BOOK RECOMMENDER

Manuale tecnico

Università degli Studi dell'Insubria – Varese
Progetto Laboratorio B: Book Recommender

Sviluppato da:

- *Emanuele Contini, matricola 756441*
- *Emanuele Gobessi, matricola 757599*
- *Diego Guidi, matricola 758420*
- *Nicola Churchi, matricola 757786*
- *Mirko Gurzau, matricola 757925*

01/2026
Versione documento 1.0

Indice

Introduzione:

Introduzione generale:

Book Recommender è un programma sviluppato come progetto per il Laboratorio Interdisciplinare presso l'Università degli Studi dell'Insubria per l'a.a. 2025/2026. Il backend è stato scritto interamente in Java, mentre il frontend XML, CSS e Java. Sviluppato e testato interamente nell'IDE Visual Studio Code, e in ambienti Windows 10 e Windows 11 e Ubuntu 24.04 ; pertanto, non è possibile garantire il corretto funzionamento in ambienti MacOs, altre distribuzioni di Linux o diversi.

Librerie esterne utilizzate:

Backend

JUnit:

Framework per test unitari.

spring-boot-starter-web:

Questo starter fornisce tutto il necessario per creare applicazioni web e REST API con Spring Boot. Include Spring MVC, un server embedded (di solito Tomcat) e strumenti per gestire richieste HTTP, controller, JSON e routing.

spring-boot-starter-data-jpa:

È lo starter per la persistenza dei dati usando JPA (Java Persistence API). Permette di interagire con il database tramite entità, repository e query astratte, riducendo drasticamente il codice SQL manuale. Sotto il cofano usa spesso Hibernate.

org.postgresql:postgresql:

È il driver JDBC per PostgreSQL. Consente a un'applicazione Java di connettersi e comunicare con un database PostgreSQL, traducendo le chiamate Java in operazioni SQL comprensibili dal DB.

spring-boot-devtools:

Fornisce strumenti per migliorare la produttività in sviluppo. Il più noto è il reload automatico dell'applicazione quando il codice cambia, evitando riavvii manuali. Include anche piccoli miglioramenti per il debugging.

org.projectlombok:lombok:

Lombok riduce il boilerplate code usando annotazioni. Genera automaticamente getter, setter, costruttori, equals, hashCode, ecc. a compile-time, rendendo il codice più pulito e leggibile senza perdere funzionalità.

io.github.cdimascio:dotenv-java:

Permette di caricare variabili di configurazione da un file .env. È utile per separare configurazione e codice, specialmente per password, chiavi

API e parametri sensibili, seguendo le buone pratiche delle app moderne.

springdoc-openapi-starter-webmvc-ui:

Genera automaticamente la documentazione OpenAPI (Swagger) delle REST API. Analizza controller e DTO per produrre una documentazione interattiva accessibile via browser, utilissima sia per sviluppo che per testing delle API.

Frontend

JUnit:

Framework per test unitari.

org.openjfx:javafx-controls:

Contiene i componenti grafici standard di JavaFX: pulsanti, tabelle, liste, menu, campi di testo, slider, ecc. È la base per costruire l'interfaccia utente vera e propria di un'app desktop JavaFX.

org.openjfx:javafx-fxml:

Serve a usare FXML, un linguaggio XML per definire l'interfaccia grafica in modo dichiarativo. Permette di separare layout (vista) e logica (controller Java), rendendo il codice più pulito e mantenibile.

com.fasterxml.jackson.core:jackson-databind:

Jackson Databind gestisce la serializzazione e deserializzazione JSON. Converte oggetti Java in JSON e viceversa. Nel front è spesso usato per consumare API REST, ad esempio per trasformare le risposte del backend in oggetti utilizzabili dall'interfaccia.

io.github.cdimascio:dotenv-java:

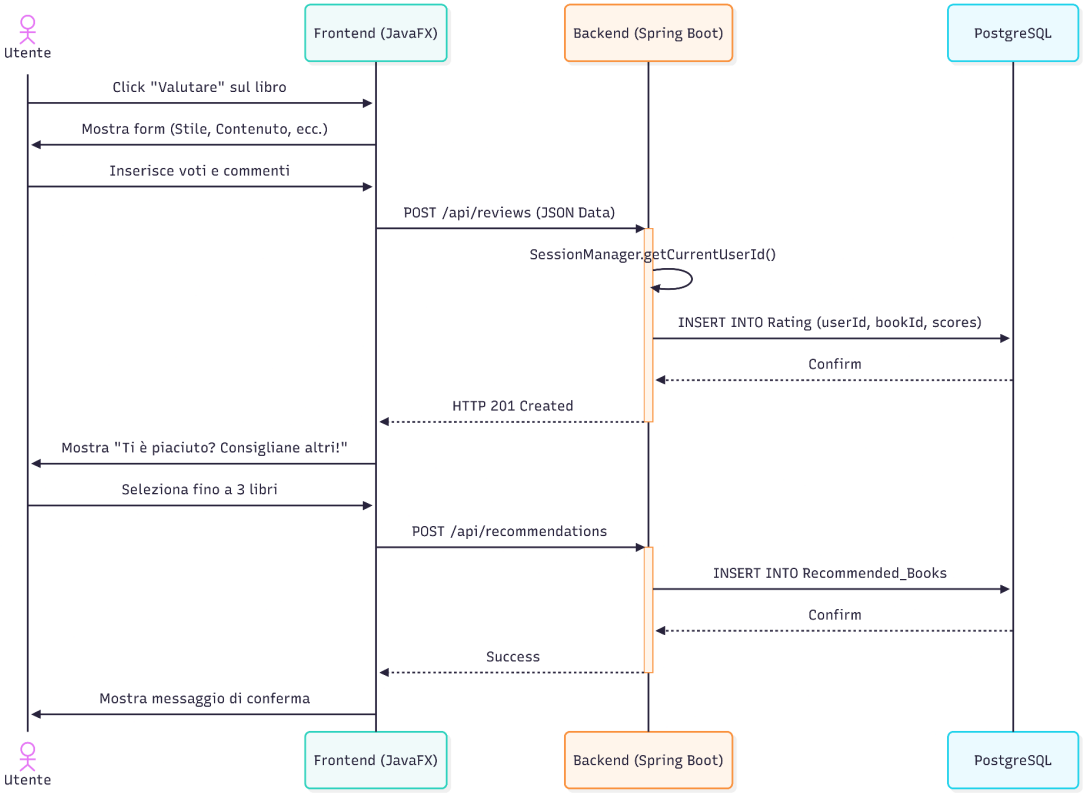
Nel front viene usata per lo stesso motivo: caricare variabili d'ambiente da .env (URL del backend, porte, token, modalità di esecuzione). La duplicazione ha senso perché backend e frontend sono due applicazioni separate, ciascuna con la propria configurazione.

com.google.inject:guice Guice:

è un framework di Dependency Injection. Gestisce la creazione e l'iniezione delle dipendenze tra classi, riducendo l'accoppiamento. Nel front JavaFX è utile per collegare controller, servizi e modelli in modo pulito, soprattutto quando l'app cresce di complessità.

Architettura del programma

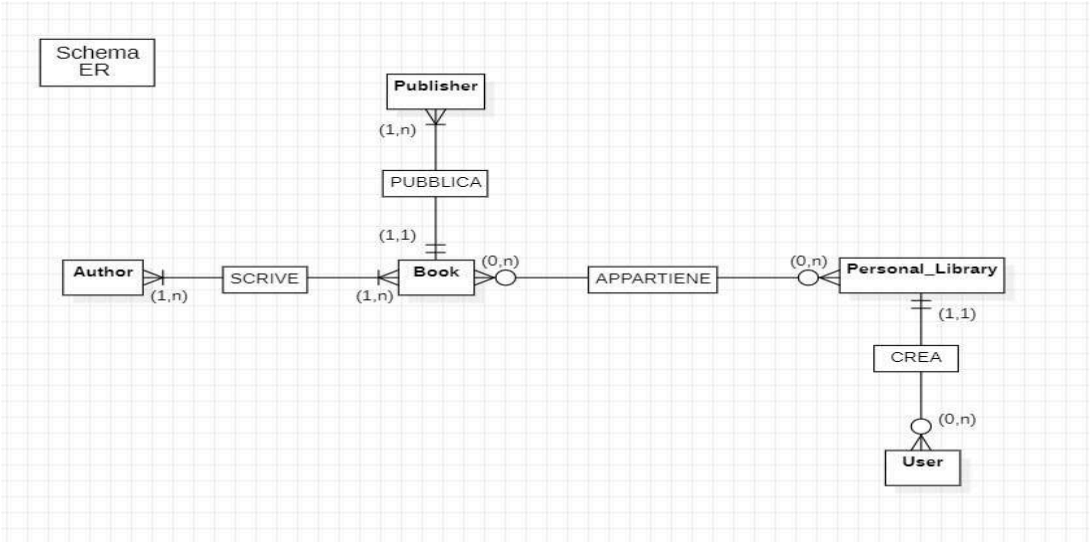
Flusso generale del programma:



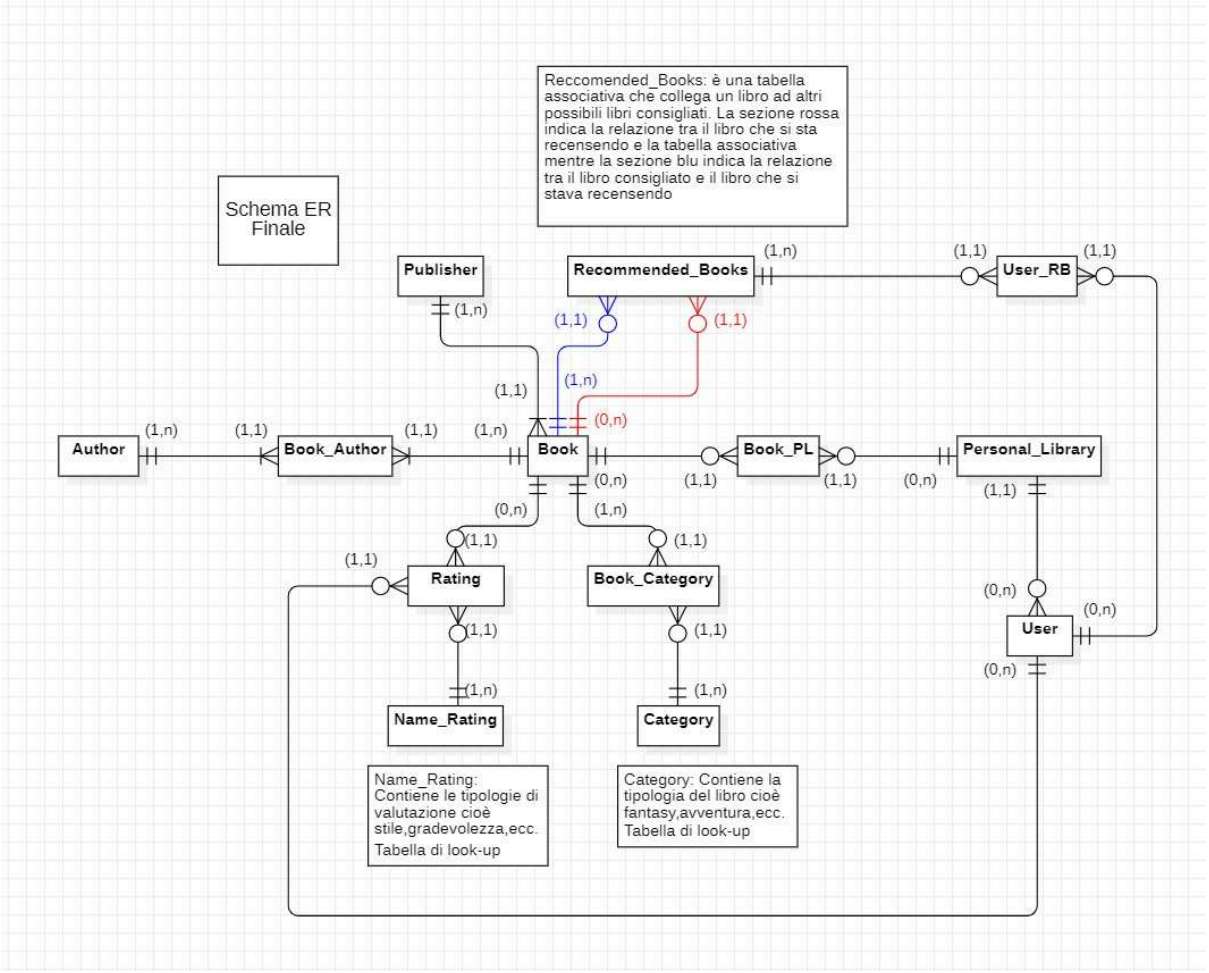
Backend:

L'architettura del backend prevede un database relazionale basato su PostgreSQL e una API in Java.

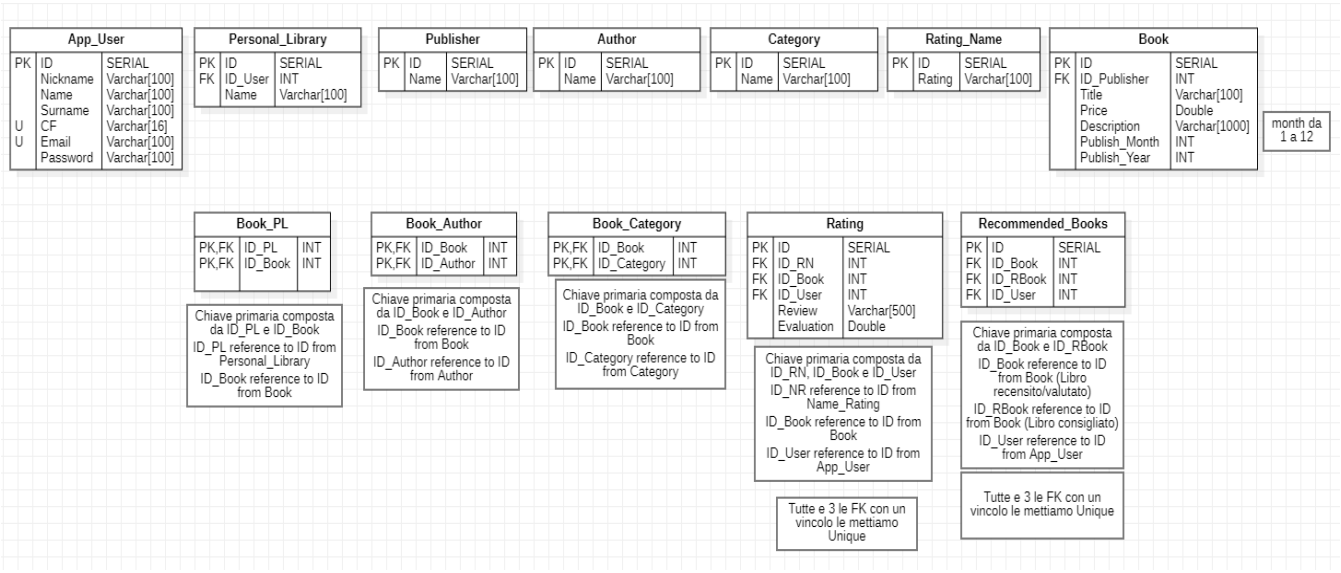
Struttura del database:



lo schema del database prevede cinque entità fondamentali: Author, Book, Publisher, Personal_Library e User; ogni enità è in relazione con un'altra in base ai criteri sopra riportati. Tali relazioni sono state implementate come di seguito:



Questo è il schema su cui si basa il database, a cui sono stati assegnati, per ogni tabella, i seguenti campi:



Architettura del Server:

Per il server è stato utilizzato il pattern architetturale MVC; le cartelle principali sono infatti le seguenti:

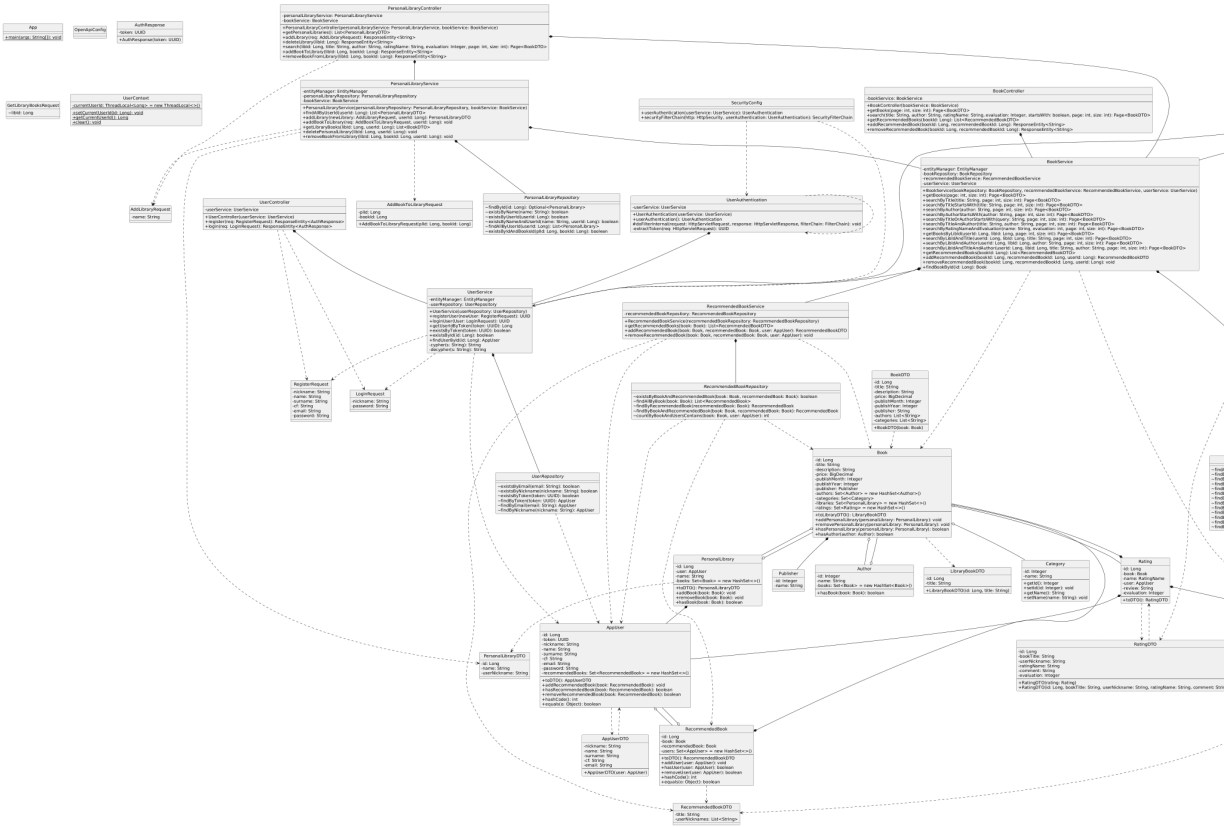
- **Controllers:** I controller gestiscono le richieste HTTP provenienti dai client. Espongono gli endpoint dell'API, validano i dati in ingresso e delegano la logica applicativa ai service. Si occupano quindi del livello di presentazione, senza contenere logica di business.
- **Services:** I service contengono la logica di business dell'applicazione. Coordinano più repository, applicano regole, controlli e trasformazioni dei dati. Fanno da ponte tra controller e livello di persistenza, mantenendo una struttura pulita e testabile.
- **Models:** le classi model rappresentano le entità del database in forma di oggetto Java.
- **Repositories:** le classi repositories servono solo ad interagire col database tramite queries.

Altre cartelle importanti:

- **DTO:** vengono usati per verificare la validità dei dati mandati dall'utente rispetto a controlli personalizzati, oppure per formattare in maniera corretta oggetti inviati al client, per evitare errori di formato o esposizione di dati sensibili.

- **Middleware:** sono classi speciali che vengono chiamate prima che una richiesta raggiunga il relativo controller, per applicare dei controlli aggiuntivi di sicurezza e autenticazione.
- **Exceptions:** contiene una classe principale che intercetta le eccezioni che si presentano all'interno del programma per stamparne una versione formattata, e per inviarla correttamente al client in forma di risposta HTTP; la cartella contiene anche alcune eccezioni personalizzate.
- **Utils/Config:** contengono le impostazioni e il contesto relativi al programma.

Class diagramm del backend



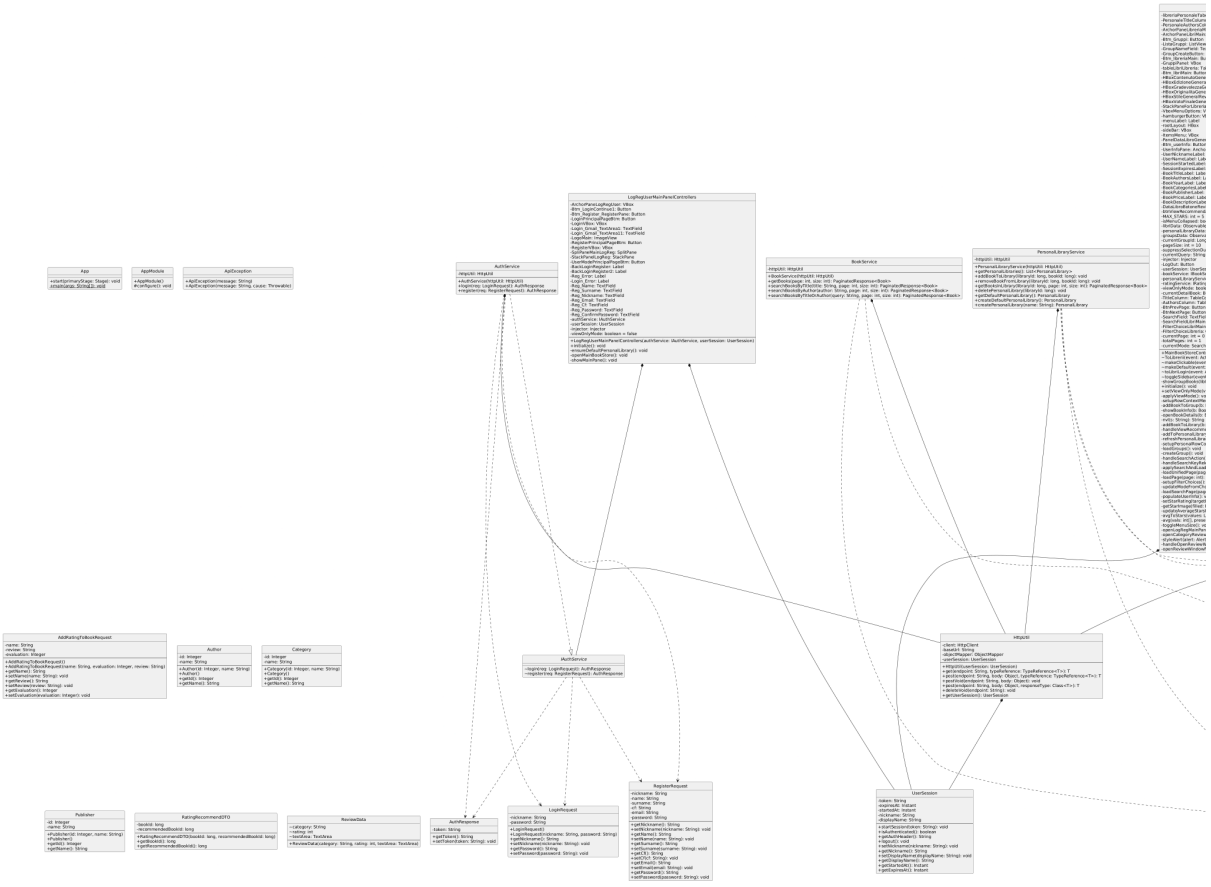
Architettura del Frontend:

è diviso nei seguenti packages:

- **Models:** le classi model rappresentano le entità del database in forma di oggetto Java.

- **Services:** I service contengono la logica di business dell'applicazione. Coordinano più repository, applicano regole, controlli e trasformazioni dei dati. Fanno da ponte tra controller e livello di persistenza, mantenendo una struttura pulita e testabile.
- **Interfaces:** contiene interfacce che definiscono la logica del frontend
- **Exceptions:** contiene alcune eccezioni personalizzate
- **Pages:** contiene i controllers per il frontend e le classi Page, che servono per formattare i dati ricevuti del database
- **DI:** il suo scopo è associare interfacce alle loro implementazioni
- **Utils:** contiene classi concrete utilizzate come implementazioni dai iniettare nei servizi

Class diagram frontend:



Strutture dati e algoritmi

Strutture:

- **Database Relazionale:** Sistema di gestione dati basato sul modello relazionale nello specifico è stato utilizzato PostgreSQL. La coerenza e l'integrità dei dati sono garantite da vincoli (come Primary e Foreign Key). L'interazione avviene tramite metodi che eseguono query indirette.
- **Liste di oggetti:** Rappresentano sequenze ordinate di istanze trasmesse dal server al client tramite protocollo HTTP. Sono il risultato della mappatura di righe del database in oggetti di business (tramite DTO - Data Transfer Objects) per essere facilmente iterabili dalle interfacce frontend.
- **Hash Sets:** Struttura dati utilizzata nei modelli per rappresentare relazioni uno-a-molti o molti-a-molti. L'uso dell'Hash Set garantisce l'unicità degli elementi correlati e offre prestazioni ottimali (O(1) in media e O(n) nei rari casi peggiori) per le operazioni di inserimento, rimozione e verifica dell'esistenza, evitando duplicati logici nelle associazioni tra entità.
- **Pages:** Tecnica di formattazione della risposta che suddivide un set di dati voluminoso in segmenti più piccoli. Questo approccio riduce il carico sulla rete, ottimizza l'utilizzo della memoria del client e migliora le performance del server limitando il numero di record estratti ad ogni chiamata.

Algoritmi:

- **Algoritmo di cifratura/decifratura:** semplice algoritmo di cifratura basato sul cifrario di Cesare modulare.

Nessun altro algoritmo particolare è stato utilizzato; la maggior parte della logica applicativa consiste in controlli e formattazione del dato.

Design Patterns

I principali utilizzati sono:

- **MVC Layered:** Design pattern architetturale che suddivide il funzionamento del programma in quattro strati cioè model, control, view e service. Ognuno di questi si trova a livelli differenti e le mansioni

svolte da ciascuno non vengono mai mescolati per garantire coerenza e pulizia del codice,

- **DTO pattern:** Il programma utilizza Data Transfer Objects (DTO), oggetti semplici, privi di logica di business, utilizzati per trasportare dati tra i livelli di un'applicazione o attraverso la rete. Il loro scopo è controllare i dati esposti e ottimizzare la serializzazione. I DTO contengono esclusivamente campi, costruttori, metodi di accesso e controlli di validità basici, senza comportamenti applicativi.
- **Singleton:** La classe UserContext.java è stata implementata come singleton per via del campo sensibile che espone l'utente visto che contiene il suo ID al momento della richiesta.

```
public class UserContext {
/* Il campo è private static final perché
      deve esistere un'unica istanza condivisa in tutta
      l'applicazione, non modificabile
      e accessibile solo tramite i metodi della classe.
*/
private static final ThreadLocal<Long>
currentUserId = new ThreadLocal<>();
      public static void setCurrentUserId(Long id){
       currentUserId.set(id);
      }
      public static Long getCurrentUserId() {
       return currentUserId.get();
      }
      public static void clear() {
       currentUserId.remove();
      }
}
```

File Aggiuntivi

Backend

lab-lib-restapi\

- **pom.xml:** file project object model che contiene metadati del progetto, dipendenze necessarie, plugins e la configurazione di Spring Boot.

src\main\resources\

- **.env:** contiene le variabili di ambiente per la connessione col database.
- **applications.properties:** contiene la configurazioen dell'applicazione, in particolare inizzializza drivers e connessioni e specifica la porta del server

Frontend

lab-lib-frontend\

- **pom.xml:** file project object model che contiene metadati del progetto, dipendenze necessarie, plugins.

src\main\resources\

- **.env:** contiene le variabili di ambiente per la connessione col database.
- **Img\:** files .png usati come icone del programma
- **com\lab_lib\frontend**
    - Css\: files .css che definiscono lo stile dei componenti grafici
    - Pages\: files .fxml che definiscono la struttura dei componenti grafici

Sitografia

- https://www.postgresql.org/docs/18/index.html
- https://docs.oracle.com/javase/8/docs/api/overview-summary.html
- https://openjfx.io/
- https://spring.io/projects/spring-boot
- https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html
- https://hibernate.org/
- https://maven.apache.org/