UNIVERSITÀ DEGLI STUDI DI FIRENZE

Dipartimento di Ingegneria

Ingegneria Software

Professore:

Enrico Vicario

Studente:

Francesko Blushi

Mat. 7047358

ANNO ACCADEMICO 2024/2025

Indice

1 Introduzione generale

1.1 Progetto

Il progetto scelto:

- Sistema di Gestione Prestiti per Biblioteca. Il presente documento descrive lo sviluppo di un sistema software per la gestione dei prestiti di libri in una biblioteca. Il sistema consente agli utenti di iscriversi, prendere in prestito e restituire libri, mentre gli amministratori possono gestire il catalogo dei libri e gli utenti iscritti. Il software e' basato su architettura client-server, con database relazionale gestito tramite JPA. L'obiettivo principale del progetto e' sviluppare un sistema efficiente e intuitivo per la gestione di una biblioteca, automatizzando le operazioni di prestito e restituzione dei libri. In particolare il sistema offre:
- Gestione del catalogo dei libri
- Gestione degli utenti registrati.
- Controllo sui prestiti e restituzioni dei libri.
- Differenziazione tra utenti standard e amministratori.

1.2 Struttura Generale del Progetto

```
Library/
 .idea/
 src/
    main/
       java/
           org/
               Library/
                   Controller/
                   DaoModels/
                   DTO/
                   Models/
                   ValidationUtils/
    test/
        java/
           Connection/
           TestDiUnita/
           TestDomainModel/
           TestFunzionali/
```

Il progetto separa la logica dell'applicazione in tre componenti principali.

Models(Entita)

- Situata nella cartella Models.
- Definisce gli oggetti principali della applicazione come Utente,Admin,User,Autore,Books,Loans.
- Uso JPA .

DAO(Data Access Objects)

- Situati nella cartella DaoModels
- Definiscono gli oggetti come UtenteDao,AdminDao,UserDao,AutoreDao,BooksDao,LoansDao
- Si occupano di interagire con il database effettuando query per CRUD.
- Abbiamo creato due database, uno per l'app, contenente i dati degli utenti veri e propri e uno per la fase di test, per evitare che le query eseguite modificano dati reali.

Controllers

- Situati nella cartella Controller

- Definiscono le Classi come AccessContorller,AdminController,PassordController,RegisterController,UtenteController

Altri Package

- DTO (Data Transfer Object)

  Composta Classe LoanBook,LibroAutore e ResponseDTO dove Loan viene usato per il Join implementato nella classe LoanDAO e la ResponseDTO viene usata dal package Controllers per restituire in modo più efficace i dati delle richieste effettuate al backend.

- ValidationsUtils

  I file contenuti nel package ValidationUtils forniscono una serie di metodi di utilità dedicati alla validazione dei dati inseriti dagli utenti e dagli amministratori all'interno dell'applicazione.

  L'obiettivo principale di questi metodi è verificare la correttezza e la completezza dei dati ricevuti dai Controller, prima che vengano elaborati o salvati nel sistema. Ogni classe è specializzata in un particolare contesto operativo.

1.3 Tecnologie Utilizzate

- Java - Linguaggio Principale.
- JPA (Java Persistence API) - per il database.
- PostgreSql - Database relazionale.
- Maven - Gestione dipendenze (pom.xml)
- Il seguente testo e' stato scritti su Latex usando Overleaf

1.4 Diagramma dei Package

Il seguente diagramma rappresenta l'architettura complessiva del progetto, suddivisa in pacchetti principali.

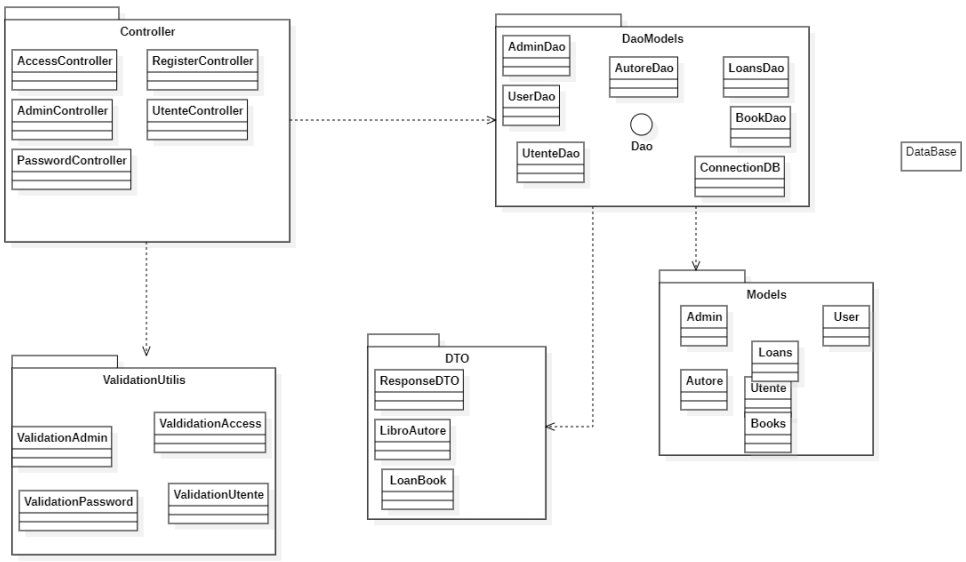


Figura 1: Diagramma UML degli Package

Le frecce tratteggiate rappresentano relazioni di `dependency`, indicando che un pacchetto fa uso delle classi presenti in un altro pacchetto.

2 Progettazione

2.1 Use Case Diagram

2.1.1 Descrizione dello Use Case Diagram

Il diagramma dei casi d'uso rappresenta un breakdown funzionale del sistema, mostrando le principali interazioni che ciascun attore può avere con il sistema. Gli attori identificati sono:**Admin**, che rappresenta l'amministratore della biblioteca **Utente**, che rappresenta un utilizzatore finale del sistema

Ogni attore ha accesso a un insieme specifico di funzionalità:

- **Admin**: può effettuare il login, aggiungere nuovi libri al catalogo, eliminare libri esistenti e gestire gli utenti (es. cancellazione).
- **Utente**: può autenticarsi nel sistema, visualizzare i libri disponibili, prendere un libro in prestito, restituirlo e consultare lo storico dei prestiti.

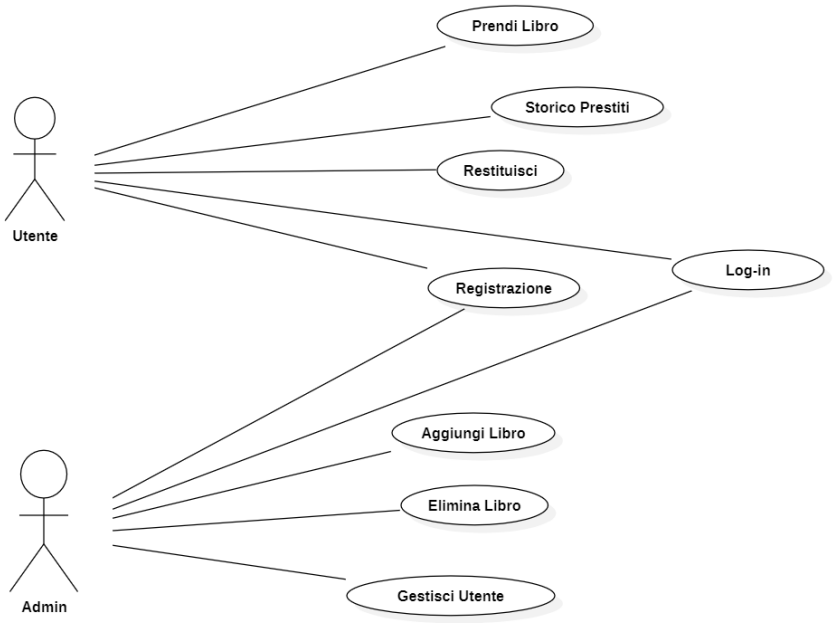


Figura 2: Diagramma UML dei Casi d'Uso per il sistema di gestione della biblioteca.

2.2 Use Case Template

Use Case #1	**Accedi al Sistema (Log-in)**
Brief Description	L'utente accede al sistema tramite le proprie credenziali.
Level	User Goal
Actors	Admin, Utente
Pre-Conditions	L'utente deve essere registrato
Basic Flow	1. L'utente seleziona se accedere come Admin o Utente (Home View). 2. L'utente inserisce le proprie credenziali. 3. L'utente invia le proprie credenziali. 4. Il sistema verifica le credenziali. 5. Il sistema autentica l'utente Test #1, #10, #11, Implementazione
Alternative Flow	3a. Se le credenziali sono errate o l'utente non è registrato, il sistema restituisce un messaggio di errore e permette di riprovare (Access Test Controller).
Post-Conditions	L'utente è autenticato nel sistema e ha accesso.

Tabella 1: Prima pagina

Use Case #2	**Aggiungi/Elimina Libro**
Brief Description	Permettere all'Admin di aggiungere o eliminare libri dal sistema..
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin deve avere fatto Log-in. .
Basic Flow	1. L'admin compila tutti i campi nel caso in cui si aggiunge un libro.(Add/Remove) 2. L'admin inserisce solo l'isbn del libro se vuole cancellarlo. 3. Il sitema elimina il libro solo nei casi in cui tutte le copie disponibili sono in bibloteca. 4. Se vengono compilati tutti i campi il libro viene aggiunto correttamente. (Test #4 #5 (Implementazione)) .
Alternative Flow	3a. Se i campi non vengono compilati tutti, il sistema restituisce un messaggio di errore e permette di riprovare (Admin Test Controller).
Post-Conditions	Un messaggio di conferma .

Tabella 2: Aggiungi o Elimina Libri

Use Case #3	**Libri in Prestito**
Brief Description	L'utente ha la possibilita di controllare lo storico dei libri presi in prestiti.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente ha effettuato il login. .
Basic Flow	1. L'utente controllo lo storico dei libri presi in prestito.(Lista Libri in Prestito) (Implementazione 2. L'utente controllo la scadenza del prestito.
Alternative Flow	Se non ci sono libri in prestito l'elenco dei libro sara vuoto. (Test #6).(Utente Test Controller)
Post-Conditions	Elenco dei libri in Prestito .

Tabella 3: Controllo storico Libri in Prestito

Use Case #4	**Prendi/Retituisci Libro**
Brief Description	L'utente ha la possibilita di prendere o restituire un libro.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente deve avere fatto Log-in. .
Basic Flow	1. L'utente scrive il titolo del libro nel caso di restituzione basta solo scrivere isbn del libro.(Utente view) (Implementazione) 2. L'utente fa search. 3. L'utente ha la possibilita di prendere il libro in prestito con tre possibili periodi. 4. L'utente conferma il prestito del libro. 5. L'utente puo restituire il libro.(Test #7).
Alternative Flow	3a. Se il libro non è disponibile, il sistema informa l'utente. (Utente Test Controller).
Post-Conditions	Un messaggio di conferma nel caso in cui il libro sia disponibile .

Tabella 4: Prendi o restituisci il libro

Use Case #5	**Gestione Utenti**
Brief Description	L'admin ha la possibilita di controllare ed eliminare un utente.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin deve avere fatto Log-in.
Basic Flow	1. L'admin scrive il codice della carta id del utente.(Gestione utenti Implementazione) 2. L'admin fa search. 3. L'admin ha la possibilita di eliminare l'utente.(Test #8).
Alternative Flow	3a. Se il codice carta è errato, nessun utente viene trovato. (Admin Test Controller).
Post-Conditions	Un messaggio di conferma in caso di eliminazione corretta. .

Tabella 5: Elimina Utente

Use Case #6	**Registrazione view**
Brief Description	Un user ha la possibilita di iscriversi.
Level	User Goal
Actors	Admin, Utente
Pre-Conditions	L'utente deve scegliere il tipo di registrazione..
Basic Flow	1. L'user sceglie se e' un admin o un utente. (Registrazione view Implementazione) 2. Per essere iscritto come admin deve conoscere il codice id della bibloteca. 3. L'user deve compilare tutti i campi, e il username deve essere unico 4. L'user si iscrive..(Test #2 #3)
Alternative Flow	3a. Se alcuni campi sono vuoti o duplicati (username già esistente), la registrazione fallisce. (Register Test Controller).
Post-Conditions	Creazione di un nuovo account utente o admin.. .

Tabella 6: Iscrizione User

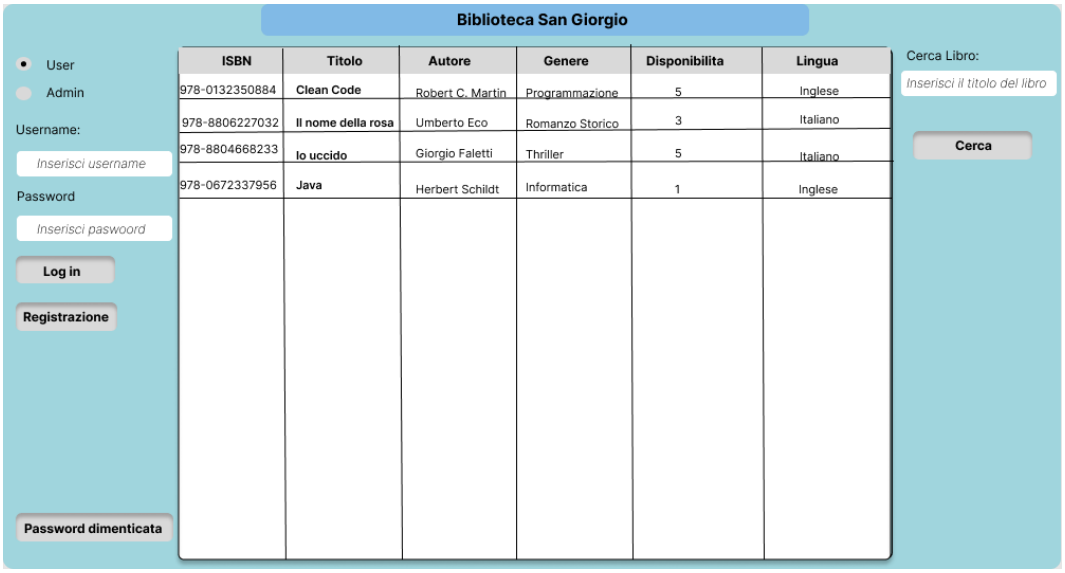
Use Case #7	**Recupera password**
Brief Description	L'user ha la possibilita di cambiare il password.
Level	User Goal
Actors	Admin,Utente
Pre-Conditions	L'user deve essere gia registrato. .
Basic Flow	1. L'user scrive il codice della carta id del utente.(Cambia Password Test #9) 2. L'user deve compilare tutti i campio 3. L'utente ha la possibilita di scriver la nuova password.().
Alternative Flow	3a. Se il codice della carta id e i altri campi non coincidano con l'user gia iscritto non e' possibile cambiare la password. La password deve rispettare certi requisiti. (Test Password Controll).
Post-Conditions	Un messaggio di conferma in caso di cambio corretta. .

Tabella 7: Nuovo Password

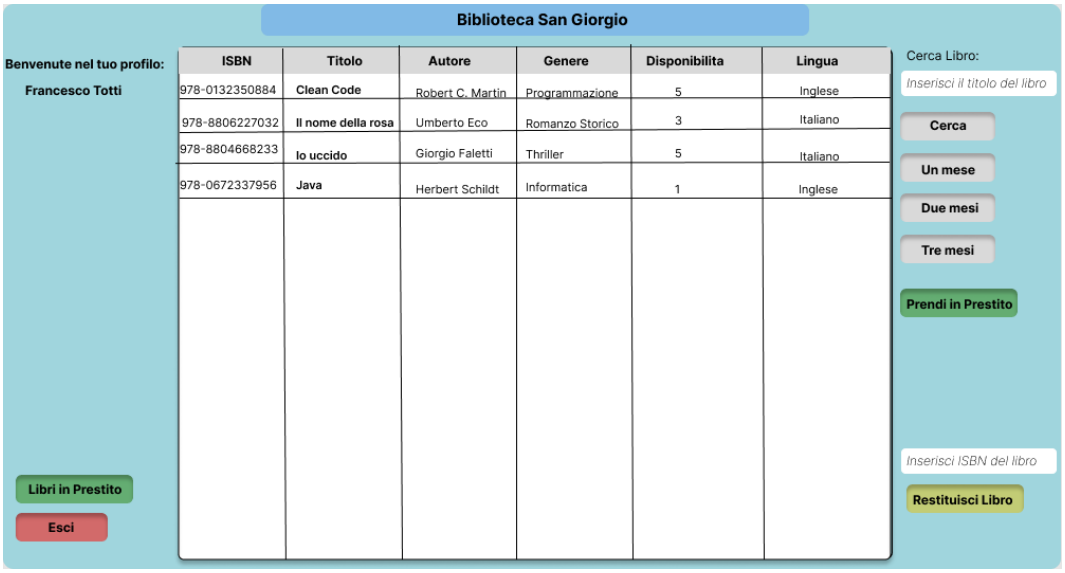
2.3 Mock-ups

- I mock-up presenti nel progetto sono stati realizzati utilizzando le funzionalità base di **Figma**, senza l'utilizzo di template o componenti predefiniti. Ogni schermata è stata progettata da zero, in autonomia, per rappresentare graficamente le principali funzionalità del sistema, come la schermata iniziale, la vista utente, la vista amministratore, la registrazione e la gestione dei libri in prestito. Figma si è rivelato uno strumento efficace per realizzare prototipi chiari e coerenti con i requisiti emersi durante la fase di analisi.

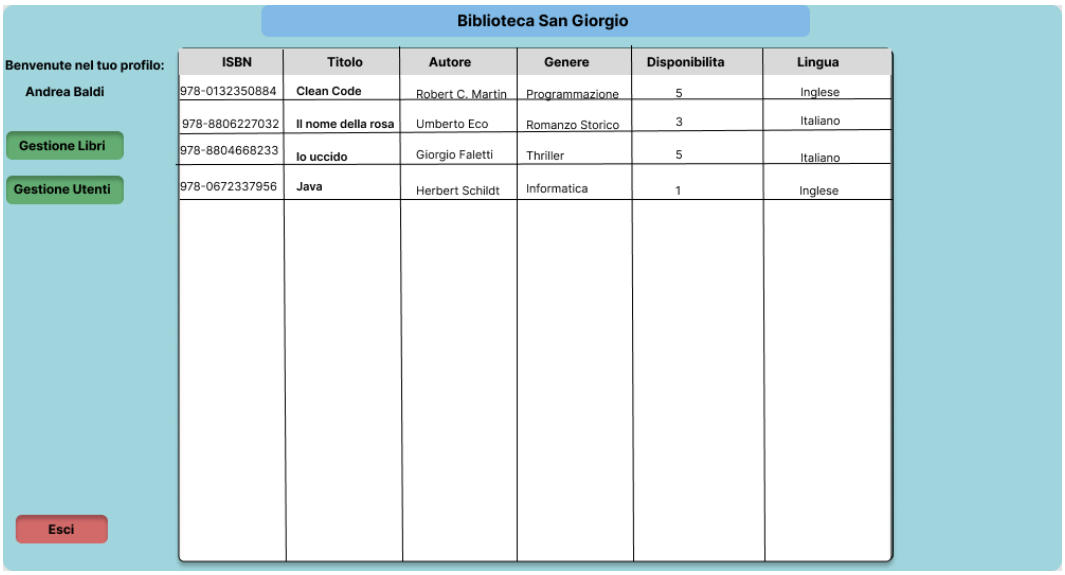
2.3.1 Home view



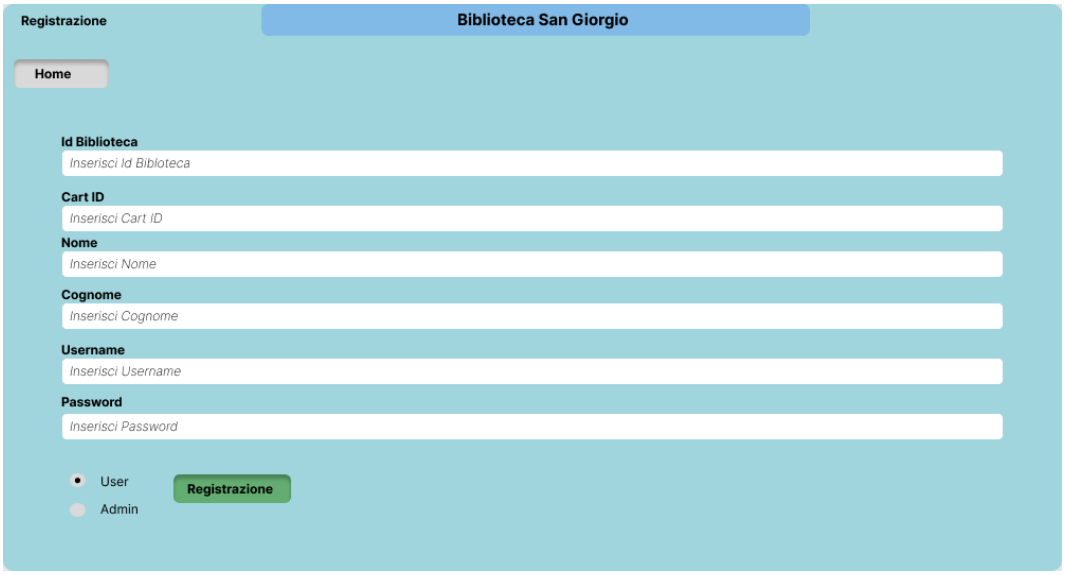
2.3.2 Utente View



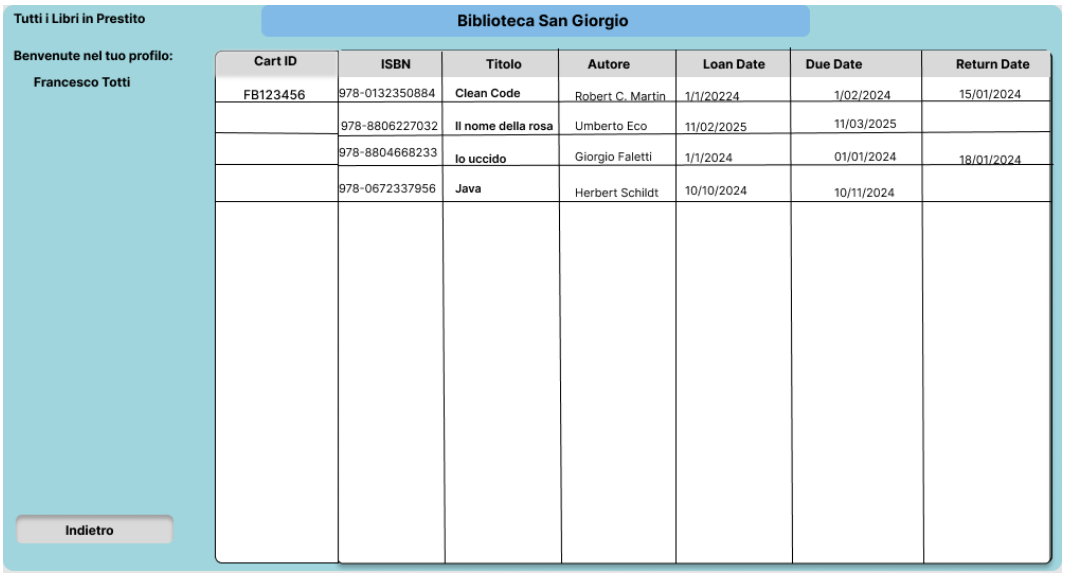
2.3.3 Admin View



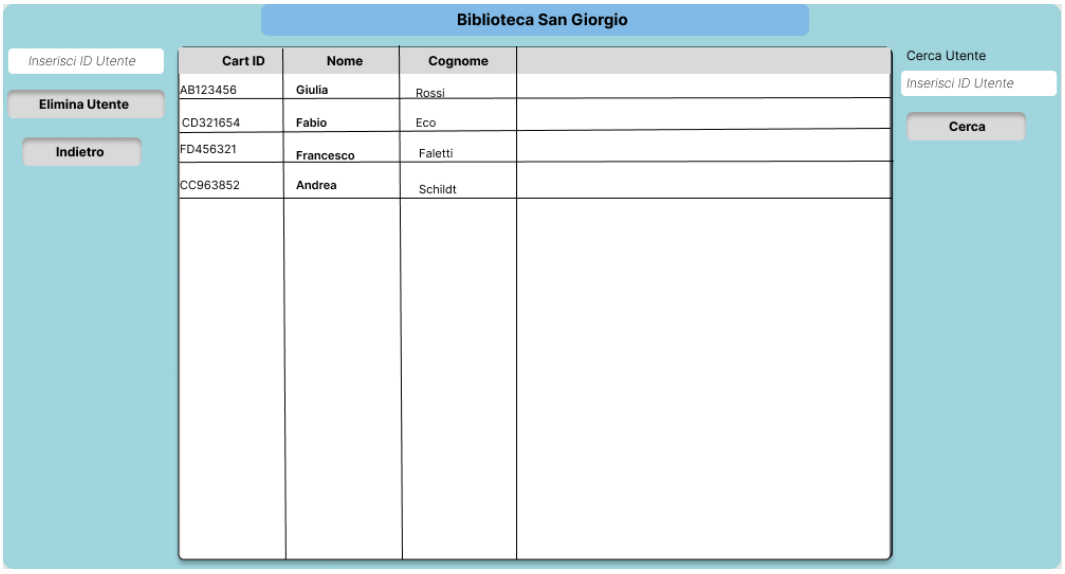
2.3.4 Registrazione View



2.3.5 Libri in Prestito

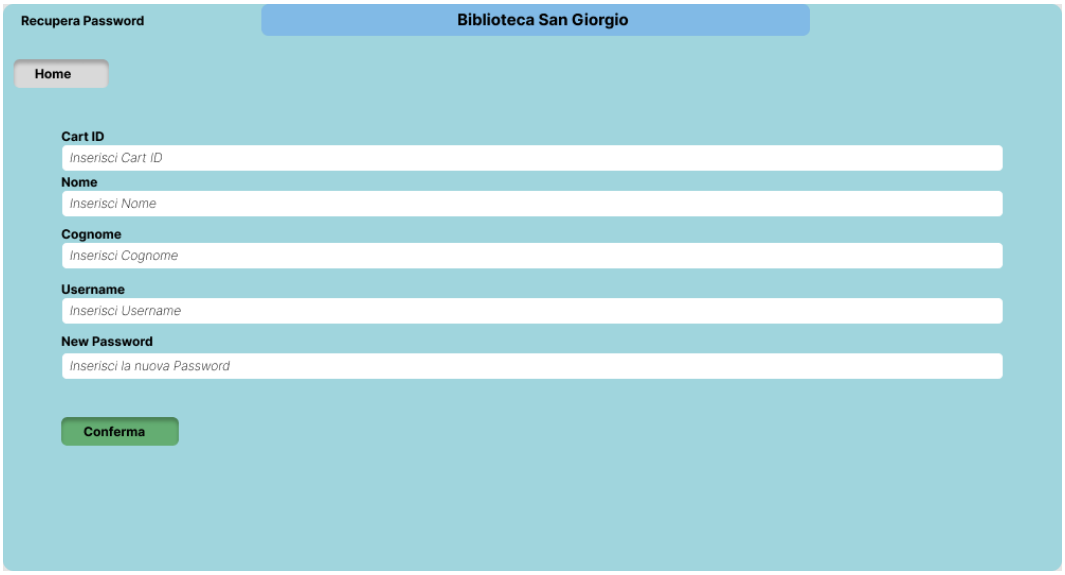


2.3.6 Gestione Utenti



2.3.7 Aggiungi Libro

2.3.8 Recupera Password



2.4 Class Diagram

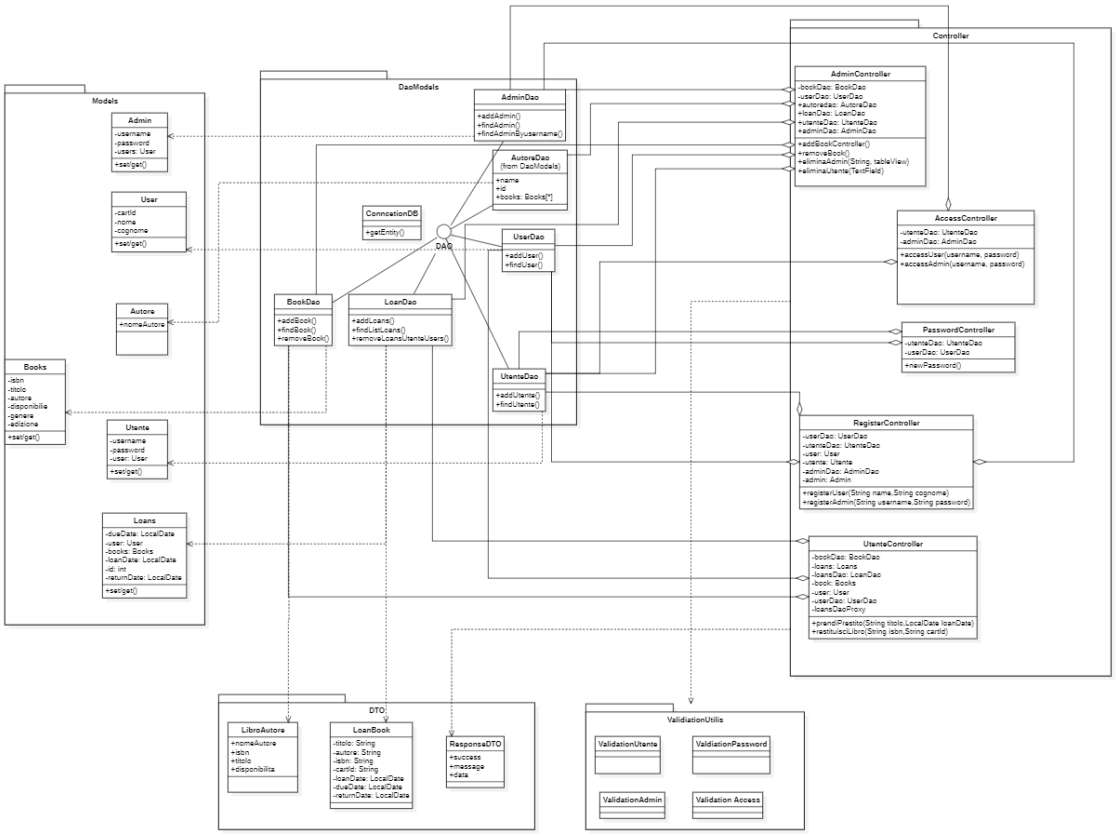


Figura 3: Rappresetazione uml delle relazioni tra le vari classi

2.4.1 Commento Diagramma delle Classi

La Figura 3 rappresenta il diagramma delle classi dell'intero sistema. Tale diagramma mostra in dettaglio le entità principali, le classi DAO, i controller, i DTO e i componenti di validazione, con tutte le relazioni che li legano.

- Il **package Models** contiene le entità persistenti mappate con JPA/Hibernate, come `User`, `Books`, `Loans`, ecc.
- Il **package DaoModels** gestisce l'accesso ai dati e contiene classi DAO che utilizzano `EntityManager` per eseguire operazioni CRUD sulle entità.
- I **Controller** orchestrano il flusso delle operazioni, utilizzando DAO, DTO e metodi di validazione. Ogni Controller è responsabile di un sottoinsieme di funzionalità (es. `RegisterController`, `UtenteController`, ecc.).
- I **DTO** sono usati per il trasferimento sicuro dei dati tra il backend e il client, evitando l'esposizione diretta delle entità.
- Il package **ValidationUtils** contiene classi statiche dedicate alla validazione dei dati in ingresso.

Nonostante la complessità visiva dovuta al numero elevato di classi e legami, il diagramma riflette fedelmente l'organizzazione logica e tecnica dell'architettura progettuale.

2.5 ER Diagram e modello relazionale

Le relazioni tra le entità nel modello ER sono definite come segue:

- **Utente - Users**: Relazione **uno-a-uno** $(1:1)$, in cui esiste un solo utente per user.. Dove in Utente username e' un Primary Key.
- **Admin - Users**: Relazione **uno-a-uno** $(1:1)$, in cui esiste solo un admin per user e in Admin Primary Key e' username.
- **Users - Loans**: Relazione **uno-a-molti** $(1:N)$, in cui ogni user puo avere piu prestiti dove in user cartId fa da Primary Key. .
- **Books - Autore**: Relazione **molti-a-uno** $(N:1)$, in cui un autore può aver scritto piu libri con isbn Primary Key nei Books.
- **Loans - Books**: Relazione **molti-a-uno** $(N:1)$, ogni prestito (*Loans*) (*Books*), ma un libro può essere prestato più volte con Id automatic come Primary Key.
- **Loans - Users**: Relazione **molti-a-uno** $(N:1)$, ogni prestito (*Loans*) è associato a un solo utente (*Users*), ma un utente può avere più prestiti.

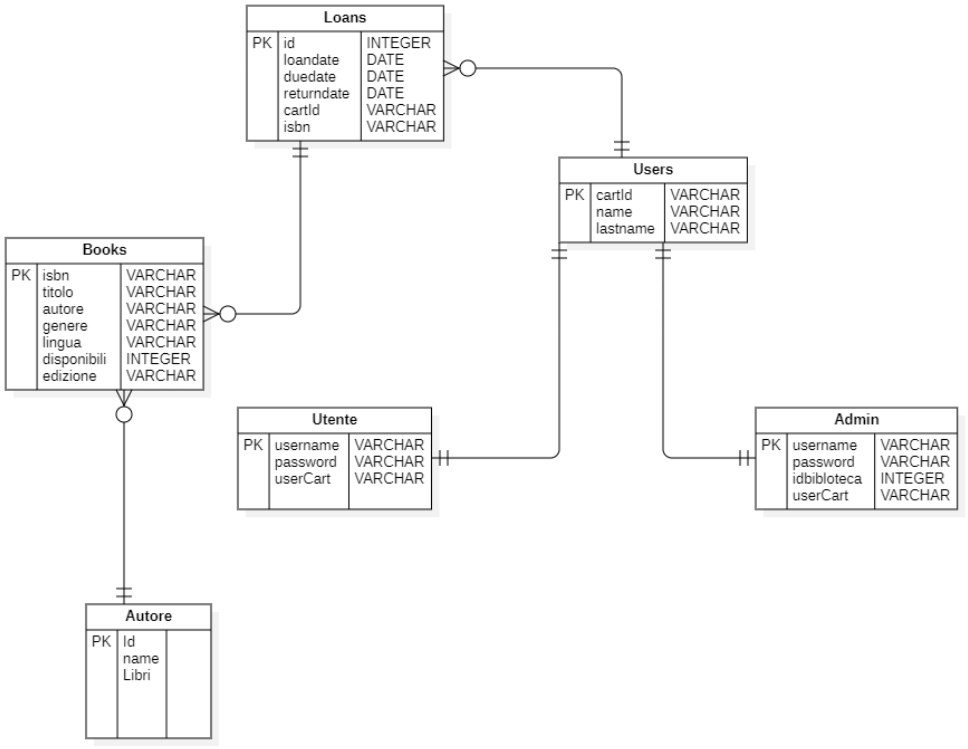


Figura 4: Rappresentazione UML delle relazioni tra le entità del sistema di gestione della biblioteca.

3 Implementazione delle Classi

Il sistema di gestione della biblioteca è stato sviluppato utilizzando il framework JPA con Hibernate per la gestione della persistenza. Di seguito sono descritte le principali classi di Models. Ogni classe di Models e' un Entity e rappresenta una tabella nel database. In seguito vengo descritte semplicimente le tre principali package del progetto,

3.1 Models

3.1.1 Admin

La classe `Admin` rappresenta un amministratore della biblioteca. Ogni amministratore è identificato da uno username, che funge anche da "Primary Key" della tabella, da una password e da un numero identificativo della biblioteca in cui lavora. Lo username è stato scelto come chiave primaria per facilitare la ricerca nel database durante la procedura di login. La classe possiede inoltre un riferimento `@OneToOne` alla classe `Users`, dove la colonna `cartId` di `Admin` è una "Foreign Key" che punta alla "Primary Key" di `Users`. Questa scelta è stata fatta per permettere di collegare i dati anagrafici dell'utente con i dati necessari per accedere ed eventualmente utilizzare il software.

```
@OneToOne(cascade = CascadeType.REMOVE)
@JoinColumn(name = "cartId", referencedColumnName = "cartId", nullable = false)
private Users users;
```

La cancellazione di un `Admin` comporta automaticamente la cancellazione del corrispondente `Users`.

Attributi:

- `username`: Nome utente dell'admin.
- `password`: Password di accesso.
- `idBiblioteca`: Identificativo della biblioteca gestita.
- `users`: (CartId) Riferimento alla classe `Users`.

Relazioni:

- `@OneToOne` con `Users`.

3.1.2 Autore

La classe `Autore` rappresenta la tabella di ogni Autore che ha scritto almeno un libro della bibliotecca. All'interno abbiamo una relazione molto importante di tipo `@OneToMany` con `Books`. Nella quale ogni libro appartiene a un autore, e ogni autore può avere molti libri. Il database mantiene la connessione grazie alla colonna autore di books, che è una foreign key che punta a autore.id.

```
@OneToMany(mappedBy = "autore", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Books> libri;
```

Un autore può avere più libri associati, e la rimozione di un autore elimina anche i suoi libri.

Attributi:

- `id`: Identificativo univoco.
- `nome`: Nome dell'autore.
- `libri`: Lista dei libri scritti.

Relazioni:

- `@OneToMany` con `Books`.

3.1.3 Books

La classe `Books` rappresenta la tabella dei libri che si trova in database. Abbiamo come ”Primary Key” la colonna ”ISBN” che identifica in modo univico il libro. In piu abbiamo una relazione `@ManytoONe` con `Autore`. In cui un Libro puo avere un solo Autore. In piu vale che nullable = false Obbliga che ogni record di Books abbia un autore valido. Questo vincolo vieta anche la cancellazione di un Autore se nella Bibliotecca virutale esistono libri suoi. Qundi per poter cancellare un Autore bisogna che essi non abbia nessun libro in bibliotecca.

Struttura generale dell’entità L’entità è definita con l’annotazione `@Entity`, che indica a Hibernate di generare una tabella corrispondente, mentre `@Table(name = "books")` specifica il nome esatto della tabella nel database.

Listing 1: Books.java

```
@Entity
@Table (name = ”books”)
public class Books {

  @Id
  @Column(name =”isbn”,length = 50,nullable = false)
  private String isbn;

  @Column (name = ”titolo”,length = 250,nullable = false)
  private String title;

  @ManyToOne
  @JoinColumn(name = ”autore”, nullable = false)
  private Autore autore;

  @Column (name = ”genere”, length = 250)
  private String genre;

  @Column(name = ”lingua”, length = 250)
  private String lingua;

  @Column(name = ”disponibili”, length = 250,nullable = false)
  private int disponibili;

  @Column(name = ”edizione”, length = 250)
  private String edizione;
  }
```

Descrizione delle annotazioni principali

- `@Entity`: Indica che la classe è un’entità persistente JPA.
- `@Table(name = "books")`: Mappa l’entità alla tabella chiamata `books` nel database.
- `@Id`: Definisce il campo `isbn` come chiave primaria dell’entità.
- `@Column(...)`: Specifica le caratteristiche di ciascuna colonna nel database, come nome, lunghezza e nullabilità.
- `@ManyToOne`: Definisce una relazione molti-a-uno con l’entità `Autore`. Più libri possono essere associati allo stesso autore.
- `@JoinColumn(name = "autore")`: Specifica il nome della colonna nel database che funge da chiave esterna verso la tabella degli autori.

Queste annotazioni permettono ad Hibernate di generare automaticamente le query SQL necessarie per creare, leggere, aggiornare e cancellare i record associati a questa entità nel database, mantenendo la coerenza tra codice Java e schema relazionale.

3.1.4 Loans

La classe `Loans` rappresenta la tabella dei libri in prestito. Al suo interno abbiamo due relazioni principali. La prima è una relazione `@ManyToOne` con `User`, in cui ogni istanza di `Loans` è collegata a un solo `User`. La colonna `cartid` di `Loans` è una foreign key che punta alla colonna `cartid` di `Users`. Questo permette di collegare in modo univoco ciascun prestito all'utente corrispondente. La seconda relazione è una `@ManyToOne` con `Books`, dove ogni `Loans` è collegato a un solo `Books`. La colonna `isbn` di `Loans` è una foreign key che punta alla colonna `isbn` di `Books`. Molti prestiti possono riferirsi allo stesso libro, perché, se abbiamo più copie di un libro, possono essere prese in prestito da utenti diversi.

```
@ManyToOne(cascade = CascadeType.REMOVE)
@JoinColumn(name = "cartid", referencedColumnName = "cartid")
private Users user;

@ManyToOne(cascade = CascadeType.REMOVE)
@JoinColumn(name = "isbn", referencedColumnName = "isbn")
private Books book;
```

Un prestito collega un utente e un libro; l'eliminazione di un utente o di un libro comporta la rimozione dei prestiti associati.

Attributi:

- `id`: Identificativo univoco.
- `loanDate`: Data di inizio prestito.
- `dueDate`: Data di scadenza.
- `returnDate`: Data di restituzione.
- `user`: Utente che ha preso in prestito.
- `book`: Libro prestato.

Relazioni:

- `@ManyToOne` con `Users`.
- `@ManyToOne` con `Books`.

3.1.5 Users

La classe `Users` rappresenta la tabella che contiene i dati principali di un utente o di un admin. Come "Primary Key" è stato scelto il `cartid`, data la sua unicità anche nella vita reale.

Attributi:

- `cartId`: Identificativo univoco della tessera.
- `name`: Nome utente.
- `lastname`: Cognome utente.

3.1.6 Utente

La classe `Utente` rappresenta la tabella degli utenti registrati. La sua funzione principale è facilitare la procedura di login. Come "Primary Key" è stato scelto il `username`, per favorire una ricerca rapida durante il processo di autenticazione. All'interno della classe è presente una relazione `@OneToOne` con la classe `Users`. La tabella `utente` contiene una colonna `cartId`, che è una foreign key riferita alla colonna `cartId` della tabella `users`. Ogni `cartId` può comparire una sola volta nella tabella `utente`, garantendo così un legame uno-a-uno tra `Utente` e `Users`. Inoltre, ogni `Utente` deve essere associato a un `Users`: non possono esistere istanze di `Utente` senza un corrispondente `Users`.

```
@OneToOne
@JoinColumn(name = "cartId", referencedColumnName = "cartId", nullable = false)
private Users users;
```

Attributi:

- `username`: Nome utente.
- `password`: Password di accesso.
- `users`: Collegamento alla classe `Users`.

Relazioni:

- `@OneToOne` con `Users`.

3.2 ValidationUtils

Il package `ValidationUtils` contiene una serie di classi di utilità dedicate alla validazione dei dati forniti dagli utenti e dagli amministratori del sistema. Queste classi svolgono un ruolo fondamentale nel garantire che i dati inseriti siano completi, coerenti e conformi ai requisiti previsti, prima che vengano elaborati o salvati nel database.

Ogni classe è specializzata in un contesto specifico:

- **ValidationUtilsAdmin**: gestisce la validazione delle operazioni amministrative, come l'aggiunta o la rimozione di libri e utenti. Controlla che campi come ISBN, autore, titolo, genere, lingua ed edizione siano correttamente compilati e che il numero di copie disponibili sia un valore numerico valido.
- **ValidationUtilsPassword**: si occupa della validazione delle credenziali degli utenti, con particolare attenzione alla creazione e al cambiamento delle password. Verifica che la password rispetti criteri di sicurezza, quali lunghezza minima, presenza di lettere maiuscole, numeri e caratteri speciali.
- **ValidationUtilsUtente**: valida le operazioni relative alla gestione dei prestiti e delle restituzioni da parte degli utenti. Controlla la correttezza dei dati relativi ai titoli dei libri, ai codici identificativi degli utenti (cartID) e alla validità delle date di prestito e scadenza.
- **ValidatioUtilsAccess**: effettua la validazione dei dati di accesso durante il processo di login, assicurandosi che username e password siano presenti e non vuoti.

Tutti i metodi presenti all'interno di queste classi sono dichiarati come `static`. Questa scelta consente di poterli richiamare direttamente senza la necessità di creare un'istanza della classe, rendendo il codice più semplice, leggero e veloce, in quanto le classi di utilità non devono mantenere stato interno e sono pensate per offrire funzionalità di servizio.

I metodi di validazione restituiscono una lista di errori (`List<String>`), che i Controller possono utilizzare per fornire all'utente risposte puntuali ed esplicative in caso di errori di inserimento. Questa organizzazione consente di mantenere separata la logica di validazione dalla logica di business, migliorando la manutenibilità, la chiarezza e la scalabilità dell'applicazione.

3.3 Controller

Nel sistema di gestione della biblioteca, i **Controller** fungono da intermediari tra il livello di persistenza (DAO) e l'interfaccia utente o altri componenti applicativi. Ogni controller ha una responsabilità specifica nella gestione delle funzionalità del sistema.

3.3.1 AccessController

Il `AccessController` è il componente responsabile della gestione del processo di autenticazione per tutti gli utenti del sistema, siano essi utenti normali oppure amministratori. L'autenticazione rappresenta uno dei punti chiave del sistema, poiché determina l'accesso sicuro alle funzionalità riservate e consente di differenziare le interazioni in base al ruolo.

Il controller espone due metodi principali:

- `accessUser`: consente a un utente registrato di accedere al sistema, previa verifica delle credenziali.
- `accessAdmin`: gestisce l'autenticazione degli amministratori, con una fase di validazione aggiuntiva rispetto agli utenti standard.

Entrambi i metodi restituiscono un oggetto `ResponseDTO` che incapsula l'esito dell'autenticazione, un messaggio descrittivo e, in caso positivo, l'entità `Utente` o `Admin` autenticata. Il controller si affida ai DAO (`UtenteDao`, `AdminDao`) per eseguire le query sul database.

```
1  public class AccessController {
2
3      public ResponseDTO<Utente> accessUser(String username, String password) {
4          boolean trova = utenteDao.doesUserExist(username, password);
5          if (trova) {
6              Utente utente = utenteDao.findUtenteUsername(username, password);
7              return new ResponseDTO<>(true, "Utente trovato con successo.", utente)
                   ;
8          }
9          return new ResponseDTO<>(false, "Utente non trovato.", null);
10     }
11     public ResponseDTO<Admin> accessAdmin(String username, String password) {
12         List<String> errors = ValidatioUtilsAccess.validateAccess(username,
               password);
13         if (!errors.isEmpty()) {
14             return new ResponseDTO<>(false, "Errore nella richiesta: " + String.
                   join(", ", errors), null);
15         }
16
17         boolean trova = adminDao.doesUtenteExist(username, password);
18         if (trova) {
19             Admin admin = adminDao.findAdminByUsernameAndPassword(username,
                   password);
20             return new ResponseDTO<>(true, "Admin trovato con successo.", admin);
21         } else {
22             return new ResponseDTO<>(false, "Admin non trovato.", null);
23         }
24     }
25 }
```

Listing 2: AccessController.java

3.3.2 AdminController

Il `AdminController` è dedicato alla gestione delle funzionalità amministrative della biblioteca digitale. È accessibile esclusivamente agli utenti di tipo amministratore e consente la gestione delle risorse principali del sistema, tra cui i libri e gli utenti.

Le operazioni principali gestite da questo controller sono:

- Inserimento di un nuovo libro nel catalogo.
- Eliminazione di un libro esistente.
- Rimozione di utenti registrati.

Tutte le operazioni sono precedute da una fase di validazione, che verifica la correttezza dei dati forniti e previene l'inserimento di valori incoerenti nel database. In caso di successo, le operazioni restituiscono un oggetto `ResponseDTO` che contiene le informazioni relative all'entità appena modificata o aggiunta (es. un libro o un utente).

Il controller collabora con le classi DAO (`BookDao`, `AutoreDao`, `UserDao`) per eseguire l'inserimento e la cancellazione degli elementi nel database. La logica è progettata per mantenere il sistema coerente anche in presenza di errori: per esempio, la rimozione di un libro è consentita solo se tutte le copie sono disponibili in biblioteca.

```
1  public class AdminController {
2
3      public ResponseDTO<Books> removeBook(String isbn, String userRole) {
4          Books book = bookDao.findBookByIsbn(isbn);
5          if (book == null) {
6              return new ResponseDTO<>(false, "Libro non trovato", null);
7          }
8          bookDao.removeBookByisbn(isbn);
9          return new ResponseDTO<>(true, "Libro eliminato con successo", null);
10     }
11
12     public ResponseDTO<Books> addBook(String isbn, String autore, String titolo,
           String genere, String lingua, String edizione, String disponibili, String
           userRole) {
13         List<String> errors = ValidationUtilsAdmin.addBook(isbn, autore, titolo,
               genere, lingua, edizione, disponibili);
14         if (!errors.isEmpty()) {
15             return new ResponseDTO<>(false, "Errore nella richiesta: " + String.
                   join(", ", errors), null);
16         }
17
18         int numeroDisponibili = Integer.parseInt(disponibili);
19         Autore autore1 = new Autore(autore);
20
21         autoreDao.insert(autore1);
22         Books book = new Books(isbn, autore1, titolo, genere, lingua,
               numeroDisponibili, edizione);
23
24
25         bookDao.insert(book);
26
27         return new ResponseDTO<>(true, "Libro aggiunto correttamente", book);
28     }
29      public ResponseDTO<Void> eliminaUtenteController(String cartID, String
            userRole) {
30         List<String> errors = ValidationUtilsAdmin.removeUtente(cartID);
31         if (!errors.isEmpty()) {
32             return new ResponseDTO<>(false, "Errore nella richiesta: " + String.
                   join(", ", errors), null);
33         }
34         boolean trova = userDao.findUser(cartID);
35         if (!trova) {
36             return new ResponseDTO<>(false, "Utente non trovato.", null);
37         }
38         loanProxy = new LoanDaoProxy(loansDao, userRole);
39         loanProxy.removeUserAndStory(cartID);
40
41         return new ResponseDTO<>(true, "Utente eliminato correttamente.", null);
42     }
43 }
```

Listing 3: AdminController.java

3.3.3 PasswordController

Il `PasswordController` gestisce l'aggiornamento della password da parte degli utenti del sistema. Prima di procedere alla modifica, il controller effettua una serie di verifiche sui dati personali inseriti (come `cartId`, `nome`, `cognome`, `username`) per garantire la sicurezza e prevenire modifiche non autorizzate.

In base al ruolo dell'utente (distinto tramite il flag `isAdmin`), il controller delega l'aggiornamento della password al rispettivo `Dao` (`UtenteDao` o `AdminDao`). Al termine dell'operazione, viene restituito un oggetto `ResponseDTO` che contiene l'esito della richiesta, un eventuale messaggio di errore o successo, e l'entità aggiornata in caso positivo.

```
1  public ResponseDTO<?> newPassword(String cartId, String nome, String cognome,
2                                    String username, String password, boolean
                                         isAdmin) {
3      List<String> errors = ValidationUtilsPassword.newPassword(cartId, nome,
           cognome, username, password);
4      if (!errors.isEmpty()) {
5          return new ResponseDTO<>(false, "Errore␣nella␣richiesta:␣" + String.join("
               ,␣", errors), null);
6      }
7
8      if (!userDao.controllUser(cartId, nome, cognome)) {
9          return new ResponseDTO<>(false, "User␣non␣trovato", null);
10     }
11
12     if (!isAdmin) {
13         utenteDao.updatePassword(username, password);
14         Utente utente = utenteDao.findUtenteUsername(username, password);
15         if (utente == null) {
16             return new ResponseDTO<>(false, "Errore␣nell'aggiornamento␣della␣
                   password", null);
17         } else {
18             return new ResponseDTO<>(true, "Password␣modificata", utente);
19         }
20     }
21
22     adminDao.updatePassword(username, password);
23     Admin admin1 = adminDao.findAdminByUsernameAndPassword(username, password);
24     if (admin1 == null) {
25         return new ResponseDTO<>(false, "User␣non␣trovato", null);
26     } else {
27         return new ResponseDTO<>(true, "Password␣modificata", admin1);
28     }
29 }
```

Listing 4: PasswordController.java

3.3.4 RegisterController

Il `RegisterController` consente la registrazione di nuovi utenti nel sistema, siano essi normali lettori oppure amministratori. Si tratta di un componente fondamentale del sistema poiché gestisce la fase di onboarding degli utenti e garantisce che tutte le informazioni siano coerenti e conformi ai vincoli del sistema.

Il controller offre due metodi principali:

- `registerUtente`: per la registrazione di un utente standard.
- `registerAdmin`: per la registrazione di un amministratore, che richiede la conoscenza di un codice identificativo della biblioteca.

Prima dell'inserimento, i dati sono sottoposti a una fase di validazione tramite `ValidationUtilsPassword`. Successivamente, viene creata l'entità `Users` e associata a un oggetto `Utente` o `Admin`, che viene poi salvato nel database tramite i rispettivi DAO.

Il controller restituisce un oggetto `ResponseDTO` che informa l'utente sull'esito dell'operazione e include l'oggetto appena registrato, in caso di successo.

```
1  public class RegisterController {
2
3      public ResponseDTO<Utente> registerUser(String cartId, String nome, String
           cognome, String username, String password) {
4          Utente utente = new Utente(username, password);
```

```
        userDao.addUser(utente);
        return new ResponseDTO<>(true, "Utente registrato", utente);
    }
    public ResponseDTO<Admin>registerAdmin(String cartId,String nome,String
        cognome,String username,String password,String idBiblioteca){
        List<String> errors = ValidationUtilsPassword.newPassword(cartId, nome,
            cognome, username, password);
        if(!errors.isEmpty()){
            return new ResponseDTO<>(false,"Errore nella richiesta: " + String.
                join(", ", errors),null);
        }else{
            users = new Users(cartId, nome, cognome);
            admin = new Admin(idBiblioteca, username, password, users);

            userDao.addUser(users);

            adminDao.addAdmin(admin);
            return new ResponseDTO<>(true,"Admin registrato",admin);
        }

    }
}
```

Listing 5: RegisterController.java

3.3.5 UtenteController

Il `UtenteController` è il punto di ingresso per tutte le operazioni che un utente registrato può compiere all'interno del sistema. Le sue responsabilità includono la gestione del prestito dei libri, la restituzione, la visualizzazione dello storico e la consultazione dei libri di uno specifico autore.

Le principali funzionalita sono:

- `prendiPrestito`: consente all'utente di prendere un libro in prestito, specificando il periodo desiderato.
- `restituisciLibro`: aggiorna il sistema per segnalare la restituzione di un libro precedentemente preso in prestito.
- `listaLibriAutore`: restituisce l'elenco dei libri associati a un determinato autore.

Ogni operazione è preceduta da una fase di validazione, e viene eseguita solo se il sistema conferma che i dati siano corretti. In caso di prestito, ad esempio, viene verificata la disponibilità del libro prima di creare un nuovo oggetto `Loans`. In fase di restituzione, la disponibilità del libro viene aggiornata di conseguenza insieme alla data di restituzione.

Il controller comunica con i DAO (`BookDao`, `LoansDao`, `UserDao`, `AutoreDao`) e restituisce una risposta strutturata tramite `ResponseDTO`.

```
public ResponseDTO<Books> prendiPrestito(String isbn, String id, LocalDate
    currentDate, LocalDate dueDate) {
        List<String> errors = ValidationUtilsUtente.prendi(isbn, id, currentDate,
            dueDate);
        if (!errors.isEmpty()) {
            return new ResponseDTO<>(false, "Errore nella richiesta: " + String.
                join(", ", errors), null);
        }
        Books books = bookDao.findBookByIsbn(isbn);
        if (books == null) {
            return new ResponseDTO<>(false, "Libro non trovato.", null);
        }
        Users user = userDao.findUserByCartId(id);
        if (user == null) {
            return new ResponseDTO<>(false, "Utente non trovato.", null);
        }
        int disponibili = books.getDisponibili();
        if (disponibili == 0) {
            return new ResponseDTO<>(false, "Libro non disponibile perch gi in
                prestito.", null);
        }
        loan = new Loans(currentDate, dueDate, user, books);
        loansDaoProxy = new LoanDaoProxy(loansDao, "user");
        loansDaoProxy.addLoan(loan);

```

```
22          bookDaoProxy = new BookDaoProxy(bookDao, "admin");
23          bookDaoProxy.cambiaDisponibilita(isbn, disponibili - 1);
24
25          return new ResponseDTO<>(true, "Prestito andato con successo.", books);
26      }
27      public ResponseDTO<List<LibroAutore>>listaLibriAutore(String nomeAutore){
28          if(nomeAutore == null || nomeAutore.isEmpty()){
29              return new ResponseDTO<>(false,"Errore: nome Autore non valido.", null
                    );
30          }
31          List<LibroAutore> lista = autorDao.libriAutore(nomeAutore);
32          if(lista == null || lista.isEmpty()){
33              return new ResponseDTO<>(false,"Nessun libro trovato", null);
34          }
35          return new ResponseDTO<>(true, "Libro trovato", lista);
36      }
37  }
```

Listing 6: UtenteController.java

.

3.4 Pattern DAO (Data Access Object)

DAO (Data Access Object) è un pattern architetturale che ha l'obiettivo di separare la logica di accesso ai dati dalla logica di business dell'applicazione. Le classi DAO implementano un'interfaccia generica `Dao<T>`, la quale definisce metodi comuni come `insert(T obj)` e `delete(T obj)`. Questo approccio consente di standardizzare le operazioni di accesso al database e di ridurre la duplicazione del codice.

Ogni classe DAO può includere, oltre ai metodi comuni, anche metodi specifici che permettono di interagire direttamente con il database, come query personalizzate, filtri o ricerche. È dunque responsabilità delle classi DAO gestire l'inserimento, la modifica, l'eliminazione e il recupero dei dati dal database, mantenendo il resto dell'applicazione disaccoppiato dalla logica di persistenza.

3.4.1 DAO Interface

DAO è un'interfaccia generica che definisce operazioni di base comuni (insert, delete) per tutte le classi DAO.

.

```
1  public interface Dao<T> {
2      public void insert(T obj);
3      public void delete(T obj);
4  }
```

Listing 7: Interface DAO.java

3.4.2 AdminDAO

AdminDao è la classe responsabile della gestione della Entity `Admin` presente nel database. Questa classe consente di effettuare operazioni fondamentali come l'inserimento, l'eliminazione e la ricerca di amministratori in base al loro `username`.

Oltre ai metodi ereditati dall'interfaccia generica `Dao`, `AdminDao` include il metodo `findAdminByUsernameAndPas` che esegue una query mirata per verificare le credenziali dell'amministratore durante la fase di login.

3.4.3 UserDao

UserDao è la classe responsabile della gestione dell'entità `Users` presente nel database. Oltre ai metodi `insert` e `delete`, ereditati dall'interfaccia generica `Dao`, include il metodo `controllUser`, che tramite una query permette di verificare se un determinato utente è già presente nel database.

3.4.4 UtenteDao

UtenteDao e' la classe responsabile della gestione dell'entita `Utente` presente nel database. Oltre ai metodi `insert` e `delete`, ereditati dall'interfaccia generica `Dao`, inculede il metodo `updatePassword` che permette di aggiornare il password e altri metodi funzionali.

3.4.5 AutoreDao

AutoreDao e' la classe responsabile della gestione dell'entita `Autore` presente nel database. Oltre ai metodi `insert` e `delete`, ereditati dall'interfaccia generica `Dao`, include il metodo `libriAutore` che permette di restituire tutti i libri scritti dall'autore. Piu altri metodi utili per il funzionamento del software.

3.4.6 LoansDao

LoansDao è la classe responsabile della gestione dell'entità `Loan` presente nel database. Oltre ai metodi `insert` e `delete`, ereditati dall'interfaccia generica `Dao`, la classe implementa ulteriori metodi specifici, tra cui `setDueDate` e `libriloan`.

Il metodo `setDueDate` consente di impostare la data di restituzione per un prestito, mentre `libriloan` permette di ottenere la lista dei libri attualmente presi in prestito da un determinato utente.

3.4.7 BookDao

BookDao è la classe responsabile della gestione dell'entità `Books` presente nel database. Oltre ai metodi `insert` e `delete`, ereditati dall'interfaccia generica `Dao`, la classe implementa ulteriori metodi specifici tra cui `public boolean cambiaDisponibilita(String isbn, int nuoviDisponibili)` e `public Books findById(String isbn)`. La prima permette di cambiare la disponibilita dei libri disponibili per il prestito la seconda invece restituire ricerca e restituisce il libro in base alla sua `Primary Key` ovver `isbn`.

Vediamo l'implementazione delle due funzioni in una delle classi DAO che implemenetano l'interface DAO.

```
1     @Override
2     public void insert(Admin obj) {
3         if (!isManaged) {
4             this.em = ConnectionDB.getInstance().getEntity();
5         }
6         try {
7             em.getTransaction().begin();
8             em.persist(obj);
9             em.getTransaction().commit();
10        } catch (Exception e) {
11            if (em.getTransaction().isActive()) {
12                em.getTransaction().rollback();
13            }
14            throw e;
15        } finally {
16            if (!isManaged) {
17                em.close();
18            }
19        }
20    }
```

Listing 8: Insert

```
1     @Override
2     public void delete(Admin obj) {
3         if (!isManaged) {
4             this.em = ConnectionDB.getInstance().getEntity();
5         }
6         try {
7             em.getTransaction().begin();
8             if (!em.contains(obj)) {
9                 obj = em.merge(obj);
10            }
11            em.remove(obj);
12            em.getTransaction().commit();
13        } catch (Exception e) {
14            if (em.getTransaction().isActive()) {
15                em.getTransaction().rollback();
16            }
17            throw e;
18        } finally {
19            if (!isManaged) {
20                em.close();
```

```
21                 }
22             }
23         }
```

3.4.8 Funzionalità Hibernate in `AdminDao`

Vediamo l'uso di Hibernate nelle due implementazioni presi come esempio che implementano l'interface `DAO` La classe `AdminDao` utilizza Hibernate per gestire le operazioni di persistenza . In particolare, i metodi `insert` e `delete` implementano le funzionalità fondamentali per inserire e rimuovere un oggetto nel database, sfruttando pienamente le capacità offerte dal *Java Persistence API* (JPA) con Hibernate come provider.

Metodo `insert(Admin obj)` Questo metodo consente di salvare un nuovo amministratore nel database. Di seguito la spiegazione dettagliata delle istruzioni Hibernate utilizzate e il resto:

- `if (!isManaged)  this.em = ConnectionDB.getInstance().getEntity();`

  Questa istruzione mi permette di gestire in modo efficace i vari Test. Visto che viene usato un'altra connessione che permette di fare i test. In questo caso se la varibile e' falsa allora la connessione avviene con il database principale.

- `em.getTransaction().begin();`
  Avvia una nuova transazione. Le operazioni di scrittura nel database devono sempre essere racchiuse in una transazione per essere persistite correttamente.

- `em.persist(obj);`
  Inserisce l'oggetto `Admin` nel database. Hibernate genera automaticamente una query `INSERT INTO` basata sulle annotazioni dell'entità.

- `em.getTransaction().commit();`
  Conferma la transazione, rendendo permanenti le modifiche nel database.

- Blocco `catch / finally`
  In caso di errore, viene eseguito un `rollback()` per annullare la transazione, evitando modifiche parziali. Se la `EntityManager` non è gestita esternamente (`isManaged = false`), viene chiusa esplicitamente per liberare risorse. Questo per lo stesso motivo per il quale viene usato principalmente all'inizio `isManged`

Metodo `delete(Admin obj)` Questo metodo elimina un amministratore esistente dal database. Utilizza alcune funzionalità aggiuntive di Hibernate per garantire che l'entità sia correttamente gestita prima dell'eliminazione.

- `if (!em.contains(obj)) { obj = em.merge(obj); }`
  Controlla se l'oggetto è attualmente gestito dalla `EntityManager`. Se non lo è, viene unito al contesto di persistenza tramite `merge()`, rendendolo gestibile.

- `em.remove(obj);`
  Elimina l'oggetto dal database. Hibernate genera una query `DELETE FROM` corrispondente.

- `begin()` / `commit()` / `rollback()` / `close()`
  Gestione della transazione e delle risorse, come già visto nel metodo di inserimento.

3.4.9 Pattern Singleton

Un pattern molto importante applicato e' quello Singleton.Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

```
1  public class ConnectionDB {
2
3      private static ConnectionDB instance;
4      private EntityManagerFactory factory;
5
6      private ConnectionDB() {
7          factory = Persistence.createEntityManagerFactory("Library");
8      }
9
10     public static ConnectionDB getInstance() {
11         if(instance == null) {
12             instance = new ConnectionDB();
13         }
14         return instance;
15     }
16
17     public EntityManager getEntity() {
18         return factory.createEntityManager();
19     }
20 }
```

Listing 9: Connesione DataBase.java

4 Note sulla connessione al Database

4.1 DBMS: PostgreSQL

Il DBMS utilizzato e' PostgreSQL. E' un RDBMS (DBMS relazionale) in cui i dati sono organizzati in tabelle messe in relazione tra loro. In questo caso, le classi Java rappresenteranno le entita', con gli attributi come colonne, e gli oggetti i records. Per praticita' il server Postgres 'e stato installato in locale, sulla stessa macchina su cui il progetto 'e stato sviluppato. Per evitare di contaminare i dati contenuti del Database dell'applicazione chiamato "LibraryDB", 'e stato creato sullo stesso server un secondo database chiamato "test" per testare anche l'esecuzione di test su dati presenti nel database.

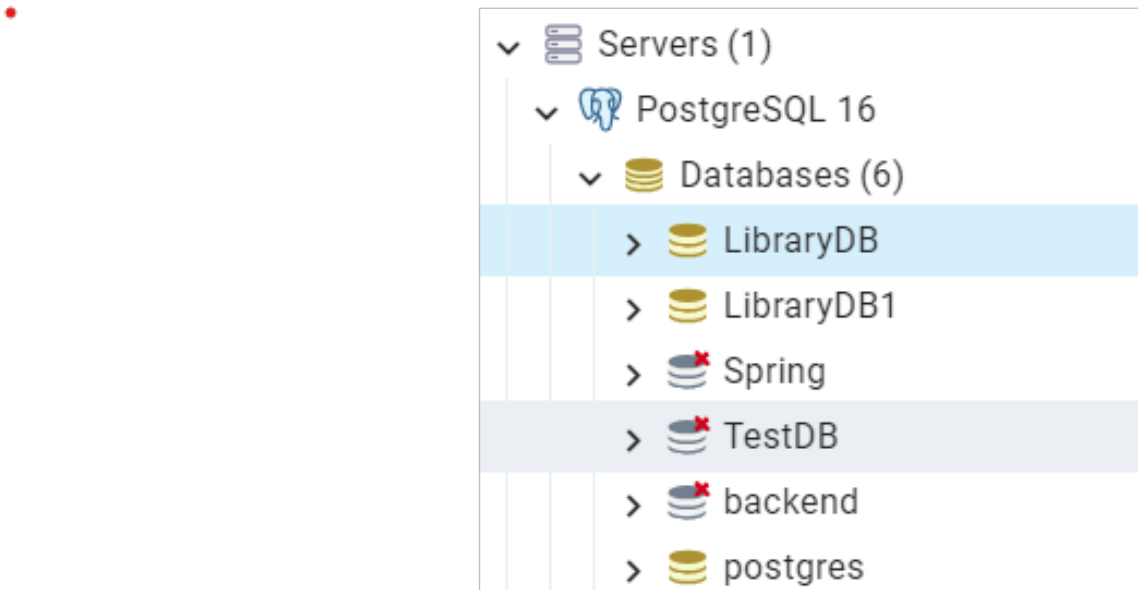


Figura 5: Database utilizzati per Applicazione e fase di Test.

In questa sezione ‘e possibile visualizzare le varie classi java instanziate come entit‘a nel database (admin, autore,books,loans,users,utente).

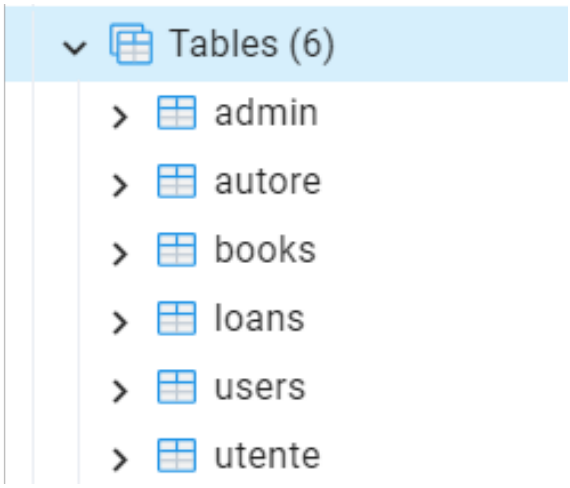


Figura 6: Tabelle presenti nel DB.

5 DTO Data Transfer Object

5.1 Introduzione

DTO è un design pattern usato per trasferire dati tra sottosistemi di un’applicazione software. I DTO sono spesso usati in congiunzione con gli oggetti di accesso ai dati (DAO) per recuperare i suddetti da una base di dati. Nel caso in esame e’ stato usato tra il livello DAO (Data Access Object) e il livello controller.

—

Il DTO ci permette di evitare il sovraccarico e la complessità delle entità JPA, restituendo solo i dati realmente utili e ben formattati. Mantenere separazione e sicurezza tra livelli dell’applicazione e semplificare i controller e l’interfaccia con un eventuale frontend.

5.1.1 Come funziona

Nel progetto abbiamo all’interno del package DTO due classi pure DTO ”LibroAutore” e ”LoanBook” e un altra classe chiamata ”ResponseDTO” . I DTO (LibroAutore, LoanBook) sono pensati per mostrare solo le informazioni utili al livello superiore (es. controller o frontend). Invece se si inizia a esporre API REST o a restituire risposte da metodi di servizio, ResponseDTO mi permette di uniformare il formato della risposta, includere un messaggio e un flag success per il client ed evitare confusione tra errori e dati nulli.

5.1.2 Implementazione nel progetto

Abbiamo creato due proxy:

- **LoanBook**: Permette di restituire la lista di libri presi in prestito dall’ Utente.
- **LibroAutore**: Permette di restituire la lista di libri scritti da un determinato Autore.
- **ResponseDTO**: serve per restituire una risposta strutturata da un metodo o servizio. Contiene lo stato dell’operazione (successo o fallimento), un messaggio, e i dati restituiti.

```
1 public class LoanBook {
2     private String titolo;
3     private Autore autore;
4     private String cartId;
5     private String isbn;
6     private LocalDate loanDate;
7     private LocalDate dueDate;
8     private LocalDate returnDate;
9
```

```
10
11      public LoanBook(String titolo, Autore autore, String cartId, String isbn,
            LocalDate loanDate, LocalDate dueDate, LocalDate returnDate)
12      }
```

Listing 10: LoanBook.java

```
1   public class LibroAutore {
2
3       private String nomeAutore;
4       private String isbn;
5       private String titolo;
6       private int disponibilita;
7
8       public LibroAutore(String nomeAutore, String isbn, String titolo, int
            disponibilita)
9       }
```

Listing 11: LibroAutore.java

```
1       public class ResponseDTO <T>{
2           private boolean success;
3           private String message;
4           private T data;
5
6           public ResponseDTO(boolean success, String message, T data)
```

Listing 12: ResponseDTO .java

5.1.3 Esempio pratico

LibroAutore

In questo caso vediamo l'esempio di una Query in qui viene utilizzata la classe LibroAutore per poter restituire una tabella formata da 4 colonne.

```
1       public List<LibroAutore>libriAutore(String nomeAutore){
2           if(!isManged){
3               this.em = ConnectionDB.getInstance().getEntity();
4           }
5           try{
6              Query query= em.createQuery("Select␣new␣org.Library.DTO.LibroAutore(a.
                   nome,b.isbn,b.title,b.disponibili)" +
7                       "␣from␣Books␣b␣join␣b.autore␣a␣" +
8                       "where␣a.nome␣=␣:nomeAutore",
9                       LibroAutore.class);
10              query.setParameter("nomeAutore",nomeAutore);
11              return query.getResultList();
12          }finally {
13              if(!isManged){
14                  em.close();
15              }
16          }
17      }
```

Listing 13: Query LibroAutore.java

LoansBook

Invece in questo caso vediamo l'esempio di una Query in qui viene utilizzata la classe LoansBooke per poter restituire una tabella formata da piu colonne. Essa permette ad ogni utente di vedere la propria cronologia dei libri presi in prestito.

```
1       public List<LoanBook>libriloan(String cartId){
2           if(!isManaged){
3               em = ConnectionDB.getInstance().getEntity();
4           }
5           try{
6               Query query= em.createQuery("SELECT␣new␣org.Library.DTO.LoanBook(b.
                    title,b.autore,l.user.cartId,l.book.isbn,l.loanDate,l.dueDate,l.
                    returnDate)"+
7                       "from␣Loans␣l␣JOIN␣Books␣b␣on␣l.book.isbn␣=␣b.isbn␣␣WHERE␣l.
                            user.cartId␣=␣:cartID",LoanBook.class);
8               query.setParameter("cartID", cartId);
```

```
            return query.getResultList();

        }finally {
            if(!isManaged){
                em.close();
            }
        }
    }
```

Listing 14: Query LoanBook.java

ResponseDTO

Le istanze di ResponseDTO sono le piu utilizzate nel progette. Vediamo un esmpio d'uso. In particolare il caso di accesso di un USER.

```
    public ResponseDTO<Utente> accessUser(String username, String password) {
        List<String> errors = ValidatioUtilsAccess.validateAccess(username,
            password);
        if (!errors.isEmpty()) {
            return new ResponseDTO<>(false, "Errore nella richiesta: " + String.
                join(", ", errors), null);
        }


        boolean trova = utenteDao.doesUserExist(username, password);
        if (trova) {
            Utente utente = utenteDao.findUtenteUsername(username, password);
            return new ResponseDTO<>(true, "Utente trovato con successo.", utente)
                ;
        } else {
            return new ResponseDTO<>(false, "Utente non trovato.", null);
        }
    }
```

Listing 15: Query ResponseDTO.java

5.2 Conclusione

L'uso dei DTO diventa necessario quando si vuole trasferire solo le informazioni essenziali tra i vari livelli dell'applicazione, evitando di esporre direttamente le entità del database. In questo caso, l'applicazione dei DTO è semplice ma efficace, e permette di mantenere il codice pulito, sicuro e ben organizzato.

6 Test

6.1 Test Funzionali

Questa sezione documenta i test unitari delle principali classi controller. Ogni classe viene testata con JUnit per verificare la correttezza delle operazioni implementate.

6.1.1 Test AccessController

La classe `AccessControllerTest` verifica il processo di auteticazione degli utenti. In primi due test l'accesso e' negato. Nel primo test dovuto al fatto che o username o il password non sono corretti. Nel secondo test si ha quando sia username che password sono entrambi vuoti.Il terzo test invece l'accesso e' positivo.

```
@Test
public void testAccessUser_NotSuccess() {
    ResponseDTO<Utente> response = accessController.accessUser("testUser", "
        testPass");
    assertFalse(response.isSuccess());
    assertEquals("Utente non trovato.", response.getMessage());
}

@Test
public void testAccessUser_InvalidInput() {
    ResponseDTO<Utente> response = accessController.accessUser("", "");
    assertFalse(response.isSuccess());
    assertEquals("Errore nella richiesta: Username e password non possono essere
        vuoti", response.getMessage());
}
@Test
    public void testAccessUser_Success() {

        ResponseDTO<Utente> response = registerController.registerUser("12345", "
            Mario", "Rossi", "mrossi", "Password123!");
        assertTrue(response.isSuccess());
        assertEquals("Utente registrato", response.getMessage());
        ResponseDTO<Utente> response1 = accessController.accessUser("mrossi", "
            Password123!");
        assertTrue(response1.isSuccess());
        assertEquals("Utente trovato con successo.", response1.getMessage());
    }
```

Listing 16: AccessControllerTest.java

6.1.2 Test AdminController

La classe `AdminControllerTest` verifica la possibilita di rimuovere o di aggiungere libri da parte dell'amministratore. Nel primo test non e' possibile rimuove il libro perche' esso non si trova nel database. Nel secondo test il libro viene aggiunto correttamente. Nel terzo test viene eliminato correttamente un libro.

```
@Test
public void testRemoveBook_BookNotFound() {
    ResponseDTO<Books> response = adminController.removeBook("0000", "admin");
    assertFalse(response.isSuccess());
    assertEquals("Libro non trovato", response.getMessage());
}

@Test
public void testAddBook_Success() {
    ResponseDTO<Books> response = adminController.addBook(
        "789456", "Isaac Asimov", "Fondazione", "Fantascienza", "Italiano", "3
            Edizione", "7", "admin"
    );
    assertTrue(response.isSuccess());
    assertEquals("Libro aggiunto correttamente", response.getMessage());
}

@Test
    public void testRemoveBook_Success() {
        adminController.addBook("654321", "George Orwell", "1984", "Distopia", "
            Italiano", "2  Edizione", "5", "admin");
```

```
20
21          ResponseDTO<Books> response = adminController.removeBook("654321",  "admin
                ");
22          assertEquals("Libro␣eliminato␣con␣successo", response.getMessage());
23          assertTrue(response.isSuccess());
24
25
26          Books libroEliminato = em.find(Books.class, "654321");
27          assertNull(libroEliminato);
28      }
```

Listing 17: AdminControllerTest.java

6.1.3 Test PasswordController

La classe `PasswordControllerTest` verifica la funzionalità di reset della password. Nel primo test visto che si puo resetare la password solo se si sanno cartId, nome,cognome,username in questo caso uno di questi elementi non corrisponde ai dati in database. Nel secondo test invece i dati corrispondo con quelli in database e di seguito si ha la conferma di cambio di password.

```
1  @Test
2  public void testNewPassword_UserNotFound() {
3      ResponseDTO<Utente> response = passwordController.newPassword("12345", "Test",
           "User", "testUser", "NewPass1!");
4      assertFalse(response.isSuccess());
5      assertEquals("User␣non␣trovato", response.getMessage());
6  }
7
8  @Test
9  public void testNewPassword_Success() {
10     ResponseDTO<Utente> response = registerController.registerUser("12345", "Mario
           ", "Rossi", "mrossi", "Password123!");
11     assertTrue(response.isSuccess());
12
13     ResponseDTO<Utente> response1 = passwordController.newPassword("12345", "Mario
           ", "Rossi", "mrossi", "NewPass2!");
14     assertTrue(response1.isSuccess());
15     assertEquals("Password␣modificata", response1.getMessage());
16 }
```

Listing 18: PasswordControllerTest.java

6.1.4 Test RegisterController

La classe `RegisterControllerTest` verifica la registrazione di utenti e amministratori. Nel primo test abbiamo la registrazione di un Utente. Quindi il nuovo user sceglie di essere un utente e deve inserire un username unico e deve rispettare determinati requisit a livello di password.Nel secondo test si verfica che se uno dei dati manca allora la registrazione non puo avvenire con successo.

```
1  @Test
2  void testRegisterUser_Success() {
3      ResponseDTO<Utente> response = registerController.registerUser("12345", "Mario
           ", "Rossi", "mrossi", "Password123!");
4      assertTrue(response.isSuccess());
5      assertEquals("Utente␣registrato", response.getMessage());
6  }
7
8  @Test
9  void testRegisterAdmin_Fail_EmptyFields() {
10     ResponseDTO<Admin> response = registerController.registerAdmin("", "", "", "",
           "", "");
11     assertFalse(response.isSuccess());
12     assertTrue(response.getMessage().contains("Errore␣nella␣richiesta"));
13 }
```

Listing 19: RegisterControllerTest.java

6.1.5 Test UtenteController

La classe `UtenteControllerTest` verifica la gestione del prestito e della restituzione dei libri. Il primo test verifica la possibilita dell'utente di prendere il prestito un libro. In questo caso il prestito non va a buon fine perche non viene trovato i titolo ricercato. Nel secondo test abbiamo la restituzione di un libro che prima pero viene preso in prestito. In questo test si ha un esito positivo della resituzione del libro. L'utlimo test e' quello legato alla lista dei libri in Prestito dall'utente.

```
@Test
public void testPrendiPrestito_BookNotFound() {
    ResponseDTO<Books> response = utenteController.prendiPrestito("Titolo␣
        inesistente", "12345", LocalDate.now(), LocalDate.now().plusDays(7));
    assertFalse(response.isSuccess());
    assertEquals("Libro␣non␣trovato.", response.getMessage());
}

@Test
public void testRestituisciLibro_Success() {
    ResponseDTO<Books> responseBook = adminController.addBook("654321", "George␣
        Orwell", "1984", "Distopia", "Italiano", "2  ␣Edizione", "5", "admin");
    assertTrue(responseBook.isSuccess());

    ResponseDTO<Utente> responseUser = registerController.registerUser("12345", "
        Mario", "Rossi", "mrossi", "Password123!");
    assertTrue(responseUser.isSuccess());

    LocalDate currentDate = LocalDate.now();
    LocalDate dueDate = currentDate.plusMonths(1);
    ResponseDTO<Books> responsePrestito = utenteController.prendiPrestito("654321"
        , "12345", currentDate, dueDate);
    assertTrue(responsePrestito.isSuccess());

    ResponseDTO<String> responseRestituzione = utenteController.restituisciLibro("
        654321", "12345");
    assertTrue(responseRestituzione.isSuccess());
    assertEquals("Libro␣restituito␣con␣successo.", responseRestituzione.getMessage
        ());
}

 @Test
    public void listaLibriUtente(){
        ResponseDTO<Books> responseBook= adminController.addBook("654321", "George
            ␣Orwell", "1984", "Distopia", "Italiano", "2  ␣Edizione", "5", "admin"
            );
        assertTrue(responseBook.isSuccess());
        ResponseDTO<Utente> response = registerController.registerUser("12345", "
            Mario", "Rossi", "mrossi", "Password123!");
        assertTrue(response.isSuccess());
        LocalDate currentDate = LocalDate.now();
        LocalDate dueDate = currentDate.plusMonths(1);
        ResponseDTO<Books> responsePrestito = utenteController.prendiPrestito("
            654321","12345",currentDate,dueDate);
        assertEquals("Prestito␣andato␣con␣successo.", responsePrestito.getMessage
            ());
        assertTrue(responsePrestito.isSuccess());
        assertNotNull(responsePrestito.getData());
        ResponseDTO<List<LoanBook>> libri = utenteController.listaPrestito("12345"
            );
        assertTrue(libri.isSuccess());
        assertNotNull(libri.getData());

    }
```

Listing 20: UtenteControllerTest.java

6.2 Test Di Unita

Lo unit testing è un metodo di test del codice, che prevede di testate singole unità di codice, e verificare il risultato ottenuto rispetto ai risultati previsti. In seguito alcuni test effettuati.

```
    @Test
    public void testInsert(){
        userDao.insert(user);
        dao.insert(admin);
        Admin controlla = dao.findAdminByUsernameAndPassword("A123456","123456");
        assertNotNull(controlla);
        assertEquals(controlla.getUsername(),"A123456");
        assertEquals(controlla.getPassword(),"123456");
    }
```

Listing 21: Insert Admin Test 2.java

```
    @Test
    public void insert(){
        utenteDao.insert(utente);
        Utente utente1 = utenteDao.findUtenteUsername("Francesco","F123456");
        Utente utente2 = utenteDao.userExist(users);
        boolean find = utenteDao.doesUserExist("Francesco","F123456");
        assertNotNull(utente1);
        assertNotNull(utente2);
        assertEquals("Francesco",utente1.getUsername());
        assertEquals("F123456",utente1.getPassword());
        assertTrue(find);
    }
```

Listing 22: Insert Utente Test 3.java

```
    @Test
    public void testInsertBook(){
        Autore autore=new Autore("George Orwell");
        autorDao.insert(autore);
        Books book=new Books("654321", autore, "1984", "Distopia", "Italiano", 2,
            "5");
        bookDao.insert(book);
        Books bookControlla = bookDao.findById("654321");
        assertNotNull(bookControlla);
    }



    }
```

Listing 23: Insert Book Test 4.java

```
     @Test
    public void testDeleteBook(){
        Autore autore=new Autore("George Orwell");
        autorDao.insert(autore);
        Books book=new Books("654321", autore, "1984", "Distopia", "Italiano", 2,
            "5");
        bookDao.insert(book);
        Books bookControlla = bookDao.findById("654321");
        assertNotNull(bookControlla);
        bookDao.delete(bookControlla);
        bookControlla = bookDao.findById("654321");
        assertNull(bookControlla);
```

Listing 24: Delete Book Test 5.java

```java
    @Test
    public void Loanstest(){
        dao.insert(loans);
        List<LoanBook> loan = dao.libriloan("A123456");

        assertNotNull(loan);
        assertEquals(1, loan.size());

        LoanBook loanBook = loan.get(0);
        assertEquals("1984", loanBook.getTitolo());
        assertEquals("George Orwell", loanBook.getAutore().getNome());
        assertEquals("A123456", loanBook.getCartId());
        assertEquals("654321", loanBook.getIsbn());
        assertNotNull(loanBook.getLoanDate());
        assertNotNull(loanBook.getDueDate());
        assertNull(loanBook.getReturnDate());
    }
```

Listing 25: Loans Test 6.java

```java
    @Test
    public void testSetDueDate() {
        dao.insert(loans);

        LocalDate returnDate = LocalDate.now().plusDays(15);
        Boolean result = dao.setDueDate("654321", "A123456", returnDate);
        em.clear();

        assertTrue(result);
        List<Loans> loansList = dao.findLoan("654321");
        Loans loans1 = loansList.get(0);

        assertNotNull(loans1);
        assertEquals(returnDate, loans1.getReturnDate());
    }
```

Listing 26: SetDueDate Test 7.java

```java

@Test
    public void insert(){
        utenteDao.insert(utente);
        Utente utente1 = utenteDao.findUtenteUsername("Francesco","F123456");
        Utente utente2 = utenteDao.userExist(users);
        boolean find = utenteDao.doesUserExist("Francesco","F123456");
        assertNotNull(utente1);
        assertNotNull(utente2);
        assertEquals("Francesco",utente1.getUsername());
        assertEquals("F123456",utente1.getPassword());
        assertTrue(find);
    }
    @Test
    public void delete(){
        utenteDao.insert(utente);
        Utente utente1 = utenteDao.findUtenteUsername("Francesco","F123456");
        assertNotNull(utente1);
        utenteDao.delete(utente1);
        Utente utente2 = utenteDao.findUtenteUsername("Francesco","F123456");
        assertNull(utente2);

    }
```

Listing 27: Insert and Delete Utente Test 8

```java
    @Test
    public void update(){
        utenteDao.insert(utente);
        utenteDao.updatePassword("Francesco","A123456");
        assertEquals("A123456",utente.getPassword());
    }
```

Listing 28: Update Password Test 9

```
    @Test
    public void testSearch(){
        userDao.insert(user);
        dao.insert(admin);
        Admin controlla = dao.findAdminByUsernameAndPassword("A123456","123456");
        assertNotNull(controlla);
        assertEquals(controlla.getUsername(),"A123456");
        assertEquals(controlla.getPassword(),"123456");
    }
```

Listing 29: Search Admin Test 10

```
    @Test
    public void search(){
        utenteDao.insert(utente);
        Utente utente1 = utenteDao.findUtenteUsername("Francesco","F123456");
        assertNotNull(utente1);
        assertEquals("Francesco",utente1.getUsername());
        assertEquals("F123456",utente1.getPassword());
    }
```

Listing 30: Search Utente Test 11

6.3 Test di Domain Model

I test di domain model, nell'ambito dello sviluppo software, si riferiscono a test mirati a verificare la correttezza e la coerenza del modello di dominio di un'applicazione. In questo progetto i test di Domain si sono concentrati sul "Costruttore" e sui metodi "Get" and "Set". In seguito un esempio dei Test di Domain Model.

```
    @TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class AdminTest {
    @Test
    public void test() {
        Users user = new Users();
        Admin admin = new Admin();
        admin.setUsername("admin");
        admin.setPassword("admin");
        admin.setIdBiblotecca("admin");
        admin.setUsers(user);

        assertEquals("admin", admin.getUsername());
        assertEquals("admin", admin.getPassword());
        assertEquals("admin", admin.getIdBiblotecca());
        assertEquals(user, admin.getUsers());
    }

    @Test
    public void test2() {
        Users user = new Users();
        Admin admin = new Admin("admin","admin","admin",user);
        assertEquals("admin", admin.getUsername());
        assertEquals("admin", admin.getPassword());
        assertEquals("admin", admin.getIdBiblotecca());
        assertEquals(user, admin.getUsers());
    }
}
```