

# Library Manager

## Progetto Software



Autore: *Gabriele Vianello*

Anno Accademico 2024 – 2025  
Politecnico di Milano



# Indice

<i>Introduzione</i> .....	4
<i>Architettura generale</i> .....	4
<i>Base di Dati</i> .....	6
<i>Comunicazione</i> .....	6
<i>MVC (Model-View-Controller)</i> .....	7
<i>DAO (Data Access Object)</i> .....	7
<i>Server</i> .....	11
<i>LibraryApplication</i> .....	13
<i>Interfaccia grafica</i> .....	15
<i>Tecnologie utilizzate</i> .....	18
<i>Test &amp; Coverage</i> .....	18
<i>Configurazione e Installazione</i> .....	19
<i>Considerazioni finali</i> .....	19

# Introduzione

Il progetto prevede la realizzazione di un applicativo software, denominato *Library Manager*, che permetta di gestire una biblioteca. Per gestione si intende l'aggiunta, modifica, ricerca, eliminazione di libri e clienti, così come per i prestiti dei libri clienti; le informazioni saranno gestite da un istanza locale di database SQL relazionale. La soluzione da realizzata dovrà essere in grado di operare su differenti sistemi operativi/piattaforme<sup>1</sup>, gestendo anche più connessioni simultaneamente. L'interfaccia console sarà utilizzata per messaggi di debug e log (sia lato server che client), mentre la GUI sarà realizzata utilizzando JavaFX.

Un importante requisito che il sistema dovrà rispettare sarà quello della disconnessione sicura dove, in presenza di un malfunzionamento o mancata connettività di un client, il server continuerà a rispondere alle richieste dei client ancora attivi.

Saranno realizzati appositi test di funzionamento per i modelli e tutte quelle classi che saranno predisposte per la loro manipolazione. Ulteriore documentazione, come JavaDoc ed il diagramma UML completo del sistema si trovano nell'apposita cartella all'interno del progetto.

# Architettura generale

Il sistema si compone di due eseguibili, uno denominato *Server*, sempre attivo e pronto ad accettare nuove connessioni da parte di applicativi client. L'eseguibile lato utente è *LibraryApplication* che avvia l'applicazione client basata su JavaFX.

Per garantire la modularità del sistema sono stati adottati il DAO Pattern per l'accesso al database e l'MVC per un'interazione ottimale ed efficiente sia con utente finale che con il server.

Le entità previste ed implementate sono *Books*, *Lends* e *Customers*, le quali rappresentano rispettivamente libri, prestiti e clienti. Contengono attributi, getters/setters e metodi per l'elaborazione dei dati.

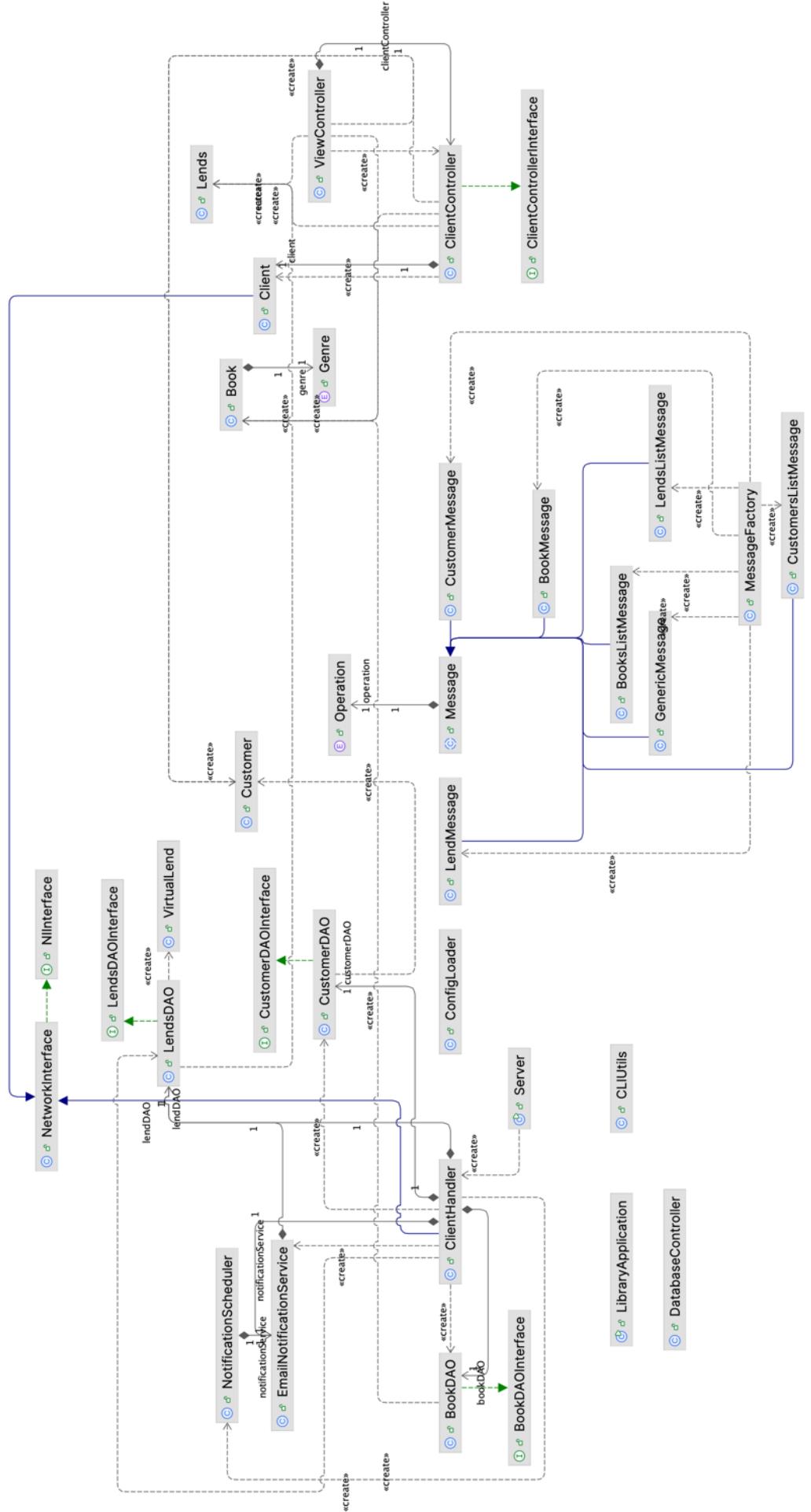
Ogni connessione server ↔ client è indipendente e viene gestita lato server come un thread che riceve e risponde alle richieste del client assegnatogli. Le richieste/risposte sfrutteranno un apposito sistema di messaggistica che permette di scambiare oggetti od altre informazioni.

Le operazioni previste dovranno essere richieste dal client al server, il tutto passando per delle classi intermedie necessarie all'elaborazione dei dati, incapsulando la richiesta in un messaggio che sarà serializzato e spedito al server. Allo stesso modo, le risposte in ingresso ripercorrono la stessa strada ma al contrario; in quest'ultimo caso, a rimanere sarà l'oggetto richiesto.



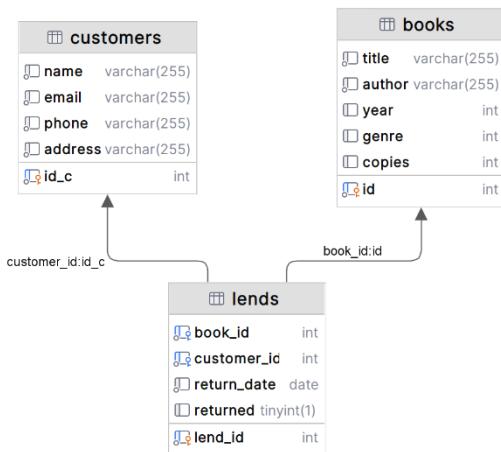
---

<sup>1</sup> A patto che abbiano una Java Virtual Machine installata.



## Base di Dati

L'infrastruttura di salvataggio dati utilizzata è un'istanza server di MySQL 8, dove sono state create le tre tabelle rappresentative dei modelli.



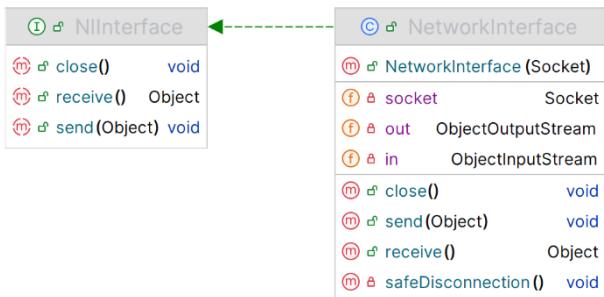
Per l'indicizzazione all'interno del database è stato scelto di utilizzare come chiave primaria un intero auto-incrementante; è possibile distinguerlo nelle tabelle come l'attributo denominato con "id".

Clienti e libri sono associati tra loro attraverso la tabella dei prestiti, dove sono presenti le relative chiavi esterne "book\_id" e "customer\_id".

È stato aggiunto nelle fasi finali di progettazione l'attributo "returned", necessario per distinguere i prestiti inattivi (libro che è già stato riconsegnato) da quelli attivi (non restituiti).

## Comunicazione

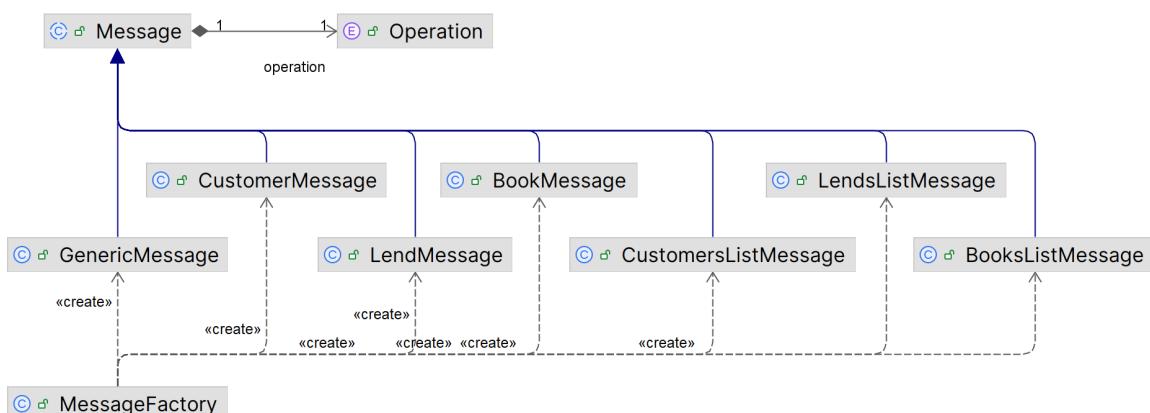
Per garantire una comunicazione efficace tra applicazione client e server, è stato implementato un sistema di messaggistica basato su oggetti serializzabili e caratterizzati da un'operazione ed un contenuto.



Quindi i modelli dei libri, utenti prestiti (o qualunque altra struttura dati) possono essere facilmente trasmessi attraverso `ObjectInputStream` e  `ObjectOutputStream`.

Saranno i metodi `send(Object)` e `receive()` a rappresentare rispettivamente output ed input per server e client.

Dall'UML si può osservare come tutti i messaggi estendano la classe astratta `Message`; la distinzione nei differenti tipi di messaggio avviene per operazione e soprattutto per tipo di contenuto che si vuole trasmettere. Ad esempio, `BookMessage` avrà come `message` (attributo che contiene l'oggetto da inviare) un oggetto di tipo `Book`, mentre `BooksListMessage` sarà utilizzato per inviare `ArrayList<Book>()`. Inoltre, per costruire dinamicamente i messaggi viene adottato il *Factory design pattern*.



## MVC (Model-View-Controller)

L'MVC è un pattern architetturale utilizzato principalmente nelle applicazioni con interfaccia utente (UI). Divide l'applicazione in tre componenti principali:

1. **Model (Modello):** rappresenta la logica di business e i dati dell'applicazione. È responsabile della gestione dello stato dell'applicazione dove gli oggetti rappresentano entità, che in questo caso saranno *libri*, *prestiti* o *clienti*.
2. **View (Vista):** è la parte che gestisce l'interfaccia utente, mostra i dati all'utente e invia i comandi dell'utente al *Controller*; in Library Manager sarà la finestra JavaFX.
3. **Controller (Controllore):** gestisce l'interazione tra il Model e la View; riceve input dall'utente tramite la View, elabora i dati (eventualmente con l'aiuto del Model) e aggiorna la View. Concettualmente, lavora come un intermediario, tra la view ed il modello ottenuto a partire dai dati contenuti nel Database.

In questo modo – oltre a garantire la separazione della logica di business da quella di visualizzazione – facilita il mantenimento e la testabilità del codice, oltre ad una maggiore flessibilità in caso di aggiornamenti futuri.

## DAO (Data Access Object)

Il DAO è un pattern che regola l'accesso ai dati. È una classe o un insieme di classi che fornisce un'interfaccia astratta per interagire con un database o un'altra fonte di dati.

La gestione della vera e propria connessione è affidata alla classe *DatabaseController*: contiene URL e credenziali per accedere alla base di dati MySQL con JDBC, oltre ad appositi metodi per instaurare e chiudere la connessione.

DatabaseController		
DB_URL	String	
DB_USER	String	
DB_PASSWORD	String	
connection	Connection	
closeConnection(Connection)	void	
getConnection()	Connection	

### Componenti principali del DAO:

1. **Interfaccia DAO:** definisce i metodi per operazioni CRUD (Create, Read, Update, Delete). Contiene le dichiarazioni dei metodi che saranno utilizzati per l'accesso/estrazione dei dati dal DB.
2. **Implementazione DAO:** contiene l'implementazione concreta dell'interfaccia, interagisce con il database grazie alle query configurate in ciascun metodo<sup>2</sup>. Maschera la complessità della gestione del database, come query e ritorno dei risultati della ricerca.

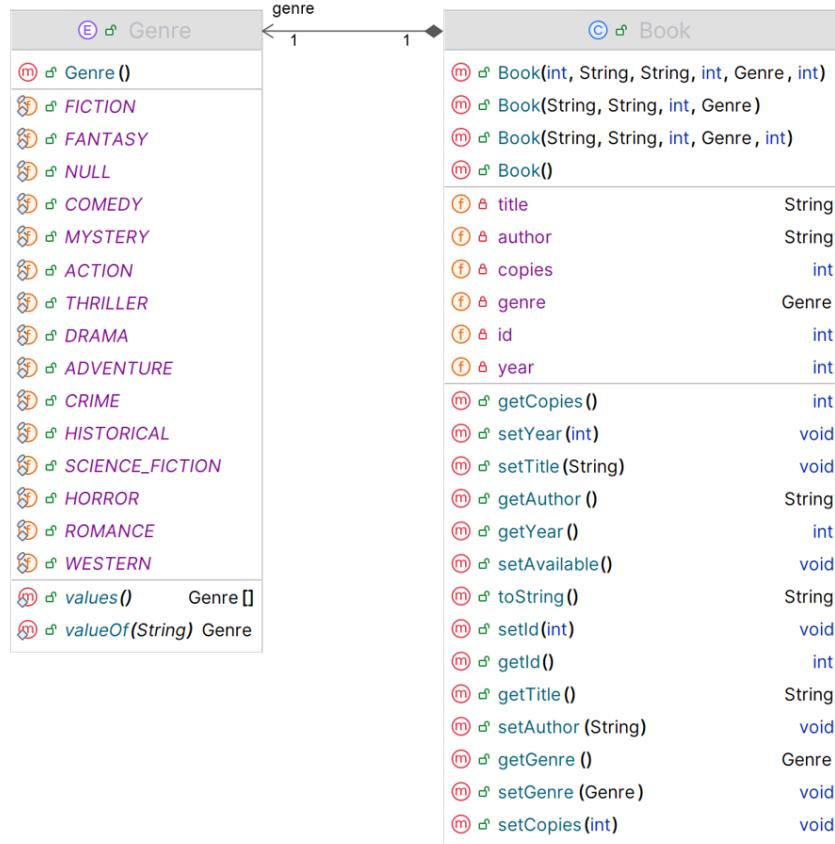
L'adozione ed implementazione del DAO favorisce la riutilizzabilità del codice; dona anche una maggiore modularità al sistema, oltre a garantire un certo grado di facilità nel testing.

Di seguito saranno introdotte le classi dei modelli e relative implementazioni DAO; si noti che per ogni modello è presente un attributo *id* intero: rappresenta l'identificativo utilizzato per individuare un certo dato (libro, prestito o cliente) nella base di dati.

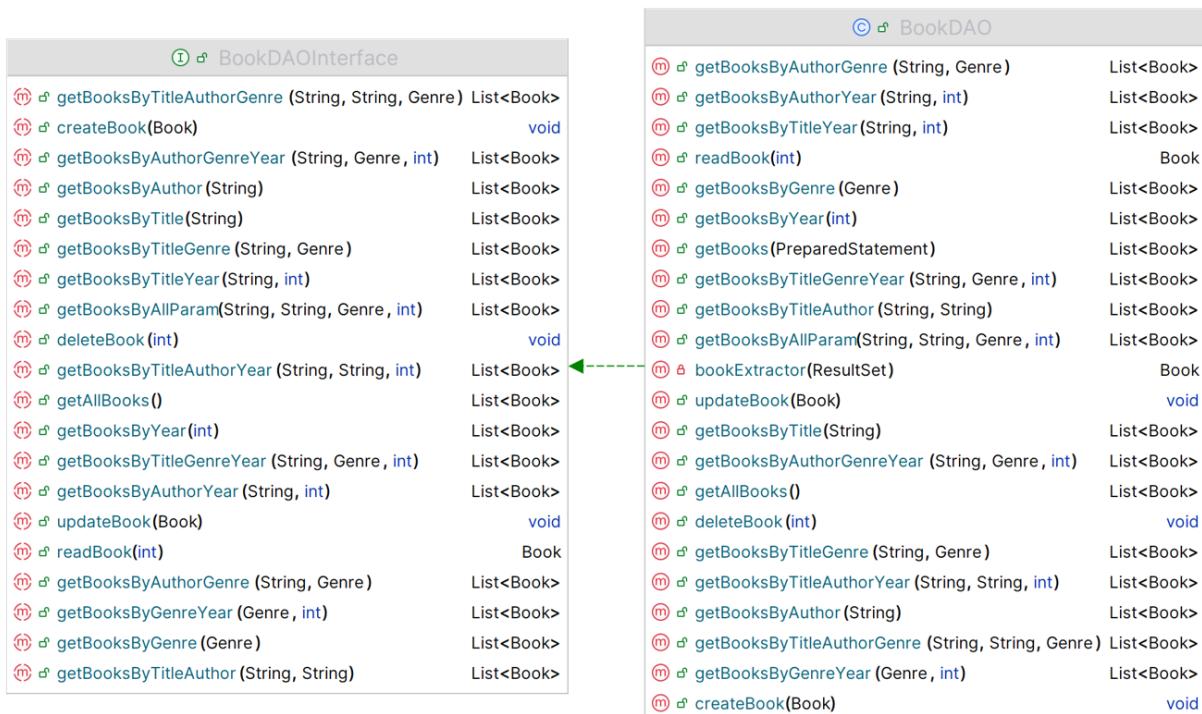
<sup>2</sup> Che saranno poi inviate al DB attraverso la connessione ottenuta dal DatabaseController.

## Gestione dei libri (Books):

- I libri saranno rappresentati dagli oggetti della classe Book.



- Interfaccia e relativa implementazione DAO

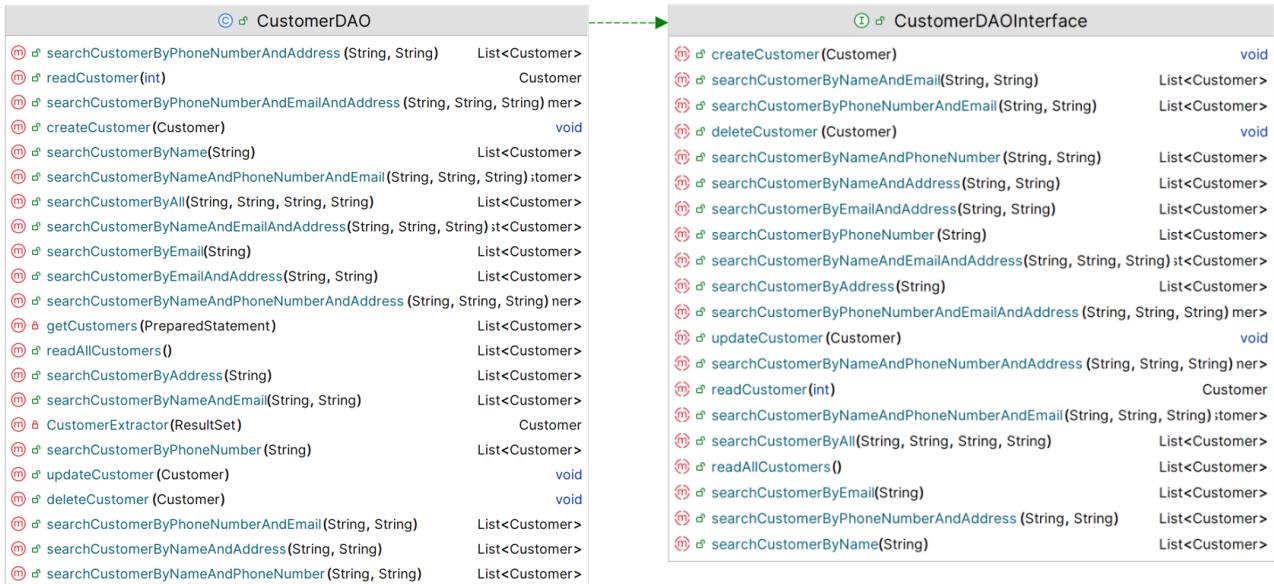


## Gestione dei clienti (Customers)

- I clienti saranno rappresentati dalla classe *Customers*

Customer	
Ⓜ Ⓛ Customer(int, String, String, String, String)	
Ⓜ Ⓛ Customer()	
Ⓜ Ⓛ Customer(String, String, String, String)	
Ⓕ Ⓛ email	String
Ⓕ Ⓛ address	String
Ⓕ Ⓛ id	int
Ⓕ Ⓛ phone	String
Ⓕ Ⓛ name	String
Ⓜ Ⓛ setEmail(String)	void
Ⓜ Ⓛ setPhone(String)	void
Ⓜ Ⓛ setAddress(String)	void
Ⓜ Ⓛ toString()	String
Ⓜ Ⓛ setName(String)	void
Ⓜ Ⓛ getName()	String
Ⓜ Ⓛ getIld()	int
Ⓜ Ⓛ getEmail()	String
Ⓜ Ⓛ getPhone()	String
Ⓜ Ⓛ getAddress()	String
Ⓜ Ⓛ setIld(int)	void

- Di seguito l'interfaccia e l'implementazione del *CustomerDAO*.

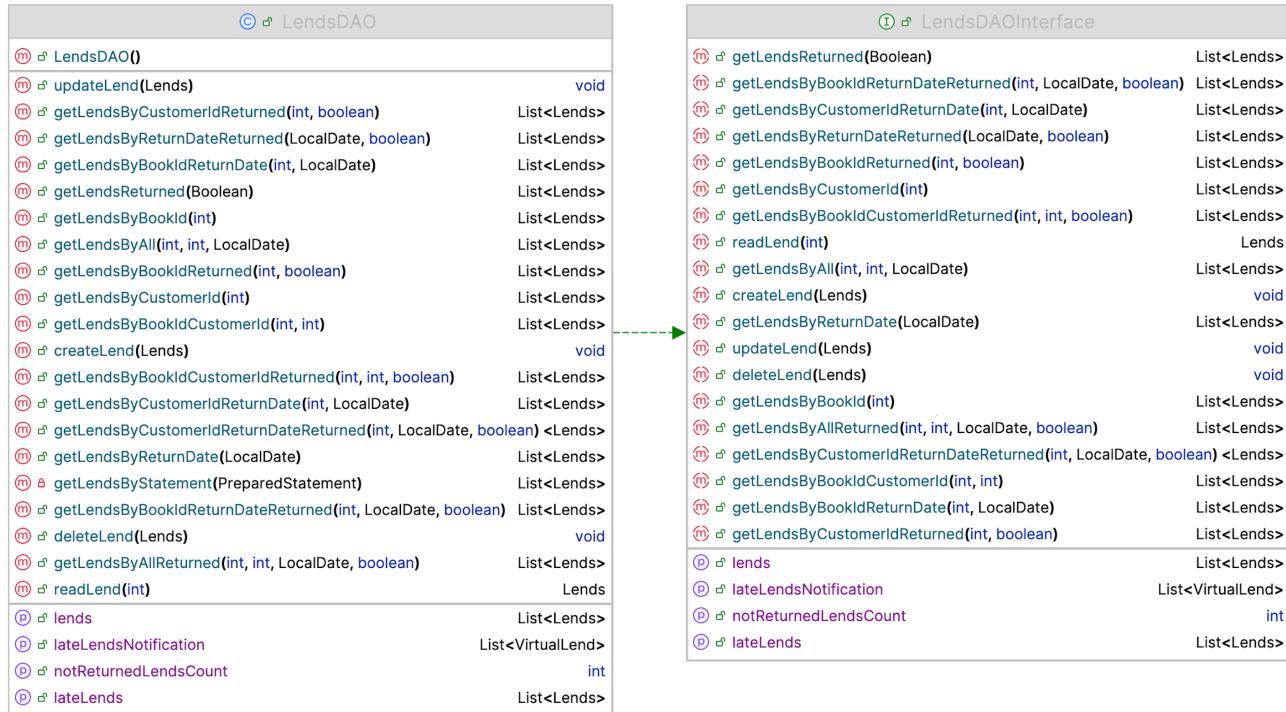


## Gestione dei prestiti (Lends)

- Descrizione delle entità e delle operazioni CRUD sui prestiti di libri

Lends	
ℳ ⚡ Lends(int, int, LocalDate, boolean)	
⌚ ⚡ id	int
⌚ ⚡ returned	boolean
⌚ ⚡ customerId	int
⌚ ⚡ bookId	int
⌚ ⚡ returnDate	LocalDate
ℳ ⚡ isLate()	boolean
ℳ ⚡ isReturned()	boolean
ℳ ⚡ getId()	int
ℳ ⚡ getReturnDate()	LocalDate
ℳ ⚡ getBookId()	int
ℳ ⚡ setId(int)	void
ℳ ⚡ toString()	String
ℳ ⚡ setReturnDate(LocalDate)	void
ℳ ⚡ getCustomerId()	int
ℳ ⚡ setCustomerId(int)	void
ℳ ⚡ setReturned(boolean)	void
ℳ ⚡ setBookId(int)	void

- Implementazione del DAO per l'interfaccia con il database



**VirtualLends:** questa classe di modello virtuale è stata introdotta per essere utilizzata dalla funzione di notifica dell'applicativo, approfondita nei paragrafi successivi.

# Server

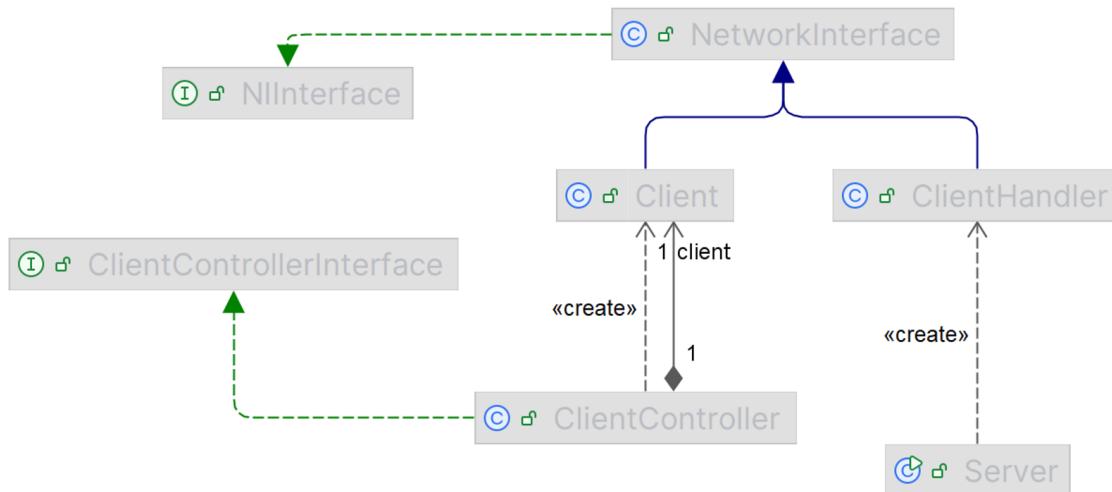
L'applicativo server svolge due operazioni fondamentali, all'avvio apre il *socket* e si prepara ad accettare nuove connessioni dove assegnerà ad ogni client collegato un proprio *ClientHandler*. Questa particolare classe implementa *Runnable* e gestirà il client associato finché la connessione non sarà interrotta; tale gestione prevede invio e ricezione dei messaggi, di seguito l'UML della classe.

© ⚡ ClientHandler		
Ⓜ ⚡ ClientHandler(Socket)		
Ⓣ ⚡ lendDAO	LendsDAO	
Ⓣ ⚡ notificationService	EmailNotificationService	
Ⓣ ⚡ customerDAO	CustomerDAO	
Ⓣ ⚡ bookDAO	BookDAO	
Ⓜ ⚡ handleSearchLendOperation(LendMessage)	void	
Ⓜ ⚡ handleCustomerOperation(CustomerMessage)	void	
Ⓜ ⚡ handleLendOperation(LendMessage)	void	
Ⓜ ⚡ handleBookOperation(BookMessage)	void	
Ⓜ ⚡ handleSearchBookOperation(BookMessage)	void	
Ⓜ ⚡ run()	void	
Ⓜ ⚡ handleSearchCustomersOperation(CustomerMessage)	void	

Al costruttore viene passato il *socket* su cui è attivo il client (che a sua volta viene passato con *super(...)* al costruttore della *NetworkInterface*).

Il metodo *run()* contiene il codice necessario alla gestione dei messaggi in arrivo dal client. A seconda del tipo di richiesta richiama uno dei metodi contrassegnati con *handle...Operation* a seconda del messaggio inviato.

Ciascuno dei metodi che gestiscono le operazioni provvederà ad inviare un'opportuna risposta sfruttando sia la funzione *send(Object)* della *NetworkInterface* che il *createMessage(...)* della classe *MessageFactory*.

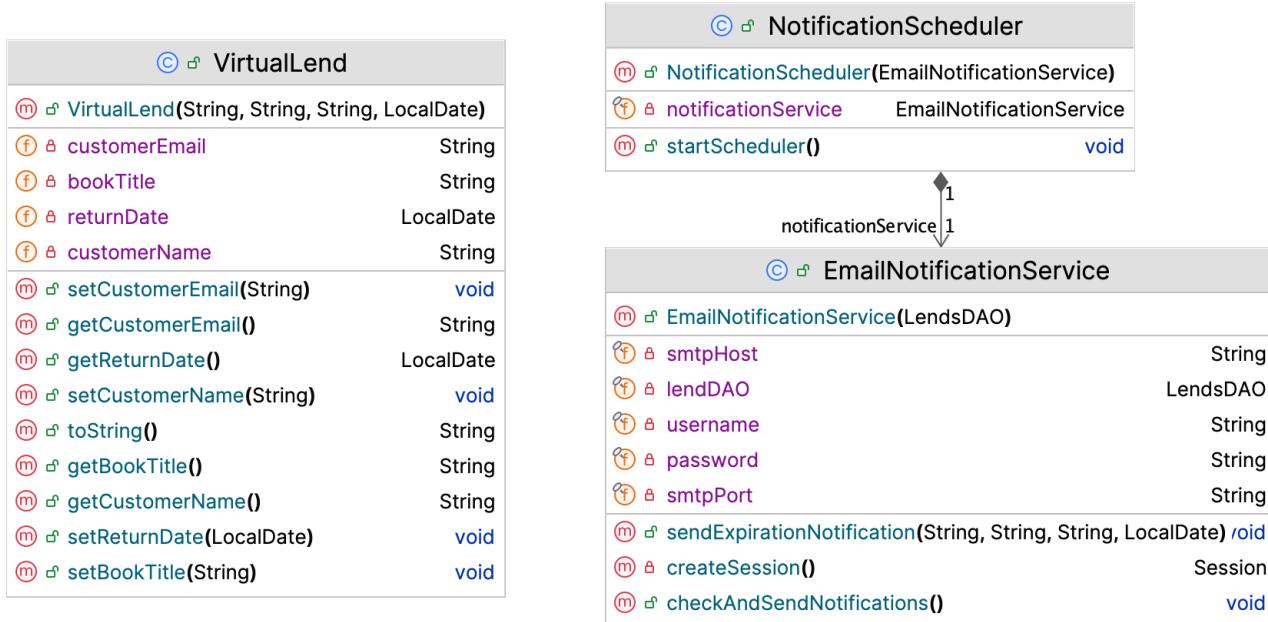


Come anticipato nelle righe precedenti, la classe *NetworkInterface* implementa tutti quei metodi utilizzati per gestire l'invio e la ricezione di messaggi (senza trattarne la tipologia, solo come oggetti serializzabili), così come la disconnessione sicura.

**Disconnessione sicura:** data la natura dell'applicazione è necessario che il server rimanga attivo anche in presenza di disconnessione dal client; il primo passo è stato gestire ciascuna connessione come thread indipendente (si può vedere la corrispondenza 1 a 1 nel diagramma precedente) ed infine è stato aggiunto un apposito metodo.

Una volta sollevata l'eccezione relativa all'errore in ricezione (sul server) allora viene chiamato *safeDisconnection()* che provvede a chiudere la comunicazione con il client ed interrompere il thread. In questo modo il sistema continuerà a funzionare senza disconnettere gli altri utenti.

**E-mail reminder:** servizio dell'applicativo che manda e-mail a tutti i clienti che sono in ritardo con la restituzione dei libri. È implementato lato server come task che viene schedulato per essere eseguito ogni 24 ore a partire da quando il server viene avviato; ad essere inviato per posta elettronica è un messaggio con aggiunti gli attributi degli oggetti *VirtualLends*.



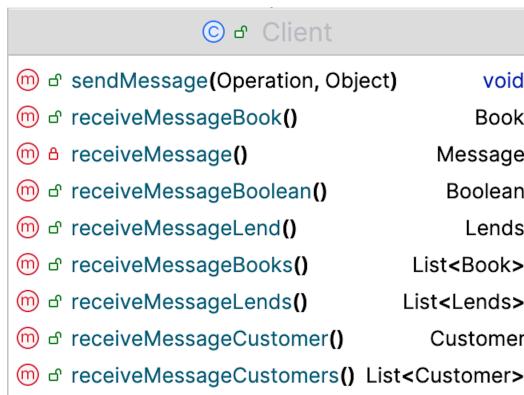
Le mail sono inviate attraverso una connessione TLS1.2 al server SMTP di Google, è richiesto l'utilizzo della porta software numero 587. È stato creato appositamente un account Gmail, con autenticazione sicura tramite *app password*, denominato “*bibliotecaprogettoosw@gmail.com*”.

Per mostrare il funzionamento in tempo reale è stato aggiunto un *button* apposito nella sezione dedicata alla gestione dei prestiti nell'applicazione client. Inoltre, è stata implementata anche la relativa gestione dei messaggi per richiedere l'esecuzione del servizio al di fuori della programmazione oraria prevista.

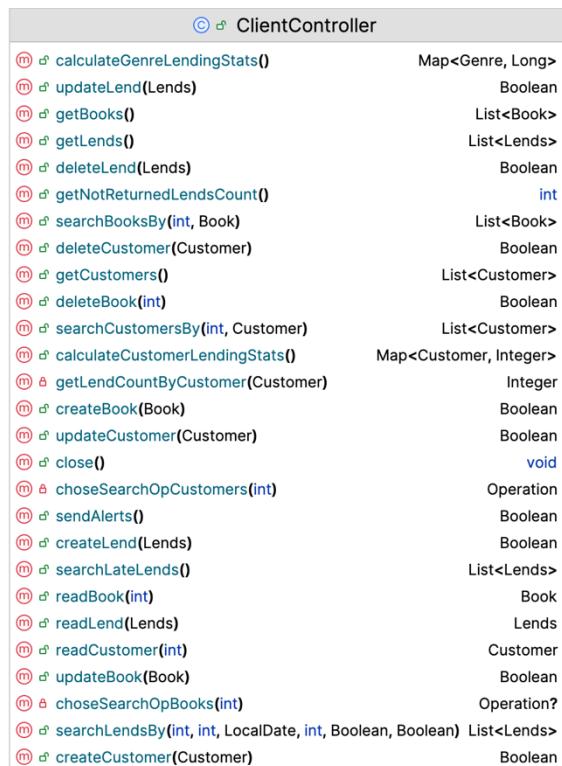
# LibraryApplication

L'applicativo client può essere concettualmente suddiviso in tre livelli di cui uno deputato alla connessione/ricezione/invio, uno stadio di elaborazione intermedia ed infine si trova il livello di presentazione

**Client:** classe che rappresenta il livello di connessione instaura la connessione con il server e gestisce invio e ricezione dei messaggi (sfruttando i metodi di *NetworkInterface*<sup>3</sup>).



**ClientController:** classe intermedia che elabora i dati/messaggi ricevuti dal client, così che possano essere utilizzati dal controller della view. Inoltre, prepara i messaggi da spedire al server, come le richieste di ricerca nel database.



Alcuni metodi sono usati per elaborazioni più “pesanti”. Né è un esempio quanto accade con quelli utilizzati per la sezione relativa alle statistiche sui prestiti del sistema (indicate nell'UML a sinistra come *calculateCustomerLendingStats()*).

Come anticipato nella sezione dedicata alla notifica via posta elettronica del precedente capitolo, è stato aggiunto il metodo *sendAlerts()* per spedire i messaggi prima delle 24 ore previste dal sistema.

<sup>3</sup> In maniera duale rispetto a quanto accade nella classe *ClientHandler*; si noti che entrambi estendono *NetworkInterface*. In questo modo si dona al sistema una maggiore separazione delle responsabilità.

**ViewController:** classe che si interfaccia direttamente con la GUI, dove per ogni evento ha associato un apposito metodo *FXML* per gestirlo e laddove sia richiesto, richiede i dati al *ClientController*.

© View Controller	
Ⓜ️ ⚡ onUpdateBookButtonClick()	void
Ⓜ️ ⚡ onAlertClick()	void
Ⓜ️ ⚡ showCustomersOnTableView(List<Customer>, TableView)	void
Ⓜ️ ⚡ onSearchCustomerButtonClick()	void
Ⓜ️ ⚡ onAddClick()	void
Ⓜ️ ⚡ onAddBookButtonClick()	void
Ⓜ️ ⚡ onSearchBookButtonClick()	void
Ⓜ️ ⚡ onUpdateLendButtonClick()	void
Ⓜ️ ⚡ addCustomer()	void
Ⓜ️ ⚡ onStatsClick()	void
Ⓜ️ ⚡ initChoiceBoxes()	void
Ⓜ️ ⚡ hideAllPanes()	void
Ⓜ️ ⚡ showPane(Pane)	void
Ⓜ️ ⚡ initialize()	void
Ⓜ️ ⚡ onSearchClick()	void
Ⓜ️ ⚡ onLateLendButtonClick()	void
Ⓜ️ ⚡ showBooksOnTableView(List<Book>, TableView)	void
Ⓜ️ ⚡ initNumusBarChart()	void
Ⓜ️ ⚡ onCustomerClick()	void
Ⓜ️ ⚡ clearBookFields()	void
Ⓜ️ ⚡ showErrorDialog(String, String, String)	void
Ⓜ️ ⚡ onReturnLendButtonClick()	void
Ⓜ️ ⚡ initLendsTableView()	void
Ⓜ️ ⚡ onLendBookButtonClick()	void
Ⓜ️ ⚡ initCustomersTableView()	void
Ⓜ️ ⚡ updateLendCount()	void
Ⓜ️ ⚡ onCancelCustomerButtonClick()	void
Ⓜ️ ⚡ updateCustomerLendsBarChart()	void
Ⓜ️ ⚡ onDeleteBookButtonClick()	void
Ⓜ️ ⚡ onLendClick()	void
Ⓜ️ ⚡ onSearchLendButtonClick()	void
Ⓜ️ ⚡ clearSearchCustomers()	void
Ⓜ️ ⚡ updateCustomer(Customer)	void
Ⓜ️ ⚡ showLendsOnTableView(List<Lends>, TableView)	void
Ⓜ️ ⚡ initGenrePieChart()	void
Ⓜ️ ⚡ onHomeClick()	void
Ⓜ️ ⚡ updateGenreChart()	void
Ⓜ️ ⚡ clearSearchLends()	void
Ⓜ️ ⚡ loadBooks()	void
Ⓜ️ ⚡ onCancelButtonClick()	void
Ⓜ️ ⚡ onAddateCustomerButtonClick()	void
Ⓜ️ ⚡ initSearchTableView()	void
Ⓜ️ ⚡ onDeleteLendButtonClick()	void
Ⓜ️ ⚡ initHomeTableView()	void
Ⓜ️ ⚡ clearSearchBooksFields()	void
Ⓜ️ ⚡ onDeleteCustomerButtonClick()	void

All'avvio dell'applicativo client, vengono eseguite le operazioni di controllo della connessione e di inizializzazione della GUI, come la preparazione delle *TableView* o di caricamento di tutti i libri presenti nella base di dati. In caso di mancata connessione, l'applicazione termina mostrando un messaggio di errore.

Per garantire una maggiore facilità e flessibilità di utilizzo nelle operazioni di ricerca, l'utente potrà inserire tutti o solo alcuni dei parametri presenti; sarà poi il sistema a comunicare al *ClientController* che tipo di ricerca (cioè, quali parametri) richiedere al server.

Questa classe contiene anche i metodi per necessari per agire sulla GUI, ad esempio per cambiare i pannelli da visualizzare nello *StackPane*; ogni set di operazioni è contenuto in un differente *Pane*, tra i quali si può navigare facilmente attraverso dei *button* posti alla sinistra dell'interfaccia.

# Interfaccia grafica

- Schermata home

The screenshot shows the 'Library Manager' application window. On the left is a dark blue sidebar with white icons and text for navigation: 'home', 'add book', 'search books', 'lend ops', 'customers', and 'stats'. The main area has a dark header with the title 'Library Manager' and a small icon of three stacked books. Below the header, a 'Welcome!' message is displayed, followed by a note about performing CRUD operations for Books, Lends, and Customers. A table lists five books with columns for ID, Title, Author, Genre, Year, and Copies.

ID	Title	Author	Genre	Year	Copies
2	Assassinio sull'Orient Exp...	Agatha Christie	CRIME	1934	0
3	Assassinio sul Nilo	Agatha Christie	CRIME	1937	1
4	L'Enciclopedia	Zingarelli	HISTORICAL	2005	2
5	Analisi Matematica 2	Ignoti	WESTERN	2023	3

- Schermata per l'aggiunta dei libri

The screenshot shows the 'Add Book Utility' screen within the 'Library Manager' application. It features a sidebar with the same navigation options as the home screen. The main area has a title 'Add Book Utility' with an icon of four books and a green plus sign. A note says 'Write book's data into text fields. Don't forget to fill all the fields!'. There are four text input fields: 'Title:' with placeholder 'book's title here...', 'Author:' with placeholder 'book's author here...', 'Year:' with placeholder 'book's publication year....', and 'Copies:' with placeholder 'book's copies here...'. To the right of the 'Copies:' field is a dropdown menu labeled 'Genre:'. At the bottom are two buttons: a blue 'add book' button and a red 'cancel' button.

- Schermata per la ricerca avanzata ed altre operazioni sui libri

The screenshot shows the 'Search, update, delete and lend books' interface. On the left is a sidebar with icons for home, add book, search books, lend ops, customers, and stats. The main area has a title 'Search, update, delete and lend books' with four input fields: 'Title:' (book's title here...), 'Author:' (book's author here...), 'Genre:' (dropdown menu), and 'Year:' (book's publication year....). Below these are four buttons: 'search' (blue), 'update' (purple), 'lend' (blue), and 'delete' (red). A table below the buttons displays book details:

ID	Title	Author	Genre	Year	Copies
2	Assassinio sull'Orient Express	Agatha Christie	CRIME	1934	0
3	Assassinio sul Nilo	Agatha Christie	CRIME	1937	1
4	L'Enciclopedia	Zingarelli	HISTORICAL	2005	2
5	Analisi Matematica 2	Ignoti	WESTERN	2023	3

- Schermata per ricerca ed operazioni sui prestiti<sup>4</sup>

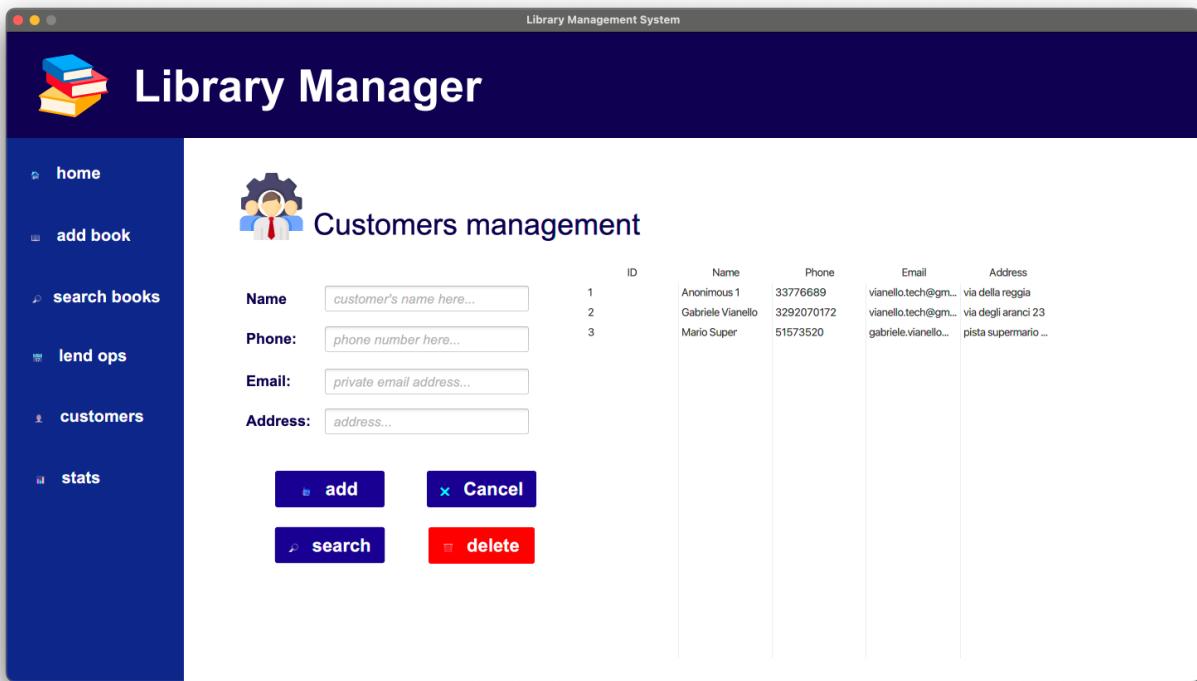
The screenshot shows the 'Search, update, delete lends and return books' interface. The sidebar is identical to the previous screen. The main area has a title 'Search, update, delete lends and return books' with four input fields: 'Title:' (book's title here...), 'Phone:' (customer's phone here...), 'Return date:' (dropdown menu), and 'Returned:' (dropdown menu). Below these are five buttons: 'late' (blue), 'search' (purple), 'update' (blue), 'return' (blue), 'late alert' (yellow), and 'delete' (red). A table below the buttons displays lending information:

ID	Book ID	Customer ID	Return Date	Returned
2	2	2	2024-12-02	false
3	5	3	2024-12-04	false
4	5	2	2024-12-02	true

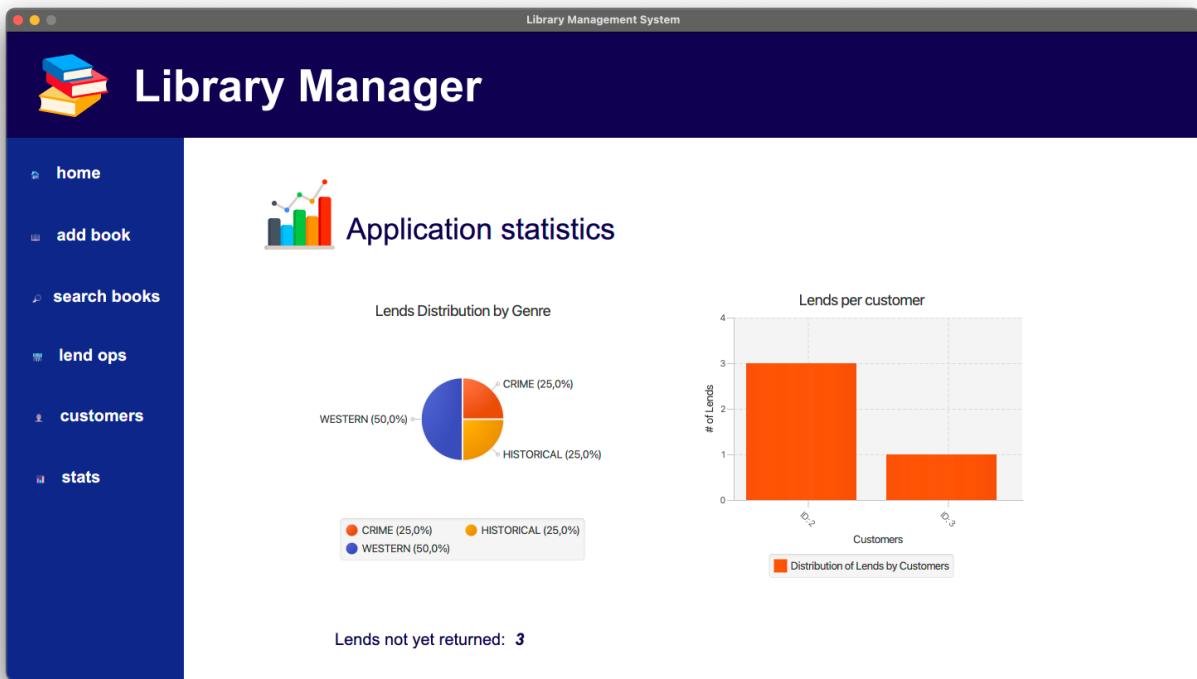
---

<sup>4</sup> Il button “late alert” è quello utilizzato per inviare le e-mail di notifica in tempo reale a tutti quei clienti che non hanno restituito in tempo il libro.

- Schermata per la gestione dei clienti



- Schermata per le statistiche del sistema<sup>5</sup>



La grandezza dei font può variare a seconda del sistema operativo utilizzato. A questo scopo sono state adottate delle misure delle schermate volte a ridurre il più possibile questo fenomeno.

---

<sup>5</sup> In particolare, la distribuzione dei prestiti per genere, il numero di prestiti per cliente ed infine il numero di prestiti in attivo (non restituiti).

# Tecnologie utilizzate

Elenco delle principali tecnologie e librerie utilizzate nell'applicativo:

- Java
- JavaFX con kit SceneBuilder per la GUI
- Java Mail per la gestione della messaggistica di posta elettronica
- Google Gmail come mail provider
- Istanza di Database relazionale MySQL
- JUnit5 per i test
- Le icone sono state prese da *FlatIcon*

**Macchina e IDE:** il software è stato sviluppato interamente su macchina Windows 10 e macOS 15, entrambe con istanza Server di MySql8.0 e utilizzando IntelliJ IDEA come ambiente di sviluppo. Le query SQL sono state scritte su MySQL WorkBench.

# Test & Coverage

Data la tipologia di applicativo realizzata, i test sono stati eseguiti sui modelli, sulle classi relative all'implementazione DAO e sulle classi dei messaggi serializzabili. Nel caso dei test sul DAO è stato verificato che nelle query di ricerca venissero effettivamente restituiti gli oggetti con le caratteristiche/parametri richiesti.

Package	Class, %	Method, %	Branch, %	Line, %
all classes	62,9% (22/35)	45,8% (141/308)	3,9% (27/701)	37% (631/1704)

## Coverage Breakdown

Package ▲	Class, %	Method, %	Branch, %	Line, %
prj.library.database	100% (1/1)	50% (2/4)		66,7% (6/9)
prj.library.database.DAO	100% (3/3)	84,5% (60/71)	72,7% (16/22)	65% (415/638)
prj.library.models	100% (4/4)	93,5% (43/46)	0% (0/2)	95,6% (86/90)
prj.library.networking.messages	100% (11/11)	92% (23/25)	12,7% (9/71)	95,3% (102/107)
prj.library.notification	16,7% (1/6)	40,9% (9/22)	0% (0/2)	22,8% (13/57)
prj.library.utils	100% (2/2)	20% (4/20)	50% (2/4)	31% (9/29)

# Configurazione e Installazione

Per funzionare correttamente si dovrà prima avviare l'applicativo *Server* e poi i client (*LibraryApplication*). Tutte le informazioni che saranno utilizzate per la configurazione e l'avvio del sistema sono contenute nel file `config.properties`: potranno essere modificate direttamente sul file senza andare ad agire direttamente sul codice.

Su tale file si dovrà specificare anche quali credenziali utilizzare sia per l'account di posta elettronica che per accedere alla base di dati.

Se la funzionalità di notifica via mail non dovesse funzionare correttamente provare a cambiare rete o le impostazioni del firewall; generalmente non tutte le reti permettono l'uso della porta 587. In caso di ulteriori problemi utilizzare la funzione di debug offerta dalla libreria “`java.net.mail`”.

Per mostrare il funzionamento delle notifiche mail e sistemi di ricerca sono stati tolti i controlli delle date inserite.

# Considerazioni finali

Il sistema realizzato rispetta i requisiti di funzionamento ed è stato testato simultaneamente su differenti dispositivi con diversi sistemi operativi, non riscontrando problematiche od altre difficoltà durante l'esecuzione.

Sono state simulate situazioni di errore di connessione da parte di un client per assicurarsi del corretto funzionamento della funzione di disconnessione sicura, dove l'applicativo server continua l'esecuzione degli altri *clientHandler* (attivi) senza errori.

Anche nell'eventualità in cui non si riuscissero ad inviare le notifiche via mail delle consegne in ritardo, il programma continuerebbe ad eseguire senza intaccare le attività di base dell'applicazione.