Scuola di Ingegneria
Corso di Laurea in Ingegneria Informatica

Progetto di Ingegneria del Software

Applicazione per la gestione di prestiti bibliotecari

BERNACCHIONI LAPO, FALLANI COSIMO

Anno Accademico 2024-2025

Bernacchioni Lapo, Fallani Cosimo : *Applicazione per la gestione di prestiti bibliotecari* , Ingegneria del Software , © Anno Accademico 2024-2025

CONTENTS

1

INTRODUZIONE

1.1 OBIETTIVI DEL PROGETTO

Il presente progetto ha l'obiettivo di sviluppare un sistema di gestione delle prenotazioni per una biblioteca, consentendo agli utenti di effettuare prenotazioni, monitorare lo stato dei libri, prendere in prestito e restituire volumi in modo automatizzato ed efficiente. Il sistema si propone di ridurre il carico gestionale del personale bibliotecario, garantendo un'organizzazione chiara delle prenotazioni e un controllo dettagliato delle operazioni di prestito e restituzione.

Gli utenti registrati possono visualizzare la disponibilità dei libri, effettuare una prenotazione e monitorarne lo stato fino alla restituzione. Il sistema prevede inoltre un controllo sulle date di scadenza, consentendo di identificare prestiti scaduti e notificare gli utenti per la restituzione. Il personale bibliotecario ha la possibilità di gestire il catalogo, aggiornare lo stato dei libri, approvare prenotazioni e verificare la regolarità delle restituzioni.

L'architettura software è strutturata secondo il paradigma Model-View-Controller (MVC), garantendo una chiara separazione tra interfaccia utente, logica applicativa e gestione dei dati. Il sistema implementa il Data Access Object (DAO) Pattern, che assicura un accesso ai dati strutturato e indipendente dall'implementazione del database, facilitando eventuali future estensioni o migrazioni.

1.2 ELEMENTI DEL PROGETTO

Il sistema di gestione delle prenotazioni per la biblioteca è strutturato attorno a tre entità principali: Utenti (User), libri (Item) e Prenotazioni (Reservation). Queste componenti interagiscono tra loro per garantire la corretta gestione del prestito e della restituzione dei libri.

- **Utenti**: rappresenta una persona registrata nel sistema, che può assumere diversi ruoli:
  - **User**: può effettuare prenotazioni, consultare lo stato dei propri prestiti e ricevere notifiche relative alle scadenze.
  - **Staff**: ha il compito di gestire e monitorare le prenotazioni attive, verificare le scadenze e registrare la restituzione dei libri. Può anche annullare prenotazioni in casi specifici e segnalare eventuali problemi legati ai prestiti.
  - **Manager**: è responsabile della gestione del catalogo, quindi può aggiungere, modificare o rimuovere libri dal sistema.
- **Item**: rappresenta un libro o un altro materiale disponibile per il prestito. Ogni libro è identificato da un codice univoco e possiede informazioni come titolo, autore, genere e stato.
- **Reservation**: è l'oggetto attraverso cui un utente richiede un libro in prestito. La prenotazione include informazioni come la data di inizio, la data di scadenza e lo stato attuale. Il sistema gestisce automaticamente la scadenza delle prenotazioni e notifica gli utenti in caso di ritardi o necessità di restituzione.

1.3 ESTENDIBILITÀ DEL PROGETTO

Il progetto è stato concepito per essere facilmente estendibile, in particolare per quanto riguarda l'integrazione di nuove funzionalità relative alla gestione delle prenotazioni e al controllo del catalogo bibliotecario. Una delle principali possibilità di estensione riguarda l'aggiunta di nuove tipologie di materiali prenotabili, come e-book o audiolibri, che richiederebbero l'implementazione di un sistema di accesso digitale e gestione delle licenze.

Un'ulteriore espansione del sistema potrebbe includere la gestione delle prenotazioni interbibliotecarie, consentendo agli utenti di prenotare libri da più sedi collegate allo stesso database e implementando un sistema di trasferimento tra biblioteche. A questa estensione si collega anche la possibilità di visualizzare una disponibilità globale degli elementi, offrendo un sistema di ricerca avanzato che tenga conto delle copie disponibili in tutte le biblioteche della rete.

Infine una espansione necessaria sarebbe una interfaccia che permetta l'assegnazione dei ruoli a gli utenti nel database da parte dei manager.

2

ANALISI DEI REQUISITI

2.1 USE CASE DIAGRAM

Il software presenta tre diversi tipi di attori: l'Utente, lo Staff e il Manager, inoltre sono presenti delle funzioni per registrarsi o autenticarsi per i guest. L'Utente può effettuare prenotazioni, visualizzare la disponibilità dei libri e gestire i propri prestiti. Lo Staff si occupa della gestione delle prenotazioni, confermando le richieste e registrando le restituzioni. Il Manager, invece, è responsabile della gestione del database, potendo aggiungere, modificare o rimuovere libri. Lo schema dei casi d'uso sottostante è stato realizzato mediante StarUML.

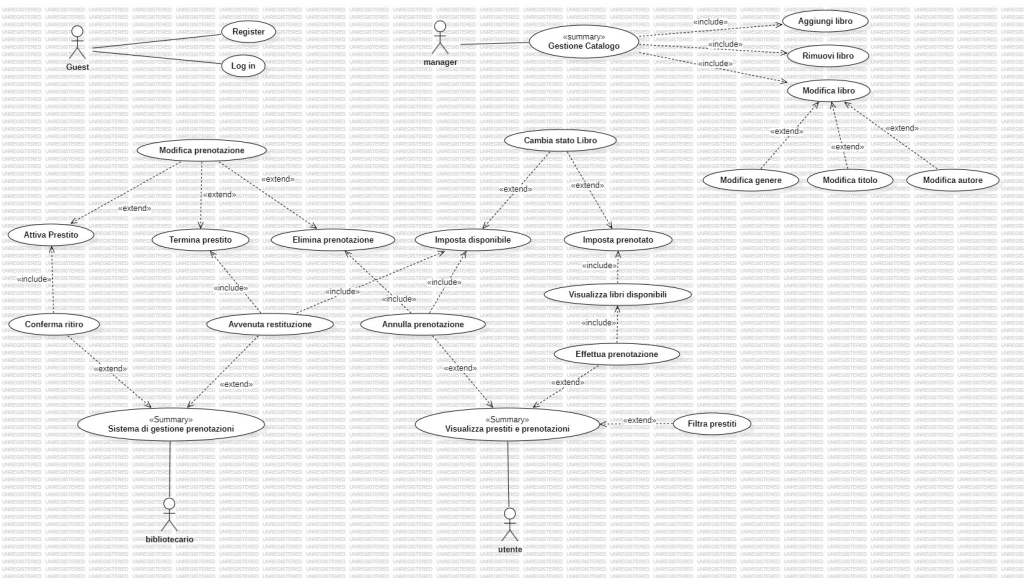

Figure 1: Use Case Diagram

2.2 USE CASE TEMPLATES

2.2.1 *Casi d'uso principali di uno User*

UC-1	**Prenotazione di un libro**
Descrizione	Lo user seleziona un libro disponibile dal catalogo e procede alla prenotazione. Il sistema associa il libro all'utente e ne aggiorna lo stato a "prenotato".
Livello	User goal
Attore principale	User
Azioni	1. Lo user accede alla dashboard personale. 2. Visualizza l'elenco dei libri disponibili. 3. Seleziona un libro e conferma la prenotazione. 4. Il sistema crea la prenotazione, associa il libro all'utente e ne aggiorna lo stato a "prenotato".
Casi straordinari	2a. Se sono stati impostati dei filtri di ricerca nella dashboard i libri visualizzati rispetteranno i filtri.

Table 1: Prenotazione di un libro da parte dello User

UC-2	**Annulla prenotazione di un libro**
Descrizione	Lo user può annullare una prenotazione effettuata in precedenza. Il sistema elimina la prenotazione associata e rende nuovamente disponibile il libro.
Livello	User goal
Attore principale	User
Azioni	1. Lo user accede alla dashboard personale. 2. Visualizza l'elenco delle prenotazioni attive. 3. Seleziona la prenotazione da annullare. 4. Il sistema elimina la prenotazione e aggiorna lo stato del libro a "disponibile", se la prenotazione non è ancora attiva.
Casi straordinari	4a. Se la prenotazione è già attiva (il libro è stato consegnato all'utente), il sistema non consente l'annullamento.

Table 2: Annullamento di una prenotazione da parte dello User

2.2.2 *Casi d'uso principali degli staffer*

UC-3	**Conferma ritiro di un libro**
Descrizione	Lo staff conferma l'effettivo ritiro del libro da parte dell'utente, attivando ufficialmente la prenotazione. Il sistema registra l'inizio del periodo di prestito.
Livello	User goal
Attore principale	Staff
Azioni	1. Lo staff accede alla dashboard dedicata. 2. Seleziona l'utente che ha effettuato la prenotazione. 3. Visualizza l'elenco delle prenotazioni gia attive, di quelle non ancora attive e di quelle concluse . 4. Clicca sul bottone dedicato relativo alla prenotazione da attivare. 5. Il sistema registra l'inizio della prenotazione e imposta lo stato su "attiva".
Casi straordinari	4a. Se la prenotazione è già attiva o risulta scaduta, il sistema non consente l'attivazione. In tal caso il bottone é disattivato.

Table 3: Conferma del ritiro di un libro da parte dello Staff

UC-4	**Avvenuta restituzione di un libro**
Descrizione	Lo staff registra la restituzione del libro da parte dell'utente. Il sistema chiude la prenotazione e rende nuovamente disponibile il libro.
Livello	User goal
Attore principale	Staff
Azioni	1. Lo staff accede alla dashboard dedicata. 2. Seleziona l'utente che ha effettuato la prenotazione. 3. Visualizza l'elenco delle prenotazioni gia attive, di quelle non ancora attive e di quelle concluse . 4. Clicca sul bottone dedicato relativo alla prenotazione da chiudere. 5. Il sistema disattiva la prenotazione e imposta lo stato del libro su "disponibile".
Casi straordinari	4a. Se la prenotazione non è già attiva o é ancora in corso, il sistema non consente la chiusura. In tal caso il bottone é disattivato.

Table 4: Registrazione della restituzione di un libro da parte dello Staff

2.2.3 *Casi d'uso principali dei manager*

UC-5	**Aggiunta di un libro al database**
Descrizione	Il manager inserisce un nuovo libro nel catalogo della biblioteca compilando i campi richiesti. Il sistema salva il libro nel database rendendolo disponibile alla prenotazione.
Livello	User goal
Attore principale	Manager
Azioni	1. Il manager accede alla dashboard di gestione. 2. Compila il modulo di inserimento con i dati del libro (titolo, autore, genere, tipo). 3. Conferma l'operazione. 4. Il sistema salva il libro nel database e lo rende disponibile per la prenotazione.
Casi straordinari	2a. Se sono assenti dei dati obbligatori non é consentita la creazione del libro.

Table 5: Aggiunta di un libro al database da parte del Manager

UC-6	**Rimozione di un libro dal database**
Descrizione	Il manager può rimuovere un libro dal catalogo della biblioteca. Il sistema elimina il libro dal database e lo rende non più disponibile alla prenotazione.
Livello	User goal
Attore principale	Manager
Azioni	1. Il manager accede alla dashboard di gestione. 2. Visualizza l'elenco dei libri presenti nel catalogo. 3. Seleziona il libro da rimuovere. 4. Il sistema elimina il libro dal database.
Casi straordinari	

Table 6: Rimozione di un libro dal database da parte del Manager

UC-7	**Modifica di un libro nel database**
Descrizione	Il manager può aggiornare le informazioni di un libro presente nel catalogo, modificando titolo, autore o genere. Il sistema salva le nuove informazioni nel database.
Livello	User goal
Attore principale	Manager
Azioni	1. Il manager accede alla dashboard di gestione. 2. Visualizza l'elenco dei libri presenti nel catalogo. 3. Seleziona il libro da modificare. 4. Inserisce i nuovi dati nel modulo di modifica. 5. Conferma le modifiche. 6. Il sistema aggiorna le informazioni del libro nel database.
Casi straordinari	5a. Se sono assenti dei dati obbligatori non é consentita la modifica del libro.

Table 7: Modifica di un libro nel database da parte del Manager

2.3 MOCKUPS

Ecco alcuni Mockups che sono veri e propri screenshot della GUI realizzata a questo proposito.

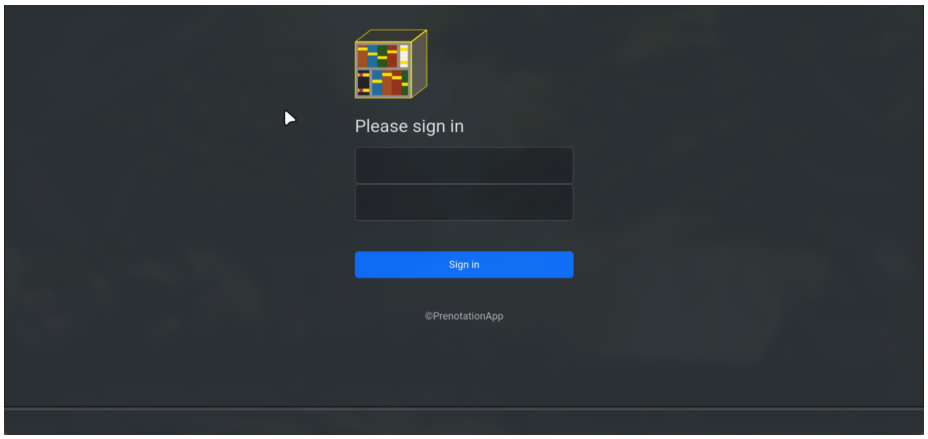

Figure 2: Pagina di login

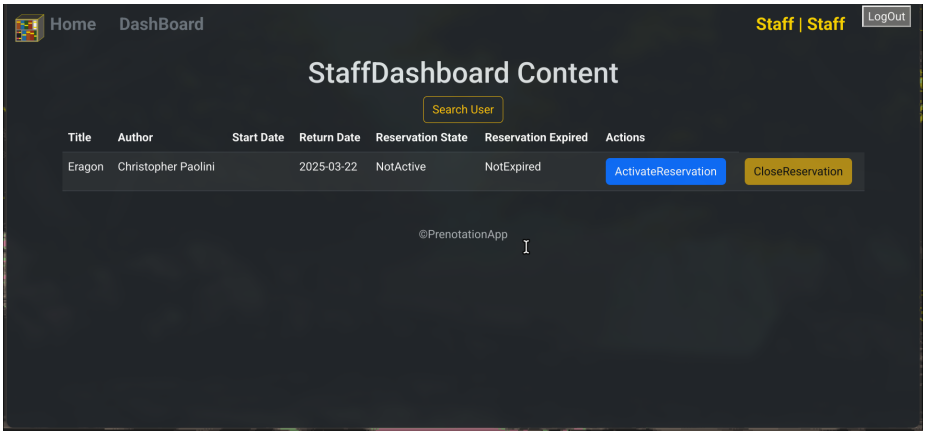

Figure 3: Dashboard dello staff

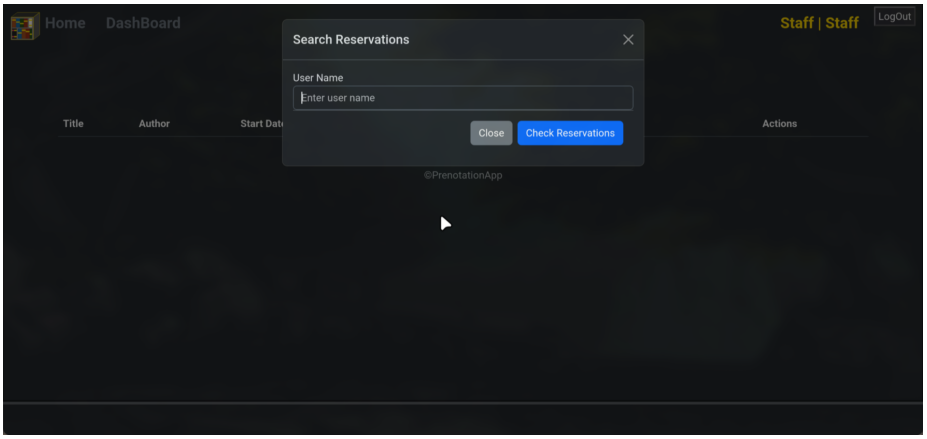

Figure 4: Barra di ricerca per l'attivazione dei prestiti

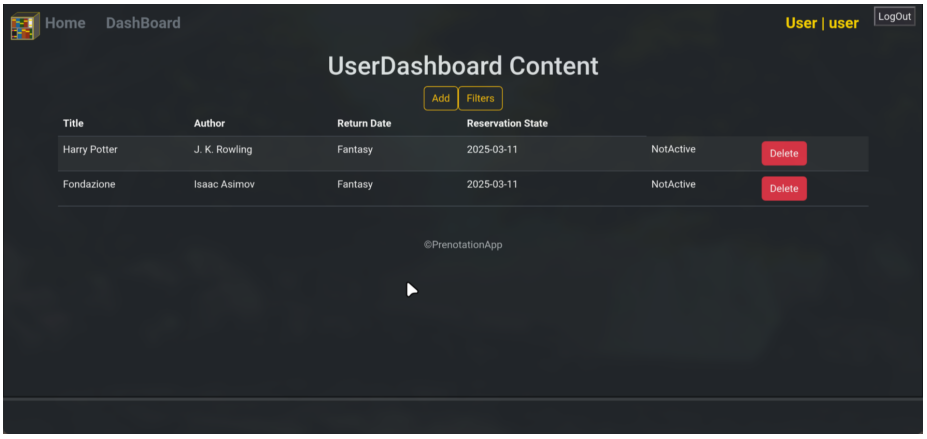

Figure 5: Dashboard degli utenti

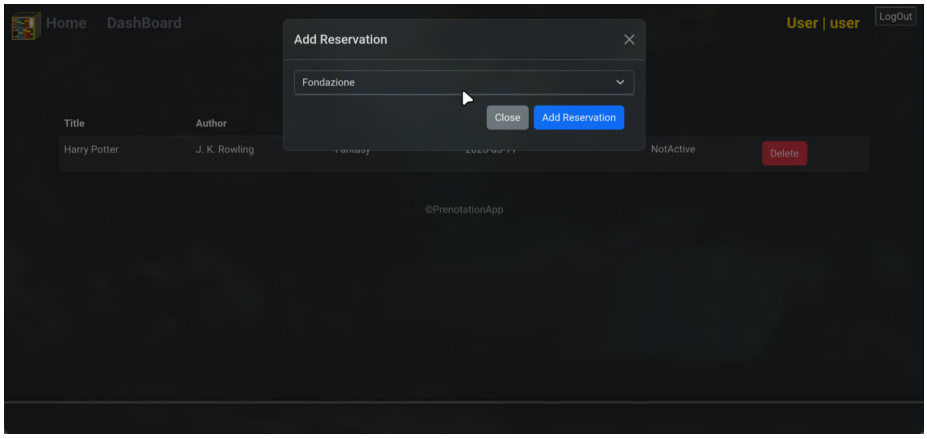

Figure 6: Aggiunta di una prenotazione

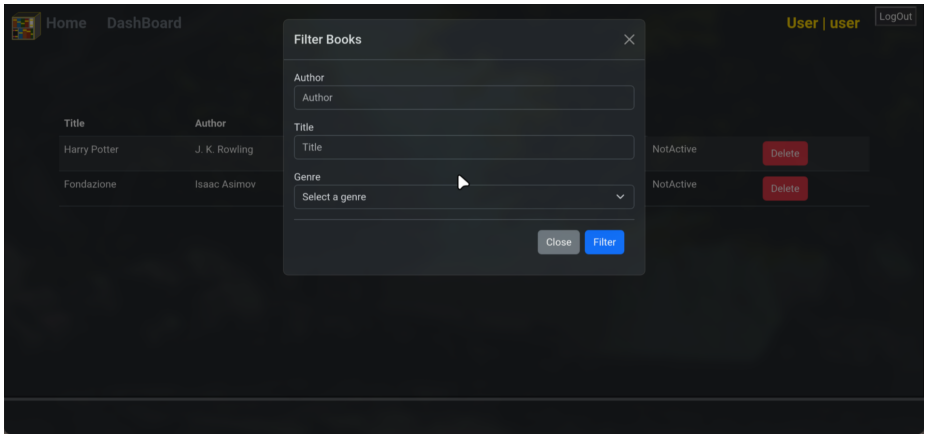

Figure 7: Form per il filtraggio dei libri

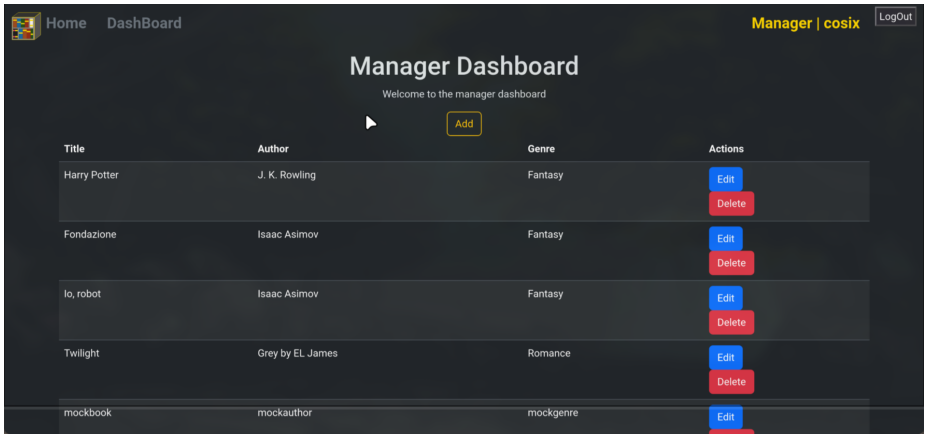

Figure 8: Dashboard del manager

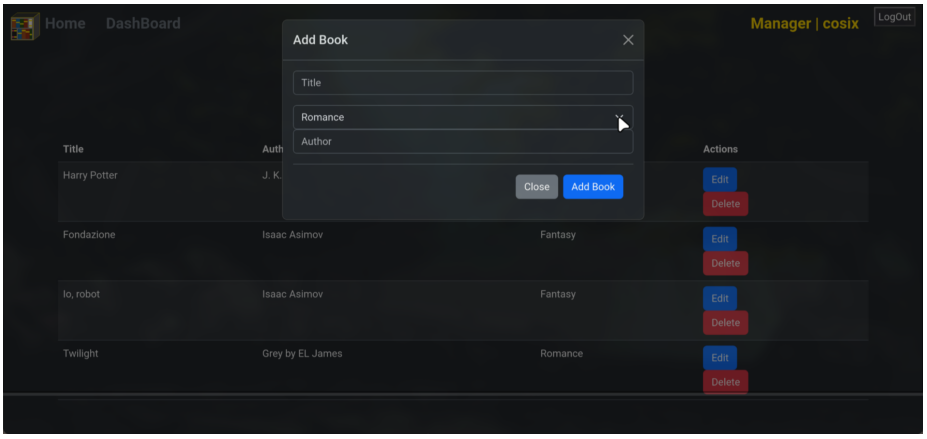

Figure 9: Form per l'aggiunta di un libro

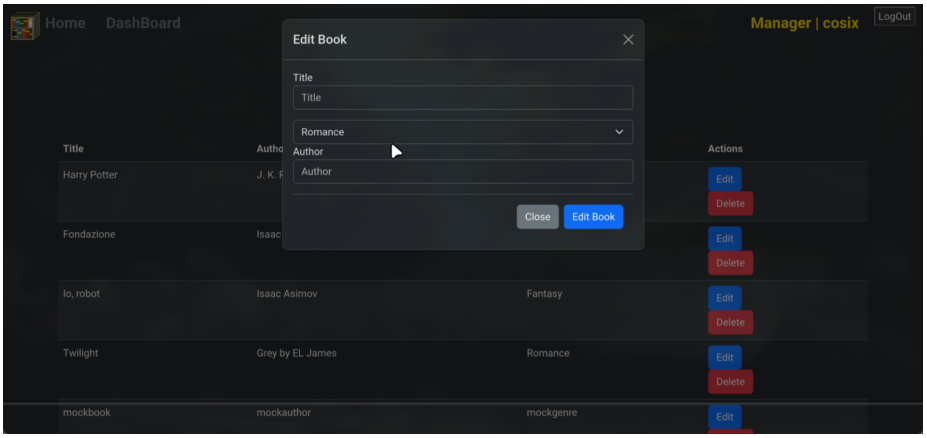

Figure 10: Form per la modifica di un libro

2.4 CLASS DIAGRAM

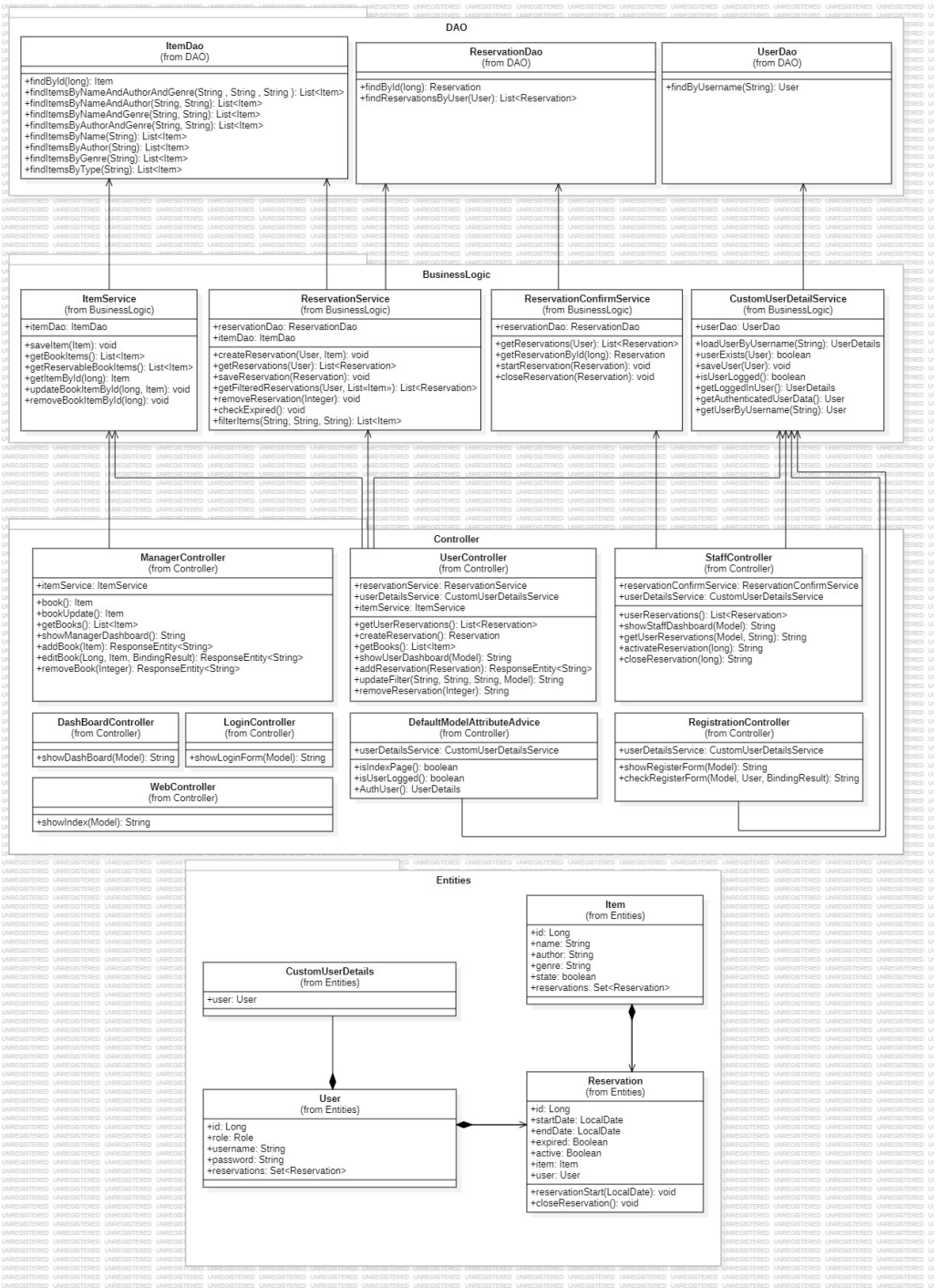

Figure 11: Use Case Diagram

Il codice del sistema è organizzato in diversi package, ognuno con responsabilità specifiche per garantire modularità, separazione dei task e manutenibilità del software. Il Class Diagram evidenzia la suddivisione del software nei livelli DAO, Business Logic, Controller e Entities.

- Entities: definisce le entità fondamentali del sistema, rappresentando gli oggetti principali come utenti, oggetti e prenotazioni. Ogni entità è modellata con attributi e relazioni che ne descrivono il comportamento all'interno del sistema.
- Business Logic: implementa la logica applicativa del sistema, fungendo da livello intermedio tra il DAO e i controller. Qui vengono definite le operazioni fondamentali come la gestione delle prenotazioni, l'elaborazione delle richieste e l'autenticazione degli utenti.
- DAO: fornisce le interfacce per l'accesso ai dati, consentendo di isolare la logica di persistenza e garantendo un'interazione efficiente con il database. Questo package include operazioni di ricerca, recupero e gestione di utenti, prenotazioni e libri.
- Controller : gestisce l'interazione con l'utente, elaborando le richieste e restituendo risposte appropriate. I controller si occupano delle richieste ai servizi offerti dal package buisness logic, come prenotazioni e modifica del catalogo bibliotecario.

È importante sottolineare che le classi presenti nel package DAO non rappresentano un'implementazione tradizionale del DAO Pattern, ma fungono da wrapper per le funzionalità di accesso ai dati fornite da Spring Boot. Poiché il framework offre già un'astrazione per la gestione del database, le nostre classi DAO sono state progettate per migliorare la separazione delle responsabilità e garantire una maggiore chiarezza strutturale. L'adozione di questo approccio consente di mantenere un'architettura modulare e facilita eventuali modifiche future.

2.5 DAO PATTERN

Abbiamo utilizzato il Design Pattern **DAO** (Data Access Object) per garantire che il client non possa interagire direttamente con il database.

Il pattern architetturale **DAO** viene impiegato per gestire la persistenza dei dati registrati nel database. La persistenza si riferisce alla capacità di un dato di "sopravvivere" all'esecuzione di un programma, mantenendo la coerenza anche dopo la terminazione del software. Questo pattern ha

lo scopo di separare la logica di accesso ai dati dal resto dell'applicazione, isolando il codice responsabile della manipolazione dei dati da quello che implementa la logica di elaborazione.

Nel nostro progetto, il DAO definisce e offre metodi CRUD (Create, Read, Update, Delete) per la gestione dei dati nel database. Le principali classi che sfruttano il DAO e che sono presenti nel database sono le seguenti:

- **UserDAO** → rappresenta gli utenti del sistema con i relativi dati di autenticazione e ruolo;
- **ItemDAO** → rappresenta gli oggetti disponibili per il prestito;
- **ReservationDAO** → gestisce le prenotazioni effettuate dagli utenti sui libri disponibili.

2.6 MVC PATTERN

Il pattern **MVC** è un'architettura software che separa i dati (**Model**), l'interfaccia utente (**View**) e la logica di controllo (**Controller**), garantendo una netta separazione delle responsabilità.

Nel caso del nostro sistema, la suddivisione del pattern **MVC** è la seguente:

- Il **Model** è rappresentato dalle classi presenti nel package `entities`, che definiscono i dati memorizzati nel database.
- La **View** corrisponde alle pagine HTML e alle risorse dell'interfaccia utente con cui l'utente interagisce.
- Il **Controller** è implementato nelle classi del package `controller`, che gestiscono le richieste dell'utente e tramite i servizi offerti dal package buisness logic aggiorna il modello di dati attraverso il pattern DAO.

2.7 ENTITY RELATIONSHIP DIAGRAM

Questo è il diagramma ER del progetto, con le varie cardinalità delle relazioni.

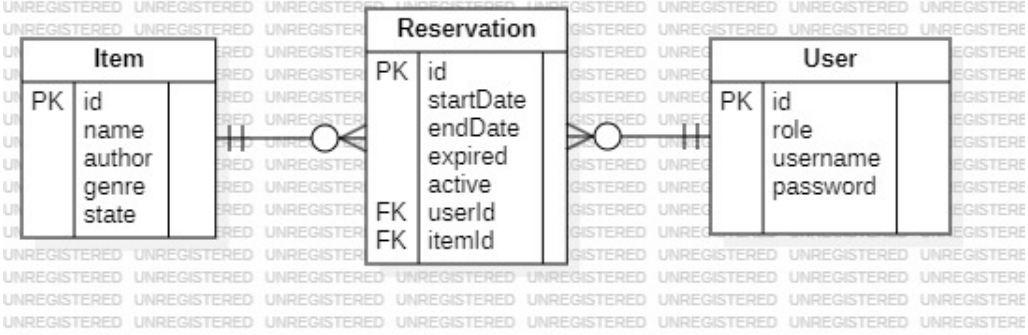

Figure 12: Use Case Diagram

3

IMPLEMENTAZIONI DELLE CLASSI

L'implementazione del sistema è organizzata in diversi package, suddivisi per garantire modularità, separazione delle responsabilità e manutenibilità del codice. Le classi sono raggruppate nei package **entities**, **dao**, **business logic** e **controller**, ciascuno con una funzione specifica.

3.1 ENTITIES

3.1.1 *User*

La classe `User` rappresenta un utente del sistema, contenendo informazioni relative all'autenticazione e al ruolo assegnato. Ogni utente può effettuare prenotazioni e avere differenti permessi a seconda del suo ruolo.

Attributi:

- `id: Long` → Identificativo univoco dell'utente.
- `role: Role` → Ruolo dell'utente nel sistema (`user`, `manager`, `staff`).
- `username: String` → Nome utente utilizzato per l'accesso, deve essere unico e non nullo.
- `password: String` → Password dell'utente, con una lunghezza compresa tra 3 e 60 caratteri.
- `reservations: Set<Reservation>` → Insieme delle prenotazioni effettuate dall'utente.

3.1.2 *CustomUserDetails*

La classe `CustomUserDetails` implementa l'interfaccia `UserDetails` di Spring Security, permettendo la gestione dell'autenticazione degli utenti e dei loro permessi all'interno del sistema.

Attributi:

- `user: User` → Istanza della classe `User` associata all'utente autenticato.

Metodi:

- `getAuthorities(): Collection<? extends GrantedAuthority>`
  - Restituisce una collezione di permessi (`GrantedAuthority`) assegnati all'utente.
  - Il ruolo dell'utente viene trasformato in un'autorità di Spring Security con il prefisso `"ROLE_"`.
- `capitalizeFirstLetter(String original): String`
  - Metodo di utilità che capitalizza la prima lettera di una stringa, trasformando il resto in lettere minuscole.
  - Utilizzato per formattare il ruolo dell'utente in modo leggibile.

3.1.3 *Item*

La classe `Item` rappresenta un libro prenotabile all'interno del sistema, questo puó essere associato a una o più prenotazioni.

Attributi:

- `id: Long` → Identificativo univoco dell'elemento.
- `type: Type` → Tipo dell'elemento.
- `name: String` → Nome dell'elemento. Non può essere vuoto.
- `author: String` → Autore del libro.
- `genre: String` → Genere del libro.
- `state: boolean` → Indica se il libro è attualmente prenotato (`true`) o disponibile (`false`).
- `reservations: Set<Reservation>` → Insieme delle prenotazioni associate all'elemento.

3.1.4 *Reservation*

La classe `Reservation` rappresenta una prenotazione effettuata da un utente su un libro. Ogni prenotazione ha una data di inizio, una data di fine e può essere attiva o scaduta.

Attributi:

- `id: Long` → Identificativo univoco della prenotazione.
- `startDate: LocalDate` → Data di inizio della prenotazione.
- `endDate: LocalDate` → Data di scadenza della prenotazione (di default, 5 giorni dalla creazione).
- `expired: Boolean` → Indica se la prenotazione è scaduta.
- `active: Boolean` → Indica se la prenotazione è attiva.
- `item: Item` → libro associato alla prenotazione.
- `user: User` → Utente che ha effettuato la prenotazione.

Metodi:

- `reservationStart(): void`
  - Imposta la data di inizio della prenotazione al giorno corrente.
  - Attiva la prenotazione.
  - Estende la data di scadenza della prenotazione di un mese dalla data iniziale.
- `closeReservation(): void`
  - Disattiva la prenotazione.
  - Cambia lo stato del libro associato, rendendolo disponibile.

3.2 DAO (DATA ACCESS OBJECT)

L'interfacce presenti in questo package estendono `CrudRepository` e forniscono metodi per l'accesso e la gestione dei libri, delle prenotazioni e degli utenti nel database. Queste interfacce vengono automaticamente implementate da Spring, permettendo l'esecuzione di operazioni CRUD (Create, Read, Update, Delete) senza necessità di una specifica implementazione manuale.

3.2.1 *ItemDao*

Metodi:

- `findById(long id): Item`
  - Restituisce il libro corrispondente all'ID specificato.
- `findByNameAndAuthorAndGenre(String name, String author, String genre): List<Item>`
  - Restituisce una lista di libri corrispondenti al nome, autore e genere specificati.
- `findByNameAndAuthor(String name, String author): List<Item>`
  - Restituisce una lista di libri corrispondenti al nome e autore specificati.
- `findByNameAndGenre(String name, String genre): List<Item>`
  - Restituisce una lista di libri corrispondenti al nome e genere specificati.
- `findByAuthorAndGenre(String author, String genre): List<Item>`
  - Restituisce una lista di libri corrispondenti all'autore e genere specificati.
- `findByName(String name): List<Item>`
  - Restituisce una lista di libri corrispondenti al nome specificato.
- `findByAuthor(String author): List<Item>`
  - Restituisce una lista di libri scritti dall'autore specificato.
- `findByGenre(String genre): List<Item>`
  - Restituisce una lista di libri appartenenti al genere specificato.
- `findByState(boolean state): List<Item>`
  - Restituisce una lista di libri in base al loro stato (`true` per prenotato, `false` per disponibile).

3.2.2 *ReservationDao*

Metodi:

- `findById(long id): Reservation`
  - Restituisce la prenotazione corrispondente all'ID specificato.
- `findByUser(User user): List<Reservation>`
  - Restituisce una lista di prenotazioni effettuate da un determinato utente.

3.2.3 *UserDao*

Metodi:

- `findByUsername(String username): User`
  - Restituisce un utente basandosi sul suo nome utente.
  - Utilizzato per il processo di autenticazione e gestione degli utenti.

3.3 BUSINESS LOGIC

Il package **business logic** implementa la logica del sistema, interagendo con i DAO per elaborare le operazioni richieste dai controller.

3.3.1 *ItemService*

La classe `ItemService` fornisce i metodi per la gestione dei libri, consentendo operazioni di creazione, aggiornamento, eliminazione e recupero di oggetti dal database.

Metodi:

- `saveItem(Item item): void`
  - Salva un nuovo libro nel database utilizzando `ItemDao`.
- `getBookItems(): List<Item>`
  - Restituisce una lista di tutti i libri presenti nel database.
- `getReservableBookItems(): List<Item>`

  - Restituisce una lista di libri disponibili per la prenotazione, ovvero quelli il cui stato è impostato su `false` (non prenotato).

- `getItemById(long id): Item`
  - Recupera un libro dal database in base al suo identificativo univoco.

- `updateBookItemById(long id, Item updatedItem): void`
  - Aggiorna i dati di un libro esistente in base all'ID fornito.
  - Se il libro non esiste, lancia un'eccezione di tipo `IllegalArgumentException`.
  - Modifica il nome, l'autore e il genere del libro e lo salva nel database.

- `removeBookItemById(Integer id): void`
  - Elimina un libro dal database in base all'ID fornito.

La classe `ItemService` centralizza la gestione dei libri e interagisce con il `ItemDao` per garantire la persistenza dei dati nel database.

3.3.2 *ReservationService*

La classe `ReservationService` fornisce i metodi per la gestione delle prenotazioni, consentendo la creazione, la rimozione e il filtraggio delle prenotazioni effettuate dagli utenti. Inoltre, verifica automaticamente le prenotazioni scadute ed elimina quelle non più valide.

Metodi:

- `createReservation(User user, Item item): void`
  - Modifica lo stato del libro prenotato, impostandolo come occupato.
  - Crea una nuova prenotazione associata all'utente e al libro selezionato.
  - Salva la prenotazione nel database.

- `getReservations(User user): List<Reservation>`
  - Restituisce la lista delle prenotazioni effettuate da un determinato utente.

- `getFilteredReservations(User user, List<Item> itemlist): List<Reservation>`

  - Filtra le prenotazioni di un utente in base alla lista di libri fornita.
  - Restituisce solo le prenotazioni che corrispondono ai libri nella lista.

- `removeReservation(Integer id): void`
  - Recupera una prenotazione dal database tramite il suo ID.
  - Se la prenotazione non è attiva né scaduta, libera il libro associato e la elimina dal database.

- `checkExpired(): void`
  - Recupera tutte le prenotazioni nel database.
  - Controlla se la data di scadenza di ogni prenotazione è superata.
  - Se una prenotazione attiva è scaduta, la imposta come `expired`.
  - Se una prenotazione non attiva è scaduta, la elimina dal database.

- `filterItems(String genre, String author, String title): List<Item>`
  - Filtra i libri in base a genere, autore e titolo.
  - Se tutti i parametri sono nulli, restituisce l'elenco completo dei libri disponibili.
  - Se almeno uno dei parametri è presente, esegue una ricerca mirata tramite i metodi di `ItemDao`.

3.3.3 *ReservationConfirmService*

La classe `ReservationConfirmService` fornisce i metodi per la gestione e la conferma delle prenotazioni effettuate dagli utenti. Consente di recuperare le prenotazioni, avviarne il periodo di validità e chiuderle in caso di scadenza.

Metodi:

- `getReservations(User user): List<Reservation>`
  - Recupera la lista delle prenotazioni effettuate da un determinato utente.

- `getReservationById(long id): Reservation`

  - Restituisce una prenotazione specifica basandosi sul suo ID.
  - Se la prenotazione non esiste, lancia un'eccezione ChangeSetPersister.NotFoundException.

- `startReservation(Reservation reservation): void`
  - Attiva una prenotazione se non è già attiva e non è scaduta.
  - Chiama il metodo `reservationStart()` per avviare il periodo di validità della prenotazione.
  - Salva lo stato aggiornato della prenotazione nel database.

- `closeReservation(Reservation reservation): void`
  - Chiude una prenotazione se è scaduta.
  - Chiama il metodo `closeReservation()` per rilasciare il libro prenotato.

La classe `ReservationConfirmService` è utilizzata dallo staff per la gestione operativa delle prenotazioni, garantendo il corretto stato dei libri prenotati nel sistema.

3.3.4 *CustomUserDetailsService*

La classe `CustomUserDetailsService` implementa l'interfaccia `UserDetailsService` di Spring Security e fornisce metodi utili per la gestione degli utenti nel sistema. Questa classe consente di recuperare i dati degli utenti dal database, verificare la loro autenticazione e salvare nuovi utenti con password crittografate.

Metodi:

- `loadUserByUsername(String username): UserDetails`
  - Recupera un utente dal database basandosi sul nome utente fornito.
  - Se l'utente non viene trovato, lancia un'eccezione `UsernameNotFoundException`.
  - Restituisce un oggetto `CustomUserDetails` che rappresenta l'utente autenticato nel sistema.

- `userExists(User user): boolean`
  - Controlla se un utente esiste nel database verificando la presenza del nome utente.

  - Restituisce `true` se l'utente esiste, `false` altrimenti.

- `saveUser(User user): void`
  - Crittografa la password dell'utente utilizzando `BCryptPasswordEncoder`.
  - Salva l'utente nel database con la password crittografata.

- `isUserLogged(): boolean`
  - Verifica se un utente è attualmente autenticato nel sistema.
  - Controlla il contesto di sicurezza di Spring Security e restituisce `true` se l'utente è loggato, `false` altrimenti.

- `getLoggedInUser(): UserDetails`
  - Restituisce i dettagli dell'utente attualmente autenticato.
  - Se nessun utente è autenticato, restituisce `null`.

- `getAuthenticatedUserData(): User`
  - Restituisce l'entità `User` dell'utente attualmente autenticato, recuperandola dal database basandosi sul nome utente.

- `getUserByUsername(String username): User`
  - Recupera un utente dal database basandosi sul nome utente fornito.

La classe `CustomUserDetailsService` è essenziale per la gestione degli utenti e dell'autenticazione all'interno del sistema, integrando Spring Security per garantire la protezione delle credenziali e l'accesso sicuro

3.4 CONTROLLER

Il package **controller** gestisce le interazioni tra utente e sistema.

3.4.1 *DashBoardController*

La classe `DashBoardController` gestisce la visualizzazione della dashboard dell'utente autenticato. In base al ruolo dell'utente, l'accesso viene reindirizzato alla dashboard corrispondente.

Metodi:

- `showDashBoard(Model model): String` → Determina la dashboard a cui reindirizzare l'utente autenticato in base al suo ruolo.

  - Se l'utente ha il ruolo `USER`, viene reindirizzato a `/user-dashboard`.
  - Se l'utente ha il ruolo `MANAGER`, viene reindirizzato a `/manager-dashboard`.
  - Se l'utente ha il ruolo `STAFF`, viene reindirizzato a `/staff-dashboard`.

3.4.2 *DefaultModelAttributeAdvice*

La classe `DefaultModelAttributeAdvice` fornisce attributi di default ai modelli dei controller dell'applicazione. È annotata con `@ControllerAdvice`, il che significa che i metodi definiti al suo interno vengono eseguiti automaticamente per tutti i controller dell'applicazione, senza la necessità di definirli manualmente in ciascuno di essi. Questa classe centralizza la gestione di attributi comuni nei modelli dei controller, riducendo la duplicazione di codice e migliorando la manutenibilità dell'applicazione.

3.4.3 *LoginController*

La classe `LoginController` gestisce la visualizzazione del modulo di login per gli utenti.

Metodi:

- `showLoginForm(Model model): String` → Gestisce la richiesta `GET` all'endpoint `/login`.
  - Aggiunge un oggetto `User` al modello per essere utilizzato nella vista del modulo di login.
  - Restituisce il nome della vista `"login"`, che corrisponde alla pagina HTML dedicata all'autenticazione degli utenti.

Questa classe fornisce il punto di accesso per il login degli utenti, preparando il modello con un'istanza di `User` che verrà popolata con i dati inseriti dall'utente nella pagina di autenticazione.

3.4.4 *ManagerController*

La classe `ManagerController` gestisce le operazioni disponibili per gli utenti con ruolo di `Manager`, consentendo la visualizzazione della dashboard di gestione e la manipolazione dei libri, come l'aggiunta, la modifica e la rimozione di libri dal catalogo.

Metodi:

- `showManagerDashboard(): String` → Gestisce la richiesta `GET` all'endpoint `/manager-dashboard`, restituendo la vista corrispondente alla dashboard del manager.
- `addBook(@ModelAttribute("book") Item book): ResponseEntity<String>`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-book-add`.
  - Aggiunge un nuovo libro al catalogo tramite il servizio `ItemService`.
  - Restituisce una risposta *"Book added successfully"* in caso di successo.
- `editBook(@PathVariable("id") Long id, @Valid @ModelAttribute("bookUpdate Item bookUpdate, BindingResult bindingResult): ResponseEntity<String>`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-book-edit/{id}` per aggiornare i dati di un libro esistente.
  - Valida i dati dell'oggetto `Item` ricevuto.
  - Se la validazione ha successo, aggiorna il libro tramite il servizio `ItemService` e restituisce una risposta con il messaggio *"Book updated successfully"*.
- `removeBook(@PathVariable("id") Integer id): ResponseEntity<String>`
  - Gestisce la richiesta `DELETE` all'endpoint `/dashboard-book-delete/{id}` per rimuovere un libro dal catalogo.
  - Se l'operazione ha successo, restituisce una risposta con il messaggio *"Book deleted successfully"*.
  - Se si verifica un errore durante l'eliminazione, restituisce una risposta con il messaggio *"Error deleting the book"*.

3.4.5 *RegistrationController*

La classe `RegistrationController` gestisce il processo di registrazione degli utenti nel sistema, fornendo la vista del modulo di registrazione e validando i dati inseriti prima di salvarli nel database.

Metodi:

- `showRegisterForm(Model model): String`
  - Gestisce la richiesta `GET` all'endpoint `/register`.
  - Aggiunge un nuovo oggetto `User` al modello, in modo che i dati possano essere inseriti nel modulo di registrazione.

  - Restituisce la vista `"register"`, che contiene il modulo di registrazione.

- `checkRegisterForm(Model model, @ModelAttribute @Valid User user, BindingResult bindingResult): String`
  - Gestisce la richiesta `POST` all'endpoint `/register`, elaborando i dati del modulo di registrazione.
  - Valida l'oggetto `User` ricevuto e, in caso di errori di validazione, restituisce nuovamente la vista `"register"` con i messaggi di errore.
  - Controlla se il nome utente esiste già nel sistema; in tal caso, aggiunge un messaggio di errore e ricarica la pagina di registrazione.
  - Se la registrazione ha successo, salva il nuovo utente nel database e reindirizza alla pagina principale.

3.4.6 *StaffController*

La classe `StaffController` gestisce le operazioni disponibili per i membri dello `Staff`, consentendo loro di visualizzare, attivare e chiudere le prenotazioni effettuate dagli utenti.

Metodi:

- `showStaffDashboard(Model model): String`
  - Gestisce la richiesta `GET` all'endpoint `/staff-dashboard`.
  - Restituisce la vista corrispondente alla dashboard dello `Staff`.

- `getUserReservations(Model model, @RequestParam(name = "searched_user", required = false) String searched_user): String`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-get-user-reservations`.
  - Recupera la lista delle prenotazioni effettuate dall'utente cercato, utilizzando il servizio `ReservationConfirmService`.
  - Aggiunge la lista delle prenotazioni al modello e aggiorna la vista `"staff-dashboard"`.

- `activateReservation(@PathVariable("id") long id): String`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-activate-reservation/{id}`.

  - Recupera la prenotazione corrispondente all'ID fornito e ne avvia il periodo di prestito.
  - Reindirizza alla vista `"staff-dashboard"`.

- `closeReservation(@PathVariable long id): String`
  - Gestisce la richiesta `DELETE` all'endpoint `/dashboard-close-reservation/{id}`.
  - Recupera la prenotazione corrispondente all'ID fornito e ne chiude il periodo di prestito.
  - Reindirizza alla vista `"staff-dashboard"`.

3.4.7 *UserController*

La classe `UserController` gestisce le operazioni disponibili per gli utenti, consentendo loro di visualizzare le proprie prenotazioni, filtrare i libri e aggiungere o rimuovere prenotazioni.

Metodi:

- `showUserDashboard(Model model): String`
  - Gestisce la richiesta `GET` all'endpoint `/user-dashboard`.
  - Restituisce la vista corrispondente alla dashboard dell'utente.

- `addReservation(@ModelAttribute("reservation") Reservation reservation): ResponseEntity<String>`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-reservation-add`.
  - Verifica se il libro è già prenotato, restituendo un messaggio di errore in caso positivo.
  - In caso contrario, crea una nuova prenotazione e la salva nel sistema.
  - Restituisce una risposta con il messaggio *"Reservation added successfully"*.

- `updateFilter(String genre, String author, String title, Model model): String`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-reservation-filter`.
  - Filtra i libri in base a genere, autore e titolo.
  - Aggiorna la lista delle prenotazioni dell'utente in base ai libri filtrati.

  - Restituisce la vista aggiornata della `"user-dashboard"`.

- `removeReservation(@PathVariable("id") Integer id): String`
  - Gestisce la richiesta `POST` all'endpoint `/dashboard-reservation-delete/{id}`.
  - Rimuove la prenotazione con l'ID specificato.
  - Reindirizza alla pagina `/user-dashboard` per evitare la duplicazione delle richieste di eliminazione.

3.4.8 *WebController*

La classe `WebController` gestisce la visualizzazione della pagina principale dell'applicazione e permette di caricare correttamente la homepage dell'applicazione, fornendo un punto di ingresso per gli utenti.

Metodi:

- `showIndex(Model model): String`
  - Gestisce la richiesta `GET` all'endpoint `/`.
  - Aggiunge al modello l'attributo `"isIndexPage"` con valore `true`, utilizzato da `Thymeleaf` per rendere visibile il link di registrazione nella pagina di indice.
  - Restituisce la vista `"index"`.

4

TEST (JUNIT)

Tutti i test sono eseguiti singolarmente, con ripristino dello stato del database dopo ogni esecuzione, così da garantire risultati affidabili e indipendenti.

4.1 MANAGERTEST

Nel contesto del `ManagerTest`, JUnit viene utilizzato per testare il comportamento del controller `ManagerController`, in particolare per verificare che un utente con ruolo *manager* possa correttamente gestire l'aggiunta e la rimozione di libri nel database. Ogni metodo annotato con `@Test` rappresenta un caso di test indipendente.

- **contextLoads**: Il test si basa sull'inizializzazione corretta delle istanze di `UserDao`, `ReservationDao`, `ItemDao` e `ManagerController`. Queste vengono iniettate nel contesto del test per garantire che l'ambiente sia correttamente configurato.
- **testAddBook**: Questo test verifica che un manager possa aggiungere un libro al database. Viene creato un nuovo oggetto `Item` di tipo libro, con titolo, autore e genere. Il libro viene aggiunto tramite il metodo `addBook()` del controller. Successivamente si verifica che il libro sia effettivamente presente nel database.
- **testDeleteBook**: Questo test verifica che un manager possa rimuovere un libro dal database. Viene utilizzato un libro preesistente (`mockBook`) e si controlla che sia presente. Il libro viene poi rimosso tramite il metodo `removeBook()` e si verifica che non sia più presente nel database.

4.2 STAFFTEST

Nel contesto del `StaffTest`, JUnit viene utilizzato per testare il comportamento del controller `StaffController`, verificando che un utente con ruolo *staff* possa gestire correttamente le prenotazioni effettuate dagli utenti.

- **testGetUserReservations**: Questo test verifica che il personale possa visualizzare le prenotazioni associate a un determinato utente. Viene simulata l'autenticazione dell'utente e si controlla che il metodo `getUserReservations()` carichi correttamente le prenotazioni nel modello.
- **testActivationReservation**: Questo test verifica che una prenotazione possa essere attivata correttamente. Dopo l'attivazione tramite il metodo `activateReservation()`, si controlla che lo stato della prenotazione sia effettivamente attivo.
- **testCloseReservation**: Questo test verifica che una prenotazione possa essere chiusa correttamente. Dopo l'attivazione della prenotazione, viene simulata la scadenza manuale. Si controlla che, dopo la chiamata a `closeReservation()`, la prenotazione risulti disattivata e che l'oggetto prenotato (il libro) sia nuovamente disponibile.

4.3 USERTEST

Nel contesto del `UserTest`, JUnit viene utilizzato per testare il comportamento del controller `UserController`, verificando che un utente con permessi standard possa creare prenotazioni, visualizzarle e rimuoverle.

- **contextLoads**: Questo test verifica che le istanze di `UserDao`, `ItemDao` e `ReservationDao` siano correttamente iniettate nel contesto del test.
- **testUserCreation**: Questo test verifica che sia possibile salvare correttamente più utenti nel database. Dopo aver creato alcuni utenti, si controlla che il numero totale di utenti sia aumentato come previsto.
- **testUserReservation**: Questo test verifica che una prenotazione possa essere correttamente associata a un utente. Si crea una prenotazione e si controlla che sia presente tra quelle dell'utente.

- **testAddReservation**: Questo test verifica che un utente possa aggiungere una nuova prenotazione. Viene simulata l'autenticazione dell'utente, viene creata la prenotazione tramite il controller, e si controlla che la prenotazione sia salvata e che l'oggetto risulti prenotato.
- **testAddReservationReserved**: Questo test verifica il comportamento del sistema nel caso in cui un utente provi a prenotare un oggetto già riservato. Si controlla che la prenotazione venga rifiutata e che non venga salvata nel database.
- **testRemoveReservationNotActive**: Questo test verifica che una prenotazione non attiva possa essere rimossa correttamente. Dopo la rimozione, si controlla che la prenotazione non sia più presente nel database.
- **testRemoveReservationActive**: Questo test verifica il comportamento del sistema quando si tenta di rimuovere una prenotazione attiva. Si controlla che la prenotazione non venga eliminata, restando quindi presente nel database.