A.A. 2021/22

EMOTIONAL SONGS

Manuale Tecnico

Versione manuale 1.1

Progetto svolto da:

Della Chiesa Mattia - 749904

Indice

Librerie e classi esterne utilizzate

Durante lo sviluppo dell'applicazione sono state utilizzate le seguenti classi e librerie esterne (non definite all'interno del package emotionalsongs):

- java.lang.management.**ManagementFactory**

  La classe **ManagementFactory** viene utilizzata per ottenere il process id (**PID**) dell'applicazione, permettendo così di effettuare la chiusura automatica del prompt dei comandi una volta terminata l'esecuzione del programma.

  Problema riscontrato durante lo sviluppo: La chiusura della finestra risulta inconsistente, data la natura del progetto, si è deciso di tralasciare il problema.

- java.util.**ResourceBundle**;

  La classe **ResourceBundle** permette lo sviluppo di un'applicazione altamente indipendente dalla lingua parlata dall'utente; isolando così la maggior parte, se non tutte, le informazioni specifiche alla localizzazione nei bundle di risorse.

- org.apache.commons.lang3.**StringUtils**

  La classe **StringUtils** permette l'utilizzo di metodi come `trim(String)`, `capitalize(String)` e `isEmpty(String)` che semplificano la gestione di oggetti di tipo String all'interno della piattaforma.

- java.io.**BufferedReader** e java.io.**BufferedWriter**;

  Le classi **BufferedReader** e **BufferedWriter** permettono di effettuare la lettura (*scrittura*) di sequenze di caratteri da (*in*) un buffer di lettura (*scrittura*).

- java.io.**File**;

  La classe **File** implementa una rappresentazione astratta dei percorsi di file e directory (*assoluti e relativi*), permettendo così all'applicazione di effettuare operazioni su dati presenti all'interno della memoria di massa.

- java.io.**FileNotFoundException**;

  La classe **FileNotFoundException** viene utilizzata all'interno dell'applicazione per segnalare che il file richiesto non è stato trovato in memoria di massa.

- java.io.**FileReader** java.io.**FileWriter**;

  Le classi **FileReader** e **FileWriter** permettono di effettuare la lettura (*scrittura*) di caratteri all'interno di File presenti in memoria di massa.

- java.io.**IOException**;

  La classe **IOException** viene utilizzata all'interno dell'applicazione per segnalare è stato riscontrato un errore durante l'esecuzione di un'operazione di Input/Output.

- java.util.**Arrays**;

  La classe **Arrays** mette a disposizione metodi come `fill(Object)`, `asList()` e `copyOfRange(int, int)` che semplificano l'utilizzo di array all'interno dell'applicazione.

- java.util.**Map** e java.util.**HashMap**;

  L'interfaccia **Map** e la sua implementazione **HashMap** permettono l'utilizzo delle tabelle hash come struttura dati all'interno dell'applicazione. Per visualizzare una tabella riassuntiva delle principali strutture dati utilizzate, consultare la sezione Strutture dati.

- java.util.**ArrayList**;

  La classe **ArrayList** - implementazione dell'interfaccia **List** - permette l'utilizzo di strutture dati Lista all'interno dell'applicazione. Per maggiori informazioni sul loro utilizzo, consultare la sezione Strutture dati.

- java.util.concurrent.**ThreadLocalRandom**;

  La classe **ThreadLocalRandom** viene utilizzata in alternativa al metodo `Math.random()` in quanto permette di semplificare la generazione di valori pseudo-casuali all'interno di un range predefinito.

- java.util.**Scanner**;

  La classe **Scanner** permette l'utilizzo di un semplice scanner di testo in grado di effettuare il parse di tipi e stringhe primitive suddividendo il suo input in token mediante un delimitatore (impostato di default a '`\s`', il whitespace). Questi token potranno essere convertiti in valori di diversi tramite i vari metodi "next".

- java.util.**Set** e java.util.**LinkedHashSet**;

  L'interfaccia **Set** e la sua implementazione **LinkedHashSet** permetto l'utilizzo insiemi all'interno dell'applicazione. Nello specifico questa classe riveste un ruolo chiave all'interno della funzione della ricerca delle canzoni per titolo in quanto quest'ultima opera mediante l'intersezione di insiemi.

- java.time.**LocalDateTime**;

  La classe **LocalDateTime** viene utilizzata all'interno dell'applicazione per implementare dei controlli basilari all'interno dell'applicazione grazie al metodo `LocalDateTime.now().getYear()`

- java.util.**UUID**;

  La classe **UUID** permette di generare un UUID (**universally unique identifier**) di tipo 3 mediante l'algoritmo di hashing MD5 (**Message Digest 5**). Questa classe riverse un ruolo chiave all'interno della maggior parte dei file con estensione ".dati".

Le classi definite all'interno del package

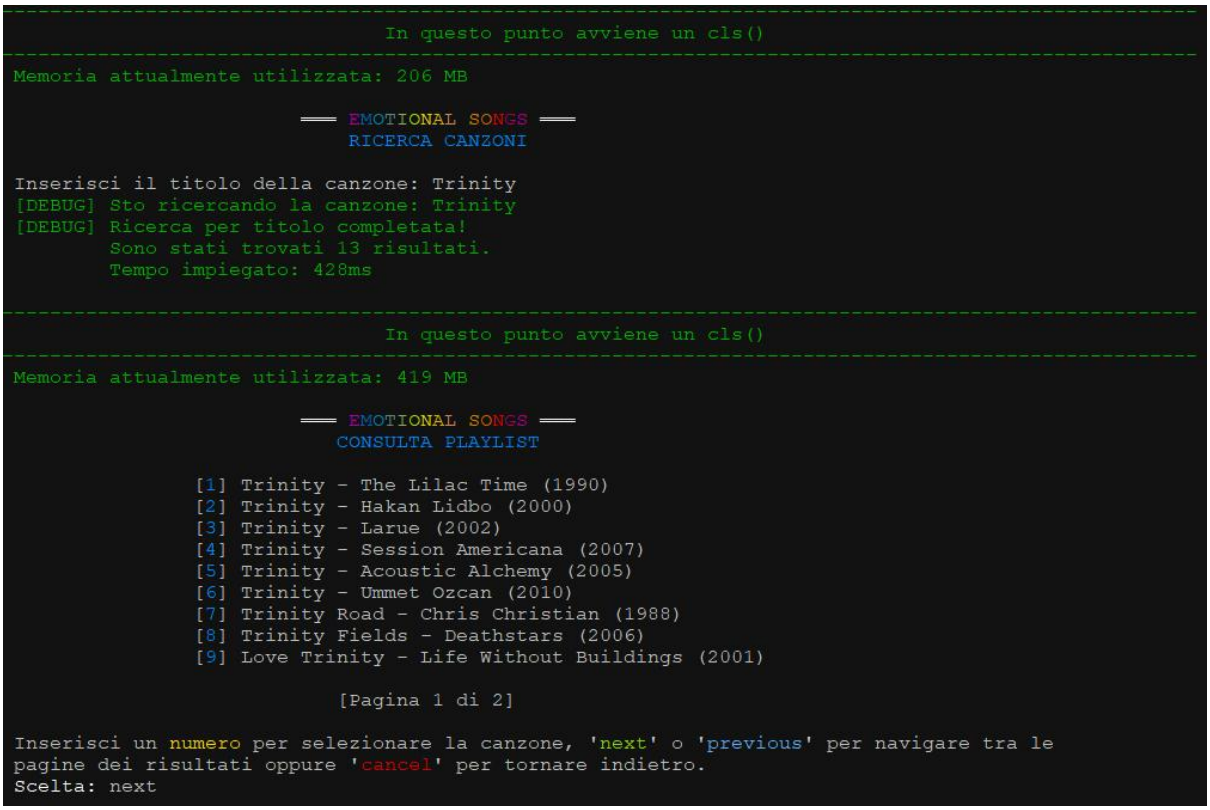
All'interno del package **emotionalsongs** sono state definite sedici classi, di cui una enumerativa:

EmotionalSongs

La classe **EmotionalSongs** è la classe principale dell'applicazione. In essa si trova il metodo main assieme ad altri metodi utili alla rappresentazione dei vari menu presenti all'interno dell'applicazione. All'interno di questa classe viene inoltre effettuato il parse degli argomenti forniti all'avvio dell'applicazione.

Nello specifico, i comandi riconosciuti dal programma sono i seguenti:

- **setDebug**
  L'argomento **setDebug** permette di abilitare la modalità di debug all'interno dell'applicazione. Questa modalità, oltre che a sostituire la "pulizia" dello schermo con un semplice messaggio di debug (*vedi immagine posta di seguito*), permetterà all'utente di visualizzare a schermo dei messaggi utili a descrivere lo stato del programma durante l'esecuzione. La sintassi per abilitare la modalità di debug è la seguente:
  `>java -jar EmotionalSongs setDebug:true`



- **setLanguage**
  L'argomento **setLanguage** permette di eseguire l'applicazione nel linguaggio desiderato. Le lingue attualmente supportate dall'applicazione sono Italiano e Inglese.
  I nomi delle lingue sono espressi nella loro lingua nativa (*Italiano, English, Deutsche, etc..*). La sintassi per impostare il linguaggio dell'applicazione in Inglese è la seguente:
  `>java -jar EmotionalSongs setLanguage:English`

  Bug: Per come è definita la struttura del metodo main, il messaggio "[DEBUG] La modalità di debug e' stata attivata." verrà sempre visualizzato in Italiano.

Canzone

La classe Canzone rappresenta un brano musicale presente all'interno della repository delle canzoni ed implementa diversi metodi getter per ottenere le informazioni che caratterizzano una canzone. Ogni istanza di quest'oggetto è caratterizzata dai seguenti campi:

- Il titolo della canzone.
- L'autore del brano musicale.

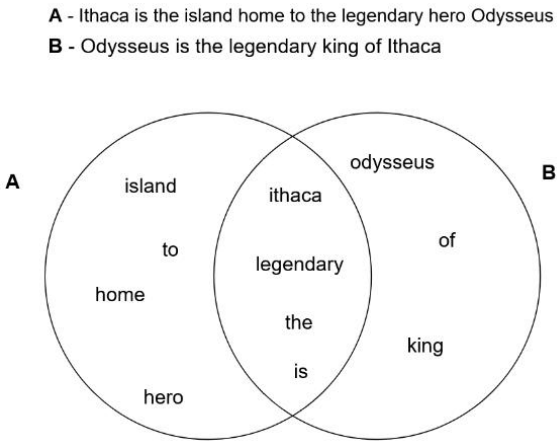
- L'anno di pubblicazione.
- Un UUID di tipo 3 generato tramite i valori assunti dai campi precedenti.

All'interno della classe è presente un metodo `toCSV()` che permette di formattare i dati sopra elencati in una stringa che rispetta lo standard CSV (**Comma Separated Values**). Questo metodo assume un ruolo di particolare rilevanza durante la fase di salvataggio dei dati in memoria di massa.

SongRepositoryManager

Di particolare rilevanza è la classe **SongRepositoryManager** in quanto, oltre a definire la struttura vera e propria della repository delle canzoni, implementa diversi metodi utili alla consultazione e alla ricerca di brani musicali presenti all'interno di essa.

La struttura dati scelta per rappresentare il repertorio delle canzoni è la struttura dati **HashMap** in quanto, in questo ambito applicativo, l'ordine con cui gli elementi vengono posti all'interno della memoria centrale non è essenziale.



Infatti, la funzione di ricerca per titolo dei brani musicali all'interno della repository non sfrutta tecniche che beneficiano dell'ordine lessicografico degli elementi ma bensì si limita semplicemente a generare un set di parole per entrambi i titoli (*il titolo ricercato e il titolo di una canzone in repertorio*) e, mediate l'operazione di intersezione, calcola un punteggio che rappresenta il numero di elementi comuni tra i due insiemi (*operazione di intersezione*).

Ne consegue che, seppur l'utilizzo di questa struttura dati comporti un incremento delle risorse necessarie durante l'esecuzione rispetto ad una struttura dati meno "esigente" come ad esempio una ArrayList, si è prediletto il costo costante **O(1)** durante le operazioni di lettura e scrittura all'interno della struttura dati.

All'interno della HashMap, ogni istanza degli oggetti di tipo **Canzone** viene mappata alla propria chiave rappresentata dall'**UUID** del brano musicale. Si ricorda che, essendo la repository gestita in maniera statica, prima di usufruire di quest'ultima sarà necessario inizializzarla mediante il metodo `SongRepositoryManager.build()`.

SearchResult

La classe **SearchResult** implementa l'interfaccia **Comparable<T>** e permette di rappresentare un risultato restituito dalle funzioni di ricerca. Ogni istanza di quest'oggetto è caratterizzata da due campi: il campo **canzone**, un riferimento ad un istanza di un oggetto **Canzone** e un campo score, che contiene un valore decimale che rappresenta il punteggio calcolato mediante la funzione di ricerca. Quest'ultimo viene semplicemente calcolato mediante la seguente formula:

$$\frac{|A \cap B|}{\mathrm{Max}(|A|, |B|)}$$

Dove **A** è l'insieme delle parole che compongono il titolo di una canzone presente all'interno della repository e **B** è l'insieme delle parole che compongono il titolo ricercato. Qualora la funzione di

ricerca ottenga un match del 100% tra i due titoli , per far si che il risultato appaia in cima alla lista, il punteggio assumerà come valore `1.1`.

Il metodo `compareTo(SearchResult)` opera facendo un semplice confronto sui risultati assunti dal campo `score`.

$$\begin{cases} -1 & \text{sse this} < \text{S} \\ 0 & \text{sse this} = \text{S} \\ 1 & \text{sse this} > \text{S} \end{cases}$$

Dove **S** è un'altra istanza di un oggetto **SearchResult**.

Language e LanguageEnglish

Le classi **Language** e **LanguageEnglish** sono estensioni della classe astratta **ListResourceBundle**, sottoclasse di **ResourceBundle.** Quest'ultima permette di gestire le risorse di localizzazione in maniera efficiente e semplice da utilizzare. Entrambe le classi **Language** e **LanguageEnglish** effettuano l'override del metodo `getContents()`, il quale restituisce un array (*bidimensionale*) in cui ciascun elemento è una coppia di oggetti dove il primo assume il ruolo di chiave (*avente come vincolo il tipo* **String**), mentre il secondo elemento è il valore associato a quest'ultima.

Ad esempio, nella coppia di elementi:
`{"NoEmotionsRegisteredForSong","Non sono presenti emozioni registrate per la canzone:"},`

"**NoEmotionsRegisteredForSong**" assume il ruolo di **chiave** mentre
"**Non sono presenti emozioni registrate per la canzone:**" assume il ruolo di **valore** associato a quest'ultima.

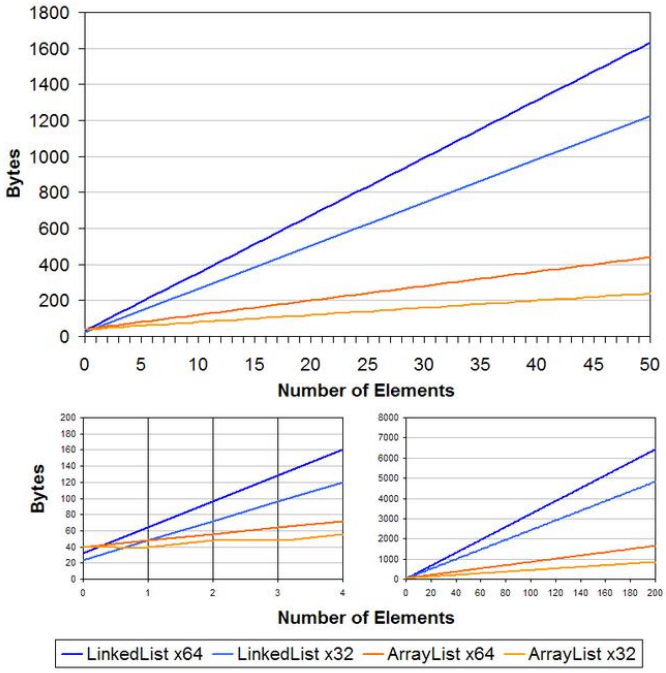
Per ottenere il valore associato alla chiave sarà sufficiente invocare il metodo `getString(String)` avente come parametro la chiave desiderata.

NOTA: Si è deciso di non utilizzare la nomenclatura convenzionale (e.g. *Language_en.java, Language_it.java, Language_de.java, etc.*.) in quanto si è preferito che l'applicazione venga eseguita direttamente in lingua Italiana cosa che, stando alle ricerche effettuate, non sarebbe stata possibile senza l'utilizzo della classe **Locale** poiché, senza quest'ultima, l'applicazione prediligerebbe l'utilizzo dei file di localizzazione Inglese (*Language_en*).

EmotionsRepositoryManager

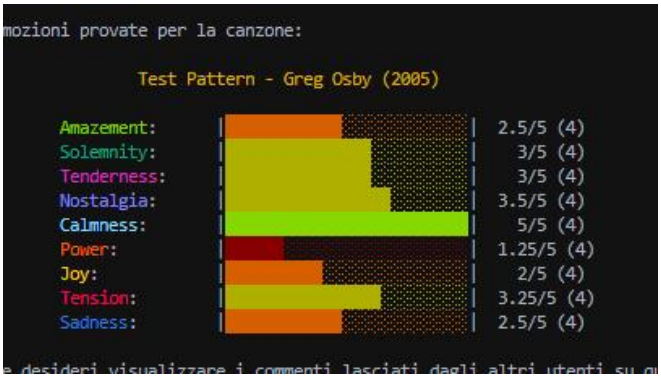
Simile alla classe **SongRepositoryManager**, **EmotionRepositoryManager** rappresenta la repository delle emozioni, ovvero, la collezione di tutti i feedback forniti da parte degli utenti raggruppati secondo l'UUID della canzone valutata. Anche in questo caso si è deciso di utilizzare come struttura dati una **HashMap** avente come chiave l'**UUID** associato alla canzone; la differenza sostanziale consiste negli elementi mappati alle varie chiavi. Questi infatti saranno delle istanze della classe **ArrayList** che, a loro volta, conterranno istanze di tipo SongEmotions (`ArrayList<SongEmotions>`). Quest'ultima struttura dati è stata preferita rispetto ad una struttura dati LinkedList in quanto offre prestazioni migliori in termini di tempo e spazio[1]:

[1] When to use LinkedList over ArrayList - Stack Overflow



Credit: [Masterfego]

Bug: Il metodo `printScores()` ha dei problemi di allineamento del testo con valori decimali. Essendo un problema di natura puramente estetica, gli è stata assegnata una priorità bassa. Purtroppo non è stato possibile correggere questo problema entro la scadenza del progetto.



SongEmotions

La classe **SongEmotions** permette di rappresentare il feedback fornito da parte di un utente durante l'ascolto di una canzone. Ogni istanza di questo oggetto è caratterizzata dai seguenti campi:

- L'**UUID** delle canzone
- L'**userId** associato all'utente che ha fornito il feedback
- I **punteggi** associati a ciascuno dei nove stati emozionali (*vedi sotto*)
- I **commenti** relativi a ciascuna di queste emozione.

In particolare, gli ultimi due campi menzionati sono rispettivamente degli array di interi e di stringhe. Si è deciso di utilizzare questa rappresentazione di natura relativamente semplice in quanto la struttura degli array permette di definire ai propri elementi, in maniera completamente implicita, un ordine compatibile con quello definito per i nove stati emozionali.
Ad esempio, l'elemento dell'array avente indice zero (`0`) sarà mappato al tag emozionale presente

alla 0-esima posizione (`AMAZEMENT`). Per maggiori informazioni sull'ordine dei nove (9) tag emozionali si consiglia di consultare la classe **Emotions** posta qui sotto.

Nota: l'array dei commenti, nel caso l'utente non abbia fornito alcun commento, sarà costituito interamente da stringhe vuote (`""`). Mentre nel caso l'utente non abbia fornito un punteggio per una data emozioni, essa avrà come punteggio `0`.
Si è deciso di utilizzare questo approccio in quanto è sembrato più sensato lasciare che il punteggio `1` venga esplicitamente fornito da parte dell'utente.

Emotions

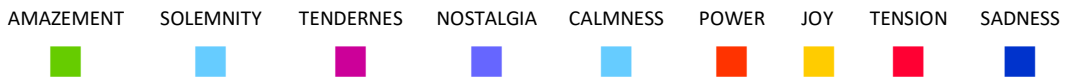
La classe enumerativa **Emotions** rappresenta i nove (9) tag emozionali provati durante l'ascolto di un brano musicale.

L'ordine delle costanti enumerative è così definito:

AMAZEMENT, SOLEMNITY, TENDERNESS, NOSTALGIA, CALMNESS, POWER, JOY, TENSION, SADNESS;

All'interno della classe vengono inoltre implementati due metodi:

- `emotionColors`, restituisce un codice ANSI che permette di cambiare il colore di foreground del testo visualizzato sul terminale al colore associato all'emozione:

  Nel caso il supporto per i codici di escape ANSI sia disabilitato (vedi classe TextUtils), verrà restituita una stringa vuota (`""`).

- `getColorFromOrdinal`, restituisce il codice di escape ANSI associato all'emozione rappresentata dalla sua posizione ordinale (*vedi sopra*).

Per maggiori informazioni riguardo al funzionamento dei codici di escape ANSI all'interno dell'applicazione, si consiglia di consultare la classe **TextUtils** presente all'interno di questa sezione.

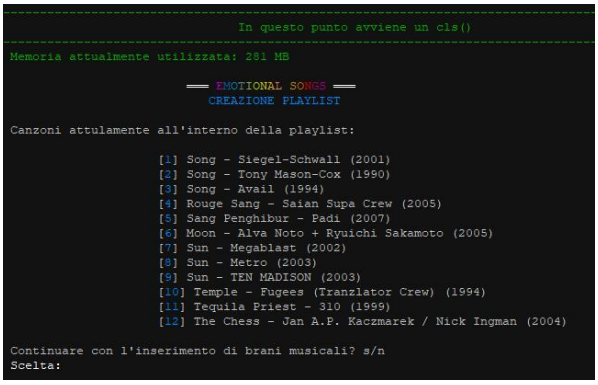
Playlist

Altra classe che riveste un ruolo di particolar rilievo all'interno dell'applicazione è la classe **Playlist** in quanto permette di rappresentare una playlist creata da un utente. Ogni istanza di questo oggetto è caratterizzata dai seguenti campi:

- Una lista di **canzoni**
- Il **nome** della playlist
- L'**username** dell'utente che ha creato la playlist

Per rappresentare la lista di canzoni presenti all'interno della playlist si è deciso di utilizzare la struttura dati **ArrayList** in quanto, come già precedentemente descritto all'interno del paragrafo dedicato alla classe **EmotionsRpositoryManager**, essa offre migliori prestazioni rispetto ad una **LinkedList**. All'interno di questa classe è presente il metodo `registraPlaylist()` che, guida l'utente durante la fase di creazione della playlist. L'aggiunta di brani musicali alla playlist potrà essere ripetuta finché lo si desidera. Inoltre, all'interno del metodo sono stato implementati controlli che impediscono l'inserimento di valori duplicati assicurando così che ogni canzone potrà essere inserita una ed una sola volta.

Una minor limitazione riscontrata in questo metodo consiste nel fatto che nel visualizzare i brani attualmente presenti all'interno della playlist, questi non vengono suddivisi per pagine come all'interno di molti altri metodi ma bensì vengono presentati in un unico elenco; questa limitazione è puramente di carattere estetico in quanto non va ad impattare le prestazioni dell'applicazione.



All'interno della classe è presente un metodo `toCSV()` che permette di formattare i dati sopra elencati in una stringa che rispetta lo standard CSV (**Comma Separated Values**). Questo metodo assume un ruolo di particolare rilevanza durante la fase di salvataggio dei dati in memoria di massa. Questo salvataggio viene effettuato all'interno della classe **PlaylistManager** tramite il metodo `scriviPlaylist()`.

PlaylistManager

La classe PlaylistManager, oltre che a gestire il salvataggio dei dati relativi alle playlist in memoria di massa, implementa diverse operazioni utili alla gestione delle playlist durante l'esecuzione dell'applicazione. All'interno di questa classe si trova anche una specie di "repertorio" delle playlist tuttavia, a differenza delle altre repository presenti all'interno dell'applicazione, quest'ultima conterrà esclusivamente i dati relativi alle playlist dell'utente attualmente loggato. Questo perché non si è ritenuto necessario mantenere in memoria delle istanze di Playlist appartenenti ad altri utenti della piattaforma in quanto, ciascun utente è in grado di consultare esclusivamente le proprie playlist.

Ciò motiva la decisione di includere all'interno della classe un metodo `clearPlaylists();` ciò assicura che, una volta effettuato il logout, la repository delle playlist venga annullata (assumendo valore `null`); Lo scopo di ciò è evitare situazioni non previste in cui un secondo utente possa essere in grado di consultare le playlist appartenenti a colui che ha precedentemente usufruito dell'applicazione senza effettuane la chiusura.

Similmente a quanto detto nei confronti della repository delle canzoni, prima di usufruire della repository delle emozioni, sarà necessario inizializzarla mediante il metodo `PlaylistManager.build()`.

UserAuthenticationManager

La classe UserAuthenticationManager si occupa di gestire tutte le operazioni necessarie durante le fasi di login e sign-up.

Durante la fase di registrazione, vengono effettuati dei semplici controlli sui dati immessi dall'utente; nello specifico:

- Il **nominativo** immesso deve contenere almeno due parole separate da uno spazio
- Il **codice fiscale** dev'essere valido (*il checksum deve risultare corretto*) e in forma normalizzata (*l'applicazione non accetta codici fiscali temporanei di lunghezza pari a 11 caratteri*). Il codice sorgente per i controllo effettuati su questo valore è stato fornito dal

sito web www.icosaedro.it. Il codice fiscale immesso NON dev'essere già presente all'interno dell'archivio utenti.

- L'**indirizzo email** deve contenere i caratteri '`@`' e '`.`'
- L'**username** inserito NON dev'essere già presente all'interno dell'archivio utenti
- La **password** NON dev'essere vuota

Su tutti questi valori viene invocato il metodo `StringUtils.trim()` che rimuove dalla stringa tutti i caratteri di controllo (*avente codice ASCII*[2] *<= 32*) e tutti gli spazi vuoti prefissi e post-fissi (e.g. "`\0x07  Mario Rossi  `" => "`Mario Rossi`").

Nota: Durante lo sviluppo dell'applicazione, per pura semplicità, si è deciso di mantenere le credenziali degli utente memorizzate in chiaro all'interno del file **UtentiRegistrati.dati**. Un alternativa a questo approccio sarebbe stato l'utilizzo di algoritmi di crittografia oppure aggiungendo alla password dell'utente una stringa casuale "salt" prima di effettuarne l'hashing.

TextUtils

La classe TextUtils implementa diverse funzionalità utili alla lettura, rappresentazione e validazione dei dati durante l'esecuzione dell'applicazione.

Tutte le funzionalità di input dei dati sono null-safe, ovvero, si assicurano che i valori immessi da parte dell'utente non risultino essere nulli o invalidi (e.g. il metodo `TextUtils.readInt` si assicurerà che, tramite l'utilizzo del metodo `TextUtils.isNumeric`, la procedura di inserimento dei valori verrà ripetuta finché il dato immesso non rispecchia un valore intero valido).

La classe si occupa inoltre di gestire la modalità di debug, ponendo a disposizione metodi che permettono di visualizzare informazioni utili al processo di troubleshooting.

Di particolare rilevanza è il supporto fornito dalla classe per i codici di escape ANSI che permettono di modificare l'aspetto del testo visualizzato sul terminale. Tuttavia, per rendere visibili queste modifiche, sarà necessario che sia presente la **DWORD** da 32 bit `VirtualTerminalLevel` all'interno del registro di Windows e che essa abbia come valore `1`. Di default, questa DWORD non è presente e, di conseguenza andrà aggiunta manualmente. Dacché non si può esigere che l'utente desideri o abbia privilegi sufficientemente elevati per modificare i registri di Windows, si è deciso di lasciare questa funzionalità **completamente opzionale**. Tuttavia, poiché la mancata presenza di questa entry all'interno del registro di Windows renderebbe il testo illeggibile, vi è stata necessità di includere un messaggio informativo che verrà visualizzato ogni qualvolta che l'applicazione viene lanciata. In quest messaggio, verrà chiesto all'utente se il testo visualizzato appare privo di colori o caratteri illeggibili.

A seconda della risposta fornita, i codici di escape ANSI rimarranno attivi oppure verranno sostituiti con delle stringhe vuote (`""`).

Questo giustifica la scelta effettuata nei confronti delle "variabili colore" presenti all'interno della classe; queste, seppur la loro nomenclatura segua la convenzione definita per le costanti (*che comporta gli identificatori delle variabile scritti in maiuscolo*), non possiedono il modificatore final poiché, qualora necessario, il loro valore verrà modificato nella stringa vuota mediante il metodo `disableANSIColorCodes(boolean)`**.**

[2] Tabella ASCII

Onde evitare la necessità di definite un secondo metodo atto all'inizializzazione di queste variabili, si è deciso di far si che, se invocato con parametro false, si occuperà di inizializzare le variabili con i valori restituiti dalla funzione `fromRGB()`.

Questa funzione ha lo scopo di semplificare la gestione delle varie sfumature di colore (*216 colori + 24 sfumature di grigio aventi range di valori 16-255*) permettendo così di fornire in input un semplice colore in formato RGB e di ottenere come valore di ritorno il codice di escape ANSI che risulta essere il più vicino al colore desiderato *(con una variazione massima di circa ± 25 unità di colore)*.

Ad esempio, invocando la funzione `fromRGB()` con parametri: 225 ,41, 149 (■) verrà calcolato un indice mediante la seguente formula:

$$36 \cdot \mathrm{rnd}(\frac{225 * 5}{255}) + 6 \cdot \mathrm{rnd}(\frac{41 * 5}{255}) + \mathrm{rnd}(\frac{149 * 5}{255}) = 144 + 6 + 3 = 153$$

Che individuerà all'interno dell'array dei colori (`colorvalues[153]`) l'escape code"`\033[38;5;169m`" (■) dove il valore 169 presente all'interno del codice di escape è dato dalla somma tra l'indice calcolato e 16.

Nota: Quest'implementazione è senz'ombra di dubbio ottimizzabile in quanto, piuttosto che salvare tutti gli escape code all'interno di un array di stringhe, sarebbe la soluzione ottimale consisterebbe nel generare l'escape code direttamente al momento del calcolo di suddetto indice (*sostituendo i doppi trattini all'interno del seguente dummy code* `\033[38;5;--m` *con l'indice incrementato di 16 unità*) tuttavia, non essendo una qualcosa essenziale allo svolgimento del progetto, a quest'ottimizzazione è stata assegnata una priorità bassa.

Utente

La classe Utente rappresenta un utente della piattaforma ed implementa diversi setter e getter per gestire le informazioni che caratterizzano le istanze di questo oggetto. Ciascuna di queste istanze è caratterizzata dai seguenti campi:

- **Nominativo**, Nome e Cognome della persona fisica separati da uno spazio.
- **codiceFiscale**, rappresenta un codice fiscale normalizzato di lunghezza pari a 16 caratteri
- **indirizzo**, un'istanza di un oggetto Indirizzo che rappresenta l'indirizzo toponomastico dell'utente.
- **email**
- **userId**, l'username scelto dall'utente.
- **password**

All'interno della classe è presente un metodo `toCSV()` che permette di formattare i dati sopra elencati in una stringa che rispetta lo standard CSV (**Comma Separated Values**). Questo metodo assume un ruolo di particolare rilevanza durante la fase di salvataggio dei dati in memoria di massa.

Indirizzo

La classe indirizzo permette di rappresentare un indirizzo toponomastico suddiviso nelle sue componenti atomiche:

- La **via**
- Il **numero civico**
- Il **codice di avviamento postale**
- La **città**
- La **provincia**

Anche qui, come per la classe precedente presente un metodo `toCSV()` che permette di formattare i dati che caratterizzano un'istanza della classe Indirizzo in una stringa che rispetta lo standard CSV (**Comma Separated Values**). Questo metodo assume un ruolo di particolare rilevanza durante la fase di salvataggio dei dati in memoria di massa.

AddressNotValidException

Il costruttore della classe Indirizzo potrebbe lanciare una **AddressNotValidException** qualora i dati forniti in input assumono dimensione diversa da quella attesa.

Ad esempio, quest'eccezione verrà lanciata da parte del costruttore qualora a quest ultimo viene passato un array vuoto oppure un array avente dimensione diversa da 5.

Tabella riassuntiva delle strutture dati utilizzate

Di seguito sono riportate le principale strutture dati utilizzate durante la fase di sviluppo dell'applicazione. Per maggiori informazioni sulle motivazioni che hanno portato alla scelta delle seguenti strutture dati, consultare la sezione "Le classi definite all'interno del package".

Elemento rappresentato	**Struttura dati scelta**
Repository delle Canzoni	HashMap<String, Canzone>
Repository delle Playlist	HashMap<String, Playlist>
Repository delle Emozioni	HashMap<String, ArrayList<SongEmotions>>
Playlist	ArrayList<Canzone>
Elenco risultati della ricerca	ArrayList<SearchResult>

Potenziali eccezioni sollevate nel codice

Di seguito viene posto un elenco - nel formato `classe:: righe` - delle possibili eccezioni controllate che si posso presentare durante l'esecuzione del codice:

NumberFormatException

EmotionalSongs:: 109-259

SongRepositoryManager:: 145-152

TextUtils:: 302-314

TextUtils:: 765-774

IOException

EmotionalSongs:: 264-270

EmotionRepositoryManager:: 426-437

EmotionRepositoryManager:: 443-475

EmotionRepositoryManager:: 1121-1135

EmotionRepositoryManager:: 1143-1175

Playlist:: 410-452

Playlist:: 485-499

SongRepositoryManager:: 39-67

UserAuthenticationManager:: 202-260

UserAuthenticationManager:: 298-329

UserAuthenticationManager:: 348-362

UserAuthenticationManager:: 382-400

FileNotFoundException

EmotionRepositoryManager:: 1143-1169

AddressNotValidException

UserAuthenticationManager:: 215-224

IndexOutOfBoundsException

UserAuthenticationManager:: 281-284

Nota: A seconda del contesto in cui verranno sollevate le eccezioni è possibile che all'utente vengano visualizzati dei messaggio di errore che lo incitano a contattare il supporto tecnico.
Di norma, si è tentato di includere all'interno del messaggio di errore la natura dell'eccezione lanciata.