Manuale Tecnico - Book Recommender

Autore:
- Nome: Alexandru
- Cognome: Raita
- Matricola: 757601
- Sede: VA

Data: 9 Settembre 2024
Versione Documento: 1.0

Indice

Report tecnico della soluzione sviluppata

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. Tra i modelli principali troviamo le classi **Libro**, **Libreria**, **Utente** e **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 JFrame, JButton, JTextField, ecc.

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.

Esempio di **controller**:

```
package controller;

import java.util.List;
import model.Utente;
import utility.CSVReaderWriter;
```

```java
/**
 * 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, array e occasionalmente collezioni più complesse per esigenze specifiche.

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;

  /**
   * 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 è utilizzato qui per limitare il numero di libri consigliati ad un massimo di tre. 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. Ricerca sequenziale

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();
  }
```

La ricerca avviene attraverso una semplice iterazione su tutte le righe del file CSV.

2. 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:

UtentiRegistrati

Nome	Cognome	Codice Fiscale	Email	Nome Utente	Password
Alex	Raita	RTALND00L20D912W	alexrraita@gmail.com	alex00	Test1
Alex	Raita	RTALND00L20D912W	alex.rar@gmail.com	alex01	Test1

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 diversi **design pattern** standard per garantire una struttura solida e mantenibile.

1. Singleton
Le classi CSVReaderWriter e GestioneSessione 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.

2. Observer
La classe RatingPage utilizza il pattern **Observer** per notificare un listener esterno quando una nuova valutazione è stata inviata, garantendo un decoupling tra il componente grafico e la logica di business.

Esempio di utilizzo di **Observer**:

```
public class RatingPage extends JFrame {

  private ValutazioneListener listener

  public RatingPage(Libro libro, ValutazioneListener listener) {
    this.listener = listener;
          //Configurazione GUI per la valutazione...
```

```
private void inviaValutazione() {
     Valutazione valutazione = new Valutazione(...);

    // Salva la valutazione tramite il gestore
    gestioneValutazione.salvaValutazione(valutazione);

    // Notifica il listener che la valutazione è stata inviata
    if (listener != null) {
      listener.valutazioneInviata();
    }
}
```

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. Un database relazionale come **MySQL** o **PostgreSQL** migliorerebbe la scalabilità e l'efficienza.

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