EMOTIONAL SONGS

MANUALE TECNICO

SOMMARIO

INTRODUZIONE

SCOPO DELL'APPLICAZIONE

L'applicazione permette all'utente di svolgere operazioni specifiche a seconda dello stato in cui si trova. La funzione principale del servizio è quella di poter **ricercare da una repository di canzon**i per le quali è possibile **visualizzare la media degli stati emozionali** ed, eventualmente, **inserire le emozioni suscitate dall'ascolto del brano musicale** a seconda di un punteggio in scala (da 0...5).

Inoltre, è possibile **scrivere dei commenti personali** per ognuna delle 9 emozioni disponibili per il brano.

Vi sono due stati rilevanti:

- UTENTE NON REGISTRATO: qualsiasi utente che apra l'applicazione è in grado di consultare la lista di canzoni presenti. Per ciascuna di esse è possibile anche consultare la media dei voti che gli utenti hanno lasciato per ognuno dei 9 stati emozionali:
- **Amazement, Tenderness, Nostalgia, Solemnity, Calmness, Power, Joy, Tension, Sadness** attraverso un grafico a barre (istogramma).

- UTENTE REGISTRATO E LOGGATO: l'utente che si è registrato ed ha eseguito l'accesso può sfruttare anche altre funzioni proposte dall'applicazione:
- Creare/Rinominare/Eliminare una o più playlist di canzoni
- Aggiungere/Eliminare una canzone alla/dalla playlist precedentemente creata
- Aggiungere/Aggiornare un voto (da zero a cinque) per ogni stato emozionale e/o lasciare relativi commenti
- Consultare il contenuto delle playlist create

REQUISITI DI SISTEMA

Per l'utilizzo del programma, è necessaria una connessione ad Internet per collegarsi ai server di Emotional Songs.

Le prestazioni del programma potrebbero essere influenzate dalle prestazioni del proprio computer e dalla velocità della propria connessione a internet, maggiormente da quest'ultima.

STRUTTURA DELL'APPLICAZIONE

FRAMEWORK ED AVVIO

- Si tratta di un'applicazione **Spring Boot**, che semplifica la creazione e l'avvio di **applicazioni Java basate su Spring Framework**.
- Il punto di avvio dell'applicazione si trova nella classe **src/main/java/emotionalsongs/Application.java**. Questa è segnata da @SpringBootApplication, un'annotazione che consente di configurare e avviare l'applicazione in modo efficiente.
- Il cuore dell'applicazione è rappresentato quindi dal metodo main, che sfrutta la classe **SpringApplication** per avviare l'intero contesto dell'applicazione Spring. Ciò include la gestione della configurazione, l'inizializzazione degli oggetti e la creazione di un ambiente pronto per l'esecuzione, mentre le dipendenze sono direttamente gestite da **Maven**. Quest'ultimo semplifica inoltre le gestione delle risorse, build, plugin, ...
- Per semplificare la creazione dell'applicazione web interattiva, è stato usato **Vaadin**. Esso è un framework per lo sviluppo di interfacce utente web basate su Java, che forniscono una serie di componenti e strumenti che consentono agli sviluppatori di creare interfacce utente dinamiche senza dover scrivere direttamente il codice **HTML, CSS o JavaScript**. Le componenti rappresentano elementi dell'interfaccia utente, come bottoni, campi di testo, tabelle e pannelli. Ognuno di essi è un'istanza di una classe specifica fornita dal framework, e può essere personalizzata attraverso metodi e attributi. Alcune di queste sono presenti nella seguente cartella src/main/java/emotionalsongs/components/appnav ed in particolare si occupano in modo flessibile della gestione della GUI e della navigazione nella web app.

ORGANIZZAZIONE DEL CODICE

L'implementazione principale dell'applicazione si trova nel percorso '**src/main/java/emotionalsongs**'.
All'interno di questa directory principale, sono presenti ulteriori tre **subpackage**:

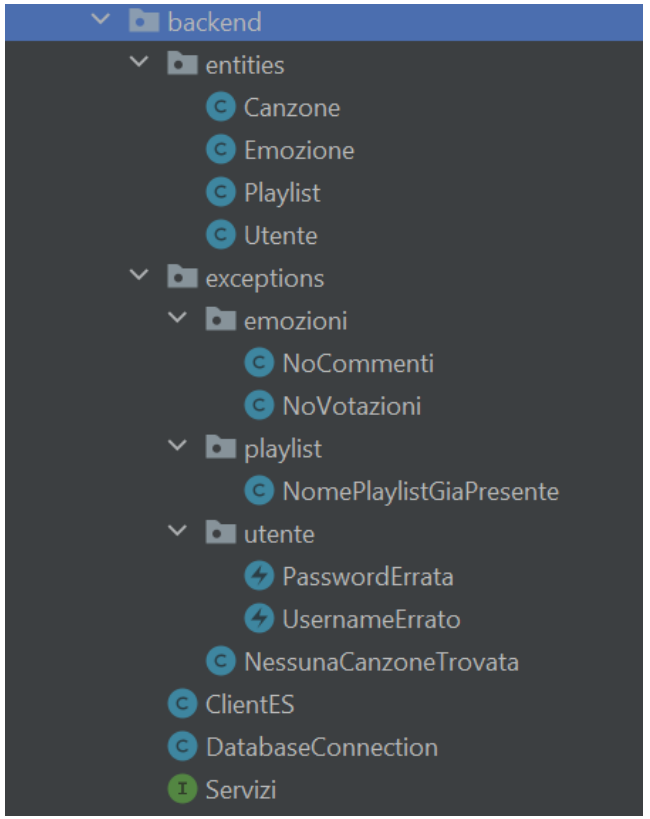
1. **backend**: contiene il nucleo funzionale dell'applicazione (LOGICA APPLICATIVA) ad esempio la gestione delle playlist, delle canzoni, delle valutazioni emotive, ecc.
2. **components.appnav**: contiene componenti specifici dell'applicazione legati alla navigazione e all'interfaccia utente (GUI).
3. **views**: contiene le varie viste (Frame e Dialog) dell'applicazione (GUI).

Inoltre, il punto di ingresso dell'applicazione si trova nella classe '**Application.java**', che è situata nello stesso percorso. Questa classe contiene il metodo principale 'main', da cui viene avviata l'esecuzione dell'intera applicazione.

BACKEND

Esso a sua volta è strutturata nel seguente modo:

- Cartella entities
- Cartella exceptions
- Classe ClientES
- Classe DatabaseConneciton
- Interfaccia Servizi



- **entites**: contiene classi, con i loro attributi e metodi, che rappresentano entità specifiche. Per ognuna è presente l'annotazione @Entity che fa parte del framework Spring. Questa viene usata per indicare che la classe rappresenta una tabella all'interno del database relazionale. Ogni istanza di questa classe corrisponde ad una riga nella tabella associata. Le entità sono:

1. Canzone

- Rappresenta una canzone con gli attributi: ID (int) , anno (int) , codice (String), titolo (String) e artista (String).
- Fornisce costruttori per creare istanze.
- Metodi getter e setter per accedere e modificare gli attributi.

2. Emozione

- Rappresenta un'emozione associata ad una canzone, in realtà questa entità rappresenta un sottoinsieme dei campi che formano la relazione 'Emozioni' all'interno del DB (si veda la sezione relativa a 'Scelte Progettuali: Database').
- Contiene gli attributi: name (nome dell'emozione in formato String) , commento (String) e score (double).
- Fornisce costruttori e metodi getter e setter per accedere/modificare gli attributi.
- Il metodo toString() per restituire una rappresentazione testuale dell'emozione.

3. Playlist

- Rappresenta una playlist con i seguenti attributi: ID (int) , titolo (String) e username (utente associato alla playlist in formato int).
- Fornisce costruttori per creare istanze ed i metodi getter e setter per accedere/modificare gli attributi.
- Rappresenta le playlist create dagli utenti dell'applicazione, permettendo di gestire e organizzare le playlist personali.

4. Utente

- Rappresenta un utente dell'applicazione con i seguenti attributi: ID (int), username (String), password (String) e nome (String).
- Fornisce costruttori per creare istanze ed i metodi getter e setter per accedere/modificare gli attributi.

- **expections**: contiene le varie eccezioni che l'app può lanciare tra cui…
  - NoCommenti : viene lanciata quando non ci sono commenti disponibili per alcuna delle emozioni della canzone specificata.
  - NoVotazioni: viene sollevata quando non ci sono votazioni disponibili, di cui calcolare la media, per le emozioni della canzone specificata.
  - NomePlaylistGiaPresente: viene lanciata quando si cerca di creare una nuova playlist con un nome che è già stato assegnato ad un'altra playlist già esistente per il proprio account.
  - PasswordErrata: viene sollevata quando in fase di autenticazione la password inserita è sbagliata.
  - UsernameErrato: viene lanciata quando in fase di autenticazione l'username inserito è sbagliato.
  - NessunaCanzoneTrovata: viene sollevata quando in fase di ricerca non viene trovata alcuna canzone corrispondente ai parametri inseriti.

- Servizi: questa **interfaccia** definisce un insieme di operazioni (o servizi) che possono essere eseguiti all'interno dell'applicazione. Questi servizi coprono le varie **funzionalità dell'applicazione**, inclusi l'autenticazione, la gestione delle canzoni, delle playlist e delle emozioni associate ad un determinato brano. Viene garantita un'astrazione dei dettagli implementativi mostrando in modo chiaro le funzionalità dell'applicazione e permettendo una flessibilità per eventuali future modifiche al sistema.

- **DatabaseConnection**: questa classe permette l'accesso e la gestione delle connessioni al database utilizzando le informazioni di connessione definite nel file di configurazione '**application.properties**' in modo tale che le operazioni di accesso ai dati siano affidabili e consistenti.

Le funzioni principali sono:

  - Inizializzazione della connessione tramite il metodo **initConnection()** che utilizza i valori di connessione letti dal file di configurazione precedentemente citato per creare un oggetto di tipo **Jdbc3PoolingDataSource** che rappresenta la connessione al database.
  - Restituzione della connessione tramite il metodo pubblico **getConnection()** che restituisce un oggetto di tipo Connection, che può essere utilizzato per interagire con il database. Se la connessione non è ancora stata inizializzata, il metodo **initConnection()** viene chiamato per inizializzarla.
  - Lettura delle informazioni di connessione tramite il metodo privato **setConnection()** che legge le informazioni di connessione dal file di configurazione attraverso operazioni di input/output (I/O) riga per riga ed identifica le linee che contengono le informazioni di URL, nome utente e password del database.
  - Gestione delle eccezioni IOException che possono verificarsi durante la lettura del file di configurazione.
  - L'istanza della classe è univoca e pertanto riprende il concetto del funzionamento del **pattern singleton**.

- **ClientES**: questa classe, che implementa l'interfaccia Servizi, realizza una serie di metodi per la gestione delle varie funzionalità messe a disposizione dall'applicazione, inclusa l'aggiunta e la rimozione di canzoni, la valutazione delle emozioni associate alle canzoni e la gestione dell'account utente. Le operazioni coinvolgono l'interazione con il database (con cui viene instaurata una connessione attraverso un oggetto di tipo **DatabaseConnection** e che viene opportunamente chiusa dopo ogni richiesta) tramite query SQL e la manipolazione di oggetti che rappresentano entità.

- I metodi principali della classe **ClientES** sono:

  - searchSong(String titoloDaCercare, String autoreDaCercare, Integer year): questo metodo rappresenta un'implementazione per la ricerca di canzoni nel database, consentendo di effettuare ricerche mirate in base ai seguenti parametri: **il titolo della canzone, l'autore e l'anno di pubblicazione**. La funzione costruisce una query SQL e cerca corrispondenze sia nel titolo che nell'autore, fornendo una maggiore flessibilità nelle ricerche (grazie alla clausola LIKE) ed adotta il **binding parametrizzato** (vengono utilizzati '?', segnaposto utili all'inserimento dinamico i valori). Inoltre, se viene specificato un anno, la query viene estesa per includere un filtro per quell'anno. Per evitare un'eccessiva quantità di risultati, è stato impostato un limite massimo di 300 canzoni restituite. Una volta costruita la query, il metodo si connette al database utilizzando l'oggetto 'dbConn' ed imposta i parametri di ricerca. Successivamente, la query viene eseguita tramite '**executeQuery()**' ottenendo il ResultSet. Quest'ultimo viene iterato su tutte le righe ottenute e vengono man mano creati oggetti 'Canzone' per ciascuna corrispondenza trovata e salvati nella lista 'result'. Nel caso in cui la lista dei risultati sia vuota, il metodo genera l'eccezione 'NessunaCanzoneTrovata'. Durante tutto il processo, vengono gestite le possibili eccezioni derivanti dalla connessione al database, dall'esecuzione della query ed infine viene chiusa la connessione assicurando un comportamento affidabile.
  - registrazione(String nome, String cognome, String indirizzo, String codiceFiscale, String email, String username, String password): questo metodo implementa la funzionalità di **registrazione di un nuovo utente** nel sistema. Accetta diversi parametri come input, tra cui nome, cognome, indirizzo, codice fiscale, email, username e password. Quest'ultima, prima di essere salvata nel database, viene hashata utilizzando la funzione **BCrypt.hashpw()** della libreria **jBCrypt** per garantire la sicurezza delle informazioni sensibili. La registrazione avviene attraverso l'esecuzione di una query SQL che inserisce i dettagli dell'utente nella **tabella 'User' del database**. Essa viene costruita con il comando **INSERT INTO** ed una volta impostati tutti i parametri, è eseguita tramite **executeUpdate()**. Sono inoltre gestite le possibili eccezioni legate alla connessione al database e all'esecuzione della query.
  - login(String userid, String password): questo metodo implementa la **funzionalità di accesso** degli utenti al sistema. Il metodo inizia formulando una **query SQL** per cercare un record corrispondente nella **tabella 'User' del database**. Se questo viene trovato, allora vengono recuperate alcune informazioni come l'ID utente, username, la password hashata ed il nome dell'utente. Queste sono quindi utilizzate per creare un oggetto di tipo Utente. Successivamente, il metodo verifica se il nome utente fornito (userid) corrisponde al nome utente recuperato dal database e se la password fornita (password) è valida rispetto alla password hashata. Questo viene fatto utilizzando la funzione **BCrypt.checkpw()** per confrontare la password fornita con la versione hashata nel database. Se entrambi i controlli vanno a buon fine, il metodo restituisce l'oggetto Utente corrispondente all'utente che ha effettuato l'accesso. Se invece il nome utente non corrisponde, viene lanciata l'eccezione UsernameErrato. Analogamente se la password non è valida, viene lanciata l'eccezione PasswordErrata. In caso di problemi di connessione al database o di esecuzione della query, sono gestite eventuali eccezioni ed infine viene chiusa la connessione con il database.

- addPlaylist(String titolo, int userId): questo metodo è responsabile della **creazione di una nuova playlist** nel sistema. Inizialmente viene definita una stringa di query SQL per inserire i dettagli della nuova playlist nella **tabella 'Playlist' del database**. Successivamente, viene istaurata una connessione al database utilizzando l'oggetto '**dbConn**' che rappresenta l'istanza della classe '**DbConnection**'. Viene quindi eseguito il '**Prepared Statement**' (i segnaposto vengono sostituiti coi valori effettivi del titolo e userId della segnatura del metodo) utilizzando il metodo '**executeUpdate()**', che procede all'operazione di inserimento delle informazioni nella tabella 'Playlist'. Il risultato che si ottiene dalla chiamata ad 'executeUpdate()' viene restituito ed indica quante righe sono state modificate o inserite, quindi tipicamente sarà 1 per indicare che una nuova playlist è stata creata con successo. Vengono inoltre chiuse correttamente tutte le risorse utilizzate per la connessione e l'esecuzione della query.
- addBraniPlaylist(int playlistId, ArrayList<Canzone> braniSelezionati): addBraniPlaylist(int playlistId, ArrayList<Canzone> braniSelezionati): questo metodo permette di **aggiungere una lista di brani selezionati ad una playlist** specificata. Vengono innanzitutto dichiarate le variabili di connessione e di statement ('**conn**' e '**stmt**') che sono utilizzate per interagire con il database. Successivamente, viene creato un oggetto '**StringBuilder**' che viene utilizzato per costruire la query SQL di inserimento dei brani nella **tabella 'CanzoniPlaylist' del database**. Questa query viene costruita iterando l'ArrayList di brani selezionati ('braniSelezionati' passato come parametro nel metodo). Da ciascun oggetto 'Canzone' viene estratto l'ID della canzone, che viene poi aggiunto alla query insieme al playlist ID tramite il metodo '**append()**' dell'oggetto 'query'. Inoltre, si controlla se l'elemento corrente è l'ultimo nella lista di brani selezionati. Se è l'ultimo, la query viene terminata con un punto e virgola (';'), altrimenti con una virgola (';'). Dopo la costruzione della query, il metodo stabilisce una connessione al database utilizzando l'oggetto '**dbConn**' che rappresenta l'istanza della classe '**DbConnection**'. Viene quindi preparata un'istruzione SQL utilizzando la query costruita ('**query.toString()**') e l'istruzione preparata viene eseguita utilizzando '**executeUpdate()**' per inserire i brani selezionati. Vengono gestite le eventuali eccezioni del tipo 'SQLException'. Infine, il metodo chiude correttamente tutte le risorse utilizzate per la connessione e l'esecuzione della query. Questa chiusura delle risorse avviene nell'istruzione 'finally', garantendo che esse siano rilasciate anche in caso di eccezioni.
- insEmoBranoPlaylist(int playlistId, int songId, List<Emozione> emozioni): questo metodo consente di **inserire le emozioni associate ad una determinata canzone** all'interno di una playlist. L'implementazione inizia costruendo una query SQL per l'inserimento dei dati nella **tabella 'Emozioni'.** Successivamente, il metodo itera attraverso ogni oggetto 'Emozione' nella lista 'emozioni', passata come parametro, e per ogni emozione viene estratto il punteggio (score) e il commento associato. La query viene costruita in base alle condizioni seguenti:

  - Se l'indice dell'emozione nell'elenco non è 8 (cioè non è l'ultima emozione), vengono aggiunti il punteggio e il commento come valori **separati da virgola**, a meno che il punteggio non sia 0 (in tal caso, viene inserito 'null') o il commento non sia vuoto (anche in tal caso viene inserito 'null').

  - Se l'indice dell'emozione nell'elenco è 8 (cioè è l'ultima emozione), vengono aggiunti il punteggio e il commento come valori **separati da virgola, seguiti da un punto e virgola** o una parentesi chiusa, a seconda se il punteggio e il commento sono presenti o meno.

- Viene quindi stabilita la connessione al database, eseguita la query di inserimento ed infine vengono chiusi i relativi oggetti di connessione. In caso di anomalie durante il processo, viene sollevata un'eccezione di tipo 'SQLException'.

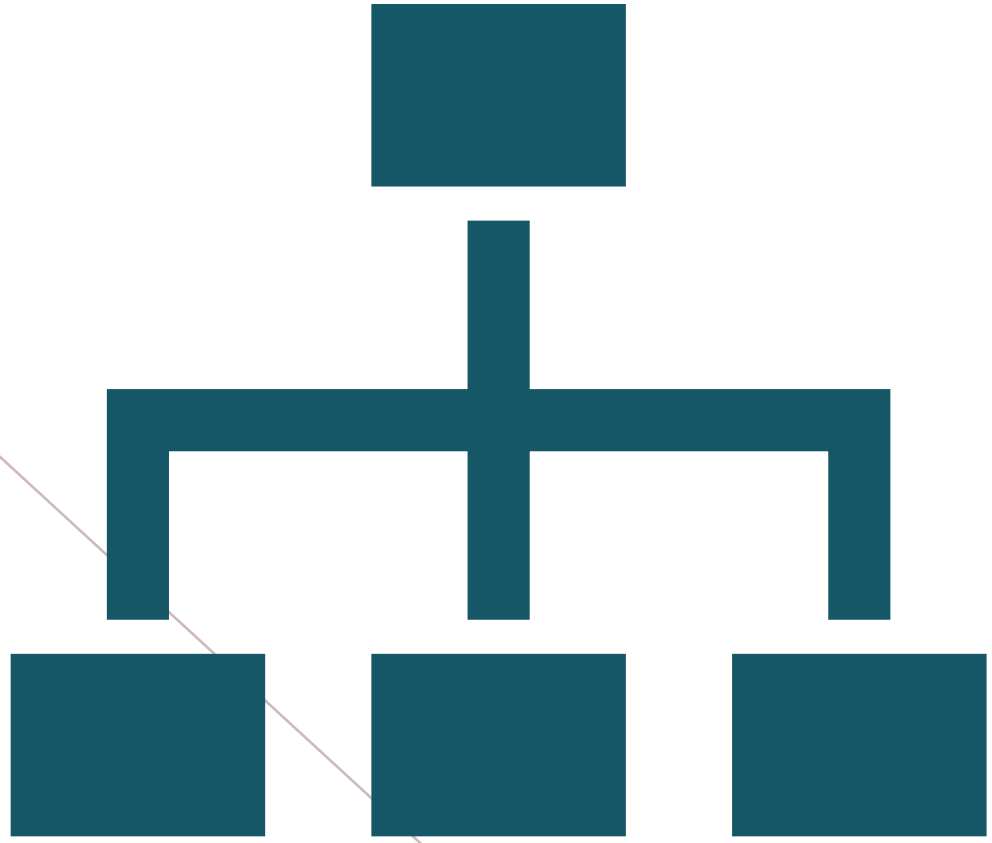
- **views**: questo file contiene le diverse viste (Frame e Dialog) della web app sviluppata utilizzando il framework Vaadin:
  - AggiuntaBraniView: questa classe rappresenta una **vista che consente agli utenti di cercare brani, selezionarli ed aggiungerli ad una playlist**. Gestisce l'interazione con gli elementi dell'interfaccia utente e comunica con il backend tramite l'oggetto **ClientES** per eseguire operazioni come la ricerca di brani o l'aggiunta di canzoni alla playlist. L'annotazione '**@PageTitle**' definisce il titolo della pagina come 'AggiuntaBrani', mentre l'annotazione '**@Route**' associa la vista all'URL 'aggiunta-brani'. La classe estende '**Dialog**', che è un componente di Vaadin utilizzato per creare una finestra pop-up all'interno dell'applicazione. Nel costruttore della classe vengono configurati i vari elementi dell'interfaccia utente. Questi includono campi di input, bottoni, una griglia ed i diversi layout per organizzare gli elementi. Ogni metodo '**configure...**' è responsabile della configurazione di un aspetto specifico dell'UI come il layout, la barra di ricerca, la griglia dei brani e i pulsanti di aggiunta e conferma. La funzione '**search**' gestisce la ricerca dei brani. Quando l'utente immette criteri di ricerca (titolo, autore, anno) e preme il pulsante '**Cerca**', viene chiamato il metodo '**searchSong**' della classe '**ClientES**' per ottenere una lista di brani corrispondenti, escludendo quelli già presenti nella playlist oppure eventualmente già selezionati. Una volta fatta la ricerca e prodotto il risultato (in una griglia), la lista degli anni viene filtrata mostrando solo quelli disponibile per le canzoni. In caso di errori o mancanza di risultati, vengono mostrate in output le notifiche appropriate. Il metodo '**aggiungiBrani**' consente invece di selezionare brani ed aggiungerli ad una lista ovvero '**braniSelezionati**'. Questi brani sono poi aggiunti alla playlist quando l'utente conferma l'operazione.
  - HistogramView: questa classe rappresenta un istogramma che raffigura **la media delle emozioni associate ad una specifica canzone**. All'interno del costruttore, viene inizializzato un oggetto '**DefaultCategoryDataset**', che è utilizzato per memorizzare i dati del grafico a barre. Successivamente, è chiamato il metodo '**fillDatasetWithAverageEmotions**' per riempire il dataset con le emozioni medie associate alla canzone specificata da 'idSong'. Se non ci sono votazioni disponibili per la canzone, il grafico non sarà visibile e viene sollevata l'eccezione 'NoVotazioni'. Altrimenti, viene creato un oggetto '**JFreeChart**' utilizzando il metodo '**createBarChart**' di '**ChartFactory**'. Questo metodo crea un grafico a barre con il titolo 'Distribuzione Emozioni', un'etichetta sull'asse X per le emozioni e sull'asse Y per la media. Il tutto utilizzando i dati del dataset precedentemente riempito. Il range dei valori dell'asse Y è impostato da 0 a 5.5 utilizzando il metodo '**setRange**', mentre il renderer (con larghezza massima delle barre settata a 0.05) è responsabile per la rappresentazione visuale delle colonne. Dopo aver apportato configurazioni alla formattazione del testo per la legenda ed il titolo, il grafico è convertito in un'immagine PNG utilizzando **la libreria JFreeChart** e memorizzato in un array di byte ('chartImageBytes'). Un componente Div ('**chartContainer**'), a cui sono impostate le dimensioni e lo sfondo, è utilizzato per ospitare l'immagine del grafico. Quest'ultima viene codificata in Base64 ed utilizzata come sfondo del componente, consentendo la visualizzazione dell'istogramma.

- InsEmozioniView: questa classe, che estende la classe Dialog di Vaadin, rappresenta **una vista per la valutazione delle emozioni associate ad una canzone**. Il costruttore accetta un parametro Canzone songSelected che rappresenta la canzone selezionata per la valutazione delle emozioni. Al suo interno, vengono configurate diverse componenti grafiche come titoli, pulsanti ed il layout. Il comportamento del pulsante di conferma (confirmButton) è definito da un Click Listener. Dopo aver raccolto dalla griglia i punteggi delle emozioni valutate, sono eseguite azioni specifiche in base alla presenza o meno della canzone selezionata nella playlist. Nel primo caso, invocando il metodo **insEmoBranoPlaylist** sul oggetto client vengono inseriti i dati di votazione e mandato in output il relativo messaggio di compimento dell'operazione, in caso contrario viene mostrata una notifica di errore. Se si verifica un'eccezione SQL durante l'inserimento dei dati, si utilizza il metodo **updateEmoBranoPlaylist** per eseguire un aggiornamento. Se anche questo approccio fallisce a sua volta, viene mostrata una notifica di errore generico. Il metodo **getGridVotazione()**, invece, crea una griglia (grid) che configura le colonne per la valutazione delle emozioni e l'aggiunta di commenti. Le emozioni visualizzate nella griglia sono ottenute tramite il metodo getEmotions().
- MainLayout: questa classe, che estende AppLayout di Vaadin, rappresenta la **vista principale dell'applicazione** e definisce la struttura dell'intero layout. In particolare, si occupa della configurazione del layout superiore (barra superiore) contenente elementi come il pulsante di login, l'avatar utente, il menu di navigazione, il messaggio di benvenuto e l'impostazione del form di login, con campi per l'username, la password, i pulsanti per l'accesso e la registrazione. Il costruttore, in base alla presenza di un utente loggato, nasconde o mostra i componenti pertinenti e stabilisce il comportamento del dialogo di login attraverso il metodo **login()**. Quest'ultimo richiede all'oggetto client di tipo **ClientES** di verificare le credenziali immesse dall'utente. Il metodo **logout()**, invece, invalida la sessione utente corrente effettuando appunto il logout. Successivamente il metodo **addDrawerContent()** è usato per stabilire il contenuto del menu laterale (drawer); mentre **createNavigation()** serve per creare la struttura di navigazione che aggiunge le voci 'Ricerca' e 'My Playlist', ognuna collegata ad una specifica vista. Infine **afterNavigation()** viene chiamato dopo che la navigazione tra le viste è stata completata. Questo metodo imposta, nella barra dell'intestazione, il titolo della vista corrente ottenuto utilizzando l'annotazione @PageTitle presente nella classe della vista corrente.

- MyPlaylistView: questa classe rappresenta **la vista per la gestione delle playlist dell'utente**. Le annotazioni '**@PageTitle**('Playlist")' e '**@Route(value = 'my-playlist', layout = MainLayout.class)**' indicano essa è mappata all'URL 'my-playlist' all'interno del layout principale 'MainLayout'. Il costruttore della classe contiene principalmente due sezioni, una per gli utenti non autenticati e una per quelli autenticati. Per i primi, viene mostrata una sezione con un pulsante 'Registrati' ed un messaggio che richiede all'utente di effettuare l'accesso per visualizzare la pagina. Questa sezione è contenuta in un layout '**VerticalLayout**' chiamato '**noLogged**'. Se invece l'utente è autenticato, viene mostrata la sezione per la gestione delle playlist personali. Vengono creati i vari componenti dell'UI come pulsanti, campi di testo, icone, layout, griglie e dialoghi per la creazione, visualizzazione e modifica delle playlist. In particolare, il pulsante '**Crea nuova playlist**' apre una finestra di dialogo. All'interno di essa, abbiamo un campo di testo (per l'assegnazione del nome) ed un altro pulsante ovvero '**Crea Playlist**' il cui click listerner richiama il metodo **addPlaylist()**. Quest'ultimo verifica che il nome inserito non sia vuoto, oppure già assegnato, e dopo aver eseguito tali controlli procede con la creazione effettiva della playlist aggiornando il database (tramite il metodo **addPlaylist()** dell'oggetto **client di tipo ClientES**). Avviene poi la configurazione del layout della pagina, tra cui il titolo 'Le tue Playlist', l'icona della nota musicale e l'impostazione di una griglia ('Grid') per mostrare le playlist dell'utente autenticato. Appaiono inoltre le colonne 'Visualizza/Modifica' ed 'Elimina' con gli appositi pulsanti dedicati a tali operazioni. Per visualizzare i dettagli della playlist selezionata si apre un nuovo dialogo che permette di vedere le canzoni nella playlist ed eseguire altre azioni. Si può procedere con il refactoring di una playlist esistente, tramite l'apertura di un altro dialogo che analogamente al caso precedente esegue i diversi controlli prima di rinominare la playlist grazie al metodo **renamePlaylist()** invocato sull'oggetto client. Sono poi implementate funzioni per l'aggiunta di brani musicali, la visualizzazione (**client.showCanzoniPlaylist()**) e la rimozione (**client.removePlaylistSong()**) di questi ultimi dalla playlist selezionata e la votazione delle emozioni associate ad una determinata canzone (tramite il metodo **open()** invocato sull'oggetto **insEmoDialog** di tipo **InsEmozioniView** che permette di accedere alla relativa vista).
- ProfileView: questa classe rappresenta **la vista relativa al profilo dell'utente**. Se l'utente è autenticato può visualizzare i propri dati. È possibile modificare alcune informazioni personali come l'indirizzo di residenza, l'e-mail e la password attraverso il metodo del client **modificaDati()**. In particolare il campo della mail viene validato solo se rispetta il formato definito grazie all'uso di **regex**, mentre la password ha il vincolo di avere necessariamente un numero di caratteri non inferiore ad 8. Le modifiche richiedono la conferma tramite un dialogo. È possibile anche procedere con l'eliminazione del proprio account che, ancora una volta, apre un nuovo dialogo di conferma dell'operazione e viene eseguito mediante l'invocazione del metodo **eliminaAccount()** sull'oggetto client di tipo '**ClientES**'. Quest'ultimo è utilizzato per comunicare con il database per recuperare/aggiornare i dati dell'utente e per eliminare il proprio account. Per quanto riguarda la disposizione dei componenti, viene usato il '**VerticalLayout**' per organizzare i componenti in una colonna verticale dove i dati personali, l'indirizzo di residenza e le informazioni di accesso sono suddivisi in sezioni distinte.

- RegistrazioneView: questa classe rappresenta la **vista per la registrazione all'app di un nuovo utente**. Il costruttore inizializza i vari componenti dell'interfaccia, tra cui il pulsante 'Registrati' e 'Calcola CF'. Il metodo '**configureLayoutTitolo()**' è responsabile dell'impostazione del layout per il titolo della pagina, mentre '**configureLayoutRegistrazione()**' crea un layout di form che contiene campi per l'inserimento dei dati utente; '**configureButton()**' configura il pulsante di registrazione ed infine **'configurePageLayout()'** definisce il layout complessivo della pagina, includendo il form di registrazione ed il pulsante 'Registrati'. Il metodo **'calcolaCodFiscale()'** sfrutta la **libreria it.kamaladafrica.codicefiscale** che consente di calcolare in modo accurato il codice fiscale italiano basato sui dati dell'utente. Il metodo **'registration()'** gestisce il processo di registrazione, verifica la correttezza dei dati inseriti dall'utente, inclusi i criteri di validità per username, password e codice fiscale. Se l'username è univoco (rispetto ai dati presenti nel db), l'utente viene registrato utilizzando il metodo 'registrazione' dell'oggetto di tipo **'ClientES'**. Viene mostrata una notifica di registrazione avvenuta e l'utente viene reindirizzato alla vista di ricerca. In caso contrario, viene mostrata una notifica che segnala la presenza di un'username già utilizzato.
- RicercaView: questa classe rappresenta la vista per la ricerca di canzoni nell'applicazione ed è basata su un **'VerticalLayout'** che contiene i vari elementi. Il costruttore inizializza i componenti dell'interfaccia, tra cui i campi di input per la ricerca, i pulsanti **'Cerca'** e **'Visualizza Emozioni'**, nonché una griglia per mostrare i risultati della ricerca. Viene chiamato il metodo **'configureLayout()'** per definire il titolo della pagina, **'configureSearchBar()'** per impostare il layout della barra di ricerca, **'configureGrid()'** per definire la griglia dei risultati e **'configureEmotions()'** per configurare il layout delle emozioni. Il metodo **'search()'** è responsabile dell'esecuzione della ricerca. Utilizzando la classe **'ClientES'**, vengono eseguite le query di ricerca basate sui campi inseriti dall'utente (titolo, autore e anno). I risultati vengono visualizzati nella griglia. Il metodo **'visualizzaEmo()'** mostra le emozioni relative ad una canzone selezionata. Viene aperta una finestra di dialogo contenente le informazioni sulla canzone ed un grafico inerente alle medie delle emozioni associate ad essa. Il metodo **'openDetailsDialog()'** configura il layout della finestra di dialogo con le informazioni sulla canzone ed il grafico relativo. Inoltre, il Frame offre la possibilità di visualizzare i commenti associati alla canzone nel caso siano presenti (altrimenti viene mandato in output il relativo messaggio di errore). Infine **'cleanComments()'** rimuove i commenti non validi (nulli) dalla lista di emozioni associate alla canzone in questione, per una visualizzazione migliore. Vengono gestite le eccezioni nel caso in cui non siano presenti emozioni o commenti per il brano selezionato.

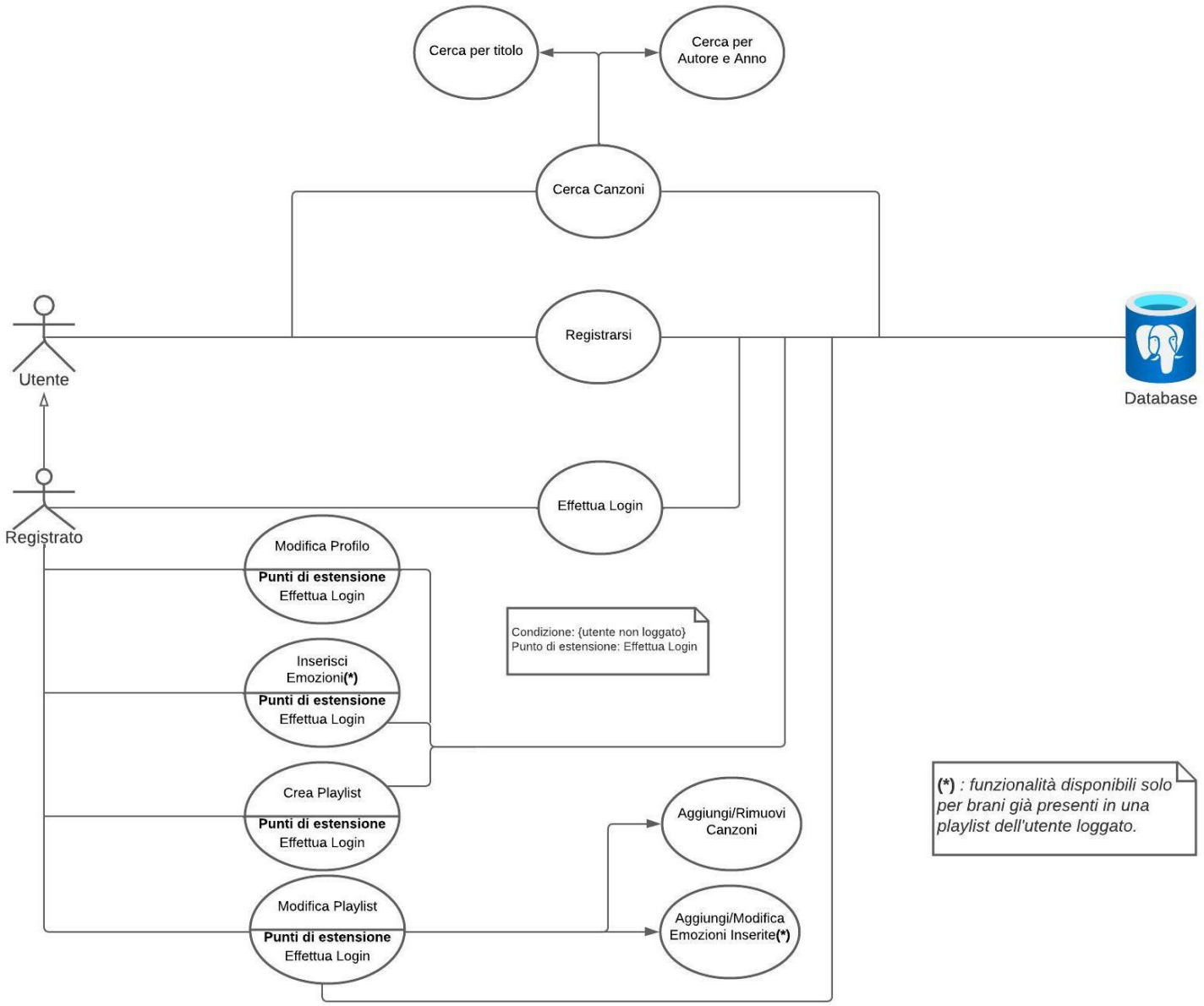
SCELTE PROGETTUALI: DIAGRAMMI UML



DIAGRAMMI UML

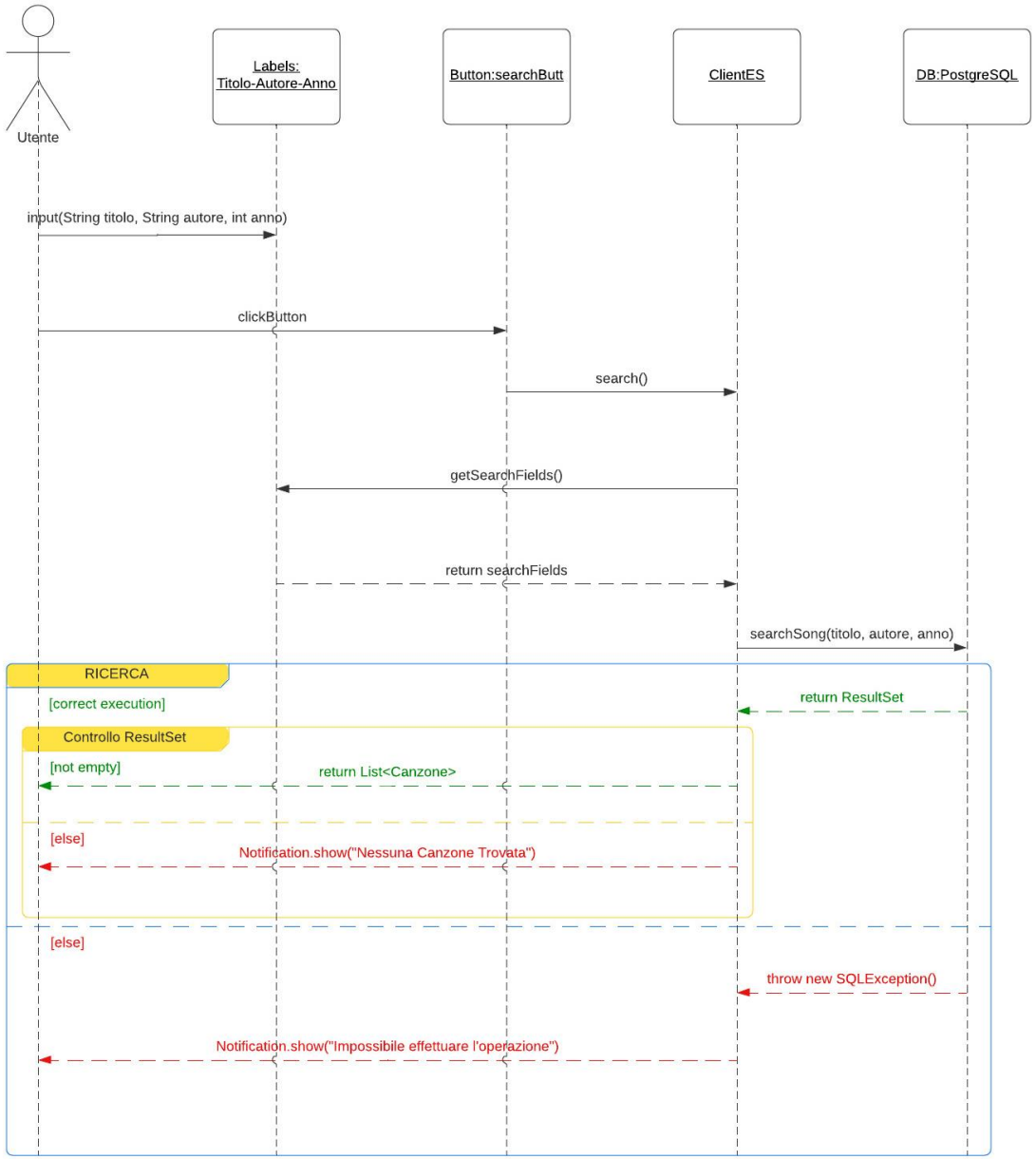
In questa sezione riportiamo i vari diagrammi UML che svolgono un ruolo chiave per quanto riguarda la progettazione dell'applicazione. Nello specifico sono stati realizzati:

- USE CASE DIAGRAM: usato per rappresentare in modo informale le diverse interazioni possibili tra utente ed applicazione in base ai requisiti forniti dai committenti del progetto.
- SEQUENCE DIAGRAM: usato per fornire la visione comportamentale del sistema in relazione alla funzionalità di ricerca. Il diagramma è da considerarsi esemplificativo per quanto riguarda tutte le altre operazioni.
- CLASS DIAGRAM: usato per mostrare la struttura statica dell'applicazione, evidenziando le dipendenze tra classi e lo scheletro di queste ultime.

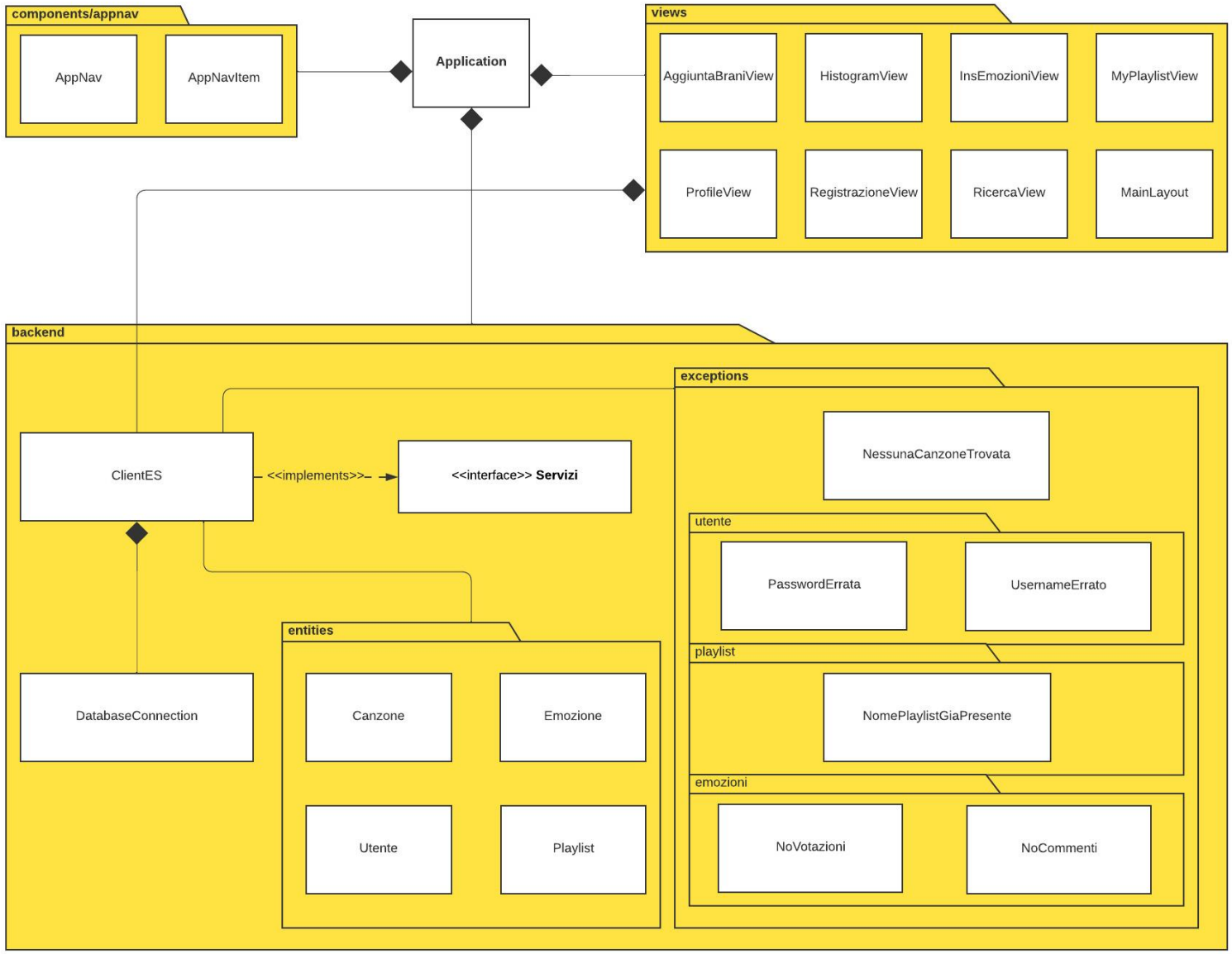

Cerca per titolo
Cerca per Autore e Anno
Cerca Canzoni
Utente
Registrarsi
Database
Registrato
Effettua Login
Modifica Profilo
Punti di estensione
Effettua Login
Condizione: {utente non loggato}
Punto di estensione: Effettua Login
Inserisci Emozioni(*)
Punti di estensione
Effettua Login
Crea Playlist
Punti di estensione
Effettua Login
Aggiungi/Rimuovi Canzoni
Modifica Playlist
Punti di estensione
Effettua Login
Aggiungi/Modifica Emozioni Inserite(*)
(*) : funzionalità disponibili solo per brani già presenti in una playlist dell'utente loggato.

USE CASE

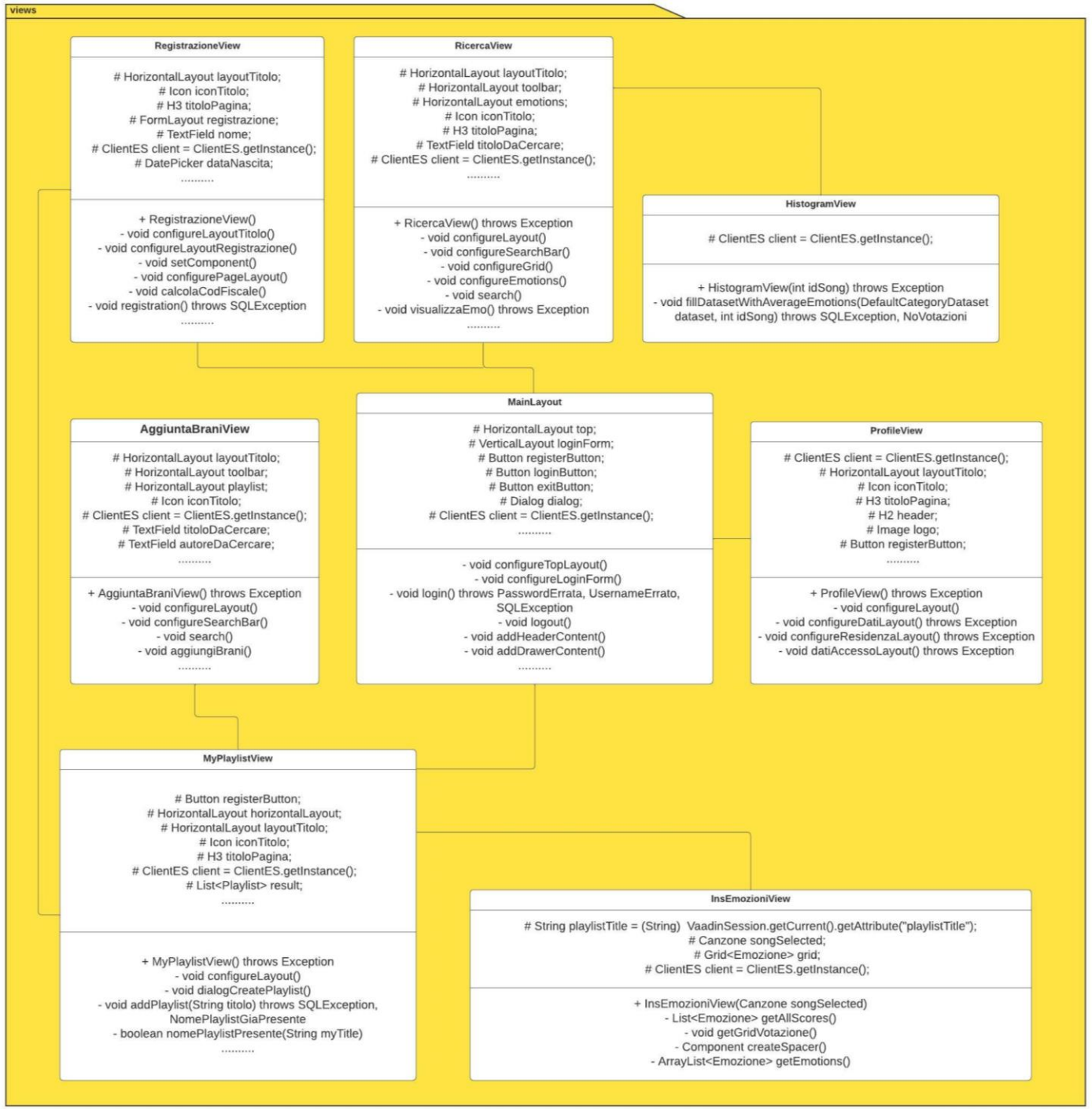
Al fine di preservare la chiarezza e la comprensibilità del diagramma, si è scelto di adottare una rappresentazione semplificata del metodo "searchSong()", evitando così di includere dettagli implementativi interni, secondo il principio di elisione consueto nei diagrammi UML. Inoltre, per le ragioni precedentemente menzionate, non sono state incluse nel diagramma alcune funzionalità aggiuntive. Queste comprendono il recupero degli anni in base ai parametri di ricerca e l'aggiornamento/visibilità della griglia preposta a contenere i brani musicali.



SEQUENCE DIAGRAM

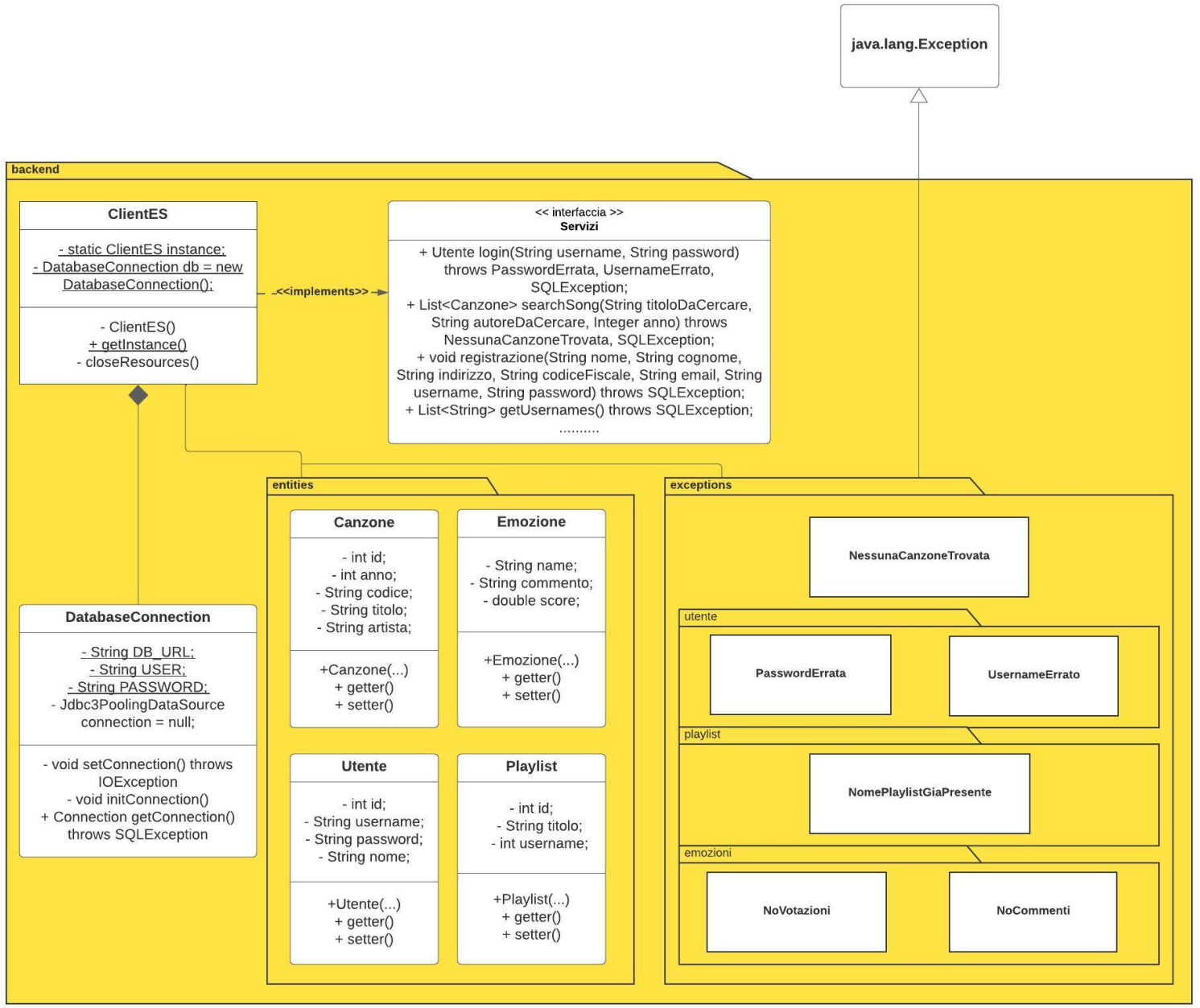


CLASS DIAGRAM



DI SEGUITO È VISIBILE UNA SPECIALIZZAZIONE DEL CLASS DIAGRAM RELATIVA AL PACKAGE 'VIEWS'

VIEWS



DI SEGUITO È VISIBILE UNA SPECIALIZZAZIONE DEL CLASS DIAGRAM RELATIVA AL PACKAGE 'BACKEND'

BACKEND

SCELTE PROGETTUALI: DATABASE

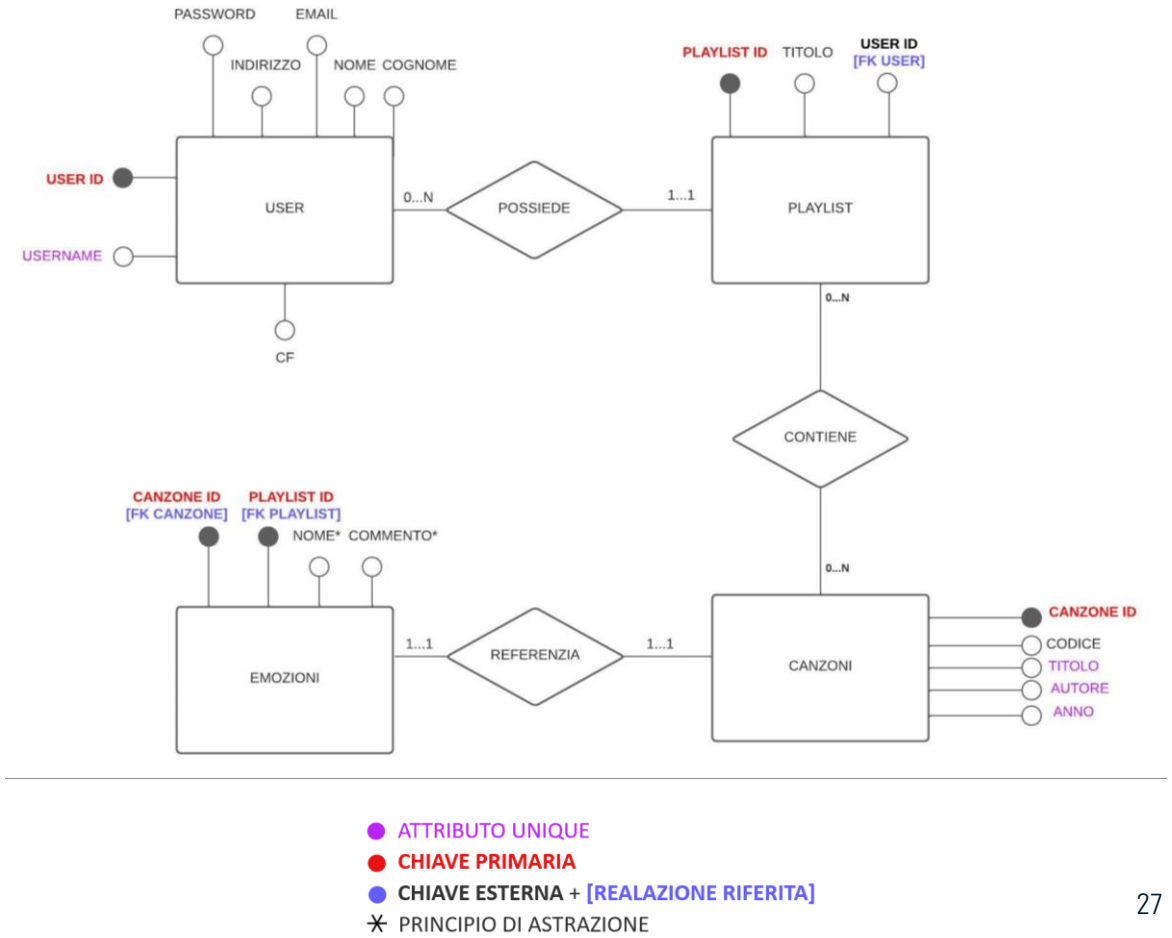
SCHEMA CONCETTUALE DB

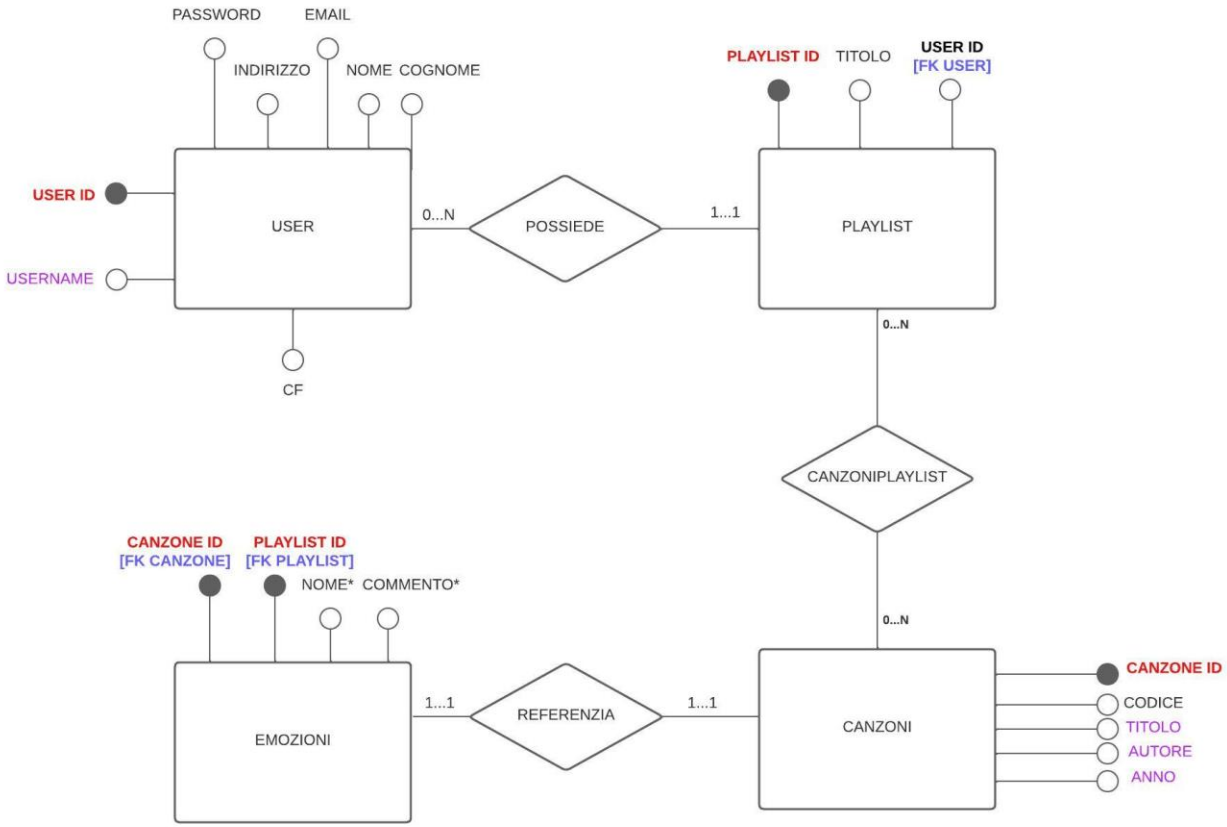
Entità

- **User**: Utente registrato che utilizza l'applicazione, con possibilità di login e gestione playlist.
- **Canzoni**: Brani musicali con codice univoco salvati nell'applicazione.
- **Playlist**: Collezioni di Canzoni appartenenti al singolo utente, a cui è possibile associare un rating diverso per ogni canzone.
- **Emozioni**: Entità debole che rappresenta tutte le 9 emozioni ed i relativi commenti, utilizzata per il rating di una canzone all'interno di una playlist.
- * : per il principio di astrazione, gli attributi 'nome' e 'commento' dell'entità 'Emozioni' rappresentano rispettivamente il valore numerico e l'eventuale commento associati ad ognuna delle 9 emozioni.

Attributi

- User : user_id, username, hashed_password, indirizzo, cf, email.
- Canzoni : canzoni_id, codice, autore, titolo, nome, cognome.
- Playlist : playlist_id, titolo.
- Emozioni : amazement, solemnity, tenderness, nostalgia, calmness, power, joy, tension, sadness (rating numerico e commento testuale per ognuna delle emozioni citate).





TRADUZIONE:

- USER(**USER ID**, NOME, COGNOME, USERNAME, PASSWORD, INDIRIZZO, CF, EMAIL)
- PLAYLIST (**PLAYLIST ID**, TITOLO, **USER ID [USER]**)
- CANZONI (**ID CANZONE**, ANNO, AUTORE, TITOLO)
- EMOZIONI (**PLAYLIST ID [PLAYLIST]**,**CANZONE ID [CANZONI]**, NOME*, COMMENTO*)
- CANZONI_PLAYLIST (**PLAYLIST ID [PLAYLIST]**, **CANZONE ID [CANZONI]**)

DIAGRAMMA ER

Relazioni

- Un utente può avere più playlists.
- Una playlist può contenere più canzoni.
- Una canzone, all'interno di una playlist può avere un solo rating.

L'entità **CanzoniPlaylist** (che sostituisce la vecchia relazione 'Contiene') è una tabella ponte tra le entità Playlist e Canzoni. Infatti, una Playlist può essere associata ad un Utente e più Canzoni, e una Canzone può essere associata a più Playlist.

Le entità *Emozioni* e *CanzoniPlaylist* sono entità deboli. Ciò significa che non hanno una chiave primaria propria e la loro chiave primaria è la combinazione delle chiavi esterne Playlist_id e Canzoni_id.

```sql
CREATE TABLE public."User"
(
    User_id          serial  NOT NULL,
    nome             varchar NULL,
    cognome          varchar NULL,
    username         varchar NOT NULL,
    hashed_password varchar NOT NULL,
    indirizzo        varchar NULL,
    cf               varchar NULL,
    email            varchar NOT NULL,
    CONSTRAINT user_pk PRIMARY KEY (User_id),
    CONSTRAINT user_un UNIQUE (username)
);
```

```sql
CREATE TABLE public."Canzoni"
(
    anno       numeric(4) NULL,
    codice     varchar(18) NOT NULL,
    autore     varchar,
    titolo     varchar,
    Canzoni_id serial      NOT NULL,
    CONSTRAINT canzoni_primary_key PRIMARY KEY (Canzoni_id),
    CONSTRAINT canzoni_unique_key UNIQUE (anno, autore, titolo)
);
```

VINCOLI D'INTEGRITÀ

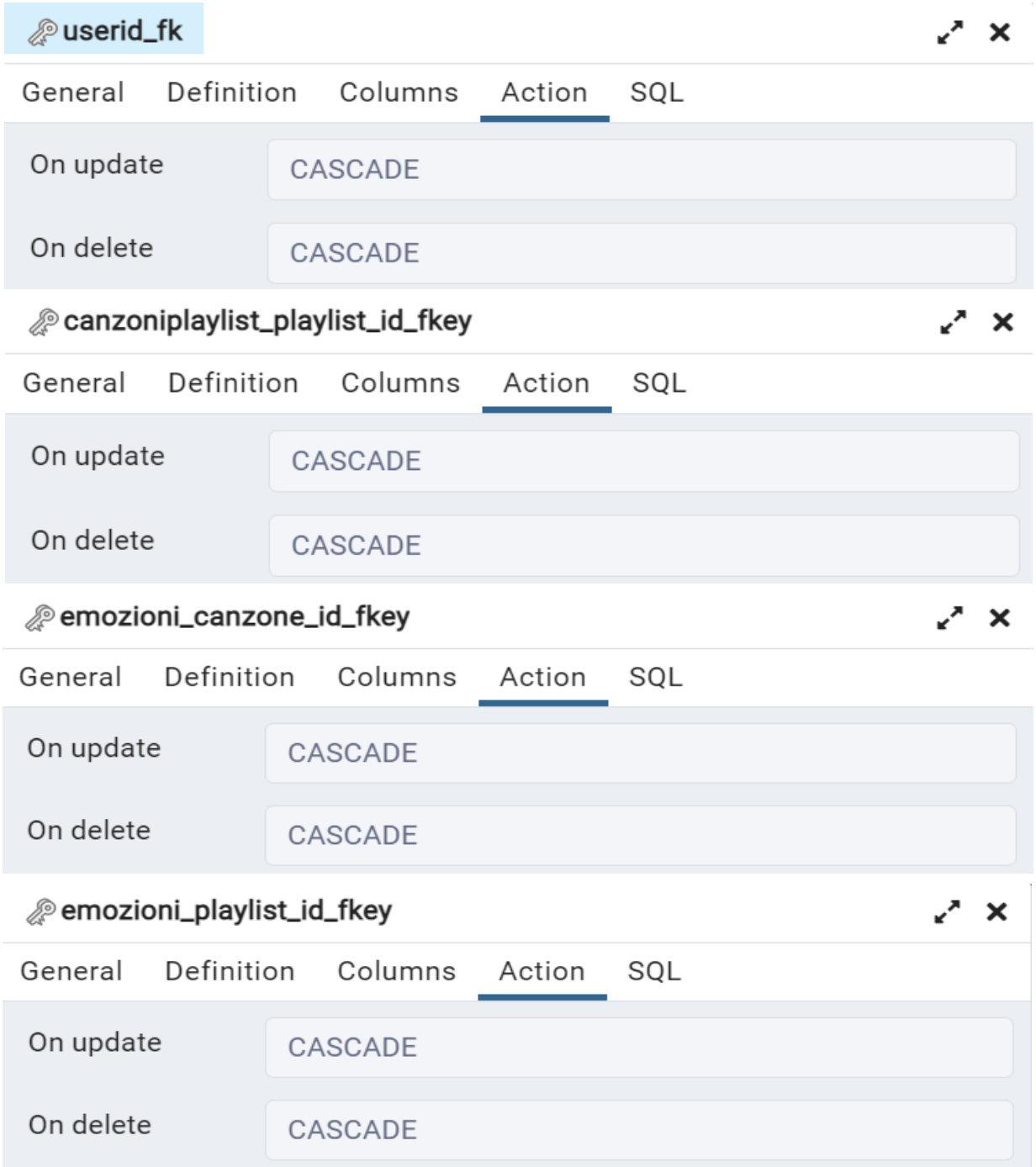
Nel database, sono definiti i seguenti vincoli d'integrità:

User

- **Username** unique per evitare due utenti con lo stesso username, il che renderebbe inconsistente la procedura di login.

Canzoni

- **Anno, Autore, Titolo** unique per evitare di salvare canzoni duplicate.



VINCOLI D'INTEGRITÀ

Playlist

- Delete e update **cascade** su entità User (PK).

CanzoniPlaylist

- Delete e update **cascade** su entità Playlist (PK).

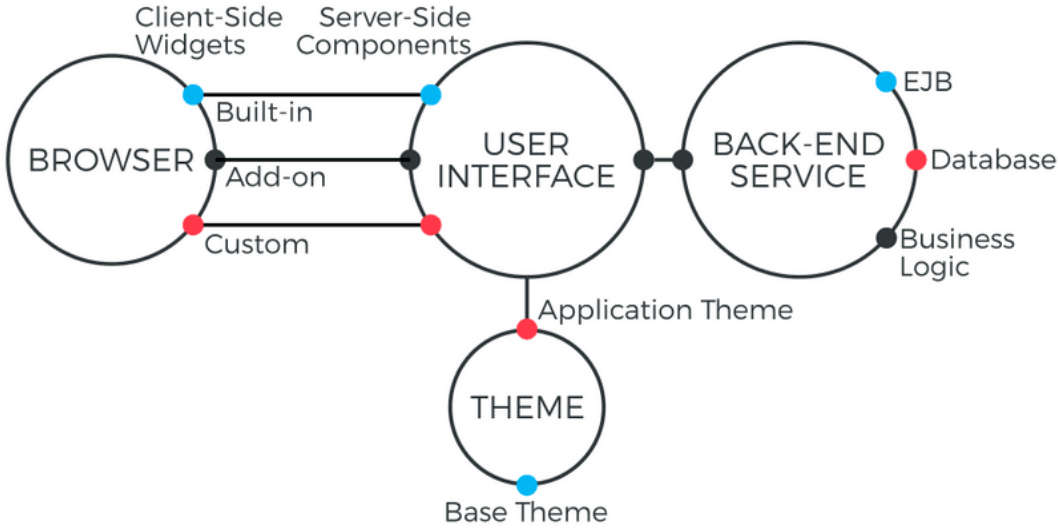
Emozioni

- Delete e update **cascade** su entità Playlist (PK).
- Delete e update **cascade** su entità Canzone (PK).

SCELTE ARCHITETTURALI

SCELTE ARCHITETTURALI

L'architettura della web app utilizza il **modello di programmazione server-side** fornito dal framework Vaadin che è provvisto di un server proprio (**Tomcat**). Questo approccio semplifica lo sviluppo di GUI. Vi sono due componenti principali: applicazione lato server ed il motore lato client. La prima viene eseguita come **servlet** (che gestisce le richieste e le risposte http) che riceve le richieste dal client e le interpreta come eventi per una particolare sessione utente. Ogni componente lato server ha un corrispondente lato client, un widget, con il quale l'utente interagisce. I componenti lato server inoltrano questi eventi alla logica dell'applicazione. Il motore lato client, eseguito nel browser, gestisce l'interazione dell'utente con il server e quindi renderizza le modifiche nell'interfaccia utente apportando modifiche alla pagina. Le comunicazioni client-server e le interazioni concorrenti tra gli utenti sono gestite automaticamente dal framework ed avvengono tramite **WebSocket**, consentendo agli aggiornamenti dell'interfaccia utente di essere trasmessi in tempo reale al client. Grazie alla struttura modulare ed all'utilizzo dell'interfaccia 'Servizi', il progetto è in grado di crescere in modo flessibile e manutenibile. Questa tipologia di architettura separa le diverse parti dell'applicazione e garitisce ortogonolità, inoltre consente l'aggiunta di funzionalità avanzate in modo ordinato, senza compromettere la stabilità dell'applicazione esistente. Ciò garantisce una possibile evoluzione nel tempo, mantenendo un ottimo livello di scalabilità e chiarezza strutturale.



Link di approfondimento:

- https://vaadin.com/docs/v8/framework/introduction/intro-overview#:~:text=Vaadin%20Framework%20is%20a%20Java,is%20the%20more%20powerful%20one.
- https://vaadin.com/docs/v8/framework/architecture/architecture-overview

STRUTTURE DATI UTILIZZATE

STRUTTURE DATI UTILIZZATE

Le strutture dati principali usate sono le **liste** (principalmente **Arraylist**), fondamentali per organizzare e gestire le informazioni in modo efficiente nel codice. Essendo **strutture dati di tipo dinamico**, consentono di memorizzare collezioni di elementi senza necessità di allocare uno spazio predeterminato ed immutabile. Sono state usate in diversi contesti per esempio: memorizzare il result set delle canzoni restituite dal database in base ai parametri inseriti dall'utente nel metodo **searchSong()**; tracciare i nomi degli utenti registrati nel metodo **getUsernames()**; memorizzare il result set delle playlist associate ad un determinato utente, ecc.

Altri vantaggi del loro utilizzo sono l'accesso facilitato ai loro elementi, grazie all'uso degli indici, e l'esecuzione efficiente delle operazioni di aggiunta.

SCELTE ALGORITMICHE

SCELTE ALGORITMICHE

Le strutture dati, come le liste, utilizzate per organizzare ed elaborare le informazioni, influenzano la complessità temporale dell'applicazione. L'inserimento, la ricerca, l'aggiornamento e la cancellazione degli elementi all'interno di queste strutture hanno un impatto sulla velocità delle operazioni. La scelta di algoritmi efficienti per gestire queste operazioni determina quanto sia scalabile l'applicazione quando il volume di dati aumenta. Nel nostro caso abbiamo scelto di utilizzare gli ArrayList. Si tratta di un'implementazione di List realizzata internamente con un array di dimensione dinamica ovvero, quando l'array sottostante è pieno, esso viene riallocato con una dimensione maggiore ed i vecchi dati vengono copiati nel nuovo array. Il ridimensionamento avviene in modo che l'operazione di inserimento (add) abbia complessità ammortizzata costante. Anche ogni operazione di accesso posizionale richiede tempo costante. La seguente tabella riassume la complessità computazionale dei principali metodi degli ArrayList.

Inoltre, l'interazione con un DBMS come PostgreSQL introduce un altro livello di complessità. Le query di ricerca, cancellazione, aggiornamento e inserimento nel database possono avere complessità variabili a seconda della struttura delle tabelle, degli indici utilizzati e della complessità delle query stesse. La progettazione adeguata e l'ottimizzazione delle query tramite l'uso di indici o altre tecniche, influenzano direttamente le prestazioni dell'applicazione.

Per esaminare le complessità algoritmiche in PostgreSQL, si può fare riferimento alla documentazione ufficiale e ai materiali di apprendimento disponibili. La documentazione ufficiale di PostgreSQL spiega come funzionano i diversi aspetti del motore del database, inclusi gli algoritmi utilizzati per le query e le loro ottimizzazioni o indicizzazioni.

Metodo	Complessità in AL
add	O(1)*
remove	O(n)
contains	O(n)
get, set	**O(1)**
addFirst, addLast, removeFirst, removeLast	-

Note:

- (*) complessità ammortizzata
- add aggiunge in coda
- remove deve trovare l'elemento prima di rimuoverlo

In particolare, si potrebbero voler esaminare i seguenti argomenti nella documentazione di PostgreSQL:

- **Piani di Esecuzione delle Query:** questa sezione spiega come PostgreSQL analizza ed ottimizza le query SQL, scegliendo il piano di esecuzione ottimale basato su algoritmi come l'Algoritmo di Selezione di Join, il Pushdown delle Operazioni, ecc.
- **Strategie di Indicizzazione:** PostgreSQL offre diverse strategie di indicizzazione (B-tree etc.) e spiega come ciascuna strategia viene utilizzata per ottimizzare le query.
- **Sistema di Pianificazione delle Query:** esamina come il pianificatore di query di PostgreSQL prende decisioni sull'ordine di esecuzione delle operazioni di query, cercando di minimizzare il tempo di esecuzione complessivo.
- **Ottimizzazioni per Query Complesse:** esplora come PostgreSQL gestisce query complesse e introduce tecniche come la fusione di subquery, la traduzione di predicati, ecc.
- **Architettura Interna:** scopri l'architettura interna di PostgreSQL per avere una visione più approfondita su come vengono gestite le transazioni, l'archiviazione dei dati e altro ancora.

La documentazione di PostgreSQL è consultabile all'indirizzo: https://www.postgresql.org/docs/

In sintesi, la complessità algoritmica dell'applicazione EmotionalSong è una combinazione dell'efficienza delle strutture dati utilizzate internamente, delle operazioni svolte su di esse e dell'efficienza delle operazioni di interrogazione e gestione dati del DBMS PostgreSQL. Una buona progettazione delle strutture dati, l'implementazione di algoritmi ottimizzati e l'utilizzo efficace delle funzionalità del DBMS sono determinanti per garantire che l'applicazione possa gestire in modo efficiente carichi di lavoro crescenti e fornire un'esperienza utente fluida.

FORMATO DEI FILE E LA LORO GESTIONE

CARTELLA RESOURCES

In un progetto Maven, la gestione dei file all'interno della directory "**resources**" è una parte essenziale del processo di sviluppo.

La directory "resources" contiene risorse non codice sorgente, come file di configurazione, file di proprietà, immagini e altro ancora, che sono utilizzati dal progetto durante l'esecuzione.

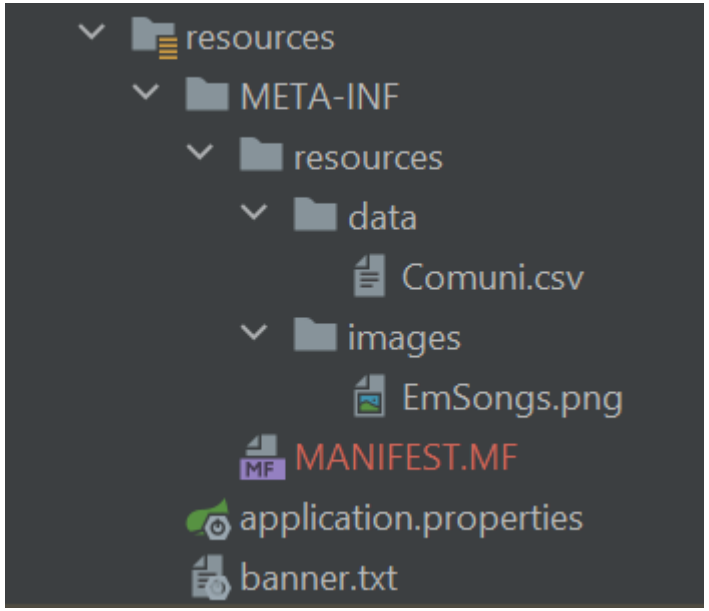
La directory "resources" si trova di solito nella radice del tuo progetto Maven, accanto alla directory "**src**". Maven gestisce automaticamente i file all'interno della directory "resources" e li include nel tuo progetto compilato. Quando compili il tuo progetto, Maven copia il contenuto della directory "resources" nel percorso appropriato dell'artefatto compilato (ad esempio, nel file JAR risultante).

Il plugin "**maven-resources-plugin**" di Maven è responsabile della gestione delle risorse nel progetto. Viene automaticamente attivato durante la compilazione e può essere configurato per eseguire operazioni come il filtraggio, la copia e la definizione di percorsi di output personalizzati.

In generale, la gestione delle risorse in un progetto Maven è progettata per semplificare l'inclusione e l'uso di risorse non codice sorgente nel tuo progetto. Maven automatizza gran parte di questo processo, riducendo la complessità delle operazioni di gestione dei file di risorse.

Nell'applicazione Emotional Songs la cartella "resources" è impiegata per la gestione:

- Del file '**Comuni.csv**' che è necessario durante la registrazione per selezionare la propria residenza.
- Dell'immagine '**EmSongs.png**' che costituisce l'icona della web app.
- Del file '**application.properties**' che contiene le informazioni di connessione al database.
- Del file '**banner.txt**' che contiene il banner che viene visualizzato durante l'avvio dell'applicazione da IDE.



PATTERN

PATTERN UTILIZZATI

```
private static ClientES instance;

/**
 * Costruttore privato per garantire il pattern Singleton.
 */
1 usage  loren
private ClientES() {}

/**
 * Ottiene un'istanza unica del client per l'accesso ai servizi.
 * @return Un'istanza di ClientES.
 */
loren +1
public static synchronized ClientES getInstance() {
    if (instance == null) {
        instance = new ClientES();
    }
    return instance;
}
```

I design pattern sono "schemi di soluzioni" riutilizzabili, creati per risolvere problemi ricorrenti e prevedibili. I vantaggi principali del loro utilizzo sono: avere a disposizione una soluzione consolidata con un alto livello di astrazione per affrontare un problema che si ripresenta più volte, progettare il software con caratteristiche predefinite ed avere un supporto alla realizzazione di sistemi complessi. Il pattern che è stato adottato, è riportato di seguito.

Singleton: si tratta di un **DP creazionale che fa in modo che una classe abbia soltanto un'istanza**.
Nell'applicazione, è stato utilizzato per garantire l'univocità dell'istanza della classe 'ClientES', accessibile dalle molteplici viste del sistema. Il costruttore di 'ClientES' è reso privato in modo che non sia possibile crearne istanze direttamente all'esterno della classe. È presente un campo 'private static ClientES instance' che contiene l'istanza unica di 'ClientES'. Il metodo 'getInstance()' è dichiarato come 'static synchronized'. Questo significa che può essere invocato tramite la classe senza dover istanziare un oggetto, il metodo restituisce l'istanza esistente solo se è già stata creata, altrimenti ne crea una nuova. La keyword 'synchronized' viene utilizzata per assicurare che l'accesso concorrente a questo metodo sia sincronizzato e thread-safe. Questo evita che più thread creino accidentalmente più istanze simultaneamente. L'uso di questo pattern è giustificato per diversi motivi:

- La classe 'ClientES' è dotata di una connessione al database (gestita da 'DatabaseConnection'). È conveniente quindi che ci sia una sola istanza di connessione attiva in modo da ottimizzare l'uso delle risorse del database e non avere inconsistenze.
- Utilizzando una singola istanza, è possibile riutilizzare la connessione al database senza doverne creare una nuova ogni volta che è necessario interagire con il database.
- Il metodo 'getInstance()' fornisce un punto unico e globale per accedere all'istanza di 'ClientES' da qualsiasi parte dell'applicazione, semplificando la gestione della connessione al database.

PATTERN DI VAADIN

- MVP (Model View Presenter): è una strategia di progettazione ampiamente utilizzata nell'ambito di sviluppo di applicazioni complesse. Esso è simile al tradizionale pattern **Model-View-Controller (MVC)**, ma con alcune differenze significative. Nel pattern MVP, il Model rappresenta i dati e la logica dell'applicazione, mentre la View è responsabile della presentazione delle informazioni all'utente e della ricezione degli input. Il Presenter, invece, funge da intermediario tra il Model e la View ed a differenza del Controller di MVC, è progettato per essere completamente indipendente dalla tecnologia di interfaccia utente. Questo rende il Presenter molto più testabile, mantenibile e flessibile nel tempo.
- Observer: questo pattern comportamentale viene usato per gestire gli eventi e le interazioni dell'utente. Gli oggetti 'observer' (listener) sono registrati per ricevere notifiche quando gli eventi si verificano sugli oggetti 'osservati' (componenti dell'interfaccia utente).
- Composite: questo pattern strutturale viene usato per creare gerarchie di componenti. Ad esempio, un layout può contenere componenti annidati, creando così strutture complesse.
- Decorator: questo pattern strutturale viene usato per estendere le funzionalità dei componenti senza modificarne il loro codice principale.
- Facade: questo pattern strutturale viene usato per semplificare l'accesso a sottosistemi complessi, offrendo un'interfaccia semplificata e unificata al client.

LIBRERIE ESTERNE

LIBRERIE ESTERNE

Il file **pom.xml** è uno strumento di gestione delle dipendenze e di build per progetti Java. Il termine 'POM' sta per 'Project Object Model' ed è un file XML che definisce la struttura, le dipendenze, le configurazioni ed altre informazioni del progetto tra cui anche le librerie. Queste ultime in particolare sono specificate nella sezione <dependencies>. Le principali sono:

- vaadin e vaadin-spring-boot-starter: la prima contiene tutte le classi e le risorse necessarie per creare l'interfaccia web utente in modo semplice, fornendo componenti per la gestione degli eventi e delle sessioni. La seconda invece è responsabile delle configurazioni per avviare l'applicazione Vaadin all'interno di Spring Boot.
- org.parttio:line-awesome: libreria che fornisce icone (font-awesome) che possono essere utilizzate nell'UI.
- org.springframework.boot: libreria che semplifica lo sviluppo, la configurazione e la distribuzione di applicazioni Java Spring.
- javax.persistence: libreria che fa parte delle API di Java Persistence, uno standard usato per la gestione di oggetti persistenti in un ambiente di database relazionale. Queste API forniscono un modo standardizzato per mappare oggetti Java a tabelle di database e per eseguire operazioni di recupero ed interrogazione sui dati.
- org.springframework.data: include la libreria Spring Data JPA, che agevola l'accesso ai dati nel database utilizzando JPA.
- org.postgresql:postgresql: libreria che è utilizzata per connettersi al database PostgreSQL.
- de.svenkubiak:jBCrypt: libreria che fornisce funzioni per la crittografia delle password.
- org.jfree:JFreeChart: libreria utile alla creazione di grafici e visualizzazioni in Java all'interno del progetto.
- it.kamaladafrica:codice-fiscale: libreria che fornisce la funzione per il calcolo automatico del codice fiscale italiano.

CLOUD E
DEPLOY

AMAZON WEB SERVICES

Amazon Web Services, Inc. (nota con la sigla AWS) è un'azienda statunitense di proprietà del gruppo Amazon, che fornisce servizi di cloud computing su un'omonima piattaforma on demand.

Questi servizi sono operativi in 26 regioni geografiche in cui Amazon stessa ha suddiviso il globo, più altre 8 regioni disponibili prossimamente.

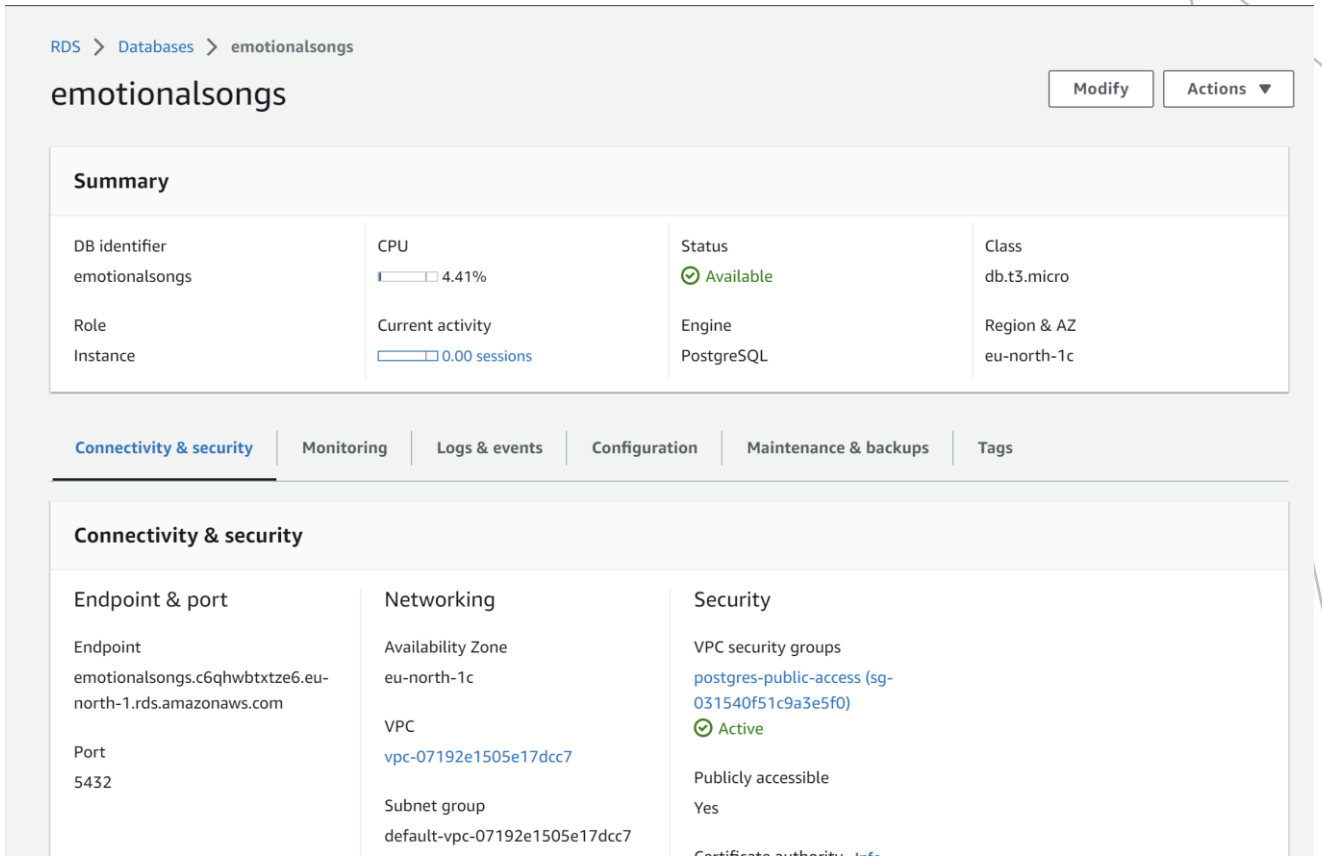
AWS offre oltre 200 prodotti, tra i quali Amazon Elastic Compute Cloud (EC2) e Amazon Simple Storage Service (S3), fornendo soluzioni on-demand con caratteristiche di high availability, ridondanza e sicurezza, in cui il costo finale deriva dalla combinazione di tipo e quantità di risorse utilizzate, caratteristiche scelte dall'utente, tempo di utilizzo e performance desiderate. A partire dai dati pubblicati nel quarto trimestre del 2018, AWS rappresenta per Amazon il 58% dei guadagni totali, rendendola la sua più grande fonte di incassi.

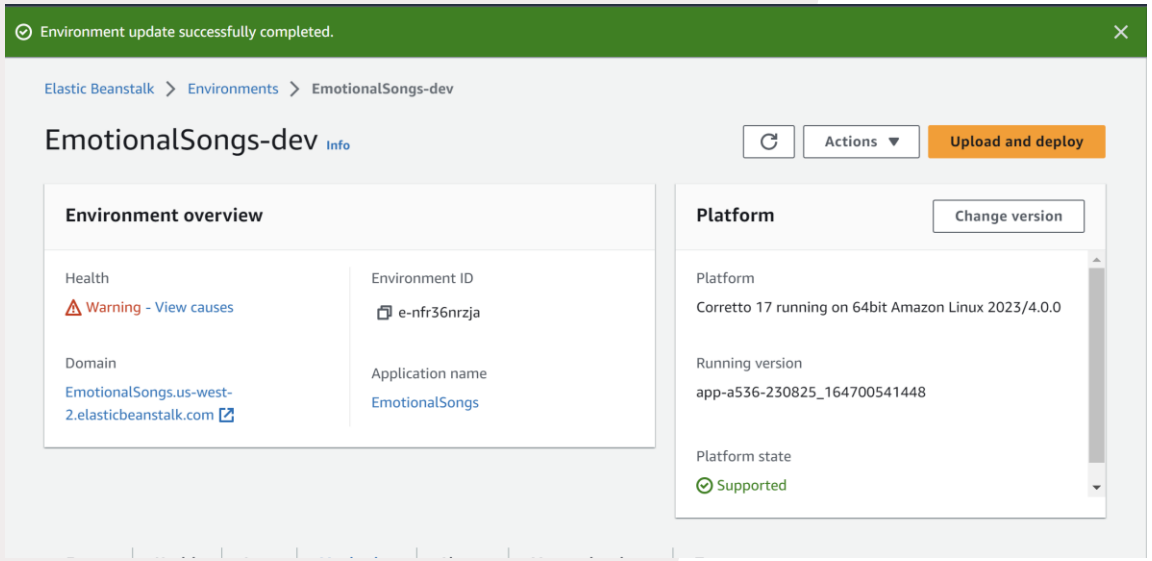
È membro dell'Istituto europeo per le norme di telecomunicazione (ETSI).

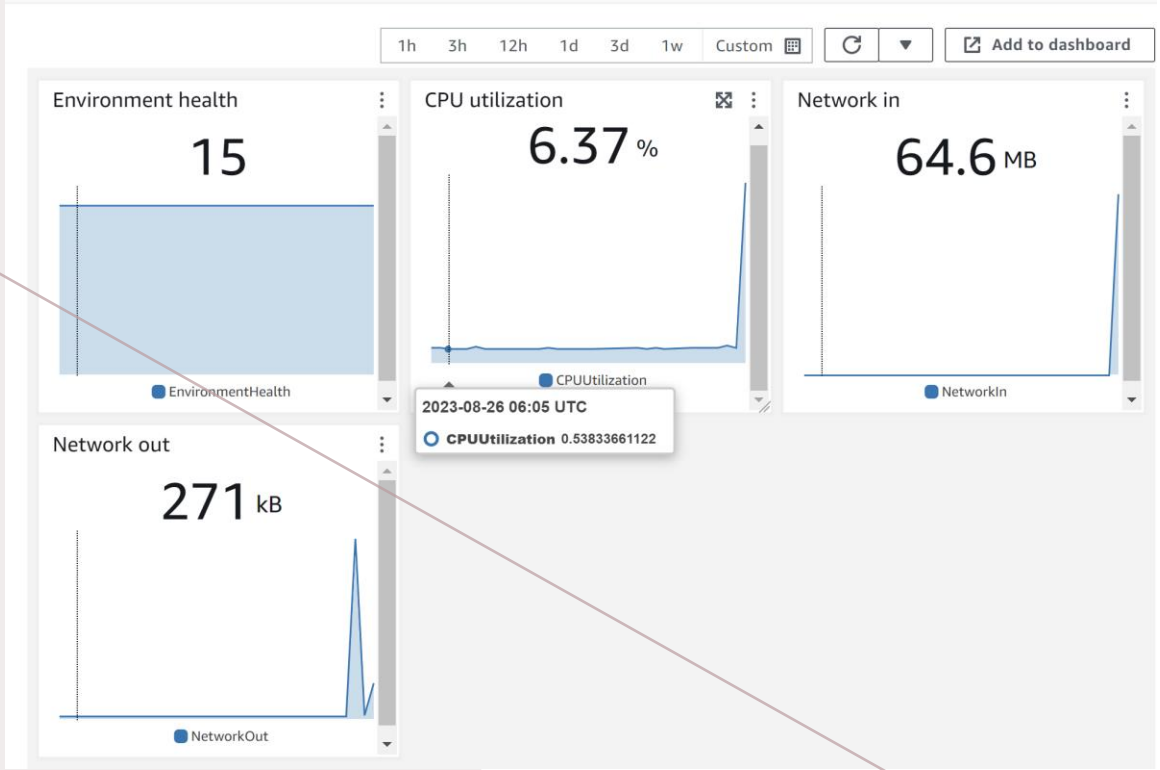
AMAZON RDS

Amazon Relational Database Service (Amazon RDS) è una raccolta di servizi gestiti che rende semplice impostare, operare e scalare i database nel cloud. Fonte: Amazon Web Services.

Nel contesto di EmotionalSongs, il database PostgreSQL è hostato su questa piattaforma, in modo da semplificare lo sviluppo e la manutenzione dello stesso rispetto ad una soluzione di self-hosting.







AMAZON ELASTIC BEANSTALK

Amazon Elastic Beanstalk è un servizio per effettuare il deploy di applicazioni web in modo scalabile e automatico. Oltre all'hosting del container presso IP statico e ai tool di orchestrazione e monitoring offerti, il servizio si occupa della registrazione DNS fornendo un URL da cui accedere all'applicativo.

Fonte: Amazon Web Services.

Nel contesto di EmotionalSongs, il servizio è utilizzato per effettuare il deploy del JAR dell'applicazione in un container Linux con una installazione di Java 17.

DEPLOY APPLICATION TO AWS

Utilizzo di AWS Elastic Beanstalk: Una delle opzioni più comuni per il deploy di applicazioni Java è AWS Elastic Beanstalk.

Elastic Beanstalk semplifica il deploy e la gestione delle applicazioni. Ecco una panoramica dei passaggi:

- **Creazione dell'applicazione:** Accedendo alla console AWS e al proprio account è possibile creare una nuova applicazione.
- **Creazione di un ambiente:** All'interno dell'applicazione, sarà necessario creare un nuovo ambiente. Durante questo processo, si dovrà specificare il tipo di ambiente (ad esempio, ambiente Java), caricare il JAR e configurare le opzioni dell'ambiente come la capacità, l'auto-scaling e le variabili d'ambiente.
- **Deploy dell'applicazione:** Una volta configurato l'ambiente, si può effettuare il deploy del JAR specificando l'URL del repository Git o caricando direttamente il JAR.
- **Monitoraggio e gestione:** Elastic Beanstalk fornisce strumenti per monitorare e gestire l'ambiente creato, inclusi aggiornamenti, rollback e scalabilità.

AWS offre anche la possibilità di eseguire il processo di deploy di un JAR Maven su piattaforme come Elastic Beanstalk o Lambda utilizzando strumenti da linea di comando. Questo consente di automatizzare il processo di deployment e facilita l'integrazione nelle pipeline di sviluppo e nell'automazione dei processi. Utilizzando gli strumenti da linea di comando di AWS, è possibile eseguire l'intero processo di creazione dell'applicazione, configurazione dell'ambiente, upload del JAR e deploy senza la necessità di interagire direttamente con l'interfaccia web di AWS.

La documentazione del processo di Deploy è consultabile all'indirizzo: https://vaadin.com/docs/latest/production/cloud-providers/aws

AUTORI

Emotional Songs è stata sviluppata da:

Acquati Luca, Jamil Muhammad Qasim, Naturale Lorenzo e Volonterio Luca nell'ambito del progetto di Laboratorio Interdisciplinare B per il corso di laurea in Informatica dell'Università degli Studi dell'Insubria.