

Documento di Progettazione

"Biblioteca Universitaria" (23)

Corso di Ingegneria del Software
Anno Accademico 2025/26



INTRODUZIONE	3
SCopo DEL DOCUMENTO	3
1. Architettura del sistema	3
1.1 Package principali	4
→ 1.1.1 biblioteca.model	4
→ 1.1.2 biblioteca.view	4
→ 1.1.3 biblioteca.controller	5
→ 1.1.4 biblioteca.persistence	5
→ 1.1.5 biblioteca.main	5
1.2 Dipendenze tra i moduli	6
1.3 Diagramma dei Package	7
1.4 Diagramma delle Classi (hide methods/attributes)	7
2. Modello statico	8
2.1 MODEL	8
→ 2.1.1 GestioneLibri CLASS	8
→ 2.1.2 GestionePrestiti CLASS	10
→ 2.1.3 GestioneUtenti CLASS	12
→ 2.1.4 Autenticazione CLASS	15
→ 2.1.5 Libro CLASS	17
→ 2.1.6 Utente CLASS	20
→ 2.1.7 Prestito CLASS	23
→ 2.1.8 DatiBiblioteca CLASS	25
2.2 CONTROLLER	27
→ 2.2.1 MainController CLASS	27
→ 2.2.2 LibriController CLASS	29
→ 2.2.3 UtentiController CLASS	33
→ 2.2.4 PrestitiController CLASS	35
→ 2.2.5 AuthController CLASS	38
2.3 PERSISTENZA DATI	40
→ 2.3.1 ArchivioFile CLASS.	40
2.4 VIEW	43
→ 2.4.1 LoginView CLASS	43

→ 2.4.2 MainFrame CLASS	45
→ 2.4.3 LibriPanel CLASS	48
→ 2.4.4 UtentiPanel CLASS	52
→ 2.4.5 PrestitiPanel CLASS	56
→ 2.4.6 MenuView CLASS	59
2.5 Coesione, accoppiamento e principi di buona progettazione.	62
→ 2.5.1 Coesione (Alta – funzionale)	62
→ 2.5.2 Accoppiamento (Basso / Gestito)	63
→ 2.5.3 Principi di buona progettazione	63
3. Modello dinamico	65
DIAGRAMMI DI SEQUENZA	65
Flusso di autenticazione (Login)	65
Inserimento Libro	67
Cancellazione Libro	70
Modifica Libro	72
Ricerca Libri	74
Registrazione Nuovo Prestito	76
Registrazione Restituzione Prestito	79
Caricamento e Salvataggio Dati	81
4. User Interface	84
4.1 Login	84
4.2 Navigazione dal menu principale	87
4.3 Gestione Libri	88
4.4: Gestione Utenti	89
4.5: Gestione Prestiti	89
→ 4.3.1: Panoramica scheda gestione libri	91
→ 4.3.2 Inserimento libro	92
→ 4.3.3 Modifica libro	92
→ 4.3.4 Eliminazione libro	93
→ 4.3.5 Ricerca libro	94
→ 4.4.1 Panoramica scheda gestione utenti	94
→ 4.4.2 Inserimento utente	95
→ 4.4.3 Modifica utente	96
→ 4.4.4 Eliminazione utente	97
→ 4.4.5 Blacklist utente	98
→ 4.4.6 Ricerca utente	98
→ 4.5.1 Panoramica scheda gestione prestiti	100
→ 4.5.2 Inserimento di un nuovo prestito	101
→ 4.5.3 Registrazione della restituzione	101
→ 4.5.4 Gestione blacklist utente	102
4.6 Barra superiore: Menu e Logout	103

INTRODUZIONE

SCOPO DEL DOCUMENTO

Lo scopo principale di un documento di design è fornire una visione d'insieme dettagliata del sistema in sviluppo, in modo che tutti possano comprendere come il sistema funzionerà e come sarà sviluppato. Questo documento è un riferimento essenziale per il team di sviluppo durante tutto il processo, in quanto definisce una base comune da cui attingere nelle successive fasi di implementazione e testing.

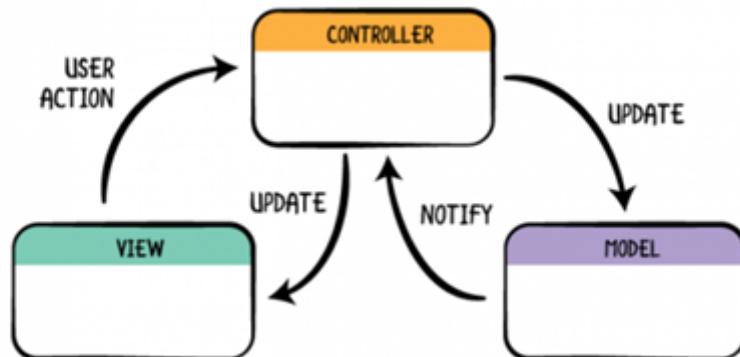
1. Architettura del sistema

L'applicazione “Biblioteca Universitaria” è organizzata in modo modulare secondo un’architettura a livelli basata sul pattern **MVC** (Model–View–Controller), con un package aggiuntivo dedicato alla persistenza dei dati e uno per il punto di avvio dell’applicazione.

L’obiettivo principale è separare:

- la gestione dei dati e delle regole di dominio (**Model**),
- la presentazione grafica e l’interazione con l’utente (**View**),
- la logica di coordinamento tra interfaccia e modello (**Controller**),
- la gestione dei file su disco (**Persistence**),
- l’avvio dell’applicazione (**Main**).

Questa suddivisione facilita manutenzione, riuso e possibilità di modificare in futuro interfaccia o modalità di salvataggio senza toccare le logiche di business.



1.1 Package principali

Di seguito si descrivono i cinque package principali, le loro responsabilità e il loro ruolo all'interno del sistema.

→ 1.1.1 **biblioteca.model**

Il package model rappresenta il cuore logico dell'applicazione, cioè il modello dei dati e le regole di business.

Al suo interno troviamo le classi di dominio:

- **Libro, Utente, Prestito**, che modellano rispettivamente i volumi a catalogo, gli utenti della biblioteca e i prestiti attivi/storici;
- **Autenticazione**, che gestisce la password del bibliotecario tramite hash.
- le classi di gestione/servizio:
 - **GestioneLibri**, che incapsula la collezione di libri e le operazioni di inserimento, modifica, eliminazione e ricerca;
 - **GestioneUtenti**, che gestisce l'anagrafica degli utenti, compresa la blacklist;
 - **GestionePrestiti**, che coordina la creazione e la chiusura dei prestiti, il controllo dei prestiti attivi, dei ritardi, ecc.;
 - **DatiBiblioteca**, che funge da contenitore aggregato dei vari gestori (libri, utenti, prestiti) e dell'oggetto Autenticazione, utile per il salvataggio complessivo dello stato della biblioteca.

Questo package non conosce l'interfaccia grafica né le modalità di salvataggio su file: espone soltanto classi e metodi che rappresentano il dominio della biblioteca e le relative regole.

→ 1.1.2 **biblioteca.view**

Il package view contiene tutte le classi dedicate all'interfaccia grafica (GUI).

Le sue responsabilità principali sono:

- mostrare i dati all'utente tramite **finestre, pannelli, tabelle e campi di input**;
- raccogliere l'input dell'utente (**click su pulsanti, testo inserito, selezioni nelle tabelle**);
- **inoltrare** le azioni dell'utente ai rispettivi controller.

Le classi della view non implementano la logica di business: non verificano vincoli sui prestiti, non aggiornano direttamente le strutture dati del modello e non accedono ai file. Tutte queste responsabilità sono delegate ai package controller e model.

→ 1.1.3 **biblioteca.controller**

Il package controller contiene le classi che fungono da ponte tra interfaccia grafica e modello.

Ogni controller è responsabile di una specifica area funzionale (ad esempio LibriController, UtentiController, PrestitiController, AuthController...).

In sintesi, i controller:

- raccolgono e validano i dati provenienti dalla view;
- invocano il model per eseguire le operazioni richieste;
- chiamano la persistenza per salvare lo stato aggiornato;
- aggiornano la view con i risultati o con eventuali messaggi di errore.

→ 1.1.4 **biblioteca.persistence**

Il package persistence è dedicato alla gestione della persistenza dei dati, cioè al salvataggio e al caricamento dell'archivio su file.

Al suo interno si trovano classi come, ad esempio, **ArchivioFile**, incaricate di caricare all'avvio lo stato della biblioteca e di salvare su file le modifiche apportate durante l'utilizzo dell'applicazione. Questo package conosce le classi del modello, perché deve poter leggere e scrivere oggetti di tipo Biblioteca, Libro, Utente, Prestito, ecc. Il resto dell'applicazione non si occupa del formato fisico dei file, ma usa esclusivamente i metodi esposti da questo package.

→ 1.1.5 **biblioteca.main**

Il package biblioteca.main raccoglie le classi che rappresentano il punto di avvio dell'applicazione e svolge il ruolo di semplice interruttore di avvio dell'applicazione: riceve il controllo dalla JVM e lo passa subito al livello di controllo che gestisce il normale funzionamento del sistema “Biblioteca”.

1.2 Dipendenze tra i moduli

Le dipendenze tra i package seguono la stratificazione MVC con persistenza e main.

- **biblioteca.model** è alla base e non dipende dagli altri package, implementa solo entità e regole di dominio;

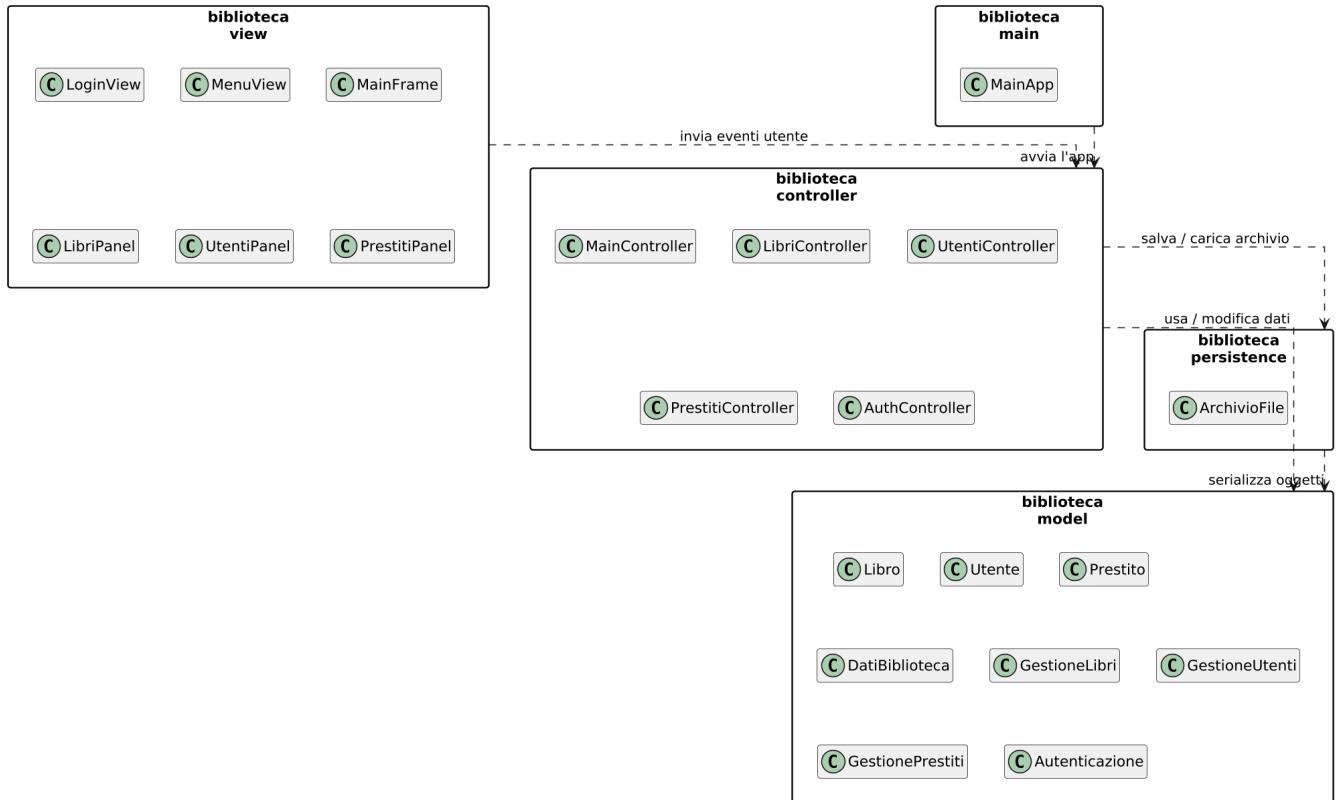
- **biblioteca.persistence** dipende da **biblioteca.model** perché deve leggere e scrivere su file oggetti come GestioneLibri, GestioneUtenti, GestionePrestiti, Autenticazione (eventualmente raggruppati in DatiBiblioteca);
- **biblioteca.controller** dipende da:
 - **biblioteca.model**, per applicare le regole di dominio;
 - **biblioteca.persistence**, per salvare/caricare lo stato;
 - **biblioteca.view**, per leggere input e aggiornare le schermate;
- **biblioteca.view** non dipende da nessun altro package.
- **biblioteca.main** dipende dal package **controller**: il metodo main crea il controller principale e gli cede il controllo, così che sia quest'ultimo a inizializzare model, view e persistenza.

Figura 1 – Riepilogo dei package principali:

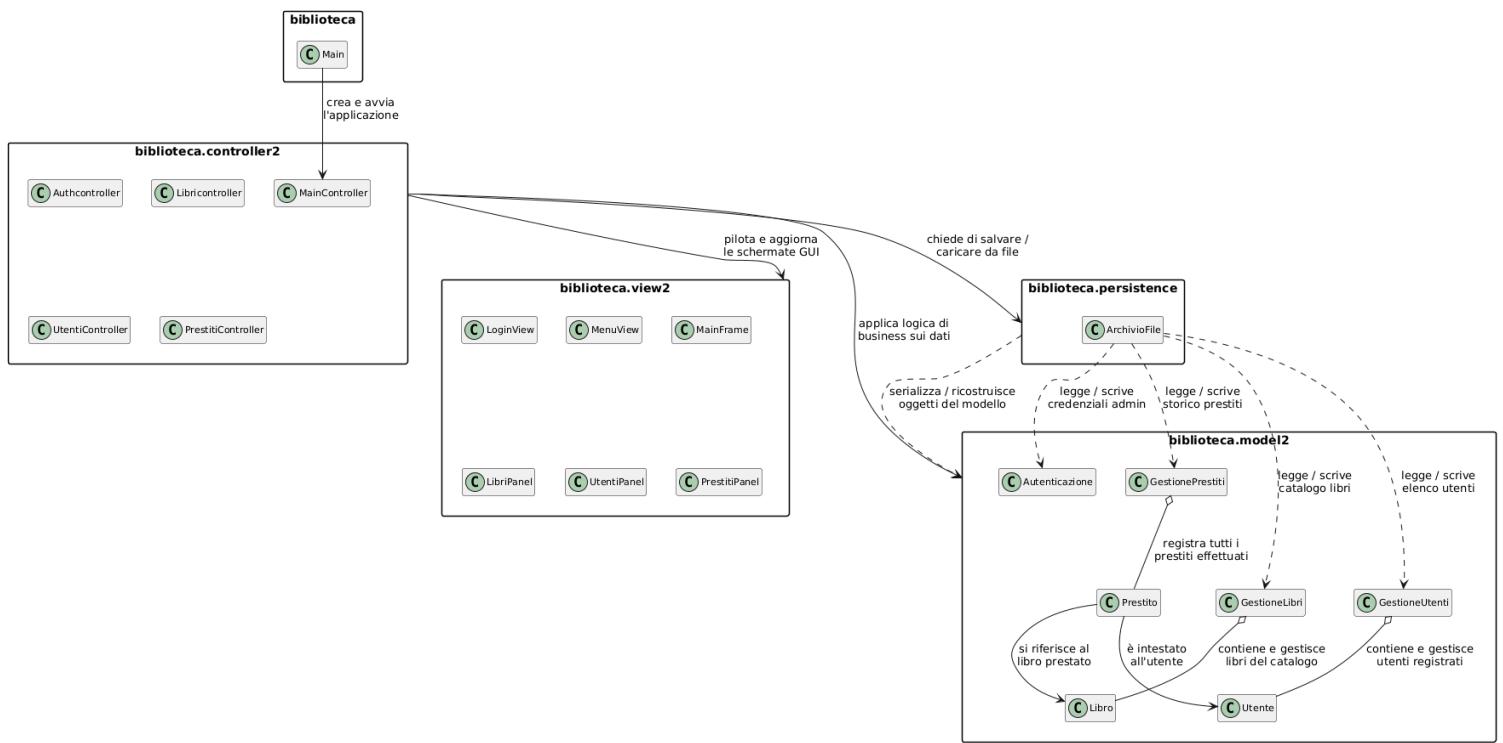
Package	Responsabilità principale	Dipende da
biblioteca.model	Dati e logica di business (entità di dominio + classe Biblioteca).	—
biblioteca.view	Interfaccia grafica (finestre, pannelli, tabelle, form).	—
biblioteca.controller	Ponte tra view e model; coordina operazioni e salvataggi.	model, view, persistence
biblioteca.persistence	Gestione della persistenza su file dell'archivio della biblioteca.	model
biblioteca.main	—	controller

1.3 Diagramma dei Package

Figura 2 :



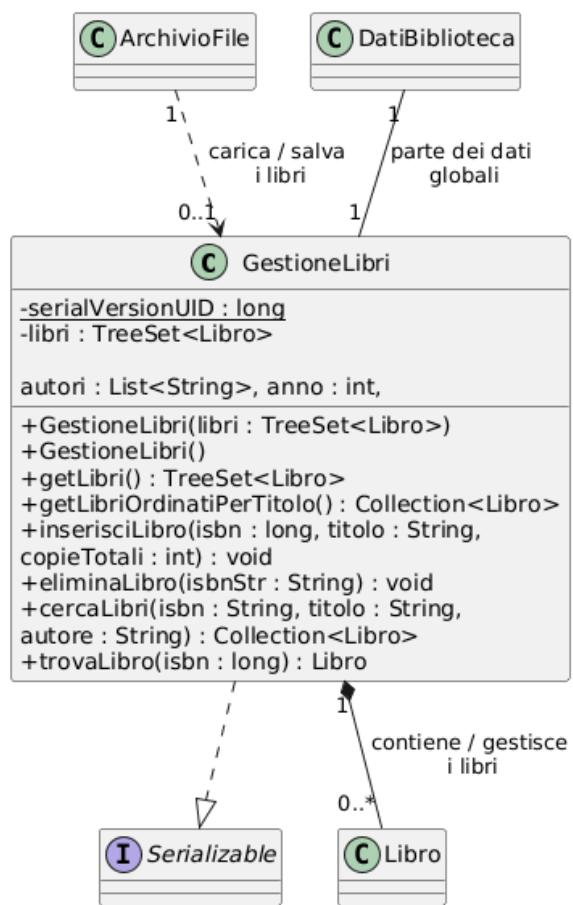
1.4 Diagramma delle Classi (hide methods/attributes)



2. Modello statico

2.1 MODEL

2.1.1 GestioneLibri CLASS



Attributi:

-serialVersionUID: long:

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

-libri : TreeSet<Libro>:

collezione che contiene i libri ordinati per titolo.

Metodi:

+GestioneLibri(libri: TreeSet<Libro>):

costruttore che permette di istanziare un oggetto di tipo GestioneLibri passando una collezione di libri già esistenti.

+GestioneLibri():

costruttore che permette di istanziare un oggetto di tipo GestioneLibri.

+getLibri(): TreeSet<Libro>:

ritorna il treeset di libri ordinati per titolo.

+getLibriOrdinatiPerTitolo(): List<Libro>:

ritorna la lista di libri ordinata per titolo.

+inserisciLibro(isbn:long, titolo: String, autori : List<String>, annoPubblicazione: int, copieTotali: int): Libro:

permette di inserire un nuovo libro specificando ISBN, titolo, autori, anno di pubblicazione e copie totali.

+modificaLibro(libro:Libro, titolo: String, autori:List<String>, annoPubblicazione: int, copieTotali:int): void:

permette di modificare un libro accedendo ai campi titolo, autori, anno di pubblicazione e copie totali.

+eliminaLibro(codiceIsbn: String):void:

permette di eliminare un libro dal sistema.

+cercaLibri(codiceIsbn: String, titolo:String, autore:String): TreeSet<Libro>:

permette di cercare i libri nel sistema filtrando per ISBN, titolo e autore.

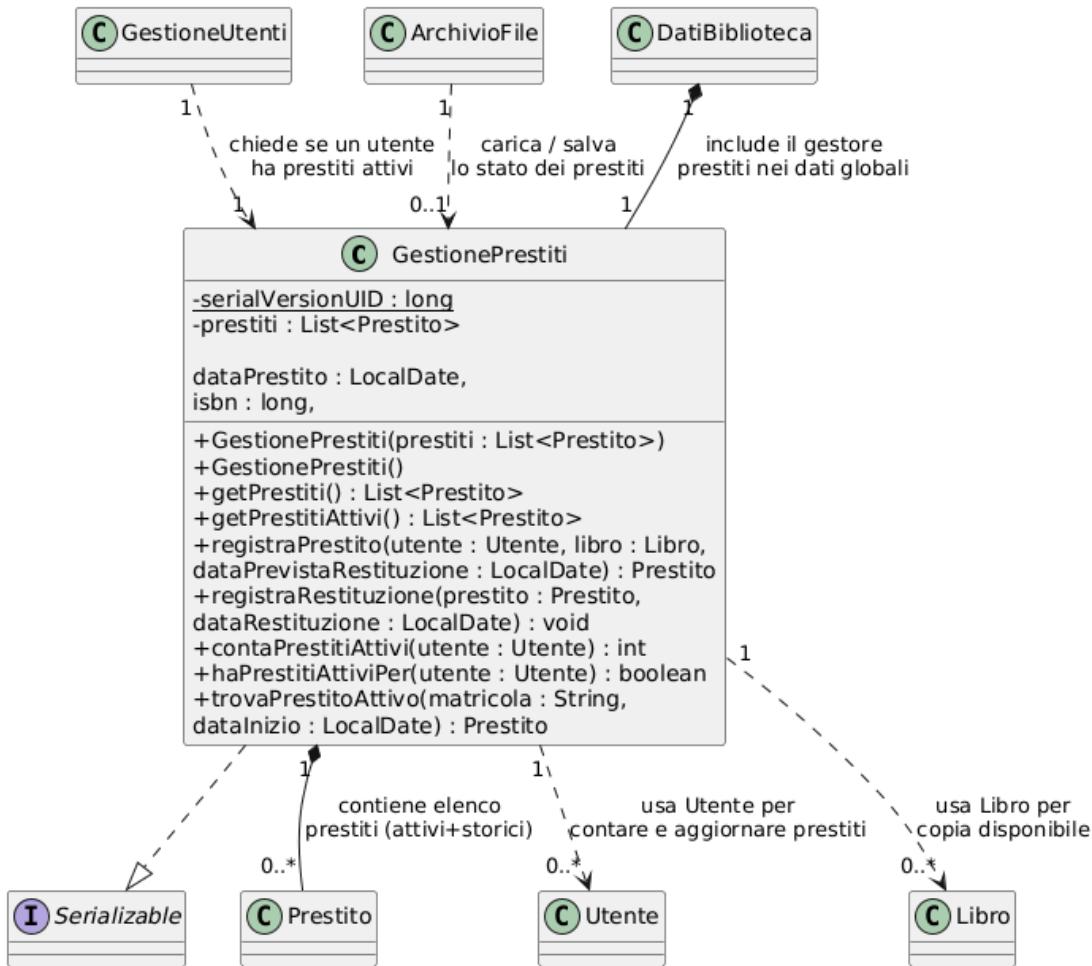
+trovaLibro(isbn:long): Libro:

permette di trovare un libro specifico tramite codice isbn.

Relazioni:

La classe **GestioneLibri** *implementa* l'interfaccia **Serializable** così che l'intera collezione dei libri può essere salvata e caricata da file e ha una relazione di *aggregazione* con **Libro**. Vi sono delle relazioni di *dipendenza* con **libriController** e **archivioFile**.

2.1.2 GestionePrestiti CLASS



Attributi:

-serialVersionUID : long :

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

-prestiti : List<Prestito>:

collezione che contiene la lista di prestiti.

Metodi:

+GestionePrestiti(prestiti:List<Prestiti>):

costruttore che permette di istanziare un oggetto Gestione Prestiti passando una lista di prestiti già esistente.

+GestionePrestiti():

costruttore che permette di istanziare un oggetto Gestione Prestiti.

+getPrestiti():List<Prestito>:

ritorna la lista di prestiti.

+getPrestitiAttivi(): List<Prestito>:

ritorna la lista di prestiti in corso.

**+registraPrestito(utente: Utente, libro: Libro,
dataPrestito:LocalDate, dataPrevistaRestituzione:LocalDate):
Prestito:**

permette di registrare un nuovo prestito inserendo matricola utente, ISBN libro e date di registrazione e restituzione prevista.

**+registraRestituzione(prestito:Prestito,dataRestituzione:LocalDa
te): void:**

permette di registrare la restituzione di un prestito attivo.

+contaPrestitiAttivi(utente:Utente): int :

restituisce il numero di prestiti attivi di un utente.

+haPrestitiAttiviPer(utente:Utente): boolean:

restituisce true se l'utente ha prestiti attivi, altrimenti false.

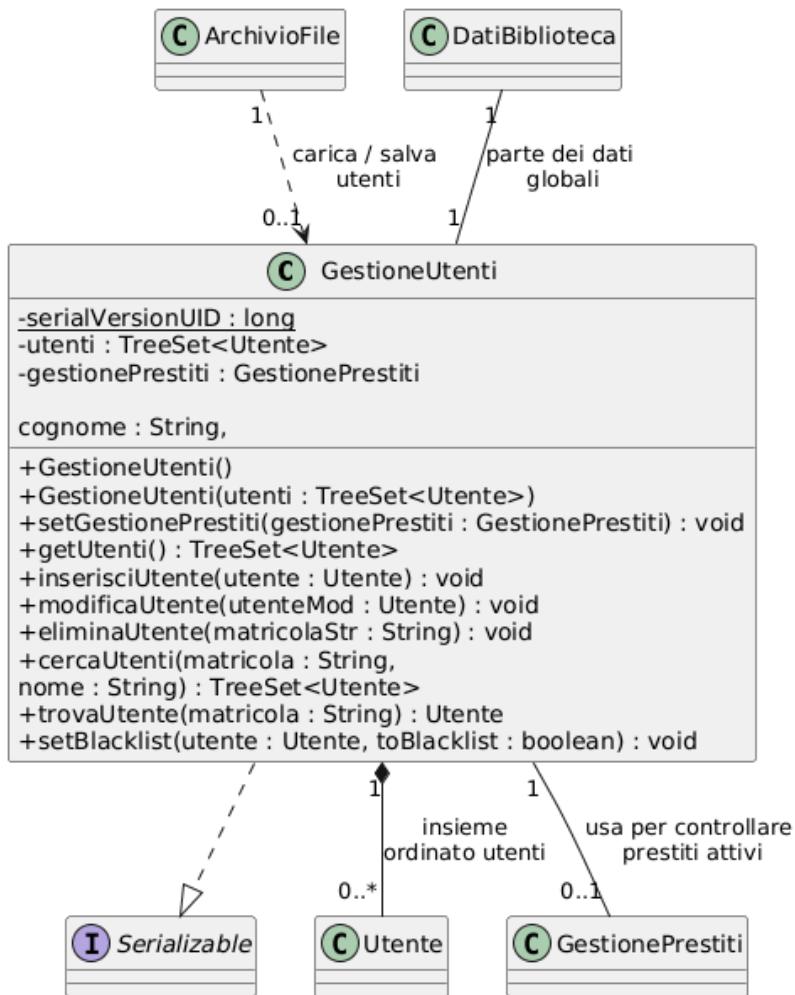
**+trovaPrestitoAttivo(matricola:String,isbn:long,dataInizio:Local
Date): Prestito:**

trova un prestito attivo specificando matricola, ISBN e data di inizio.

Relazioni:

GestionePrestiti implementa l'interfaccia **Serializable** così che l'intera collezione dei prestiti può essere salvata e caricata da file. Ha una relazione di *composizione* con **Prestito** perché contiene la lista di tutti i prestiti ed è il componente responsabile della loro gestione. Ha *dipendenze* verso **Utente** e **Libro**, che usa nei metodi per controllare il numero di prestiti attivi di un utente, verificare la disponibilità del libro e aggiornare le copie disponibili. **GestioneUtenti** è collegata a GestionePrestiti tramite una *dipendenza* 1–1, perché la usa per sapere se un utente ha ancora prestiti aperti prima di poterlo eliminare. **ArchivioFile** ha una *dipendenza* 1–0..1 verso GestionePrestiti e **DatiBiblioteca** ha una *composizione* 1–1 con GestionePrestiti.

2.1.3 GestioneUtenti CLASS



Attributi

-serialVersionUID:long:

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

-utenti: TreeSet<Utente>:

contiene la collezione di utenti ordinata per cognome e nome.

-gestionePrestiti: GestionePrestiti:

porta con sé le informazioni sui prestiti, utili per gestire gli utenti.

Metodi

+GestioneUtenti():

costruttore che permette di istanziare un oggetto GestioneUtenti.

+GestioneUtenti(utenti: TreeSet<Utente>):

costruttore che permette di istanziare un oggetto GestioneUtenti passando una lista di utenti già esistente.

+getUtenti(): TreeSet<Utente>:

ritorna l'elenco degli utenti ordinato per cognome e nome.

+InserisciUtente(utente:Utente): void:

permette di registrare un nuovo utente nel sistema.

+modificaUtente(utenteMod:Utente): void:

permette di modificare un utente presente nel sistema.

+eliminaUtente(matricolaStr:String): void:

permette di eliminare un utente del sistema.

+cercaUtenti(matricola: String, cognome : String, nome: String): TreeSet<Utente>:

permette di ricercare utenti nel sistema filtrandoli per cognome e nome.

+trovaUtente(matricola:String): Utente:

permette di ricercare un utente tramite la matricola.

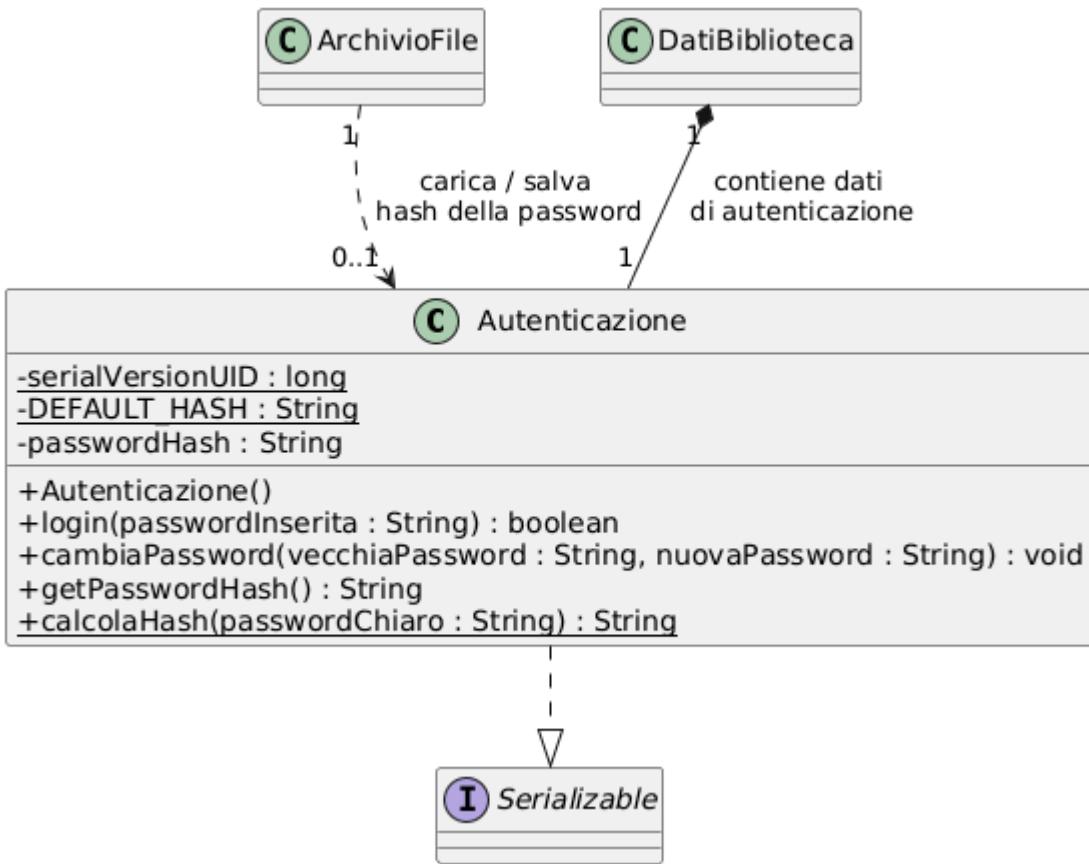
+setBlackList(utente: Utente, toBlacklist: boolean): void:

permette di aggiungere/rimuovere un utente in blacklist.

Relazioni:

La classe **GestioneUtenti** *implementa Serializable*, quindi i suoi oggetti possono essere salvati e caricati da file. C'è una relazione di *composizione* con la classe **Utente** che gestisce tramite un TreeSet, creando una relazione stabile uno-a-molti. Inoltre mantiene una relazione di *dipendenza* con **gestionePrestiti**, **utentiController** e **archivioFile** per quanto riguarda la gestione di prestiti attivi, la gestione degli utenti e l'uso della serializzazione.

2.1.4 Autenticazione CLASS



Attributi

-serialVersionUID:long:

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

-passwordHash:String:

rappresenta la password codificata.

Metodi:

+Autenticazione():

costruttore che permette di istanziare.

+login(passwordInserita:String):boolean:

ritorna un valore booleano che deriva dal confronto tra la password codificata e la password inserita al momento del login appena hashata, serve ad autenticare il bibliotecario per accedere al sistema.

**+cambiaPassword(vecchiaPassword:String,
nuovaPassword:String):void:**

permette di cambiare la password di accesso.

+getPasswordHash():String:

ritorna la stringa contenente la password hashata.

+calcolaHash(passwordChiaro:String):String:

metodo statico che permette di calcolare l'hash della password secondo l'algoritmo SHA-256.

Relazioni:

La classe **Autenticazione** *implementa* l'interfaccia **Serializable** per permettere il salvataggio della password su file. Questo consente al programma di ricordare la password anche dopo la chiusura dell'applicazione. C'è una relazione di *dipendenza* con **loginController** e **archivioFile** per salvare e caricare l'hash della password.

2.1.5 Libro CLASS



Attributi

-serialVersionUID: long :

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

+isbn: long:

codice univoco del libro.

-titolo: String :

titolo del libro.

-autori: List<String> :

lista contenente i nomi degli autori del libro.

-annoPubblicazione: int:

anno di pubblicazione del libro.

-copieTotali: int:

identifica le copie totali di un determinato libro.

-copieDisponibili: int :

identifica il numero di libri con questo ISBN non ancora prestati.

Metodi:

**+Libro(isbn:String, titolo:String, autore:String,
annoPubblicazione:int, copieTotali:int) : List <String>**

oppure

**+Libro(isbn:String, titolo:String, autore:String,
annoPubblicazione:int, copieTotali:int,copiedisponibili:int) :
List <String>:**

costruttore che crea un nuovo libro inizializzando i campi principali.

**+getIsbn() : String,
+getTitolo() : String,
+getAutore() : String,
+getAnnoPubblicazione() : int,
+getCopieTotali() : int,**

+getCopieDisponibili() : int :

metodi getter che restituiscono i rispettivi attributi del libro.

**+setIsbn(nuovoIsbn:String),
+setTitolo(nuovoTitolo:String),
+setAutore(nuovoAutore:String),
+setAnnoPubblicazione(nuovoAnno:int),
+setCopieTotali(nuovoTotale:int),**

+setCopieDisponibili(nuoveDisponibili:int) : void :
metodi setter che aggiornano i rispettivi attributi del libro.

+isDisponibile() : boolean :
indica se il libro ha almeno una copia disponibile per il prestito.

+incrementaCopiaDisponibile() : void,
+decrementaCopiaDisponibile() : void :
aggiornano il numero di copie disponibili quando un libro viene restituito o prestato.

+compareTo(other:Libro) : int :
confronta due libri in base all'ISBN per l'ordinamento nelle collezioni.

+equals(o:Object) : boolean
verifica se questo Libro è uguale all'oggetto passato.

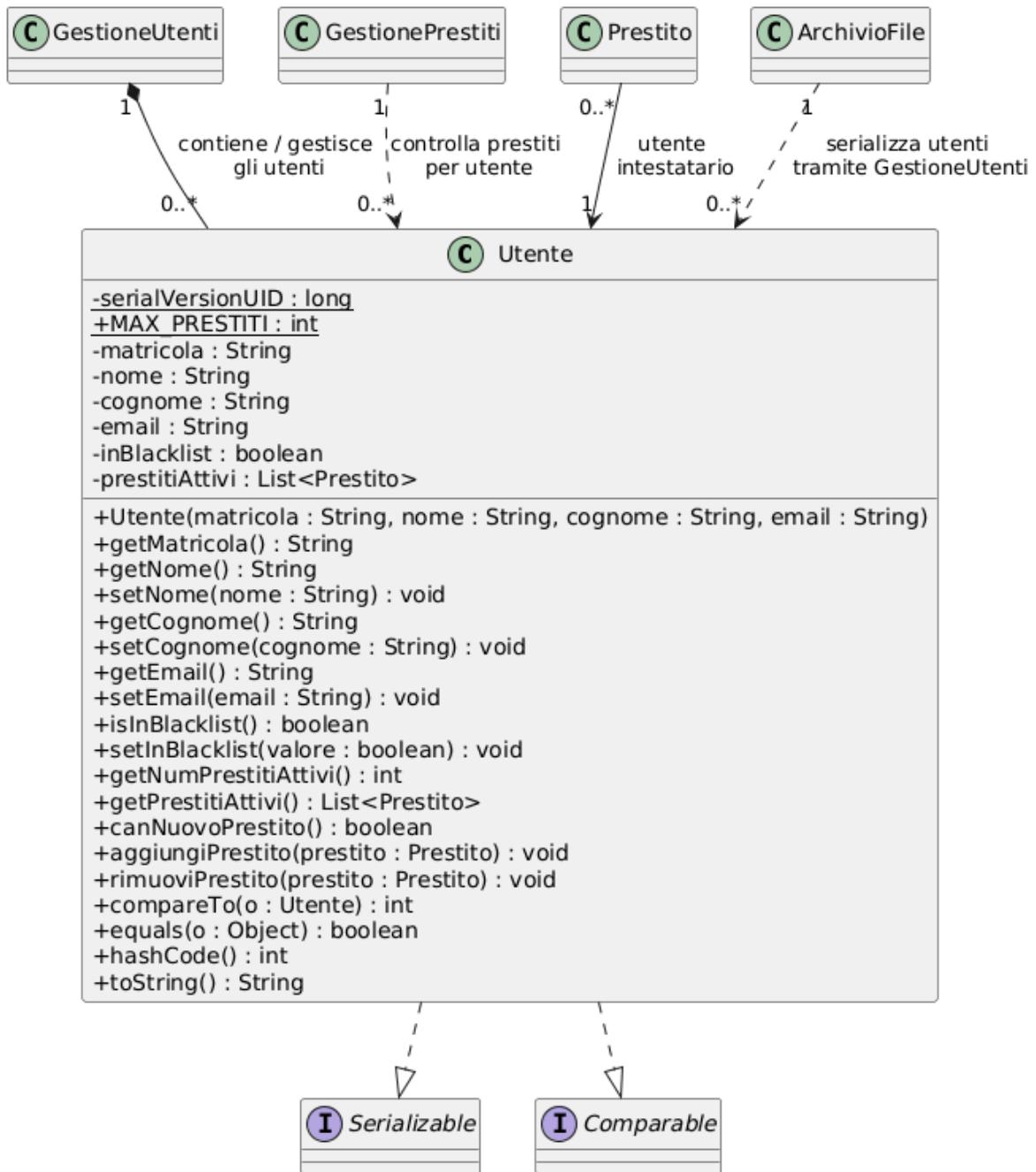
+ hashCode() : int :
restituisce il codice hash associato a questo Libro.

+toString() : String :
stampa a video le informazioni dell'oggetto.

Relazioni:

La classe **Libro** *implementa Serializable e Comparable*, per il salvataggio dei dati e l'ordinamento con compareTo; è una classe *aggregante* della classe **GestioneLibri**. C'è una relazione di tipo *dipendenza* con **gestionePrestiti**, **Prestito** e **archivioFile** rispettivamente per verificare la disponibilità delle copie e aggiornarle , verificare chi sottoscrive quel prestito e per serializzare e deserealizzare i libri su file.

2.1.6 Utente CLASS



Attributi

-serialVersionUID: long :

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

+MAX_PRESTITI: int :

numero massimo di prestiti consentiti per ogni utente.

-matricola: String :
codice identificativo univoco dello studente.

-nome: String :
nome dell'utente.

-cognome: String :
cognome dell'utente.

-email: String :
indirizzo email associato all'utente.

-inBlacklist: boolean :
indica se l'utente è inserito nella blacklist.

-prestitiAttivi: List<Prestito> :
elenco dei prestiti attualmente attivi associati a questo utente.

Metodi:

+Utente(matricola: String, nome: String, cognome: String, email: String) :
costruttore che crea un nuovo utente impostando matricola, nome, cognome ed email.

+getMatricola() : String ;, +getNome() : String ;, +getCognome() : String ;, +getEmail() : String :
metodi getter che restituiscono i valori degli attributi dell'oggetto.

+setNome(nome: String) : void :
setter che aggiorna il nome dell'utente.

+setCognome(cognome: String) : void :
setter che aggiorna il cognome dell'utente.

+setEmail(email: String) : void :
setter che aggiorna l'indirizzo email dell'utente.

+isInBlacklist() : boolean :
indica se l'utente risulta attualmente in blacklist.

+setInBlacklist(valore: boolean) : void :
imposta o rimuove l'utente dalla blacklist in base al valore passato.

+getNumPrestitiAttivi() : int :
restituisce il numero di prestiti ancora attivi per questo utente.

+getPrestitiAttivi() : List<Prestito> :
restituisce la lista dei prestiti attivi dell'utente.

+canNuovoPrestito() : boolean :

verifica se l'utente può effettuare un nuovo prestito, controllando che non superi MAX_PRESTITI e che non sia in blacklist.

+aggiungiPrestito(prestito: Prestito) : void :

aggiunge un nuovo prestito all'elenco dei prestiti attivi dell'utente.

+rimuoviPrestito(prestito: Prestito) : void :

rimuove il prestito indicato dalla lista dei prestiti attivi.

+compareTo(o: Utente) : int :

confronta questo utente con un altro per ordinarli all'interno di una collezione.

+equals(o: Object) : boolean :

determina se questo utente è uguale all'oggetto passato.

+hashCode() : int :

restituisce il codice hash associato a questo utente.

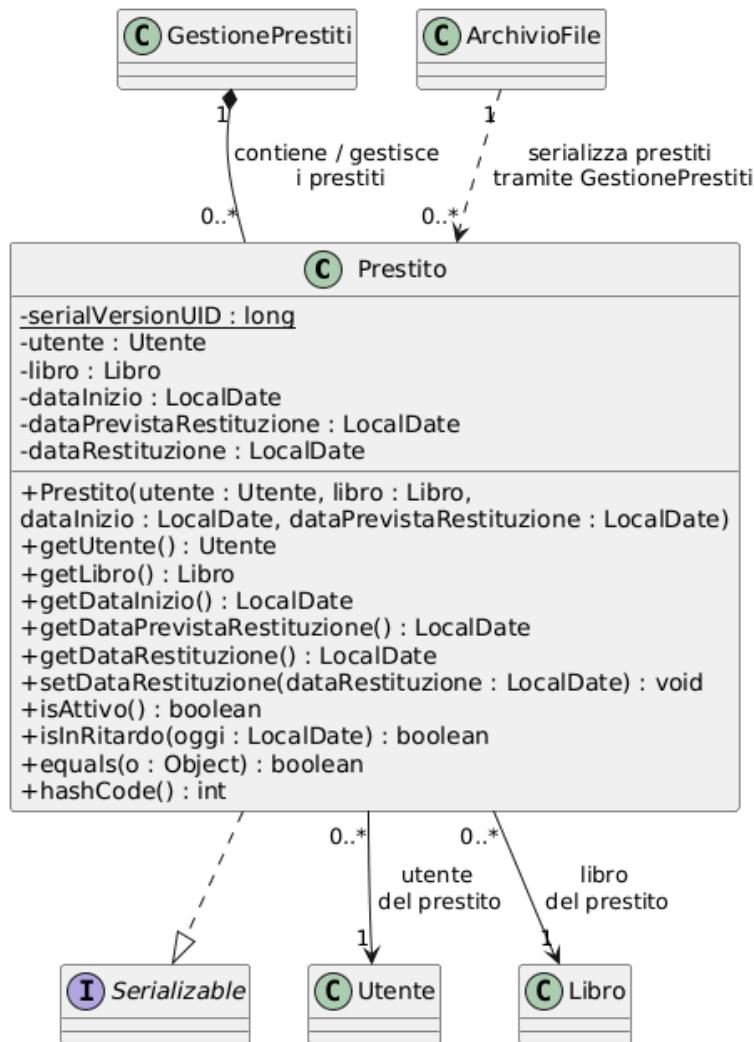
+toString() : String :

aggiorna il system.out.println

Relazioni:

La classe **Utente** *implementa Serializable* quindi può essere salvata e ripristinata insieme ai **dati della biblioteca** e *implementa anche Comparable*, così gli utenti possono essere ordinati. Abbiamo una relazione di *aggregazione* con **gestioneUtenti** e una *associazione uno a molti* con **Prestito**. Infine abbiamo una relazione di *dipendenza* con **gestionePrestiti** e **archivioFile**.

2.1.7 Prestito CLASS



Attributi

-serialVersionUID: long :

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

-utente: Utente :

riferimento all'utente che ha effettuato il prestito del libro.

-libro: Libro :

riferimento al libro che è stato prestato.

-dataInizio: LocalDate :

data in cui il prestito è stato registrato.

-dataPrevistaRestituzione: LocalDate :
data entro la quale l'utente dovrebbe restituire il libro.

-dataRestituzione: LocalDate :
data in cui il libro è stato effettivamente restituito (se il prestito è ancora attivo, questo attributo ha valore null).

Metodi:

+Prestito(utente: Utente, libro: Libro, dataInizio: LocalDate, dataPrevistaRestituzione: LocalDate) :
costruttore che crea un nuovo prestito.

+getUtente() : Utente ,
+getLibro() : Libro ,
+getDataInizio() : LocalDate ,
+getDataPrevistaRestituzione() : LocalDate ,
+getDataRestituzione() : LocalDate :
metodi getter che restituiscono i valori degli attributi dell'oggetto.

+setDataRestituzione(dataRestituzione: LocalDate) : void :
metodo setter che imposta la data di restituzione del prestito quando il libro viene riportato in biblioteca.

+isAttivo() : boolean :
indica se il prestito è ancora attivo, cioè se il libro non è stato ancora restituito.

+isInRitardo(oggi: LocalDate) : boolean :
verifica se il prestito risulta in ritardo rispetto alla data prevista di restituzione.

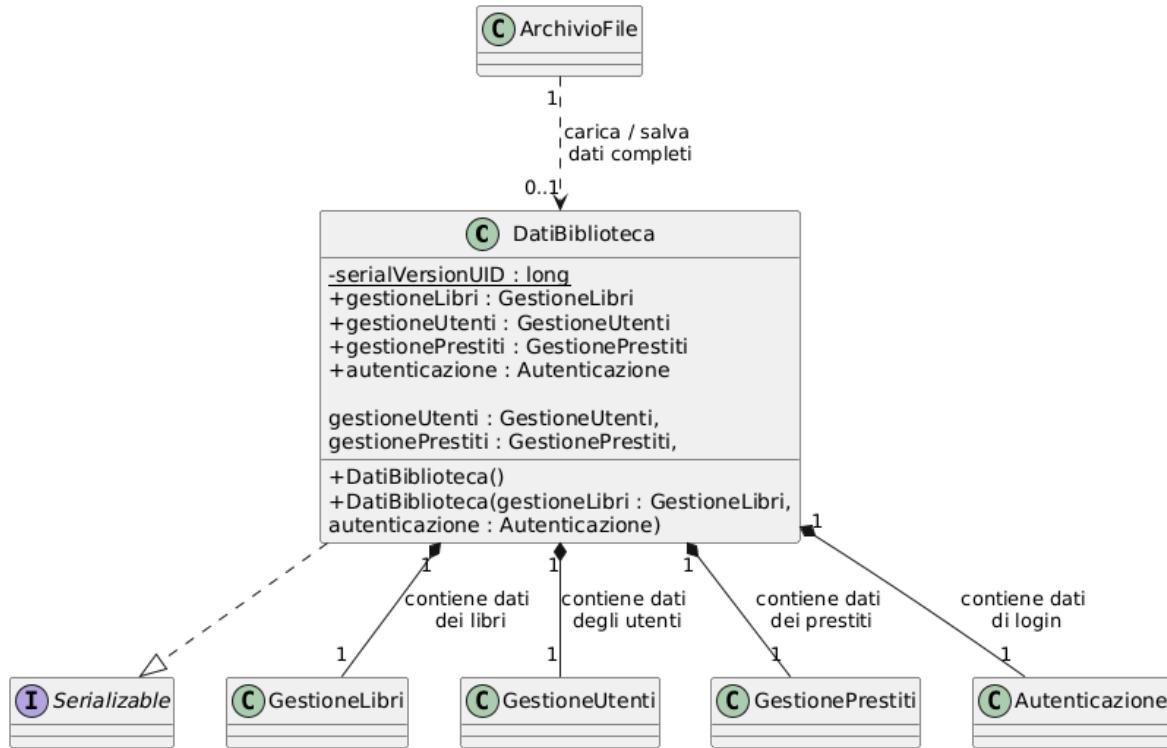
+equals(o: Object) : boolean :
confronta questo prestito con un altro prestito per stabilire se rappresentano lo stesso prestito.

+hashCode() : int :
restituisce il codice hash associato a questo prestito.

Relazioni:

La classe **Prestito** *implementa Serializable*, quindi i suoi oggetti possono essere salvati insieme allo stato della biblioteca. Ogni prestito collega in modo diretto un singolo **Utente** con un'associazione uno a molti e un singolo **Libro**, con una relazione *uno a molti*. Abbiamo una relazione di composizione con **gestionePrestiti** perché i prestiti vengono gestiti da gestionePrestiti. Abbiamo una relazione di dipendenza con **archivioFile**.

2.1.8 DatiBiblioteca CLASS



Attributi:

-serialVersionUID: long :

attributo statico che rappresenta l'identificativo di versione della classe serializzabile, necessario per la persistenza dei dati.

+gestioneLibri: GestioneLibri :
riferimento al gestore dei libri.

+gestioneUtenti: GestioneUtenti :
riferimento al gestore degli utenti.

+gestionePrestiti: GestionePrestiti :
riferimento al gestore dei prestiti.

+autenticazione: Autenticazione :
riferimento all'oggetto che gestisce le informazioni di autenticazione.

Metodi:

+DatiBiblioteca() :
costruttore dell'oggetto.

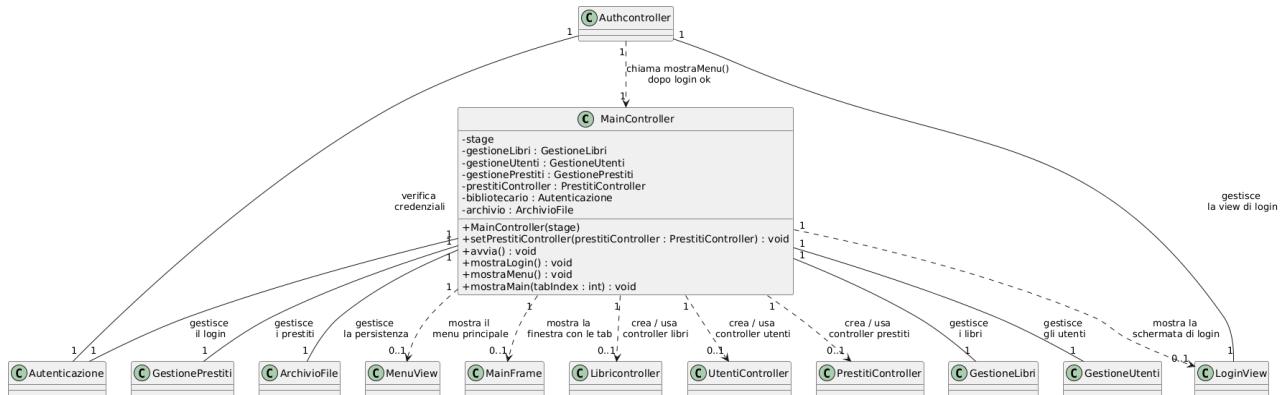
+DatiBiblioteca(gl : GestioneLibri, gu : GestioneUtenti, gp : GestionePrestiti, a : Autenticazione) :
costruttore che inizializza i riferimenti a gestore libri, gestore utenti, gestore prestiti e autenticazione.

Relazioni:

La classe **DatiBiblioteca** *implementa Serializable* e funge da contenitore unico per i moduli principali del sistema cioè **GestioneLibri**, **GestioneUtenti**, **GestionePrestiti** e **Autenticazione** con cui avviene una relazione di *aggregazione*. C'è una *relazione di dipendenza* con **archivioFile** perchè *usa* datiBiblioteca per leggere e scrivere dati su file.

2.2 CONTROLLER

2.2.1 MainController CLASS



Attributi:

-stage: Stage :

finestra principale JavaFX dell'applicazione, sulla quale vengono mostrate di volta in volta login, menu e finestra principale.

-gestioneLibri: GestioneLibri : gestore del modello dei libri.

-gestioneUtenti: GestioneUtenti : gestore del modello degli utenti.

-gestionePrestiti: GestionePrestiti : gestore del modello dei prestiti.

-prestitiController: PrestitiController : riferimento al controller della sezione prestiti.

-bibliotecario: Autenticazione : oggetto che gestisce le informazioni di autenticazione del bibliotecario.

-archivio: ArchivioFile : componente che si occupa di salvare e caricare su file lo stato di tutti i libri, utenti, prestiti e autenticazione.

Metodi:

+MainController(stage : Stage) : costruttore che inizializza il controller principale associandolo alla finestra JavaFX.

+setPrestitiController(prestitiController : PrestitiController): void:

metodo setter che imposta il riferimento al controller dei prestiti, collegandolo al controller principale dopo la sua creazione.

+avvia() : void :

punto di ingresso del flusso grafico: inizializza ciò che serve e mostra la prima schermata cioè quella di login.

+mostraLogin() : void :

metodo per visualizzare la schermata di login.

+mostraMenu() : void :

sostituisce il contenuto della finestra di login con il menu principale, accessibile solamente dopo un login corretto.

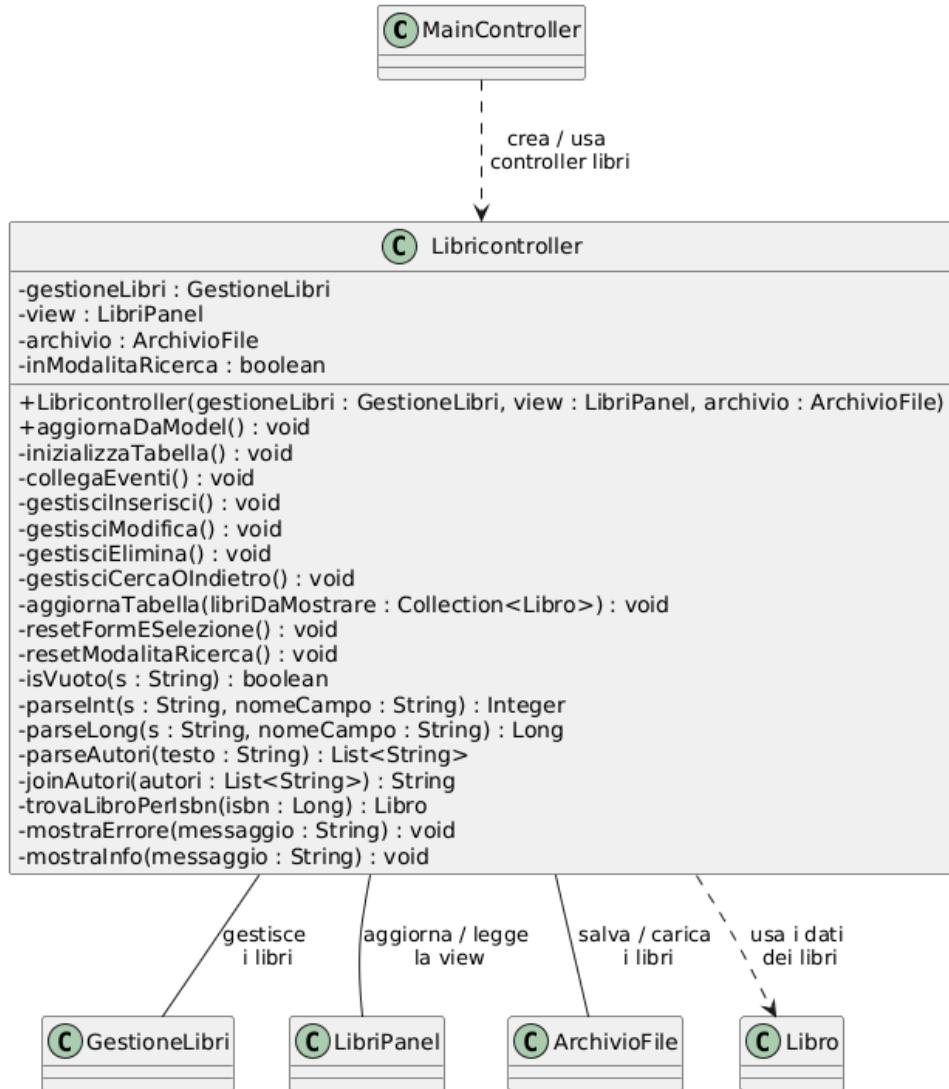
+mostraMain(tabIndex : int) : void :

mostra la finestra principale (Libri, Utenti, Prestiti), aprendo inizialmente la tab indicata dall'indice passato come parametro.

Relazioni:

La classe **MainController** è il controller principale dell'applicazione, è la classe con più dipendenze, perchè gestisce l'avvio, con **LoginView**, **MenuView**, **MainFrame**, **Authcontroller**, **Libricontroller**, **Utenticontroller**, **MainController**. E ci sono relazioni di associazione con **GestioneLibri**, **GestioneUtenti**, **GestionePrestiti**, **Autenticazione**, **ArchivioFile**. e **prestitiController**.

2.2.2 LibriController CLASS



Attributi:

-gestioneLibri: GestioneLibri :

gestore dei libri, contiene la collezione dei volumi e le operazioni di inserimento, modifica, eliminazione e ricerca.

-view: LibriPanel :

pannello grafico dedicato alla gestione dei libri, con campi di input, tabella dei libri e pulsanti di azione.

-archivio: ArchivioFile :

componente incaricato di salvare e caricare su file lo stato dei libri (e, in generale, dei dati dell'applicazione).

-inModalitaRicerca: boolean :

attributo booleano indica se la tabella sta mostrando i risultati di una ricerca (true) oppure l'elenco completo dei libri (false).

Metodi:**+Libricontroller(gestioneLibri: GestioneLibri, view: LibriPanel, archivio: ArchivioFile) :**

costruttore che inizializza il controller dei libri collegando model, vista e sistema di persistenza .

-inizializzaTabella() : void :

configura la tabella dei libri nella vista e la popola inizialmente con l'elenco dei libri presenti nel sistema .

-collegaEventi() : void :

associa ai pulsanti del pannello libri le azioni corrispondenti .

-gestisciInserisci() : void :

legge i dati inseriti nei campi, li valida e crea un nuovo libro nel model aggiornando anche la tabella.

-gestisciModifica() : void :

applica le modifiche ai dati del libro selezionato nella tabella, sincronizzando model e interfaccia grafica .

-gestisciElimina() : void :

elimina dal model il libro selezionato e aggiorna la tabella dei libri.

-gestisciCercaOIndietro() : void :

se non è in modalità ricerca, esegue una ricerca filtrando i libri in base ai campi compilati; se è già in ricerca, ripristina la visualizzazione completa .

-aggiornaTabella(libriDaMostrare: Collection<Libro>) : void :

aggiorna il contenuto della tabella visualizzando l'insieme di libri passato come parametro .

-resetFormESelezione() : void :

svuota i campi di input e deseleziona eventuali righe selezionate nella tabella dei libri .

-resetModalitaRicerca() : void :

disattiva la modalità ricerca e riporta la tabella all'elenco completo dei libri.

-isVuoto(s: String) : boolean :

verifica se la stringa passata è nulla, vuota o composta solo da spazi.

-parseInt(s: String, nomeCampo: String) : Integer :

prova a convertire la stringa in un numero intero, gestendo eventuali errori legati al campo indicato.

-parseLong(s: String, nomeCampo: String) : Long :

converte la stringa in un valore long.

-parseAutori(testo: String) : List<String> :

interpreta il testo inserito nel campo autori e lo suddivide in una lista di singoli autori .

+aggiornaDaModel() : void :

aggiorna rispetto agli input che arrivano da model.

-joinAutori(autori: List<String>) : String :

ricompone la lista di autori in una singola stringa da mostrare nella tabella o nei campi di input.

-trovaLibroPerIsbn(isbn: Long) : Libro :

cerca nel model il libro corrispondente all'ISBN indicato e lo restituisce, se presente.

-mostraErrore(messaggio: String) : void :

mostra un messaggio di errore all'utente.

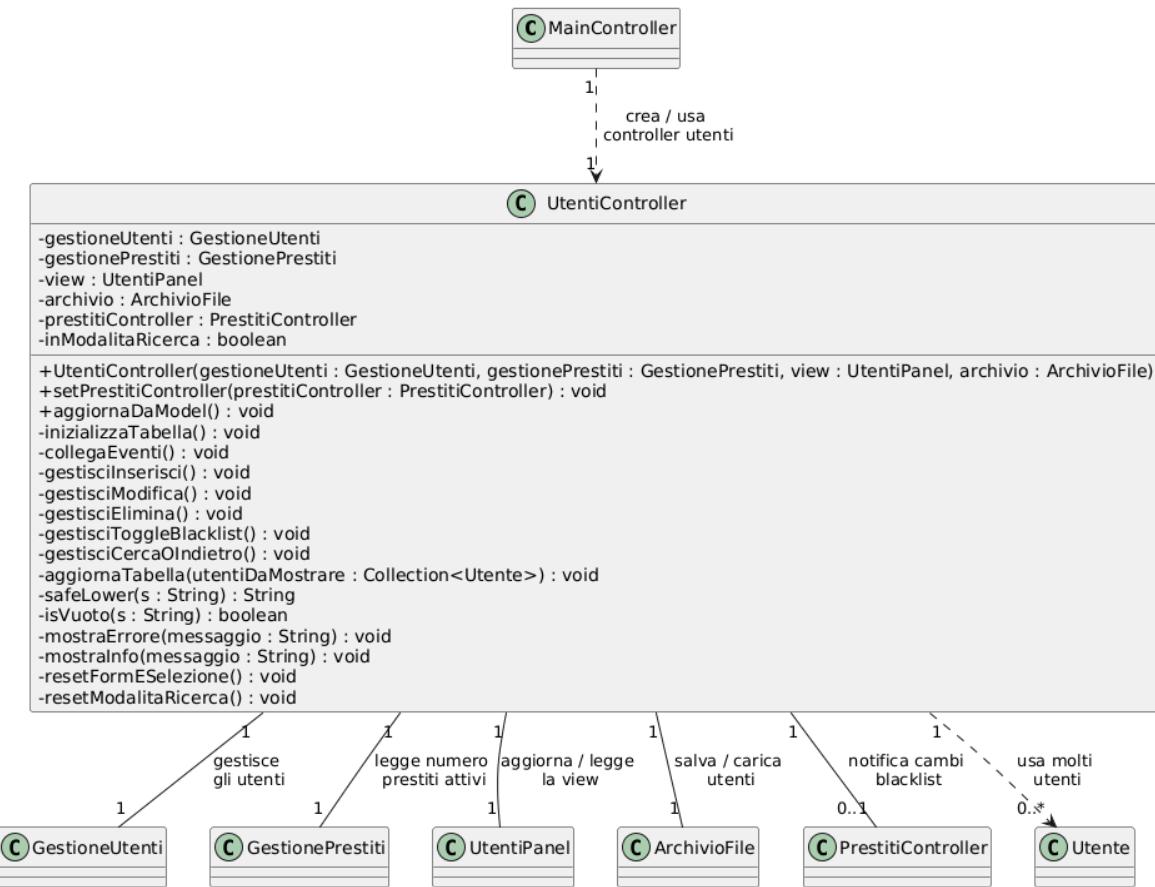
-mostraInfo(messaggio: String) : void :

mostra un messaggio informativo che conferma il buon esito di un'operazione.

Relazioni:

La classe **Libriconroller** collega la parte grafica della sezione “Gestione Libri” (**LibriPanel**) con il modello **GestioneLibri** e con la persistenza gestita da **ArchivioFile**. Avviene una relazione di *dipendenza* con **MainController** e **Libro** perchè usa Libro come tipo di dato principale per rappresentare i singoli volumi mostrati e manipolati dall’utente e la relazione con getioneLibri , LibriPanel e ArchivioFile è di *associazione*.

2.2.3 UtentiController CLASS



Attributi:

-gestioneUtenti: GestioneUtenti :
gestore degli utenti.

-gestionePrestiti: GestionePrestiti :
oggetto che gestisce i prestiti.

-view: UtentiPanel :
pannello grafico dedicato alla gestione degli utenti, con campi di input, tabella e pulsanti.

-archivio: ArchivioFile :

componente responsabile del salvataggio e caricamento su file dei dati relativi agli utenti .

-prestitiController: PrestitiController :
riferimento al controller dei prestiti.

-inModalitaRicerca: boolean :
attributo booleano che indica se la tabella sta mostrando il risultato di una ricerca (true) oppure l'elenco completo degli utenti (false).

Metodi:

**+UtentiController(gestioneUtenti: GestioneUtenti,
gestionePrestiti: GestionePrestiti, view: UtentiPanel, archivio:
ArchivioFile) :**
costruttore che inizializza il controller collegando il model degli utenti, il model dei prestiti, la vista utenti e il sistema di persistenza sui file.

**+setPrestitiController(prestitiController: PrestitiController)
:void :**
imposta il riferimento al controller dei prestiti, così da poter notificare aggiornamenti.

-inizializzaTabella() : void :
configura la tabella degli utenti nella vista e la popola.

-collegaEventi() : void :
associa ai pulsanti del pannello utenti le azioni corrispondenti.

-gestisciInserisci() : void :
legge i dati inseriti nei campi, valida l'input e crea un nuovo utente nel model e aggiorna la tabella.

-gestisciModifica() : void :
applica le modifiche ai dati dell'utente selezionato in tabella, aggiornando sia il model che la vista.

-gestisciElimina() : void :
rimuove l'utente selezionato e aggiorna la tabella.

-gestisciToggleBlacklist() : void :

aggiunge o rimuove l'utente selezionato dalla blacklist e notifica il controller dei prestiti per aggiornare i dati.

-gestisciCercaOIndietro() : void :

gestisce il tasto indietro.

-aggiornaTabella(utentiDaMostrare: Collection<Utente>) : void :

aggiorna il contenuto della tabella mostrando l'insieme di utenti passato come parametro.

-safeLower(s: String) : String :

restituisce la stringa in minuscolo, gestendo il caso in cui il valore sia nullo.

-isVuoto(s: String) : boolean :

verifica se una stringa è nulla, vuota o composta solo da spazi.

-mostraErrore(messaggio: String) : void :

mostra un messaggio di errore all'utente.

-mostraInfo(messaggio: String) : void :

mostra un messaggio informativo .

-resetFormESelezione() : void :

svuota i campi di input e deseleziona eventuali righe selezionate nella tabella.

-resetModalitaRicerca() : void :

disattiva la modalità ricerca e riporta la tabella alla visualizzazione dell'elenco completo.

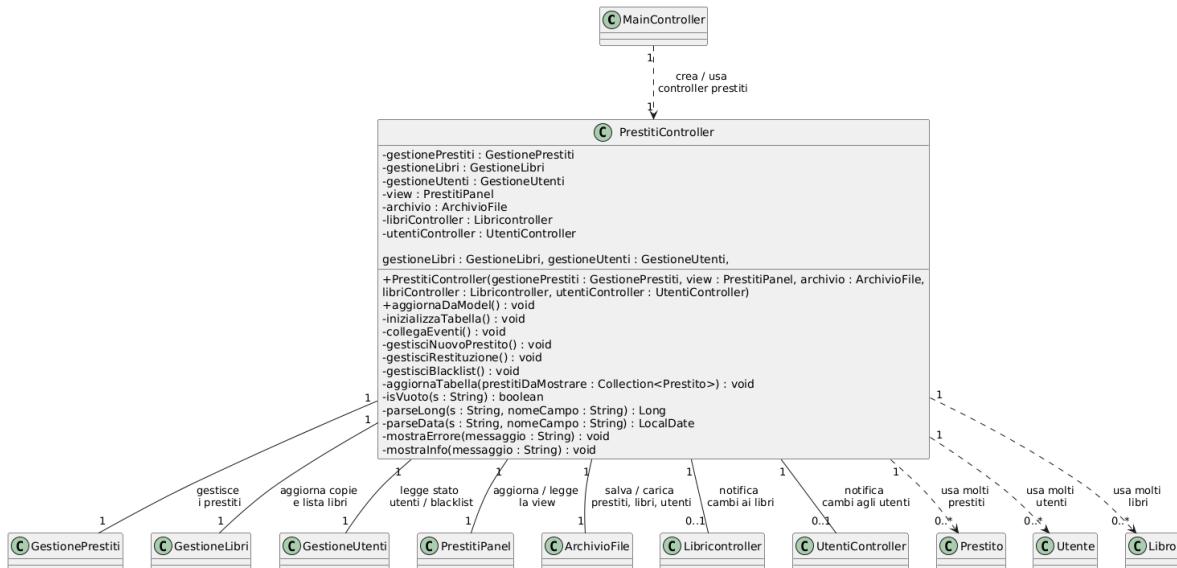
+aggiornaDaModel() : void :

ricarica l'elenco utenti dal model e aggiorna la tabella.

Relazioni:

UtentiController è il controller della sezione “Gestione Utenti”: ha associazioni (1–1) con **GestioneUtenti**, **GestionePrestiti**, **UtentiPanel** e **ArchivioFile**, che usa rispettivamente per gestire i dati degli utenti, contare i prestiti attivi, aggiornare la view e salvare/caricare i dati. È collegato a **PrestitiController** con una associazione 1–0..1, e viene creato e usato da un unico **MainController**, quindi tra MainController e UtentiController c’è una relazione 1–1 di dipendenza/associazione. E dipende da molti oggetti **Utente** , che vengono creati, cercati e modificati tramite i metodi del model.

2.2.4 PrestitiController CLASS



Attributi

-gestionePrestiti: GestionePrestiti :
riferimento all'oggetto che gestisce la logica dei prestiti.

-gestioneLibri: GestioneLibri :
riferimento al gestore dei libri.

-gestioneUtenti: GestioneUtenti :
riferimento al gestore degli utenti.

-view: PrestitiPanel :
pannello grafico dedicato alla gestione dei prestiti.

-archivio: ArchivioFile :
oggetto incaricato di salvare e caricare su file.

-libriController: LibriController :
controller della sezione libri, richiamato quando è necessario aggiornare la vista dei libri.

-utentiController: UtentiController :
controller della sezione utenti, usato per aggiornare le informazioni sugli utenti.

Metodi

+PrestitiController(gestionePrestiti: GestionePrestiti, view: PrestitiPanel, archivio: ArchivioFile, gestioneLibri: GestioneLibri, gestioneUtenti: GestioneUtenti, libriController: Libricontroller, utentiController: UtentiController) :
costruttore che inizializza il controller dei prestiti collegando model, view, archivio e gli altri controller coinvolti.

-inizializzaTabella() : void :
configura la tabella dei prestiti nella vista.

-collegaEventi() : void :
associa ai pulsanti del pannello dei prestiti le azioni corrispondenti.

-gestisciNuovoPrestito() : void :
legge i dati inseriti nella vista, verifica matricola e disponibilità del libro e, se tutto è corretto, registra un nuovo prestito.

-gestisciRestituzione() : void :
gestisce la restituzione del prestito selezionato, aggiornando data di restituzione e copie disponibili.

-gestisciBlacklist() : void :
inserisce o rimuove l'utente legato al prestito selezionato dalla blacklist, aggiornando anche la vista degli utenti.

-aggiornaTabella(prestitiDaMostrare: Collection<Prestito>) : void :
aggiorna il contenuto della tabella dei prestiti.

-isVuoto(s: String) : boolean :
verifica se una stringa è nulla o vuota e quindi non è valida come input.

-parseLong(s: String, nomeCampo: String) : Long :
converte la stringa in un valore long , gestendo eventuali errori di formattazione legati al campo indicato.

-parseData(s: String, nomeCampo: String) : LocalDate :
converte la stringa in una data , controllando che il formato sia valido per il campo specificato.

-mostraErrore(messaggio: String) : void :
mostra nella GUI un messaggio di errore.

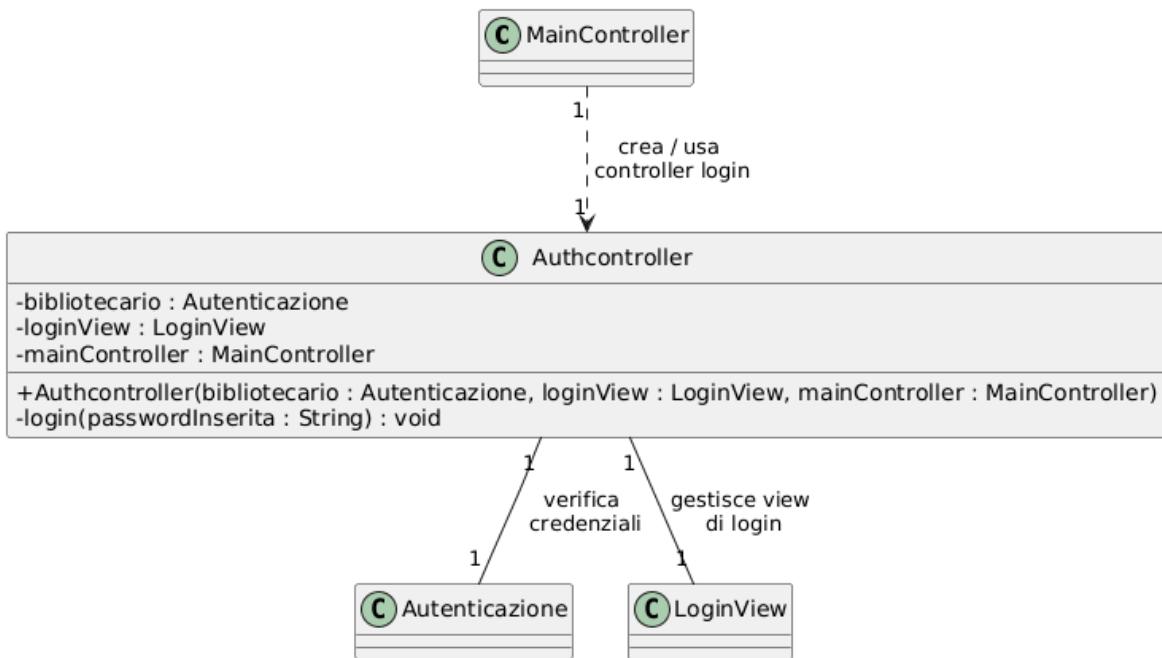
-mostraInfo(messaggio: String) : void :
mostra nella GUI un messaggio informativo relativo all'esito positivo di un'operazione.

+aggiornaDaModel() : void :
ricarica i dati dei prestiti dal model e aggiorna la tabella.

Relazioni:

PrestitiController è il controller della sezione “Gestione Prestiti”: ha associazioni (1-1) con **GestionePrestiti**, **GestioneLibri**, **GestioneUtenti**, **PrestitiPanel** e **ArchivioFile**, che sono usati per gestire i prestiti, aggiornare copie/libri e stato utenti, e salvare su file prestiti, libri e utenti. È collegato con associazioni 1-0..1 a **Libricontroller** e **UtentiController**, che vengono notificati quando cambiano i prestiti o la blacklist per aggiornare le view. E tra **MainController** e **PrestitiController** c’è una relazione 1-1 di dipendenza/associazione. E dipende da molti **Prestito**, **Utente** e **Libro**.

2.2.5 AuthController CLASS



Attributi

-bibliotecario: Autenticazione:

attributo che contiene i dati del bibliotecario e le informazioni necessarie per l’autenticazione.

-loginView: LoginView :

attributo per la schermata di login usata per inserire credenziali.

-mainController: MainController :

riferimento al controller principale.

Metodi

+AuthController(Autenticazione Bibliotecario , LoginView loginView, MainController mainController):

costruttore che inizializza il controller di autenticazione collegandolo al modello, alla view di login e al controller principale.

+login(String password): void :

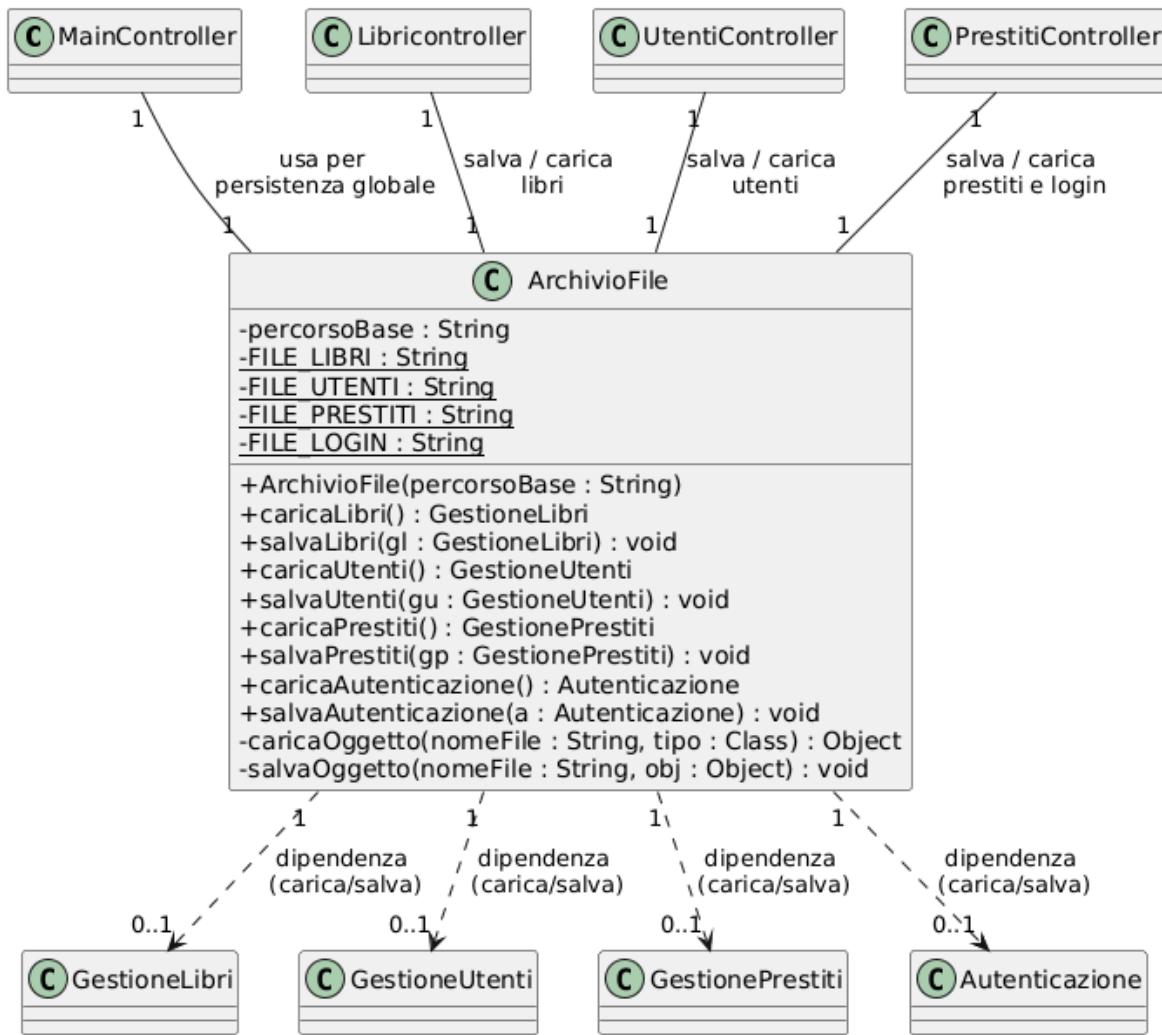
verifica le credenziali del bibliotecario.

Relazioni:

Authcontroller è il controller della schermata di login: ha *associazioni* con **Autenticazione** e **LoginView**, che le usiamo per verificare le credenziali inserite e per leggere/aggiornare la GUI di login. E ha una relazione 1–1 di *dipendenza/associazione* con il **MainController**.

2.3 PERSISTENZA DATI

→ 2.3.1 ArchivioFile CLASS.



Attributi

-percorsoBase: String :

percorso di base dove vengono salvati e letti i file dell'archivio.

-FILE_LIBRI: String :

nome del file che contiene i dati serializzati relativi ai libri .

-FILE_UTENTI: String :

nome del file che contiene i dati serializzati relativi agli utenti .

-FILE_PRESTITI: String :

nome del file che contiene i dati serializzati relativi ai prestiti .

-FILE_LOGIN: String :

nome del file che contiene i dati serializzati relativi alle informazioni di login.

Metodi

+ArchivioFile(percороBase: String) :

costruttore che inizializza l'oggetto impostando il percorso di base dove si trovano i file di salvataggio .

+caricaLibri() : GestioneLibri :

legge dal file dei libri e restituisce un oggetto GestioneLibri

+salvaLibri(gl: GestioneLibri) : void :

salva su file l'oggetto GestioneLibri passato come parametro.

+caricaUtenti() : GestioneUtenti :

legge dal file degli utenti e restituisce un oggetto GestioneUtenti.

+salvaUtenti(gu: GestioneUtenti) : void :

salva su file l'oggetto GestioneUtenti, sovrascrivendo i dati precedenti.

+caricaPrestiti() : GestionePrestiti :

legge dal file dei prestiti e restituisce un oggetto GestionePrestiti ei prestiti precedentemente salvati.

+salvaPrestiti(gp: GestionePrestiti) : void :

salva su file l'oggetto GestionePrestiti, aggiornando lo stato dei prestiti .

+caricaAutenticazione() : Autenticazione :

legge dal file di login e restituisce un oggetto Autenticazione con le informazioni salvate sulla password .

+salvaAutenticazione(a: Autenticazione) : void :

salva su file l'oggetto Autenticazione, memorizzando l'hash della password corrente .

-caricaOggetto<T>(nomeFile: String, tipo: Class<T>) : T :

metodo di utilità generico che apre il file indicato, deserializza un oggetto e lo restituisce .

-salvaOggetto(nomeFile: String, obj: Object) : void :

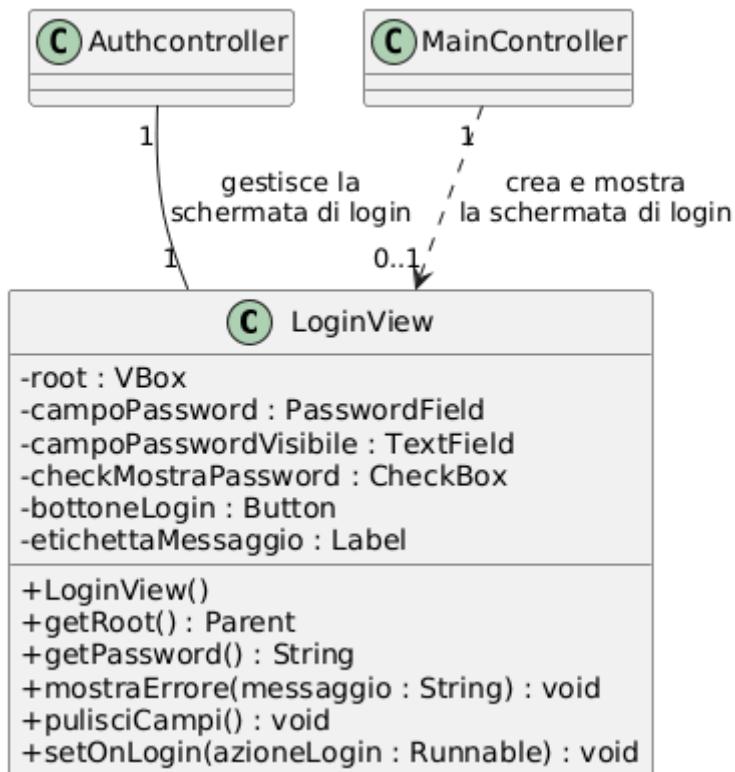
metodo che serializza l'oggetto passato e lo scrive nel file indicato dal nome ricevuto in input .

Relazioni:

ArchivioFile è la classe di persistenza: ha *associazioni 1–1* con **MainController**, **Libricontroller**, **UtentiController** e **PrestitiController**, che lo usano per salvare e caricare i dati da file. Verso **GestioneLibri**, **GestioneUtenti**, **GestionePrestiti** e **Autenticazione** ha relazioni di *dipendenza 1–0..1*. In pratica ArchivioFile funge da “ponte” tra i controller e il model, centralizzando tutte le operazioni di I/O in un unico punto.

2.4 VIEW

2.4.1 LoginView CLASS



Attributi

-root: VBox:

contenitore principale verticale della schermata di login .

-campoPassword: PasswordField :

campo di testo per l'inserimento della password .

-checkmostrapassword: CheckBox:

checkbox per visualizzare la password

-campoPasswordVisible: PasswordField :

campo di testo per visualizzare la password quando clicchiamo il tasto.

-bottoneLogin: Button :

pulsante per inviare la richiesta di login .

-etichettaMessaggio: Label :

etichetta utilizzata per mostrare messaggi di errore o informazione .

Metodi

+LoginView():

costruttore che inizializza i componenti grafici .

+getRoot(): Parent :

restituisce il contenitore grafico principale della schermata di login .

+getPassword(): String :

restituisce la password inserita.

+mostraErrore(String messaggio): void :

visualizza un messaggio di errore dove è richiesto .

+pulisciCampi(): void :

svuota il campo della password .

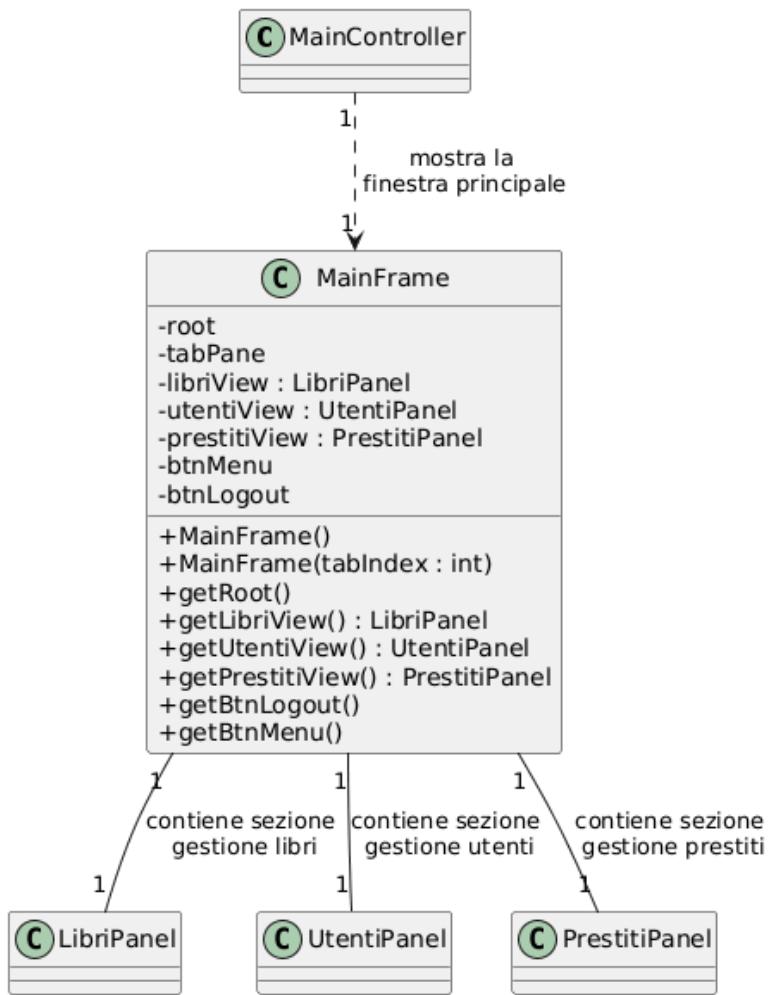
+setOnLogin(Runnable azioneLogin) void :

imposta il codice da eseguire quando l'utente preme il pulsante “Login”

Relazioni:

LoginView è la view iniziale per l'autenticazione: ha *un'associazione 1–1* con **Authcontroller**, che la usa per leggere la password, mostrare eventuali errori e reagire al click del bottone di login. Mentre **MainController** ha una relazione di *dipendenza 1–0..1* con LoginView, perché la crea solo quando deve mostrare la schermata di login e la inserisce nella Scene, senza mantenerla come campo fisso.

2.4.2 MainFrame CLASS



Attributi:

-root: BorderPane :

contenitore principale della finestra; ospita l'intera interfaccia grafica con barra superiore e area centrale con le tab.

-tabPane: TabPane :

componente che contiene le schede (cioè le tab) della finestra principale: Libri, Utenti e Prestiti.

-libriView: LibriPanel :

pannello associato alla scheda che gestisce la visualizzazione e l'interazione con i libri.

-utentiView: UtentiPanel :

Pannello associato alla scheda che gestisce la visualizzazione e l'interazione con gli utenti.

-prestitiView: PrestitiPanel :

pannello associato alla scheda che gestisce la visualizzazione e l'interazione con i prestiti.

-btnMenu: Button :

pulsante nella barra superiore che permette di tornare al menu principale dell'applicazione.

-btnLogout: Button :

pulsante nella barra superiore che consente al bibliotecario di effettuare il logout.

Metodi:

+MainFrame() :

costruttore che crea la finestra principale .

+MainFrame(tabIndex : int) :

costruttore che crea la finestra principale e apre inizialmente la scheda indicata dall'indice passato (0=Libri, 1=Utenti, 2=Prestiti) .

+getRoot() : Parent :

metodo getter che restituisce un pezzo di interfaccia.

+getLibriView() : LibriPanel :

metodo getter che restituisce il pannello dedicato alla gestione dei libri, associato alla relativa scheda.

+getUtentiView() : UtentiPanel :

metodo getter che restituisce il pannello dedicato alla gestione degli utenti.

+getPrestitiView() : PrestitiPanel :

metodo getter che restituisce il pannello dedicato alla gestione dei prestiti.

+getBtnLogout() : Button :

metodo getter che restituisce il pulsante di logout .

+getBtnMenu() : Button :

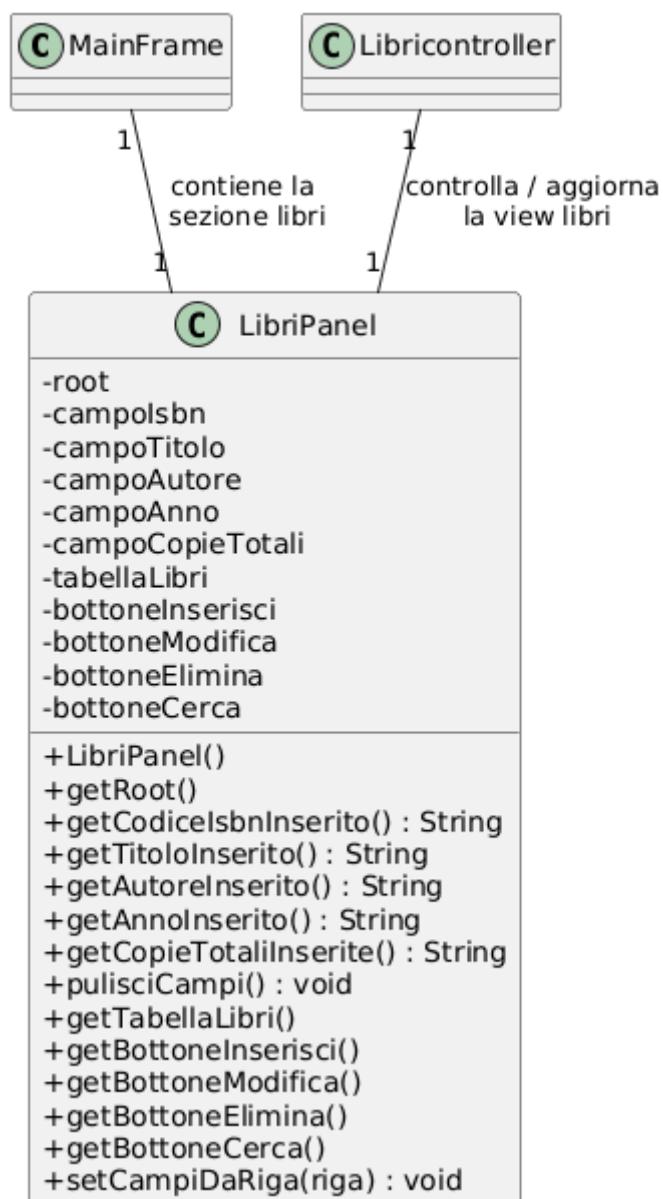
metodo getter che restituisce il pulsante che permette di tornare al menu principale.

Relazioni:

MainFrame è la finestra operativa principale: ha *associazioni 1–1* con **LibriPanel**, **UtentiPanel** e **PrestitiPanel**, che rappresentano le tre sezioni della dashboard (tab Libri, Utenti e Prestiti) e viene creato e usato da un unico **MainController**, con cui ha una relazione 1–1 di

dipendenza/associazione, perché il controller principale lo istanzia e lo inserisce nella scena per mostrare la finestra con le schede.

2.4.3 LibriPanel CLASS



Attributi:

-tabellaLibri: Table :

tabella che visualizza l'elenco dei libri presenti nella biblioteca.

-campoCodiceIsbn: TextField :

campo per l'inserimento del codice ISBN.

-campoTitolo: TextField :

campo per l'inserimento del titolo del libro.

-campoAutore: TextField :

campo per l'inserimento dell'autore.

-campoAnno: TextField :

campo per l'inserimento dell'anno di pubblicazione .

-campoCopieTotali: TextField :

campo per l'inserimento del numero di copie totali .

-bottoneInserisci: Button :

pulsante per inserire un nuovo libro.

-bottoneModifica: Button :

pulsante per modificare un libro esistente selezionato.

-bottoneElimina: Button :

pulsante per eliminare un libro selezionato.

-bottoneCerca: Button :

pulsante per effettuare la ricerca nella collezione.

-controller: LibriController :

controller che gestisce le azioni dei libri.

Metodi

+LibriPanel():

costruttore che inizializza i componenti grafici del pannello dei libri.

+getRoot() : Parent :

metodo getter che restituisce un pezzo di interfaccia.

+getCodiceIsbnInserito(): int :

metodo getter che restituisce il codice ISBN inserito nel relativo campo.

+getTitoloInserito(): String :

metodo getter che restituisce il titolo inserito.

+getAutoreInserito(): String :

metodo getter che restituisce l'autore inserito.

+getAnnoInserito(): int :

metodo getter che restituisce l'anno di pubblicazione inserito (convertito da un numero intero).

+getCopieTotaliInserite(): int :

metodo getter che restituisce il numero di copie totali inserito.

+pulisciCampi() : void :

cancella il contenuto di tutti i campi di input del pannello, riportandoli vuoti.

+getLibroSelezionato(): Libro :

metodo getter che restituisce il libro attualmente selezionato nella tabella, se presente.

+getTabellaLibri() : TableView<ObservableList<String>> :

metodo getter che restituisce la tabella che visualizza l'elenco dei libri presenti in biblioteca.

+getBottoneInserisci() : Button :

metodo getter che restituisce il pulsante usato per confermare l'inserimento di un nuovo libro.

+getBottoneModifica() : Button :

metodo getter che restituisce il pulsante che permette di confermare la modifica dei dati del libro selezionato.

+getBottoneElimina() : Button :

metodo getter che restituisce il pulsante che consente di eliminare il libro selezionato dalla tabella.

+getBottoneCerca() : Button :

metodo getter che restituisce il pulsante utilizzato per avviare la ricerca dei libri in base ai dati inseriti nei campi.

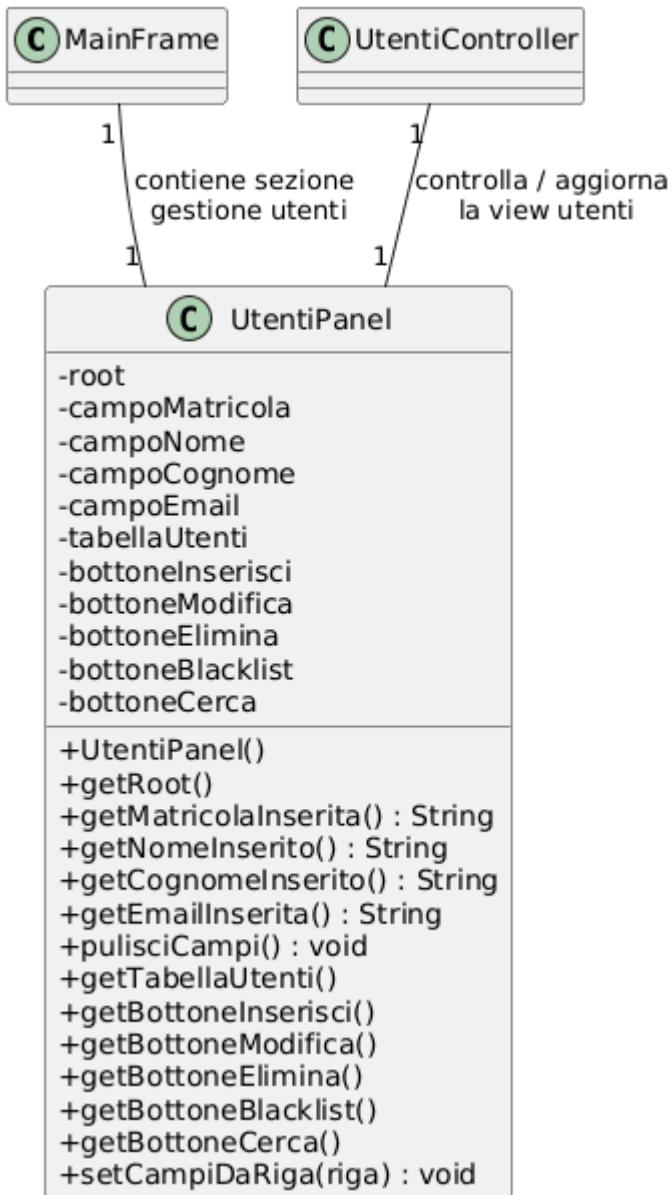
+setCampiDaRiga(riga : ObservableList<String>) : void :

metodo setter che imposta automaticamente i campi di input del pannello con i valori presenti nella riga selezionata della tabella libri.

Relazioni:

LibriPanel è la view per la gestione del catalogo libri e ha *associazioni 1-1* con **MainFrame** e con **libriController**, che legge i valori dei campi e aggiorna la tabella.

2.4.4 UtentiPanel CLASS



Attributi

-tabellaUtenti: Table :

tabella che visualizza l'elenco degli utenti già registrati.

-campoMatricola: TextField :

campo per l'inserimento della matricola.

-campoNome: TextField :

campo per l'inserimento del nome.

-campoCognome: TextField :

campo per l'inserimento del cognome.

-campoEmail: TextField :

campo per l'inserimento dell'email istituzionale.

-bottoneInserisci: Button :

pulsante per inserire un nuovo utente .

-bottoneModifica: Button :

pulsante per modificare l'utente selezionato .

-bottoneElimina: Button :

pulsante per eliminare l'utente selezionato .

-bottoneCerca: Button :

pulsante per cercare utenti in base al filtro .

-bottoneBlacklist: Button :

pulsante per aggiungere/rimuovere un utente dalla blacklist .

-controller: UtentiController :

controller che gestisce le azioni relative agli utenti .

Metodi

+UtentiPanel():

costruttore che inizializza i componenti grafici del pannello utenti .

+getRoot() : Parent :

metodo getter che restituisce un pezzo di interfaccia.

+getMatricolaInserita(): int :

metodo getter che restituisce la matricola inserita nel relativo campo.

+getNomeInserito(): String :

metodo getter che restituisce il nome inserito.

+getCognomeInserito(): String :

metodo getter che restituisce il cognome inserito.

+getEmailInserita(): String :

metodo getter che restituisce l'email istituzionale inserita.

+getUtenteSelezionato(): Utente :

metodo getter che restituisce l'utente selezionato nella tabella, se presente.

+pulisciCampi(): void :

svuota i campi di input del pannello.

+getTabellaUtenti() : TableView<ObservableList<String>> :

metodo getter che restituisce la tabella che visualizza l'elenco degli utenti presenti nel sistema.

+getBottoneInserisci() : Button :

metodo getter che restituisce il pulsante usato per confermare l'inserimento di un nuovo utente.

+getBottoneModifica() : Button :

metodo getter che restituisce il pulsante che permette di confermare la modifica dei dati dell'utente selezionato.

+getBottoneElimina() : Button :

metodo getter che restituisce il pulsante che consente di eliminare l'utente selezionato dalla tabella.

+getBottoneBlacklist() : Button :

metodo getter che restituisce il pulsante che permette di aggiungere o rimuovere l'utente selezionato dalla blacklist.

+getBottoneCerca() : Button :

metodo getter che restituisce il pulsante utilizzato per avviare la ricerca degli utenti in base ai dati inseriti nei campi.

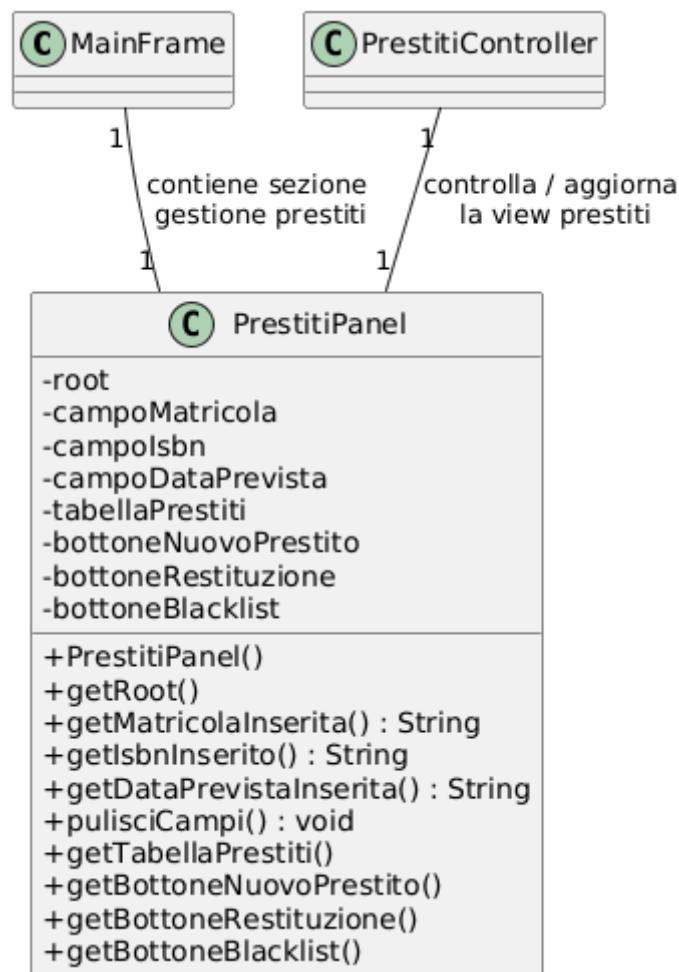
+setCampiDaRiga(riga : ObservableList<String>) : void :

metodo setter che imposta automaticamente i campi di input del pannello con i valori presenti nella riga selezionata della tabella utenti.

Relazioni:

UtentiPanel è la view per la gestione utenti e ha una *associazione 1–1* con MainFrame, che lo include come tab “Utenti” nella finestra principale, e una *associazione 1–1* con **UtentiController**, che legge i dati inseriti, aggiorna la tabella e collega i pulsanti alle operazioni sul model.

2.4.5 PrestitiPanel CLASS



Attributi:

-tabellaPrestiti: Table :

tabella che visualizza l'elenco dei prestiti .

-campoMatricola: TextField :

campo per l'inserimento della matricola dell'utente che richiede il prestito.

-campoCodiceIsbn: TextField :

campo per l'inserimento del codice del libro.

-campoDataPrevista: TextField :

campo di inserimento della data prevista di restituzione .

-bottoneNuovoPrestito : Button :

pulsante per registrare un nuovo prestito.

-bottoneRestituzione: Button :

pulsante per registrare la restituzione di un prestito.

-bottoneBlacklist: Button :

pulsante per inserire in blacklist l'utente selezionato o associato al prestito.

-controller: PrestitiController :

riferimento a prestitiController per la gestione dei prestiti.

Metodi:

+PrestitiPanel():

costruttore che inizializza i componenti grafici del pannello prestiti .

+mostraPrestiti(List<Prestito> prestiti): void :

aggiorna il contenuto della tabella con la collezione dei prestiti passati come parametro.

+getMatricolaInserita(): int :

metodo getter che restituisce la matricola inserita per creare un prestito.

+getCodiceIsbnInserito(): int :

metodo getter che restituisce il codice ISBN inserito per creare un prestito.

+getDataPrevistaInserita(): LocalDate :

metodo getter che restituisce la data prevista di restituzione inserita, convertita in LocalDate.

+getPrestitoSelezionato(): Prestito :

metodo getter che restituisce il prestito selezionato nella tabella, se presente.

+mostraMessaggio(String messaggio): void :

mostra un messaggio informativo o di errore .

+pulisciCampi(): void :

svuota i campi di input del pannello .

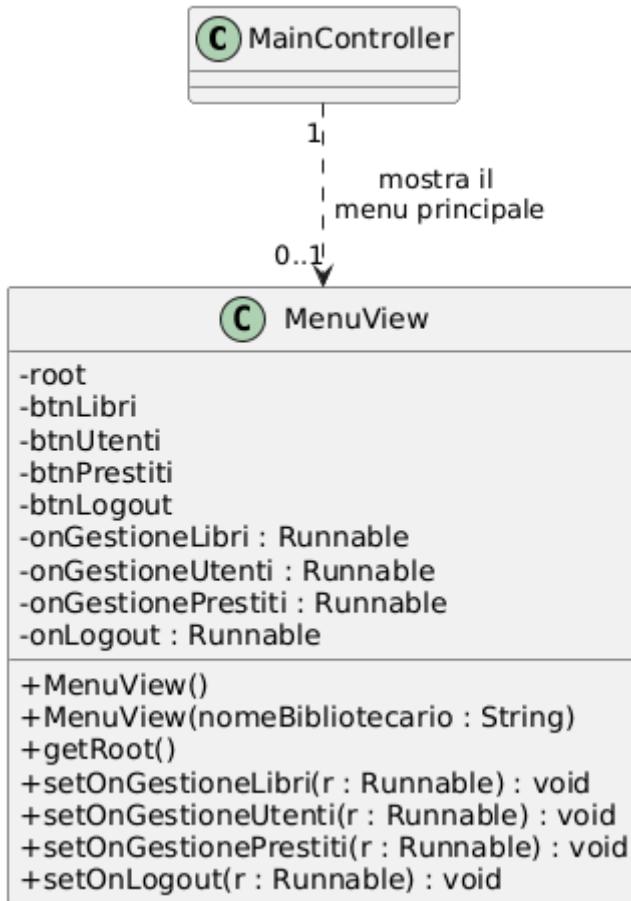
+setController(PrestitiController controller): void :

metodo setter che registra il controller che gestirà le azioni del pannello.

Relazioni:

PrestitiPanel è la view dedicata alla gestione dei prestiti e ha *associazioni 1–1* con **MainFrame**, che lo include come tab “Prestiti”, e con **PrestitiController**, che legge i dati inseriti, aggiorna la tabella e collega i bottoni alle operazioni sul model .

2.4.6 MenuView CLASS



Attributi:

-root: BorderPane :

pannello principale che contiene la struttura grafica del menu dell'applicazione.

-btnLibri: Button :

pulsante che permette di accedere alla sezione di gestione libri.

-btnUtenti: Button :

pulsante che permette di accedere alla sezione di gestione utenti.

-btnPrestiti: Button :

pulsante che permette di accedere alla sezione di gestione prestiti.

-btnLogout: Button :

pulsante che consente al bibliotecario di effettuare il logout dall'applicazione.

-onGestioneLibri: Runnable :

riferimento all'azione da eseguire quando l'utente seleziona la gestione libri.

-onGestioneUtenti: Runnable :

riferimento all'azione da eseguire quando l'utente seleziona la gestione utenti.

-onGestionePrestiti: Runnable :

riferimento all'azione da eseguire quando l'utente seleziona la gestione prestiti.

-onLogout: Runnable :

riferimento all'azione da eseguire quando l'utente preme il pulsante di logout.

Metodi:**+MenuView() :**

Costruttore che crea e inizializza l'interfaccia del menu senza parametri aggiuntivi.

+MenuView(nomeBibliotecario: String) :

Costruttore che crea il menu inizializzando eventualmente elementi grafici legati al nome del bibliotecario loggato.

+getRoot() : Parent :

Restituisce il nodo radice dell'interfaccia del menu, da impostare come contenuto della scena JavaFX.

+setOnGestioneLibri(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione dei libri (clic sul pulsante corrispondente).

+setOnGestioneUtenti(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione degli utenti.

+setOnGestionePrestiti(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione dei prestiti.

+setOnLogout(r: Runnable) : void :

Imposta l'azione da eseguire quando viene premuto il pulsante di logout.

Relazioni:

La classe **MainController** crea e gestisce la **MenuView**, che rappresenta il menu principale dopo il login, per questo c'è una relazione di *associazione* in cui il controller controlla il comportamento della schermata di menu.

2.5 Coesione, accoppiamento e principi di buona progettazione.

→ 2.5.1 Coesione (Alta – funzionale)

La coesione misura quanto gli elementi all'interno dello stesso modulo sono legati tra loro (intra-modulo).

Nel nostro sistema ogni classe e ogni package ha un compito ben definito:

Nel package **model** abbiamo un'alta coesione funzionale:

le classi di dominio **Libro**, **Utente**, **Prestito**, **Autenticazione** rappresentano singoli concetti del dominio e incapsulano solo i dati e i metodi che li riguardano;

le classi di servizio **GestioneLibri**, **GestioneUtenti** e **GestionePrestiti** si occupano rispettivamente della gestione del catalogo libri, dell'anagrafica utenti e del ciclo di vita dei prestiti, senza mischiare responsabilità.
Inizialmente avevamo pensato di creare una sola classe **Biblioteca**, ma avrebbe rovinato la coesione.

eventuali classi di supporto per la persistenza (es. **DatiBiblioteca**) raggruppano solo i dati da salvare/caricare.

Questa separazione netta del modello si riflette anche negli altri package:

- nel package **controller** ogni controller gestisce una sola funzionalità (**LibriController** solo libri, **UtentiController** solo utenti, **PrestitiController** solo prestiti), evitando un “mega-controller” che fa tutto;
- nel package view ogni pannello grafico (**LibriPanel**, **UtentiPanel**, **PrestitiPanel**, ecc.) è responsabile solo della propria parte di interfaccia.

In questo modo ogni modulo ha una responsabilità chiara e coerente, e all'interno dello stesso modulo le parti lavorano tutte allo stesso obiettivo.

→ 2.5.2 Accoppiamento (Basso / Gestito)

L'accoppiamento è una caratteristica che misura il grado di interdipendenza tra moduli diversi.

Nel sistema l'accoppiamento è mantenuto basso e controllato:

Le dipendenze seguono una struttura gerarchica e unidirezionale:

view → controller → model → persistence

i livelli inferiori non conoscono quelli superiori .

Nelle classi di servizio usiamo principalmente un accoppiamento per dati: si passano ai metodi solo le informazioni strettamente necessarie , riducendo l'impatto delle modifiche.

Tra le classi di dominio utilizziamo un accoppiamento per timbro quando ha senso passare l'intero oggetto, sfruttando i suoi metodi e il polimorfismo come nel caso di **Prestito** mantiene riferimenti a **Utente** e **Libro** per poter chiedere direttamente i dati necessari (nome utente, titolo libro, ecc.).

→ 2.5.3 Principi di buona progettazione

Essendo un sistema progettato con l'approccio object-oriented, sono stati adottati alcuni principi di buona progettazione per ridurre il debito tecnico ed evitare complessità inutili, seguendo linee guida come KISS (Keep It Simple, Stupid) e DRY (Don't Repeat Yourself).

Il principio maggiormente rispettato è il Single Responsibility Principle (SRP):

ogni classe svolge un solo compito ben definito (es. **GestioneLibri** gestisce solo libri, **GestioneUtenti** solo utenti, **GestionePrestiti** solo prestiti, **ArchivioFile** solo la persistenza), e ogni package ha una responsabilità chiara (**model**, **controller**, **view**, **persistence**).

Grazie a questa organizzazione, il sistema è anche vicino al principio di Open/Closed: è possibile estendere le funzionalità senza dover modificare pesantemente le classi esistenti.

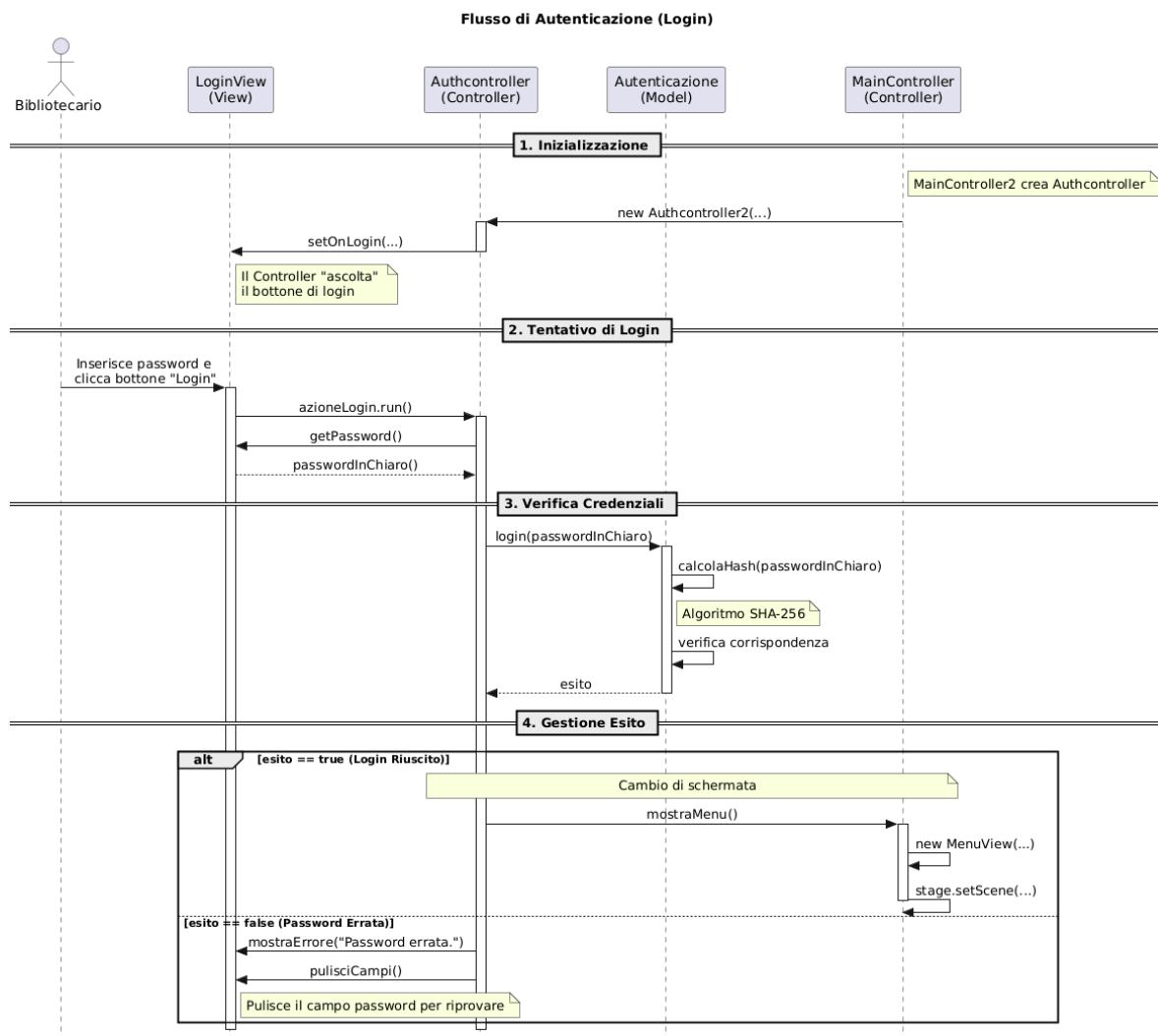
L'uso di ereditarietà e polimorfismo consente di scrivere codice che lavora in termini di interfacce/tipi generali, rispettando i principi SOLID più rilevanti per questo progetto (in particolare SRP e, in parte, OCP e ISP).

Infine, sono stati inseriti controlli sugli input e gestione degli errori (ad esempio ritorni di false, eccezioni controllate e messaggi alla GUI) per proteggere lo stato interno degli oggetti: dati non validi non compromettono la stabilità del sistema. Questo va nella direzione del principio di robustezza, rendendo il comportamento prevedibile anche in caso di input errati.

3. Modello dinamico

DIAGRAMMI DI SEQUENZA

UC-15 Login



Descrizione

Il diagramma rappresenta il **Flusso di Autenticazione (Login)** del sistema per la gestione della biblioteca. Il diagramma è diviso in 4 sezioni logiche:

Sezione 1: Inizializzazione

All'avvio dell'applicazione, il **MainController** (controller principale) crea un'istanza dell'**AuthController** (controller dedicato al processo di

autenticazione), che invia un messaggio alla **LoginView** (Interfaccia grafica della sezione di login), per mettersi in ascolto di azioni che l’utente effettua sull’interfaccia di login.

Sezione 2: Tentativo di Login

A seguito dell’input dell’utente, che inserisce una password (oscurata) e preme il bottone per effettuare il login, la **LoginView** segnala l’evento all’**AuthController**, che richiede e riceve la password in chiaro inserita nel form di autenticazione attraverso il metodo **getPassword()**.

Sezione 3: Verifica delle Credenziali

L’**AuthController** delega la verifica al **Model** inviando la password in chiaro all’oggetto **Autenticazione** (classe che incapsula la logica di business relativa alla sicurezza) attraverso il metodo **login(...)**. Il **Model** procede all’hashing della password mediante l’algoritmo **SHA-256** (standard sicuro che permette di non dover salvare password in chiaro) e alla verifica della corrispondenza tra hash della password salvato nel database e hash della password immessa dall’utente. Autenticazione restituisce quindi l’esito di queste operazioni all’**AuthController**.

Sezione 4: Gestione dell’Esito

Abbiamo ora due scenari (blocco alternativo **alt**):

→ Scenario 1: esito positivo (Login Riuscito)

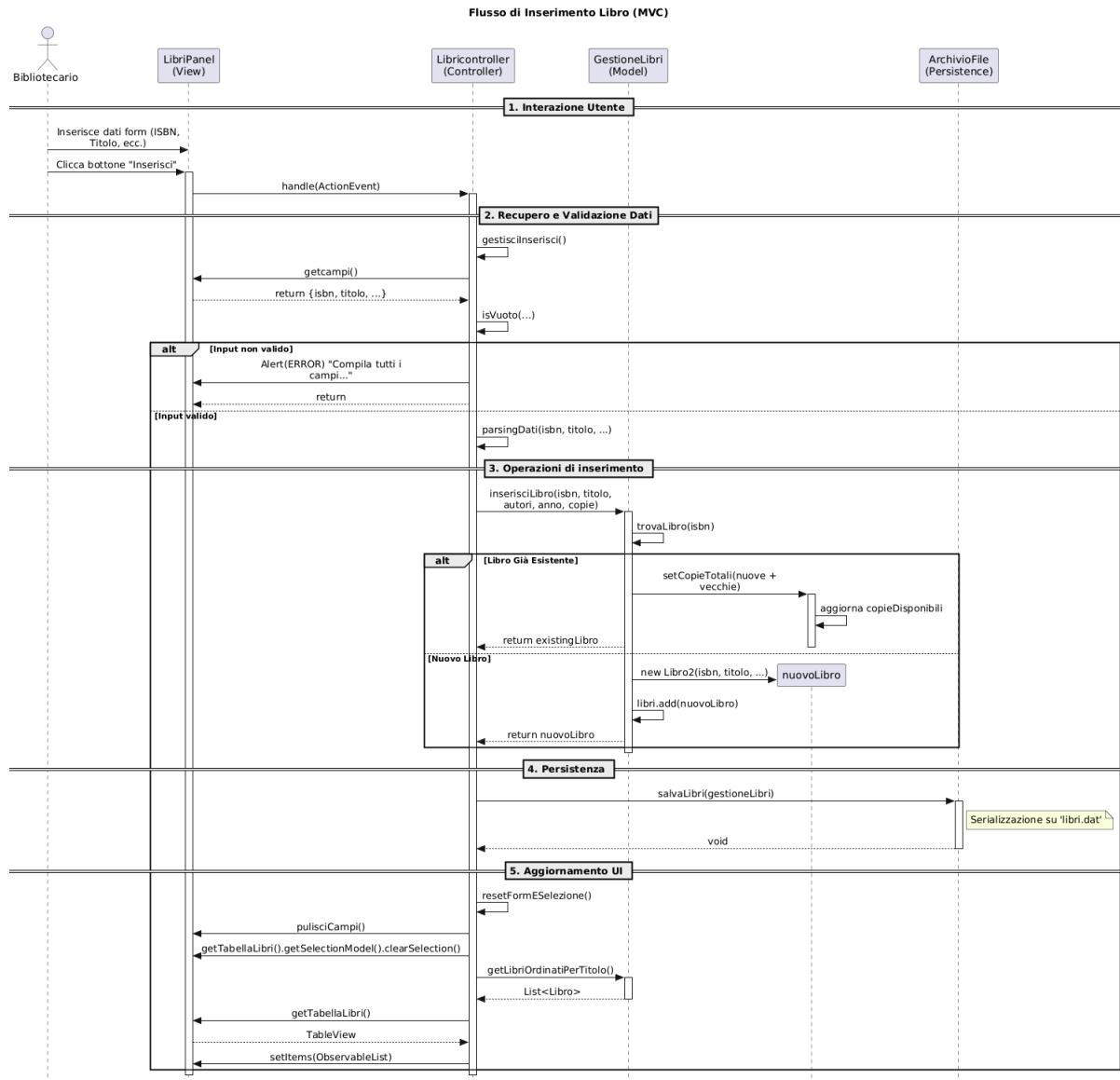
L’autenticazione è riuscita e l’**AuthController** invoca il cambio di schermata al **MainController** per passare al menù principale dell’applicazione.

Il **MainController** quindi crea l’oggetto che rappresenta l’interfaccia da mostrare e setta la scena su di essa.

→ Scenario 2: esito negativo (Password Errata)

L’**AuthController** invoca la **LoginView** una volta per mostrare un messaggio di errore “*Password errata*” e una seconda volta per resettare il campo password e permettere un nuovo tentativo.

UC-2 Inserimento Libro



Descrizione

Il diagramma rappresenta il processo di **Inserimento di un nuovo Libro** (o aggiornamento di uno esistente) all'interno del sistema di gestione della biblioteca.

Il diagramma è diviso in 5 sezioni logiche:

Sezione 1: Interazione Utente

Tutto inizia con l'azione del bibliotecario sulla **LibriPanel** (Modulo View, interfaccia grafica della sezione Libri). L'utente inserisce i dati del libro (ISBN, Titolo, Autori, ecc.) nell'apposito form e clicca sul bottone "*Inserisci*". La **View** intercetta questo evento e invoca **LibriController** (controller della sezione Libri), avviando le operazioni per l'inserimento.

Sezione 2: Recupero e Validazione Dati

Il **LibriController** deve elaborare i dati inseriti. Per farlo, richiede i valori dei campi alla View. Una volta ottenuti i dati, il controller esegue una verifica preliminare (**isVuoto()**) per assicurarsi che i campi obbligatori siano compilati. Qui si aprono due scenari (blocco alternativo **alt**):

- **Input non valido:** Se i dati sono incompleti, il **Controller** comanda alla **View** di mostrare un **Alert di errore** ("Compila tutti i campi...") e il flusso si interrompe.
- **Input valido:** Se i dati sono corretti, il **Controller** procede al parsing (riadatta il formato dei dati, ricevuti come stringhe, in altri più adatti alle operazioni del flusso di esecuzione) dei dati per prepararli all'inserimento.

Sezione 3: Operazioni di Inserimento

Il **LibriController** delega l'operazione di inserimento al **Model** (**GestioneLibri**, classe che incapsula la logica di business e i dati relativi ai libri della biblioteca), inviando il comando **inserisciLibro(...)** con i dati processati. Il **Model** verifica innanzitutto se il libro è già presente in archivio tramite il codice ISBN (**trovaLibro(...)**). Anche qui abbiamo due scenari (**alt**):

- **Libro Già Esistente:** Se il libro viene trovato, il sistema non crea un duplicato ma aggiorna semplicemente il numero di copie totali, sommando le nuove a quelle vecchie.
- **Nuovo Libro:** Se il libro non esiste, il sistema istanzia un nuovo oggetto **Libro** e lo aggiunge alla lista interna gestita dal **Model**.

Sezione 4: Persistenza

Per salvare le operazioni appena effettuate, il **LibriController** invoca la classe **ArchivioFile** (classe utilitaria dedicata alla persistenza dei dati). Viene chiamato il metodo **salvaLibri()**, che si occupa di serializzare l'intero oggetto **GestioneLibri** sul file fisico **libri.dat**.

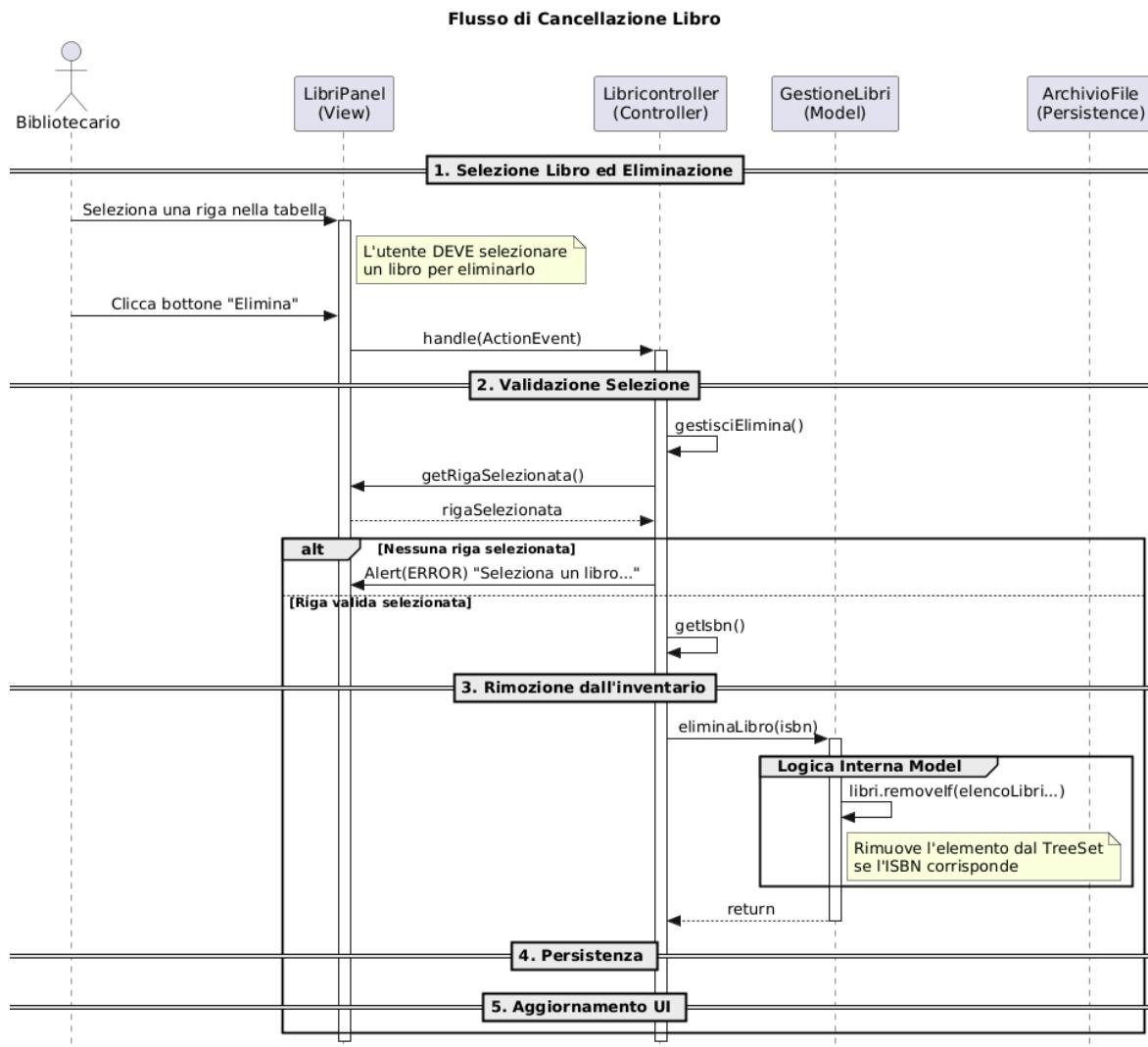
Sezione 5: Aggiornamento UI

Il **LibriController** esegue una routine di pulizia e aggiornamento:

1. Comanda alla **LibriPanel** di svuotare i campi di input per prepararli a un nuovo inserimento.
2. Richiede al Model la lista aggiornata dei libri, ordinata per titolo.
3. Aggiorna la tabella visualizzata nella View.

NOTA: D'ora in poi, ove non strettamente necessario, i flussi di esecuzione relativi a **Persistenza**, **Aggiornamento UI** e operazioni ausiliarie (come il **Parsing**) saranno omessi dalle descrizioni per evitare ridondanze e focalizzare l'attenzione sulla logica core del caso d'uso.

UC-4 Cancellazione Libro



Descrizione

Il diagramma rappresenta il processo di **Cancellazione di un Libro** dall'inventario del sistema di gestione della biblioteca. Il diagramma è diviso in 5 sezioni logiche:

Sezione 1: Selezione Libro ed Eliminazione

L'interazione inizia con il bibliotecario che operando sul **LibriPanel**, seleziona una riga nella tabella dei libri e successivamente clicca sul bottone

"*Elimina*". La View intercetta l'evento e invoca il **LibriController**, avviando la procedura di gestione dell'eliminazione.

Sezione 2: Validazione Selezione

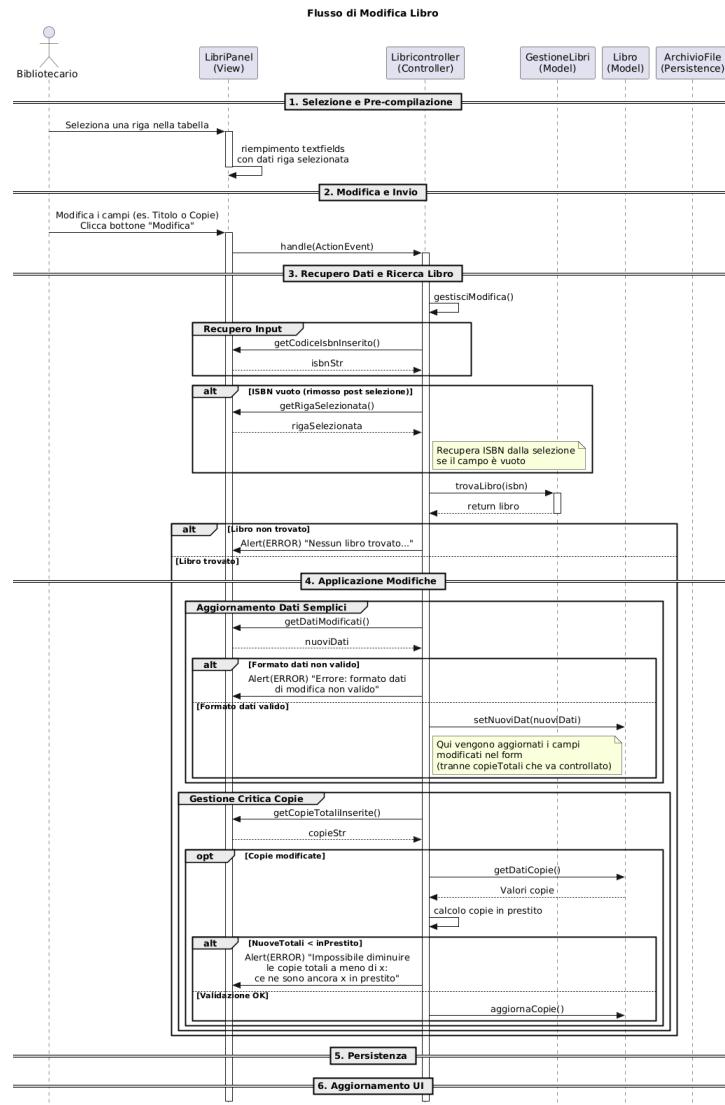
Il **LibriController** avvia la logica interna tramite **gestisciElimina()** e interroga la View tramite **getRigaSelezionata()** per verificare quale elemento è stato scelto. Qui si aprono due scenari basati sullo stato della selezione (blocco **alt**):

- **Nessuna riga selezionata:** Se l'utente ha cliccato "*Elimina*" senza aver selezionato un libro, il Controller comanda alla View di mostrare un **Alert** di tipo ERROR ("Selezione un libro...") e il flusso si interrompe.
- **Riga valida selezionata:** Se la selezione è corretta, il Controller estrae il codice ISBN dalla riga selezionata tramite il metodo **getIsbn()**, necessario per l'operazione successiva.

Sezione 3: Rimozione dall'inventario

Il **LibriController** delega la rimozione al Model (**GestioneLibri**), inviando il comando **eliminaLibro(isbn)** contenente l'identificativo del libro da cancellare. All'interno del Model il sistema cerca e rimuove l'elemento dalla collezione (di tipo **TreeSet**) verificando la corrispondenza dell'ISBN. Al termine, il controllo ritorna al Controller.

UC-3 Modifica Libro



Descrizione

Il diagramma rappresenta il processo di **Modifica dei dati di un Libro** esistente all'interno del sistema di gestione della biblioteca. Il flusso è particolarmente articolato poiché prevede diversi controlli di coerenza sui dati. Il diagramma è diviso in 6 sezioni logiche:

Sezione 1: Selezione e Pre-compilazione

L'interazione inizia sulla **LibriPanel** (View). Quando il bibliotecario seleziona una riga specifica nella tabella dei libri, la **View** reagisce automaticamente pre-compilando i campi di testo con i dati del libro selezionato, facilitando l'operazione di modifica.

Sezione 2: Modifica e Invio

L'utente modifica i valori desiderati nei campi e clicca sul bottone "Modifica". La View intercetta l'evento e invoca il **LibriController**, avviando la logica di gestione.

Sezione 3: Recupero Dati e Ricerca Libro

Il **LibriController** chiama il metodo **gestisciModifica()**. La prima operazione è identificare il libro target:

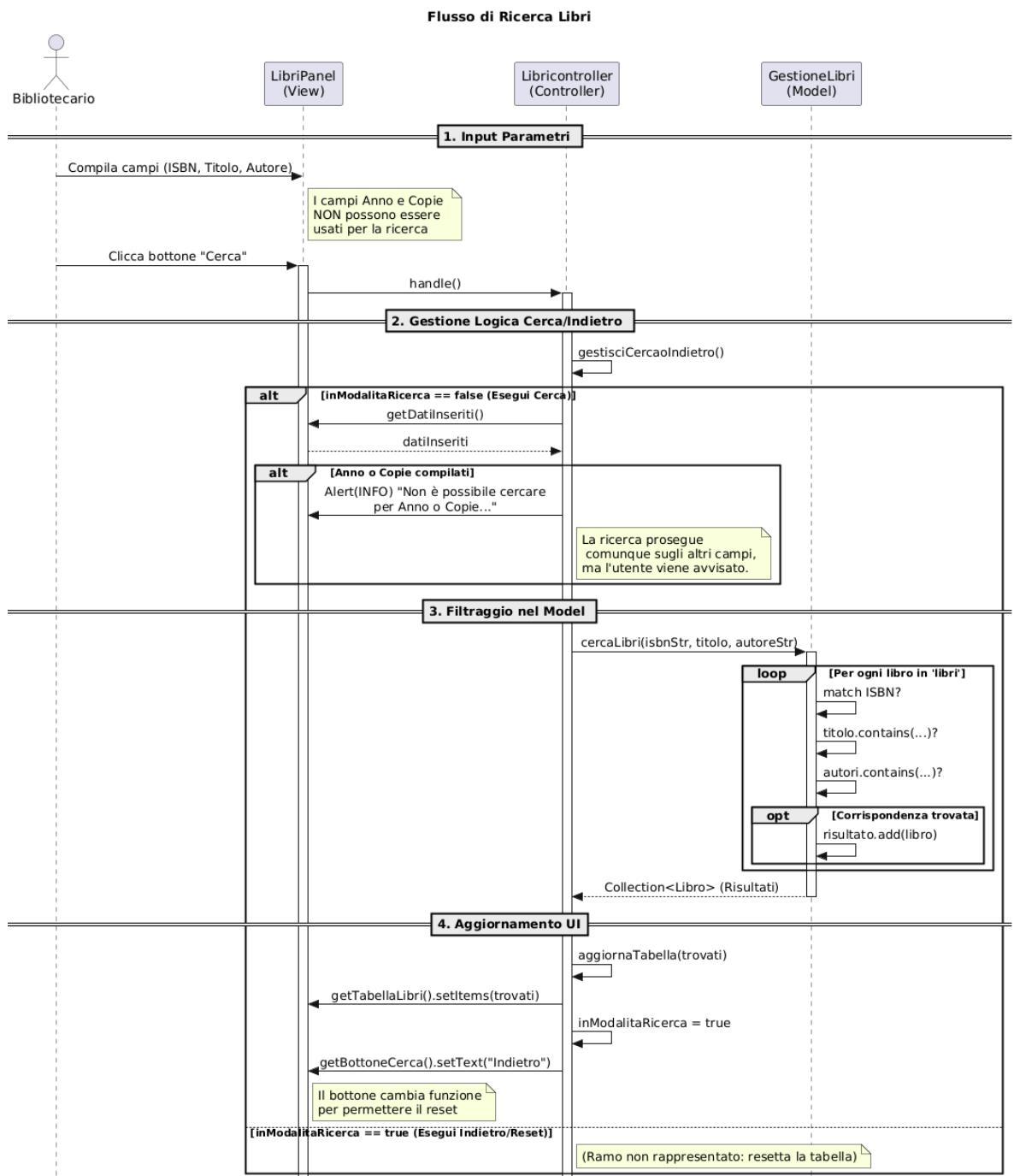
- Tenta di recuperare l'ISBN dal campo di input tramite **getCodiceIsbnInserito()**. Se l'utente ha cancellato l'ISBN dal campo durante la modifica, il Controller recupera l'ISBN originale direttamente dalla riga selezionata in precedenza (**getRigaSelezionata()**).
- Il Controller interroga il Model (**GestioneLibri**) cercando il libro tramite **trovaLibro(isbn)**:
 - ◆ **Libro non trovato (alt):** Se la ricerca fallisce, viene mostrato un Alert di errore e il flusso si interrompe.
 - ◆ **Libro trovato:** Se il libro esiste, si procede alla sezione successiva.

Sezione 4: Applicazione Modifiche

Divisa in due sotto-logiche:

- **Aggiornamento Dati Semplici:** Il Controller recupera i nuovi dati tramite **getDatiModificati()**. Se il formato è valido, i nuovi valori vengono applicati all'oggetto **Libro** tramite **setNuoviDati()**.
- **Gestione Critica Copie:** Il sistema deve garantire la coerenza dell'inventario. Il Controller verifica il nuovo numero di copie totali (**getCopieTotaliInserite()**).
 - ◆ **Controllo Vincoli (alt):** Se le copie sono state modificate, il Model verifica quante copie sono attualmente in prestito con **getDatiCopie()**. Se il bibliotecario tenta di ridurre il numero totale di copie al di sotto del numero di copie attualmente in prestito (es. ridurre a 2 quando 3 sono fuori), il sistema blocca l'operazione mostrando un errore ("Impossibile diminuire le copie totali a meno di X...").
 - ◆ **Validazione OK:** Se il vincolo è rispettato, il numero di copie viene aggiornato con **aggiornaCopie()**.

UC-5 Ricerca Libri



Descrizione

Il diagramma rappresenta il processo di **Ricerca e Filtraggio** dei libri presenti in archivio. Il pulsante di ricerca ha una doppia funzione, alternando tra "*Cerca*" e "*Indietro*". Il diagramma è diviso in 4 sezioni logiche:

Sezione 1: Input Parametri

L'interazione inizia sulla **LibriPanel** (View). Il bibliotecario compila uno o più campi di ricerca (ISBN, Titolo o Autore). L'utente clicca sul bottone "Cerca" e la View intercetta l'evento invocando il **LibriController**.

Sezione 2: Gestione Logica Cerca/Indietro

Il **LibriController** esegue il metodo **gestisciCercaOIndietro()**, che lavora in base allo stato attuale della vista (variabile **inModalitaRicerca**). Qui si apre un blocco **alt**:

- **Ramo Ricerca:** Se l'utente sta avviando una nuova ricerca, il Controller recupera i dati inseriti (**getDatInseriti()**).

- ◆ **Validazione Campi (alt nidificato):** Il sistema controlla se l'utente ha compilato erroneamente i campi "Anno" o "Copie". In tal caso, viene mostrato un **Alert** di tipo INFO ("Non è possibile cercare per Anno o Copie..."), ma il flusso non si interrompe, la ricerca prosegue sugli altri campi validi.

Sezione 3: Filtraggio nel Model

Il **LibriController** delega l'operazione di filtro al Model (**GestioneLibri**), inviando i parametri validi tramite **cercaLibri(...)**. All'interno del Model avviene la logica di matching (blocco **loop**):

- Il sistema itera su ogni libro presente nella lista interna.
- Per ogni libro, verifica la corrispondenza dei criteri (match esatto per ISBN, parziale per Titolo o Autori).
- **Esito Positivo (opt):** Se viene trovata una corrispondenza, il libro viene aggiunto alla lista dei risultati. Infine, il Model restituisce al Controller una **Collection** contenente solo i libri trovati.

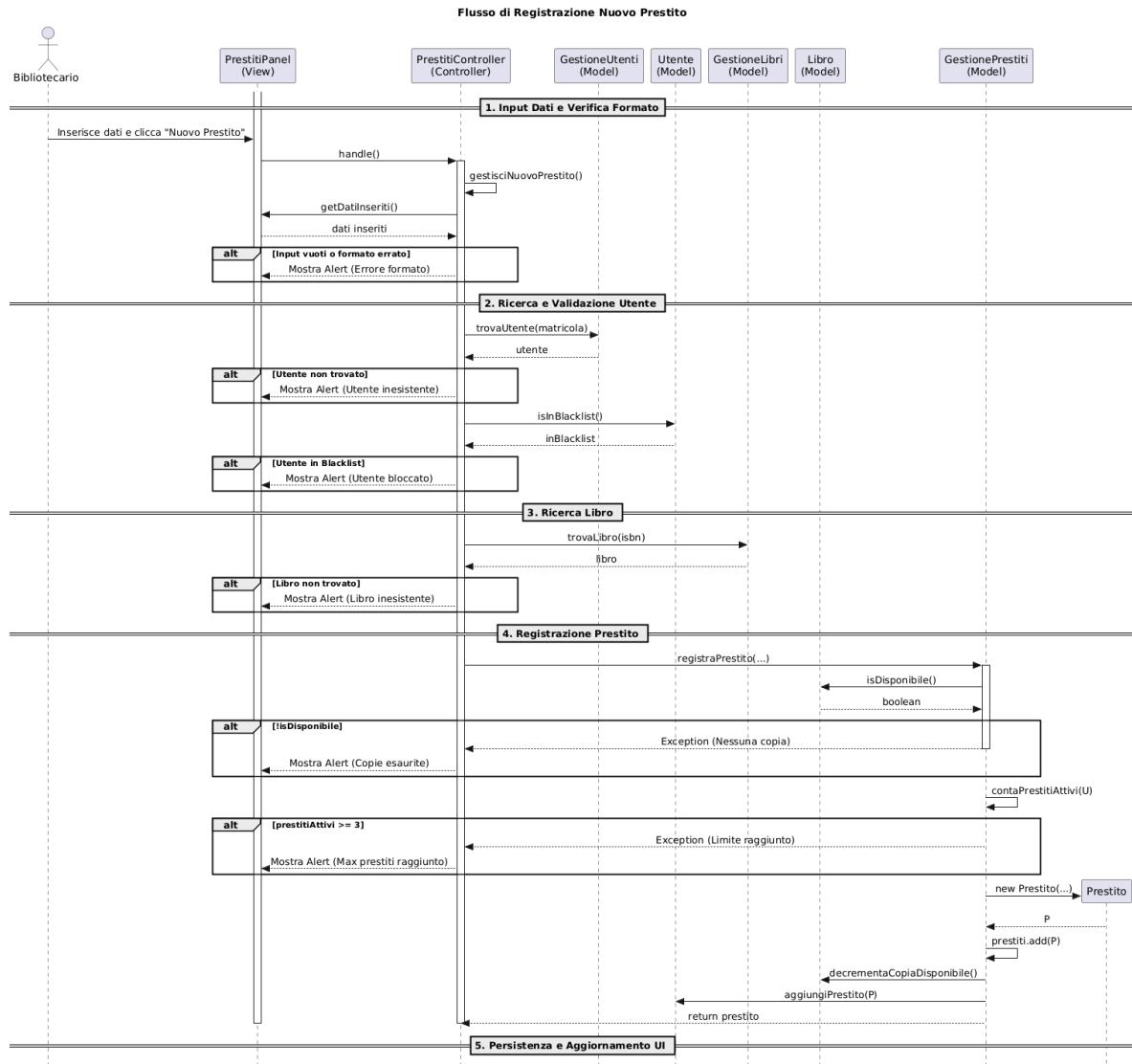
Sezione 4: Aggiornamento UI e Cambio di Stato

Il **LibriController** aggiorna l'interfaccia per mostrare i risultati:

1. Invoca **aggiornaTabella(trovati)** per sostituire il contenuto della tabella nella View con i soli libri filtrati.
2. Imposta il flag **inModalitaRicerca** come true.
3. Cambia il testo del pulsante da "Cerca" a "Indietro". Questo permette all'utente, cliccando nuovamente lo stesso tasto, di resettare la vista e tornare all'elenco completo.

NOTA: I diagrammi di sequenza inerenti alla gestione utenti sono stati omessi data la loro forte somiglianza con quelli appena descritti per i libri..

UC-11 Registrazione Nuovo Prestito



Descrizione

Il diagramma descrive il processo di **creazione di un nuovo prestito** all'interno del sistema di gestione della biblioteca. Il diagramma è diviso in 5 sezioni logiche:

Sezione 1: Input Dati e Verifica Formato

Il bibliotecario inserisce i dati e conferma l'operazione sulla **PrestitiPanel** (Modulo View, Interfaccia grafica della sezione prestiti) cliccando su “**Nuovo prestito**”. La View invoca il **PrestitiController** (Controller della sezione prestiti) che avvia la procedura di inserimento (**gestisciNuovoPrestito()**). Il Controller recupera i dati inseriti (**getDatatiInseriti()**) ed esegue una prima validazione tramite un blocco **alt**:

- **Errore:** Se vengono rilevati campi vuoti o con formato errato, il Controller comanda alla **View** di mostrare un **Alert** ("Errore formato").
- **Proseguo:** Se il formato è corretto, il flusso prosegue alla sezione successiva.

Sezione 2: Ricerca e Validazione Utente

Il **PrestitiController** verifica l'idoneità dell'utente inserito interagendo con il Model:

1. **Ricerca:** Invoca **trovaUtente(matricola)** su **GestioneUtenti** (Classe che gestisce l'elenco degli utenti iscritti alla biblioteca).
 - **alt [Utente non trovato]:** Se l'utente non esiste, viene mostrato un **Alert** ("Utente inesistente") e il flusso si arresta.
2. **Verifica Status:** Se l'utente esiste, il Controller interroga direttamente l'oggetto **Utente** (Oggetto che rappresenta il singolo utente iscritto alla biblioteca) chiamando **isInBlacklist()**.
 - **alt [Utente in Blacklist]:** Se l'utente risulta bloccato, viene mostrato un **Alert** ("Utente bloccato") impedendo il prestito.

Sezione 3: Ricerca Libro

Il Controller verifica l'esistenza del volume chiamando **trovaLibro(isbn)** su **GestioneLibri**. Un blocco **alt** gestisce l'esito:

- **Errore:** Se il libro non viene trovato, viene mostrato un **Alert** ("Libro inesistente").
- **Proseguo:** Se il libro viene trovato, il sistema dispone di tutti i dati per procedere.

Sezione 4: Registrazione Prestito

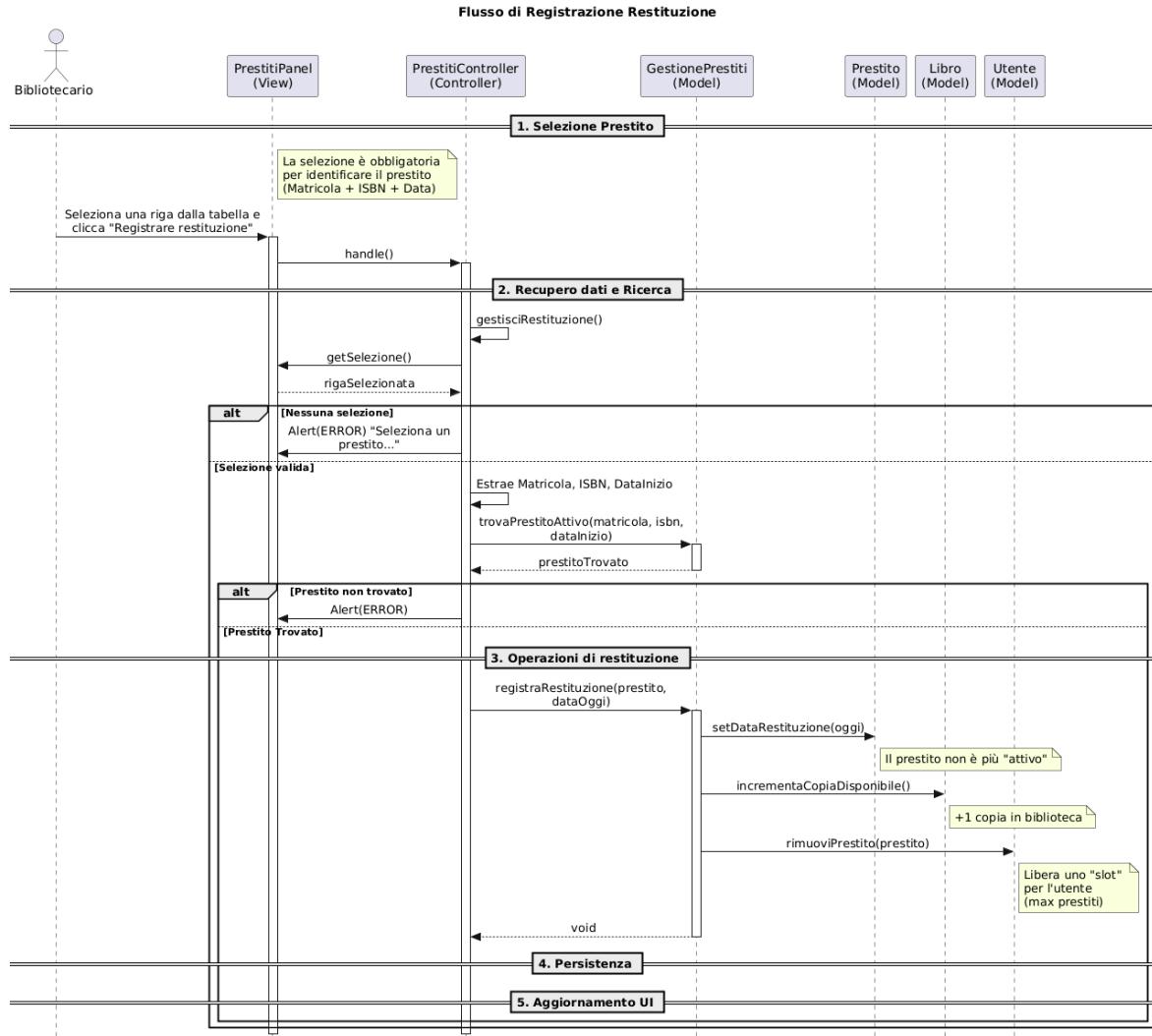
Il Controller delega l'operazione a **GestionePrestiti** tramite **registraPrestito(...)**. Vi sono una serie di controlli gestiti tramite eccezioni (blocchi **alt**):

1. **Disponibilità:** **GestionePrestiti** interroga l'oggetto **Libro** (**isDisponibile()**).
 - **alt [Non disponibile]:** Se non ci sono copie, viene sollevata un'**Exception** ("Nessuna copia") che il Controller cattura mostrando l'errore "Copie esaurite".
2. **Lime Prestiti:** **GestionePrestiti** calcola i prestiti in corso dell'utente (**contaPrestitiAttivi(U)**).
 - **alt [prestiti attivi >= 3]:** Se il limite è superato, viene sollevata un'**Exception** ("Limite raggiunto") visualizzata come errore "Max prestiti raggiunti".

3. **Finalizzazione:** Se nessun errore viene sollevato, il sistema esegue in sequenza:

- Creazione del nuovo oggetto **Prestito**.
- Aggiunta alla lista prestiti (**prestiti.add(P)**).
- Aggiornamento del libro: **decrementaCopiaDisponibile()** sull'oggetto **Libro**.
- Aggiornamento dell'utente: **aggiungiPrestito(P)** sull'oggetto **Utente**.
- Ritorno dell'oggetto **Prestito** al **Controller**.

UC-12 Registrazione Restituzione Prestito



Descrizione

Il diagramma rappresenta il processo di **Restituzione di un Libro** da parte di un utente registrato nel database della biblioteca e con almeno un prestito all'attivo. Il diagramma è diviso in 5 sezioni logiche:

Sezione 1: Selezione Prestito

Sulla **PrestitiPanel** (View), Il bibliotecario consulta la tabella dei prestiti attivi, seleziona la riga corrispondente al prestito da chiudere e clicca sul bottone "Registra restituzione". La **View** intercetta l'evento e invoca il **PrestitiController**.

Sezione 2: Recupero dati e Ricerca

Il **PrestitiController** avvia la procedura di restituzione (**gestisciRestituzione()**). La prima operazione è recuperare l'oggetto della selezione tramite **getSelezione()**. Qui si apre un blocco alternativo (**alt**):

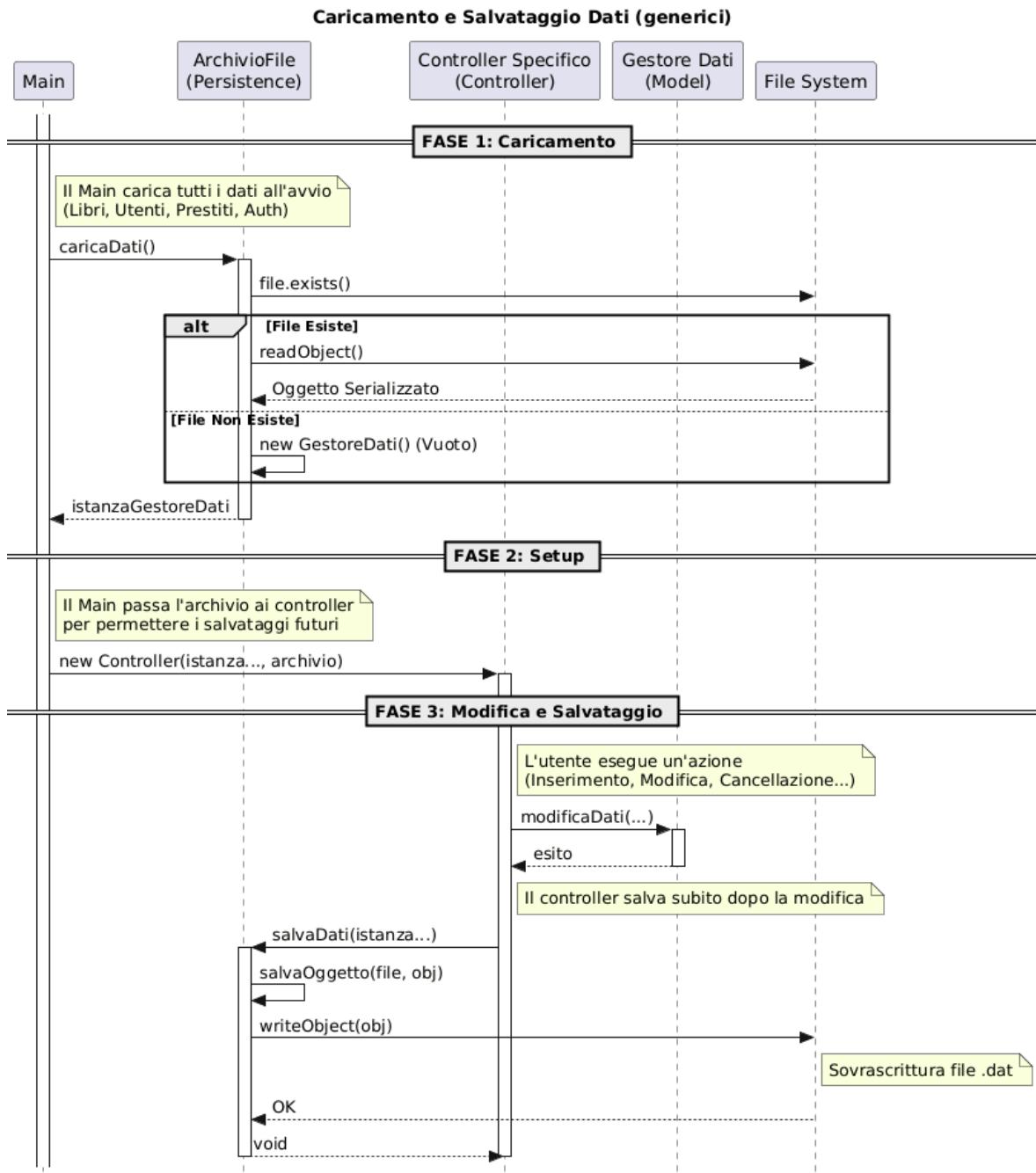
- **Nessuna selezione:** Se il bibliotecario non ha evidenziato alcuna riga, il Controller mostra un **Alert** di errore ("Seleziona un prestito...") e il flusso si interrompe.
- **Selezione valida:** Se i dati sono presenti, il Controller estrae le chiavi per la ricerca (Matricola, ISBN, Data) e interroga il Model (**GestionePrestiti**) tramite il metodo **trovaPrestitoAttivo(...)**.
 - ◆ **Prestito non trovato (alt):** Se il prestito non viene trovato, viene mostrato un errore.
 - ◆ **Prestito Trovato:** Se l'oggetto prestito viene recuperato correttamente, si procede alla restituzione.

Sezione 3: Operazioni di restituzione

Il **PrestitiController** delega l'aggiornamento logico al Model (**GestionePrestiti**), inviando il comando **registraRestituzione(...)**. All'interno del Model avviene una sequenza di aggiornamenti su tre diverse entità:

1. **Aggiornamento Prestito:** Viene settata la data di restituzione sull'oggetto **Prestito** (**setDataRestituzione(oggi)**), rendendolo di fatto "storico" e non più "attivo".
2. **Aggiornamento Libro:** Viene invocato il metodo **incrementaCopiaDisponibile()** sull'oggetto **Libro**, rendendo nuovamente disponibile una copia fisica in biblioteca.
3. **Aggiornamento Utente:** Viene invocato **rimuoviPrestito(prestito)** sull'oggetto **Utente**, liberando uno slot prestito nel profilo.

Caricamento e Salvataggio Dati



Descrizione

Il diagramma descrive le operazioni effettuate dal sistema per gestire la persistenza dei dati all'avvio e durante l'uso dell'applicazione. Il diagramma è diviso in 3 sezioni logiche:

Fase 1: Caricamento

Questa fase avviene all'avvio dell'applicazione. La classe **Main** invoca l'**ArchivioFile** (il componente di persistenza dell'applicazione) chiamando

il metodo **caricaDati()**. Il sistema verifica l'esistenza del file fisico su disco (**file.exists()**). Qui si aprono due scenari (blocco alternativo **alt**):

- **File Esiste:** Se il file .dat è presente, l'**ArchivioFile** procede alla **deserializzazione** tramite **readObject()**, recuperando lo stato salvato nella sessione precedente.
- **File Non Esiste:** Se è il primo avvio o il file è stato rimosso, viene istanziato un nuovo oggetto vuoto (**new GestoreDati()**). Al termine, l'istanza dei dati viene restituita al **Main**.

Fase 2: Setup

Una volta caricati i dati, il **Main** configura l'architettura MVC. Crea le istanze dei **Controller Specifici** (es. **LibriController**, **PrestitiController**) e passa loro, tramite costruttore, sia l'istanza dei dati appena caricata, sia il riferimento all'oggetto **ArchivioFile**, passaggio fondamentale affinché i Controller abbiano accesso ai dati e, soprattutto, abbiano la capacità di salvare le modifiche future.

Fase 3: Modifica e Salvataggio

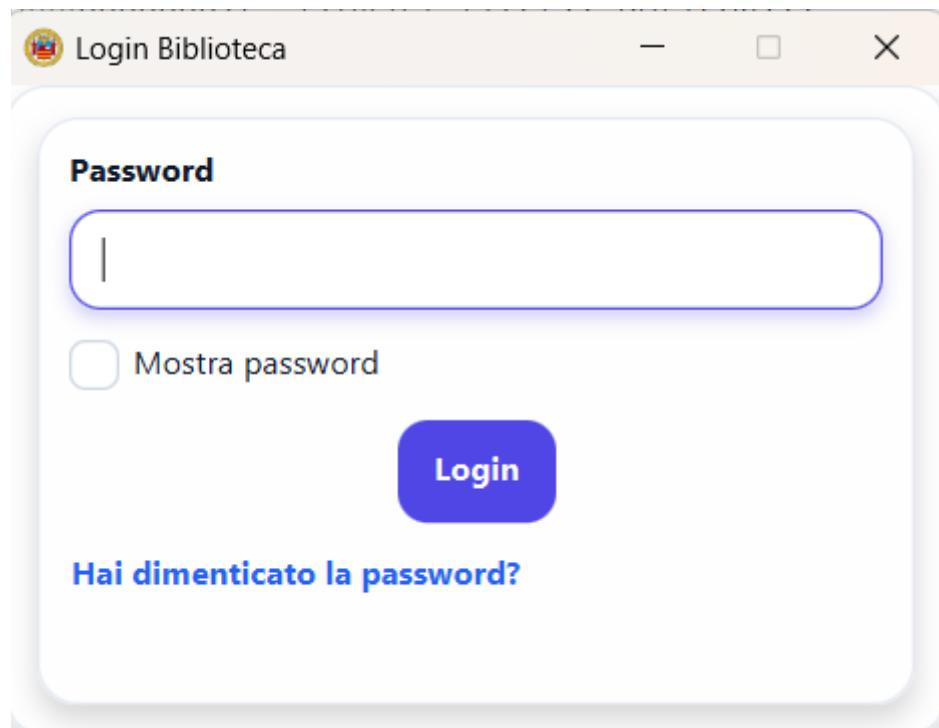
Questa fase è ciclica e avviene durante l'uso dell'applicazione. Quando un utente esegue un'azione che altera lo stato dei dati (es. Inserimento, Modifica o Cancellazione), il **Controller Specifico** invoca il metodo **modificaDati(...)**. Immediatamente dopo la modifica in memoria, il Controller invoca il salvataggio su disco chiamando **salvaDati(...)** su **ArchivioFile**. Quest'ultimo esegue la **serializzazione** dell'intero oggetto tramite **writeObject(obj)**, sovrascrivendo il file .dat esistente e garantendo la persistenza immediata delle modifiche.

4. User Interface

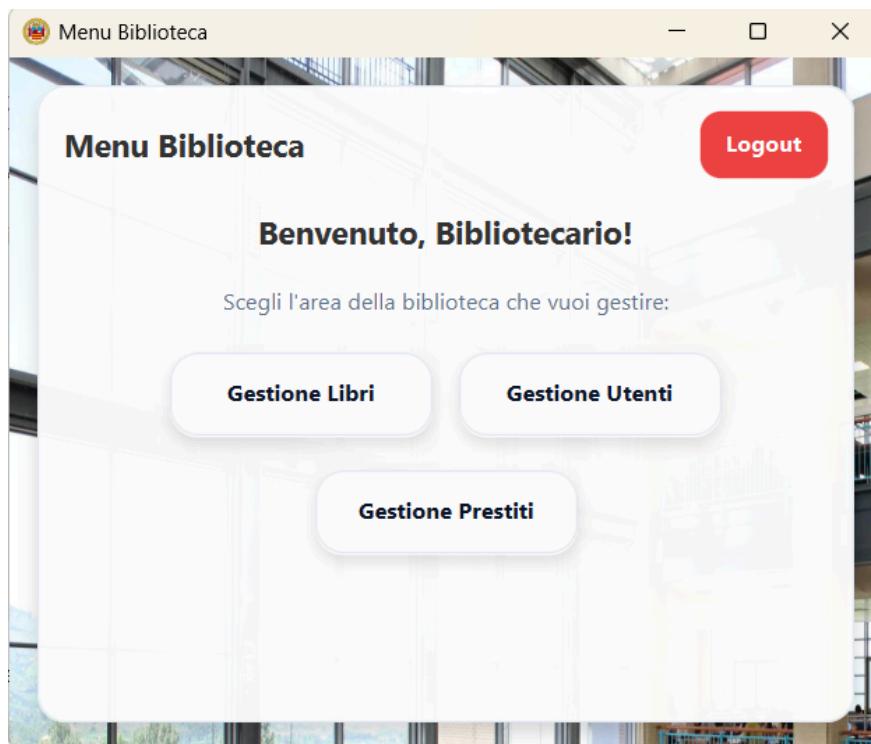
In questa sezione viene mostrata una sequenza di screen nei quali sono visualizzate le possibili interazioni dell'interfaccia.

4.1 Login

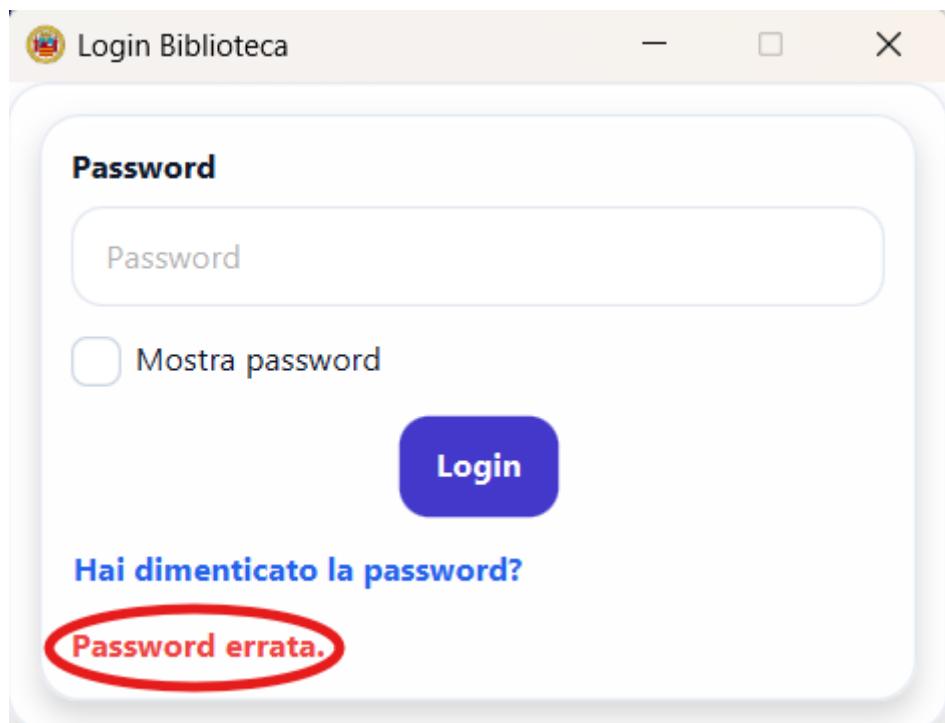
Prima: l'utente inserisce nel textField una sequenza di caratteri che vengono mascherati .



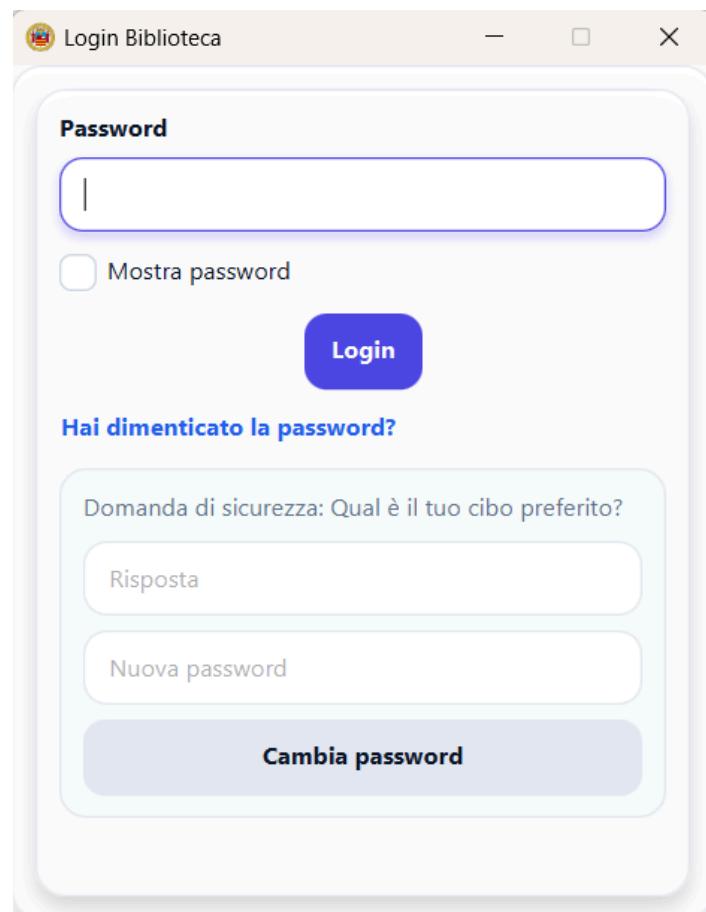
Dopo: se la password è corretta, l'autenticazione va a buon fine e viene aperta la finestra del menu.



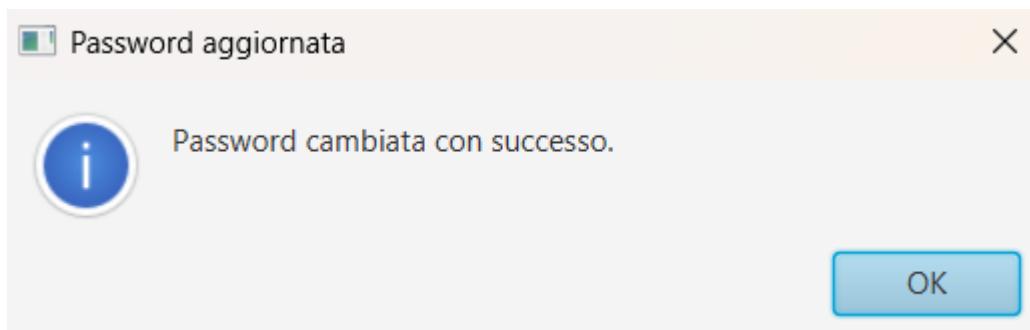
Altrimenti: se la password è errata la finestra di login rimane visibile e viene mostrato un messaggio di errore.



Scenario dopo aver cliccato la voce “**Hai dimenticato la password?**”



Happy path: Rispondo correttamente alla domanda di sicurezza e procedo a reimpostare la password



Error path: Fornisco la risposta errata dunque non posso modificare la password.

The screenshot shows a window titled "Login Biblioteca". Inside, there's a "Password" section with a text input field containing "Password", a "Mostra password" checkbox, and a blue "Login" button. Below this is a link "Hai dimenticato la password?". A secondary panel displays a security question: "Domanda di sicurezza: Qual è il tuo cibo preferito?" with two options: "pasta" and ".....". A large grey button labeled "Cambia password" is at the bottom. At the very bottom, a red oval highlights the text "Risposta di sicurezza errata."

— X

Password

Password

Mostra password

Login

[Hai dimenticato la password?](#)

Domanda di sicurezza: Qual è il tuo cibo preferito?

pasta

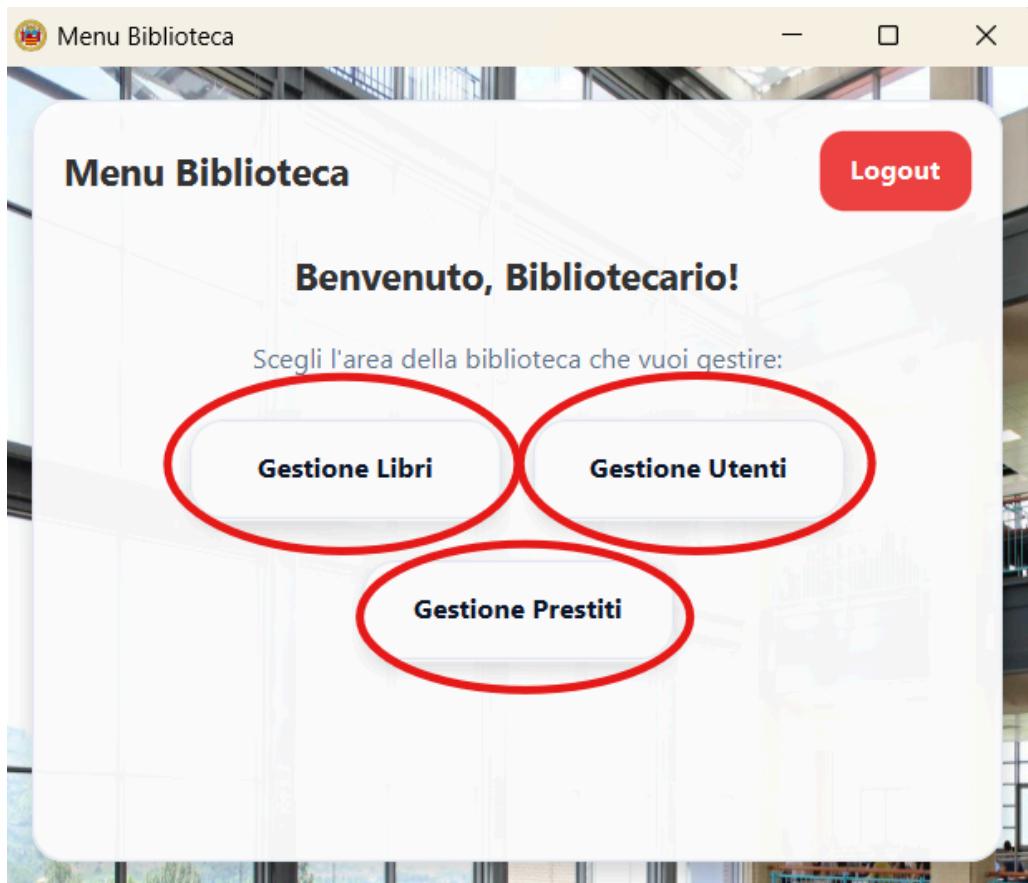
.....

Cambia password

Risposta di sicurezza errata.

4.2 Navigazione dal menu principale

Prima: Viene mostrata la finestra **Menu Biblioteca**, con i pulsanti **Gestione Libri**, **Gestione Utenti**, **Gestione Prestiti** e **Logout**.



Dopo: Se il bibliotecario clicca su uno dei **pulsanti di gestione**, si apre la corrispondente schermata.

4.3 Gestione Libri

The screenshot shows a window titled 'Gestione Biblioteca' with a background image of a library interior. At the top, there's a menu bar with 'Biblioteca universitaria' and a 'Logout' button. Below the menu, there are three tabs: 'Libri' (selected), 'Utenti', and 'Prestiti'. In the center, there are search fields for ISBN, Titolo, Autore/i, and Anno, along with a field for Copie totali. Below these fields is a table displaying book details:

ISBN	Titolo	Autori	Anno	Copie totali	Copie disp.
978880000002	ciao	i	2024	1	1
978880000001	io e te	Mirko alessandrini	2021	2	1

At the bottom right of the table are four buttons: 'Inserisci' (blue), 'Modifica' (grey), 'Elimina' (red), and 'Cerca' (light blue).

In questa sezione possiamo visionare tutti i dettagli dei libri presenti nel sistema: ISBN, titolo, Autori, Anno di pubblicazione, Copie totali e Copie disponibili per il prestito.

4.4: Gestione Utenti

Biblioteca universitaria

Gestione Biblioteca

Logout

Libri Utenti Prestiti

Matricola: Nome:

Cognome: Email:

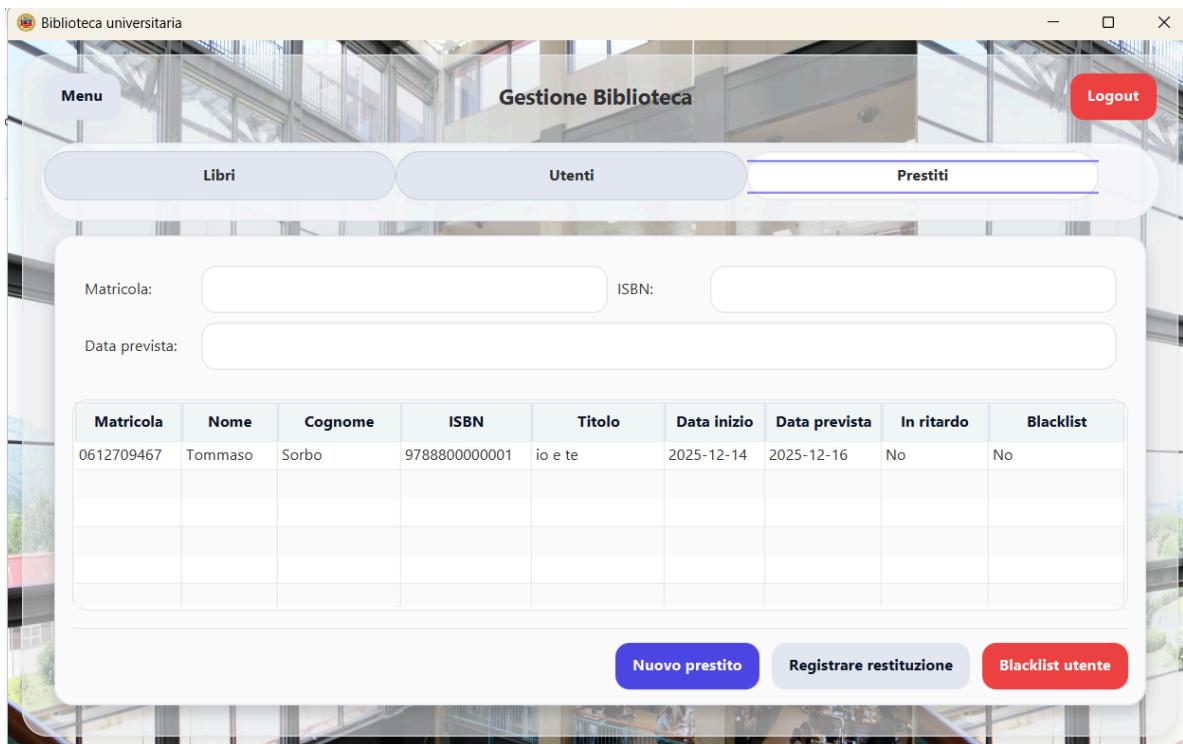
Matricola	Nome	Cognome	Email	Prestiti attivi	Blacklist
0612709456	Carmine	Esposito	c.esposito@unisa.it	0	No
0612709467	Tommaso	Sorbo	t.sorbo@unisa.it	0	No

Inserisci Modifica Elimina Blacklist Cerca

In questa sezione è possibile visionare tutti i dettagli della sezione utenti : Matricola, Nome, Cognome, Email, Prestiti Attivi per quell'utente e Blacklist(se l'utente è o meno nella lista nera).

4.5: Gestione Prestiti

In questa sezione è possibile avere una panoramica sui prestiti presenti nel sistema tramite Matricola, Nome, Cognome, ISBN, Titolo del libro, Data di inizio del prestito, data prevista per la restituzione, se la restituzione è o meno in ritardo e se l'utente è o meno nella blacklist.

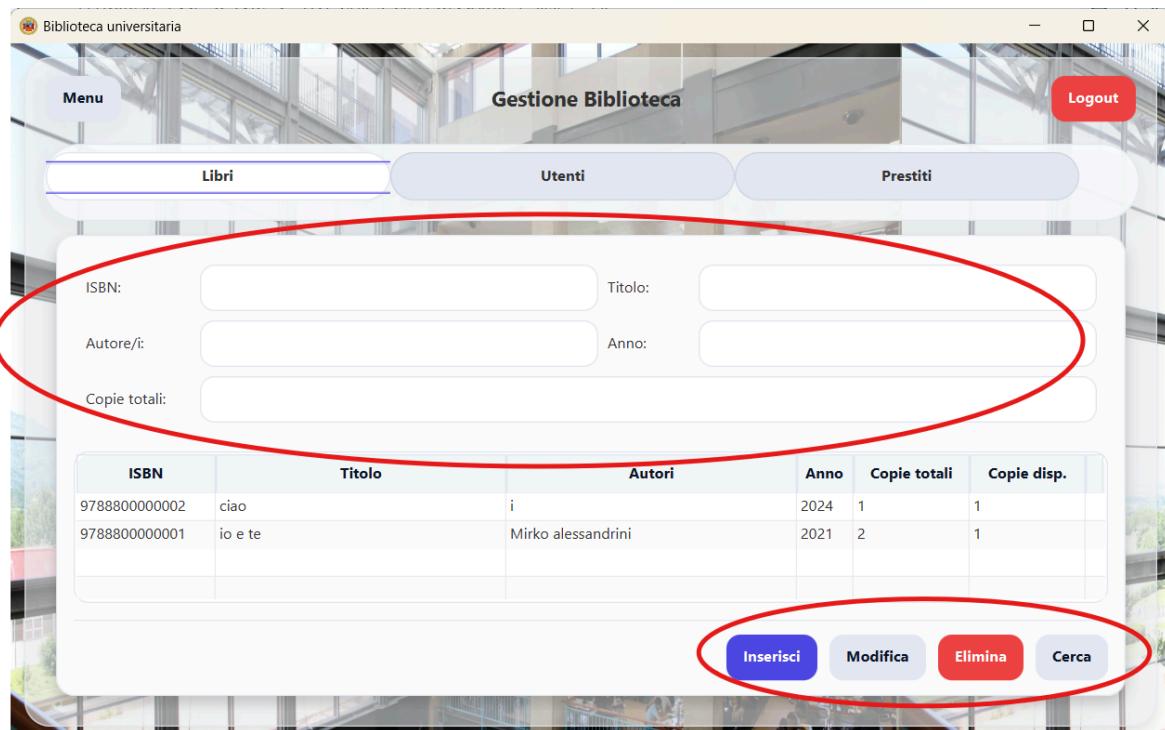


LOGOUT

se il bibliotecario clicca su **logout**, la finestra del menu si chiude e l'applicazione torna alla schermata di login (4.1).

→ 4.3.1: Panoramica scheda gestione libri

Prima: Nella scheda **Libri** il bibliotecario inserisce nei campi ISBN, Titolo, Autore, Anno e Copie totali i dati del libro (oppure seleziona un libro dalla tabella) e clicca sul pulsante **Inserisci**, **Modifica**, **Elimina** oppure **Cerca**.



Dopo: La tabella dei libri viene aggiornata: in caso di inserimento o modifica mostra la nuova riga con i dati corretti, in caso di eliminazione il libro scompare dall'elenco, mentre con **Cerca** rimangono visibili solo i volumi che soddisfano i criteri indicati.

→ 4.3.2 Inserimento libro

Nota: dopo aver cliccato il tasto "**Inserisci**", i libri verranno visualizzati in ordine di titolo.

The screenshot shows the 'Gestione Biblioteca' interface. At the top, there are three tabs: 'Libri' (selected), 'Utenti', and 'Prestiti'. Below the tabs, there are input fields for ISBN, Titolo, Autore/i, and Anno. A table lists books with columns for ISBN, Titolo, Autori, Anno, Copie totali, and Copie disp. The first book in the table is circled in red. At the bottom right of the table, the 'Inserisci' button is highlighted with a red circle.

ISBN	Titolo	Autori	Anno	Copie totali	Copie disp.
9788800000003	Autobiografia di William	William	2004	11	11
9788800000002	ciao	i	2024	1	1
9788800000001	io e te	Mirko alessandrini	2021	2	1

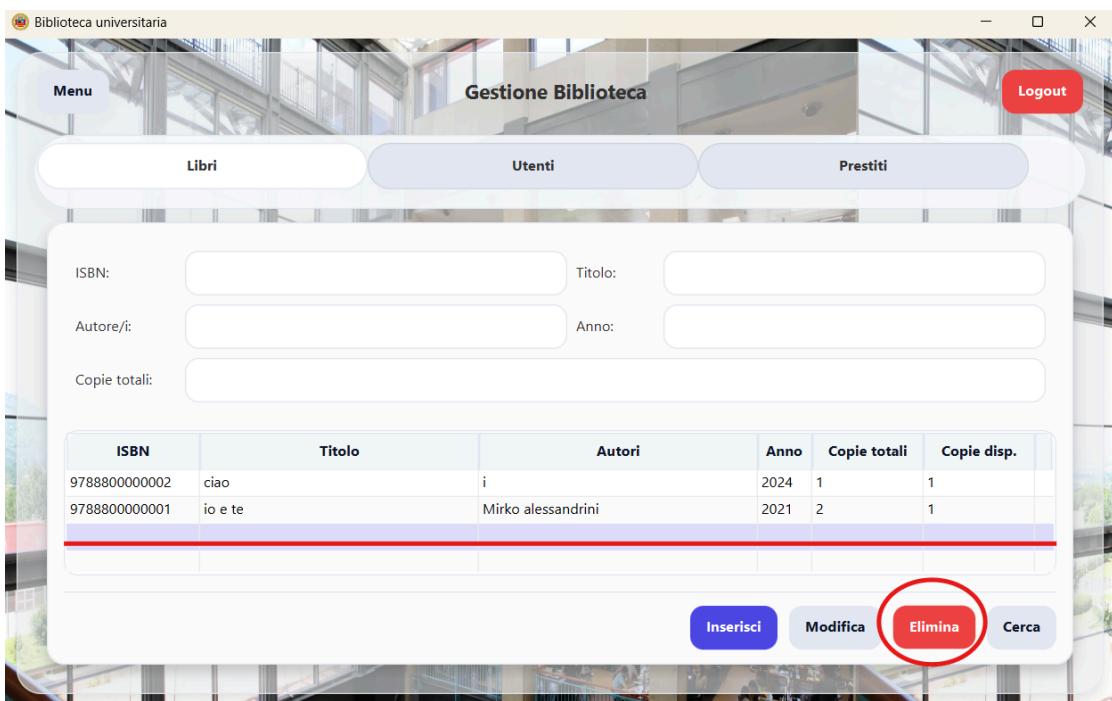
Inserisci **Modifica** **Elimina** **Cerca**

The screenshot shows the 'Gestione Biblioteca' interface after an insertion. The table now has four entries. The second book from the top is circled in red. The 'Modifica' button at the bottom right of the table is highlighted with a red circle.

ISBN	Titolo	Autori	Anno	Copie totali	Copie disp.
9788800000003	Autobiografia di Tommaso	Tommaso	2025	11	11
9788800000002	ciao	i	2024	1	1
9788800000001	io e te	Mirko alessandrini	2021	2	1

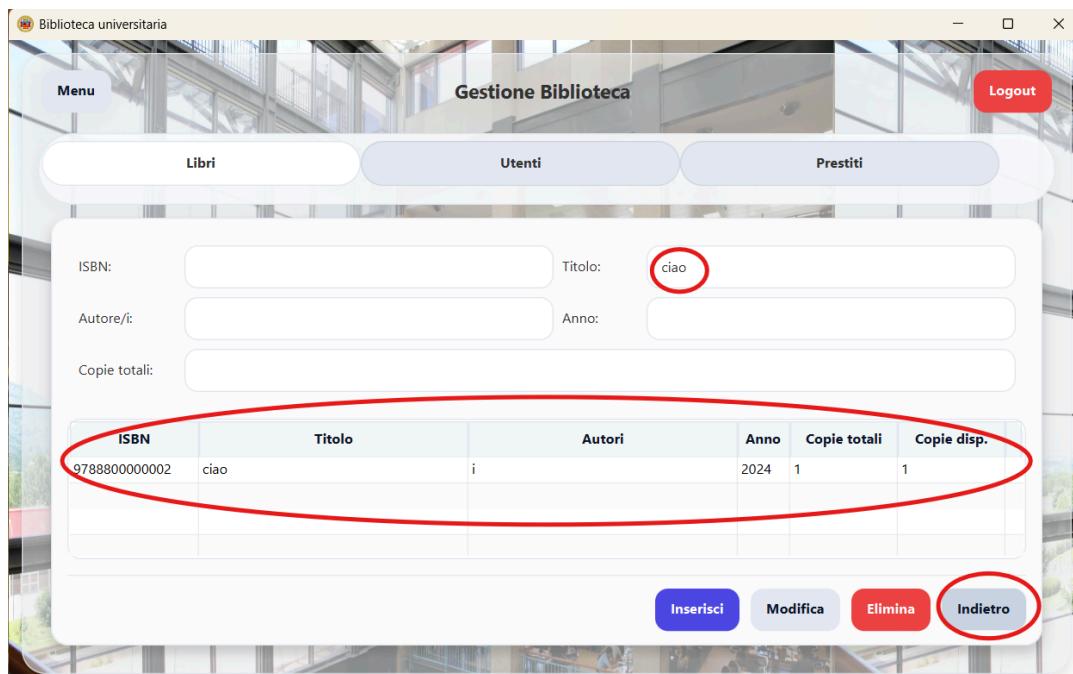
Inserisci **Modifica** **Elimina** **Cerca**

→ 4.3.4 Eliminazione libro



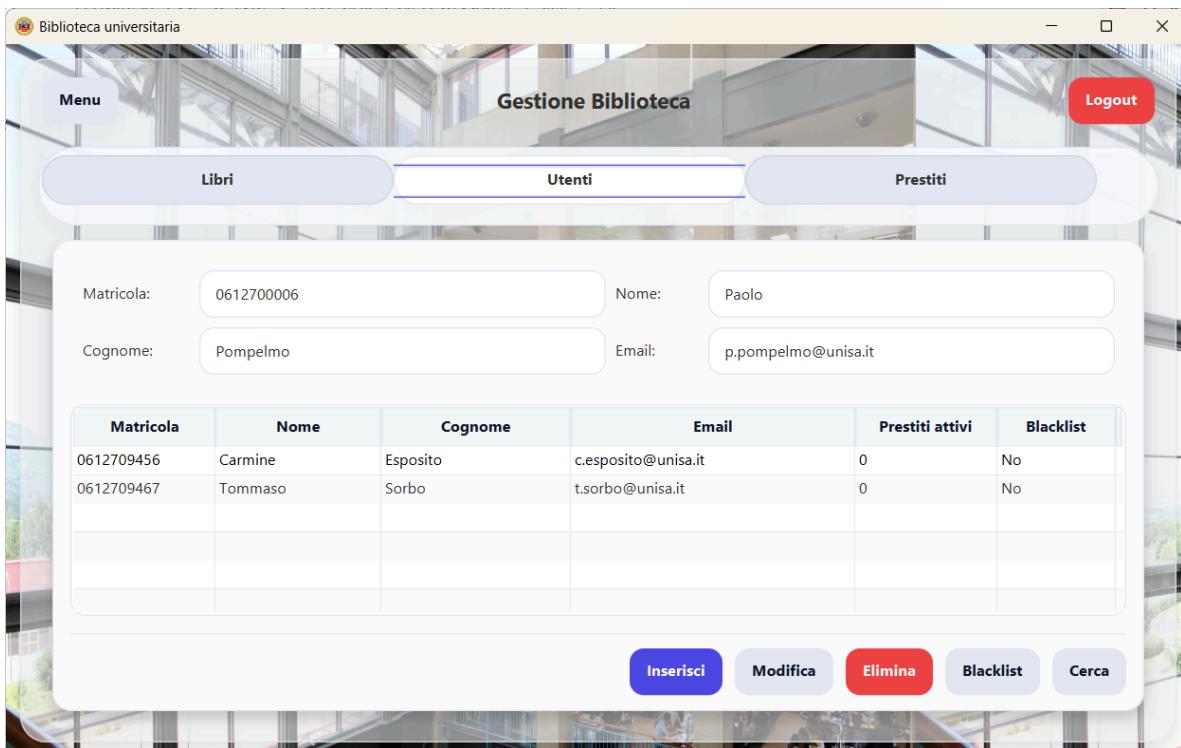
→ 4.3.5 Ricerca libro

Nota: per cercare un libro è sufficiente inserire il titolo, l'autore o l'ISBN e cliccare su "**Cerca**". I risultati verranno mostrati in ordine di titolo. Quando vogliamo smettere di cercare, il pulsante "Cerca" diventa "**Indietro**".



→ 4.4.1 Panoramica scheda gestione utenti

Prima: Nella scheda **Utenti** il bibliotecario compila i campi Matricola, Nome, Cognome ed Email (oppure seleziona una riga dalla tabella) e clicca su **Inserisci**, **Modifica**, **Elimina**, **Blacklist** o **Cerca**.



Dopo: La tabella degli utenti viene aggiornata: viene aggiunta o modificata la riga scelta, in caso di eliminazione l'utente scompare, mentre con **Cerca** restano visualizzati solo gli utenti che soddisfano i dati inseriti (compreso lo stato **Blacklist**).

→ 4.4.2 Inserimento utente

Nota: l'inserimento dell'utente avviene in ordine di **cognome** e **nome**.

Biblioteca universitaria

Menu

Gestione Biblioteca

Logout

Libri Utenti Prestiti

Matricola: Nome:

Cognome: Email:

Matricola	Nome	Cognome	Email	Prestiti attivi	Blacklist
0612709456	Carmine	Esposito	c.esposito@unisa.it	0	No
0612700006	Paolo	Pompelmo	p.pompelmo@unisa.it	0	No
0612709467	Tommaso	Sorbo	t.sorbo@unisa.it	0	No

Inserisci Modifica Elimina Blacklist Cerca

→ 4.4.3 Modifica utente

Nota: per modificare un utente è sufficiente selezionare l'utente, oppure inserire i suoi dati (con la stessa matricola) e fare clic su "**Modifica**".

Biblioteca universitaria

Menu

Gestione Biblioteca

Logout

Libri Utenti Prestiti

Matricola: Nome:

Cognome: Email:

Matricola	Nome	Cognome	Email	Prestiti attivi	Blacklist
0612709456	Carmine	Esposito	c.esposito@unisa.it	0	No
0612700006	Pasquale	Pompelmo	p.pompelmo@unisa.it	0	No
0612709467	Tommaso	Sorbo	t.sorbo@unisa.it	0	No

Inserisci Modifica Elimina Blacklist Cerca

→ 4.4.4 Eliminazione utente

Biblioteca universitaria

Gestione Biblioteca

Logout

Libri Utenti Prestiti

Matricola: Nome: Cognome: Email:

Matricola	Nome	Cognome	Email	Prestiti attivi	Blacklist
0612709456	Carmine	Esposito	c.esposito@unisa.it	0	No
0612709467	Tommaso	Sorbo	t.sorbo@unisa.it	0	No

Inserisci Modifica Elimina Blacklist Cerca

→ 4.4.5 Blacklist utente

Biblioteca universitaria

Gestione Biblioteca

Logout

Libri Utenti Prestiti

Matricola: 0612709456 Nome: Carmine

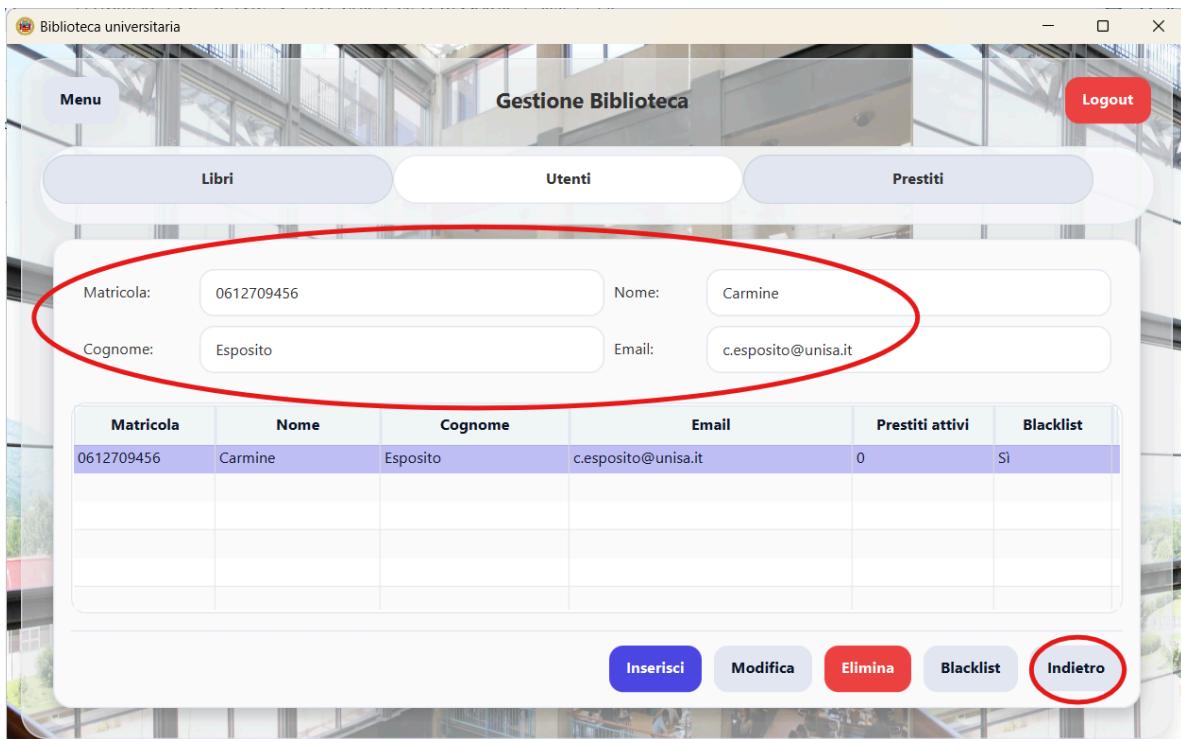
Cognome: Esposito Email: c.esposito@unisa.it

Matricola	Nome	Cognome	Email	Prestiti attivi	Blacklist
0612709456	Carmine	Esposito	c.esposito@unisa.it	0	Si
0612709467	Tommaso	Sorbo	t.sorbo@unisa.it	0	No

Inserisci Modifica Elimina Blacklist Cerca

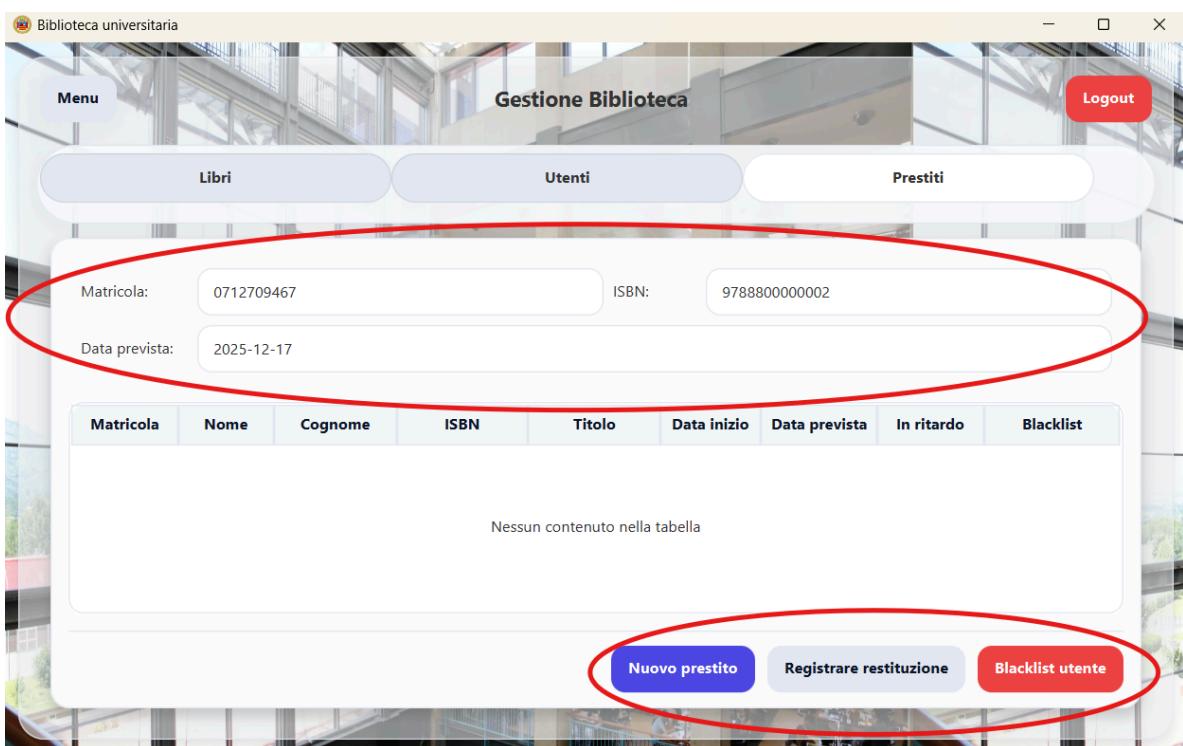
→ 4.4.6 Ricerca utente

Nota: per cercare un utente è sufficiente inserire il nome, il cognome o la matricola e cliccare su "**Cerca**". I risultati verranno mostrati in ordine di cognome. Quando vogliamo smettere di cercare, il pulsante "**Cerca**" diventa "**Indietro**".



→ 4.5.1 Panoramica scheda gestione prestiti

Prima: Nella scheda **Prestiti** il bibliotecario inserisce Matricola, ISBN e Data prevista, oppure seleziona un prestito dalla tabella, e preme **Nuovo prestito**, **Registrare restituzione** o **Blacklist utente**.



Dopo: La tabella dei prestiti viene aggiornata: viene aggiunta una nuova riga per il prestito registrato, contrassegnato con le date corrette e la colonna **In ritardo**, oppure il prestito selezionato viene chiuso/aggiornato o il relativo utente viene marcato in blacklist.

→ 4.5.2 Inserimento di un nuovo prestito

Biblioteca universitaria

Menu Gestione Biblioteca Logout

Libri Utenti Prestiti

Matricola: ISBN:

Data prevista:

Matricola	Nome	Cognome	ISBN	Titolo	Data inizio	Data prevista	In ritardo	Blacklist
0612709467	Tommaso	Sorbo	9788800000002	ciao	2025-12-14	2025-12-17	No	No

Nuovo prestito Registrare restituzione Blacklist utente

→ 4.5.3 Registrazione della restituzione

Biblioteca universitaria

Menu Gestione Biblioteca Logout

Libri Utenti Prestiti

Matricola: ISBN:

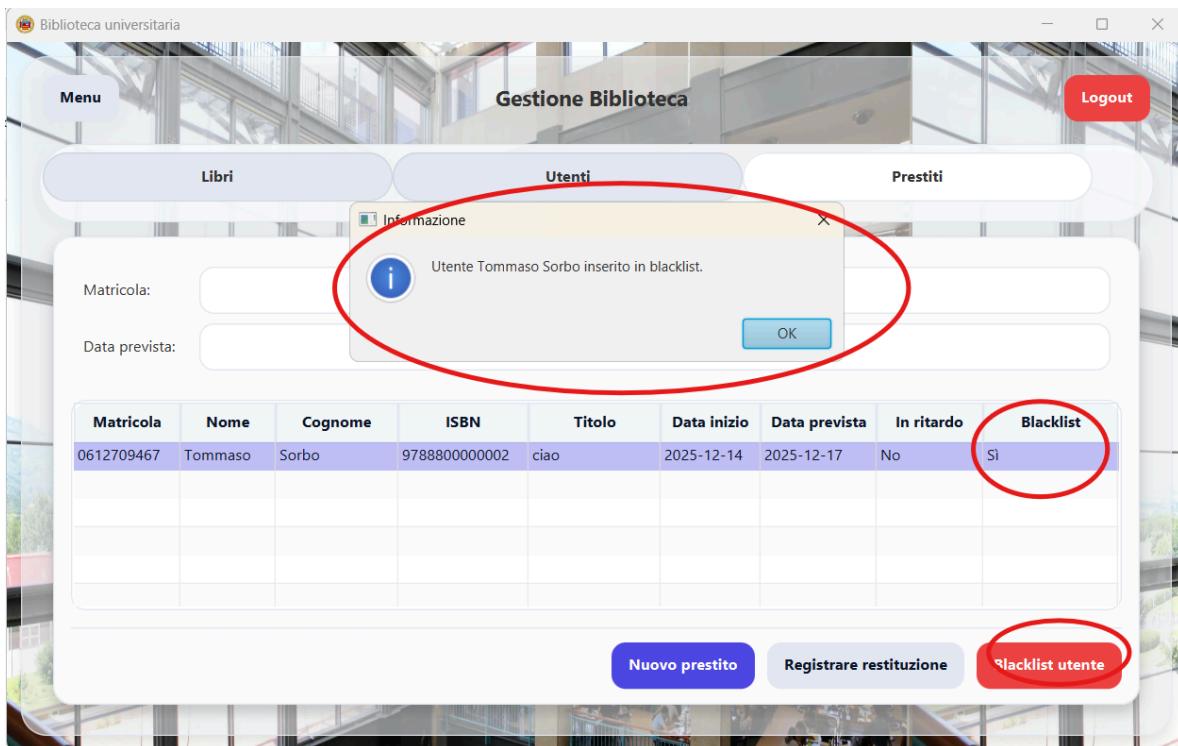
Data prevista:

Matricola	Nome	Cognome	ISBN	Titolo	Data inizio	Data prevista	In ritardo	Blacklist
Nessun contenuto nella tabella								

Nuovo prestito Registrare restituzione Blacklist utente

→ 4.5.4 Gestione blacklist utente

Nota: nel momento in cui un utente viene inserito in blacklist, viene aggiornato automaticamente anche nella schermata "Utenti".



4.6 Barra superiore: Menu e Logout

Prima: In tutte le finestre di gestione è presente la barra superiore con il pulsante **Menu** e il pulsante **Logout**.



Dopo: Se il bibliotecario clicca su **Menu**, l'applicazione chiude la schermata corrente e ritorna al **Menu Biblioteca**(4.2).

5. CSS

Per migliorare la qualità dell'interfaccia grafica abbiamo introdotto un foglio di stile file.css applicato alle schermate JavaFx.

L'abbiamo fatto per valorizzare le conoscenze acquisite durante il corso di Tecnologie del Web in questo modo l'interfaccia risulta più leggibile e più semplice anche da modificare in futuro.