# Documentazione di Progetto

Manuale Tecnico

**Book Recommender**

Lorenzo Monachino - 757393 - VA
Lyan Curcio - 757579 - VA
Sergio Saldarriaga - 757394 - VA
Nash Guizzardi - 756941 - VA

Anno Accademico 2024/2025

# Indice

# Capitolo 1

# Progettazione della Soluzione

Questo capitolo illustra le scelte progettuali effettuate per la realizzazione del sistema *Book Recommender*. L'analisi copre l'architettura software distribuita, definita tramite UML, e la progettazione della base di dati relazionale, con particolare attenzione ai vincoli di integrità e alla sicurezza.

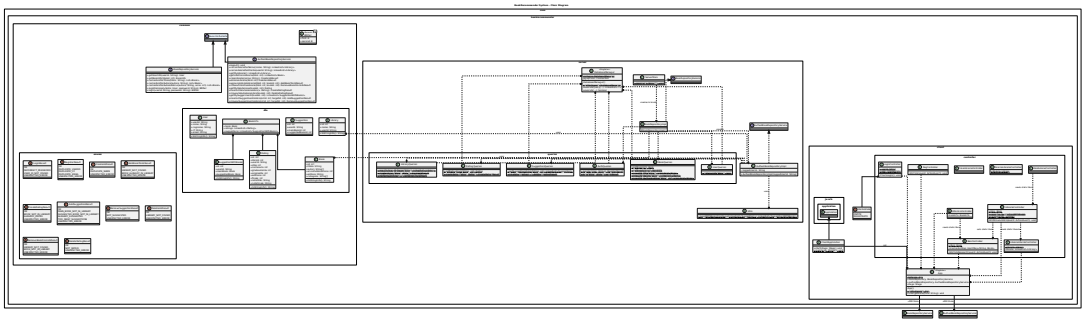
## 1.1 Progettazione Software (UML)

L'architettura del sistema segue il pattern Client-Server e sfrutta la tecnologia Java RMI per la comunicazione remota. Il progetto è organizzato in quattro moduli logici (Maven Modules), ognuno con responsabilità specifiche.

### 1.1.1 Struttura Statica (Class Diagram)

Il diagramma delle classi (Figura 1.1) mostra la suddivisione dei componenti e le dipendenze tra i pacchetti.

Figura 1.1: Diagramma delle Classi: Architettura Completa del Sistema



#### Modulo Common

È il pacchetto condiviso che definisce le regole di comunicazione.

- **Interfacce RMI:**
  - `BookRepositoryService`: Interfaccia pubblica per utenti non autenticati (ricerca, login, registrazione).
  - `AuthedBookRepositoryService`: Interfaccia protetta per utenti autenticati, ottenibile solo post-login.
- **DTO (Data Transfer Objects):** Classi immutabili (`Libri`, `UtentiRegistrati`, `Valutazione`) usate per il trasporto dati. Implementano `Serializable` per la trasmissione RMI.
- **Enums:** Definiscono una serie di "eventi" per gestire gli esiti senza ricorrere alle eccezioni per la logica di controllo.

#### Modulo Server (Back-end)

Gestisce la logica applicativa e la persistenza.

- **Gestione Sessione (tramite Factory Pattern):** `BookRepositoryImpl` (in caso di autenticazione) agisce da Factory. I metodi `register` e `login` restituiscono un'istanza di `AuthedBookRepositoryImpl` specifica per l'utente, incapsulando l'`userId` nella sessione.
- **Database Manager:** Implementa il pattern Singleton per centralizzare la gestione della connessione JDBC.
- **Queries:** Classi statiche (`AuthQueries`, `LibraryQueries`, ecc.) che isolano le query SQL dal codice RMI.

**Modulo Client (Front-end)**

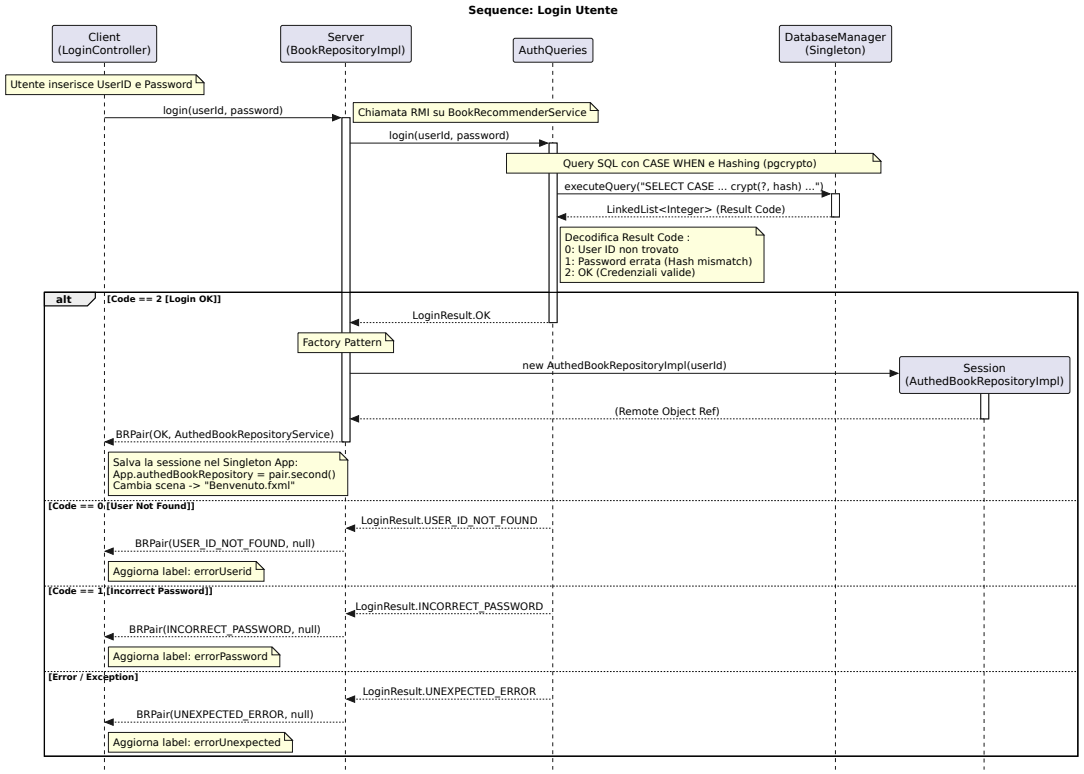
Applicazione JavaFX basata sul pattern MVC.

- **Navigazione:** La classe `App` (Singleton) gestisce lo `Stage` primario e il cambio delle scene (`changeScene`).
- **Controller:** Classi come `LoginController` e `BenController` gestiscono l'interazione UI e invocano i metodi remoti.

### 1.1.2 Struttura Dinamica (Sequence Diagrams)

Viene analizzato lo scenario critico di autenticazione per evidenziare la gestione della sicurezza.

Figura 1.2: Sequence Diagram: Login e creazione Sessione




Come mostrato in Figura 1.2, il Client invoca `login()`. Il Server delega la verifica hash a `AuthQueries`. Se valida, viene istanziato un oggetto remoto di sessione che viene restituito al Client all'interno di un oggetto `BRPair`.

**Sequence Diagram: Inserimento Suggerimento**

Il seguente diagramma (Figura 1.3) mostra il flusso per l'inserimento di un consiglio tra libri.

## 1.2 Progettazione Database

Il livello dati è gestito da PostgreSQL. La progettazione include vincoli per garantire la coerenza dei dati.

## 1.3 Analisi dei Requisiti

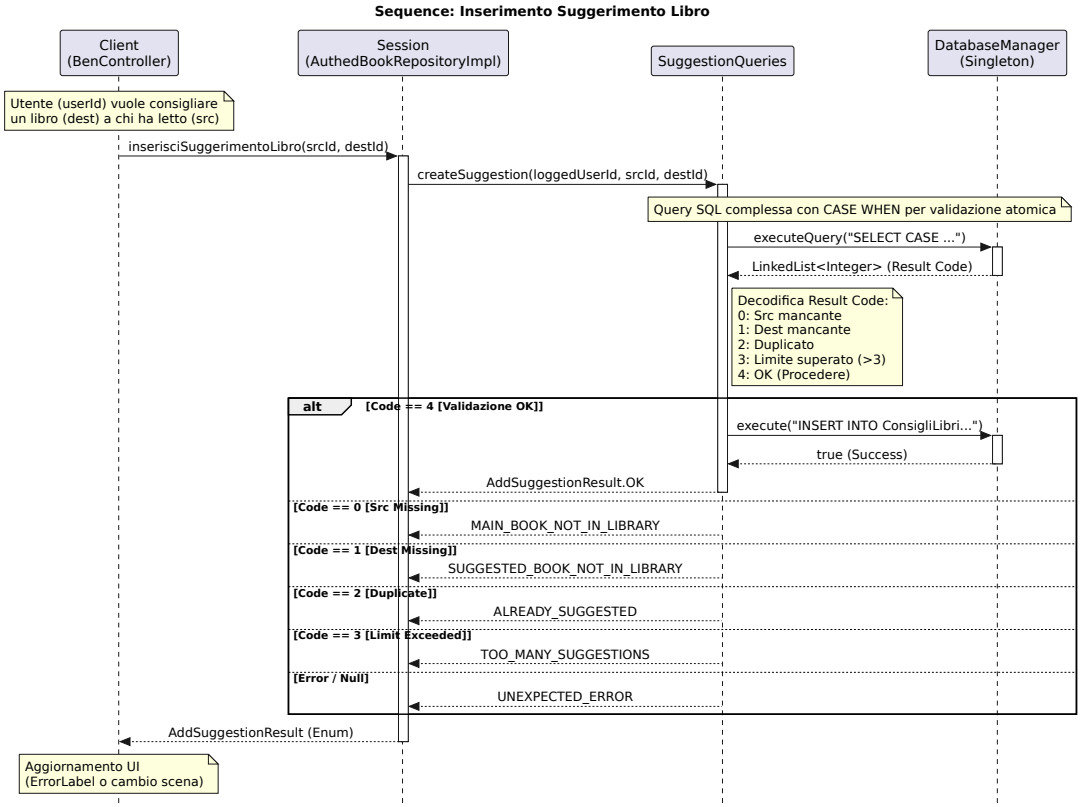
In conformità con le specifiche di progetto, il sistema è stato progettato per soddisfare requisiti funzionali distinti per tipologia di utenza e requisiti non funzionali vincolanti per l'architettura.

### 1.3.1 Requisiti Funzionali

Il sistema prevede due livelli di accesso con funzionalità incrementali:

- **Utente Non Registrato:**
  - **Registrazione:** Inserimento dati anagrafici (Nome, Cognome, CF, Email, UserID, Password) per creare un nuovo account.

Figura 1.3: Sequence Diagram: Logica di validazione inserimento suggerimento




  - **Login:** Autenticazione tramite UserID e Password.
  - **Consultazione:** Ricerca di libri per titolo, autore o autore e anno di pubblicazione. Visualizzazione dei dettagli del libro.

- **Utente Registrato:** Possiede tutte le funzionalità dell'ospite, con l'aggiunta di:
  - **Gestione Librerie:** Creazione di librerie personali personalizzate; aggiunta e rimozione di libri dalle proprie librerie.
  - **Valutazioni:** Inserimento di recensioni composte da punteggi (Stile, Contenuto, Gradevolezza, Originalità, Edizione) e note testuali per i libri posseduti.
  - **Suggerimenti:** Possibilità di consigliare un libro correlato a un altro, a patto di possederli entrambi nelle proprie librerie.

### 1.3.2 Requisiti Non Funzionali e Vincoli di Sistema

- **Architettura:** Client-Server distribuita basata su Java RMI.
- **Persistenza:** Utilizzo di database relazionale PostgreSQL.
- **Concorrenza:** Gestione di accessi multipli e contemporanei al server.
- **Interfaccia:** GUI realizzata con JavaFX.

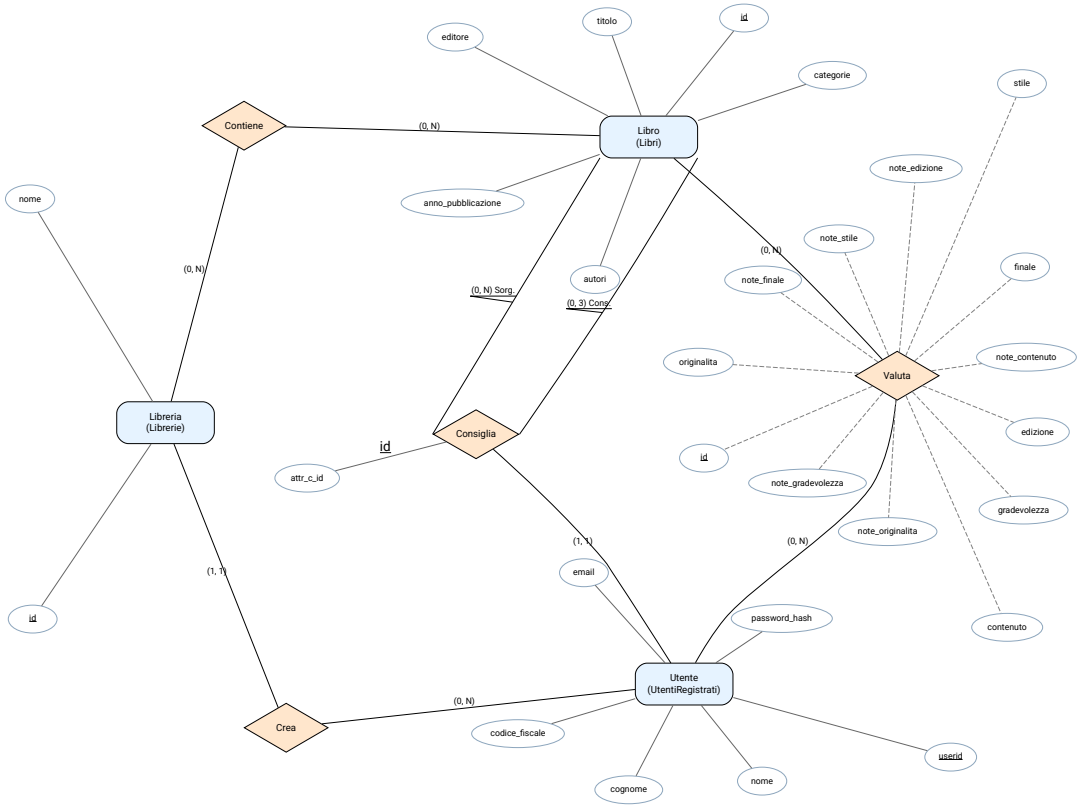
### 1.3.3 Vincoli di Integrità dei Dati

Oltre ai requisiti funzionali, sono stati implementati vincoli rigidi a livello di database per garantire la consistenza semantica dei dati:

1. **Unicità Utente:** Non è possibile registrare più utenti con lo stesso `UserID`, `Email` o `Codice Fiscale`.
2. **Unicità Librerie:** Un utente non può creare due librerie con lo stesso nome.
3. **Regola di Valutazione:** Un utente può inserire una valutazione per un libro *solo se* tale libro è presente in almeno una delle sue librerie personali (gestito tramite Trigger).
4. **Singola Valutazione:** Un utente non può valutare lo stesso libro più di una volta.
5. **Regola dei Consigli:** Per suggerire un libro $B$ a chi ha letto $A$, l'utente deve possedere *entrambi* i libri nelle proprie librerie (gestito tramite Trigger).

### 1.3.4 Schema Entity-Relationship (ER)

Figura 1.4: Schema Entity-Relationship




Le entità principali (Figura 1.4) sono:

- **Utente:** Contiene le credenziali e l'hash della password.
- **Libro:** Catalogo statico importato da dataset CSV.
- **Libreria:** Gestiscono le librerie utente.

### 1.3.5 Transizione dal Modello ER al Modello Relazionale

Nel passaggio dal modello concettuale (Diagramma ER) allo schema logico relazionale (SQL), le entità sono state tradotte in tabelle e le associazioni complesse hanno subito un processo di "reificazione", trasformandosi a loro volta in tabelle fisiche.

- **Associazione "Contiene" (Molti-a-Molti):** La relazione tra *Librerie* e *Libri* presenta cardinalità N:M. Nel modello relazionale, questo vincolo strutturale impone la creazione di una tabella ponte ("bridge table") denominata **LibriXLibrerie**. Essa contiene le chiavi esterne verso le due entità collegate; la chiave primaria della tabella è costituita dalla chiave composta (*libreria_id*, *libro_id*).
- **Associazione "Valuta" con attributi:** Sebbene concettualmente definita come relazione tra *Utente* e *Libro*, questa associazione possiede numerosi attributi propri (punteggi parziali e note testuali). Pertanto, è stata tradotta nella tabella **ValutazioniLibri**.
- **Associazione "Consiglia" (Ternaria):** Questa relazione coinvolge tre entità logiche: l'Utente (autore del consiglio), il Libro Sorgente e il Libro Consigliato. Nello schema logico si traduce nella tabella **ConsigliLibri**, dove tre chiavi esterne garantiscono l'integrità referenziale.

### 1.3.6 Analisi delle Cardinalità e Vincoli

Le cardinalità definiscono i vincoli di partecipazione e molteplicità delle relazioni:

- **Utente - Libreria (1:N):** Un singolo utente può creare molteplici librerie ($N$), ma ogni libreria appartiene univocamente a un solo utente (1). Esiste un vincolo di integrità referenziale forte: la cancellazione dell'utente comporta la cancellazione a cascata (*ON DELETE CASCADE*) di tutte le sue librerie.

- **Libreria - Libro (M:N):** Una libreria può contenere un numero arbitrario di libri e uno stesso libro può essere catalogato nelle librerie di diversi utenti.
- **Utente - Valutazione (1:N):** Un utente può redigere molte recensioni. Tuttavia, vige un vincolo di unicità sulla coppia (*userid*, *libro_id*), che impedisce l'inserimento di valutazioni multiple per il medesimo libro da parte dello stesso utente.

### 1.3.7 Schema Logico Relazionale

Di seguito viene riportato lo schema logico delle tabelle implementate nel database PostgreSQL. Sono evidenziate le chiavi primarie (sottolineate) e le chiavi esterne (FK).

- **UtentiRegistrati**(userid: VARCHAR(50), nome: VARCHAR(50), cognome: VARCHAR(50), codice_fiscale: CHAR(16) UNIQUE, email: VARCHAR(100) UNIQUE, password_hash: VARCHAR(256))
- **Libri**(id: SERIAL, titolo: VARCHAR(500), autori: VARCHAR(500), anno_pubblicazione: INT, editore: VARCHAR(200), categorie: VARCHAR(200))
- **Librerie**(id: SERIAL, nome: VARCHAR(100), *userid*: VARCHAR(50) FK)
  *Vincolo Unique: (nome, userid)*
- **LibriXLibrerie**(*libreria_id*: INT FK, *libro_id*: INT FK)
- **ValutazioniLibri**(id: SERIAL, *userid*: VARCHAR(50) FK, *libro_id*: INT FK, stile: INT, contenuto: INT, gradevolezza: INT, originalita: INT, edizione: INT, finale: INT[1], note_stile: VARCHAR(256), note_contenuto: VARCHAR(256), note_gradevolezza: VARCHAR(256), note_originalita: VARCHAR(256), note_edizione: VARCHAR(256), note_finale: VARCHAR(256))
  *Vincolo Unique: (userid, libro_id)*
- **ConsigliLibri**(id: SERIAL, *userid*: VARCHAR(50) FK, *libro_sorgente_id*: INT FK, *libro_consigliato_id*: INT FK)
  *Vincolo Unique: (userid, libro_sorgente_id, libro_consigliato_id)*

[1] Attributo calcolato (Generated Column).

# Capitolo 2

# Report Tecnico della Soluzione Sviluppata

In questa sezione vengono dettagliate le scelte implementative, i design pattern adottati per risolvere specifici problemi architetturali e le ottimizzazioni algoritmiche utilizzate.

## 2.1 Requisiti e Ambiente di Sviluppo

Le seguenti versioni software sono state utilizzate per lo sviluppo e il build del progetto, come definito nei file `pom.xml` dei vari moduli Maven:

| **Componente** | **Versione** |
|---|---|
| Java JDK | 25 |
| PostgreSQL Driver | 42.7.8 |
| JavaFX | 25.0.1 |
| Maven Compiler | 3.14.1 |

## 2.2 Scelte Architetturali

Il sistema *Book Recommender* adotta un'architettura distribuita basata sul middleware **Java RMI** (Remote Method Invocation), che permette di invocare metodi su oggetti residenti in una Java Virtual Machine (JVM) diversa da quella del chiamante.

Una scelta architetturale critica è stata la separazione delle interfacce nel modulo `Common`: questo permette al Client di conoscere i metodi invocabili (il "contratto") senza avere accesso al codice sorgente dell'implementazione server.

### 2.2.1 Strategia di Gestione degli Errori e Protocollo Applicativo

Per la comunicazione degli esiti delle operazioni tra Server e Client, è stato deciso di non utilizzare le Eccezioni Java (se non per errori critici di rete RMI), preferendo un approccio basato su **Enum**.

Ogni operazione (es. Login, Creazione Libreria) restituisce un Enum specifico (es. `LoginResult`) che codifica tutti i possibili stati logici (Successo, Password Errata, Utente non Trovato).

## 2.3 Gestione del Build e Gestione delle Dipendenze

Il progetto adotta una struttura **Maven Multi-Module** per garantire la separazione delle responsabilità. La configurazione è definita in un `pom.xml` padre che orchestra quattro moduli figli:

- **common:** Contiene le interfacce RMI, i DTO e gli Enum. È una dipendenza sia del Client che del Server.
- **serverBR:** Dipende da `common` e dai driver PostgreSQL per la comunicazione con il DB.
- **clientBR:** Dipende da `common` e dalle librerie JavaFX. Utilizza i plugin specifici di JavaFX per la gestione dei moduli grafici.
- **db_init:** Modulo di utilità per il setup iniziale, separato dal server per separare la logica di popolamento DB dal server. Dipende da `common` e da `serverBR` per la comunicazione con il DB.

Questa struttura impedisce, ad esempio, che il Client possa accidentalmente accedere alle classi di connessione al Database, poiché non ha il modulo `serverBR` tra le dipendenze.

### 2.3.1 Documentazione Javadoc

In conformità con le specifiche di progetto, l'intero codice sorgente è stato commentato seguendo lo standard Javadoc. La documentazione HTML è stata generata automaticamente tramite il plugin Maven `maven-javadoc-plugin` (configurato nel `pom.xml` padre).

#### Modularizzazione e Visibilità Javadoc

Una scelta tecnica rilevante è stata l'adozione del *Java Platform Module System* (JPMS), introdotto da Java 9 e pienamente supportato dalla versione 25 utilizzata. Ogni modulo Maven (`common`, `server`, `client`) contiene un descrittore `module-info.java`.

Questa architettura influenza direttamente la generazione della documentazione Javadoc e la sicurezza del codice:

- **Incapsulamento Forte:** Solo i package esplicitamente marcati con la direttiva `exports` nel file `module-info.java` sono visibili agli altri moduli e inclusi nella documentazione Javadoc pubblica.
- **Esempio Pratico:** Nel modulo `common`, il package `com.bookrecommender.common.dto` è esportato affinché Client e Server possano scambiarsi i dati, ed è quindi documentato. Le classi interne di utilità non esportate rimangono nascoste, mantenendo pulita l'API pubblica documentata.

```
module com.bookrecommender.common {
    requires java.rmi;
    requires java.sql;
    // Espone i DTO e le interfacce RMI alla documentazione e agli altri moduli
    exports com.bookrecommender.common.dto;
    exports com.bookrecommender.common;
    exports com.bookrecommender.common.enums.auth;
    // ... altri exports
}
```

Listing 2.1: Esempio di module-info.java nel modulo Common

La documentazione completa è consultabile dal file "`target/reports/apidocs/index.html`".

## 2.4 Script di Avvio e Automazione

Per facilitare il deployment e l'esecuzione dei vari moduli del sistema distribuito, sono stati predisposti degli script di avvio automatizzati nella root del progetto. Al fine di garantire la portabilità su diversi sistemi operativi, per ogni componente è stata fornita una doppia versione dello script:

- **.bat (Windows Batch File):** Per ambienti Microsoft Windows.
- **.sh (Shell Script):** Per ambienti Unix-like (Linux, macOS).

L'utilizzo di questi script astrae la complessità dei comandi di avvio (come la definizione del *classpath* o la gestione del *module-path* di JavaFX), permettendo l'esecuzione dei moduli con un singolo click o comando da terminale.

Gli script devono essere eseguiti nel seguente ordine logico per garantire il corretto funzionamento del sistema distribuito:

1. `InitDB.bat / .sh`: Esegue il modulo `db_init`. Questo script si occupa di compilare il progetto, connettersi al database PostgreSQL (utilizzando le credenziali configurate), creare lo schema delle tabelle e popolare i dati iniziali dal dataset CSV. Deve essere eseguito *una tantum* o quando si desidera resettare lo stato del database.
2. `Server.bat / .sh`: Esegue il modulo `serverBR`. Avvia il registro RMI sulla porta 1099 e pubblica il servizio `BookRecommenderService`, mettendosi in ascolto per le connessioni in ingresso. Questo processo deve rimanere attivo affinché l'applicazione sia utilizzabile.
3. `Client.bat / .sh`: Esegue il modulo `clientBR`. Lancia l'interfaccia grafica JavaFX. Possono essere avviate molteplici istanze di questo script contemporaneamente per simulare l'accesso concorrente di più utenti al sistema.

## 2.5 Design Pattern Utilizzati

Per garantire un codice modulare, manutenibile e robusto, sono stati applicati diversi design pattern.

### 2.5.1 Pattern Singleton

Il pattern Singleton è stato utilizzato nella classe `DatabaseManager` (lato Server) e nella classe `App` (lato Client).

- **Scopo:** Garantire che esista una e una sola istanza della connessione al database (Server) o del gestore delle scene (Client) per tutto il ciclo di vita dell'applicazione.
- **Implementazione:** Il costruttore è reso privato e l'accesso all'istanza avviene tramite un metodo statico sincronizzato.

```
public class DatabaseManager {
    private static DatabaseManager instance;
    private Connection pgsqlConn;

    private DatabaseManager() {
        // Inizializzazione driver e connessione JDBC
    }

    public static synchronized DatabaseManager getInstance() {
        if (instance == null) {
            instance = new DatabaseManager();
        }
        return instance;
    }
}
```

Listing 2.2: Implementazione Singleton in DatabaseManager.java

### 2.5.2 Pattern Data Transfer Object (DTO)

I DTO sono stati impiegati per il trasferimento dei dati tra Client e Server. Le classi `Libri`, `UtentiRegistrati` e `Valutazione` (nel package `common.dto`) ne sono l'esempio.

- **Caratteristiche:** Sono classi "povere" di logica, serializzabili (`implements Serializable`) e immutabili (campi `final`), progettate esclusivamente per trasportare dati attraverso la rete.
- **Vantaggio:** Riducono il numero di chiamate remote permettendo di inviare interi grafi di oggetti in un'unica invocazione.

### 2.5.3 Pattern Factory e Session Object

Il sistema implementa una gestione delle sessioni basata su oggetti remoti dinamici.

- **Funzionamento:** La classe `BookRepositoryImpl` (nel caso di registrazione o login) agisce come una *Factory*. I metodi `register` e `login`, in caso di successo, istanziano e restituiscono un nuovo oggetto `AuthedBookRepositoryImpl`.
- **Sessione:** L'oggetto restituito incapsula lo stato della sessione (l'ID dell'utente), garantendo che tutte le chiamate successive siano automaticamente autenticate.

```
@Override
public BRPair<LoginResult, AuthedBookRepositoryService> login(String userid,
    String pass) throws RemoteException {
    LoginResult result = AuthQueries.login(userid, pass);
    return new BRPair<>(
    result,
    // Se login OK, crea il nuovo oggetto remoto di sessione
    result == LoginResult.OK
    ? new AuthedBookRepositoryImpl(userid)
    : null
    );
}
```

Listing 2.3: Creazione dinamica della Sessione (BookRepositoryImpl.java)

### 2.5.4 Pattern MVC (Model-View-Controller)

Lato Client, l'applicazione JavaFX segue il pattern MVC:

- **View:** Definita nei file `.fxml` (es. `Login.fxml`).
- **Controller:** Gestisce gli eventi (es. `LoginController.java`).
- **Model:** I dati sono rappresentati dai DTO ricevuti dal server (es. `LoginResult` e `AuthedBookRepositoryService`).

## 2.6 Strutture Dati e Algoritmi

### 2.6.1 Strutture Dati

- **LinkedList:** Utilizzata nella classe `DatabaseManager` per accumulare i risultati delle query. È stata preferita all'`ArrayList` per la sua efficienza nelle operazioni di inserimento in coda durante la lettura sequenziale del `ResultSet`.
- **Java Record (BRPair):** È stata introdotta la classe generica `BRPair<A, B>` (implementata come `record` Java) per gestire il ritorno di valori multipli dai metodi RMI (es. Esito Operazione + Oggetto Remoto).

### 2.6.2 Algoritmi e Ottimizzazioni

#### Hashing Sicuro delle Password

La sicurezza non è gestita tramite algoritmi Java lato applicativo, ma delegata al DBMS tramite l'estensione `pgcrypto`. L'algoritmo utilizzato è **Blowfish** (tramite `bf` salt), che è molto resistente agli attacchi di forza bruta.

```
1         INSERT INTO "UtentiRegistrati" (...)
2         VALUES (..., crypt(?, gen_salt('bf')))
```

Listing 2.4: Hashing in AuthQueries.java

#### Bulk Loading del Dataset (Importazione Massiva)

Per il popolamento iniziale del catalogo libri (file CSV), non viene eseguito un loop di `INSERT` (che sarebbe lento a causa dell'overhead di rete per ogni riga). È stato implementato l'algoritmo di **Copy Manager** di PostgreSQL Driver, che apre uno stream diretto verso il server DB.

```
1         CopyManager copyManager = ((PGConnection) conn).getCopyAPI();
2         String sql = "COPY \"Libri\" ... FROM STDIN WITH (FORMAT csv, HEADER true)";
3         // Trasferimento diretto dallo stream del file al DB
4         copyManager.copyIn(sql, inputStream);
```

Listing 2.5: Ottimizzazione importazione CSV (Main.java)

Questa soluzione riduce il tempo di importazione da diversi minuti a pochi secondi per grandi dataset.

#### Ottimizzazione delle Comunicazioni di Rete

Un aspetto critico in un'architettura distribuita RMI è la latenza di rete. Un approccio tradizionale per l'inserimento di dati complessi (es. suggerimenti tra libri) richiederebbe più passaggi:

1. Interrogare il DB per verificare che i dati esistano.
2. Verificare che non ci siano duplicati.
3. Eseguire l'inserimento effettivo.

Questo comporterebbe 3 chiamate separate tra Client e Server.

Per risolvere questo problema, nelle classi `LibraryQueries` e `SuggestionQueries` è stata adottata una strategia di **Query Condizionale** usando il costrutto SQL `CASE WHEN`. In questo modo, tutta la logica di controllo viene eseguita direttamente dal DBMS in un'unica operazione.

- **Vantaggio:** L'operazione diventa atomica e riduce le interazioni di rete da 3 a 1.
- **Risultato:** Il tempo di esecuzione diminuisce sensibilmente, migliorando l'esperienza utente.

### 2.6.3 Analisi delle Prestazioni e Ottimizzazioni

In questa sezione si analizzano le prestazioni delle operazioni critiche del sistema, evidenziando come le scelte architetturali influenzino i tempi di risposta più che la pura complessità algoritmica asintotica.

#### Ottimizzazione dell'Importazione Massiva

Per il popolamento iniziale del database (file CSV), il sistema evita l'approccio ingenuo basato su iterazioni di query `INSERT`.

- **Strategia:** Utilizzo dell'API `CopyManager` di PostgreSQL per trasferire i dati via stream diretto.
- **Analisi del miglioramento:** Sebbene la complessità di lettura del file rimanga lineare rispetto alla sua dimensione, il guadagno prestazionale deriva dall'**abbattimento dell'overhead di rete**. Invece di eseguire $N$ richieste/risposte tra applicazione e database per $N$ libri, viene eseguita un'unica operazione di I/O. Questo riduce i tempi di importazione da minuti a pochi secondi, eliminando la latenza del protocollo TCP/IP per ogni singolo record.

#### Ricerca e Scansione Dati

Le funzionalità di ricerca permettono all'utente di trovare libri filtrando per sottostringhe nel titolo o nell'autore (pattern SQL `%query%`).

- **Lato Database (Server):** L'uso del carattere jolly iniziale (`%`) nelle query `ILIKE` obbliga il DBMS a effettuare una scansione sequenziale della tabella. Il tempo di esecuzione dipende quindi linearmente dalla dimensione del dataset ($N$) e dall'efficienza interna dell'ottimizzatore di PostgreSQL nel gestire l'I/O.
- **Lato Client (Costruzione UI):** Una volta ricevuti i risultati, il Client deve mappare i DTO ricevuti in oggetti grafici per la `ListView`. Questo processo ha un costo lineare $\mathcal{O}(R)$, dove $R$ è il numero di libri restituiti dalla ricerca.

#### Autenticazione

Il processo di login si compone di due fasi distinte a livello di database:

1. **Recupero Utente:** Il database ricerca la riga corrispondente allo `userid` fornito. Poiché `userid` è definito come Chiave Primaria, PostgreSQL utilizza un indice per questa operazione. Il costo computazionale è quindi logaritmico $\mathcal{O}(\log U)$, dove $U$ è il numero di utenti registrati.
2. **Verifica Hash:** Una volta trovato il record, la funzione `crypt` verifica la password. Questa operazione ha un costo costante $\mathcal{O}(1)$ rispetto alla mole di dati nel DB, dipendente solo dal fattore di costo dell'algoritmo impostato.

## 2.7 Gestione dei File

Il progetto gestisce diverse tipologie di file, utilizzando percorsi relativi (Classpath) per garantire la portabilità su diversi sistemi operativi.

- **Dataset Dati (CSV):** Il file `BooksDatasetClean.csv` è collocato nella cartella `resources` del modulo `db_init`. Viene letto tramite `getResourceAsStream()`, evitando path assoluti (es. `C:/Users/...`).
- **Script SQL:** Il file `create-tables.sql` contenente la DDL del database viene letto come stream di testo, filtrando commenti e righe vuote prima dell'esecuzione.
- **Interfacce Grafiche (FXML):** I file di layout JavaFX sono caricati dinamicamente dalla classe `App` utilizzando `FXMLLoader.load(getClass().getResource(...))`.
- **Configurazione Ambiente (.env):** È stato introdotto un meccanismo ibrido per la configurazione delle credenziali del database all'avvio del server. Il sistema verifica la presenza di un file `.env` nella directory di esecuzione.
  - **Caricamento Automatico:** Se il file esiste, le credenziali vengono caricate automaticamente senza input utente.
  - **Fallback Console:** Se il file è assente, il sistema richiede le credenziali via console (come da specifiche). In questa modalità, premendo semplicemente `Invio`, vengono utilizzati i valori di default preimpostati.

### 2.7.1 Specifica del Formato Dataset

Il sistema si aspetta in input un file denominato `BooksDatasetClean.csv`. Il file deve rispettare il formato CSV, con intestazione obbligatoria:

```
titolo,autori,anno_pubblicazione,editore,categorie
"The Great Gatsby","F. Scott Fitzgerald",1925,"Scribner","Classic,Fiction"
"1984","George Orwell",1949,"Secker & Warburg","Dystopian,Sci-Fi"
...
```

Listing 2.6: Struttura del file CSV atteso

La corrispondenza tra le colonne del CSV e le colonne della tabella database è mappata posizionalmente nel comando `COPY` eseguito durante l'inizializzazione.

# Capitolo 3

# Limiti della Soluzione Sviluppata

Nonostante l'architettura proposta soddisfi i requisiti funzionali del progetto, un'analisi della soluzione evidenzia alcuni limiti tecnici.

## 3.1 Gestione della Connessione al Database

Attualmente, la classe `DatabaseManager` implementa il pattern Singleton mantenendo un'unica istanza di `java.sql.Connection` attiva verso PostgreSQL.

- **Problema:** Sebbene i metodi siano sincronizzati per garantire la thread-safety, l'uso di una singola connessione serializza di fatto l'accesso al database. In uno scenario con molteplici utenti concorrenti, questo crea un bottleneck, aumentando i tempi di attesa per le operazioni di I/O.
- **Soluzione Futura:** Implementazione di un *Connection Pool* per gestire un pool di connessioni riutilizzabili, permettendo l'esecuzione parallela delle query.

## 3.2 Scalabilità e Paginazione dei Risultati

Le metodi di ricerca nella classe `BookQueries` (es. `searchByTitle`) restituiscono una `LinkedList<Libri>` completa contenente tutti i record trovati.

- **Problema:** In presenza di un dataset di grandi dimensioni (migliaia di libri), caricare l'intero Result Set in memoria e trasferirlo via RMI in un unico blocco può causare un elevato consumo di RAM lato Server e latenza di rete significativa.
- **Soluzione Futura:** Introduzione della paginazione lato server (SQL `LIMIT` e `OFFSET`) per restituire i dati a blocchi gestibili, migliorando la reattività dell'interfaccia utente.

## 3.3 Sicurezza del Canale di Comunicazione

Il sistema gestisce la sicurezza delle password tramite hashing (pgcrypto), ma il canale di trasporto RMI utilizza socket standard.

- **Problema:** Il traffico di rete tra Client e Server viaggia in chiaro. Sebbene le password non siano salvate in chiaro nel DB, esse transitano sulla rete durante la fase di login/registrazione, esponendo il sistema a potenziali attacchi *Man-in-the-Middle*.
- **Soluzione Futura:** Configurazione di RMI per utilizzare socket SSL/TLS (RMI over SSL) per cifrare l'intero canale di comunicazione.

## 3.4 Ottimizzazione del Calcolo del Voto Finale

Attualmente, il punteggio complessivo di un libro è calcolato dinamicamente a runtime, aggregando le singole recensioni (Stile, Contenuto, ecc.) ogni volta che viene richiesto il dettaglio di un libro.

- **Problema:** Al crescere del numero di valutazioni, l'esecuzione ripetuta di funzioni di aggregazione SQL (`AVG`) impatta negativamente sulle performance del DBMS, aumentando la latenza per l'utente finale durante la navigazione del catalogo.
- **Soluzione Futura:** Si prevede di aggiungere una colonna `voto_medio` direttamente nella tabella `Libri`, aggiornata automaticamente tramite comando SQL o processi batch ogni volta che viene inserita una nuova valutazione. Questo renderebbe la lettura del voto un'operazione immediata a costo costante.

## 3.5 Gestione Errori nei Metodi Get

L'attuale implementazione dei metodi di lettura (es. getter per ID o query di ricerca) gestisce i casi di "dato non trovato" restituendo valori `null` o collezioni vuote, senza fornire un contesto dettagliato sull'esito.

- **Problema:** L'uso di valori `null` espone il Client al rischio di `NullPointerException` se i controlli non sono capillari e crea ambiguità semantica (non è chiaro se il dato non esiste o se c'è stato un errore di connessione).
- **Soluzione Futura:** Gestire esplicitamente la presenza o l'assenza del valore, migliorando la robustezza del codice e la chiarezza.

# Bibliografia


[1] Oracle, *Javadoc Tool Documentation*, https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

[2] Oracle, *Java Remote Method Invocation (RMI) Specification*, https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html

[3] PostgreSQL Global Development Group, *PostgreSQL 14 Documentation*, https://www.postgresql.org/docs/14/index.html

[4] PostgreSQL Modules, *pgcrypto Extension - Cryptographic functions for PostgreSQL*, https://www.postgresql.org/docs/current/pgcrypto.html

[5] OpenJFX, *JavaFX Documentation*, https://openjfx.io/