

**Università Politecnica delle Marche**  
**Facoltà di Ingegneria**

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

---



**Relazione progetto per il corso di Programmazione mobile**

**Implementazione di un'app mobile per la gestione di una  
biblioteca a livello nazionale**

Professore

Prof. Storti Emanuele

Componenti del gruppo

Bellante Luca  
Coccia Giansimone  
Di Sabatino Walter

---

**Anno Accademico 2022-2023**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Progettazione</b>	<b>3</b>
2.1	Glossario dei termini . . . . .	3
2.2	Requisiti . . . . .	4
2.2.1	Requisiti funzionali . . . . .	4
2.2.2	Requisiti non funzionali . . . . .	7
2.3	Casi d'uso . . . . .	8
2.4	Diagramma dei componenti . . . . .	12
2.5	Progettazione Database Flutter . . . . .	12
2.5.1	Schema E-R . . . . .	12
2.5.2	Traduzione verso il mondo relazionale . . . . .	13
2.6	Progettazione Databse Firebase . . . . .	13
2.7	Mockup . . . . .	14
2.8	API utilizzate . . . . .	15
2.8.1	Google-books API . . . . .	15
2.8.2	Google-maps API . . . . .	15
2.8.3	opacmobilegw API . . . . .	16
<b>3</b>	<b>Programmazione in Android</b>	<b>22</b>
3.1	MVVM . . . . .	22
3.2	Struttura delle cartelle del progetto . . . . .	23
3.3	Approfondimenti sul codice . . . . .	25
3.3.1	MainActivity . . . . .	25
3.3.2	Activity . . . . .	26
3.3.3	Adapter . . . . .	26
3.3.4	Cache . . . . .	27
3.3.5	Components . . . . .	28
3.3.6	Firebase . . . . .	28
3.3.7	Fragments . . . . .	28
3.3.8	Model . . . . .	32
3.3.9	ViewModel . . . . .	34
3.3.10	Utils . . . . .	35
3.4	Test . . . . .	35
3.4.1	Unit test . . . . .	35
3.4.2	Instrumented test . . . . .	37
<b>4</b>	<b>Programmazione in Flutter</b>	<b>41</b>
4.1	Approfondimento del codice . . . . .	42
<b>5</b>	<b>Errori e possibili bug</b>	<b>52</b>
<b>6</b>	<b>Ringraziamenti</b>	<b>52</b>

## Elenco delle figure

1	Requisiti funzionali . . . . .	5
2	Requisiti non funzionali . . . . .	7
3	Attori . . . . .	8
4	Casi d'uso relativi alla gestione di un utente . . . . .	9
5	Casi d'uso relativi alla gestione delle attività . . . . .	10
6	Diagramma dei componenti . . . . .	12
7	Diagramma E-R database utilizzato per l'implementazione in Flutter . . . . .	13
8	Struttura di <i>Firebase</i> . . . . .	14
13	Inizio conversazione con <i>Inera</i> . . . . .	16
14	Risposta e documentazione di <i>Inera</i> . . . . .	17
15	Esempio di chiamata <i>Http</i> e risposta da parte dell'API <i>opacmobilegw</i> . . . . .	18
16	Comunicazione del nuovo problema ad <i>Inera</i> . . . . .	21
17	Risposta di <i>Inera</i> al nuovo problema . . . . .	21
18	Andata a buon fine dello Unit test MiniBookTest . . . . .	36
19	Andata a buon fine dello Unite test UserTest . . . . .	37
20	Andata a buon fine dell'Instrumented test LoginTest . . . . .	38
21	Andata a buon fine dell'Instrumented test LogoutTest . . . . .	39
22	Andata a buon fine dell'Instrumented test RegisterTest . . . . .	40
23	Andata a buon fine dell'Instrumented test UpdateInfoUserTest . . . . .	41
24	Classe AuthManager . . . . .	42
25	Classe DBBooks . . . . .	43
26	Classe Users . . . . .	44
27	Classe AppRoutes . . . . .	45
28	Metodo loginUser . . . . .	46
29	Metodo deleteProfile . . . . .	47
30	Regex utilizzate per la convalida dei campi email e password . . . . .	48
31	Implementazione per il funzionamento dell'API Geocoding di Google Maps . . . . .	49
32	Implementazione per il funzionamento dell'API di Google Books . . . . .	50
33	Implementazione per il funzionamento dell'API di Opac per l'ottenimento delle informazioni su un libro . . . . .	51

# 1 Introduzione

La presente relazione documenta lo sviluppo di un'app innovativa per la gestione di prestiti di libri a livello nazionale. L'obiettivo principale del progetto è stato fornire agli utenti un'esperienza moderna e coinvolgente, consentendo loro di prenotare libri, interagire con la comunità di lettori e godere appieno del vasto patrimonio letterario offerto dalle varie biblioteche.

Inizialmente, l'app è stata sviluppata in Kotlin per Android, sfruttando le potenzialità del sistema operativo per offrire un'interfaccia utente intuitiva e reattiva. Sono stati implementati una serie di moduli chiave, tra cui la funzionalità di ricerca, che consente agli utenti di cercare libri per titolo o autore. Inoltre, è stato integrato un sistema di prenotazione, che permette agli utenti di richiedere libri disponibili per il prestito, con la possibilità di selezionare la sede preferita per il ritiro. Ovviamente, a meno che il libro non risulti già prenotato presso quella biblioteca dall'utente, il prestito sarà sempre disponibile, poiché sarebbe stato complicato da parte nostra contattare tutte le biblioteche presenti sul territorio ed organizzare un'applicazione del genere.

Successivamente, l'app è stata portata su Flutter, consentendo di ampliare la copertura del progetto anche per gli utenti di dispositivi iOS. Questo ha permesso di raggiungere un pubblico più ampio e di offrire una piattaforma uniforme per i lettori, indipendentemente dal sistema operativo utilizzato. Durante questa fase di sviluppo, è stata prestata particolare attenzione all'ottimizzazione delle prestazioni e alla coerenza dell'interfaccia utente su entrambe le piattaforme.

Inoltre, l'app ha incorporato funzionalità sociali per stimolare l'interazione tra gli utenti. È stato possibile inserire commenti e recensioni per i libri, consentendo agli utenti di condividere le proprie opinioni e consigliare letture ad altri membri della comunità. È stata anche implementata la funzionalità "Mi piace", che permette agli utenti di esprimere apprezzamento per i libri preferiti e di scoprire quali libri sono popolari tra gli altri lettori.

Questa relazione illustra il processo di sviluppo dell'app, le sfide affrontate durante l'implementazione e le soluzioni adottate. Vengono inoltre presentate le funzionalità principali dell'app, insieme alle scelte di design e agli aspetti tecnici rilevanti.

# 2 Progettazione

## 2.1 Glossario dei termini

Di seguito, si riportano i termini maggiormente utilizzati correlati di sinonimi e descrizione, per facilitare la lettura di questo progetto:

Glossario dei termini			
Termini	Sinonimi	Omonimi	Descrizione
Isil	//	//	E' l'identificativo standard internazionale conforme alla norma ISO 15511 per le biblioteche e le organizzazioni collegate come archivi e musei ed è ovviamente utilizzato nella base dati Anagrafe.
Isbn	//	//	E' una sequenza numerica di 13 cifre, utilizzata in tutto il mondo in maniera univoca per la classificazione dei libri.
Bid	//	//	Il Bid è il codice di identificazione bibliografica ed è assegnato, in maniera univoca, a tutte le notizie bibliografiche.

## 2.2 Requisiti

Di seguito vengono riportati i requisiti *funzionali* e *non funzionali* dell'applicazione.

### 2.2.1 Requisiti funzionali

I requisiti funzionali si presentano come elenchi di funzionalità che il sistema deve fornire, per maggiori dettagli è possibile osservare la Figura 1.

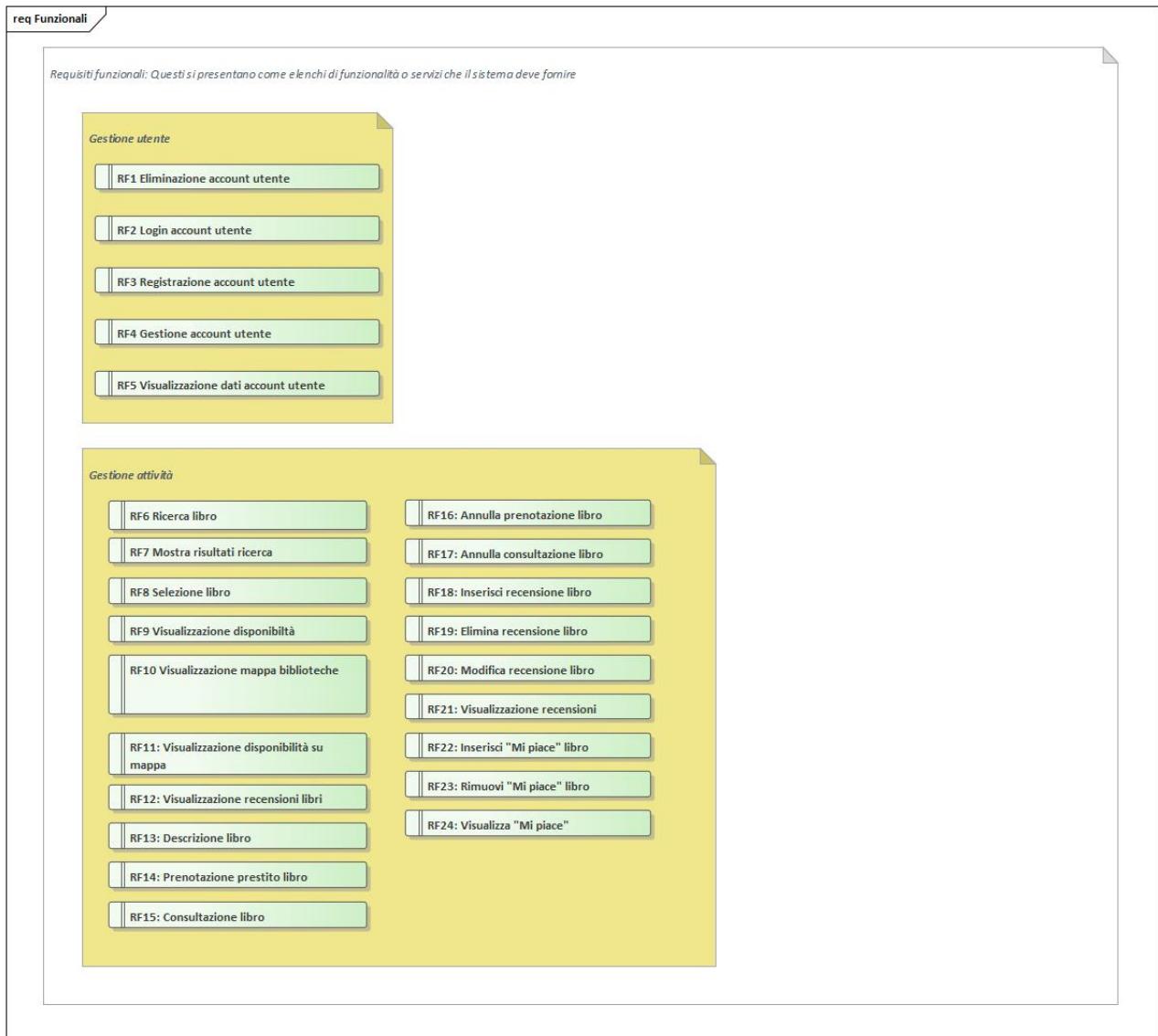


Figura 1: Requisiti funzionali

- **Gestione utente**

- *RF1: Eliminazione account utente* - Il sistema si occuperà dell'eliminazione dell'account relativo all'utente.
- *RF2: Login account utente* - Il sistema si occuperà di consentire l'accesso all'utente all'interno della piattaforma.
- *RF3: Registrazione account utente* - Il sistema consentirà la registrazione dell'utente.

- *RF4: Gestione account utente* - Il sistema consentirà la gestione dell'account relativo all'utente come visualizzazione e modifica dati.
- *RF5: Visualizzazione dati account utente* - Il sistema permetterà la visualizzazione dei propri dati all'utente.

- **Gestione attività**

- *RF6: Ricerca libro* - Il sistema permetterà la ricerca del libro interessato.
- *RF7: Mostra risultati ricerca* - Il sistema visualizzerà i risultati della ricerca effettuata.
- *RF8: Selezione libro* - Il sistema consentirà la selezione del libro di interesse.
- *RF9: Visualizzazione disponibilità* - Il sistema mostrerà la disponibilità del libro cercato.
- *RF10: Visualizzazione mappa biblioteca* - Il sistema visualizzerà su mappa la biblioteca di interesse.
- *RF11: Visualizzazione disponibilità su mappa* - Il sistema permetterà la visualizzazione delle disponibilità sulla mappa.
- *RF12: Visualizzazione recensione libri* - Il sistema consentirà la visualizzazione delle recensioni dei libri.
- *RF13: Descrizione libro* - Il sistema permetterà di visualizzare una breve descrizione del libro in questione.
- *RF14: Prenotazione prestito libro* - Il sistema consentirà la prenotazione per effettuare un prestito del libro voluto.
- *RF15: Consultazione libro* - Il sistema permetterà di consultare le informazioni dettagliate di un libro.
- *RF16: Annulla prenotazione libro* - Il sistema permetterà di annullare una prenotazione effettuata per un libro.
- *RF17: Annulla consultazione libro* - Il sistema consentirà di annullare la consultazione del libro.
- *RF18: Inserisci recensione libro* - Il sistema permetterà l'inserimento della recensione per il libro.
- *RF19: Elimina recensione libro* - Il sistema consentirà l'eliminazione di una recensione inserita.
- *RF20: Modifica recensione libro* - Il sistema permetterà la modifica di una recensione inserita.
- *RF21: Visualizzazione recensioni* - Il sistema permetterà la visualizzazione delle recensioni inserite dagli altri utenti.
- *RF22: Inserisci "Mi piace" libro* - Il sistema permetterà l'inserimento del "Mi piace" per un libro.
- *RF23: Rimuovi "Mi piace" libro* - Il sistema permetterà la rimozione del "Mi piace" precedentemente inserito.
- *RF24: Visualizza "Mi piace"* - Il sistema permetterà la visualizzazione dei "Mi piace" inseriti.

## 2.2.2 Requisiti non funzionali

I requisiti non funzionali rappresentano i vincoli e le proprietà/caratteristiche che il sistema deve rispettare, come vincoli di natura temporale, vincoli sul processo di sviluppo e sugli standard da adottare. In Figura 2 è possibile osservare i requisiti non funzionali.

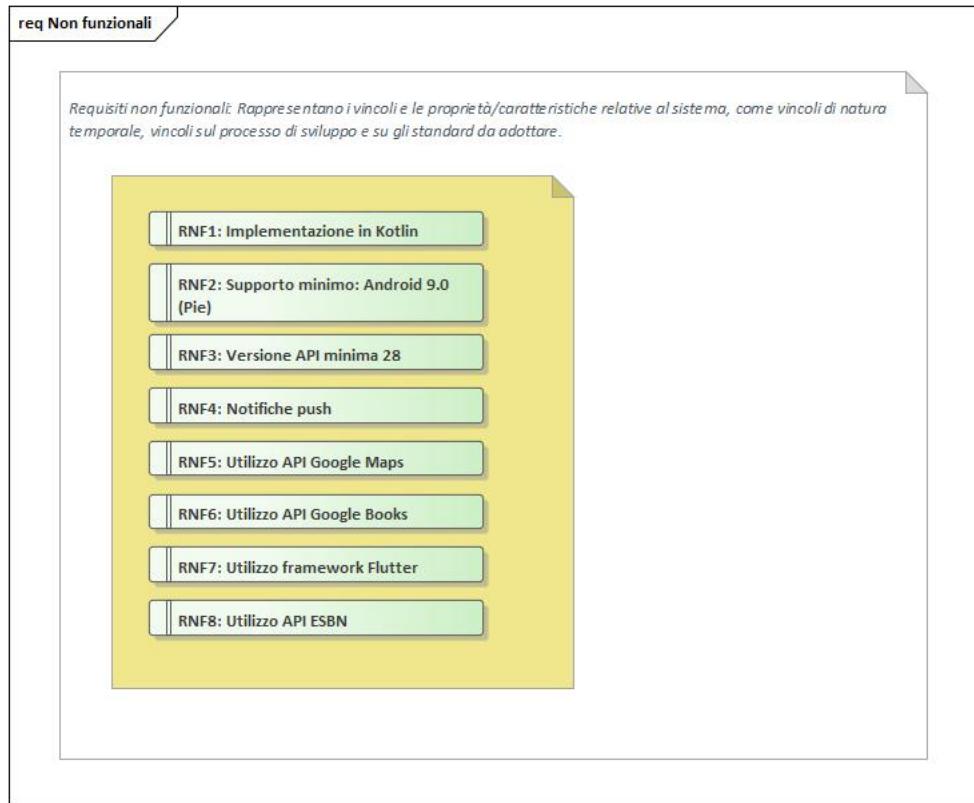


Figura 2: Requisiti non funzionali

- *RNF1: Implementazione in Kotlin* - Il progetto sarà implementato con linguaggio di sviluppo Kotlin.
- *RNF2: Supporto minimo: Android 9.0 (Pie)* - Il progetto avrà una versione minima Android come precedentemente riportato.
- *RNF3: Versione API minima 28* - Il progetto avrà una versione di API minima come precedentemente riportato.
- *RNF4: Notifiche push* - Il progetto utilizzerà un sistema di notifiche push per allertare l'utente.
- *RNF5: Utilizzo API Google Maps* - Il progetto utilizzerà le API Google Maps per informazioni relative alle biblioteche.
- *RNF6: Utilizzo API Google Books* - Il sistema utilizzerà le API di Google Books per informazioni relative ai libri.

- *RNF7: Utilizzo framework Flutter* - Il progetto sfrutterà il framework Flutter per l'implementazione sia in iOS che in Android.
- *RNF8: Utilizzo API ESBN* - Il sistema sfrutterà le API messe a disposizione da ESBN.

### 2.3 Casi d'uso

In Figura 3 sono riportati gli attori che svolgono le azioni all'interno della nostra applicazione e danno luogo ai seguenti casi d'uso (Figura 4 e Figura 5), descritti dettagliatamente uno per uno successivamente.

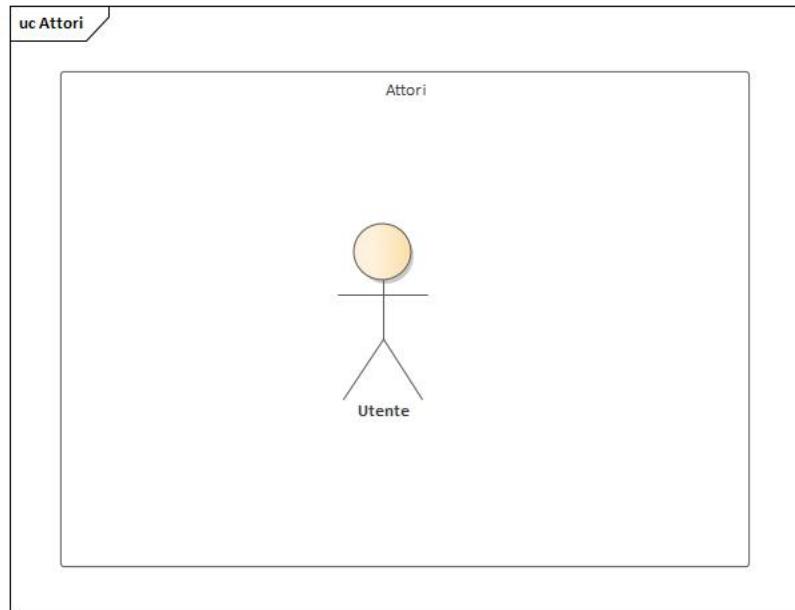


Figura 3: Attori

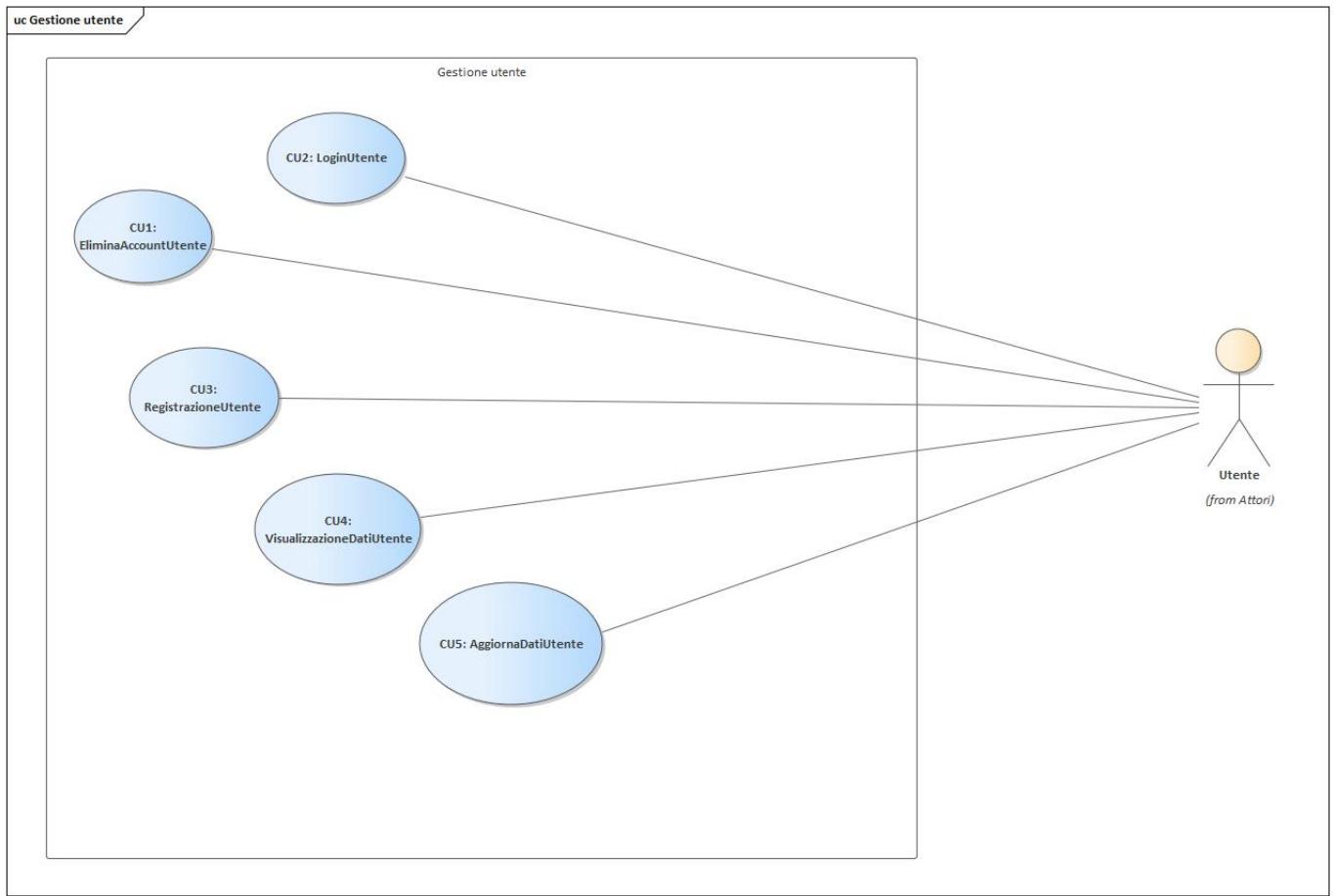


Figura 4: Casi d'uso relativi alla gestione di un utente

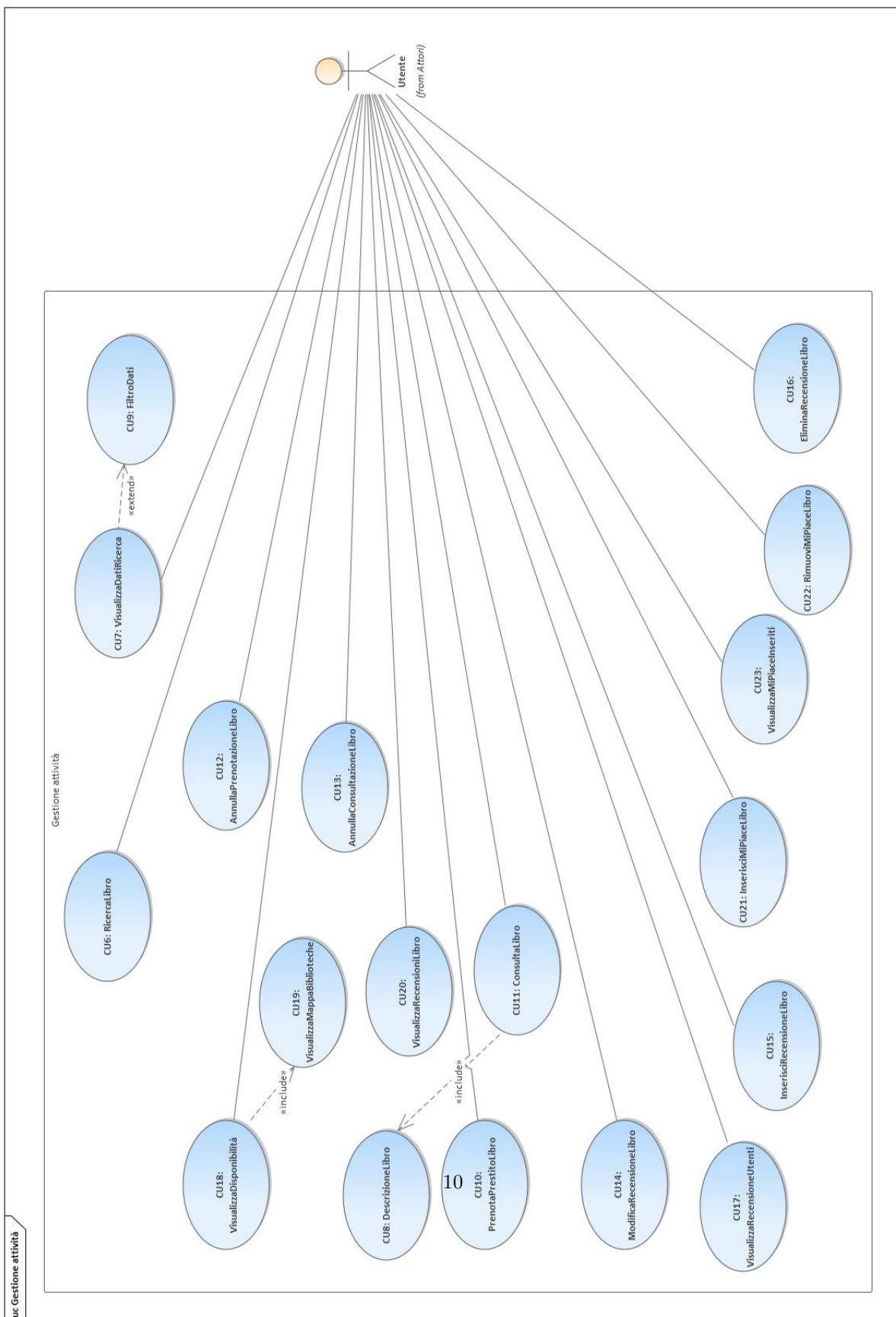


Figura 5: Casi d'uso relativi alla gestione delle attività

- *Gestione utente*
  - *CU1: EliminaAccountUtente* - Il caso d'uso consente l'eliminazione dell'account relativo all'utente.
  - *CU2: LoginUtente* - Il caso d'uso consente all'utente di effettuare il login nell'app.
  - *CU3: RegistrazioneUtente* - Il caso d'uso consente la registrazione di un nuovo utente.
  - *CU4: VisualizzazioneDatiUtente* - Il caso d'uso consente la visualizzazione dei dati relativi ad un utente.
  - *CU5: AggiornaDatiUtente* - Il caso d'uso consente l'aggiornamento dei dati relativi ad un utente.
- *Gestione attività*
  - *CU6: RicercaLibro* - Il caso d'uso consente la ricerca di un libro.
  - *CU7: VisualizzaDatiRicerca* - Il caso d'uso consente la visualizzazione dei dati ricercati per un libro. **Include CU10: FiltroDati.**
  - *CU8: DescrizioneLibro* - Il caso d'uso consente la visualizzazione della descrizione di un libro.
  - *CU9: FiltroDati* - Il caso d'uso consente il filtraggio dei dati per la ricerca di un libro.
  - *CU10: PrenotaPrestitoLibro* - Il caso d'uso consente la prenotazione per un prestito di un libro.
  - *CU11: ConsultaLibro* - Il caso d'uso consente la consultazione dei dettagli relativi ad un libro. **Extend CU10: DescrizioneLibro.**
  - *CU12: AnnullaPrenotazioneLibro* - Il caso d'uso consente l'annullamento di una prenotazione.
  - *CU13: AnnullaConsultazioneLibro* - Il caso d'uso consente l'annullamento della consultazione.
  - *CU14: ModificaRecensioneLibro* - Il caso d'uso consente la modifica della recensione relativa ad un libro.
  - *CU15: InserisciRecensioneLibro* - Il caso d'uso consente l'inserimento di una recensione relativa ad un libro.
  - *CU16: EliminaRecensioneLibro* - Il caso d'uso consente l'eliminazione di una recensione.
  - *CU17: VisualizzaRecensioneUtente* - Il caso d'uso consente la visualizzazione delle recensioni dell'utente.
  - *CU18: VisualizzaDisponibilità* - Il caso d'uso consente di conoscere la disponibilità di un libro. **Include CU22: VisualizzaMappaBiblioteca.**
  - *CU19: VisualizzaMappaBiblioteca* - Il caso d'uso consente la visualizzazione su mappa delle biblioteche.
  - *CU20: VisualizzaRecensioniLibro* - Il caso d'uso consente la visualizzazione delle recensioni relative ad un libro.
  - *CU21: InserisciMiPiaceLibro* - Il caso d'uso consente l'inserimento da parte dell'utente, del gradimento verso un determinato libro inserendo il "like".
  - *CU22: RimuoviMiPiaceLibro* - Il caso d'uso consente la rimozione del "like" precedentemente inserito.
  - *CU23: VisualizzaMiPiaceInseriti* - Il caso d'uso consente la visualizzazione dei "like" inseriti.

## 2.4 Diagramma dei componenti

In Figura 6 è riportato il diagramma dei componenti. Possiamo osservare come, tra le istanze principali vi siano l'utente, il libro, l'account dell'utente, la recensione ed il "Mi piace". Ciascuna di queste entità viene attivata dall'utente che può, ad esempio, aggiungere o rimuovere una recensione oppure un "Mi piace", aggiungere, rimuovere o consultare un libro, ma anche apportare modifiche al proprio account, come ad esempio l'eliminazione del suo account, l'aggiornamento delle sue credenziali o la visualizzazione di queste.

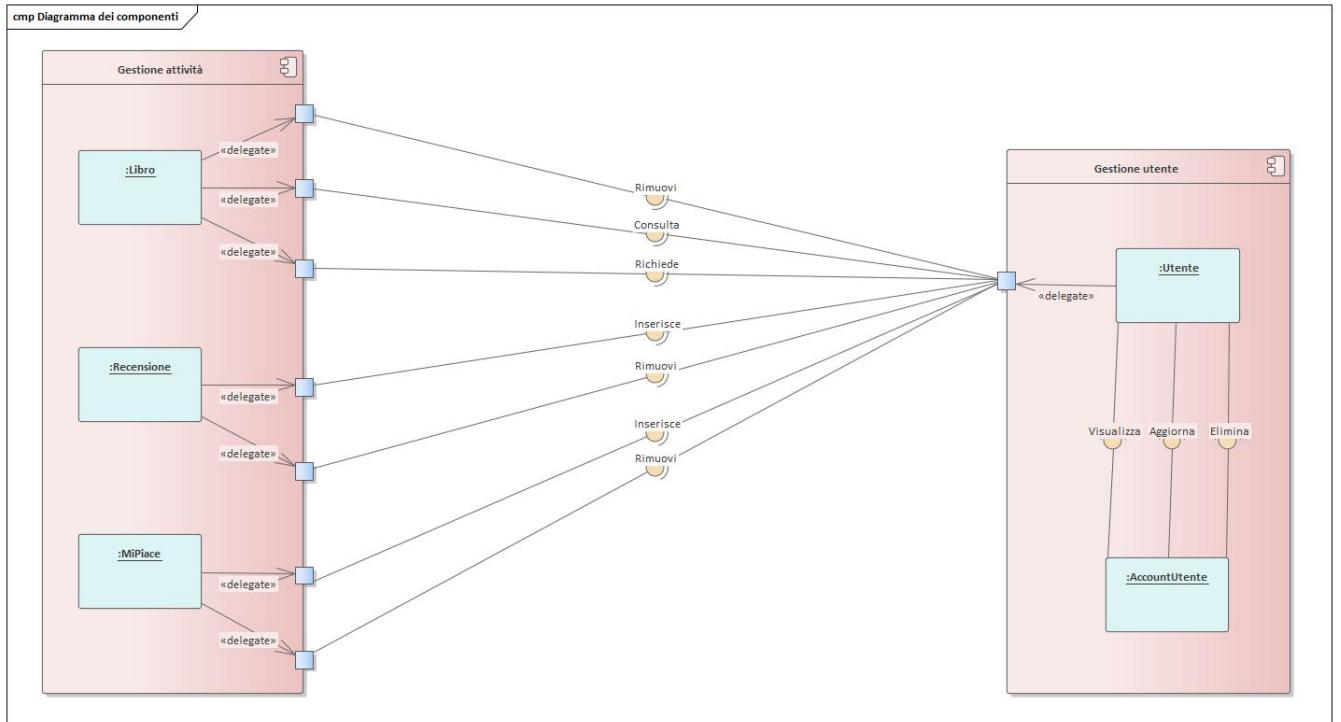


Figura 6: Diagramma dei componenti

## 2.5 Progettazione Database Flutter

La base di dati è locale, ed è frutto dello studio delle esigenze provenienti dall'interazione tra l'applicazione e l'utente. Si sottolinea, ancora una volta, che la base di dati è locale al dispositivo, quindi a differenza della base di dati in Android, presente su un server e quindi accessibile da molti dispositivi contemporaneamente, quella in Flutter è accessibile ad un solo utente per volta.

### 2.5.1 Schema E-R

In Figura 7, viene riportato lo schema E-R. Esso rappresenta come le *entità(entity)* si *relazionano* tra di loro all'interno del sistema.

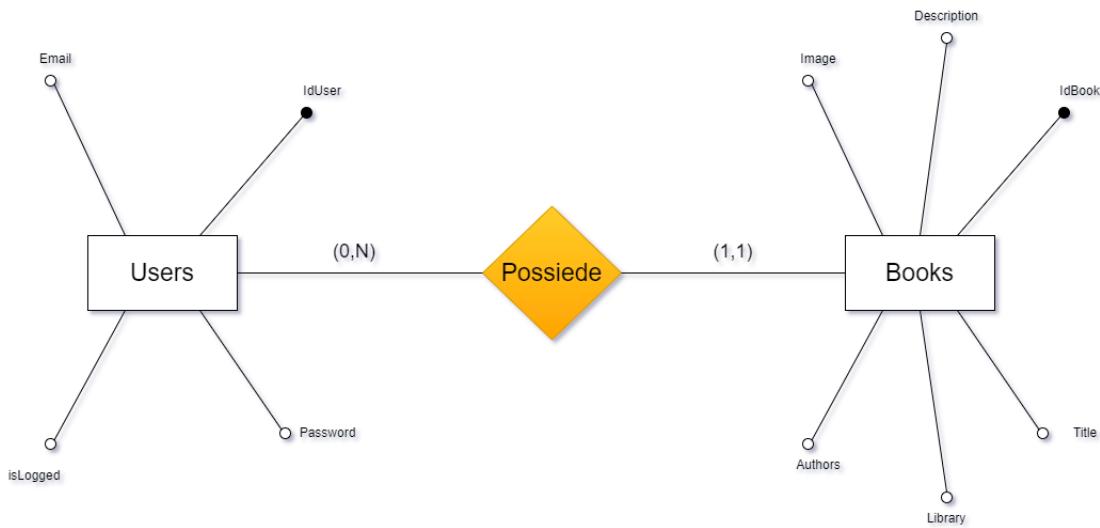


Figura 7: Diagramma E-R database utilizzato per l'implementazione in Flutter

In questa semplice base di dati, l'entità **”Users”** e l'entità **”Books”** si relazionano attraverso la *relationship* **”Possiede”**. Particolare attenzione viene data alle *cardinalità* delle entità. Lo schema deve essere interpretato come segue:

- Lo *Users* può possedere **0 o più Books**.
- Il *Books* può essere posseduto da **uno ed un solo utente**.

### 2.5.2 Traduzione verso il mondo relazionale

Traduzione verso il mondo relazionale	
Entità / Relazione	Traduzione
Users	Users(idUser, Email, Password, isLogged)
Books	Books(idBook, Title, Library, Authors, Image, Description)

La chiave primaria di *Entità* è **idUser**, mentre quella di *Books* è **idBook**.

## 2.6 Progettazione Databse Firebase

Per l'app in *Kotlin*, abbiamo utilizzato sia per l'autenticazione che per la gestione degli utenti il database non relazionale di casa Google, chiamato **Firebase**. Non essendo relazionale, non segue le stesse regole di progettazione e sviluppo di una base di dati relazionale. La logica di funzionamento è efficiente quanto essenziale. È possibile racchiudere i dati in **documenti** che sono organizzati in **raccolte**. Ciascun documento contiene delle coppie *chiave-valore*. Ovviamente possiamo ampliare e complicare a piacimento la struttura del documento, poiché *Firebase* ci consente di aggiungere, all'interno del documento considerato, riferimenti ad altre *sotto-raccolte* e/o *oggetti nidificati*. La struttura di un documento *Firebase* è molto

simile alla struttura di un file *Json*, con alcune differenze legate perlopiù alle dimensioni dei documenti. Infatti *Firebase* è ottimizzato per lavorare con molti documenti di piccole dimensioni.

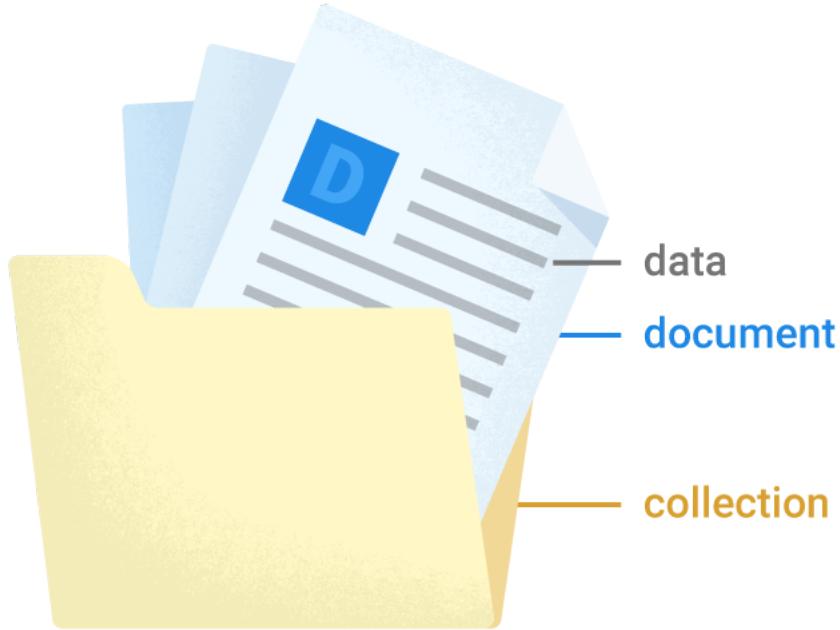


Figura 8: Struttura di *Firebase*

## 2.7 Mockup

Di seguito riportiamo i mockup utilizzati per strutturare graficamente l'applicazione. In Figura 9a è riportata la pagina che si apre quando l'applicazione viene lanciata. Da questa è possibile scegliere tra due alternative, effettuare direttamente il login (Figura 9c), nel caso in cui si è già registrati, oppure essere rimandati alla pagina di registrazione (Figura 9b) nella quale, dopo aver inserito correttamente i propri dati ed essersi registrati, si viene automaticamente reindirizzati alla pagina di login per effettuare l'accesso.

Una volta autenticati, si arriva alla homepage dell'applicazione, che consiste nella pagina che viene mostrata in Figura 9d. Questa pagina fornisce una search bar in alto, attraverso la quale è possibile ricercare i libri di interesse, e da una bottom navigation in basso, utilizzando la quale è possibile spostarsi tra le varie pagine dell'applicazione.

Inserito il titolo del libro, viene fornita una lista di libri restituita dall'API di Google Books (Figura 10a). Una volta individuato il libro di interesse, cliccando su di esso, comparirà la pagina delle info del libro, attraverso la quale è anche possibile selezionare una delle biblioteche presso le quali è possibile ritirarlo (Figura 10b), ed è anche possibile visualizzare delle recensioni inserite per quel libro con le relative valutazioni e "Mi piace" archiviati.

Se si desidera richiedere il libro presso una determinata biblioteca, verrà prima mostrato un calendario che richiede l'inserimento della data di inizio del prestito (ovviamente tutti i prestiti di libri presso biblioteche differenti andranno a buon fine, visto che non abbiamo implementato un meccanismo di comunicazione con quest'ultime) (Figura 10d).

Successivamente l'aver inserito una preferenza da uno a cinque stelle sulla rating bar, vi è la possibilità di inserire una recensione (Figura 11a). Ovviamente è poi possibile visualizzare tutte le proprie recensioni attraverso l'apposita schermata riportata in Figura 12c, con l'opportunità di modificarle ulteriormente o eliminarle del tutto.

Pigando sull'icona omino nella bottom navigation, comparirà la pagina del profilo riportante le proprie informazioni e con la possibilità di modifiche delle proprie credenziali (Figura 11c). Ovviamente, l'aggiornamento dei propri dati è fornito di vari metodi per la cattura di eccezioni e per la corretta scrittura di email e password inserite. Se la convalida andrà a buon fine, oltre alla visualizzazioni di piccoli e brevi messaggi a schermo, sarà visualizzato un fragment per la corretta modifica delle proprie credenziali (Figura 11d).

Per la gestione dei libri presi in prestito, vi è una pagina apposita accessibile tramite l'icona libro della bottom navigation. Qui saranno mostrati tutti i libri richiesti con le relative informazioni e la data di scadenza (Figura 12a). Cliccando su uno dei libri richiesti, si verrà ricondotti ad una pagina con i dettagli del libro e la possibilità di cancellazione del prestito.

In Figura 12d, in Figura 12c ed in Figura 12b sono riportati ulteriori mockup che riprendono le pagine da noi implementate per la visualizzazione di tutte le notifiche, accessibile dall'icona nella bottom navigation, la visualizzazione di tutti i "Mi piace" inseriti per i libri (accessibile da un floating action butto dalla homepage) e la pagina per tutte le recensioni fornite.

## 2.8 API utilizzate

Di seguito riportiamo le API da noi utilizzate per la ricerca dei libri e la localizzazione sulla mappa.

### 2.8.1 Google-books API

L'API di Google Books viene da noi utilizzata per la ricerca dei libri, ed è richiamata quando l'utente inserisce una stringa nella barra di ricerca in alto. In particolare, l'API fornita è la seguente: [https://www.googleapis.com/books/v1/volumes?q=search terms](https://www.googleapis.com/books/v1/volumes?q=search%20terms).

- $q$  è il parametro da inserire, che corrisponde al titolo del libro, autore... che si intende ricercare.

### 2.8.2 Google-maps API

Per quanto riguarda la geolocalizzazione delle biblioteche sulla mappa, è stato necessario l'utilizzo di due API di Google Maps per la corretta implementazione di questa funzione. La prima viene utilizzata per ottenere, data una stringa, le coordinate latitudine e longitudine del posto inserito. La seconda invece, viene utilizzata per localizzare il posto sulla mappa attraverso le coordinate latitudine e longitudine ottenute.

È stato necessario utilizzare l'API di "conversione" da stringa (il nome della biblioteca di nostro interesse) in coordinate, poiché l'API Opac utilizzata azzerava questi dati geografici, non consentendoci la corretta acquisizione sulla mappa (ciò è dovuto principalmente al fatto che l'API è ormai obsoleta e non viene più aggiornata e gestita da quasi un decennio, vedere riferimento: 2.8.3).

Le API in questione sono dunque:

- [https://maps.googleapis.com/maps/api/geocode/json?address=INDIRIZZO&key=TUA\\_CHIAVE\\_API](https://maps.googleapis.com/maps/api/geocode/json?address=INDIRIZZO&key=TUA_CHIAVE_API). Prende i seguenti parametri:
  - *address*: l'indirizzo stringa di cui si vogliono le coordinate.
  - *key*: l'API KEY ottenuta in fase di registrazione.
- *com.google.android.geo.API\_KEY* per localizzare le coordinate su mappa. I parametri sono i seguenti:
  - *API\_KEY*: la chiave API ottenuta in fase di registrazione.

### 2.8.3 opacmobilegw API

Questa API aveva come scopo quello di fornirci i dettagli su latitudine e longitudine di tutte le biblioteche a livello nazionale che avevano un determinato libro, individuato in maniera univoca dal suo **ISBN**. Sin da subito abbiamo avuto problemi con questa API. Il primo problema, è stato ricercare documentazione aggiornata sull'API. Per risolvere il problema, abbiamo scritto alla *software-house* che ha prodotto l'API, ovvero: "Inera". Di seguito si mostra la conversazione avvenuta con l'azienda:

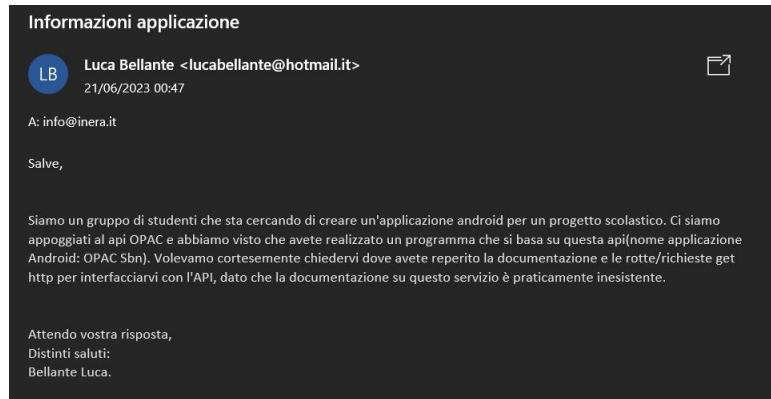


Figura 13: Inizio conversazione con *Inera*

In Figura 13, si osservi che la software-house ha prodotto non solo l'API ma anche un'applicazione mobile(Android) che utilizza quell'API. L'applicazione è diventata oramai obsoleta e non più presente nel Play-Store. La risposta fornitaci è la seguente:

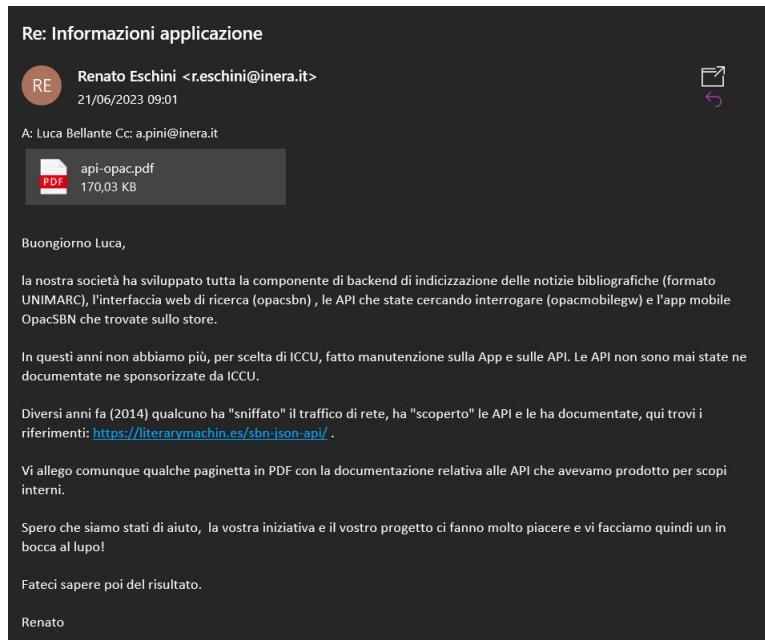


Figura 14: Risposta e documentazione di *Inera*

In Figura 14, il signor **Eschini** ci ha gentilmente fornito della documentazione inerente all'API completa di esempi di chiamate e risposte.

Il funzionamento dell'API doveva essere semplice, ovvero, dal *BID* dovevamo ottenere tutte le notizie bibliografiche legate ad un particolare testo, comprese le biblioteche che lo avevano con tanto di coordinate geografiche da passare successivamente all'API di Google-Maps. Dopo qualche prova ci siamo resi conto che in realtà tutte le coordinate geografiche erano 0.

Ci siamo accorti di questa anomalia analizzando un'esempio di chiamata http proveniente dalla documentazione dell'API.

La chiamata fatta all'API è stata la seguente:

```
http://opac.sbn.it/opacmobilegw/full.json?bid=ITICCUCAG1083075
```

In Figura 15, si mostra parte della documentazione con il risultato da noi sperato, ovvero con la presenza delle giuste coordinate geografiche delle biblioteche che avevano quel determinato libro individuato in maniera univoca dal codice *BID*.

## 1.4 Full bibliografico

La chiamata **full.json** permette di visualizzare il dettaglio di una notizia bibliografica specificando almeno uno tra i seguenti parametri:

Nome	Descrizione	Obbligatorietà	Valori ammessi	Default
bid	Codice identificativo	No	Testo	Nessuno
isbn	Codice identificativo internazionale	No	Testo	Nessuno

Ad esempio, la chiamata  `${url.opacmobilegw}/full.json?bid=IT\ICCU\CAG\1083075` produrrà la risposta in formato REST del tipo:

```
{"codiceIdentificativo": "IT\\ICCU\\CAG\\1083075", "titolo": "Dante", "descrizioneFisica": "1 CD-ROM ; in contenitore, 12 cm. + 1 fasc", "pubblicazione": "Milano : RCS, c2004", "livello": "Monografia", "tipo": "Risorsa elettronica", "collezione": "I grandi classici BUR. Letteratura italiana", "numeri": [], "note": ["Requisiti del sistema: Pentium 200 MHZ ; 32 Mb ; Window 95, 98, 2000 ; lettore di CD-ROM: 8X; scheda video 800x600 a migliaia di colori ; scheda audio sound blaster 16 bit o compatibile", "Tit. del contenitore."], "nomi": [], "luogoNormalizzato": [], "linguaPubblicazione": "ITALIANO", "paesePubblicazione": "ITALIA", "localizzazioni": [{"isil": "IT-B00300", "sbn": "UBOLS", "denominazione": "Biblioteca del Quartiere San Donato", "comune": "Bologna", "provincia": "BO", "latitudine": 44.5111006, "longitudine": 11.3953449}, {"isil": "IT-CA0010", "sbn": "CAGSS", "denominazione": "Biblioteca comunale Generale e di Studi Sardi", "comune": "Cagliari", "provincia": "CA", "latitudine": 39.2210322, "longitudine": 9.1050152}, {"isil": "IT-T00226", "sbn": "T01S1", "denominazione": "Biblioteca
```

```
Civica Multimediale Archimede", "comune": "Settimo Torinese", "provincia": "TO", "latitudine": 45.1366083, "longitudine": 7.7662135}], "citazioni": [{"standard": "mla", "valore": "Dante Milano RCS, 2004"}, {"standard": "apa", "valore": "(2004). Dante Milano RCS."}]}]
```

Figura 15: Esempio di chiamata *Http* e risposta da parte dell'API *opacmobilegw*

Come si può osservare dalla Figura 15, sono presenti le coordinate geografiche di tutte le biblioteche che hanno disponibile quel determinato testo/libro.

Al contrario, la risposta da noi ottenuta è stata la seguente:

```
{  
    "codiceIdentificativo": "IT\\ICCU\\CAG\\1083075",  
    "titolo": "Dante",  
    "descrizioneFisica": "1 CD-ROM ; in contenitore , 12 cm. + 1 fasc",  
    "pubblicazione": "Milano : RCS, c2004",  
    "livello": "Monografia",  
    "tipo": "Risorsa elettronica",  
    "collezione": "I grandi classici BUR. Letteratura italiana",  
    "numeri": [],  
    "note": [  
        "Requisiti del sistema: Pentium 200 MHZ ; 32 Mb ; Window 95, 98, 2000 ;  
        lettore di CD-ROM: 8X; scheda video 800x600 a migliaia di colori ; scheda audio  
        sound blaster 16 bit o compatibile",  
        "Tit. del contenitore."  
    ],  
    "nomi": [],  
    "luogoNormalizzato": [],  
    "linguaPubblicazione": "ITALIANO",  
    "paesePubblicazione": "ITALIA",  
    "localizzazioni": [  
        {  
            "isil": "IT-CA0010",  
            "denominazione": "CAGSS",  
            "latitudine": 0.0,  
            "longitudine": 0.0,  
            "shelfmarks": [  
                {  
                    "shelfmark": "Biblioteca comunale Generale e di Studi Sardi  
                    (presso la MEM)",  
                    "invNum": "Cagliari (CA)",  
                    "notes": null,  
                    "fruition": null  
                }  
            ]  
        },  
        {  
            "isil": "IT-TO0135",  
            "denominazione": "TO1M5",  
            "latitudine": 0.0,  
            "longitudine": 0.0,  
            "shelfmarks": [  
                {  
                    "shelfmark": "Biblioteca civica \"A. Arduino\"",  
                }  
            ]  
        }  
    ]  
}
```

```

        "invNum": "Moncalieri (TO)" ,
        "notes": null ,
        "fruition": null
    }
]
},
{
    "isil": "IT-TO0226" ,
    "denominazione": "TO1S1" ,
    "latitudine": 0.0 ,
    "longitudine": 0.0 ,
    "shelfmarks": [
        {
            "shelfmark": "Biblioteca Civica Multimediale Archimede" ,
            "invNum": "Settimo Torinese (TO)" ,
            "notes": null ,
            "fruition": null
        }
    ]
},
{
    "isil": "IT-BO0300" ,
    "denominazione": "UBOLS" ,
    "latitudine": 0.0 ,
    "longitudine": 0.0 ,
    "shelfmarks": [
        {
            "shelfmark": "Biblioteca Luigi Spina nel quartiere San Donato" ,
            "invNum": "San Vitale" ,
            "notes": "Bologna (BO)" ,
            "fruition": null
        }
    ]
},
],
"citazioni": [
    {
        "standard": "mla" ,
        "valore": "Dante Milano RCS,2004"
    },
    {
        "standard": "apa" ,
        "valore": "(2004). Dante Milano RCS."
    }
]
}

```

Come è possibile osservare dal codice nella Sez. 2.8.3, le coordinate geografiche sono tutte nulle, anche se il codice *BID* dei due esempi è identico(si veda Figura: 15).

Abbiamo quindi esposto il "nuovo" Problema alla software-house *Inera*, scrivendo un'altra email(vedere figura: 16):

**R: Informazioni applicazione**

Luca Bellante <lucabellante@hotmail.it>

21/06/2023 10:26

A: Renato Eschini Cc: a.pini@inera.it

message.txt  
2,83 KB

Buongiorno,

Nuovamente mi scuso per il disturbo. La ringrazio per il tempo che ci ha dedicato. Tramite PostMan abbiamo provato ad effettuare varie chiamate, alcune andate a buon fine ed altre no.  
In particolare, seguendo l'esempio del paragrafo 1.4(pagina 6), abbiamo riscontrato un risultato diverso da quello atteso. Nel dettaglio, **latitudine** e **longitudine** delle biblioteche sono tutte zero. Secondo Lei, potrebbe essere dovuto ad un problema a livello di Database di ICCU ? Magari dopo diversi anni possono aver cambiato qualcosa.

In allegato, risposta alla chiamata(come da documentazione):  
<http://opac.sbn.it/opacmobilegw/full.json?bid=ITICCUCAG1083075>

Saluti,  
Bellante Luca.

Figura 16: Comunicazione del nuovo problema ad *Inera*

Dopo l'email 16, *Inera* ci ha prontamente risposto:

**Re: R: Informazioni applicazione**

Renato Eschini <r.eschini@inera.it>

22/06/2023 16:35

A: Luca Bellante

Ciao Luca,  
come ti dicevo non mettiamo mano a quel codice da anni, più delle informazioni che ti ho dato non saprei aiutarti.

Mi spiace.

Renato.

Figura 17: Risposta di *Inera* al nuovo problema

Dopo la risposta di Figura: 17, abbiamo deciso di optare per una soluzione diversa.

Dal codice *BID*, effettuiamo la chiamata tramite l'API di *opacmobilegw*, che ci restituisce un *Json* con vari risultati, tra cui una lista di biblioteche che hanno disponibile quel particolare libro. I nomi delle biblioteche, saranno i parametri da inserire in un'altra chiamata *Http*, questa volta fatta all'API: "Google-Maps API". La chiamata *Http*, ci restituisce in output le coordinate geografiche(latitudine e longitudine), questa volta non nulle, delle biblioteche date in input. Abbiamo, quindi "Bypassato" il problema dell'API *opacmobilegw*, facendo, per ciascun libro, tante chiamate *Http* all'api di *Google-Maps API* quante sono le biblioteche che hanno quel determinato libro/testo. Noi pensiamo che il problema del malfunzionamento sia da attribuire alla mancata manutenzione dell'API.

### 3 Programmazione in Android

Come accennato precedentemente (Sez. 1), l'applicazione sarà sia per dispositivi iOS che per dispositivi Android. Per la programmazione in Android abbiamo utilizzato l'IDE Android Studio, mentre come linguaggio Kotlin, per la programmazione e lo sviluppo della logica dell'applicazione, e l'XML per lo sviluppo dei relativi layout corrispondenti ad activity, fragment ecc.

Contrariamente all'età e alla presenza di documentazione del linguaggio, è stata presa la decisione di utilizzare *Kotlin* per gestire la logica del programma anziché *Java*. Questa scelta è stata motivata dal fatto che *Kotlin*, sviluppato da **JetBrains**, è un linguaggio più conciso e offre strumenti integrati per evitare situazioni indesiderate durante l'esecuzione dell'applicazione, come la *NullPointerException*, che viene evitata grazie all'operatore "elvis" e all'operatore di "safe call".

Inoltre, per definire l'interfaccia utente dell'applicazione, abbiamo utilizzato il linguaggio *XML* (eXtensible Markup Language), il quale è un linguaggio di markup che consente di strutturare i *layout* che vengono utilizzati per definire l'interfaccia utente dell'applicazione.

L'utilizzo di XML per definire l'interfaccia utente presenta diversi vantaggi. In primo luogo separa la logica dell'applicazione dalla presentazione dell'interfaccia, consentendo una maggiore modularità e facilitando la manutenzione del codice grazie alla sua leggibilità e strutturazione.

L'integrazione di *Kotlin* e *XML* all'interno di Android Studio offre un ambiente di sviluppo potente e intuitivo per la creazione di app Android. Questo ci ha permesso di sfruttare al meglio le funzionalità offerte dai due linguaggi e ha semplificato lo sviluppo di un'interfaccia utente accattivante e funzionale.

#### 3.1 MVVM

Alla base della struttura dell'applicazione si trova l'adozione di un design pattern specifico chiamato **MVVM** (Model View ViewModel). Questo pattern architettonico è composto da tre componenti principali: il Model, la View e il ViewModel.

Il *Model* rappresenta i dati e la logica di business dell'applicazione. Questa componente si occupa di gestire i dati, effettuare operazioni di recupero o salvataggio, e fornire le funzionalità necessarie per l'interazione con il sistema sottostante.

La *View* invece rappresenta l'interfaccia utente dell'applicazione. Questo elemento si occupa di visualizzare i dati e gestire l'interazione con l'utente. La *View* è responsabile dell'aspetto grafico e dell'interazione dell'utente con gli elementi dell'interfaccia.

Il *ViewModel* funge da intermediario tra il Model e la View. Questa componente si occupa di gestire lo stato dell'applicazione e coordinare le operazioni tra il Model e la View. Il *ViewModel* esegue il binding dei dati tra il Model e la View, consentendo loro di comunicare in modo indipendente.

L'utilizzo del pattern MVVM (Model-View-ViewModel) offre una serie di vantaggi significativi nell'architettura delle applicazioni Android. Questo approccio promuove una migliore separazione delle

responsabilità, un codice più modulare e testabile, nonché un’interazione più fluida tra la logica di business e l’interfaccia utente. Inoltre, il ViewModel svolge un ruolo chiave nel mantenere uno stato coerente durante i cambiamenti di configurazione dell’applicazione.

### 3.2 Struttura delle cartelle del progetto

L’adozione dell’MVVM e di vari componenti di Android ha portato a una struttura ben organizzata delle cartelle dell’applicazione. Ecco una panoramica delle principali cartelle utilizzate per strutturare il codice Kotlin:

- *activity*: questa cartella contiene le classi che rappresentano le activity dell’applicazione. Le activity sono responsabili di gestire l’interazione con l’utente e coordinare la logica dell’applicazione.
- *adapter*: all’interno di questa cartella sono presenti gli adapters utilizzati per associare i dati alle viste recyclerView. Gli adapters vengono utilizzati per mostrare elenchi o griglie di elementi e consentono di collegare i dati del modello all’interfaccia utente.
- *cache*: questa cartella ospita classi o utility per gestire invece la memorizzazione nella cache dei dati. La cache è utilizzata per memorizzare temporaneamente i dati recuperati da un’origine remota, consentendo un accesso più rapido e riducendo la dipendenza dalla connessione di rete.
- *components*: all’interno di questa cartella sono presenti componenti personalizzati utilizzati nell’interfaccia utente. Questi componenti possono includere widget personalizzati, controlli o altri elementi riutilizzabili che vengono utilizzati in diverse parti dell’applicazione.
- *firebase*: questa cartella contiene classi o utility per l’integrazione con il servizio Firebase. Firebase è una piattaforma di sviluppo mobile offerta da Google che offre una serie di servizi utili, come autenticazione degli utenti, database in tempo reale e notifiche push.
- *fragments*: all’interno di questa cartella sono presenti i fragments, che sono componenti modulari utilizzati per creare interfacce utente complesse. I fragments consentono di separare l’interfaccia utente in parti riutilizzabili e possono essere combinati per creare layout flessibili che si adattano a diverse dimensioni e orientamenti dello schermo.
- *interface*: questa cartella contiene le interfacce che definiscono i contratti di comunicazione. Le interfacce consentono una comunicazione chiara e definiscono i metodi che devono essere implementati dalle classi che le utilizzano, promuovendo un basso accoppiamento tra i componenti del sistema.
- *model*: all’interno di questa cartella sono presenti le classi che rappresentano i modelli dei dati dell’applicazione. Queste classi definiscono la struttura dei dati e possono includere metodi per l’accesso o la manipolazione dei dati correlati.
- *utils*: questa cartella ospita classi di utility o helper che forniscono funzionalità comuni. Le classi di utilità possono contenere metodi di supporto generici, funzioni di formattazione dei dati o altre funzionalità di supporto che possono essere utilizzate in diverse parti dell’applicazione.
- *viewmodel*: all’interno di questa cartella sono presenti le classi che implementano i ViewModel del pattern MVVM. I ViewModel sono responsabili di fornire i dati necessari all’interfaccia utente e di gestire le azioni dell’utente. Essi comunicano con il modello e possono esporre metodi per l’aggiornamento dei dati o l’elaborazione delle interazioni dell’utente.

L'organizzazione del codice sorgente Kotlin dell'applicazione utilizza queste cartelle per favorire la separazione delle responsabilità, la riusabilità del codice e la manutenibilità complessiva dell'applicazione. Grazie a questa struttura ben definita, siamo stati in grado di lavorare in modo più efficiente, rendendo più facile la ricerca delle classi e promuovendo una migliore comprensione del flusso dell'applicazione.

Per poter sviluppare un'opportuna applicazione Android completa e funzionale, è stato necessario anche utilizzare un altro linguaggio, l'**XML**, i cui file si trovano all'interno di un'altra sezione del programma, **res**, che contiene tutte le cartelle delle risorse utili per definire l'aspetto visivo, la grafica e il comportamento dell'app:

- *anim*: questa cartella contiene file XML che definiscono animazioni personalizzate utilizzate nell'applicazione. Questi file descrivono le trasformazioni, le transizioni e gli effetti visivi che possono essere applicati agli elementi dell'interfaccia utente.
- *drawable*: all'interno di questa cartella sono presenti invece le risorse grafiche come immagini e icone utilizzate nell'applicazione. È qui che vengono salvate le immagini di sfondo, le icone dei pulsanti e tutto ciò che serve per arricchire l'aspetto visivo dell'app.
- *layout*: questa cartella contiene i file XML che definiscono la struttura e l'organizzazione degli elementi dell'interfaccia utente. Qui vengono definiti i layout delle schermate, come ad esempio la disposizione dei pulsanti, le caselle di testo, le liste e così via.
- *menu*: all'interno di questa cartella sono presenti i file XML che definiscono i menu dell'applicazione. Questi menu possono includere opzioni di navigazione, scelte di configurazione o altre azioni specifiche che l'utente può eseguire all'interno dell'app.
- *mipmap*: questa cartella è utilizzata per conservare le icone dell'applicazione in diverse densità di pixel. Android infatti selezionerà automaticamente l'icona corretta in base al dispositivo su cui viene eseguita l'applicazione, garantendo un aspetto uniforme su diverse risoluzioni del display.
- *navigation*: all'interno di questa cartella sono presenti i file XML che definiscono il flusso di navigazione dell'applicazione. Qui è possibile specificare i percorsi tra le diverse schermate e le azioni associate a ciascuna transizione.
- *raw*: questa cartella è utilizzata per memorizzare file di dati non elaborati utilizzati dall'applicazione, come ad esempio file audio, video o JSON.
- *values*: all'interno di questa cartella sono presenti i file XML per la definizione dei valori di risorse. Qui è possibile definire stringhe, colori, dimensioni, stili e temi che verranno utilizzati nell'applicazione. Questa cartella aiuta a separare la logica di programmazione dal contenuto visivo dell'applicazione.
- *xml*: questa cartella viene utilizzata per archiviare altri file XML utilizzati nell'applicazione che non rientrano in nessuna delle altre cartelle menzionate in precedenza. Questi file possono contenere, ad esempio, configurazioni specifiche o definizioni personalizzate.

L'utilizzo di queste cartelle e dei relativi file XML ci ha permesso di organizzare le risorse dell'applicazione in modo strutturato e accessibile durante lo sviluppo dell'app Android. Inoltre, l'utilizzo del

linguaggio XML ci ha offerto una configurazione flessibile e modulare delle risorse, consentendo di separare chiaramente l'aspetto e il comportamento dell'applicazione dalla logica di programmazione.

Infine particolare attenzione si deve dare anche a tre file fondamentali nello sviluppo di un'app Android:

- *AndroidManifest.xml*: questo file è essenziale per ogni app Android in quanto contiene informazioni cruciali sull'app stessa. Definisce il pacchetto dell'app, il nome dell'app, le attività principali e i servizi offerti dall'app. Specifica anche le autorizzazioni richieste dall'app per accedere alle risorse o alle funzionalità del dispositivo. Inoltre, l'AndroidManifest.xml definisce anche le impostazioni di sicurezza dell'app e le configurazioni dell'interfaccia utente.
- *build.gradle (Module: app)*: questo file viene utilizzato per configurare il processo di compilazione e le dipendenze dell'app. Contiene le informazioni sulle versioni delle librerie utilizzate, le configurazioni dei plugin e altre impostazioni di compilazione. È possibile specificare la versione minima e massima di Android supportata dall'app. È anche possibile aggiungere dipendenze esterne alle librerie utilizzate nell'app, come librerie per il networking, la grafica, i test, ecc. Il build.gradle (Module: app) è specifico per il modulo dell'app corrente.
- *build.gradle (Project: <Biblioteca\_Nazionale>)*: questo file definisce le impostazioni globali per l'intero progetto Android. Contiene la configurazione del repository Maven, che viene utilizzato per scaricare le dipendenze delle librerie esterne. È possibile configurare le impostazioni di build comuni a tutti i moduli dell'app, ad esempio la versione del compilatore Java utilizzata o il numero di versione dell'applicazione. È possibile anche aggiungere script personalizzati per automatizzare attività come il rilascio dell'app o la generazione di build firmate.

In conclusione quindi, questi file combinati insieme ci hanno garantito un'adeguata configurazione e gestione dell'app durante il processo di sviluppo, permettendoci di sviluppare un'esperienza utente ottimizzata e affidabile.

### 3.3 Approfondimenti sul codice

Da questo punto in poi, quindi, sarà svolta un'analisi approfondita di tutti i file presenti all'interno delle nostre cartelle in maniera da offrire una visione chiara e generale delle funzionalità del progetto.

#### 3.3.1 MainActivity

La classe `MainActivity` rappresenta l'attività principale dell'applicazione. Nel suo metodo `onCreate()`, vengono eseguite diverse operazioni, tra cui l'impostazione del layout, l'inizializzazione dei pulsanti di accesso e registrazione, e l'ottenimento dell'istanza di `FirebaseAuth` per la gestione dell'autenticazione degli utenti.

Successivamente, viene effettuato un controllo per verificare se l'utente è già autenticato. Se sì, viene avviata l'attività `HomePageActivity`. Altrimenti, vengono impostati i listener di click per i pulsanti di accesso e registrazione.

Quando l'utente preme il pulsante di accesso, viene avviata l'attività `LoginActivity`. Analogamente, quando l'utente preme il pulsante di registrazione, viene avviata l'attività `RegistrationActivity`.

### 3.3.2 Activity

Come detto in precedenza qui vengono definite le attività principali dell'applicazione, i file sono:

- *HomePageActivity*: in questa activity, che fa riferimento al layout *home\_page.xml*, viene settato il NavHostFragment dal supportFragmentManager utilizzando l'ID fragmentContainer. Questo frammento viene utilizzato per ospitare i fragment dell'applicazione e gestire la navigazione tra di essi.

Viene inoltre ottenuta un'istanza di NavController dal NavHostFragment per gestire la navigazione tra i fragment. Successivamente viene configurata la BottomNavigationView con ID bottom\_navigation per utilizzare il NavController per la gestione della sua navigazione.

- *LoginActivity*: all'interno di questa activity, che fa riferimento al layout *login.xml*, vengono gestiti i processi di accesso degli utenti attraverso diversi metodi.

Il login con Google è stato configurato creando un oggetto GoogleSignInOptions e istanziando un client per l'accesso tramite Google.

Quando l'utente seleziona il pulsante di registrazione con Google, viene avviato il processo di accesso tramite Google. I risultati di questa operazione vengono gestiti nel metodo *handleResults*, in cui viene ottenuto l'account Google dell'utente e l'interfaccia utente viene aggiornata di conseguenza.

Nel metodo *onStart* viene effettuato un controllo per verificare se l'utente è già autenticato. In caso positivo, viene avviata direttamente l'attività *HomePageActivity* per l'utente loggato.

Quando l'utente seleziona il pulsante "Accedi", viene eseguito il processo di accesso utilizzando l'email e la password fornite. Se l'accesso ha successo, viene verificato se l'utente è presente nella collezione "Utenti" nel database Firebase. Se l'utente non è presente, viene salvato come nuovo utente.

Nel caso si verifichino errori durante il processo di accesso, la situazione viene gestita nel metodo *loginErrorHandling*. Vengono identificati i tipi di errore e vengono visualizzati messaggi di errore appropriati all'utente tramite una finestra di dialogo.

- *RegistrationActivity*: in questa activity, che fa riferimento al layout *register.xml*, viene gestita la registrazione dell'utente.

Quando l'utente preme il pulsante "Accedi" nell'attività di registrazione, viene avviata l'attività *LoginActivity* per consentire all'utente di effettuare l'accesso.

Quando l'utente preme il pulsante "Registrati", vengono prelevati l'email, la password e la conferma della password fornite dall'utente. Se tutti i campi sono compilati correttamente e la password corrisponde alla conferma della password, viene eseguita la creazione dell'utente utilizzando il metodo *createUserWithEmailAndPassword* di FirebaseAuth. Se la registrazione ha successo, l'utente viene reindirizzato all'attività di accesso (LoginActivity). In caso contrario, viene gestito l'errore tramite il metodo *registerErrorHandling*.

### 3.3.3 Adapter

Qui vengono settati gli adapter per la gestione delle recyclerView:

- *BookAdapter*: questo adapter è stato utilizzato per popolare di oggetti *MiniBook* i layout *item\_book.xml* della RecyclerView presente nel file *fragment\_my\_books.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi, di associare i dati e di gestire gli eventi di clic sugli elementi.

- *BookListAdapter*: questo adapter è stato utilizzato per popolare di oggetti *BooksResponse* i layout *searching\_result.xml* della RecyclerView presente nel file *fragment\_book\_list.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi, di associare i dati e di gestire gli eventi di clic sugli elementi.
- *LikesAdapter*: questo adapter è stato utilizzato per popolare di oggetti *Book* i layout *searching\_result.xml* della RecyclerView presente nel file *fragment\_my\_likes.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi, di associare i dati e di gestire gli eventi di clic sugli elementi.
- *MyReviewsAdapter*: questo adapter è stato utilizzato per popolare di oggetti *Review* i layout *single\_my\_review.xml* della RecyclerView presente nel file *fragment\_my\_reviews.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi, di associare i dati, di gestire gli eventi e, in questo caso, anche di gestire la visualizzazione del testo delle recensioni e dei pulsanti di espansione tramite i metodi *updateTextReviewVisibility* e *updateTextChangeReviewText*.
- *NotificationAdapter*: questo adapter è stato utilizzato per popolare di oggetti *Triple* i layout *notification.xml* della RecyclerView presente nel file *fragment\_notifications.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi e di associare i dati.
- *ReviewsAdapter*: questo adapter è stato utilizzato per popolare di oggetti *TemporaryReview* i layout *single\_review.xml* delle RecyclerView presenti nei file *fragment\_reviews.xml* e *fragment\_book\_info.xml*. L'adattatore si occupa di creare le visualizzazioni degli elementi, di associare i dati e, in questo caso, anche di gestire la visualizzazione del testo delle recensioni e dei pulsanti di espansione tramite i metodi *updateTextReviewVisibility* e *updateTextChangeReviewText*.

### 3.3.4 Cache

Come detto in precedenza qui vengono impostati i sistemi di memorizzazione cache dei dati:

- *GeocodingCache*: questa classe implementa una cache per gli oggetti *GeocodingResult* associandoli ad una chiave univoca. La cache utilizza una LruCache, che è una cache a dimensione limitata basata sull'algoritmo "Least Recently Used" (LRU).

La dimensione massima della cache è specificata dalla costante *CACHE\_SIZE*, che è impostata su 1000.

Questa classe viene utilizzata nel file *BookInfoFragment*.

- *LibrariesCache*: questa classe implementa una cache per le istanze di *List* contenenti oggetti di tipo *RequestCodeLocation*. Ogni lista è associata a una chiave univoca. La cache si basa sull'utilizzo della classe LruCache, che implementa un algoritmo di gestione della cache chiamato "Least Recently Used" (LRU).

La dimensione massima della cache è determinata dalla costante *CACHE\_SIZE*, che attualmente è impostata su 50.

La classe descritta è utilizzata nel file *RequestViewModel*.

- *ReviewCache*: questa classe implementa una cache per le istanze di *List* contenenti oggetti di tipo *Review*. Ogni lista è associata a una chiave univoca. La cache si basa sull'utilizzo della classe LruCache, che implementa un algoritmo di gestione della cache chiamato "Least Recently Used" (LRU).

La dimensione massima della cache è determinata dalla costante *CACHE\_SIZE*, che attualmente è impostata su 50.

La classe descritta è utilizzata nel file *BookListAdapter*.

### 3.3.5 Components

All'interno di questa cartella sono presenti componenti personalizzati utilizzati nell'interfaccia utente:

- *CustomMapView*: questa classe estende la classe *MapView* del framework Google Maps, ed è utilizzata per personalizzare il comportamento degli eventi di tocco all'interno della mappa.

Nel metodo *dispatchTouchEvent*, viene chiamato il metodo *requestDisallowInterceptTouchEvent(true)* sul genitore del *CustomMapView*. Questo impedisce al genitore di intercettare gli eventi di tocco nella mappa, consentendo al *CustomMapView* di gestirli autonomamente.

Ciò è stato implementato al fine di inserire la mappa all'interno di una *ScrollView*. Infatti, la classe è stata utilizzata all'interno del file *fragment\_book\_info.xml* il quale è proprio composto da una *ScrollView*.

### 3.3.6 Firebase

Questa cartella, come detto in precedenza, contiene classi o utility per l'integrazione con il servizio Firebase:

- *FirebaseDB*: questa classe fornisce funzionalità per gestire l'accesso e la manipolazione dei dati nel database Firebase, consentendo di interagire con il database e recuperare le informazioni necessarie per l'applicazione.

### 3.3.7 Fragments

In questa cartella sono presenti i fragments, che sono componenti modulari e riutilizzabili adoperati per creare interfacce utente complesse:

- *BookInfoFragment*: questo fragment, che fa riferimento al layout *fragment\_book\_info.xml*, rappresenta la pagina principale in cui sono riportate le informazioni di un libro. Questa schermata può essere divisa in sezioni. La sezione superiore è quella in cui vengono riporatati i dati del libro, come la copertina, l'autore e la descrizione del libro. La visibilità di quest'ultima viene gestita dal metodo *manageDescription*, che imposta il pulsante "show more" in modo tale da mostrare tutta la descrizione o solo le prime cinque righe.

Di fianco alla copertina c'è un *imageView* a forma di cuore che funge da pulsante per i "mi piace", con accanto il numero di questi ultimi. Questo tasto viene gestito dal metodo *manageLikes*, a cui bisogna passare l'oggetto che rappresenta il libro. All'interno di questo metodo viene utilizzato il *FirebaseViewModel* con il metodo *getMiPiace* per ottenere tutti i "mi piace" relativi al libro.

All'interno di *manageLikes* vengono anche utilizzati i metodi *setLike* e *unSetLike* che, rispettivamente, utilizzano il metodo *addNewMiPiace* e *deleteMiPiace* del *FirebaseViewModel* per gestire l'aggiunta e l'eliminazione dei "mi piace". All'interno sia di *setLike* che di *unSetLike* si richiama il *clickListener* del bottone, e all'interno del listener di *setLike* si richiama anche *unSetLike*, e viceversa, così da gestire in maniera dinamica l'aggiunta e l'eliminazione dei "mi piace".

Sotto questa prima sezione c'è un'altra dedicata alla mappa di prenotazione, implementata utilizzando la classe *MapView*. La mappa viene impostata in modo asincrono utilizzando il metodo *getMapAsync*. All'interno della mappa sono presenti tutti i marker e i cluster che rappresentano le biblioteche in cui è disponibile il libro da prenotare. I nomi delle biblioteche vengono ottenuti tramite il metodo *fetchDataBook* del *modelRequest*. Successivamente, i nomi delle biblioteche vengono passati all'oggetto *GeoApiClient* per ottenere le rispettive longitudini e latitudini, in modo da impostare i marker e i cluster sulla mappa.

Sulla mappa viene inoltre impostata opportunamente la posizione iniziale della telecamera. Nel caso in cui l'utente abbia disattivato il GPS, la posizione di partenza viene impostata sul primo marker della lista. Altrimenti, viene impostata sul marker più vicino all'utente. Per utilizzare la posizione dell'utente, è stato implementato un sistema di *requestPermission* per richiedere i permessi e controllarne lo stato.

Per impostare la telecamera predefinita, viene utilizzato il metodo *setDefaultCamera*, che verifica i permessi relativi alla posizione. All'interno di questo metodo, viene utilizzato anche il metodo *noLibraryFound*, che gestisce il caso in cui non siano state trovate biblioteche, disabilitando quindi il pulsante di prenotazione. Nel caso in cui i permessi siano stati concessi, utilizzando il metodo *findNearestMarker*, viene individuato il marker più vicino alla posizione dell'utente e viene applicato il metodo *setDefaultLibraryCamera* per spostare la telecamera e impostare il listener del pulsante di prenotazione. Nel caso in cui i permessi non siano stati concessi, viene semplicemente utilizzato *setDefaultLibraryCamera* sul primo elemento della lista dei marker. Ogni marker sulla mappa è associato a un click listener che utilizza *setDefaultLibraryCamera* per impostare in modo opportuno la telecamera e il listener del pulsante di prenotazione.

Nel metodo *setDefaultLibraryCamera*, viene utilizzato *setDefaultLibrary* per impostare adeguatamente il pulsante di prenotazione e gestire il caso in cui l'utente abbia già prenotato il libro in quella specifica biblioteca. Grazie a questo metodo, quando l'utente fa clic su un marker, compare una *TextView* che mostra il nome della biblioteca e viene impostato il listener del pulsante di prenotazione. Qui viene valutato se il libro è già stato prenotato utilizzando il metodo *bookIsBooked* del *FirebaseViewModel*.

Quindi, se l'utente fa clic sul pulsante di prenotazione e ha già prenotato il libro, viene visualizzato un messaggio che lo informa che il libro è già stato prenotato. Altrimenti, viene visualizzato un calendario in cui l'utente può selezionare la data di prenotazione del prestito. Dopo aver effettuato questa scelta, compare una *TextView* che mostra la data di scadenza del prestito, che è fissata a 14 giorni dopo la data selezionata. Nel caso in cui un utente abbia già prenotato il libro in una biblioteca, viene utilizzato il metodo *expirationDate* del *FirebaseViewModel* per mostrare direttamente la data di scadenza del libro al clic del marker della biblioteca.

La sezione successiva è quella dedicata alla gestione della ratingBar e della recensione dell'utente. Tutto ciò avviene all'interno del metodo *manageRatingBars* in cui vengono settate tutte le rating bars. Ciò avviene utilizzando il metodo *getAllCommentsByIsbn* del *FirebaseViewModel*, che prende tutte le recensioni con voti di un libro, e poi calcolando le statistiche. A questo punto si usa il metodo *getUsersCommentByIsbn* del *FirebaseViewModel* per prendere, nel caso ci fosse, la recensione dell'utente di quel libro. Nel caso essa fosse assente si darebbe la possibilità all'utente di scrivere la recensione premendo sul pulsante "Write review", il quale compare dopo che l'utente ha inserito un voto nella rating bar di valutazione. Il pulsante rimanda al fragment *WriteReviewFragment*. Nel caso fosse presente, invece, la recensione comparirebbe e si darebbe la possibilità

all’utente di modificarla attraverso il pulsante ”change your review”, che rimanderebbe sempre al fragment *WriteReviewFragment*.

L’ultima sezione del fragment riguarda la recyclerView delle recensioni. Questa recyclerView viene configurata utilizzando il metodo *manageRecyclerView*, nel quale viene utilizzato l’adapter *ReviewsAdapter*. All’adapter vengono passati le tre recensione più recenti relativi al libro visualizzato. Le recensioni vengono recuperate utilizzando il metodo *getUserByCommentsOfBooks* del *FirebaseViewModel*, che recupera tutte le recensioni per quel determinato libro. Nel caso in cui non siano presenti recensioni, il pulsante ”Show all reviews” viene nascosto. In caso contrario, viene mostrato e reindirizza al fragment *ReviewsFragment*, dove vengono visualizzate tutte le recensioni.

- *BookListFragment*: questo fragment, che fa riferimento al layout *fragment\_book\_list.xml*, è progettato per visualizzare una lista di libri che è il risultato della ricerca dell’utente.

All’interno del fragment vengono configurati alcuni elementi UI. Viene associato il pulsante di azione floating *fab* in maniera tale che sia effettuata una navigazione verso un altro frammento chiamato *MyLikesFragment* utilizzando il *NavController* associato al frammento corrente.

Nel layout, vengono visualizzati degli elementi a schermo che vengono impostati in maniera tale che vengono settati a visibilità HIDE al focus della search bar.

Successivamente, vengono configurati i listener per il campo di ricerca. Quando l’utente invia una query di ricerca o modifica il testo della query, vengono eseguite azioni specifiche. In entrambi i casi, viene mostrata la barra di avanzamento e nascosta la lista dei libri. La stringa di query viene confrontata con l’ultima query effettuata per evitare ricerche duplicate. Se la query è diversa dall’ultima effettuata, viene eseguita una ricerca effettiva utilizzando il metodo *performBookSearch*.

Il metodo *performBookSearch* gestisce l’esecuzione effettiva della ricerca dei libri. Viene impostato un layout manager di tipo *LinearLayoutManager* per la RecyclerView che mostrerà i risultati della ricerca. Viene quindi chiamato il metodo *searchBooks* del modello *BooksViewModel* per ottenere una lista di libri corrispondenti alla query. La RecyclerView viene popolata con i risultati ottenuti utilizzando un adapter personalizzato chiamato *BookListAdapter*. Viene inoltre impostato un listener per gestire il clic su un libro nella lista, consentendo all’utente di visualizzare ulteriori informazioni su quel libro.

Infine, vengono effettuate alcune operazioni per gestire il layout e la visibilità degli elementi UI in base allo stato della ricerca.

- *CredentialUpdated*: questo fragment, che fa riferimento al layout *fragment\_credential\_updated.xml*, contiene solo un’immagine con un testo di conferma per l’aggiornamento del profilo. Questo fragment include oltre all’immagine, anche un pulsante che, attraverso il *navController*, reindirizza all’fragment *ProfileFragment*.
- *DeleteBookingFragment*: questo fragment, che fa riferimento al layout *fragment\_delete\_booking.xml*, contiene le informazioni riguardanti il libro che è stato prenotato. L’unica parte interattiva del layout è un pulsante che, mediante il metodo *removeBookId* del *FirebaseViewModel*, consente di eliminare la prenotazione.
- *MyBooksFragment*: questo fragment fa riferimento al layout *fragment\_my\_books.xml*, il quale rappresenta l’elenco di tutti i libri prenotati dall’utente. Per realizzare ciò, viene utilizzato l’adattatore *BookAdapter*, al quale viene passato un *ArrayList* di *MiniBook* ottenuto sfruttando il metodo *getAllUser* del *FirebaseViewModel*.

- *MyLikesFragment*: questo fragment fa riferimento al layout *fragment\_my\_likes.xml*, il quale rappresenta l'elenco di tutti i "Mi Piace" dell'utente. Per realizzare ciò, viene utilizzato l'adapter *LikesAdapter*, al quale viene passato un *ArrayList* di oggetti *Book*. Questo elenco viene ottenuto utilizzando il metodo *getUserMiPiace* del *FirebaseViewModel*, che recupera tutti i "Mi Piace" dell'utente, e il metodo *searchBooksById* del *BooksViewModel*, che cerca i libri su Google Books in base all'ID salvato nel mi piace. Inoltre, viene assegnato un *click listener* agli elementi della *RecyclerView*, in modo da consentire la navigazione verso il *BookInfoFrag*.
- *MyReviewsFragment*: questo fragment fa riferimento al layout *fragment\_my\_reviews.xml*, il quale rappresenta l'elenco di tutte le recensioni dell'utente. Per realizzare ciò, viene utilizzato l'adapter *MyReviewsAdapter*, al quale viene passato un *ArrayList* di oggetti *Review*. Questo elenco viene ottenuto utilizzando il metodo *getUsersComments* del *FirebaseViewModel*, che recupera tutte le recensioni dell'utente. Inoltre, viene assegnato un *click listener* al pulsante "change review" presente negli elementi della *RecyclerView*, permettendo la navigazione verso il *WriteReviewFragment* per la modifica o l'eliminazione della recensione.
- *NotificationsFragment*: il NotificationsFragment viene utilizzato per visualizzare una lista di notifiche nell'interfaccia utente, esso corrisponde al layout *fragment\_notifications.xml*, con una material-toolbar in alto, ed una recyclerview che verrà riempita con le notifiche che l'utente riceverà quando i libri in prestito andranno sotto scadenza (mancano meno di due giorni alla data entro la quale restituire il libro). I singoli elementi di notifica sono rappresentati dal layout *notifications.xml*, che verrà correttamente riempito con i dati dall'adapter corrispondente, e verrà poi aggiunto alla recyclerview del fragment\_notifications.xml.
- *ProfileFragment*: questo fragment, che è associato al layout *fragment\_profile.xml*, che definisce l'aspetto visuale della schermata del profilo e contiene diversi elementi per mostrare e modificare i dati dell'account dell'utente.

All'interno del layout, sono presenti degli *editText* che consentono all'utente di inserire nuovi valori per i dati del proprio account, come l'indirizzo email o la password. Questi campi di input consentono all'utente di apportare modifiche direttamente dal profilo.

Inoltre, sono presenti vari pulsanti che permettono all'utente di eseguire diverse azioni legate al proprio account. Il pulsante di logout, che è presente nella *Toolbar*, consente all'utente di disconnettersi dall'applicazione, mentre il pulsante di modifica consente di avviare il processo di aggiornamento dei dati. Il pulsante di eliminazione dell'account permette all'utente di eliminare definitivamente il proprio account.

Tutti questi pulsanti sono gestiti attraverso l'utilizzo del *FirebaseViewModel*, che fornisce le funzionalità necessarie per gestire l'autenticazione e le operazioni sull'account utente tramite Firebase. In particolare, se l'utente ha effettuato l'accesso tramite un account Google, verrà consentita solo l'opzione di eliminazione dell'account e il logout, poiché i dati dell'account Google sono gestiti direttamente dal provider di autenticazione.

- *ReviewsFragment*: questo fragment fa riferimento al layout *fragment\_reviews.xml*, il quale rappresenta l'elenco di tutte le recensioni di un libro. Per realizzare ciò, viene utilizzato l'adapter *ReviewsAdapter*, al quale viene passato un *ArrayList* di oggetti *TemporaryReview*, una data class di supporto definita nel fragment stesso. Questo elenco viene ottenuto utilizzando il metodo *getUserByCommentsOfBooks* del *FirebaseViewModel*, il quale recupera tutti gli utenti che hanno recensito il libro.

- *WriteReviewFragment*: il WriteReviewFragment è quel fragment che viene richiamato quando l’utente decide di inserire una recensione. Esso corrisponde al layout *fragment\_write\_review.xml*, costituito dall’immagine del libro ed il titolo ad esso corrispondente. Procedendo verso il basso troviamo una rating bar nella quale è possibile inserire una valutazione da uno a cinque stelle e, sotto di essa vi è un pulsante ”delete your review” che consente di eliminare la propria recensione. Da metà pagina in giù, vi sono, poi, due caselle di testo che permettono l’inserimento del titolo della recensione ed il testo della recensione.

### 3.3.8 Model

In questa cartella, troviamo una serie di classi, le quali rappresentano la parte dell’applicazione che si occupa di gestire i dati e la logica di business. Le classi sono le seguenti:

1. *Book*: rappresenta un libro ed implementa l’interfaccia *Parcelable* per consentire il passaggio degli oggetti tra componenti dell’applicazione. La classe ”*Book*” ha due proprietà principali: ”*id*” di tipo String e ”*info*” di tipo *InfoBook*. La proprietà ”*id*” rappresenta l’identificatore univoco del libro, mentre la proprietà ”*info*” contiene ulteriori informazioni sul libro, rappresentate dall’oggetto di tipo *InfoBook*. La classe include anche metodi per la serializzazione e la deserializzazione degli oggetti Book tramite l’interfaccia *Parcelable*. Questi metodi consentono di scrivere l’oggetto Book in un *Parcel* (un contenitore per passare dati tra processi) e di leggerlo successivamente. Inoltre, la classe include un companion object che implementa l’interfaccia *Parcelable*. *Creator<Book>*. Questo companion object fornisce i metodi necessari per creare un’istanza di Book da un *Parcel* e per creare un array di Book di dimensione specificata. Complessivamente, la classe ”*Book*” rappresenta un libro con un identificatore univoco e informazioni dettagliate, e fornisce i mezzi per essere passato tra componenti dell’applicazione utilizzando la serializzazione *Parcelable*.
2. *BookFirebase*: rappresenta un libro con dati specifici per l’integrazione con Firebase e fornisce un metodo per ottenere una rappresentazione testuale dei dati contenuti nelle sue proprietà, ovvero un *override* del metodo *toString()*.
3. *BookResponse*: rappresenta la risposta di ricerca di libri e contiene una lista di oggetti ”*Book*” corrispondenti ai libri trovati. Questo permette di deserializzare i dati JSON ricevuti dalla chiamata API in oggetti ”*BooksResponse*” per utilizzarli nell’applicazione.
4. *InfoBook*: è un file Kotlin all’interno del quale risiedono due classi: ”*InfoBook*” e ”*ImageLink*”. La classe ”*InfoBook*” rappresenta le informazioni di un libro, inclusi il titolo, gli autori, la descrizione, l’editore, la data di pubblicazione e i link alle immagini. Le annotazioni ”*@SerializedName*” vengono utilizzate per mappare i dati provenienti dalla risposta JSON nei rispettivi attributi della classe. La classe implementa l’interfaccia ”*Parcelable*” per consentire la serializzazione e deserializzazione degli oggetti. La classe ”*ImageLinks*” rappresenta i link alle immagini del libro, inclusi il link per l’immagine in ”miniatura” e il link per l’immagine ”completa”. Anche quest’ultima implementa l’interfaccia ”*Parcelable*”. Entrambe le classi includono metodi per la scrittura e la lettura dei dati nell’oggetto ”*Parcel*” per la serializzazione e deserializzazione.
5. *Like*: rappresenta un like o un’aprezzamento dato a un libro all’interno dell’applicazione. Questo permette di gestire e trasmettere l’informazione riguardante i like dati ai libri. Prende in input, nel costruttore primario, solo l’id del libro a cui fa riferimento. La classe include anche un costruttore secondario che legge i dati da un oggetto ”*Parcel*” per creare un’istanza di ”*Like*”. Inoltre,

include i metodi per la scrittura e la lettura dei dati nell'oggetto "Parcel" per la serializzazione e deserializzazione.

6. *MiniBook*: rappresenta un libro con informazioni di base, come: l'ISBN, il luogo, l'immagine, la data e il titolo. Implementa l'interfaccia "Parcelable" per la serializzazione. Fornisce metodi per scrivere/leggere i dati in un "Parcel" e include un costruttore per la deserializzazione.
  7. *RequestBookCodes*: è una semplice classe che ha come attributi: il titolo del libro, il codice identificativo (id) e l'autore principale.
  8. *RequestBookName*: rappresenta una classe che contiene, al suo interno, come attributo: una lista di *RequestBookCodes*.
  9. *RequestCode*: come nel caso precedente, rappresenta una classe che contiene un unico attributo, ovvero una lista di *RequestCodeLocation*.
  10. *RequestCodeLibrary*: rappresenta una classe che contiene due attributi; "shelfmark" e "invNum", sono due stringhe dove *shelfmark* è il nome della biblioteca, mentre *invNum* è la località geografica dove si trova la biblioteca.
  11. *RequestCodeLocation*: rappresenta una classe che contiene due attributi; *isil* e *shelfmark*. L'*isil* è un codice identificativo univoco per ciascuna biblioteca, ed è mappato come una stringa, *Shelfmark*, invece, rappresenta una lista di *RequestCodeLibrary*.
- Tutte le classi di Request, cioè: *RequestBookCodes*, *RequestBookName*, *RequestCode*, *RequestCodeLibrary* e *RequestCodeLocation*; sono state utilizzate all'interno del codice con lo scopo di effettuare il parsing del Json restituito dall'API di *opacmobilegw*(vedere riferimento: 2.8.3) in oggetti facilmente utilizzabili all'interno dell'applicazione.
12. *Review*: Review è una data class contenente tutte le proprietà di una recensioni, quali ad esempio *idComment*, *reviewText*, *reviewTitle*, *isbn*, *vote*, *date*, *title* e *image*. Questa classe implementa l'interfaccia *Parcelable*, che in Android viene utilizzata per consentire l'incapsulamento di un oggetto in un "Parcel", che può essere utilizzato per passare dati tra componenti del sistema Android come attività (Activity) e frammenti (Fragment) o per archiviare gli oggetti su disco. Il processo di rendere una classe "Parcelable" comporta l'implementazione di determinati metodi (come quelli successivamente implementati) per consentire la scrittura e la lettura degli oggetti all'interno del "Parcel".
  13. *Users*: la classe Users rappresenta un modello di dati per gli utenti nel contesto dell'applicazione "Biblioteca Nazionale". La classe ha i seguenti attributi:
    - (a) *UID*: stringa che rappresenta l'ID univoco dell'utente.
    - (b) *email*: stringa che rappresenta l'indirizzo email dell'utente.
    - (c) *userSettings*: oggetto "UserSettings" che contiene le impostazioni dell'utente.
  14. *UserSettings*: rappresenta le impostazioni dell'utente nell'applicazione. La classe ha i seguenti attributi:
    - *libriPrenotati*: un elenco di oggetti "MiniBook" che rappresentano i libri prenotati dall'utente.
    - *commenti*: elenco di oggetti "Review" che rappresentano i commenti dell'utente.

- *miPiace*: elenco di oggetti "Like" che rappresentano i "Mi piace" dati dall'utente.

Tra i vari metodi implementati, invece, troviamo i seguenti:

- *toString()*: questo metodo ridefinisce il comportamento predefinito del metodo *toString()* per restituire una rappresentazione testuale dell'oggetto "UserSettings". Restituisce una stringa che combina i valori degli attributi.
- *addNewBook()*: questo metodo consente di aggiungere un nuovo libro prenotato all'elenco "libriPrenotati". Crea un nuovo oggetto "MiniBook" con i parametri forniti e lo aggiunge all'elenco. Se l'elenco "libriPrenotati" è null, viene inizializzato come un nuovo oggetto *ArrayList* prima di aggiungere l'elemento.
- *addNewLike()*: Questo metodo consente di aggiungere un nuovo oggetto Like all'elenco *miPiace*. Crea un nuovo oggetto Like con l'ID del libro fornito e lo aggiunge all'elenco. Se l'elenco *miPiace* è null, viene inizializzato come un nuovo oggetto *ArrayList* prima di aggiungere l'elemento.
- *deleteLike()*: questo metodo consente di eliminare un oggetto "Like" dall'elenco *miPiace* in base all'ID del libro fornito. Utilizza un iteratore per scorrere l'elenco e rimuove l'oggetto "Like" corrispondente all'ID del libro.
- *removeBook()*: questo metodo consente di rimuovere un libro prenotato dall'elenco *libriPrenotati* in base all'ISBN del libro fornito. Utilizza un iteratore per scorrere l'elenco e rimuove l'oggetto "MiniBook" corrispondente all'ISBN del libro.
- *addNewComment()*: questo metodo consente di aggiungere un nuovo commento all'elenco *commenti*. Crea un nuovo oggetto "Review" con i parametri forniti, inclusa la generazione di un ID univoco per il commento se non viene fornito un ID specifico. Aggiunge quindi l'oggetto "Review" all'elenco. Se l'elenco *commenti* è null, viene inizializzato come un nuovo oggetto *ArrayList* prima di aggiungere l'elemento.
- *removeComment()*: questo metodo consente di rimuovere un commento dall'elenco *commenti* in base all'ID del commento fornito. Utilizza un iteratore per scorrere l'elenco e rimuove l'oggetto "Review" corrispondente all'ID del commento.

### 3.3.9 ViewModel

In questa sezione, verranno trattati tutti i View-Model utilizzati nel progetto. Si ricorda che l'utilizzo di questi componenti è dettata dalla scelta progettuale di utilizzare il pattern *MVVM*(vedere riferimento 3.1).

1. *BooksViewModel*: fornisce metodi per cercare libri sia per titolo che per Id, utilizzando *GoogleBooksApiClient* per interagire con l'API di Google Books. Le risposte alle chiamate vengono fornite come oggetti *LiveData*, che possono essere osservati e reagire ai cambiamenti dei dati nell'applicazione.
2. *FirebaseViewModel*: Fornisce metodi per interfacciarsi con *FirebaseDB*(si veda riferimento 3.3.6) al fine di effettuare operazioni di:
  - (a) Lettura di un'utente e/o libro e/o commento.
  - (b) Scrittura di un nuovo utente e/o libro e/o commento.
  - (c) Modifica di un'utente e/o libro e/o commenti.

- (d) Eliminazione di un’utente e/o libro e/o commento.

I metodi ritornano diversi tipi di dati, come ad esempio: *LiveData*, *CompleteFuture*, *String* e *MutableLiveData*. In *FirebaseViewModel*, sono inoltre presenti dei metodi per convertire delle *HashMap* nei rispetti oggetti, di classi: ”*MiniBook*”, ”*Review*” e ”*Like*”.

3. *RequestViewModel*: viene utilizzato per ottenere informazioni sulla disponibilità di un libro presso le biblioteche. La classe utilizza *Retrofit* per effettuare chiamate di rete e *GsonConverterFactory* per la conversione dei dati *JSON*. La funzione principale, ”*fetchDataBook*”, recupera i dati relativi al libro richiesto. Se i dati sono presenti nella cache, vengono utilizzati direttamente, altrimenti viene effettuata una chiamata di rete per ottenere i dati. Successivamente i dati vengono elaborati per ottenere i codici e le posizioni delle biblioteche. Infine, i risultati vengono restituiti e memorizzati nella cache per utilizzi futuri.

### 3.3.10 Utils

In questa sezione sono presenti delle classi utili per il completo e corretto funzionamento dell’applicazione.

1. *GoogleBooksApiClient*: viene utilizzata per creare un oggetto *Retrofit* al fine di interagire con l’API di Google Books. La classe utilizza la libreria *Retrofit* e il convertitore *GSON* per gestire le chiamate di rete e la conversione dei dati *JSON*. All’interno del costruttore della classe viene creato un oggetto *Retrofit* e viene specificato l’URL di base dell’API di Google Books. Viene quindi creata un’istanza di ”*GoogleBooksApiService*” che viene utilizzata per effettuare le chiamate alle API di Google Books. Infine, viene fornito un metodo pubblico ”*getApiService*” per ottenere l’oggetto ”*GoogleBooksApiService*” creato.
2. *NotificationReceiver*: Estende la classe ”*BroadcastReceiver*”. *NotificationReceiver* viene utilizzata per gestire la ricezione di notifiche all’interno dell’applicazione. Quando viene ricevuto un broadcast di notifica, vengono estratti i dati relativi alla notifica come il titolo, il testo e l’ID univoco. Viene quindi gestito il salvataggio di queste informazioni utilizzando l’oggetto ”*SharedPreferences*” Per memorizzarle in modo persistente. Inoltre, viene generato un ID univoco per la notifica e viene creato un canale di notifica se il dispositivo utilizza una versione di Android 8.0 o successiva. Infine, viene costruito un oggetto ”*NotificationCompat.Builder*” Per creare la notifica e viene mostrata utilizzando il ”*NotificationManager*”.

## 3.4 Test

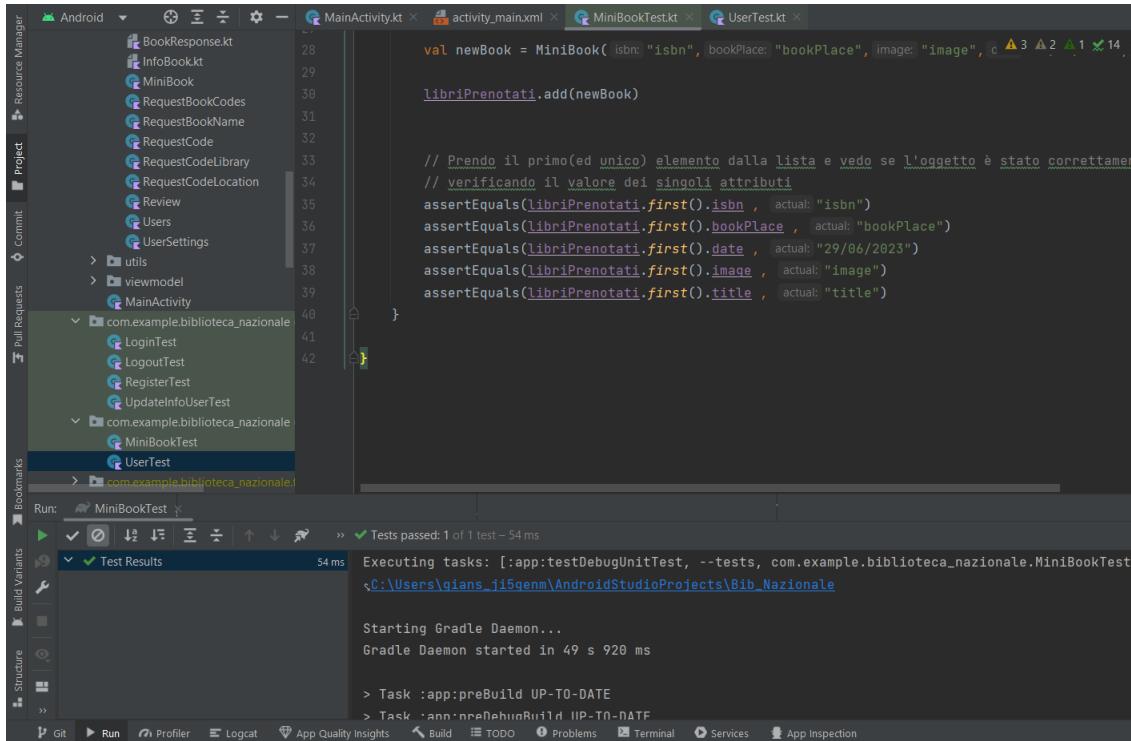
Abbiamo prodotto numerosi test, alcuni sono semplici Unit test, quindi test unitari svolti su singoli componenti per volta, altri sono degli Instrumented test, che testano la logica passando attraverso l’interfaccia, simulando quindi l’utilizzo vero e proprio da parte dell’utente. Di seguito la spiegazione dei test prodotti:

### 3.4.1 Unit test

Gli Unit test sono una pratica comune nello sviluppo del software che mira a testare le unità di codice in modo isolato per verificarne il corretto funzionamento. Una ”unità” in questo contesto può essere una singola funzione, un metodo di una classe o una classe stessa.

Gli Unit test sono classi di test costituite da un ambiente di "setting", nel quale vengono dichiarate le variabili da utilizzare ed identificati dal costrutto `@Before`, all'interno del metodo `setUp()`, l'ambiente di test nel quale vengono effettuati controlli sul corretto funzionamento della proprietà in questione, contrassegnati dal costrutto `@Test` sul metodo da utilizzare, e la parte dedicata alla pulizia delle risorse utilizzate, contrassegnata dalla direttiva `@After` nel metodo `tearDown()`.

Abbiamo due classi di Unit test, `MiniBookTest` e `UserTest`. `MiniBookTest` è un test basato sul costruttore della classe `MiniBook` per accertarci del suo corretto funzionamento (in Figura 18 è riportato il corretto funzionamento del test).



The screenshot shows the Android Studio interface during the execution of the `MiniBookTest`. The code editor displays the `MiniBookTest.kt` file with the following content:

```

    val newBook = MiniBook( isbn: "isbn", bookPlace: "bookPlace", image: "image", title: "title" )
    libriPrenotati.add(newBook)

    // Prendo il primo(ed unico) elemento dalla lista e vedo se l'oggetto è stato correttamente
    // verificando il valore dei singoli attributi
    assertEquals(libriPrenotati.first().isbn, actual: "isbn")
    assertEquals(libriPrenotati.first().bookPlace, actual: "bookPlace")
    assertEquals(libriPrenotati.first().date, actual: "29/06/2023")
    assertEquals(libriPrenotati.first().image, actual: "image")
    assertEquals(libriPrenotati.first().title, actual: "title")
}

```

The Project structure on the left shows the package `com.example.biblioteca_nazionale` containing several test classes: `LoginTest`, `LogoutTest`, `RegisterTest`, `UpdateInfoUserTest`, `MiniBookTest`, and `UserTest`.

The Run tab at the bottom shows the output of the test execution:

```

Tests passed: 1 of 1 test – 54 ms
Executing tasks: [:app:testDebugUnitTest, --tests, com.example.biblioteca_nazionale.MiniBookTest]
C:\Users\gians_j15genm\AndroidStudioProjects\Bib_Nazionale

Starting Gradle Daemon...
Gradle Daemon started in 49 s 920 ms

> Task :app:preBuild UP-TO-DATE
> Task :app:preReleaseBuild UP-TO-DATE

```

Figura 18: Andata a buon fine dello Unit test MiniBookTest

`UserTest`, viceversa, contiene al suo interno diversi test; tra questi abbiamo deciso di inserire il testing per la corretta aggiunta di un nuovo utente, per l'aggiunta e la rimozione di una nuova recensione e di un "Mi piace" (in Figura 19 è riportato il corretto funzionamento del test).

```

package com.example.biblioteca_nazionale

import ...

@JUnit4::class
class UserTest {

    lateinit var uid: String
    lateinit var email: String
    lateinit var userSettings: UserSettings
    lateinit var newUser: Users

    @Before
    fun beforeTest() {
        uid = "123456"
        email = "emailProva@gmail.com"
        userSettings = UserSettings(ArrayList(), ArrayList())
    }

    newUser = Users(uid, email, userSettings)
}

```

Figura 19: Andata a buon fine dello Unite test UserTest

In pratica questi test si basano sul confronto dei valori ottenuti con quello che noi ci aspettiamo, attraverso la direttiva messa a disposizione dall'ambiente di test, *assertEquals*.

### 3.4.2 Instrumented test

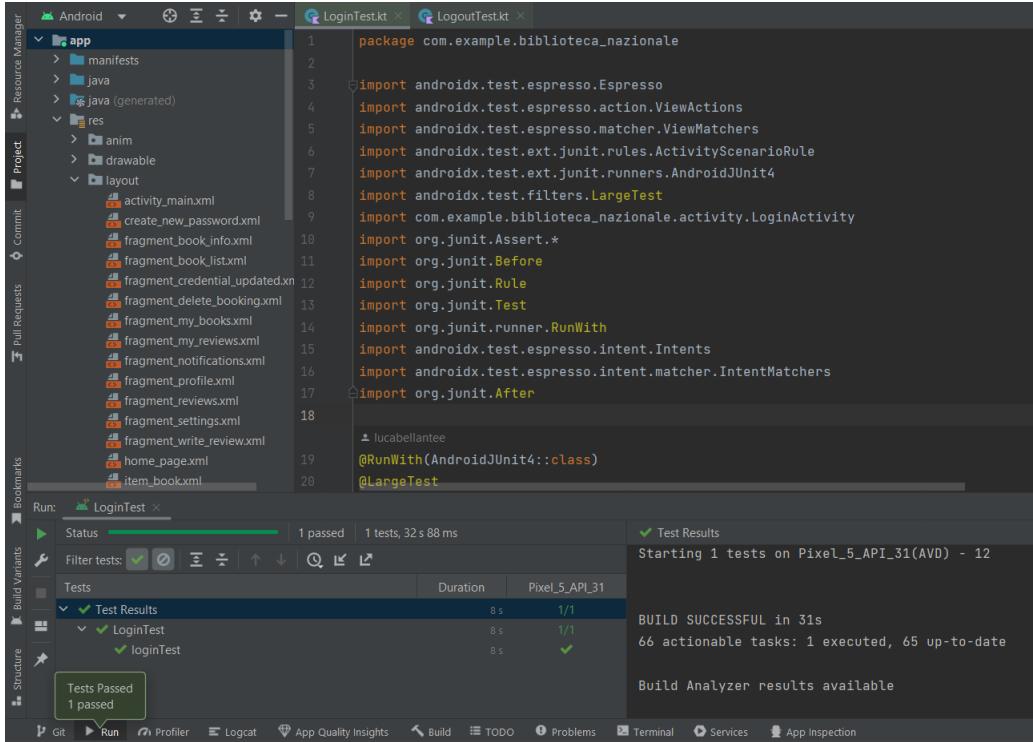
Per gli Instrumented test abbiamo utilizzato il framework di Google *Espresso*, che semplifica la scrittura e l'esecuzione di test UI affidabili e coerenti. Gli Android Instrumented Test con Espresso sono un modo per testare l'interfaccia utente (UI) delle app Android in modo automatizzato.

Gli strumented test vengono eseguiti direttamente sul dispositivo o sull'emulatore Android, consentendo di simulare l'interazione dell'utente con l'app e verificare il comportamento corretto dell'interfaccia utente. Gli strumented test con Espresso sono utili per garantire che le varie parti dell'app funzionino correttamente insieme e per identificare eventuali errori o problemi di usabilità.

Abbiamo implementato i seguenti Instrumented test (da testare nel seguente ordine per motivi legati al database: RegisterTest, LoginTest, UpdateInfoUserTest e LogoutTest):

- *LoginTest*: in questo test ci siamo accertati del corretto funzionamento della procedura di login facendo inserire un'email ed una stringa a nostro piacimento (ovviamente dopo aver fatto la registrazione, altrimenti il test fallisce in quanto nel database non sono presenti istanze corrispondenti a quelle credenziali) (Vedi Figura 20 per il corretto funzionamento del test). Tutto questo è stato possibile sfruttando delle direttive che il framework mette a disposizione, come ad esempio, *Espresso.onView*, *Intents.intended(IntentMatchers.hasComponent(LoginActivity::class.java.name))*

per verificare se è stata effettuata un'intenzione (Intent) per aprire una specifica attività (activity) all'interno dell'applicazione, `perform(ViewActions.typeText("test@gmail.com"))` per l'inserimento di un valore all'interno della casella di testo, e `Espresso.closeSoftKeyboard()` per la chiusura della tastiera utilizzata per l'inserimento dei dati.



The screenshot shows the Android Studio interface during the execution of an instrumented test. The left side displays the project structure under the 'app' module, showing various Java and XML files. The right side shows the code editor with the `LoginTest.kt` file open, containing Java code for testing. Below the code editor is the 'Run' tool window, which shows the test results: 1 passed test in 32 seconds. The status bar at the bottom indicates 'Tests Passed 1 passed'. To the right of the run window is the 'Test Results' panel, which shows the test was run on a Pixel\_5\_API\_31 device and was successful. The log output includes 'BUILD SUCCESSFUL in 31s' and '66 actionable tasks: 1 executed, 65 up-to-date'.

```

1 package com.example.biblioteca_nazionale
2
3 import androidx.test.espresso.Espresso
4 import androidx.test.espresso.action.ViewActions
5 import androidx.test.espresso.matcher.ViewMatchers
6 import androidx.test.ext.junit.rules.ActivityScenarioRule
7 import androidx.test.ext.junit.runners.AndroidJUnit4
8 import androidx.test.filters.LargeTest
9 import com.example.biblioteca_nazionale.activity.LoginActivity
10 import org.junit.Assert.*
11 import org.junit.Before
12 import org.junit.Rule
13 import org.junit.Test
14 import org.junit.runner.RunWith
15 import androidx.test.espresso.intent.Intents
16 import androidx.test.espresso.intent.matcher.IntentMatchers
17 import org.junit.After
18
19 lucabellante
20 @RunWith(AndroidJUnit4::class)
21 @LargeTest

```

Figura 20: Andata a buon fine dell'Instrumented test LoginTest

- *LogoutTest*: questo test ci è servito per accertarci che la procedura di logout venga effettivamente eseguita correttamente. Il tutto è svolto attraverso le direttive che mette a disposizione il framework, cioè `onView` e `perform(click())` per il clic sull'elemento. In Figura 21 è riportata l'andata a buon fine del test.

The screenshot shows the Android Studio interface with the project navigation bar at the top. Below it is the code editor displaying Java test code for `LogoutTestkt`. The code uses Espresso to perform UI interactions like clicking buttons and checking for text presence. The bottom half of the screen is the "Run" tool window, which displays the test results. It shows 1 passed test named `logoutTest` with a duration of 9.5 seconds. The status bar at the bottom indicates "Tests Passed 1 passed". To the right of the run window, there is a "Test Results" panel showing the build log: "BUILD SUCCESSFUL in 29s" and "66 actionable tasks: 5 executed, 61 up-to-date".

Figura 21: Andata a buon fine dell'Instrumented test LogoutTest

- *RegisterTest*: questo test garantisce che la procedura di registrazione venga effettuata così come era stata pensata. Infatti il test va a buon fine (come è possibile osservare in Figura 22). Il tutto utilizzando le solite direttive messe a disposizione del framework Espresso, quali ad esempio `onView`, `perform(ViewActions.typeText("topolini23@gmail.com"))` per l'inserimento nei campi di testo, e `Espresso.closeSoftKeyboard()` per la chiusura della tastiera utilizzata in fase di inserimento.

```

1 package com.example.biblioteca_nazionale
2
3 import androidx.test.espresso.Espresso
4 import androidx.test.espresso.action.ViewActions
5 import androidx.test.espresso.intent.Intents
6 import androidx.test.espresso.intent.matcher.IntentMatchers
7 import androidx.test.espresso.matcher.ViewMatchers
8 import androidx.test.ext.junit.rules.ActivityScenarioRule
9 import androidx.test.filters.LargeTest
10 import com.example.biblioteca_nazionale.activity.RegistrationActivity
11 import org.junit.After
12 import org.junit.Before
13 import org.junit.Rule
14 import org.junit.Test
15 import org.junit.runner.RunWith
16
17 import lucabellante
18
19 @RunWith(AndroidJUnit4::class)
20 @LargeTest

```

Run: RegisterTest

Status	1 passed	1 tests, 31 s 294 ms	
Filter tests:	<input checked="" type="checkbox"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/>	<input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/>	
Tests		Duration	Pixel_5_API_31

Test Results

Starting 1 tests on Pixel\_5\_API\_31(AVD) - 12

BUILD SUCCESSFUL in 31s  
66 actionable tasks: 1 executed, 65 up-to-date

Build Analyzer results available

Tests Passed  
1 passed

Figura 22: Andata a buon fine dell’Instrumented test RegisterTest

- *UpdateInfoUserTest*: questo test verifica il corretto funzionamento del procedimento di aggiornamento dei dati. Per fare ciò, anche qui, come nei precedenti casi, ci siamo serviti di varie direttive, tra cui il solito *on View, perform(click())* per il clic sull’elemento, *perform(typeText("test2@gmail.com"))* per l’inserimento dell’email nella casella di testo corrispondente, *closeSoftKeyboard()* per la chiusura della tastiera, e *check(matches(isDisplayed()))* per verificare se un elemento dell’interfaccia utente (UI) è visualizzato correttamente sullo schermo durante l’esecuzione di un test.

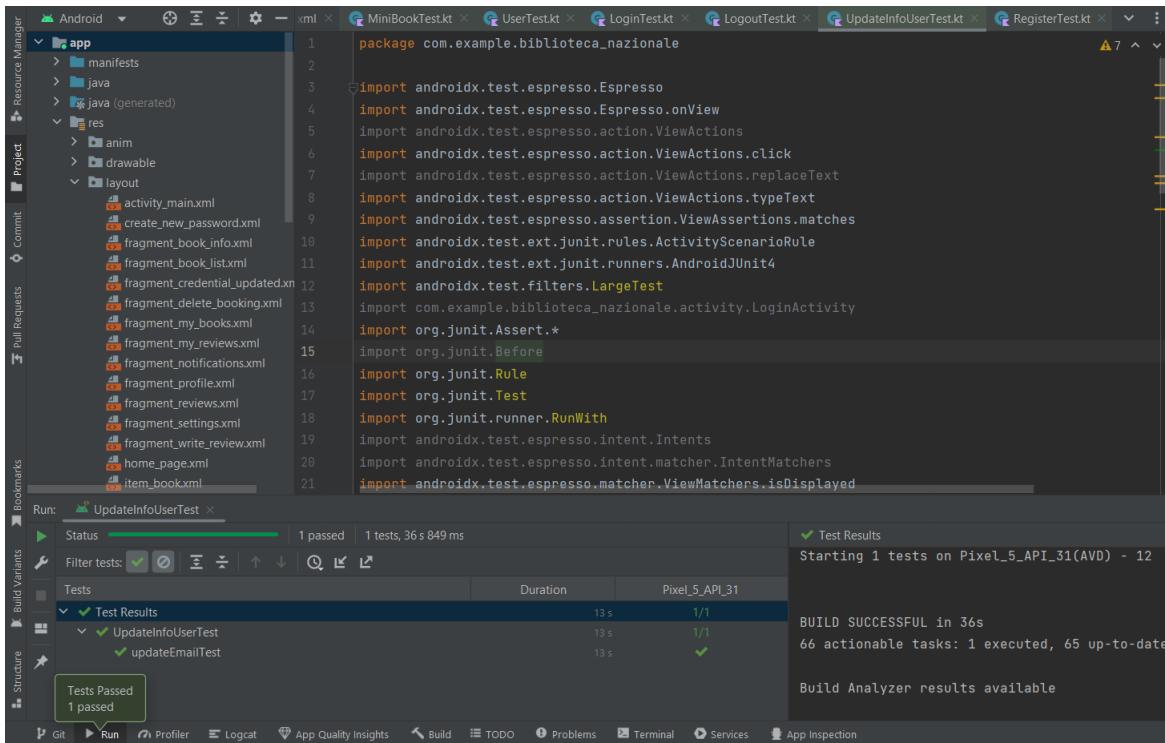


Figura 23: Andata a buon fine dell’Instrumented test UpdateInfoUserTest

## 4 Programmazione in Flutter

Flutter è un framework open-source sviluppato da Google per la creazione di applicazioni native cross-platform. Utilizza il linguaggio di programmazione Dart per costruire interfacce utente accattivanti e performanti su diverse piattaforme, come Android, iOS, web e desktop.

Di seguito verranno elencati i concetti chiavi della programmazione in Flutter:

- *Widget*: in Flutter, tutto è un widget. Un widget rappresenta un elemento dell’interfaccia utente, come un pulsante, un’immagine o una schermata intera. I widget possono essere combinati e nidificati per creare interfacce complesse.
- *Stateful e Stateless Widget*: Flutter offre due tipi principali di widget: stateful e stateless. Un widget stateful è un widget che può cambiare nel tempo, mentre un widget stateless è un widget immutabile. I widget stateful sono utilizzati per rappresentare elementi che possono avere uno stato interno, come un pulsante che cambia di colore quando viene premuto.
- *Hot Reload*: una delle caratteristiche distintive di Flutter è il ”hot reload”. Consente agli sviluppatori di apportare modifiche al codice e vedere immediatamente i risultati nell’applicazione senza dover ricompilare l’intero progetto. Questo velocizza notevolmente il processo di sviluppo.

- *Layout*: Flutter offre un potente sistema di layout per posizionare i widget sulla schermata. Puoi utilizzare vari widget per organizzare gli elementi, ad esempio "Container" per creare un layout personalizzato, "Row" e "Column" per allineare i widget in righe o colonne, o "Stack" per sovrapporli.
- *Gestione degli eventi*: è possibile gestire gli eventi utente, come il tocco di un pulsante o lo scorriamento di una lista, utilizzando vari widget di gestione degli eventi forniti da Flutter. È possibile ascoltare gli eventi e reagire di conseguenza per fornire un'esperienza interattiva agli utenti.
- *Pacchetti e librerie*: Flutter offre una vasta gamma di pacchetti e librerie pronte all'uso per semplificare lo sviluppo delle tue app. Si possono utilizzare queste librerie per aggiungere funzionalità come la connettività di rete, il salvataggio dei dati, la navigazione, la gestione degli stati e molto altro.

## 4.1 Approfondimento del codice

Procedendo in ordine di cartelle sulla vista del progetto nella sinistra dell'IDE, approfondiamo la struttura e il motivo dell'utilizzo delle classi implementate. È opportuno specificare che la suddivisione in cartelle è stata fatta per consentire una migliore visualizzazione e manovrabilità tra i vari file del progetto.

- *models*: questa è la cartella nella quale abbiamo deciso di inserire i "modelli" della nostra applicazione, forniti dei metodi necessari per l'interazione con altre componenti.
  1. *auth\_manager.dart*: utilizzata per consentire l'accesso all'id dell'utente attualmente loggato, anche in file diversi (ref. Figura 24).

```
class AuthManager {
  static final AuthManager _instance = AuthManager._internal();

  factory AuthManager() {
    return _instance;
  }

  int? currentUserID;

  AuthManager._internal();
}
```

Figura 24: Classe AuthManager

2. *DatabaseProvider.dart*: utilizzato per la definizione della nostra base di dati locale. Aprendo il file, infatti, possiamo notare metodi come "createDatabase" per la creazione delle tabelle di nostro interesse attraverso l'esecuzione delle query passate al metodo "execute" dell'istanza

”db”. Vi sono anche altri metodi utilizzati per la creazione dell’utente, il login, aggiunta, rimozione e aggiornamento dei libri ecc.

3. *dbbooks.dart*: file al cui interno è definito il modello di utilizzo per un oggetto libro all’interno del database (Figura 25).

```
class DBBook {  
    final int? id;  
    final String title;  
    final String authors;  
    final String image;  
    final String description;  
    final String library;  
  
    DBBook({  
        this.id,  
        required this.title,  
        required this.authors,  
        required this.image,  
        required this.description,  
        required this.library,  
    });  
  
    Map<String, dynamic> toMap() {  
        return {  
            'id': id,  
            'title': title,  
            'authors': authors,  
            'image': image,  
            'description': description,  
            'library': library,  
        };  
    }  
}
```

Figura 25: Classe DBBooks

4. *users.dart*: file al cui interno è definito il modello di utilizzo per un oggetto ”user” all’interno

del database (Figura 26).

```
class Users {  
    final int id;  
    final String email;  
    final String password;  
    bool isLoggedIn;  
  
    Users({  
        required this.id,  
        required this.email,  
        required this.password,  
        this.isLoggedIn = false,  
    });  
  
}
```

Figura 26: Classe Users

- *routes*: questa è la cartella in cui abbiamo definito le nostre rotte principali per la navigazione all'interno dell'app. Questo è stato fatto nel file *routes.dart*, dove sono state assegnate delle stringhe che, attraverso uno switch all'interno del metodo "generateRoute", possono essere richiamate per attivare la creazione delle vari classi implementate (Figura 27).

```

class AppRoutes {
    static const String home = '/';
    static const String login = '/login';
    static const String registration = '/registration';
    static const String homepage = '/homepage';
    static const String profile = '/profile';
    static const String bookInfo = '/bookInfoPage';
    static const String myBooks = '/myBooks';
    static const String deleteBook = '/deleteBooks';

    static Route<dynamic> generateRoute(RouteSettings settings) {
        switch (settings.name) {
            case home:
                return MaterialPageRoute(builder: (_) => MyHomePage());
            case login:
                return MaterialPageRoute(builder: (_) => Login());
            case registration:
                return MaterialPageRoute(builder: (_) => Register());
        }
    }
}

```

Figura 27: Classe AppRoutes

- *screens*: la cartella corrente contiene le "schermate" utilizzate dalla nostra applicazione. Vediamole nel dettaglio:
  1. *homepage.dart*: questa è la classe nella quale abbiamo definito la struttura della homepage con i relativi metodi. Essa è costituita anche da una bottom navigation che permette la navigazione tra le diverse schermate implementate.
  2. *login.dart*: questa è la classe utilizzata per l'implementazione dell'interfaccia utilizzata per il login. Dunque contiene tutto il necessario per definire la grafica, ed alcuni metodi importanti come *"loginUser"* (Figura 28), all'interno della classe *"LoginCardState"*, che consente l'autenticazione dell'utente semplicemente andando a prelevare le due stringhe inserite nel campi di email e password, richiamando il metodo del login implementato nella classe *"DatabaseProvider"* definita per l'interazione con il database, per poi sfociare nella gestione della grafica con la visualizzazione dei messaggi di andata a buon fine o meno.

```

Future<void> loginUser(BuildContext context) async {
    final String email = emailController.text;
    final String password = passwordController.text;

    final bool loginSuccess = await DatabaseProvider().loginUser(email, password);

    if (loginSuccess) {
        Navigator.pushNamed(context, '/homepage');
    } else {
        showDialog(
            context: context,
            builder: (context) => AlertDialog(
                title: Text('Login Failed'),
                content: Text('Invalid email or password.'),
                actions: [
                    TextButton(
                        child: Text('OK'),
                        onPressed: () {
                            Navigator.of(context).pop();
                        }
                    )
                ] // TextButton
            ) // AlertDialog
        );
    }
}

```

Figura 28: Metodo loginUser

3. *profile.dart*: questo file contiene tutto l'importante per la schermata del profilo accessibile dalla bottom navigation. È bene osservare l'utilizzo della classe "AuthManager" utilizzata per l'acquisizione dell'id dell'utente attualmente "loggato". Oltre alla solita costruzione dell'interfaccia grafica, ci teniamo a sottolineare l'utilizzo di metodi importanti come quello denominato "fetchCurrentEmail", che ci consente un aggiornamento istantaneo della stringa mostrata nella parte alta della finestra, nel momento in cui andiamo ad aggiornare le nostre credenziali. Altri metodi utilizzati sono "performDeleteProfile", che si occupa della visualizzazione di messaggi di avviso per l'azione di eliminazione dell'utente, oppure "deleteProfile" (Figura 29), il metodo che effettua l'azione vera e propria dell'eliminazione dell'utente andando a richiamare il metodo definito della classe "DatabaseProvider". Per concludere la presentazione di questa schermata, spendiamo alcune parole sul metodo da noi utilizzato per l'aggiornamento dei dati, "updateProfile". Esso utilizza la classe AuthManager per l'acquisizione dell'id dell'utente che è attualmente connesso, acquisisce le stringhe inserite nei campi attraverso dei controller impostati su di essi, ed effettua dei controlli sul corretto inserimento dell'email e delle password,

per poi andare ad aggiornare i dati attraverso l'utilizzo della solita classe "DatabaseProvider".

```
void deleteProfile(BuildContext context) {
    final int? currentUserId = AuthManager().currentUserId;
    if (currentUserId != null) {
        databaseProvider.removeUser(currentUserId);
        Navigator.pushReplacementNamed(context, '/login');
    } else {
        showDialog(
            context: context,
            builder: (BuildContext context) {
                return AlertDialog(
                    title: Text('Error'),
                    content: Text('User ID not found.'),
                    actions: [
                        TextButton(
                            child: Text('OK'),
                            onPressed: () {
                                Navigator.of(context).pop();
                            },
                        ),
                    ],
                );
            },
        );
    }
}
```

Figura 29: Metodo deleteProfile

4. *register.dart*: questa classe si occupa della registrazione dell'utente. A livello grafico il procedimento è sempre lo stesso, costruzione dei vari widget ed implementazione dei metodi che vogliamo vengano eseguiti quando si interagisce con essi. Tra questi, quello che maggiormente risalta è il metodo ”\_registerUser”, che si occupa della registrazione dell'utente attraverso, prima la convalida dei vari campi inseriti (è possibile noatre una serie di condizioni if-else in catena, e delle funzioni booleane che adoperano delle regex (Figura 30)), poi attraverso il metodo implementato nella classe ”DatabaseProvider” per la creazione dell'utente nel database.

```

class _RegisterFormState extends State<RegisterForm> {
    final TextEditingController _emailController = TextEditingController();
    final TextEditingController _passwordController = TextEditingController();
    final TextEditingController _confirmPasswordController =
        TextEditingController();

    bool isValidEmail(String email) {
        final RegExp emailRegex = RegExp(r'^[\w-]+(\.[\w-]+)*@[\w-]+(\.[\w-]+)+$');
        return emailRegex.hasMatch(email);
    }

    bool isValidPassword(String password) {
        final RegExp passwordRegex =
            RegExp(r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)[A-Za-z\d@$!%*?&]{8,}$');
        return passwordRegex.hasMatch(password);
    }
}

```

Figura 30: Regex utilizzate per la convalida dei campi email e password

Queste regex implementano semplicemente delle condizioni da rispettare per i campi inseriti, come ad esempio il numero di caratteri minimo che deve rispettare la password, l'obbligo di inserire almeno una lettera maiuscola ed un numero, oppure il fatto di avere delle stringhe divise dal carattere "@" per la stringa inserita come email.

- *utils*: in questa cartella abbiamo inserito tutto quello che sarebbe potuto tornare utile per l'implementazione degli altri componenti. Tra i vari elementi abbiamo deciso di comprendere anche le API utilizzate.
  1. *geocoding\_api*: in questa classe abbiamo implementato la logica di funzionamento per la chiamata dell'API di Google Maps che ci è servita per l'ottenimento delle coordinate latitudine e longitudine, partendo dal nome di una biblioteca (questo perché l'API di Opac azzerava questi valori) (Figura 31). Semplicemente viene passata all'API il nome della biblioteca e la nostra API KEY. Dopodiché, dal JSON restituito andiamo ad acquisire i singoli valori di latitudine e longitudine che verranno ritornati.

```

class GeocodingApi {
    final String baseUrl = "https://maps.googleapis.com/maps/api/geocode/json";
    final String apiKey = "AIzaSyCtTj2ohggFHTNX2asYNXL1kj31p08w0_Y";

    Future<Map<String, double>?> getCoordinates(String libraryName) async {
        try {
            final response = await http.get(Uri.parse("$baseUrl?address=$libraryName&key=$apiKey"));
            if (response.statusCode == 200) {
                final jsonData = json.decode(response.body);
                final results = jsonData['results'];
                if (results.isNotEmpty) {
                    final location = results[0]['geometry']['location'];
                    final latitude = location['lat'];
                    final longitude = location['lng'];
                    return {'latitude': latitude, 'longitude': longitude};
                }
            }
            return null;
        } catch (e) {
            // Gestione degli errori
            return null;
        }
    }
}

```

Figura 31: Implementazione per il funzionamento dell'API Geocoding di Google Maps

2. *google-api*: in questa classe abbiamo invece implementato la struttura per l'utilizzo dell'API di Google Books, che abbiamo utilizzato quando l'utente inserisce una stringa nella barra di ricerca. Semplicemente l'API ritorna tutti i libri che contengono alcune delle parole inserite. Infatti, come è possibile osservare nella Figura 32, il valore di ritorno del primo metodo equivale ad una lista di oggetti "Book".

```

class GoogleApi{
    // Costruttore
    GoogleApi();

    Future<List<Book>> getListOfBookByName(String bookName) async {
        List<Book> books = await GoogleBooksApi().searchBooks(
            bookName.toString(),
            //maxResults: 20,
            printType: PrintType.books,
            orderBy: OrderBy.relevance,
        );
        return books;
    }

    Future<Book> getBookById(String idBook) async{
        Book books = await GoogleBooksApi().getBookById(idBook);
        return books;
    }
}

```

Figura 32: Implementazione per il funzionamento dell'API di Google Books

3. *library-api*: library\_api.dart, in questo file abbiamo organizzato la struttura per l'utilizzo della seconda API di Opac. In pratica, mentre la prima (vedere riferimento: 4) restituisce una lista di libri trovati con tutte le relative informazioni, tra cui il codice identificativo, la seconda, attraverso quest'ultimo parametro, è in grado di fornirci anche tutte le biblioteche presso le quali è disponibile questo libro. Dunque, del JSON restituito, noi andiamo ad accedere agli oggetti "localizzazioni" e "shelfmarks" così da arrivare ad ottenere tutte le biblioteche, di cui si otterranno poi le coordinate e verranno infine localizzate sulla mappa.
4. *search-api*: questa è la prima API che viene chiamata tra le due di Opac. Essa restituisce, come già accennato nel punto precedente, un insieme di libri e tutte le loro proprietà. Di queste, ne acquisiamo una in particolare, il *codiceIdentificativo*, che viene richiesto come parametro dalla seconda API di Opac per l'ottenimento dell'elenco delle biblioteche che hanno disponibili il libro in questione (Figura 33).

```

class SearchApi {
    final String baseUrl = "http://opac.sbn.it/opacmobilegw/search.json";

    Future<List<String>> getBookId(String bookName) async {
        try {
            final url = Uri.parse("$baseUrl?any=${Uri.encodeQueryComponent(bookName)}");
            final response = await http.get(url);

            if (response.statusCode == 200) {
                final jsonData = jsonDecode(response.body);
                final results = jsonData['briefRecords'] as List<dynamic>;
                final bookIds = results.map((result) => result['codiceIdentificativo'] as String).toList();
                print('bookIds: $bookIds');
                return bookIds;
            } else {
                // Gestione degli errori
            }
        } catch (e) {
            print(e);
            print("Errore getBookId fallito");
            // Gestione degli errori
        }

        return [];
    }
}

```

Figura 33: Implementazione per il funzionamento dell'API di Opac per l'ottenimento delle informazioni su un libro

- *widgets*: in questa cartella abbiamo inserito alcuni estratti di codice, per l'implementazione grafica, che verranno richiamati in altri file. Abbiamo utilizzato questa logica ispirandoci ai fragment utilizzati per l'implementazione in Android.
  1. *bookDelete.dart*: file utilizzato per la parte di grafica riguardante l'eliminazione di un libro.
  2. *bookInfo.dart*: file utilizzato per la parte di grafica riguardante le info generali di un libro, come immagine di copertina, titolo, autore ecc.
  3. *bookList.dart*: file utilizzato per la parte di grafica riguardante la struttura sotto forma di lista, dei libri restituiti dalla ricerca.
  4. *my\_books.dart*: file utilizzato per la parte di grafica riguardante la visualizzazione di tutti i libri in possesso dell'utente.
  5. *searchResult.dart*: file utilizzato per la parte di grafica riguardante il singolo elemento da inserire all'interno della "listView" del file *bookList.dart*.
- *main.dart*: il file main.dart è il primo file ad essere mandato in esecuzione dall'app. Nel nostro caso, questo file contiene il necessario per impostare la grafica della pagina iniziale.

- *app\_theme.dart*: è il file nel quale è possibile impostare i colori scelti per le due tipologie di temi, bianco o nero. Per entrambe le tipologie, i colori sono stati da noi scelti identici così da consentire solo il tema bianco, per motivi qualitativi e di semplicità.
- *pubspec.yaml*: è il file nel quale devono essere inserite tutte le dipendenze per gli strumenti da utilizzare, e tutte le immagini devo essere dichiarate al suo interno prima di essere effettivamente utilizzate.

## 5 Errori e possibili bug

Dopo aver testato le due app prodotte su vari dispositivi, sia fisici che emulatori, non abbiamo constatato particolari comportamenti o anomalie che potrebbero portare a crash o bug improvvisi.

Tuttavia, intendiamo specificare un dettaglio. Per quanto riguarda l'app Android sviluppata in Kotlin, per la sua implementazione è stata utilizzata l'autenticazione di Firebase, consentendo l'utilizzo di due soli metodi, accesso tradizionale attraverso email e password, oppure accesso con Google. In merito all'accesso con Google vi sono delle considerazioni da fare.

Abbiamo notato che l'accesso è funzionante e correttamente implementato, nonostante ciò, esso funziona correttamente solo su alcuni dispositivi, mentre su altri no. Questo non vuol dire che l'app crea problemi o vada in errore, ma semplicemente rimane ferma sulla schermata di Login.

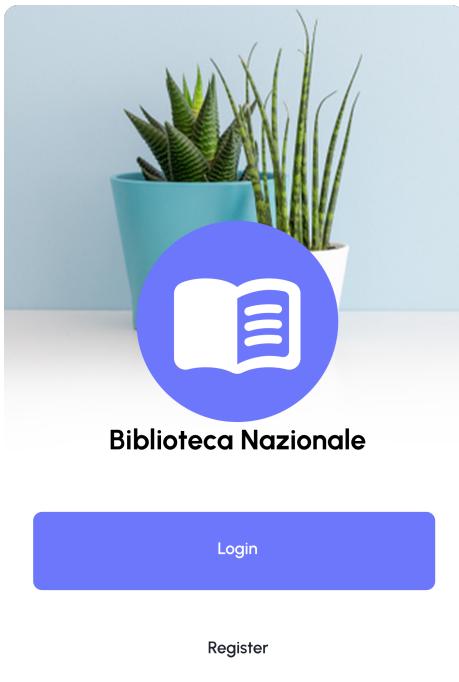
L'idea che ci siamo fatti su una possibile causa che potrebbe far scaturire questo problema, è che dipenda dalla versione di Android e dalle specifiche del dispositivo sul quale viene lanciata l'applicazione.

## 6 Ringraziamenti

Ringraziamo il professore *Emanuele Storti* per averci dato l'opportunità di metterci in gioco, insegnandoci le odierne tecniche di programmazione per dispositivi mobile(Android e iOS).

Ringraziamo la software-house *Inera*, in particolare il signor *Renato Eschini* per averci gentilmente fornito parte della loro documentazione riguardante l'API di *opacmobilegw*(si veda Sez. 2.8.3).

Ringraziamo anche i nostri amici e parenti che hanno testato l'applicazione evidenziandoci bug ed imperfezioni al fine di migliorarla e perfezionarla.



Login

Register

Register

Register

(a) Pagina di benvenuto

Hello! Register to get started

Username

Email

Password

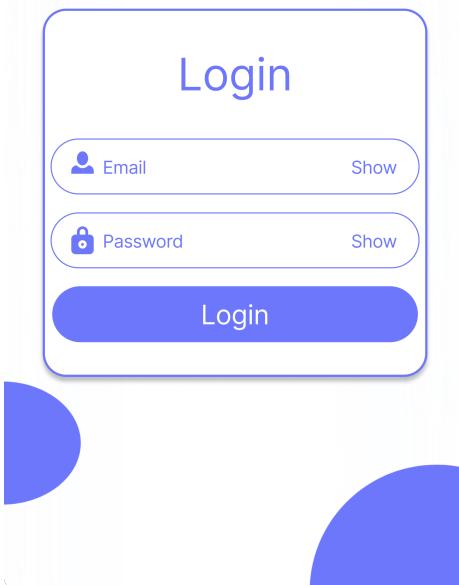
Confirm password

Register

Already have an account? [Login Now](#)

(b) Pagina di registrazione

Search



(c) Pagina di login

53



Welcome to the app of Biblioteca Nazionale



(d) Homepage



**Lord of the rings saga**  
★★★★★ 5.0 (123)  
The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.

**Lord of the rings: the fellowship of the ring**  
The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.

**Lord of the rings: the return of the king**  
The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.

**Lord of the rings: the two towers**  
The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.

**The Hobbit**  
The Hobbit is set in Middle-earth and follows Bilbo Baggins, the titular hobbit, who joins the wizard Gandalf and the dwarves that make up Thorin Oakenshield's Company, on a quest to reclaim the dwarves' home and treasure from the dragon Smaug.



(a) Pagina del risultato della ricerca



**Dettagli sulla valutazione**  
4,5 su 5 30.987 recensioni  
 60%  
 20%  
 10%  
 5%  
 5%

[Scrivi una recensione](#)

#### Recensioni

watson@gmail.com

I 100% recommend this book

I loved this book

Nov 09, 2022

connor@gmail.com

Very interesting book

Very nice

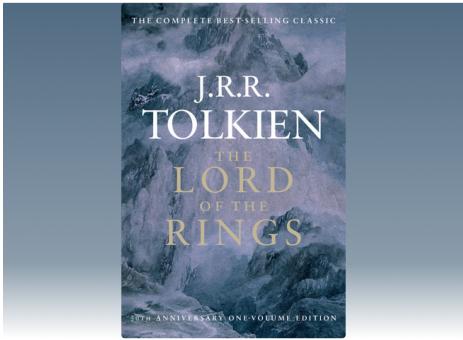
May 14, 2021



(c) Sezione delle recensioni di un libro



**The Lords of the rings trilogy**  
J.R.R. tolkien  
★★★★★ 5.0 (123)



#### Descrizione

The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil....

[Read more →](#)

Biblioteca tecnico scientifica  
UNIVPM facoltà di ingegneria  
Da riconsegnare entro il  
30/05/2023

[Prendi in prestito](#)



(b) Pagina di un libro



**The Lords of the rings trilogy**  
J.R.R. tolkien  
★★★★★ 5.0 (123)

september						
M	T	W	T	F	S	S
					1	2
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

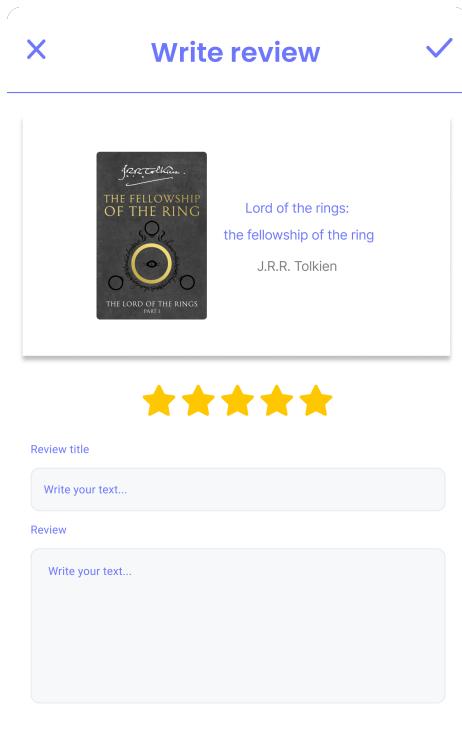
[Read more →](#)

Biblioteca tecnico scientifica  
UNIVPM facoltà di ingegneria  
Da riconsegnare entro il  
30/05/2023

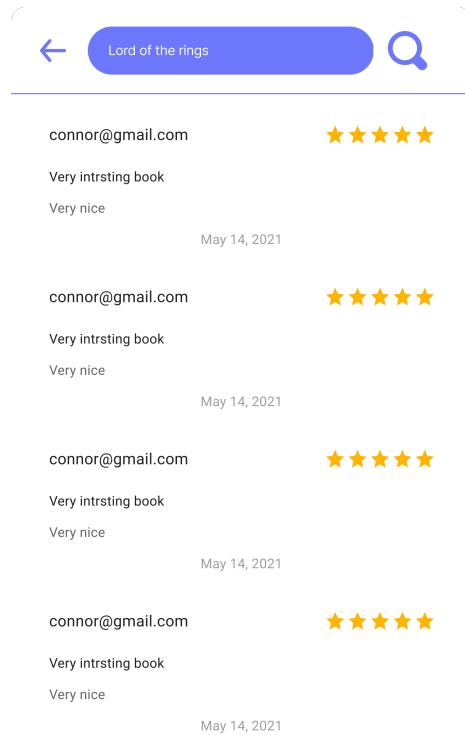
[Prendi in prestito](#)



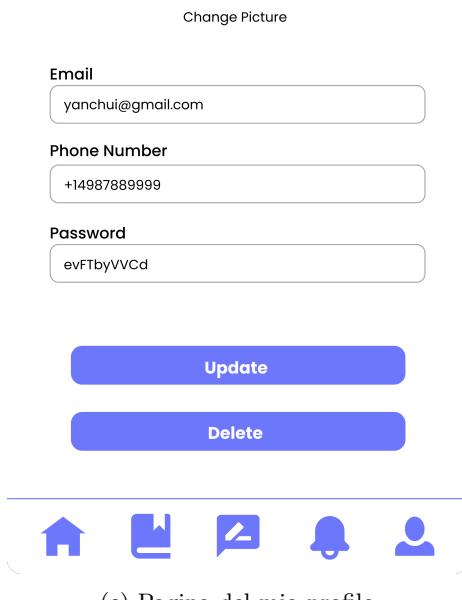
(d) Calendario di prenotazione



(a) Pagina di scrittura delle recensioni

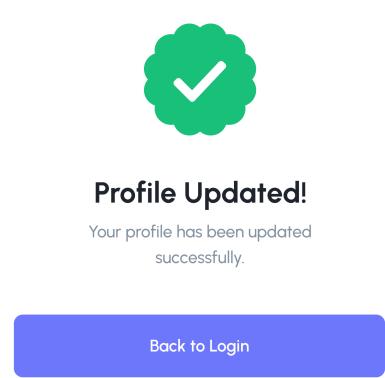


(b) Pagina delle recensioni di un libro



(c) Pagina del mio profilo

55



(d) Pagina di aggiornamento profilo

## My books

- 

[Lord of the rings saga](#) >

Da riconsegnare entro il 30/05/2023
- 

[Lord of the rings: the fellow...](#) >

Da riconsegnare entro il 30/05/2023
- 

[Lord of the rings: the return...](#) >

Da riconsegnare entro il 30/05/2023
- 

[Lord of the rings: the two...](#) >

Da riconsegnare entro il 30/05/2023
- 

[The Hobbit](#) >

Da riconsegnare entro il 30/05/2023



(a) Pagina dei miei libri prenotati

## My reviews

- 

connor@gmail.com 

Very intrstng book

Very nice

[Change Review](#) May 14, 2021
- 

connor@gmail.com 

Very intrstng book

Very nice

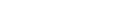
[Change Review](#) May 14, 2021
- 

connor@gmail.com 

Very intrstng book

Very nice

[Change Review](#) May 14, 2021
- 

connor@gmail.com 

Very intrstng book

Very nice

[Change Review](#) May 14, 2021



(c) Pagina delle mie recensioni

## Notifications

- 

La scadenza di "Lord of the rings saga" è vicina. Devi riconsegnarlo entro il 30/05/2023 >
- 

La scadenza di "Lord of the rings: the fellowship of the ring" è vicina. Devi riconsegnarlo entro il 30/05/2023 >
- 

La scadenza di "Lord of the rings: the return of the king" è vicina. Devi riconsegnarlo entro il 30/05/2023 >
- 

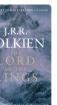
La scadenza di "Lord of the rings: the two towers" è vicina. Devi riconsegnarlo entro il 30/05/2023 >
- 

La scadenza di "The Hobbit" è vicina. Devi riconsegnarlo entro il 30/05/2023 >



(b) Pagina delle mie notifiche

## My likes

- 

[Lord of the rings saga](#)  
 5.0 (123)

The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.
- 

[Lord of the rings: the fellowship of the ring](#)

The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.
- 

[Lord of the rings: the return of the king](#)

The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.
- 

[Lord of the rings: the two towers](#)

The Lord of the Rings is the saga of a group of sometimes reluctant heroes who set forth to save their world from consummate evil.
- 

[The Hobbit](#)

The Hobbit is set in Middle-earth and follows Bilbo Baggins, the titular hobbit, who joins the wizard Gandalf and the dwarves that make up Thorin Oakenshield's Company, on a quest to reclaim the dwarves' home and treasure from the dragon Smaug.



(d) Pagina dei miei like