Book Recommender

Manuale Tecnico

Alessandro Grassi, Luca Giorgio Rotter, Aleksandar Kastratovic, Davide Bilora

Indice

1.Introduzione

"Book Recommender" è un software per la valutazione e raccomandazione di libri, in grado di permettere agli utenti registrati di inserire recensioni e a tutti gli utenti di consultare le valutazioni e ricevere consigli di lettura. Il sistema è dunque destinato alla sola categoria di utenti come lettori. Non è previsto l'utilizza dell'applicazione da parte di moderatori che possono modificare utenti registrati e libri.

2.Premesse

Essendo l'applicativo sviluppato nell'ambito del progetto di "Laboratorio B" come proposta data dal corso universitario, la sua struttura e le scelte prese per la sua progettazione non sono basate necessariamente verso il suo impiego a livello industriale, ma tengono anche conto di altri fattori in modo da non complicarne, più del necessario, la realizzazione: non sono stati usati infatti framework java esterni, come Spring ad esempio, per la realizzazione del server.
Si è comunque mantenuta una struttura il più completa e modificabile possibile in modo da renderla scalabile inserendo altre funzionalità attraverso aggiornamenti e con lo scopo di poter fare manutenzione, mantenendo quindi l'architettura tipica di un applicazione enterprise.

Inoltre, all'interno del manuale, verrà discusso il funzionamento e le caratteristiche dei componenti più importanti e complicati, senza entrare nel dettaglio di quelli con un comportamento più semplice (è possibile consultare la Javadoc per vederne le specifiche). Questo vale anche per i diagrammi UML prodotti in fase di progettazione (che sono disponibili nella cartella dedicata)

3.Software Design

Per lo sviluppo dell'applicazione si è deciso di utilizzare una versione recente di Java, nello specifico Java 21, per sfruttare appieno le moderne innovazioni del linguaggio, riducendo così la quantità di codice boilerplate e migliorando la manutenibilità del software.

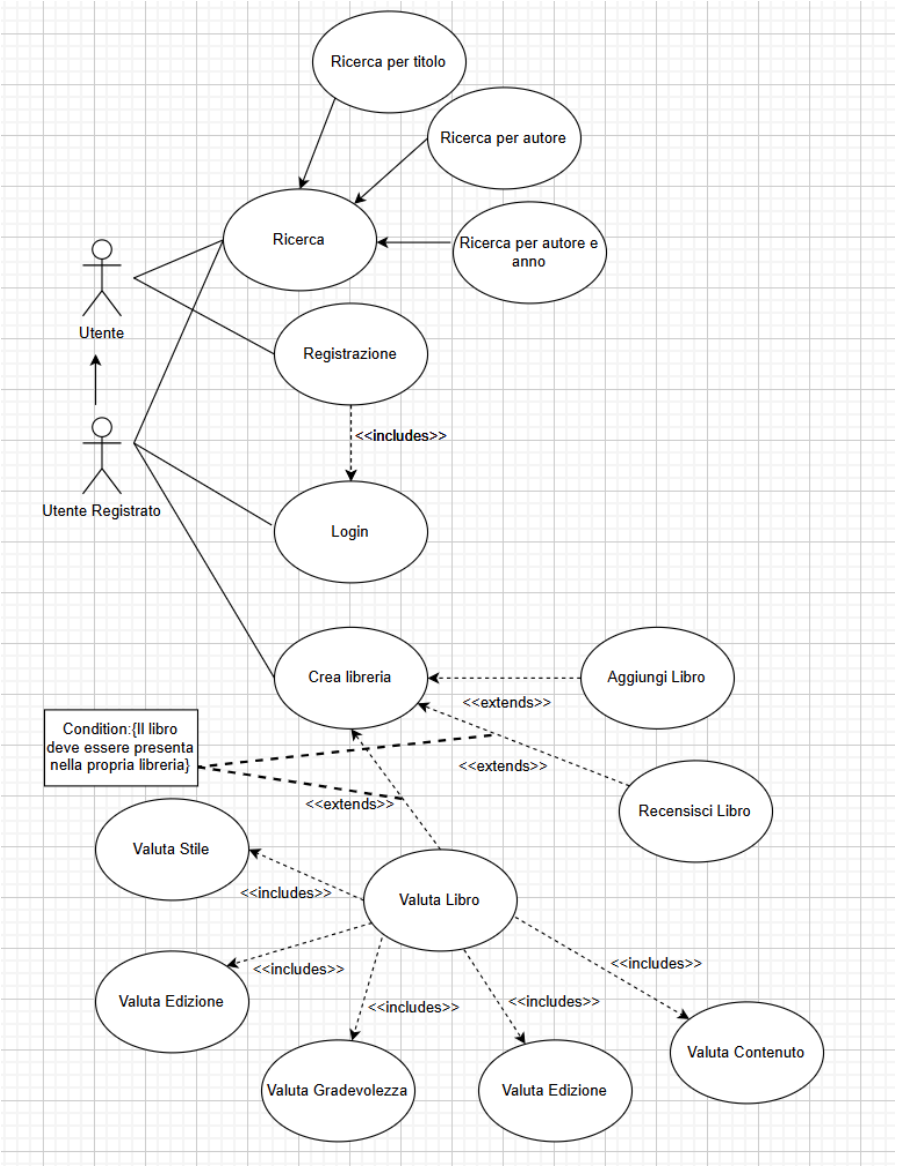
L'applicazione è stata dunque progettata e sviluppata in Java 21 e testata per sistemi operativi Windows (10 e 11). Per la persistenza dei dati, è stato utilizzato il DBMS PostgreSQL.

Invece, per la comunicazione tra client e server, è stata adottata la tecnologia RMI (Remote Method Invocation), al fine di semplificare la logica di connessione e rendere trasparente la gestione del multi-threading, necessario per gestire i diversi client, consentendo così di focalizzarsi sulla logica applicativa. L'interfaccia grafica del client è stata sviluppata con JavaFX, che facilita la separazione tra la logica applicativa e le View, migliorando la leggibilità e la manutenibilità del codice.

3.1 Struttura dell'applicazione

E' stato richiesto la realizzazione di un'applicazione client-server in Java.
Per soddisfare questa esigenza, il sistema è stato suddiviso in tre moduli distinti:

- backend : contenente il server;
- frontend: contenente il client;
- common: contenente le interfacce, le eccezioni e le entità condivise tra client e server



Come sistema di build è stato utilizzato Maven: ogni modulo possiede le sue dipendenze, specificate nel suo pom.xml, ed eventuali dipendenze condivise sono specificate nel file pom.xml padre(nel modulo di progetto LaboratorioB).

Si segnala, infine, che nel modulo server sono state implementate diverse funzionalità non esplicitamente richieste, ma sono state implementate per una corretta funzionalità del programma o per migliorie alle prestazioni/estetiche

3.2 Connessione al database

La connessione al database relazionale PostgreSQL avviene tramite **HikariCP**, un pool di connessioni JDBC basato su protocollo TCP, gestito centralmente dalla classe DatabaseManager. L'utilizzo di HikariCP consente una gestione efficiente e performante delle connessioni al database, riducendo i tempi di latenza, migliorando la scalabilità

dell'applicazione e ottimizzando l'uso delle risorse grazie al riutilizzo delle connessioni e alla loro gestione automatica.
L'utilizzo di HikariCP consente inoltre di risolvere i problemi legati alla scadenza di una connessione e aiuta a ridurre i possibili errori che si possono presentare, rendendo la connessione al database più affidabile.
In questa classe è presente un metodo statico in cui viene creata la pool di connessioni con il database:

```
static {
    try {
        // Configurazione di HikariCP per PostgreSQL
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl(jdbcUrl: "jdbc:postgresql://localhost:5432/LaboratorioB");
        System.out.println(x: "Inserisci username e password del database:");
        //String username = in.nextLine();
        config.setUsername(username: "postgres");
        //String pwd = in.nextLine();
        config.setPassword(password: "@Aleks13082002");
        config.setDriverClassName(driverClassName: "org.postgresql.Driver");

        // Parametri opzionali del pool
        /*config.setMaximumPoolSize(10);
        config.setMinimumIdle(2);
        config.setIdleTimeout(30000);
        config.setConnectionTimeout(30000);
        config.setLeakDetectionThreshold(60000);*/

        dataSource = new HikariDataSource(config);

        System.out.println(x: "Pool HikariCP inizializzato con successo!");

    } catch (Exception e) {
        System.err.println(x: "Errore durante l'inizializzazione del pool HikariCP:");
        e.printStackTrace();
    }
}
```

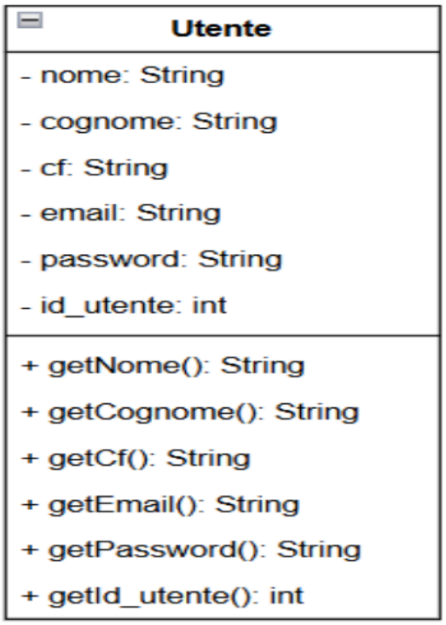
questo metodo è stato configurato in modo che le credenziali di accesso possono essere inserite al momento della richiesta di connessione.
Inoltre sono presenti i metodi getConnection() ogni volta che si deve eseguire una query e il metodo shutdown() per chiudere la connessione

3.3 Traduzione delle entità relazionali in entità Java

Prima di illustrare l'architettura del sistema, è utile chiarire il processo di traduzione delle entità relazionali in entità Java.
Le principali entità sono:

- Utenti(Utente)
- Libri(Libro)
- Libreria_Libri(Libreria)
- Libri_consigliati(Consiglio)
- Valutazione_Libri(Valutazione)

Lato applicativo le entità sono state implementate oggetti Java. La traduzione degli attributi SQL nei rispettivi campi Java è stata sostanzialmente 1:1.



Utenti		
PK	id_utente	INTEGER GENERATED ALWAYS AS IDENTITY
NOT NULL	nome	varchar(255)
NOT NULL	cognome	varchar(255)
NOT NULL	email	UNIQUE varchar(255)
NOT NULL	password	varchar(255)
NOT NULL	cf	UNIQUE varchar(255)

Di particolare interesse è l'uso di una classe enumerativa per gestire in maniera più efficiente la ricerca dei libri: la classe enumerativa Ricerca è così realizzata:

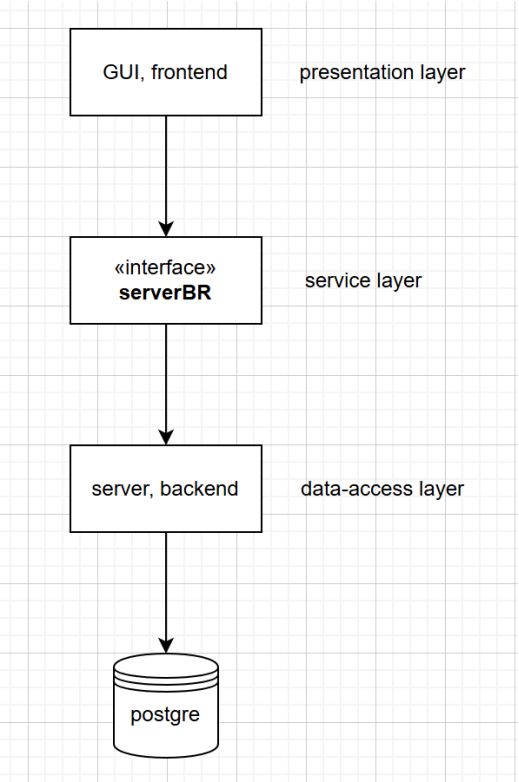
```
public enum Ricerca {
    TITOLO,
    AUTORE,
    ANNO
}
```

Tutte le entità descritte implementano l'interfaccia Serializable e sono collocate nel modulo common, poiché devono essere costantemente scambiate tra client e server tramite RMI.

3.4 Architettura

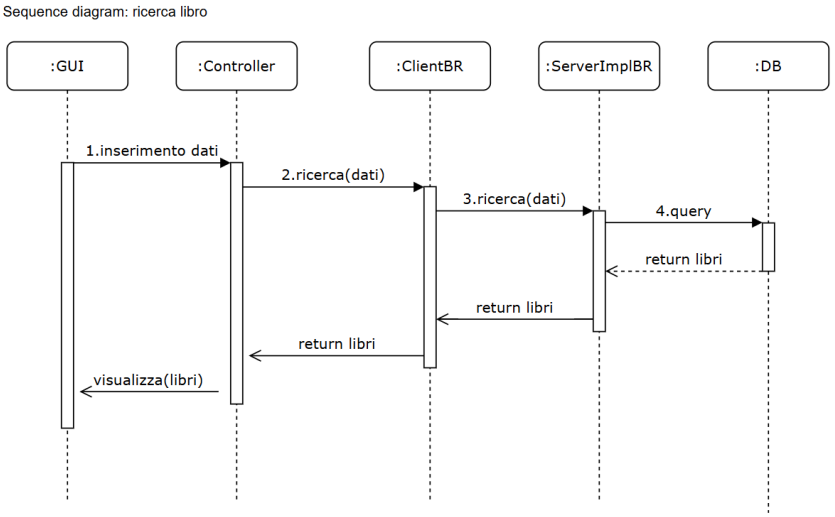
Nell'applicazione è stata adottata una classica architettura a tre livelli.
Dal basso verso l'alto, l'architettura è composta dai seguenti livelli:

- Persistence Layer (Database): il livello di persistenza dei dati, implementato utilizzando il database relazionale PostgreSQL.
- Data Access Layer: questo livello fornisce un'interfaccia di accesso ai dati, permette di isolare la logica di accesso ai dati dalle altre parti del sistema.
- Service Layer: qui viene implementata la logica di business dell'applicazione. Nella fase iniziale il service layer si occupa di instaurare un collegamento con il database e mettersi in ascolto sulla porta 1099 per accettare collegamenti con i client.
- Presentation Layer: corrisponde all'interfaccia grafica utente (Graphical User Interface, GUI) e alla logica di presentazione, che gestisce l'interazione tra l'utente e il sistema.



Ogni livello, fatta eccezione per quello di presentazione, mette a disposizione delle interfacce che vengono sfruttate dalle implementazioni del livello immediatamente superiore. Il principale vantaggio di questa architettura risiede nel fatto che ogni livello affida al livello sottostante le funzionalità di cui necessita. Questo metodo assicura una chiara separazione delle responsabilità, rendendo più semplice la manutenzione del sistema. In questo modo, ogni livello può dedicarsi unicamente al proprio ruolo, diminuendo la complessità complessiva e aumentando la modularità del codice.

Esempio di richiesta e scambio dei dati tra i vari livelli:



DATA ACCESS LAYER

Il data access layer è implementato fornisce l'accesso e la consultazione dei dati tramite il collegamento al database che avviene con HikariCP. Per effettuare le query è stato importato all'interno della classe serverBRImpl java.sql in modo da poter utilizzare l'oggetto

query e l'eccezione *SQLException* per gestire opportunamente gli errori generati durante la fase di richiesta dei dati.
La funzione principale che permette di usare gli oggetti di tipo *query* è:

```
try (Connection conn = DatabaseManager.getConnection(); PreparedStatement ps = conn.prepareStatement(query)) {

    // imposta i parametri della query
    ps.setInt(parameterIndex: 1, id);
```

In cui inizialmente attraverso un driver JDBC viene creato un oggetto di *Connection* per collegarsi al DB e il metodo *prepareStatement(query)* permette di inviare la query. Usando quest'ultimo metodo è possibile inserire i parametri in modo sicuro, con migliori prestazioni e un codice più leggibile. Per inserire i parametri nella query, indicati con "?", si utilizza il *setInt()* per numeri interi oppure *setString()* per stringhe.

```
try (ResultSet rs = ps.executeQuery()) {
    if (rs.next()) {
        String nome = rs.getString(columnLabel: "nome");
        String cognome = rs.getString(columnLabel: "cognome");
        String cf = rs.getString(columnLabel: "cf");
        String email = rs.getString(columnLabel: "email");
        String password = rs.getString(columnLabel: "password");
        user = new Utente(nome, cognome, cf, email, password, id);
    }

}
```

la query SQL precedentemente preparata al database e restituisce un oggetto *ResultSet*. Quest'ultimo rappresenta l'insieme dei risultati prodotti dall'interrogazione.
Il costrutto try-with-resources garantisce che il *ResultSet* venga chiuso automaticamente al termine dell'elaborazione, evitando possibili perdite di risorse e migliorando la gestione della memoria.
Tramite il metodo next(), il cursore del ResultSet viene posizionato sulla prima riga del risultato. Se la query ha prodotto almeno un record, i valori delle colonne, in questo esempio: nome, cognome, cf, email e password vengono estratti utilizzando il metodo getString(), che consente di recuperare i dati in base al nome della colonna.

SERVICE DATA LAYER
Le interfacce fornite dal Service Layer, sono esposte tramite RMI (estendono infatti java.rmi.Remote) e condivise tra client e server[13]. Di conseguenza, esse sono collocate nel modulo common.
Esse costituiscono, quindi, i servizi offerti dal server al client e i loro riferimenti concreti vengono pubblicati nel Registry di RMI all'avvio del server.
I metodi del Service Data layer seguono pressoché la stessa struttura:

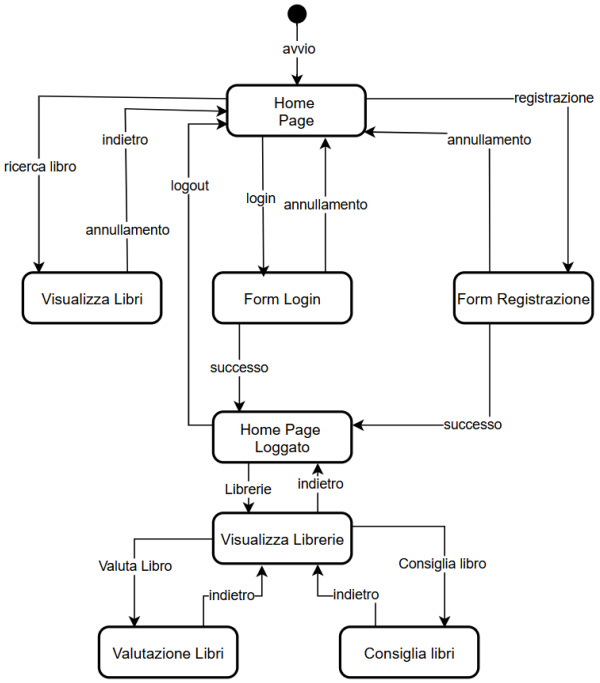
1. Validazione dei dati(se necessaria e se non viene fatta nel presentation layer)
2. Query
3. Esecuzione della query
4. Ricezione dei dati e salvataggio di essi in apposite variabili
5. Elaborazione dei dati(se necessaria)
6. Creazione di oggetti da restituire

PRESENTATION LAYER

Il Presentation Layer è responsabile della logica di visualizzazione e della gestione dell'interazione tra l'utente e l'applicazione. Per rispondere ai requisiti del sistema, è stata realizzata un'interfaccia grafica (GUI) tramite la libreria JavaFX, avvalendosi di Scene Builder per agevolare la progettazione delle View, definite in file FXML, e per favorire una netta separazione dal codice dei Controller, incaricati della gestione degli eventi e della logica di presentazione.

Al fine di garantire una distinzione chiara tra la logica di presentazione e il modello dei dati, è stato adottato un pattern basato sui ViewModel. Queste classi fungono da collegamento tra le entità del dominio (modelli POJO) e l'interfaccia utente, incapsulando le Property di JavaFX che, grazie all'implementazione dell'interfaccia Observable, consentono il binding e l'aggiornamento automatico della GUI.

L'utilizzo dei ViewModel offre come vantaggio principale il rispetto del Principio di Singola Responsabilità: le entità condivise tra client e server rimangono infatti prive di dipendenze dall'interfaccia grafica. Inoltre, i ViewModel gestiscono le proprietà osservabili richieste da alcuni componenti dell'interfaccia, permettendo un aggiornamento immediato degli elementi grafici in risposta alle azioni dell'utente, senza alterare la coerenza del modello dei dati.



Infine, è stato utilizzato un semplice file CSS per definire lo stile di vari elementi della GUI, con l'obiettivo di migliorare l'aspetto visivo dell'interfaccia e renderla più coerente e rappresentativa della realtà professionale dei committenti

PACKAGE BY LAYER

L'adozione dell'architettura a livelli ha come naturale conseguenza il sistema di packaging basato sui livelli (*package by layer*). In questo meccanismo, ogni package rappresenta un livello e contiene tutte le classi relative a quel determinato livello.

Un'alternativa più modulare e maggiormente indicativa del dominio applicativo è il sistema di packaging basato sulle funzionalità (*package by feature*): in questo scenario, i package non contengono più ciascuno tutte le classi di un determinato livello, ma al contrario, contengono tutte le classi di una determinata macro-funzione

3.5 Strutture dati utilizzate

In questa applicazione non sono state utilizzate strutture dati particolarmente complesse. Sono state impiegate:

- LinkedList: per liste dei libri consigliati e valutazioni di libri
- ArrayList<Libro>: per le librerie
- List<T>: per l'implementazione di LinkedList e ArrayList
- ResultSet: legata al risultato del database
- PreparedStatement: per gestire query SQL parametrizzate

3.6 Design Pattern adottati

Nel lato server dell'applicazione sono stati adottati diversi design pattern.
Service Locator: il Registry di RMI, che espone tutti i servizi offerti dal server, funge da Service Locator, centralizzando la ricerca e l'accesso ai servizi necessari. Le entità del dominio (Libro, Utente, Libreria, Valutazione) fungono da Data Transfer Object (DTO) per il trasferimento dei dati tra client e server.
L'accesso al database segue i principi del DAO pattern, seppur in forma parziale, tramite l'uso di JDBC e di una classe centralizzata per la gestione delle connessioni.
È inoltre presente un meccanismo di Lazy Loading per il caricamento incrementale dei libri, al fine di migliorare le prestazioni. Il server RMI opera come un Singleton concettuale, mentre le invocazioni remote possono essere ricondotte al Command pattern.

4.Algoritmi impiegati

La logica del codice, al momento dell'implementazione, risulta molto semplice e gli algoritmi non sono particolarmente complessi: l'unica funzione che ha senso studiare più dettagliatamente è "lazyLoadingLibri", che permette di caricare grandi quantità di dati poco alla volta in base alla richiesta, per non sovraccaricare il sistema.

```
final int LIMIT = 20;
List<Libro> libri = new ArrayList<>();

// Identifica il client RMI
String clientHost;
try {
    clientHost = java.rmi.server.RemoteServer.getClientHost();
} catch (java.rmi.server.ServerNotActiveException e) {
    clientHost = "unknown";
}

String query = """
SELECT l.id_libro, l.titolo, l.autore, l.genere, l.editore, l.anno
FROM libri l
LEFT JOIN libri_inviati li
    ON l.id_libro = li.id_libro AND li.client_host = ?
WHERE li.id_libro IS NULL
""";
if (genere_ricerca != null && !genere_ricerca.isBlank()) {
    query += " AND l.genere ILIKE ? ";
}
query += """
ORDER BY l.id_libro
LIMIT ?
""";
```

Il codice riportato è la porzione cruciale che si occupa del funzionamento corretto della funzione *lazyLoadingLibri*, infatti legge e memorizza solo un certo numero di dati dal database (LIMIT), occupandosi anche di non inviare duplicati: per fare ciò ci si appoggia ad

una tabella nel database (libri_inviati) in cui vengono registrati gli oggetti già inviati, facendo poi un JOIN tra questa e la tabella libri.
La tabella si occupa anche di memorizzare i dati per ogni client (getClientHost()), in modo da tenere separati i dati di ogni utente diverso che si collega e usufruisce della funzione.
lazyLoadingLibri può essere utilizzata sia per caricare dei libri generali, sia per caricarli di un genere scelto dall'utente, come si può vedere dalla foto sopra inserita.

Due soluzioni che sono state adottate spesso e che si notano in questa porzione di codice sono:

- l'utilizzo di ILIKE nelle query per consentire di trovare risultati anche inserendo solo delle sottostringhe
- la costruzione di una query attraverso la concatenazione di più parti in modo da poterla adattare alla richiesta

5.Gestione della concorrenza

Nell'applicazione "Book Recommender" non è stato necessario implementare una gestione diretta della concorrenza. Infatti, dal momento che HikariCP (che si basa su JDBC) offre di default l'auto-commit delle transazioni, l'accesso al database come conseguenza delle richieste dei client è gestito in modo autonomo dal sottosistema di gestione della concorrenza del DBMS. Inoltre, al momento della stesura di questo manuale, non esistono risorse condivise a livello applicativo tra i vari client.

6.Strumenti, librerie e linguaggi utilizzati

Per lo sviluppo dell'applicazione *Book Recommender* si sono usati i seguenti strumenti:

- JDK 21;
- IDE: VS code e Eclipse;
- Glucon Scene Builder per la realizzazione di GUI
- Git e Github per versionamento e gestione dei branch dei membri del gruppo
- PostgreSQL 18 come DBMS
- PgAdmin 4 per interazione con la base di dati attraverso GUI
- draw.io per la realizzazione dei diagrammi UML e ER
- Google Documenti per la realizzazione dei manuali e in generale del materiale testuale
- HikariCP 5.1.0 per la gestione della connessione con il database

Sono stati usati anche i seguenti linguaggi:

- Java per la realizzazione di client e server
- SQL per la realizzazione e l'interrogazione del database
- (F)XML per la costruzione dell'interfaccia
- CSS per lo styling dei componenti della GUI
- Python per la realizzazione di un algoritmo per popolare il database partendo dal file csv

Inoltre si sono utilizzati:

- PostgreSQL JDBC driver 42.7.7
- JavaFX (controls, graphics, base e fxml) versione 22.0.1
- Plugin Maven
    - exec-maven-plugin 3.1.0
    - maven-clean-plugin 3.4.0
    - maven-resource-plugin 3.3.1
    - maven-compiler-plugin 3.13.0
    - maven-surefire-plugin 3.3.0
    - maven-jar-plugin 3.4.2
    - maven-install-plugin 3.1.2
    - javafx-mave-plugin 0.0.8

7.Limiti dell'applicazione e conclusioni

Nonostante la ricerca di completezza dell'applicazione ci sono dei limiti dati dalla natura didattica del progetto:

- supporta solo una lingua, l'inglese, e non altre, rendendo l'utilizzo difficile per chi non lo parla
- la struttura delle query non sempre risulta ottimizzata non avendone necessità, ma può essere modificata semplicemente in caso di incremento di dati memorizzati
- potrebbe essere utile indicizzare alcune tabelle nel database in caso di crescita della quantità di informazione
- la scalabilità risulta limitata, sarebbe più efficiente utilizzare una struttura a microservizi per supportare una scalabilità migliore
- ulteriori controlli sui dati e l'aggiunta di token di sessione renderebbero l'applicativo più consistente e sicuro, nonostante i controlli necessari siano già presenti

Nel complesso gli obiettivi del progetto sono stati raggiunti cercando di mantenere la massima efficienza possibile e rispettando i requisiti richiesti.

Contatti

Alessandro Grassi, Project Manager: agrassi6@studenti.uninsubria.it