Laboratorio Interdisciplinare B A.A. 2024/2025

Book Recommender

Manuale TECNICO

Autori:

Abou Aziz Sara Hesham Abdel Hamid

Hidri Mohamed Taha

Zoghbani Lilia

Ben Mahjoub Ali

Versione 2.0

Indice

1. Introduzione

“Book Recommender” è un sistema per la valutazione e raccomandazione di libri, che permette agli utenti registrati di inserire recensioni e a tutti gli utenti di consultare le valutazioni e ricevere consigli di lettura. Per ulteriori informazioni sul funzionamento del programma, incluse le istruzioni per l’avvio, consultare il Manuale Utente.

2. Premessa

In questo manuale non verrà discusso il funzionamento di tutte le classi e i metodi contenuti al loro interno: verranno menzionati i componenti cruciali al funzionamento del programma e quelli di maggior rilevanza e/o complessità. Per un elenco completo e dettagliato, fare riferimento alla Javadoc del progetto.

Si segnala, inoltre, che non tutti i diagrammi ***UML*** prodotti sono inclusi in questo manuale. Essi sono allegati alla documentazione tecnica e possono essere consultati nella cartella dedicata.

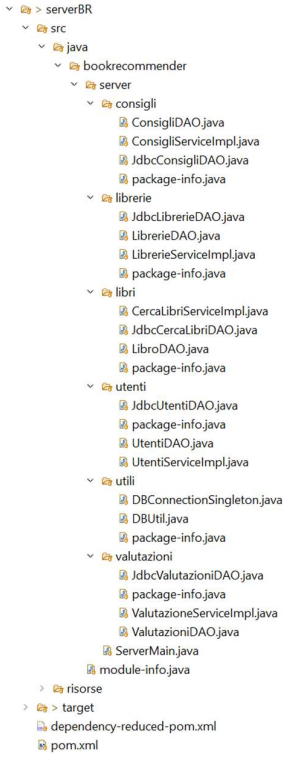
Infine, si consiglia di leggere il ***Manuale Tecnico del Database*** prima di questo, poiché per cogliere alcune scelte architetturali è necessaria una chiara comprensione delle entità relazionali e delle loro associazioni.

3. Software Design

Per lo sviluppo dell’applicazione si è scelto Java SE 17 per sfruttare le funzionalità moderne del linguaggio, ridurre il codice boilerplate e migliorare la manutenibilità complessiva. Il sistema è stato progettato con un’architettura client-server: il client fornisce l’interfaccia grafica sviluppata in JavaFX, che favorisce la separazione tra View e logica applicativa, mentre il server espone i servizi via Java RMI per rendere semplice e diretta la comunicazione remota. La persistenza è affidata a PostgreSQL tramite JDBC. L’applicazione è stata testata su sistemi Windows (10/11) e Linux (Fedora 40).

3.1. Struttura dell’applicazione

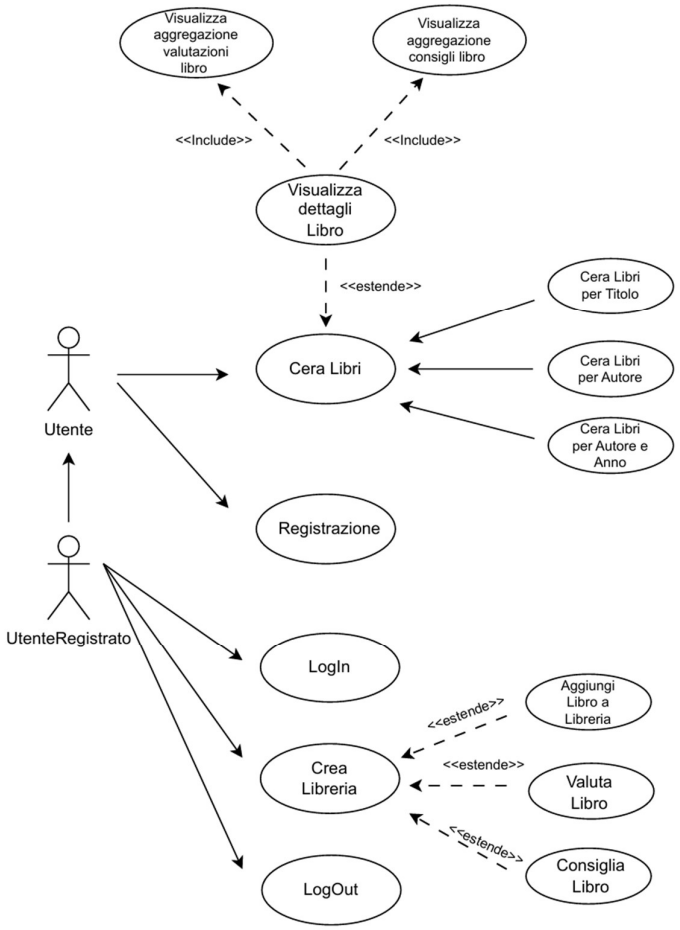
Di segito viene mostrata la struttura dei principali moduli dell’applicazione:



Il sistema BookRecommender è stato sviluppato con un'architettura client-server, seguendo un approccio modulare per garantire una chiara separazione delle responsabilità, facilitare la manutenzione e promuovere il riutilizzo del codice. Il progetto è suddiviso in tre moduli Maven distinti:

- **serverBR**: contiene il server;
- **clientBR**: contiene il client;
- **inComune**: contenente le interfacce e le entità condivise tra client e server.

Di seguito è riportato l'Use Case Diagram elaborato a seguito della definizione dei requisiti. In caso di dubbi, si consiglia di fare riferimento al documento di *Analisi e Specifica dei Requisiti*.



Figura[1]: Use Case Diagram

3.2. Connessione al database

Il sistema centralizza la gestione delle credenziali e della connessione al database PostgreSQL tramite la classe ***DBConnectionSingleton***. All'avvio del server, i parametri di accesso (JDBC URL, username, password) vengono inizializzati una sola volta, evitando che ogni componente debba gestire direttamente le credenziali.
L'inizializzazione avviene tramite:

```
public static void initialiseConnection(String jdbcUrl, String user, String pwd) throws SQLException {
```

```
        DBConnectionSingleton.jdbcUrl = jdbcUrl;
        DBConnectionSingleton.username = user;
        DBConnectionSingleton.password = pwd;

        if (connection == null) {
            synchronized (DBConnectionSingleton.class) {
                if (connection == null) {
                    closeConnectionQuietly();
                    connection = DriverManager.getConnection(jdbcUrl, user, pwd);
                }
            }
        }
    }
```

Dopo l'inizializzazione, l'accesso operativo al database avviene sempre tramite il metodo:

```
public static Connection openNewConnection() throws SQLException {
    if (jdbcUrl == null) {
        throw new SQLException("Database non inizializzato. Chiamare initialiseConnection(...)
prima.");
    }
    return DriverManager.getConnection(jdbcUrl, username, password);
}
```

Ogni **DAO** (Data Access Object.) utilizza *openNewConnection()* all'interno di un blocco try-with-resources, garantendo che ogni thread lavori su una connessione separata e che le risorse vengano sempre rilasciate correttamente.
Questa scelta progettuale permette di mantenere il codice semplice e leggibile, centralizzando la logica di connessione ma evitando al tempo stesso l'uso di una connessione unica condivisa, che avrebbe potuto generare problemi di concorrenza.

Attenzione: l'uso del pattern Singleton in questo contesto è stato adottato principalmente per semplicità. Tuttavia, in applicazioni reali e ad alto carico si consiglia di sostituire questa implementazione con un connection pool. Una soluzione moderna e affidabile è HikariCP, che offre un pooling efficiente e migliora la gestione delle risorse in scenari multi-thread complessi.

3.3. Traduzione delle entità relazionali in entità Java

Le principali entità sono:

- Utenti registrati
- Libri
- Librerie
- Valutazioni
- Consigli

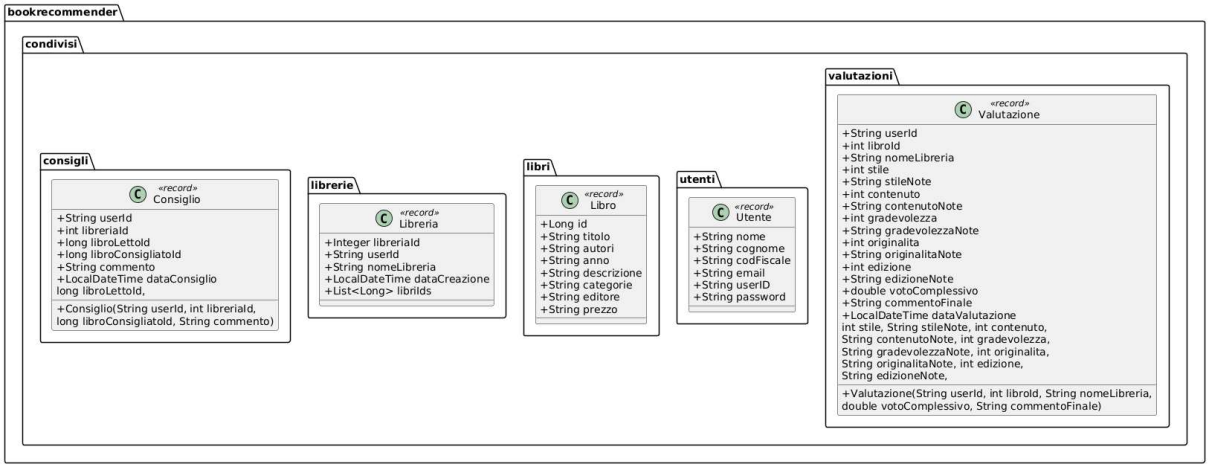
Queste corrispondono alle tabelle del database UtentiRegistrati, Libri, Librerie, ValutazioniLibri, ConsigliLibri.

Per ciascuna tabella, è stato creato un record Java (POJO, Plain Old Java Object) in cui ogni campo Java rappresenta esattamente una colonna della tabella SQL. Nella Figura [2] è presente il Class Diagram di queste entità.

Questa scelta consente una mappatura chiara e immediata tra il livello dati e il livello logico, semplificando la serializzazione, la validazione e la gestione delle informazioni tra client, server e database.

L’approccio adottato favorisce la leggibilità del codice e la manutenzione futura, riducendo la complessità di conversione tra formati e minimizzando il rischio di errori di mapping. Inoltre, la struttura uniforme delle entità facilita l’integrazione con i servizi RMI e la realizzazione di funzionalità avanzate come la raccomandazione e l’analisi delle valutazioni.

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



Figura[2] Class Diagram Entità POJO.

3.3.1. Architettura dei record Java: modello base e dettagliato

Nel sistema sono stati adottati due livelli di rappresentazione per le entità Valutazione e Consiglio: un record "base" e un record "dettagliato", in modo da separare chiaramente le responsabilità tra persistenza dei dati e presentazione nell'interfaccia utente.

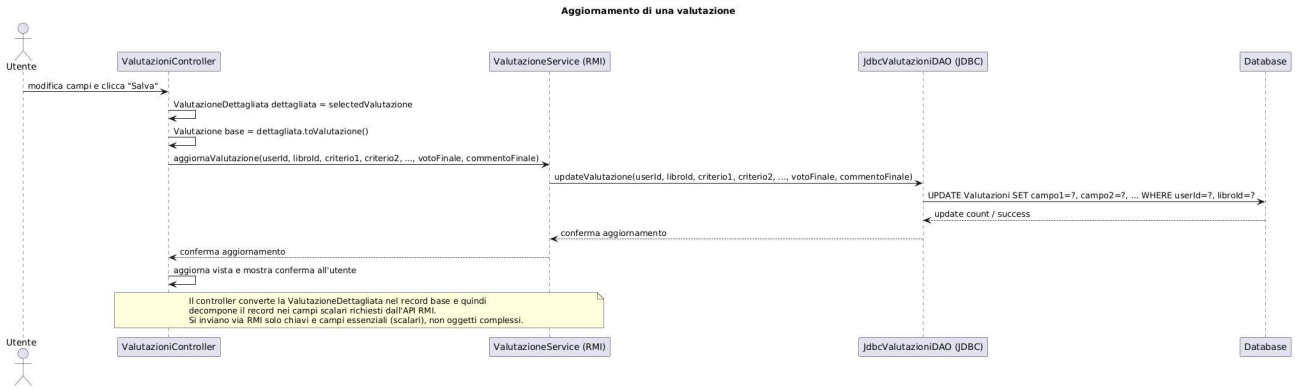
- **Record base (Valutazione, Consiglio):** struttura normalizzata, corrisponde alle colonne/tabelle DB; usato per persistenza (INSERT/UPDATE/DELETE) e per la logica di business.
- **Record dettagliato (ValutazioneDettagliata, ConsiglioDettagliato):** arricchisce il record base con campi testuali (titoli, autori, nome libreria) ottenuti con JOIN lato server; pensato per la visualizzazione nella UI.

Esempio di flusso:

Il controller carica la lista dettagliata via RMI:

```
List<ValutazioneDettagliata> valutazioni = valutazioneService.listValutazioniDettagliateByUser(currentUserId);
valutazioniListView.getItems().setAll(valutazioni);
```

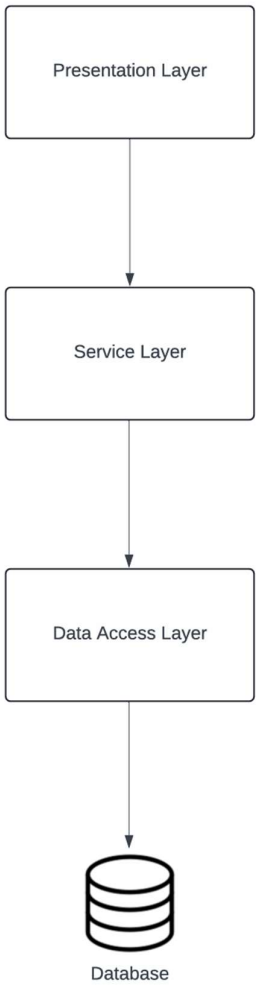
Per operazioni di scrittura (aggiornamento o cancellazione) il controller estrae dalla versione dettagliata i dati essenziali e li invia al service; nella API attuale i metodi RMI accettano parametri scalari; quindi, il controller deve decomporre il record in quei parametri. Questo riduce il carico di dati scambiati e mantiene separazione di responsabilità (presentazione vs persistenza).
Il sequence diagram riportato nella Figura[3] mostra le interazioni tra gli oggetti durante l’aggiornamento di una valutazione.



Figura[3]: sequence diagram aggiornamento di una valutazione

3.4. Architettura

Nell'applicazione è stata adottata una classica architettura a tre livelli, come mostrato dalla Figura[4] seguente.



Figura[4]: 3-Tier Architecture.

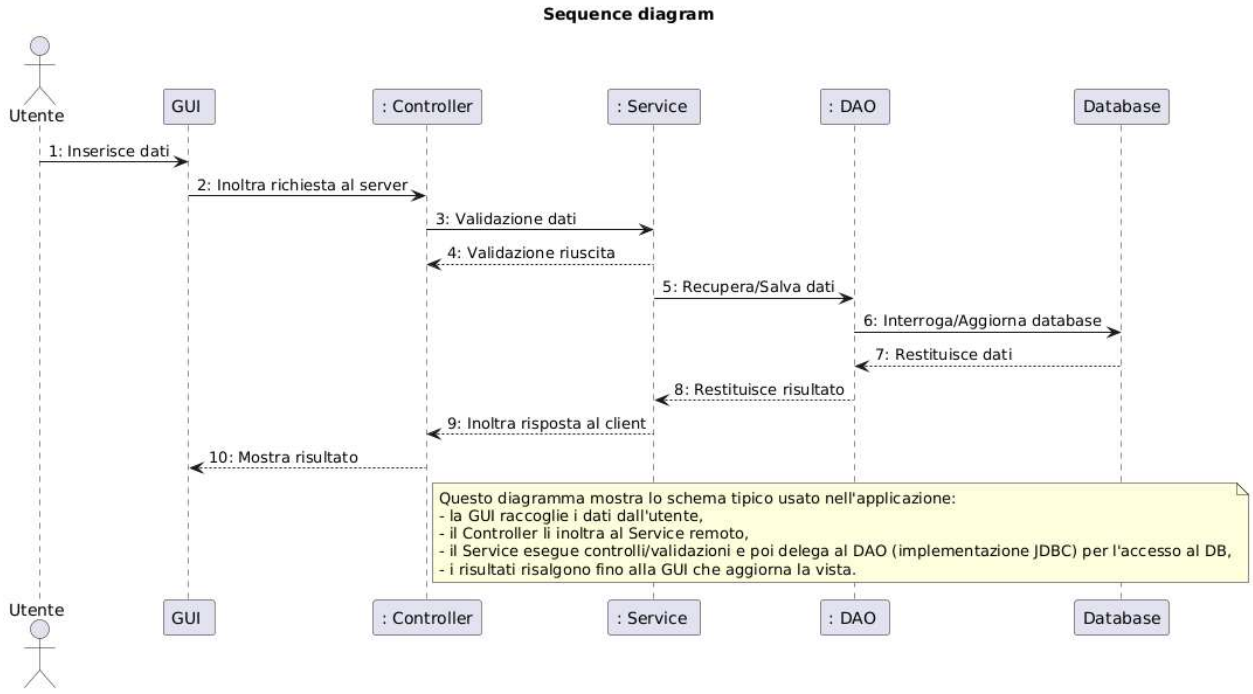
3.4.1. Descrizione dei livelli

Dal basso verso l'alto, l'architettura è composta dai seguenti livelli:

- **Persistence Layer (Database):** livello di persistenza dei dati, implementato utilizzando il database relazionale PostgreSQL.
- **Data Access Layer:** fornisce un'interfaccia di accesso ai dati, astratta tramite il pattern strutturale *Data Access Object (DAO)*, isolando la logica di accesso ai dati dalle altre parti del sistema.
- **Service Layer:** implementa la logica di business dell'applicazione, occupandosi della validazione dei dati provenienti dal client e del logging delle eccezioni provenienti dal *Data Access Layer*.
- **Presentation Layer**: corrisponde all'interfaccia grafica utente (GUI) e alla logica di presentazione, gestendo l'interazione tra l'utente e il sistema.

Ogni livello, ad eccezione del livello di presentazione, espone delle interfacce utilizzate dalle implementazioni del livello superiore. Questo approccio garantisce una **netta separazione delle responsabilità**, facilitando la manutenzione del sistema e migliorando la modularità del codice.

Nella Figura[5] è rappresentato un Sequence Diagram semplificato che mostra una tipica interazione tra gli oggetti del sistema.



Figura[5]: Sequence Diagram

3.4.2. Data Access Layer

Il Data Access Layer (DAL) implementa il pattern Data Access Object (DAO), fornendo un'interfaccia astratta per l'accesso ai dati persistenti del sistema. Questo strato si frappone tra il Service Layer e il database PostgreSQL, isolando la logica di accesso ai dati dal resto dell'applicazione.

3.4.2.1. Pattern DAO implementati

l sistema implementa i seguenti componenti DAO principali:

- **UtentiDAO**: gestisce la persistenza degli utenti registrati nel sistema, esponendo operazioni come:
  ```
  boolean save(Utenti utente);
  Utenti findByUsername(String username);
  boolean update(Utenti utente);
  boolean delete(String username);
  ```
- **LibrerieDAO**: si occupa delle operazioni relative alle librerie personali degli utenti:
  ```
  Libreria creaLibreria(Libreria libreria);
  List<Libreria> getLibrerieByUserId(String userId);
  boolean aggiungiLibroALibreria(int libreriaId, long libroId);
  boolean rimuoviLibroDaLibreria(int libreriaId, long libroId);
  boolean isLibroInLibreria(int libreriaId, long libroId);
  ```
- **ValutazioniDAO:** gestisce le valutazioni dei libri:
  ```
  boolean salvaValutazione(String userId, int libroId, String nomeLibreria,
                           int stile, String stileNote, /* altri parametri... */);
  List<ValutazioneDettagliata> findValutazioniDettagliateByUser(String userId);
  double calcolaMediaValutazioni(int libroId);
  ```
- **ConsigliDAO:** responsabile della persistenza e del recupero dei consigli sui libri:

  ```
  void add(String userId, int libreriaId, long libroLettoId, long libroConsigliatoId, String
  commento);
  List<Consiglio> findByUser(String userId);
  List<ConsiglioDettagliato> findDettagliatiByUser(String userId);
  ```

3.4.2.2. Implementazione JDBC

Ogni interfaccia DAO è implementata da una classe che utilizza JDBC con:

- **Prepared statements** per prevenire SQL injection
- **Try-with-resources** per la gestione corretta delle connessioni
- **Metodi di mapping** per convertire ResultSet nei record Java

Un aspetto distintivo è l'uso di metodi find...Dettagliati() che eseguono join SQL per recuperare in un'unica query tutti i dati necessari alla visualizzazione:

Esempio di query per recuperare consigli con dettagli sui libri e librerie:

```
SELECT c.*, l.nome_libreria,
ll.titolo AS titolo_letto, ll.autori AS autore_letto,
lc.titolo AS titolo_consigliato, lc.autori AS autore_consigliato
FROM ConsigliLibri c
JOIN Librerie l ON c.libreria_id = l.libreria_id
JOIN Libri ll ON c.libro_letto_id = ll.id
JOIN Libri lc ON c.libro_consigliato_id = lc.id
```

```
WHERE c.user_id = ?
```

Nella Figura [6] è rappresentato il class diagram che chiarisce le principali classi e interfacce coinvolte nella gestione dei consigli, mostrando DTO, interfaccia RMI, implementazione server e DAO JDBC e le loro relazioni di delega.

Figura[6]: Class Diagram Servizio Consigli: Service, DAO JDBC e DTO

3.4.3. Service Layer

Il Service Layer costituisce funge da intermediario tra l'interfaccia utente (GUI) e il layer di accesso ai dati (DAL). È stato progettato seguendo il principio del "Design for Change": nonostante la logica di business attuale sia relativamente semplice, questo livello è strutturato per supportare la crescita e adattarsi facilmente a nuove funzionalità o cambiamenti di specifica.

L'introduzione di questo strato intermedio comporta un lieve aumento di complessità iniziale, ma i vantaggi superano ampiamente questo costo: si ottiene una netta separazione delle responsabilità, dove la logica di validazione e le regole applicative rimangono confinate nel Service Layer, mentre il Data Access Layer si concentra esclusivamente sulla persistenza dei dati.

3.4.3.1. Interfacce condivise

Le interfacce di servizio sono definite nel modulo condiviso inComune poiché devono essere accessibili sia al client che al server. Tutte queste interfacce estendono java.rmi.Remote per consentire l'invocazione remota attraverso RMI.

Esempio di interfaccia di servizio dal package bookrecommender.condivisi.utenti

```
public interface UtentiService extends Remote {

   boolean authenticateUser(String username, String password) throws RemoteException;
   boolean registerUser(Utenti utente) throws RemoteException;
        // … altri metodi …
}
```

Per i dettagli implementativi del service layer, si rimanda alla Figura[6], che mostra il class diagram del servizo “consigli” (interfaccia remota, implementazione server, DAO JDBC e DTO).

3.4.3.2. Separazione delle responsabilità

Il Service Layer permette di separare chiaramente:

- **Logica di presentazione**: gestita nei controller client (come ***ConsigliaLibroController***, ***ValutazioniController***)
- **Logica di business**: implementata nelle classi service (come ***ConsigliServiceImpl***, ***ValutazioneServiceImpl***)
- **Logica di accesso ai dati**: confinata nei DAO (come ***JdbcConsigliDAO***, ***JdbcValutazioniDAO***)

Inoltre il Service Layer funge da "gatekeeper" impedendo accessi diretti al database dal livello di presentazione, riducendo significativamente la superficie di attacco. Questo strato centralizza i controlli di autorizzazione, i meccanismi di auditing e la validazione dei dati in ingresso.

La separazione introdotta dal Service Layer migliora:

- **Manutenibilità**: modifiche alla logica di business non impattano il layer di persistenza
- **Testabilità**: possibilità di testare la logica di business in isolamento tramite mock dei DAO
- **Modularità**: componenti più coesi con responsabilità ben definite

3.4.3.3. Ciclo di vita dei servizi

Nella fase di avvio del server (classe ***ServerMain***), il flusso operativo segue questi passi:

1. Inizializzazione dei parametri di connessione al database:
   ```
   DBConnectionSingleton.initialiseConnection(jdbcUrl, user, password);
   ```

2. Creazione delle istanze dei servizi:
   ```
   UtentiService utentiService = new UtentiServiceImpl();
   ValutazioneService valutazioneService = new ValutazioneServiceImpl();
   ConsigliService consigliService = new ConsigliServiceImpl();
   LibrerieService librerieService = new LibrerieServiceImpl();
   ```

```
CercaLibriService cercaLibriService = new CercaLibriServiceImpl();
```

3. Pubblicazione dei riferimenti nel registro RMI:
```
Registry registry = LocateRegistry.createRegistry(1099);
registry.rebind("UtentiService", utentiService);
registry.rebind("ValutazioneService", valutazioneService);
// ... altri servizi ...
```

Questo approccio garantisce che i servizi vengano esposti solo dopo che la configurazione di persistenza è stata correttamente inizializzata.

3.4.3.4. Gestione delle eccezioni

Il Service Layer implementa un modello di gestione delle eccezioni che cattura le eccezioni tecniche (es. SQLException) poi le trasforma in eccezioni di business o RemoteException, registra dettagli dell'errore tramite il sistema di logging e restituisce al client informazioni appropriate.

3.4.4. Presentation Layer

Il Presentation Layer dell'applicazione ***clientBR*** rappresenta il componente responsabile dell'interfaccia utente (GUI) e della gestione dell'interazione con l'utente. Per soddisfare i requisiti dei committenti e offrire un'esperienza utente moderna e intuitiva, è stata sviluppata una GUI robusta e dinamica basata sul framework JavaFX.

L'architettura del Presentation Layer si fonda su tre elementi chiave:

- **File FXML** per la definizione della struttura visuale delle interfacce
- **Controller JavaFX** per la gestione degli eventi e della logica di presentazione
- **Un sistema centralizzato di navigazione** tramite la classe ViewsController

3.4.4.1. Separazione delle responsabilità tramite pattern MVVM-like

Per mantenere una rigorosa separazione tra la logica di presentazione e il modello dei dati, l'architettura del client si ispira ai principi del pattern Model-View-ViewModel (MVVM). Questo approccio è evidente nella struttura dei controller come ***ConsigliaLibroController*** e ***CercaLibriController***, che fungono da intermediari tra l'interfaccia utente e i servizi remoti.

I controller gestiscono la trasformazione dei dati provenienti dal backend in formato adatto alla visualizzazione, mantenendo le entità di dominio (come ***Libro*** e ***Consiglio***) libere da dipendenze JavaFX. Questo design consente di separare chiaramente:

- Entità di dominio: Definite nel modulo condiviso inComune, sono Java record puri
- Logica di presentazione: Implementata nei controller JavaFX
- Struttura dell'interfaccia: Definita nei file FXML

3.4.4.2. Gestione centralizzata della navigazione

L'applicazione utilizza un gestore centralizzato per la navigazione tra le diverse viste, implementato nella classe ***ViewsController***. Questo componente fornisce metodi statici per caricare le diverse schermate

dell'applicazione, mantenendo una singola Scene sul Stage principale e sostituendo il nodo radice quando necessario.

I controller interagiscono con il *ViewsController* per gestire le transizioni tra le viste, come evidenziato in ***BenvenutiController***, per esempio questo metodo gestisce l'evento di click sul pulsante "Accedi", ferma l'animazione di sfondo e naviga alla vista di login.

```
    public void showLoginView() {
    stopAnimation();
    ViewsController.mostraLogin();
}
```

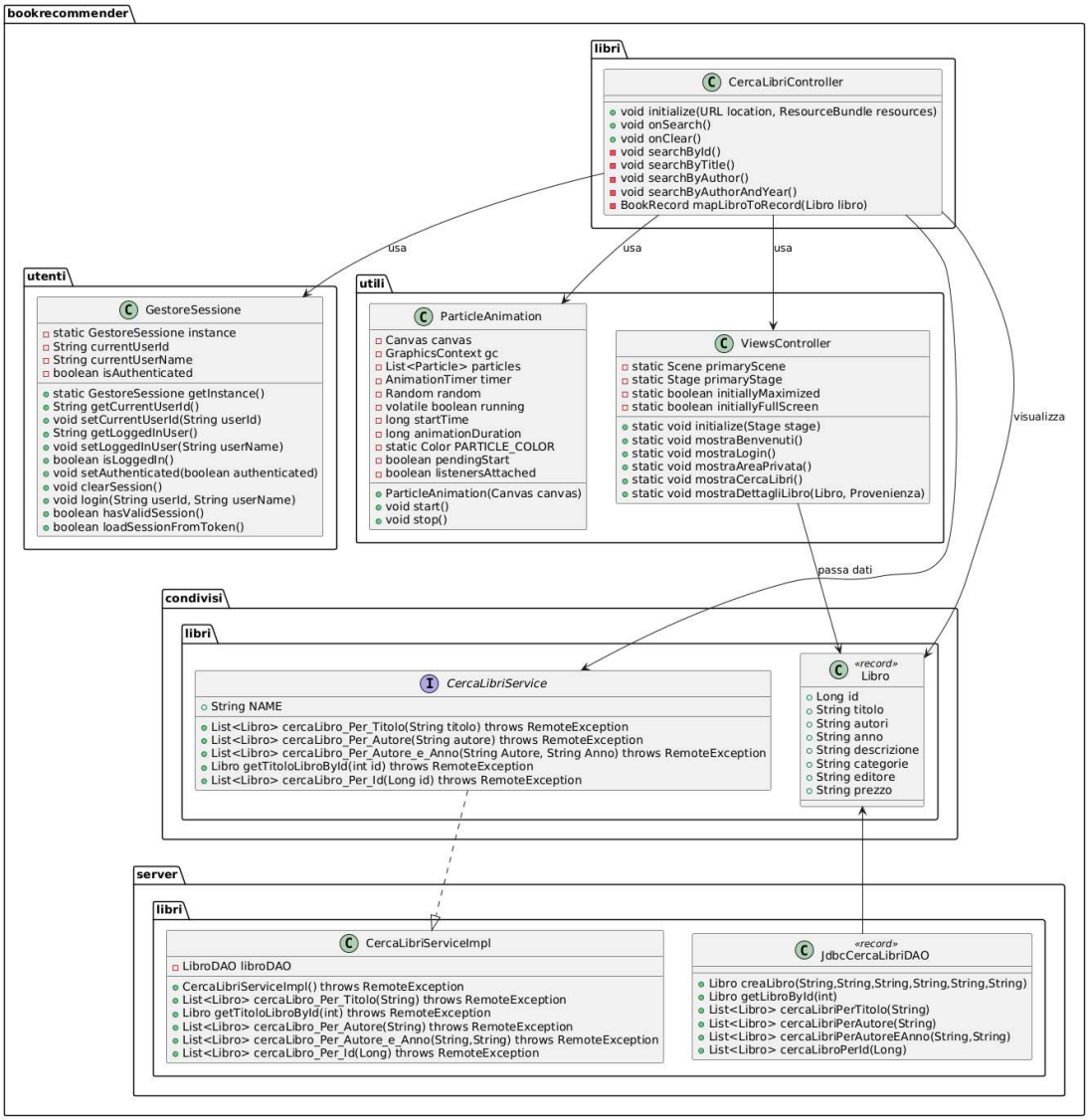
Questo approccio centralizzato garantisce coerenza nella navigazione e semplifica la manutenzione dell'applicazione.

3.4.4.3. Interazione con i servizi remoti

I controller del Presentation Layer comunicano con i servizi remoti del server tramite le interfacce RMI definite nel modulo inComune.

I controller stabiliscono connessioni RMI durante l'inizializzazione, recuperano dati dai servizi remoti e gestiscono le eccezioni per fornire feedback appropriati all'utente.

Nella Figura[7] viene illustrato il flusso di interazione per la ricerca dei libri dal Controller client fino al DAO JDBC passando tramite l'interfaccia RMI e l'implementazione server, includendo il passaggio dei DTO Libro.



Figura[7]: Componenti e dipendenze del servizio Libri

3.4.4.4. View e relativi Controller

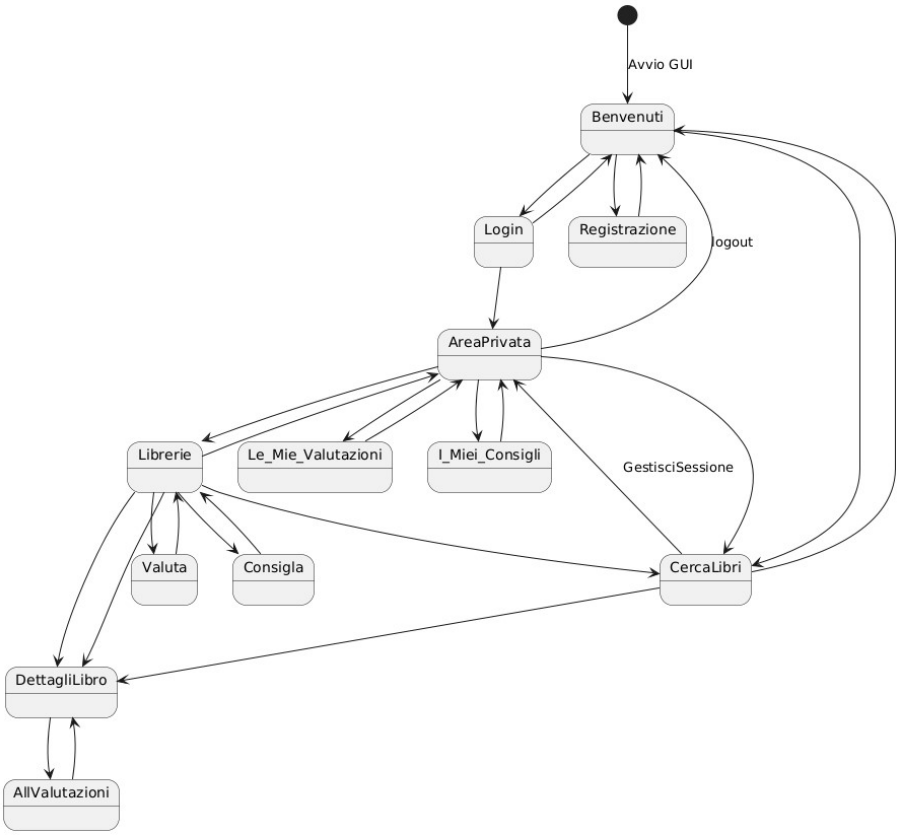
Nello specifico, le View e i rispettivi controller implementano le seguenti funzionalità:

- ***benvenuti-view.fxml*** e ***BenvenutiController***
  - Avvio/gestione dell'animazione di sfondo (***ParticleAnimation***) e sua pulizia alla chiusura.
  - Gestione dei pulsanti di navigazione: avvia la vista di login, registrazione o ricerca.
  - Micro-interazioni UI (effetti di hover sui pulsanti) per migliorare l'usabilità.
- ***login-view.fxml*** e ***LoginController***
  - Raccolta e validazione delle credenziali (username/email/codice fiscale + password).
  - Contatto con il servizio remoto utenti (via RMI) per autenticazione e gestione degli esiti (successo / errore).
  - Inizializzazione animazione di sfondo e gestione risorse correlate.
- ***registrazione-view.fxml*** e ***RegistrazioneController***

    - Raccolta dei dati di registrazione e validazione lato client (campi richiesti, conferma password).
    - Invio dati al servizio remoto utenti per la registrazione.
    - Feedback all'utente tramite dialoghi/alert e navigazione guidata.
- ***areaprivata-view.fxml*** e ***DashboardController***
    - Punto di ingresso per l'area privata: visualizza messaggio di benvenuto e offre navigazione verso sezioni principali.
    - Controllo della sessione utente (tramite ***GestoreSessione***) e reindirizzamento se non autenticato.
    - Avvio/stop dell'animazione di sfondo e gestione eventi di navigazione (ricerca, librerie, valutazioni, consigli).
- ***cercaLibri-view.fxml*** e ***CercaLibriController***
    - Modalità di ricerca multiple (ID, titolo, autore, autore+anno) con UI dinamica che mostra i campi rilevanti.
    - Lookup RMI per ***CercaLibriService***, invio query e visualizzazione risultati in TableView.
    - Modalità fallback demo se il servizio remoto non è disponibile; gestione errori di rete e UI.
- ***dettagliLibro-view.fxml*** e ***DettagliLibroController***
    - Visualizzazione dettagli completa del libro (titolo, autori, anno, editore, categorie, descrizione, prezzo).
    - Recupero e visualizzazione delle valutazioni medie e suggerimenti collegati.
- ***librerie-view.fxml*** e ***LibrerieController***
    - Gestione delle librerie personali: creazione, eliminazione, selezione libreria.
    - Aggiunta/rimozione di libri in una libreria (interazione con ***LibrerieService*** via RMI).
    - Popolamento e gestione di ListView e TableView
- ***consiglia-libro-view.fxml*** e ***ConsigliaLibroController***
    - Ricerca libri (via ***CercaLibriService***) e selezione fino a 3 libri da consigliare.
    - Raccolta commento opzionale e invio consigli al backend (***ConsigliService***).
    - Validazioni di business lato client (es. non consigliare a se stessi, evitare duplicati, rispetto del limite massimo).
- ***consigli-view.fxml*** e ***ConsigliController***
    - Visualizzazione elenco consigli dati dall'utente (ListView di ConsiglioDettagliato).
    - Visualizzazione pannello di dettaglio, modifica commento e cancellazione di un consiglio con conferma.
    - Coordinamento UI (stato, messaggi) e chiamate a servizi remoti per aggiornamento/persistenza.
- ***valutazioni-view.fxml*** e ***ValutazioniController***
    - Elenco delle valutazioni dell'utente e editor dettagliato per modificarle.
    - Interfacce a stelle per i vari criteri, calcolo dinamico del voto complessivo.
    - Persistenza delle modifiche via ***ValutazioneService*** (RMI) e gestione della navigazione.
- ***all-valutazioni-view.fxml*** e ***AllValutazioniController***
    - Caricamento e rendering delle valutazioni di un libro in una vista composta dinamicamente.
    - Creazione di nodi UI per ogni valutazione (metodo che costruisce pannelli con stelle e testi).
    - Gestione dei casi di errore e feedback visivo all'utente.
- ***valutazione-completa.fxml*** e ***ValutazioneCompletaController***

    - Visualizzazione e editing approfondito di una singola valutazione come aggregazione di tutte le altre.
- Risorse e utilità:
    - ***ViewsController*** (package bookrecommender.utili): controller centralizzato di navigazione che inizializza lo Stage principale e carica le viste FXML.
    - ***ParticleAnimation*** (package bookrecommender.utili): componente riutilizzabile che gestisce l'animazione a particelle (start/stop, adattamento al ridimensionamento, durata limitata).
    - ***GUI*** / ***BookRecommender***: punti di avvio dell'applicazione client e integrazione con ***ViewsController***.
    - ***JGestoreSessione***: fornisce l'ID e lo stato dell'utente autenticato ai controller.

Nella Figura[8] viene illustrato il grafo degli stati di navigazione: dagli stati iniziali pubblici si raggiungono gli stati privati e, tramite transizioni etichettate, gli stati secondari di consultazione e azione.



Figura[8]: Componenti e dipendenze del servizio Libri

3.4.4.5. Stile e Sistema di Animazione a Particelle

Per garantire una user experience curata e professionale, l'applicazione utilizza un file CSS dedicato (styles.css) che definisce l'aspetto visivo degli elementi dell'interfaccia.

Inoltre, per arricchire l'esperienza utente e conferire un'identità visiva unica e moderna all'applicazione, è stato sviluppato un sistema di animazione a particelle personalizzato. Questo non è un semplice effetto

decorativo, ma una componente studiata per essere elegante, performante e non intrusiva, che funge da sfondo per le viste principali dell'applicazione.

L'animazione mira a creare un'identità visiva distintiva: l'effetto particellare, presente in diverse schermate chiave, stabilisce un filo conduttore visivo che rende l'applicazione immediatamente riconoscibile e coerente.

L'implementazione tecnica di questo aspetto si basa su un colore distintivo condiviso tra tutte le particelle:

```
private static final Color PARTICLE_COLOR = new Color(0, 0.847, 0.941, 1.0);
```

Inoltre fornisce un feedback dinamico e gradevole, attivandosi al caricamento di una vista per comunicare all'utente che l'interfaccia è viva e reattiva.

Le scelte tecniche di questa componente mirano a massimizzare l'impatto visivo minimizzando il carico sul sistema. L'animazione è intrinsecamente orientata alle performance, con una durata fissa di 5 secondi al termine della quale l'oggetto AnimationTimer di JavaFX viene completamente fermato:

```
public void stop() {
    running = false;
    try {
        timer.stop();  // Ferma completamente l'AnimationTimer
    } catch (Exception ignored) {}
}
```

Questa decisione è cruciale poiché, una volta conclusa la fase dinamica, l'animazione non consuma più cicli di CPU o GPU, trasformandosi in un semplice sfondo statico e garantendo che l'applicazione rimanga leggera e reattiva, senza impattare sulla durata della batteria dei dispositivi.

Inoltre, il design è responsivo: il numero di particelle e la loro area di movimento si adattano automaticamente alla dimensione della finestra. Questa responsività è ottenuta tramite il calcolo dinamico del numero di particelle in base all'area del canvas e l'adattamento automatico alle dimensioni del contenitore:

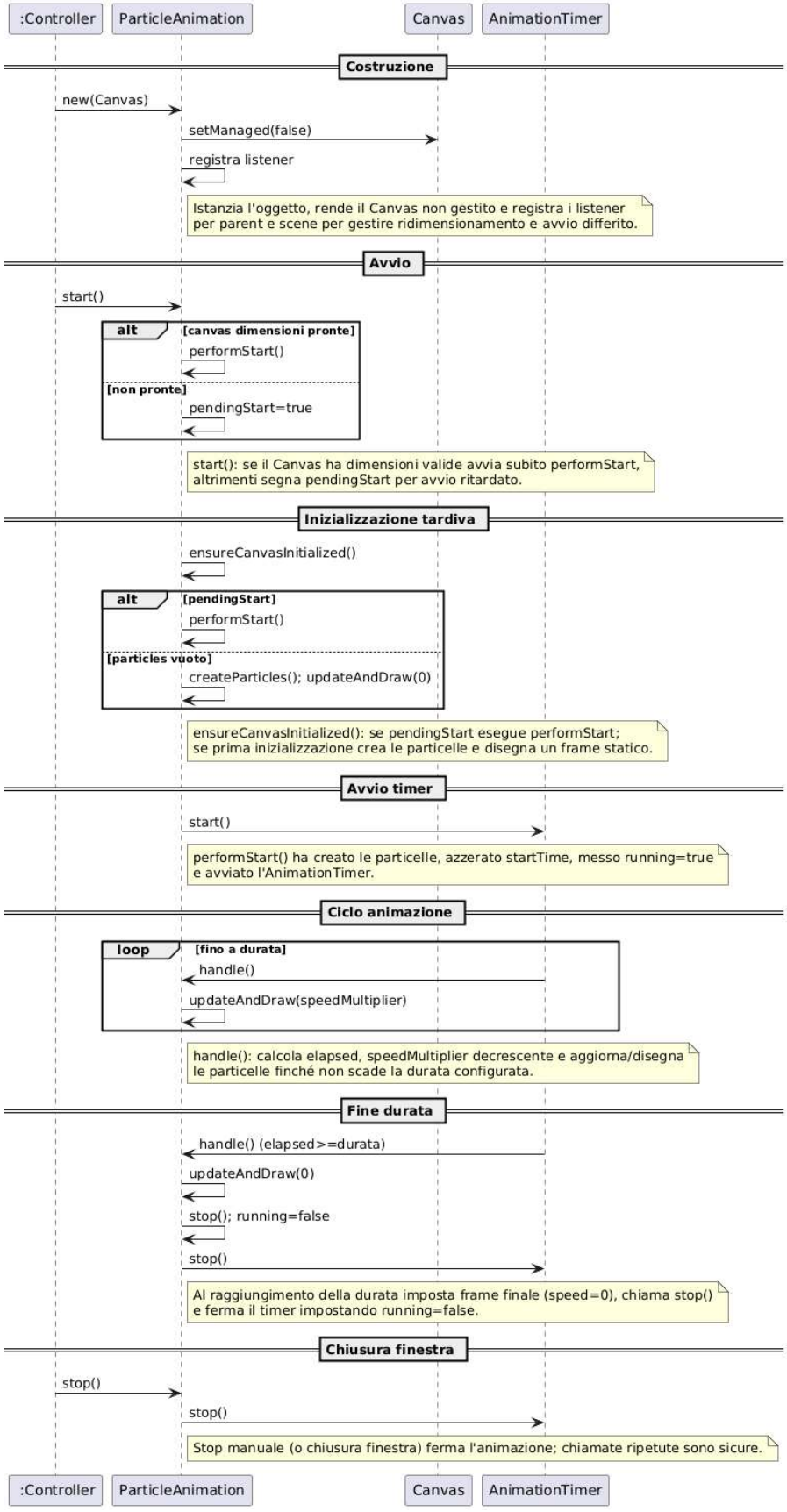
```
int numberOfParticles = (int) Math.max(1, (w * h) / 5000);

parent.layoutBoundsProperty().addListener((obs, oldBounds, newBounds) -> {
    applyBoundsToCanvas(newBounds);
    Platform.runLater(this::ensureCanvasInitialized);
});
```

La componente è auto-contenuta, con la logica interamente incapsulata nella classe ParticleAnimation e la classe Particle implementata come classe interna privata.
Questa scelta di design rende il componente robusto e di facile manutenzione, prevenendo dipendenze indesiderate nel resto del codice. L'utilizzo nei controller è infatti estremamente semplice.

L'effetto finale è un'animazione leggera, elegante e non invasiva che contribuisce significativamente all'identità visiva dell'applicazione senza compromettere l'usabilità o le prestazioni, dimostrando come anche elementi decorativi possano essere implementati con attenzione ai dettagli tecnici e all'efficienza.

Nella Figura[8] viene rappresentata la sequenza essenziale del ciclo di vita di ParticleAnimation: dalla costruzione con registrazione dei listener, allo start (immediato o differito), all'inizializzazione, al ciclo temporizzato di update/draw fino all'arresto per durata o chiusura finestra.



Figura[9]: Sequence Diagram Animazione di Sfondo ParticleAnimation

3.4.5. Organizzazione dei Package nel Sistema

Il sistema BookRecommender adotta un approccio "Package by Layer" per l'organizzazione del codice, con una chiara separazione dei componenti in base al loro ruolo architetturale e responsabilità tecnica. Questa strutturazione permette una netta divisione delle competenze e facilita sia la manutenzione che l'evoluzione del sistema.

All'interno di ciascun layer, i package sono ulteriormente suddivisi per area funzionale del dominio, migliorando così l'organizzazione e la leggibilità del codice:

- **Presentation Layer (clientBR)**:
  - bookrecommender.consigli: Controller per le funzionalità di raccomandazione
  - bookrecommender.librerie: Controller per la gestione delle librerie personali
  - bookrecommender.libri: Controller per la ricerca e visualizzazione dei libri
  - bookrecommender.utenti: Controller per l'autenticazione e gestione utente
  - bookrecommender.valutazioni: Controller per il sistema di recensioni
  - bookrecommender.utili: Componenti di utilità condivisi
- **Service Layer (inComune):**
  - bookrecommender.condivisi.consigli: Interfacce e DTO per il sistema di raccomandazione
  - bookrecommender.condivisi.librerie: Interfacce e DTO per le librerie personali
  - bookrecommender.condivisi.libri: Interfacce e DTO per la ricerca libri
  - bookrecommender.condivisi.utenti: Interfacce e DTO per la gestione utenti
  - bookrecommender.condivisi.valutazioni: Interfacce e DTO per le recensioni
- **Data Access Layer (serverBR)**
  - bookrecommender.server.consigli: Servizi e DAO per le raccomandazioni
  - bookrecommender.server.librerie: Servizi e DAO per le librerie personali
  - bookrecommender.server.libri: Servizi e DAO per la gestione dei libri
  - bookrecommender.server.utenti: Servizi e DAO per la gestione utenti
  - bookrecommender.server.valutazioni: Servizi e DAO per le valutazioni

3.5. Strutture dati utilizzate

Il sistema BookRecommender utilizza diverse strutture dati ottimizzate per le specifiche esigenze delle diverse componenti dell'applicazione. La scelta di queste strutture dati è stata guidata da considerazioni di efficienza, chiarezza semantica e coerenza con il paradigma di programmazione Java adottato. Questa sezione analizza le principali strutture dati impiegate nel sistema.

3.5.1. Java Records per Data Transfer Objects

BookRecommender fa ampio uso dei Java Records (introdotti in Java 16) come Data Transfer Objects (DTO) per il trasferimento di dati tra client e server tramite RMI.

I Records utilizzati sono:

- **Libro**: Rappresenta le informazioni essenziali di un libro nel catalogo
- **Utenti**: Contiene i dati di un utente registrato

- **Valutazione/ValutazioneDettagliata**: Rappresentano valutazioni con diversi livelli di dettaglio
- **Consiglio/ConsiglioDettagliato**: Contengono dati relativi alle raccomandazioni tra utenti
- **Libreria**: Memorizza i metadati di una libreria personale

3.5.2. List e derivati

List<T> è utilizzato estensivamente per rappresentare sequenze ordinate di dati, particolarmente per i risultati di ricerca e per le collezioni di entità correlate.

Implementazione principale:

- **ArrayList<T>**: Usata quando l'accesso casuale è frequente e l'inserimento/rimozione avviene principalmente alla fine.
    - Esempi di utilizzo: ***ParticleAnimation*** (gestione delle particelle), risultati di ricerca libri in ***ConsigliaLibroController***, liste di entità restituite dai servizi.

3.5.3. ObservableList per UI

JavaFX utilizza ObservableList<T> come estensione reattiva delle liste standard, permettendo il binding bidirezionale con i componenti dell'interfaccia utente.

```
private final ObservableList<Libro> selectedBooks = FXCollections.observableArrayList();
```

E' stata utilizzata principalmente per :

- ***ConsigliaLibroController:*** Gestione dei libri selezionati per le raccomandazioni
- Liste di risultati di ricerca: Aggiornamento immediato dell'interfaccia quando cambiano i dati
- ***LibrerieController***: Visualizzazione delle librerie personali dell'utente

3.5.4. Map

Le implementazioni di Map<K,V> sono utilizzate per diverse esigenze di mappatura chiave-valore nel sistema:

HashMap<K,V>: Utilizzata per lookup efficienti e conteggio occorrenze, per esempio in ***JdbcValutazioniDAO*** per costruire una rappresentazione flessibile di una valutazione:

```
Map<String, Object> v = new HashMap<>();
v.put("userId", rs.getString("user_id"));
v.put("libroId", rs.getInt("libro_id"));
v.put("stile", rs.getInt("stile_score"));
v.put("stileNote", rs.getString("stile_note"));
// ... altri campi ...
out.add(v);
```

La scelta di utilizzare HashMap offre accesso in O(1) alle operazioni di inserimento, ricerca e rimozione, contribuendo alla scalabilità e alle prestazioni del sistema quando si gestiscono collezioni di dati associativi.

3.5.5. Optional per Valori Potenzialmente Assenti

Optional<T> viene utilizzato sistematicamente per gestire valori che potrebbero non esistere, aumentando la robustezza del codice.

Migliora la sicurezza e la leggibilità del codice in situazioni dove un valore potrebbe essere assente. Come si vede nella gestione delle conferme dialog in ***ValutazioniController***, alert.showAndWait() restituisce un Optional<ButtonType> che permette di verificare esplicitamente con isPresent() se l'utente ha effettuato una selezione prima di accedervi con get(). Questo approccio elimina i null pointer exceptions, rende l'API più espressiva rispetto all'uso di valori null, e obbliga gli sviluppatori a considerare il caso in cui nessun valore sia disponibile. Altri esempi nel progetto includono l'uso di Optional nei risultati delle query database e nelle operazioni di ricerca, garantendo sempre una gestione esplicita dei casi limite senza compromettere l'integrità del codice.

Esempio di utilizzo in **ValutazioniController**:

```
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK) {
```

3.5.6. Wrapper per Binding JavaFX

JavaFX fornisce classi wrapper come ***SimpleStringProperty*** e ***SimpleObjectProperty<T>*** che incapsulano valori primitivi o oggetti con funzionalità di binding e listener.

Utilizzi tipici:

- Campi di testo reattivi che aggiornano automaticamente altre parti dell'UI
- Stati di abilitazione dei controlli (pulsanti, menu)
- Visualizzazione sincronizzata di oggetti di dominio nell'interfaccia utente
- Binding bidirezionale tra componenti UI e modelli di dati

In questo modo si semplifica significativamente la gestione dello stato dell'interfaccia utente e riduce il codice di sincronizzazione manuale.

3.6. Design Pattern adottati

Questa sezione elenca i principali design pattern utilizzati nel progetto, con lo scopo di fornire una panoramica architetturale delle soluzioni adottate durante la progettazione dell'applicazione. L'analisi dei pattern evidenzia le pratiche che hanno permesso di strutturare il codice in modo modulare, migliorare la manutenibilità e favorire il riuso dei componenti. Attraverso l'identificazione di questi pattern è possibile comprendere meglio l'organizzazione interna del sistema e le motivazioni dietro determinate scelte implementative:

- **Singleton:**
  Nel progetto il Singleton è usato per centralizzare risorse condivise e stato applicativo in modo semplice e immediato: ad esempio la gestione delle connessioni al database è incapsulata in una classe che garantisce un'unica inizializzazione e un punto di accesso globale, così come il gestore di sessione lato client fornisce uno stato utente condiviso accessibile da più controller. Questa scelta evita la dispersione dei parametri di configurazione in molteplici punti del codice e semplifica

l'apertura/chiusura controllata delle risorse, permettendo allo stesso tempo un controllo centralizzato del lifecycle (vedi ***DBConnectionSingleton*** e ***GestoreSessione***).

- **DAO (Data Access Object):**
  La persistenza è organizzata attorno a DAO che separano la logica SQL dalla logica di business e dai servizi remoti. Questo approccio rende naturale isolare query, mapping risultato-oggetto e gestione delle transazioni dentro classi dedicate (le implementazioni Jdbc*DAO), così che il resto dell'applicazione possa lavorare con metodi ben definiti senza conoscere i dettagli del database. Il vantaggio pratico è la testabilità e la possibilità di cambiare l'implementazione di persistenza con impatto minimo sul resto del codice.
- **Remote Facade / Service (RMI):**
  Il server espone insiemi di operazioni via interfacce remote che fungono da facciata per le funzionalità di business; le implementazioni concrete estendono l'infrastruttura RMI e incapsulano l'accesso ai DAO e alla logica server. Dal punto di vista dell'architettura, i client JavaFX comunicano con queste facciate remote come se fossero API locali, ottenendo un contratto stabile per operazioni come ricerca libri, consigli e valutazioni (es. ***CercaLibriService*** e le sue impl).
- **MVC lato client (JavaFX):**
  L'interfaccia utente è organizzata secondo il modello View-Controller: le view (FXML e componenti grafici) sono separate dai controller che orchestrano eventi, validazione e chiamate ai servizi. I controller mantengono solo la logica di interazione e delegano la persistenza e i servizi remoti, favorendo una separazione delle responsabilità che semplifica la manutenzione dell'interfaccia e il binding con i dati.
- **Observer / Binding (JavaFX):**
  Per mantenere la UI sincronizzata con i dati si fa uso estensivo di proprietà osservabili e liste osservabili: le ObservableList e le StringProperty permettono alla tabella e ad altri controlli di aggiornarsi automaticamente quando cambiano i dati sottostanti. Questo paradigma reattivo riduce il codice "plumbing" necessario per aggiornare manualmente la view e rende più fluida l'interazione utente.
- **Service Locator (comportamento di lookup RMI):**
  I client ottengono i riferimenti ai servizi remoti tramite lookup sul registry RMI anziché tramite injection: questa strategia è pratica per una soluzione RMI tradizionale perché rende esplicita la fase di discovery, ma in cambio accoppia i client al meccanismo di lookup e rende i test unitari un po' più laboriosi rispetto a un approccio basato su dependency injection.
- **DTO (Data Transfer Object)/ Record per trasferimento dati:**
  I dati scambiati fra client e server sono incapsulati in oggetti semplici, spesso implementati come record, che definiscono il formato delle informazioni trasferite. Questi DTO riducono l'overhead di serializzazione e mantengono il contratto dei dati chiaro e immutabile, facilitando sia il debugging sia la compatibilità tra versioni client/server.
- **Facade/Helper locali:**
  Nel client e nel server sono presenti componenti che centralizzano operazioni comuni (per esempio la navigazione delle view o animazioni riutilizzabili). Queste classi non sono facciate architetturali grandi ma svolgono la fu
  nzione pratica di nascondere dettagli implementativi e offrire API semplici ai consumatori interni.

4. Algoritmi impiegati

In questa sezione descriviamo gli algoritmi e le tecniche implementate nel progetto, spiegandone il ruolo funzionale e fornendo riferimenti al codice sorgente. Per ogni algoritmo indicheremo i file di riferimento in cui il comportamento è implementato e documentato.

- **Algoritmi di ricerca e filtro**
  La ricerca lato client segue logiche semplici ma efficaci: i controller leggono l'input utente, lo validano e poi eseguono la ricerca usando il servizio remoto se disponibile, altrimenti impiegano filtri su una lista locale. In modalità demo le operazioni di filtro sono implementate con Java Stream API (operazioni filter + collect) su una lista di BookRecord, con confronti case-insensitive per sottostringhe. Questo approccio è utilizzato per mantenere l'esperienza utente anche senza connessione al servizio remoto; è visibile in ***CercaLibriController.java*** (metodi *searchByTitle*, *searchByAuthor*, *searchByAuthorAndYear*) e in ***ConsigliaLibroController.java*** per la gestione della selezione dei libri.
- **Parsing e validazione dell'input**
  Le funzioni di validazione e parsing sono semplici e robuste: trim degli input, controlli di formato (es. *isValidYear* con regex \d{4}) e parsing sicuro di interi con gestione di NumberFormatException. Queste routine riducono errori runtime e sono centralizzate in helper come *safeGet* e *isValidYear* in ***CercaLibriController.java.***
- **Da Mapping DTO a View**
  La conversione tra oggetti trasferiti dal server (Libro DTO/record) e gli oggetti usati nella UI (BookRecord) è gestita da semplici mapper che normalizzano i valori null in stringhe "N/D". Questa mappatura è fatta in *CercaLibriController.mapLibroToRecord(Libro)* e garantisce che la TableView lavori sempre con proprietà JavaFX (SimpleStringProperty) per il binding.
- **Algoritmi di persistenza e query (server-side, SQL)**
  Sul server la logica di accesso ai dati è organizzata in DAO (Jdbc*DAO). I DAO eseguono query SQL che includono operazioni comuni di aggregazione e join per ricavare informazioni come valutazioni medie, conteggi o dettagli correlati. Nei DAO si osservano idiomi come:
  - uso di try-with-resources per la gestione sicura della connessione JDBC;
  - costruzione e parsing di risultati in strutture come List<Map<String,Object>> o mapping in DTO;
  - pattern get-or-create (es. ensureLibreriaId) per ottenere o creare risorse correlate nel DB. Riferimenti: ***JdbcUtentiDAO.java***, ***JdbcCercaLibriDAO.java***, ***JdbcConsigliDAO.java***, ecc..
- **Query e aggregazioni tipiche (DBMS)**
  Le query server utilizzano aggregazioni SQL (COUNT, AVG), raggruppamenti (GROUP BY) e join per ottenere statistiche sulle valutazioni e per generare insiemi di risultati utili ai servizi di suggerimento. Le DDL e gli script di creazione tabelle, in ***CreateDatabaseAndTablesBR.java*** (e relativo codice di creazione DB), definiscono gli indici e le chiavi che supportano queste operazioni; la scelta di indici sulle colonne di join e sulle colonne usate nelle WHERE è cruciale per le prestazioni delle query di ricerca e aggregazione.
- **Algoritmi grafici e di animazione (UI)**
  Per l'effetto grafico di sfondo è presente un semplice motore di particelle che mantiene una lista di particelle e le aggiorna in un loop di animazione. La struttura usa una ArrayList<Particle> e

ciclo di update/draw, visibile in ***ParticleAnimation.java.*** Questo algoritmo è pensato per essere leggero e arrestabile al momento della chiusura della finestra per evitare memory leak.

5. Gestione della concorrenza

La gestione della concorrenza nel server è progettata in modo semplice ma solido: le chiamate RMI vengono instradate ai service che delegano la persistenza ai DAO, i quali aprono e chiudono autonomamente una Connection per ogni operazione usando il metodo *openNewConnection()* (es. ***JdbcConsigliDAO.java***). All'avvio l'applicazione configura i parametri di accesso in ***ServerMain.java*** e verifica la raggiungibilità del database; nelle fasi critiche di inizializzazione e spegnimento la classe ***DBConnectionSingleton*** utilizza blocchi *synchronized* e un metodo *shutdown()* sincronizzato per serializzare l'accesso alle risorse condivise, garantendo avvio/stop sicuri. Questo modello mantiene il codice server leggero e concentrato sulla logica di business, affidando al motore del database le garanzie transazionali e di concorrenza necessarie alle singole istruzioni eseguite via JDBC.

6. Strumenti, librerie e linguaggi utilizzati

Per lo sviluppo dell'applicazione "BookRecommender" e i suoi artefatti sono stati utilizzati i seguenti strumenti:

- JDK SE 17
- IDE: Visual Studio Code
- Apache Maven
- Script di utilità Windows per setup/avvio
- Packaging/assembly: JAR eseguibili prodotti con plugin Maven
- Git e Github per il versioning dell'applicazione e la gestione delle varie branch dei membri del gruppo;

I linguaggi utilizzati sono:

- Java per la realizzazione effettiva dell'applicazione, sia client che server;
- FXML per le view JavaFX
- CSS per lo styling dei componenti della GUI;
- SQL per la definizione e interrogazione del database
- Batch (.bat) per script Windows

Librerie e plugin

- JavaFX (org.openjfx: javafx-controls, javafx-fxml, javafx-graphics, javafx-base), versione 17.0.2
- PostgreSQL JDBC Driver (org.postgresql:postgresql), versione 42.7.3
- Log4j2 (org.apache.logging.log4j:log4j-core e log4j-api), versione 2.20.0
- JFoenix (componenti Material Design per JavaFX), versione 9.0.10
- FontAwesomeFX (de.jensd:fontawesomefx-fontawesome), versione 4.7.0-9.1.2
- Maven plugins principali:
    - maven-compiler-plugin, versione 3.11.0 (configurato per Java 17).
    - maven-shade-plugin, versione 3.4.1
    - maven-assembly-plugin, versione 3.7.0

7. Limiti dell’applicazione e conclusioni

Il progetto nasce con finalità didattiche ma con l’ambizione di avvicinarsi a scelte architetturali usate in contesti più strutturati; per questo motivo offre una base chiara e facilmente estendibile.
Esistono però alcune aree in cui, volendo evolverlo ulteriormente, si possono ottenere benefici significativi: la gestione delle connessioni al database è centralizzata tramite DBConnectionSingleton e oggi fa affidamento su connessioni create per operazione, una scelta semplice e robusta che può essere potenziata con un pool per scenari con carico elevato; il client contiene porzioni di codice con un lieve debito tecnico, risultato naturale della collaborazione fra più sottoteam. Inoltre l’interfaccia e i testi sono pensati principalmente in italiano.

Nel complesso il lavoro ha raggiunto gli obiettivi formativi prefissati e fornisce un’ottima base per interventi incrementali e non invasivi (connection pooling) che ne miglioreranno scalabilità e manutenzione senza stravolgere l’architettura.