare l'accesso
post-condizioni	la canzone selezionata viene aggiunta alla coda
normale svolgimento	1) l'utente apre la pagina di ricerca (mockup in figura 4) 2) inserisce il nome di una canzone 3) seleziona una voce dall'elenco proposto 4) seleziona l'opzione "aggiungi in coda"
svolgimenti alternativi	4b) l'utente seleziona l'opzione "aggiungi in testa"

UC2	modifica playlist
livello	user goal
descrizione	l'utente modifica una delle sue playlist personali
attori	cliente
pre-condizioni	l'utente deve avere le credenziali per effettuare l'accesso ed avere una playlist salvata
post-condizioni	la playlist presenta i cambiamenti apportati dall'utente
normale svolgimento	1) l'utente apre la pagina "le mie playlist" 2) l'utente seleziona la playlist da modificare 3) l'utente cerca una canzone da aggiungere 4) l'utente termina salvando le modifiche
svolgimenti alternativi	3b) l'utente seleziona una canzone da rimuovere 4b) l'utente annulla le modifiche alla playlist

UC3	aggiunta album
livello	user goal
descrizione	l'artista carica un nuovo album sul suo profilo
attori	artista
pre-condizioni	l'artista deve avere le credenziali per effettuare l'accesso
post-condizioni	il nuovo album è visibile se cercato dagli utenti
normale svolgimento	1) l'artista seleziona l'opzione carica album 2) inserisce i nomi delle canzoni ed i rispettivi dati 3) l'artista termina l'inserimento salvando l'album
svolgimenti alternativi	3b) l'artista annulla il caricamento dell'album

UC4	iscrizione ad un artista
livello	function
descrizione	l'utente aggiunge un artista agli artisti seguiti
attori	cliente
pre-condizioni	l'utente deve avere le credenziali per effettuare l'accesso
post-condizioni	nel momento in cui l'artista carica un nuovo album l'utente può visualizzarlo nella sezione "nuovi rilasci"
normale svolgimento	1) l'utente apre la pagina di ricerca 2) inserisce il nome di una canzone 3) seleziona una voce dall'elenco proposto 4) seleziona l'opzione "aggiungi in coda"

UC5	visualizzazione dei consigliati
livello	user goal
descrizione	l'utente visualizza le canzoni ed i podcast consigliati
attori	cliente
pre-condizioni	l'utente deve avere le credenziali per effettuare l'accesso e avere ascoltato almeno una canzone o podcast
post-condizioni	i consigliati sono mostrati nella sezione apposita
normale svolgimento	1) l'utente va alla homepage 2) accede alla sezione dei consigliati 3) visualizza le tracce consigliate (mockup in figura 5)

3.3 Mockups

3.3.1 Pagina di accesso

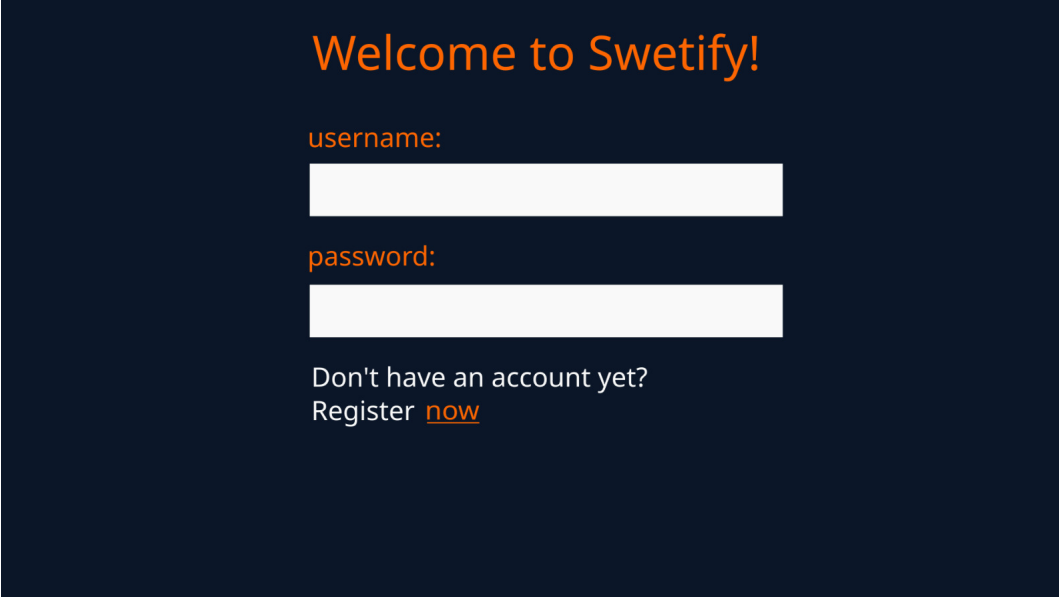


Figure 3 shows a login page mockup with a dark blue background. At the top, the text "Welcome to Swetify!" is displayed in orange. Below this, the labels "username:" and "password:" are shown in orange, each followed by a white input field. At the bottom, the text "Don't have an account yet?" is displayed in white, followed by the text "Register [now](#)" where "now" is a link in orange.

Figura 3: prototipo della pagina di login

3.3.2 Pagina di ricerca



Figure 4 shows a search page mockup with a dark blue background. At the top, there is a search bar with the placeholder text "searchName" and a close button (X). Below the search bar, the page is divided into three columns: "Songs:", "Podcasts:", and "Artists:". Each column contains a list of 9 items, each with a number (1-9) and a horizontal line for the item name. At the bottom right, there is a "go back" button with a right-pointing arrow.

Figura 4: prototipo della pagina di ricerca

3.3.3 Suggerimenti

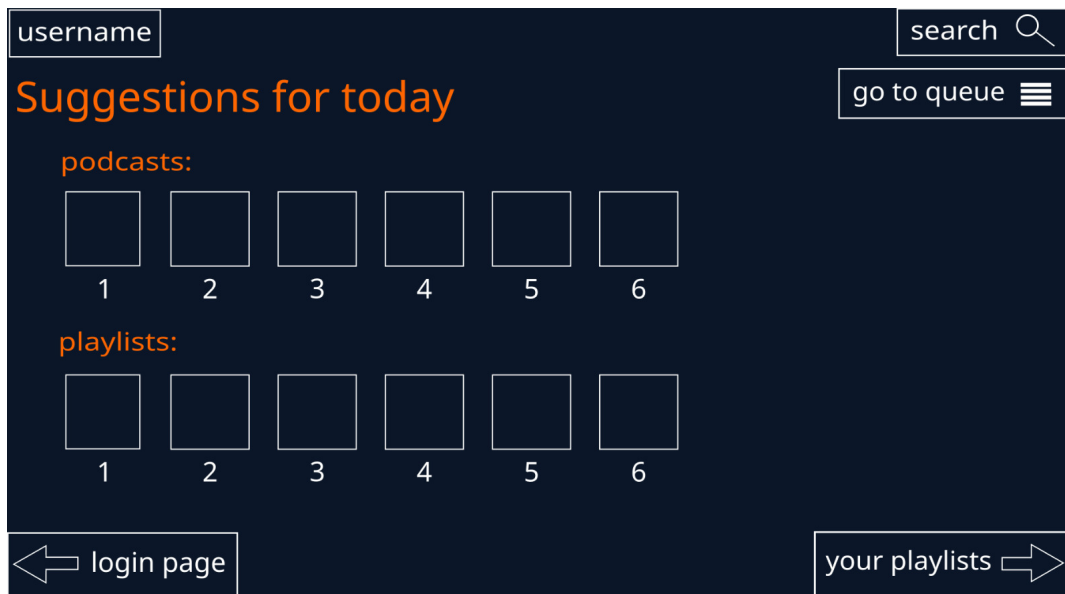


Figura 5: prototipo della pagina dei consigliati

3.3.4 Coda di riproduzione

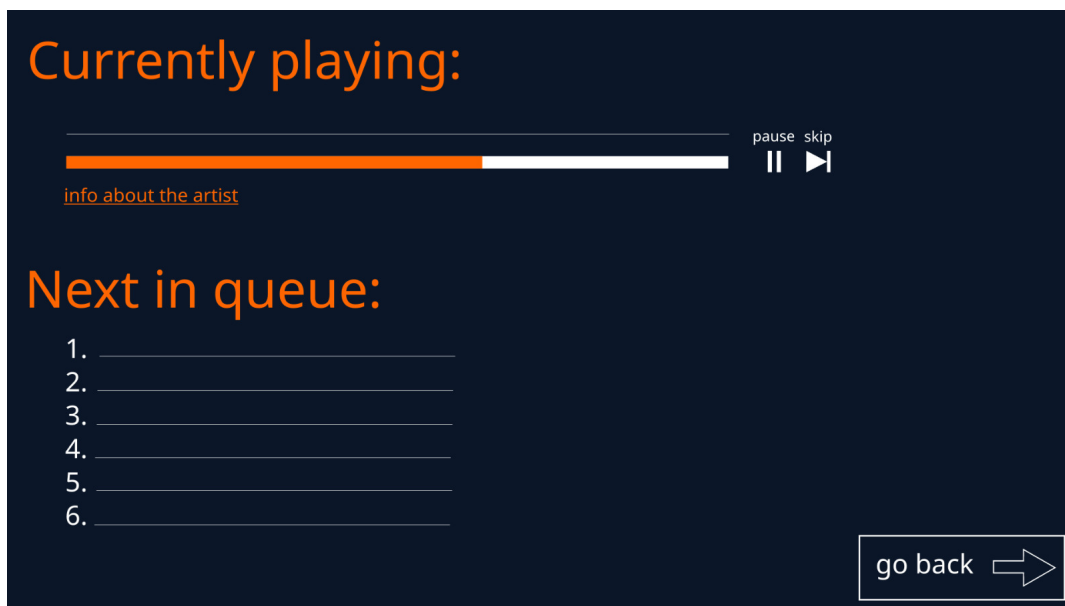


Figura 6: prototipo della coda di riproduzione

3.4 Struttura delle classi

3.4.1 Domain model

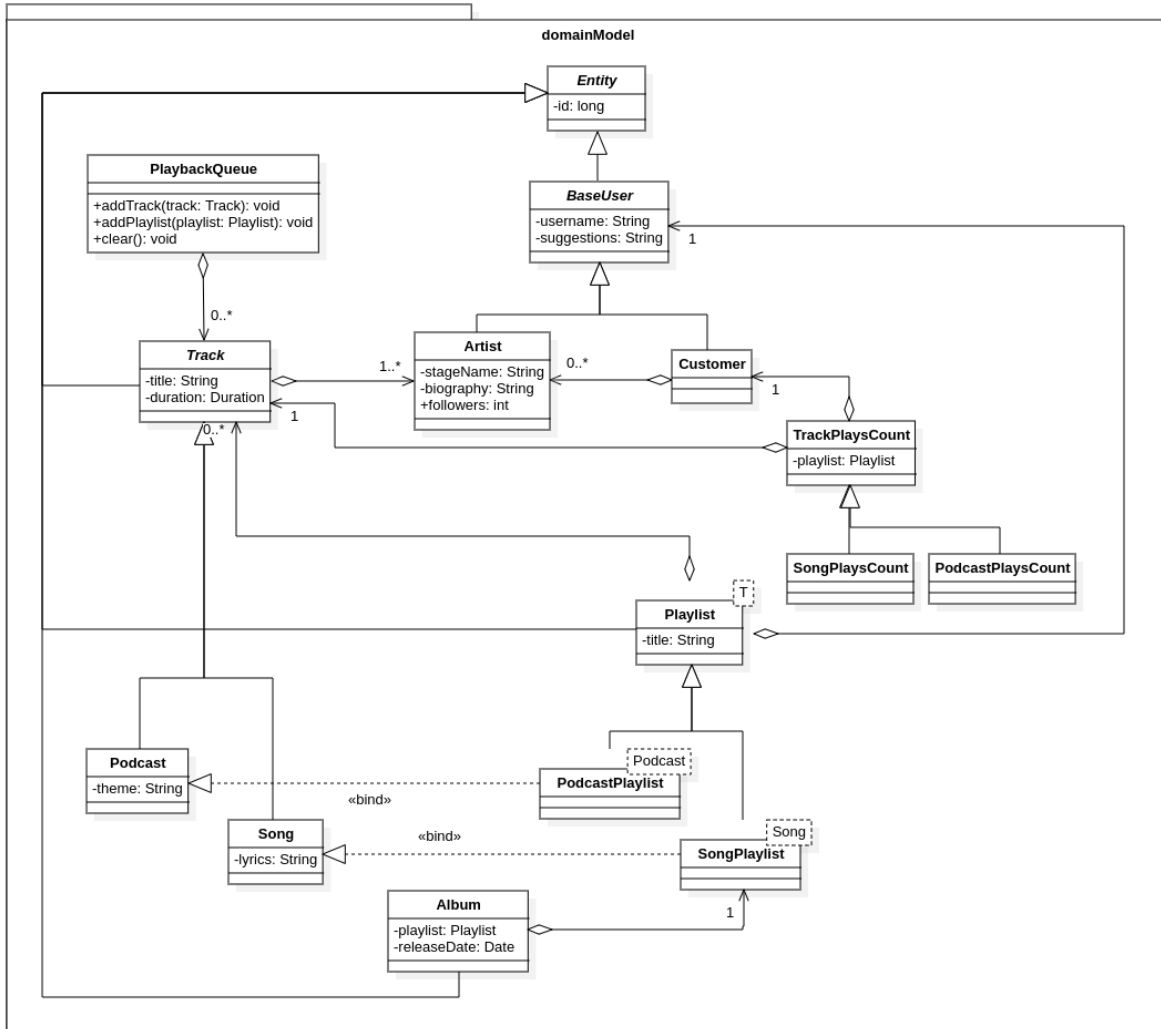


Figura 7: diagramma delle classi contenute nel package *domainModel*

Definisce la rappresentazione dei dati all'interno dell'applicazione e come gli oggetti sono mappati nel database. Per fare questo è stata usata la tecnica ORM (Object Relational Mapping), realizzata tramite l'utilizzo delle annotazioni fornite dalle JPA (package `jakarta.persistence`).

3.4.2 Business logic

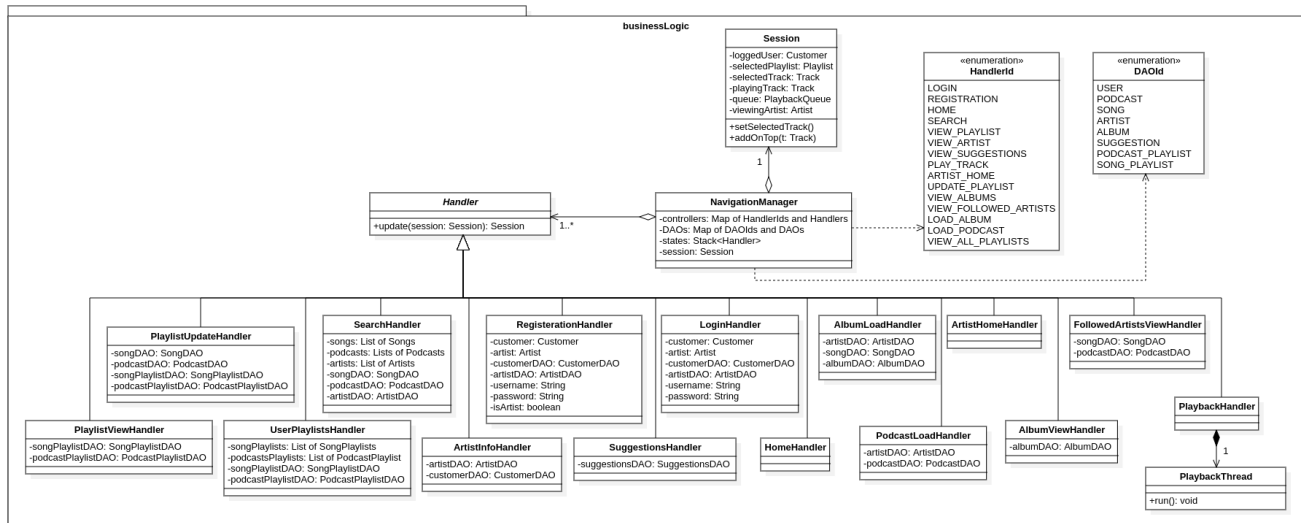


Figura 8: diagramma delle classi contenute nel package *businessLogic*

Definisce le regole, i processi e le funzionalità che guidano il comportamento dell'applicazione, in base alle diverse esigenze degli utenti. All'interno del package **businessLogic** (figura 8) sono presenti le classi che si occupano della visualizzazione, da linea di comando, del contenuto delle finestre dell'applicazione, distribuite in modo da averne una per pagina; queste classi sono dette *Handlers* (come ad esempio *HomeHandler* per la visualizzazione della homepage, *SearchHandler* per la finestra di ricerca). Il passaggio da una finestra all'altra viene gestito dalla classe *NavigationManager*.

3.4.3 DAO

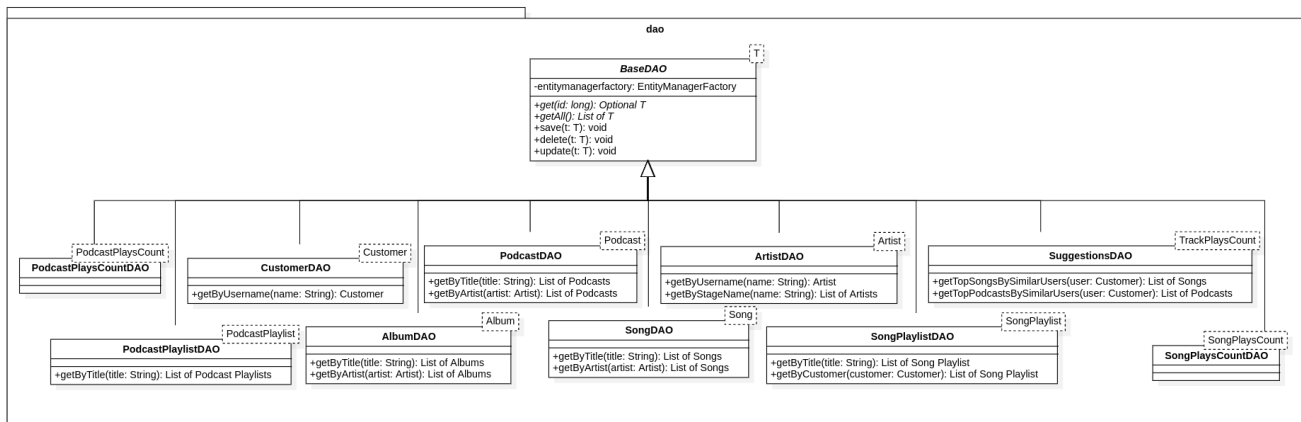


Figura 9: diagramma delle classi contenute nel package *dao*

Contiene le classi che realizzano il Data Access Object pattern che si occupa della persistenza dei dati.

3.5 Design patterns

3.5.1 DAO

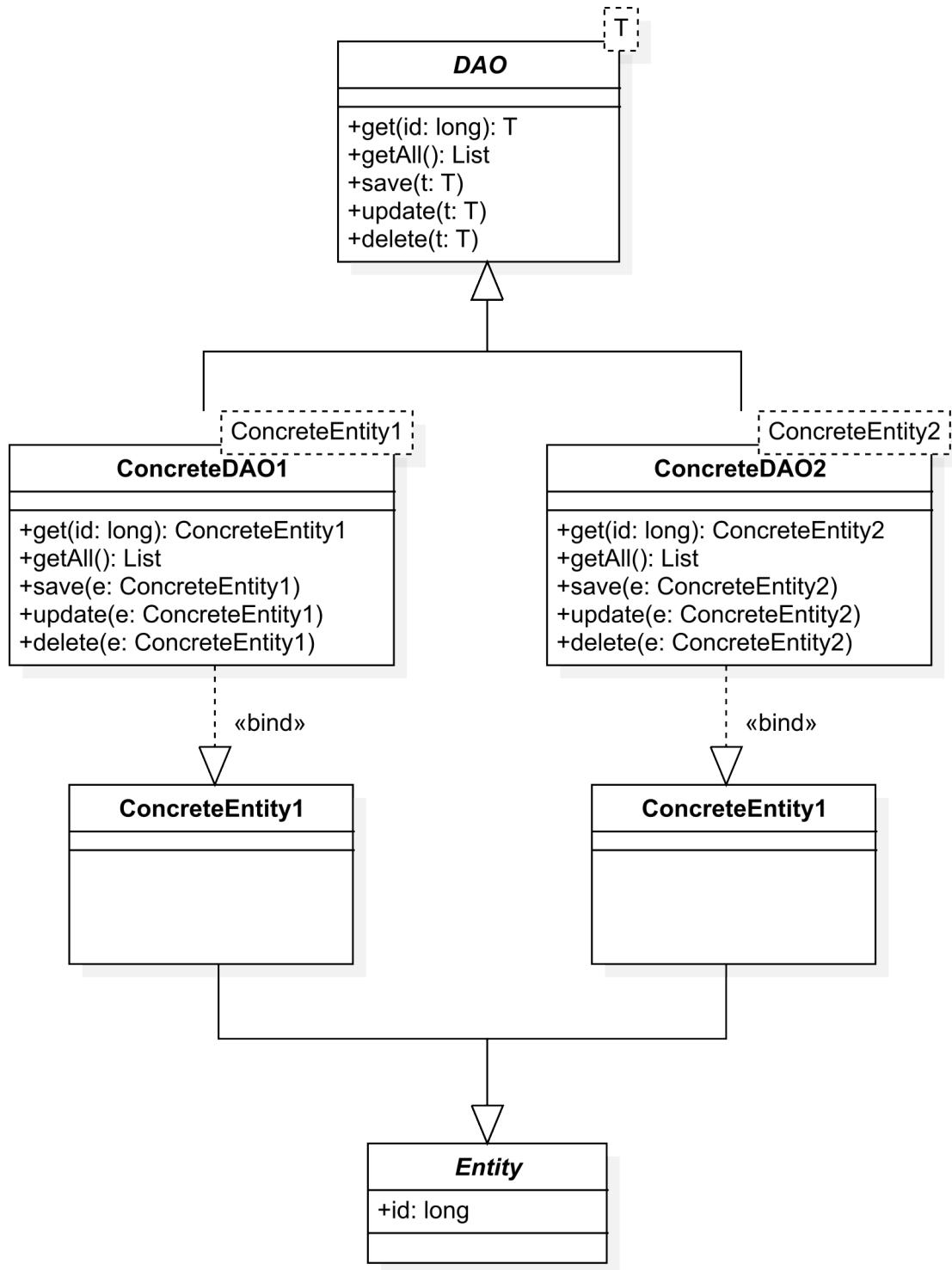


Figura 10: diagramma delle classi del design pattern DAO

Il DAO (Data Access Object) è un pattern strutturale che permette di disaccoppiare il layer di business logic da quello che si occupa di rendere gli oggetti persistenti nel database. I DAO espongono alla business logic un'API per effettuare le operazioni di CRUD. Ciò permette di rispettare il principio di singola responsabilità.

Nel nostro caso, è stato prevista una classe base astratta e generica che definisce l'implementazione di default per le operazioni CRUD di base. Nella maggior parte dei casi questa implementazione è adeguata per tutti i DAO concreti. Nei casi in cui erano necessari metodi di accesso ai dati più specifici, ad esempio filtraggio per attributi che non siano l'ID, essi sono stati implementati nel rispettivo DAO concreto.

3.5.2 State

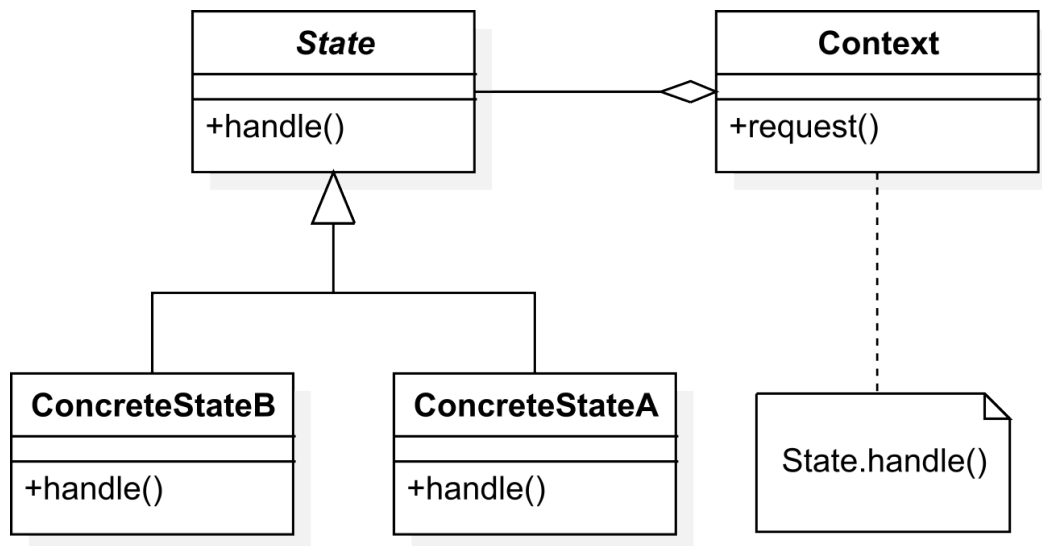


Figura 11: State pattern

Lo *State* è un pattern comportamentale, mostrato in figura 11, che permette ad un oggetto di cambiare comportamento sulla base di un proprio stato. Ha tre componenti principali:

- **State**: classe base astratta rappresentante uno stato generico dell'oggetto. Espone un metodo *handle()* implementato dalle classi derivate
- **ConcreteState**: classe concreta derivata da *State* rappresentante uno specifico stato che l'oggetto può assumere
- **Context**: rappresenta l'oggetto il cui comportamento dipende dallo stato che possiede; ha un riferimento a *State* e un metodo *request()*, per l'aggiornamento dello stato

Quando un oggetto di tipo *Context* deve mostrare un determinato comportamento, viene chiamato il metodo *request()* che internamente chiama *State.handle()* e delega tale responsabilità allo stato corrente dell'oggetto, di tipo *ConcreteState*. Se in un qualsiasi momento avviene una transizione ad un altro *ConcreteState*, il *Context* aggiornerà il proprio riferimento e ciò si riflette in un cambiamento dello stato interno.

Lo state pattern si può ritrovare nelle classi rappresentanti gli *Handler* all'interno del package *businessLogic* (figura 8). Infatti, la classe *NavigationManager* può essere vista come un *Context* il cui comportamento dipende dall'*Handler* "attivo" nell'applicazione (*ConcreteState*); dettagli maggiori sull'implementazione sono presenti nella sezione 4.2.3

3.6 Diagramma di sequenza

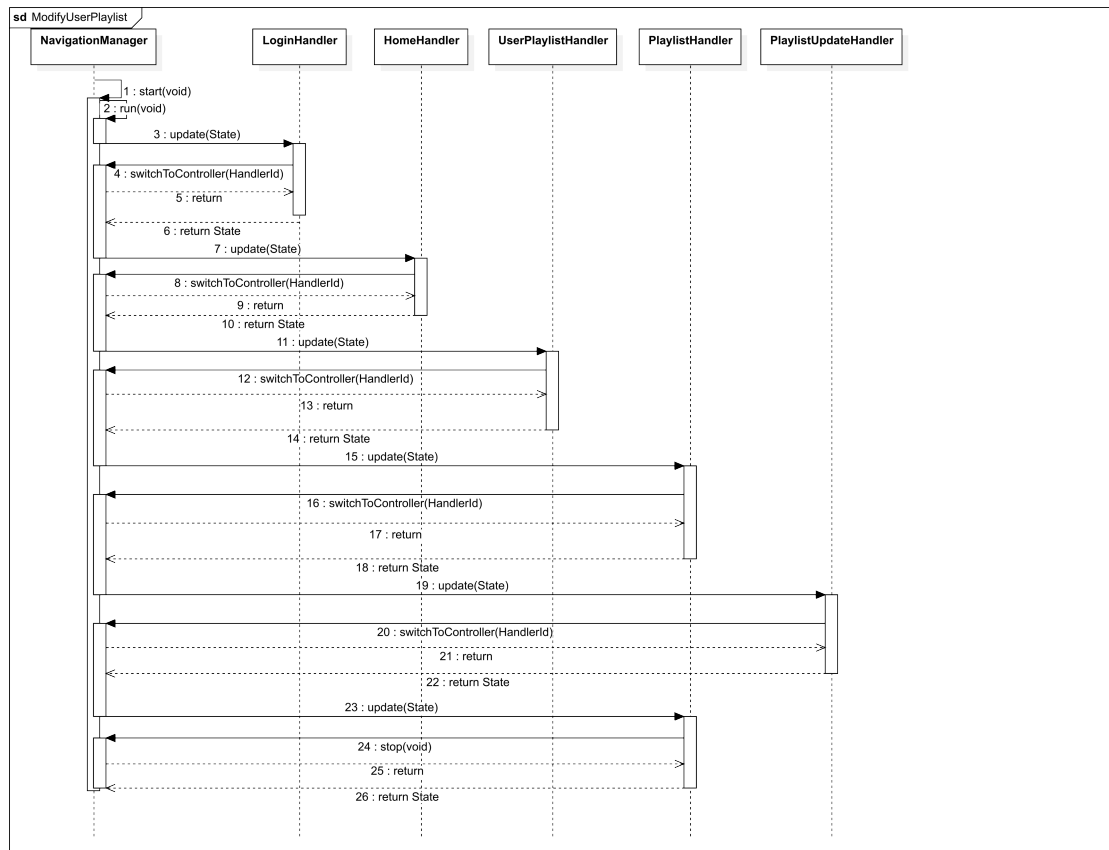


Figura 12: diagramma di sequenza relativo alla modifica della playlist di un utente

Un **diagramma di sequenza** mostra la successione temporale delle chiamate dei metodi delle classi dell'applicazione in un determinato scenario di utilizzo. La figura 12 rappresenta il diagramma di sequenza relativo allo scenario in cui un utente decide di modificare una delle proprie playlist.

4 Implementazione

4.1 Domain model

4.1.1 BaseEntity

Classe base astratta che garantisce la presenza di un ID in tutte entità concrete.

4.1.2 BaseUser

Classe base per Customer e Artist che contiene le credenziali di accesso.

4.1.3 Customer

Estende il BaseUser aggiungendo una lista di artisti seguiti.

4.1.4 Artist

Estende il BasUser aggiungendo il nome d'arte, la biografia e il numero di follower.

4.1.5 Track

Rappresenta una traccia audio che può essere aggiunta alla coda di riproduzione.

4.1.6 DurationConverter

Implementa l'interfaccia AttributeConverter di Jakarta per convertire la durata delle Track dal tipo Duration a un tipo supportato dal DBMS.

4.1.7 Song

Concretizzazione di Track che rappresenta una canzone.

4.1.8 Podcast

Concretizzazione di Track che rappresenta un podcast.

4.1.9 Playlist

Rappresenta una playlist generica ed espone i metodi per aggiungere o rimuovere delle track in testa oppure in coda.

4.1.10 SongPlaylist

Rappresenta una playlist a cui possono essere aggiunte soltanto canzoni.

4.1.11 PodcastPlaylist

Rappresenta una playlist a cui possono essere aggiunti soltanto podcast.

4.1.12 Album

Incapsula una playlist di cui espone selettivamente soltanto i metodi di lettura, in modo da rendere immutabile la lista di canzoni dopo che è stata creata.

4.1.13 PlaybackQueue

Rappresenta la coda di riproduzione e permette di aggiungere in testa ed in coda oltre ad eliminare in testa. Non è persistente.

4.1.14 TrackPlaysCount

Classe astratta che associa un Customer a ogni Track che ha ascoltato mantenendo un conteggio del numero di riproduzioni. Questa informazione serve per il calcolo delle tracce suggerite.

4.1.15 SongPlaysCount

Concretizzazione di TrackPlaysCount che conta quante volte un utente ha riprodotto ogni canzone

4.1.16 PodcastPlaysCount

Concretizzazione di TrackPlaysCount che conta quante volte un utente ha riprodotto ogni canzone

4.1.17 TrackPlaysCountListener

Trigger che mantiene aggiornato il numero totale di riproduzioni di una traccia. Viene registrato tramite delle annotazioni fornite dalla JPA e gestito in maniera automatica da Hibernate.

4.2 Business logic

4.2.1 Handler

Classe base astratta che rappresenta un *Handler* generico e corrisponde alla classe *State* dello State pattern (sezione 3.5.2). Ha un metodo *update()* per aggiornare ciò che viene visualizzato da linea di comando come finestra corrente dell'applicazione, definito all'interno delle classi derivate.

4.2.2 Session

Classe che rappresenta una sessione dell'applicazione; permette ai vari *Handler* di comunicare tra loro dati come l'utente che ha effettuato l'accesso, la canzone o la playlist selezionata

4.2.3 NavigationManager

Gestisce la navigazione tra le pagine passando il controllo ai vari handler; rappresenta il *Context* dello State pattern (sezione 3.5.2). Ha uno stack di riferimenti a *Handler* chiamato *states*, attraverso il quale viene tenuta traccia della finestra visualizzata in un certo momento. Il metodo *switchToController()* (figura 13) effettua un *push* sullo stack dell'*Handler* a cui deve essere passato il controllo. *NavigationManager.switchToController()* si trova all'interno dell'*update()* di ciascun *Handler*, metodo che viene eseguito all'interno di un loop nella *run()* del *NavigationManager* (figura 14)

```
public void switchToController(HandlerId id) {
    if (handlers.containsKey(id)) {
        states.push(handlers.get(id));
    } else {
        throw new IllegalArgumentException("Controller not present");
    }
}
```

Figura 13: metodo *switchToController()* di *NavigationManager*

```
public void run() {
    while (!states.empty()) {
        session = states.peek().update(session);
    }
}
```

Figura 14: metodo *run()* di *NavigationManager*

4.2.4 AlbumViewHandler

Permette di visualizzare o riprodurre un album.

4.2.5 ArtistInfoHandler

Mostra le informazioni salienti di un artista.

4.2.6 HomeHandler

Contiene la schermata di ingresso e instrada l'utente verso le varie pagine.

4.2.7 LoginHandler

Permette all'utente di effettuare l'accesso con un nome utente e una password o eventualmente passare alla schermata di registrazione.

4.2.8 PlaybackHandler

Gestisce la coda di riproduzione e permette all'utente di visualizzare i brani contenuti in essa. Utilizza un *thread* che viene fatto partire all'apertura della coda di riproduzione, e che si occupa della visualizzazione della *progress bar* della canzone in cima alla coda. In figura 15 viene mostrato il metodo *run()* del thread della coda di riproduzione.

```
public void run() {
    int i=0;
    try {
        while (true) {
            if (i <= totalSteps && !skip) {
                if (!paused){
                    Thread.sleep(100);
                    printProgressBar(i, totalSteps);
                    i++;
                }
                else{
                    Thread.sleep(100);
                    printProgressBar(i, totalSteps);
                }
            }
            else{
                skip = false;
                Thread.sleep(100);
                i=0;
                Track newTrack = session.getQueue().getNextSong();
                session.setPlayingTrack(newTrack);
                this.totalSteps = (int) session.getPlayingTrack().getDuration().toSeconds();
            }
        }
    } catch (Exception ignored){}
}
```

Figura 15: metodo *run()* del thread della coda di riproduzione

4.2.9 PlaylistHandler

Mostra all'utente i brani contenuti in una data playlist e permette ad esso di aggiungerla alla coda di riproduzione.

4.2.10 RegistrationHandler

Permette all'utente di registrarsi all'interno dell'applicazione con un nome utente ed una password.

4.2.11 SearchHandler

Gestisce la ricerca all'interno delle canzoni, dei podcast e degli artisti disponibili.

4.2.12 SuggestionsHandler

Mostra all'utente le tracce consigliate dall'applicazione sulla base dei suoi ultimi ascolti.

4.2.13 UserPlaylistsHandler

Mostra l'elenco delle playlist create dall'utente.

4.3 DAO

4.3.1 BaseDAO

Classe base generica del Data Access Object che definisce le operazioni CRUD di base per interfacciarsi col DBMS e ne fornisce un'implementazione di default. Metodi implementati:

- *get(long id)*: restituisce un oggetto di tipo T il cui ID nel database corrisponde a quello dato.
- *getAll()*: restituisce tutti gli oggetti di tipo T nel database.
- *save(T t)*: rende persistente l'oggetto dato.
- *update(T t)*: aggiorna nel database lo stato dell'oggetto.
- *delete(T t)*: elimina dal database l'oggetto.

I metodi per eseguire filtri sulla base di parametri specifici sono definiti nelle classi derivate (figura 9).

4.3.2 CustomerDAO

Permette di ottenere oggetti Customer dal database a partire dal loro username attraverso il metodo *getUserName()*.

4.3.3 ArtistDAO

Permette di ottenere oggetti Artist dal database a partire dal loro nome d'arte o username attraverso i metodi *getStageName()* e *getUserName()*.

4.3.4 SongDAO

Permette di ottenere oggetti Song dal database a partire da una parola chiave, attraverso il metodo *getByTitle()*, o un Artist, attraverso il metodo *getByArtist()*.

4.3.5 PodcastDAO

Permette di ottenere oggetti Podcast dal database a partire da una parola chiave (*getByTitle()*) o un Artist (*getByArtist()*).

4.3.6 SongPlaylistDAO

Permette di ottenere oggetti SongPlaylist dal database a partire dal titolo attraverso il metodo *getByTitle()*.

4.3.7 PodcastPlaylistDAO

Permette di ottenere oggetti PodcastPlaylist dal database a partire dal titolo (*getByTitle()*).

4.3.8 AlbumDAO

Permette di ottenere oggetti Album dal database a partire dal titolo, attraverso il metodo *getByTitle()*, o dall'autore, attraverso il metodo *getByArtist()*.

4.3.9 SongPlaysCountDAO

Permette di aggiornare nel database il conteggio delle riproduzioni di una canzone attraverso il metodo *incrementPlays()*.

4.3.10 PodcastPlaysCountDAO

Permette di aggiornare nel database il conteggio delle riproduzioni di un podcast attraverso il metodo *incrementPlays()*.

4.3.11 SuggestionDAO

Permette, dato un utente, di ottenere una lista di canzoni suggerite in base a cosa hanno ascoltato utenti con preferenze affini. Il sistema di suggerimenti propone ad ogni utente delle canzoni che potrebbe apprezzare cercando le canzoni più ascoltate da utenti "simili" a lui. Gli utenti sono considerati simili all'utente corrente se hanno ascoltato le sue canzoni più ascoltate. Il funzionamento dettagliato del processo è descritto dalla seguente query:

```
WITH Top10TracksUser1 AS (  
    SELECT TRACK  
    FROM TRACKPLAYSCOUNT  
    WHERE CUSTOMER = 'User1'  
    ORDER BY PLAYS DESC  
    LIMIT 10  
)  
,  
UsersWhoListenedTop10 AS (  
    SELECT DISTINCT CUSTOMER  
    FROM TRACKPLAYSCOUNT  
    WHERE TRACK IN (SELECT TRACK FROM Top10TracksUser1)  
)  
,  
TopTracksByUsers AS (  
    SELECT TRACK, SUM(PLAYS) AS total_plays  
    FROM TRACKPLAYSCOUNT  
    WHERE CUSTOMER IN (SELECT CUSTOMER FROM UsersWhoListenedTop10)  
    GROUP BY TRACK  
)  
SELECT TRACK  
FROM TopTracksByUsers  
ORDER BY total_plays DESC  
LIMIT 10;
```

Figura 16: Query SQL che restituisce le canzoni suggerite all'utente

5 Test

5.1 Tipologie di test effettuati e organizzazione

Per poter garantire il corretto funzionamento di *Swetify* sono stati effettuate due tipologie di test:

- **test di integrazione:** verificano il comportamento che le componenti dell'applicazione mostrano quando interagiscono tra di loro; nello specifico, vengono testate le operazioni che i DAO utilizzano per interagire con il database di sistema
- **test funzionali:** verificano il comportamento delle singole componenti dell'applicazione in un possibile scenario reale in cui quest'ultima viene utilizzata. Tra i test funzionali rientrano, per esempio, quelli relativi alla navigazione tra una schermata e l'altra dell'applicazione

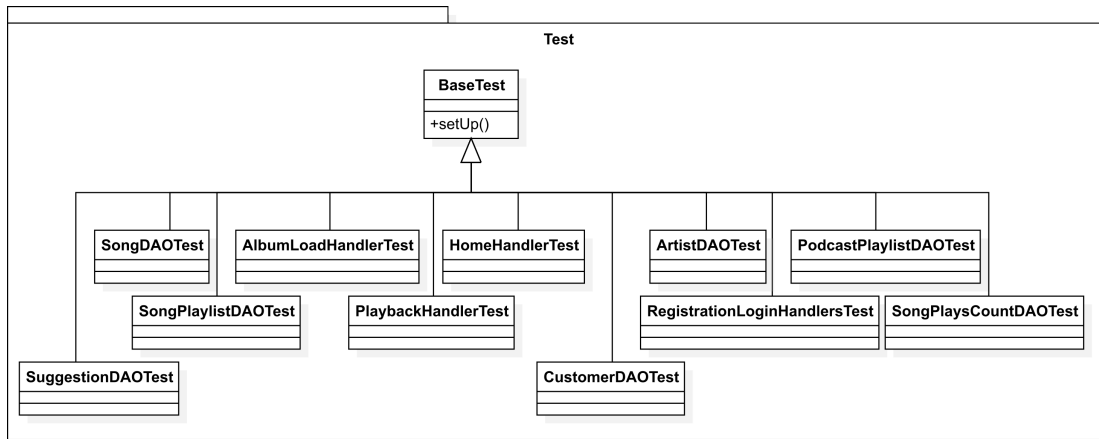


Figura 17: struttura del package *Test*

In figura 17 viene mostrata l'organizzazione delle classi di test, contenute nel package *Test*. Come si può notare, è presente una classe base ***BaseTest*** contenente le funzionalità comuni a tutte le altre classi, come la funzione ***setUp()***; questa funzione serve per ripulire il database tra un test e l'altro, motivo per il quale viene eseguita prima di ciascun test.

```

@BeforeEach
public void setUp(){
    super.setUp();
    songDatabase = new SongDAO();

    song1.setTitle("title1");
    song1.setDuration(Duration.ofSeconds(10));
    song2.setTitle("title2");
    song2.setDuration(Duration.ofSeconds(20));
    song3.setTitle("title3");
    song3.setDuration(Duration.ofSeconds(15));
    songDatabase.save(song1);
    songDatabase.save(song2);
    songDatabase.save(song3);
}
  
```

Figura 18: esempio di implementazione di *setUp()* (in questo caso relativa a *SongDAOTest*)

Come si può vedere in figura 18, la *setUp()* della classe base viene chiamata all'inizio della *setUp()* delle classi derivate, seguita dall'inizializzazione del database e degli oggetti necessari per l'esecuzione dei test.

5.2 Test di integrazione

Di seguito vengono descritti i metodi presenti nelle classi relative ai test di integrazione; ogni metodo consiste nel verificare il comportamento di una specifica query, scritta in linguaggio SQL, per l'accesso al database.

5.2.1 SongDAOTest

- ***testGet()***: verifica che *SongDAO.getByTitle()* e *SongDAO.get()* restituiscano lo stesso risultato, confrontando i titoli delle canzoni (oggetti di tipo *Song*) restituite
- ***testGetByTitle()***: prima viene verificata la presenza di alcuni oggetti di tipo *Song* salvati precedentemente sul database, attraverso *SongDAO.getByTitle()*; successivamente, viene creato un nuovo oggetto *song* di tipo *Song* senza però salvarlo, e viene verificata l'assenza di risultati restituiti da *SongDAO.getByTitle(song.getTitle())*

- ***testGetAll()***: verifica che *SongDAO.getAll()* restituisca il giusto numero di risultati, sulla base di quanti oggetti di tipo *Song* sono presenti nel database

5.2.2 ArtistDAOTest

- ***testGet()***: verifica che *ArtistDAO.getByStageName()* e *ArtistDAO.get()* restituiscano lo stesso risultato, confrontando nome, numero di seguaci e biografia degli artisti (oggetti di tipo *Artist*) restituiti
- ***testGetByUserName()***: prima viene verificata la presenza di alcuni oggetti di tipo *Artist* salvati precedentemente sul database, attraverso *ArtistDAO.getByUserName()*; successivamente, viene creato un nuovo oggetto *artist* di tipo *Artist* senza però salvarlo, e viene verificata l'assenza di risultati restituiti da *ArtistDAO.getByUserName()*
- ***testByStageName()***: dopo aver creato e salvato due nuovi oggetti di tipo *Artist* con lo stesso nome di uno degli artisti presenti nel database, verifica che *ArtistDAO.getByStageName()* restituisca il giusto numero di risultati
- ***testGetAll()***: prima verifica che *ArtistDAO.getAll()* restituisca il giusto numero di risultati, sulla base di quanti oggetti di tipo *Artist* sono presenti nel database, e successivamente verifica che tali risultati siano corretti, controllando nome, numero di seguaci e biografia per ciascuno di essi

5.2.3 CustomerDAOTest

- ***testGet()***: verifica che *CustomerDAO.getByStageName()* e *CustomerDAO.get()* restituiscano lo stesso risultato, confrontando nome utente e password dei customer (oggetti di tipo *Customer*) restituiti
- ***testGetByUserName()***: prima viene verificata la presenza di alcuni oggetti di tipo *Customer* salvati precedentemente sul database, attraverso *CustomerDAO.getByUserName()*; successivamente, viene creato un nuovo oggetto *user* di tipo *Customer* senza però salvarlo, e viene verificata l'assenza di risultati restituiti da *CustomerDAO.getByUserName(user.getUsername())*
- ***testGetAll()***: prima verifica che *CustomerDAO.getAll()* restituisca il giusto numero di risultati, sulla base di quanti oggetti di tipo *Customer* sono presenti nel database, e successivamente verifica che tali risultati siano corretti, controllando nome utente e password per ciascuno di essi

5.2.4 SuggestionDAOTest

- ***testSuggestions()***: verifica che la query effettuata dalla funzione *getTopSongsBySimilarUsers()* della classe *SuggestionDAO* sia conforme alla logica desiderata calcolando in maniera indipendente il risultato atteso grazie alle funzioni *getUserTopTen()*, *getCustomersWhoListenedTopTenSongs()*, *getTopTenSongsByTopTenListeners()*.

5.2.5 SongPlaylistDAOTest

- ***testGet()***: si assicura che la Playlist restituita da *getByTitle()* coincida con quella restituita da *get()* nel caso in cui sia presente solo una playlist col nome utilizzato.
- ***testGetAll()***: verifica che la lunghezza della lista restituita da *getAll()* sia conforme a quanto atteso;
- ***testGetByTitle()***: verifica che la lunghezza della lista restituita da *getByTitle()* sia conforme a quanto atteso;

5.2.6 SongPlaysCountDAOTest

- ***testIncrementPlays()***: verifica che, dopo aver chiamato il metodo *incrementPlays()* di *SongPlaysCount*, il conteggio del numero di riproduzioni di una canzone venga effettivamente incrementato.

5.3 Test funzionali

Come per i test di integrazione, di seguito vengono descritti i metodi presenti nelle classi relative ai test funzionali; nella *setUp()* di ciascun metodo viene inizializzato il *NavigationManager* e viene chiamato il metodo *pushHandler()* specificando l'identificativo dell'handler coinvolto nei test all'interno di una certa classe. Lo scenario di utilizzo dell'applicazione è rappresentato da un oggetto di tipo *ByteArrayInputStream* contenente la sequenza di input da testare; un esempio di sequenza testata è mostrata in figura 19

```
ByteArrayInputStream input = new ByteArrayInputStream("1\nstio\n1\nstio\nstio\n12\n13\n2\n2\n3\n".getBytes());
System.setIn(input);
```

Figura 19: esempio di sequenza di input testata

5.3.1 HomeHandlerTest

- ***testHomeSearch()***: verifica la navigazione dalla homepage alla pagina di ricerca, confrontando l'identificativo dell'ultimo Handler che ha chiamato l'*update()* con quello di *SearchHandler*
- ***testHomeViewPlaylists()***: verifica la navigazione dalla homepage alla pagina di visualizzazione delle playlist di un utente, confrontando l'identificativo dell'ultimo Handler che ha chiamato l'*update()* con quello di *UserPlaylistsHandler*
- ***testHomeSuggestions()***: verifica la navigazione dalla homepage alla pagina di visualizzazione dei suggeriti, confrontando l'identificativo dell'ultimo Handler che ha chiamato l'*update()* con quello di *SuggestionHandler*

5.3.2 PlaybackHandlerTest

- ***testInitialPlaybackState()***: verifica che lo stato iniziale della coda di riproduzione sia corretto. In particolare, dopo aver inserito una canzone *song* nella coda di riproduzione, verifica che il thread in esecuzione rappresentante la coda stessa sia attivo (con il metodo *Thread.isAlive()*) e che *song* sia in pausa
- ***testPlayPauseFunctionality()***: verifica la funzionalità di pausa di una canzone, controllando se il thread relativo alla coda sia in esecuzione (se una canzone è in riproduzione) oppure no
- ***testSkipFunctionality()***: verifica la funzionalità di skip di una canzone
- ***testTrackSwitching()***: verifica che la coda di riproduzione passi correttamente da una canzone in riproduzione alla successiva; in particolare, controlla che la successiva canzone nella coda sia in riproduzione dopo che quella corrente è terminata

5.3.3 RegistrationLoginHandlersTest

- ***testRegistrationCustomer()***: dopo aver aggiunto un utente (oggetto di tipo *Customer*) al database, verifica che sia effettivamente presente attraverso *CustomerDAO.getByUsername()* e che ce ne sia uno solo
- ***testRegistrationArtist()***: dopo aver aggiunto un artista (oggetto di tipo *Artist*) al database, verifica che sia effettivamente presente attraverso *ArtistDAO.getByUsername()* e che ce ne sia uno solo
- ***testLoginCustomer()***: dopo aver aggiunto un utente (oggetto di tipo *Customer*) al database e dopo aver fatto registrazione e login, verifica che nome e password di tale utente corrispondano a quelli memorizzati nella sessione corrente
- ***testLoginArtist()***: dopo aver aggiunto un artista (oggetto di tipo *Artist*) al database e dopo aver fatto registrazione e login, verifica che nome e password di tale artista corrispondano a quelli memorizzati nella sessione corrente

5.3.4 AlbumLoadHandlerTest

- *testSuccessfulAlbumLoad()*: dopo aver creato un album contenente una sola canzone, verifica che tale canzone sia presente e che sia effettivamente l'unica al suo interno