**Manuale Tecnico - Book Recommender**

**Autore:**

* Nome: Alexandru
* Cognome: Raita
* Matricola: 757601
* Sede: VA

**Data:** 9 Settembre 2024

**Versione Documento:** 1.0

**Indice**

1. [Report tecnico della soluzione sviluppata](#report)
   * [Architettura dell'applicazione](#architettura)
   * [Strutture dati utilizzate](#strutture)
   * [Algoritmi](#algoritmi)
   * [Formato dei file e gestione](#formato)
   * [Pattern utilizzati](#pattern)
2. [Limiti della soluzione sviluppata](#limiti)
3. [Sitografia / Bibliografia](#sitografia)

**Report tecnico della soluzione sviluppata**

**Architettura dell’Applicazione**

L'architettura di **Book Recommender** è organizzata seguendo il pattern **MVC (Model-View-Controller)**, un modello che separa chiaramente le responsabilità di gestione dei dati (Model), la logica di controllo (Controller) e la presentazione all'utente (View).

**1. Model (Modello)**

Il **model** contiene le classi che definiscono le entità principali del dominio. Queste classi sono responsabili della gestione dei dati e della loro persistenza. I modelli utilizzati nel progetto sono **Libro**, **Libreria**, **Utente**, **LibroConsigliato**, **Valutazione**.

**2. View (Vista)**

Le viste, realizzate tramite il framework **Swing**, forniscono all'utente un'interfaccia grafica per interagire con l'applicazione. Ogni finestra o componente visivo è costruito utilizzando le componenti Swing come JPanel, JFrame, JButton, JTextField, ecc...  
  
I file sono organizzati all’interno di diversi package in base al ruolo ricoperto nel progetto. Ad esempio *“gui.auth”* contiene le classi relative all’interfaccia utente per il login e la registrazione.

**3. Controller (Controllo)**

I **controller** sono responsabili della logica di business e della gestione dei dati. Interagiscono sia con le viste che con i modelli, orchestrando le operazioni logiche.

Ad esempio, la classe GestioneUtente gestisce la logica di autenticazione e registrazione degli utenti. Alcune di queste utilizzano il **pattern Singleton** per essere istanziate, in quanto a volte è utile avere un’unica istanza per tutta l’applicazione (ad esempio nella gestione dell’accesso dell’utente).

Esempio di **controller**:

package controller;

import java.util.List;

import model.Utente;

import utility.CSVReaderWriter;

/\*\*

\* La classe GestioneUtente fornisce metodi per la gestione degli utenti, inclusi

\* la registrazione, l'autenticazione e il login.

\*/

public class GestioneUtente {

private CSVReaderWriter csvManager;

private String fileCSV;

private static GestioneUtente instance;

/\*\*

\* Costruttore privato della classe. Utilizza il pattern Singleton.

\*/

private GestioneUtente() {

this.csvManager = CSVReaderWriter.getInstance();

this.fileCSV = "UtentiRegistrati.csv";

}

/\*\*

\* Restituisce l'istanza Singleton di GestioneUtente.

\* Se non esiste, viene creata una nuova istanza.

\*

\* @return L'istanza Singleton di GestioneUtente.

\*/

public static synchronized GestioneUtente getInstance() {

if(instance == null) {

instance = new GestioneUtente();

}

return instance;

}

/\*\*

\* Registra un nuovo utente nel sistema.

\* Se l'email o lo username sono già presenti, viene lanciata un'eccezione.

\*

\* @param nome Nome dell'utente.

\* @param cognome Cognome dell'utente.

\* @param codiceFiscale Codice fiscale dell'utente.

\* @param email Email dell'utente.

\* @param userId UserID per l'accesso al sistema.

\* @param password Password per l'accesso al sistema.

\* @throws UtenteEsistenteException Se l'utente esiste già.

\*/

public void registraUtente(String nome, String cognome, String codiceFiscale, String email, String userId, String password) throws UtenteEsistenteException {

String[] datiUtente = {nome, cognome, codiceFiscale, email, userId, password};

if (csvManager.indexIsEqualToCSV(fileCSV, datiUtente, new int[]{3})) {

throw new UtenteEsistenteException("Utente già esistente!);

}

if (csvManager.indexIsEqualToCSV(fileCSV, datiUtente, new int[]{4})) {

throw new UtenteEsistenteException("Utente già esistente!);

}

csvManager.scriviSuCSV(fileCSV, datiUtente);

}

/\*\*

\* Verifica le credenziali dell'utente per l'accesso al sistema.

\*

\* @param userId UserID dell'utente.

\* @param password Password dell'utente.

\* @return true se l'autenticazione ha successo, false altrimenti.

\*/

public boolean checkCredenziali(String userId, String password) {

List<String[]> contenutoCSV = csvManager.leggiDaCSV(fileCSV);

return contenutoCSV.stream()

.anyMatch(riga -> riga.length == 6 && riga[4].equals(userId) && riga[5].equals(password));

}

/\*\*

\* Effettua il login di un utente e imposta l'utente loggato nella sessione corrente.

\*

\* @param userId UserID dell'utente.

\* @param password Password dell'utente.

\* @return L'oggetto Utente se il login ha successo, null altrimenti.

\*/

public Utente loginUtente(String userId, String password) {

String[] datiUtente = csvManager.getAllData(fileCSV, userId, 4);

if (datiUtente != null && datiUtente.length >= 6) {

//etc..

Questo controller utilizza la classe **CSVReaderWriter** per leggere e scrivere i dati nel file CSV degli utenti registrati. Le credenziali sono validate confrontando i dati presenti nel file.

**Strutture dati utilizzate**

Le strutture dati utilizzate nell'applicazione sono principalmente liste ed array.

**1. Liste (List)**

Le collezioni principali sono gestite tramite interfacce **List** di Java, in quanto permettono l'aggiunta, rimozione e ricerca dinamica di elementi.

La scelta di **ArrayList** è ottimale per le operazioni di lettura e aggiunta sequenziale, che sono le più frequenti in un contesto come quello delle librerie di libri.

**2. Array**

Gli **array** vengono utilizzati in situazioni dove il numero di elementi è noto in anticipo o dove l'accesso ai dati deve essere estremamente veloce.

Esempio di utilizzo di un array di libri consigliati:

public class LibroConsigliato {

private String utenteId;

private String libroPrincipale;

private String[] libriConsigliati; //Ecco qui l’array di libri

/\*\*

\* Costruttore della classe LibroConsigliato.

\*

\* @param utenteId L'ID dell'utente che consiglia i libri

\* @param libroPrincipale Il titolo del libro principale per cui sono consigliati altri libri

\* @param libriConsigliati Un array contenente i titoli dei libri consigliati

\*/

public LibroConsigliato(String utenteId, String libroPrincipale, String[] libriConsigliati) {

this.utenteId = utenteId;

this.libroPrincipale = libroPrincipale;

this.libriConsigliati = libriConsigliati;

}

L'array verrà utilizzato nella classe LibroConsigliato per limitare il numero di libri consigliati ad un massimo di tre. Questo perché gli array permettono un **controllo rigido del numero di elementi**.

**Algoritmi**

Gli algoritmi utilizzati in **Book Recommender** sono relativamente semplici, poiché il sistema non presenta logiche computazionali complesse. Tuttavia, ci sono alcune logiche algoritmiche importanti:

**1. Algoritmi di ricerca e filtraggio**

La funzione di ricerca di libri implementa una ricerca sequenziale, iterando su tutti i libri presenti nel CSV per trovare corrispondenze basate su titolo o autore.

Esempio di **ricerca per titolo**:

public List<Libro> cercaLibroPerTitolo(String parolaTitolo) {

List<String[]> contenutoFile = csvManager.leggiDaCSV(FILE\_CSV);

return contenutoFile.stream()

.filter(riga -> riga[0].toLowerCase().trim().contains(parolaTitolo.toLowerCase().trim()))

.map(riga -> creaLibroDaRiga(riga))

.toList();

}

Il cerca i libri il cui titolo contiene una determinata parola all'interno di un file CSV, utilizzando i seguenti passaggi e metodi:

1. **stream()**: converte la lista in uno stream per consentire operazioni funzionali.
2. **filter(riga -> riga[0].toLowerCase().trim().contains(parolaTitolo.toLowerCase().trim()))**: filtra le righe dove il titolo (presente nella prima colonna, riga[0]) contiene la parola parolaTitolo, ignorando maiuscole/minuscole e spazi.
3. **map(riga -> creaLibroDaRiga(riga))**: trasforma ogni riga filtrata in un oggetto Libro utilizzando il metodo creaLibroDaRiga, che crea un'istanza di Libro a partire dai dati della riga.
4. **toList()**: raccoglie i risultati filtrati e trasformati in una lista di oggetti Libro.

In altri casi vengono utilizzati algoritmi di filtraggio, in modo da filtrare i dati in un file csv per “validare” una determinata operazione o semplicemente per ritornare una lista di elementi filtrati. Un esempio è **getLibrerieByUser**:

public List<Libreria> getLibrerieByUser(String userId) {

List<Libreria> librerieUtente = new ArrayList<>();

List<String[]> contenutoFile = csvManager.leggiDaCSV(FILE\_CSV);

for (String[] riga : contenutoFile) {

if (riga[0].equals(userId)) {

String nomeLibreria = riga[1];

List<Libro> libri = parseLibri(riga);

librerieUtente.add(new Libreria(nomeLibreria, GestioneSessione.getInstance().getUtenteLoggato(), libri));

}

}

return librerieUtente;

}

In questo caso vengono trovate le librerie associate all’userId dell’utente, utile per mostrare le proprie librerie in UserLibraryPage.

**2. Algoritmo di lettura/scrittura in un file .csv**

La classe CSVReaderWriter contiene vari metodi che leggono e scrivono i dati da/su un file .csv fornito come argomento. Un esempio lampante è il metodo **leggiDaCSV,** che si avvale del package OpenCSV per un’implementazione semplice e diretta (codice più avanti).

**3. Algoritmi di validazione**

Nella classe Registrazione sono presenti alcuni algoritmi di validazione di email e codice fiscale che usano delle regular expression per fornire maggiore sicurezza, come ad esempio:

private boolean emailIsFormatted(String email) {

String regexPattern = "^[0-9a-zA-Z]+([.\_-]?[0-9a-zA-Z]+)\*@[0-9a-zA-Z]+([.-]?[0-9a-zA-Z]+)\*\\.[a-zA-Z]{2,7}$";

Pattern pattern = Pattern.compile(regexPattern);

Matcher matcher = pattern.matcher(email);

return matcher.matches();

}

**4. Calcolo della valutazione media**

Il sistema di valutazione permette agli utenti di valutare i libri su diversi parametri. Il voto finale è calcolato come la media aritmetica di più voti:

private int calcolaVotoFinale() {

double media = (votoStile + votoContenuto + votoGradevolezza + votoOriginalita + votoEdizione) / 5.0;

return (int) Math.round(media);

}

Questo algoritmo prende cinque voti in ingresso, calcola la media e arrotonda il risultato al numero intero più vicino.

**Formato dei file e gestione**

I dati dell'applicazione vengono salvati in file **CSV** (Comma Separated Values) nel percorso src/main/resources/. I file CSV vengono utilizzati per memorizzare informazioni sugli utenti registrati, i libri disponibili, le librerie, i libri consigliati e le valutazioni.

Esempio di struttura di un file CSV per gli utenti registrati:

Immagine che contiene testo, schermata, Carattere, bianco

Descrizione generata automaticamente

La gestione dei file CSV avviene attraverso la classe CSVReaderWriter, che sfrutta la libreria esterna **OpenCSV** per semplificare la lettura e scrittura dei dati.

Esempio di lettura di un file CSV:

public List<String[]> leggiDaCSV(String nomeFile) {

String percorsoFile = Paths.get("src/main/data/", nomeFile).toString();

List<String[]> contenutoFile = new ArrayList<>();

try (CSVReader reader = new CSVReaderBuilder(new FileReader(percorsoFile))

.withCSVParser(new RFC4180ParserBuilder().build())

.build()) {

String[] rigaSuccessiva;

while ((rigaSuccessiva = reader.readNext()) != null) {

contenutoFile.add(rigaSuccessiva);

}

} catch (IOException | CsvValidationException e) {

e.printStackTrace();

System.err.println("Errore nella lettura del file CSV: " + percorsoFile);

}

return contenutoFile;

}

In questo esempio, il file CSV viene letto riga per riga e il suo contenuto viene memorizzato in una lista di array di stringhe.

**Pattern utilizzati**

Il progetto **Book Recommender** utilizza alcuni **design pattern** standard per garantire una struttura solida e mantenibile.

**Singleton**

Le classi CSVReaderWriter, GestioneSessione e GestioneUtente utilizzano il pattern **Singleton** per assicurarsi che ci sia una sola istanza di questa classe in esecuzione, evitando conflitti di lettura e scrittura nei file CSV ed evitando sessioni multiple per lo stesso utente

Esempio di implementazione del pattern Singleton:

public static synchronized GestioneSessione getInstance() {

if (instance == null) {

instance = new GestioneSessione();

}

return instance;

}

Questo approccio garantisce che ogni chiamata a getInstance() restituisca la stessa istanza dell'oggetto, fornendo un controllo centralizzato sulle operazioni sui file CSV.

**Limiti della soluzione sviluppata**

**Archiviazione su file CSV**: Sebbene i file CSV siano una soluzione semplice per la gestione dei dati, non sono ideali per applicazioni su larga scala o con molte operazioni concorrenti.

1. **UI Swing**: Swing è un framework maturo ma datato per lo sviluppo di interfacce grafiche desktop. Migrare verso un framework più moderno come **JavaFX** migliorerebbe notevolmente l'esperienza utente e fornirebbe strumenti più potenti per la gestione delle interfacce.
2. **Mancanza di supporto multiutente simultaneo**: Il sistema non supporta sessioni concorrenti per più utenti. Questo potrebbe essere risolto con l'integrazione di sessioni web o l'utilizzo di un'applicazione client-server.
3. **Crittografia delle password**: In uno use-case reale bisognerebbe introdurre un sistema di crittografia delle password, per motivi di sicurezza e tutela dei dati dell’utente.

**Sitografia / Bibliografia**

1. **Oracle Java Documentation**: <https://docs.oracle.com/javase/>
2. **OpenCSV Documentation**: http://opencsv.sourceforge.net