READING STATS
RELAZIONE DI PROGRAMMAZIONE MOBILE

Cristian Di Cintio - S1110150
Federico Di Giovannangelo - S0000000

Contents

1 Introduzione

L'applicazione sviluppata ha come obiettivo la memorizzazione delle statistiche di libri in lettura dell'utente. Tali statistiche riguardano la percentuale di completamento del libro e il tempo impiegato, memorizzato per ciascuno libro.

L'utente per usufruire delle funzionalità dell'app deve registrarsi con nome, cognome, email, username e password. Per lo username e l'email vengono effettuati dei controlli per verificare l'univocità e non permettere la registrazione di utenti con medesimo username o email.

In seguito l'utente può effettuare l'accesso con la propria email e password.

L'utente può ricercare i libri in un catalogo tramite la digitazione del titolo in una barra di ricerca o scansionando, con un pulsante apposito, il codice a barre di un libro fisico. In seguito può decidere di aggiungerlo in tre liste distinte, "da leggere", "in lettura" e "letti", per avere un resoconto delle pagine lette e del tempo impiegato nella lettura dei libri salvati. Il catalogo permette anche il filtraggio di libri tramite la selezione di categorie predefinite.

Al momento della lettura l'utente può avviare un timer per un libro in lettura, presente nell'omonima lista, e disattivarlo al termine della sessione.

L'utente può visualizzare e modificare i propri dati personali e interagire con altri utenti, visualizzando principalmente quali libri hanno salvato e quali stanno leggendo, ognuno con le loro rispettive metriche.

L'intera applicazione è stata sviluppata inizialmente in Kotlin per ambiente Android e in seguito in Flutter per un funzionamento multipiattaforma, in modo tale da permettere una maggiore compatibilità e una migliore interazione tra utenti con diversi dispositivi mobile.

Lo sviluppo in Kotlin è stato applicato con Jetpack Compose, per permettere uno sviluppo moderno e in linea con le attuali applicazioni sul mercato, con una struttura architetturale basata sul Model-View-ViewModel (MVVM) garantendo una composizione comprensibile per lo sviluppo dell'app. Le informazioni principali dei libri vengono visualizzate tramite l'utilizzo di API fornite da Google Books.

2 Progettazione e sviluppo Android

2.1 Requisiti e casi d'uso

Si riportano di seguito i requisiti *funzionali* e *non funzionali*

2.1.1 Requisiti funzionali

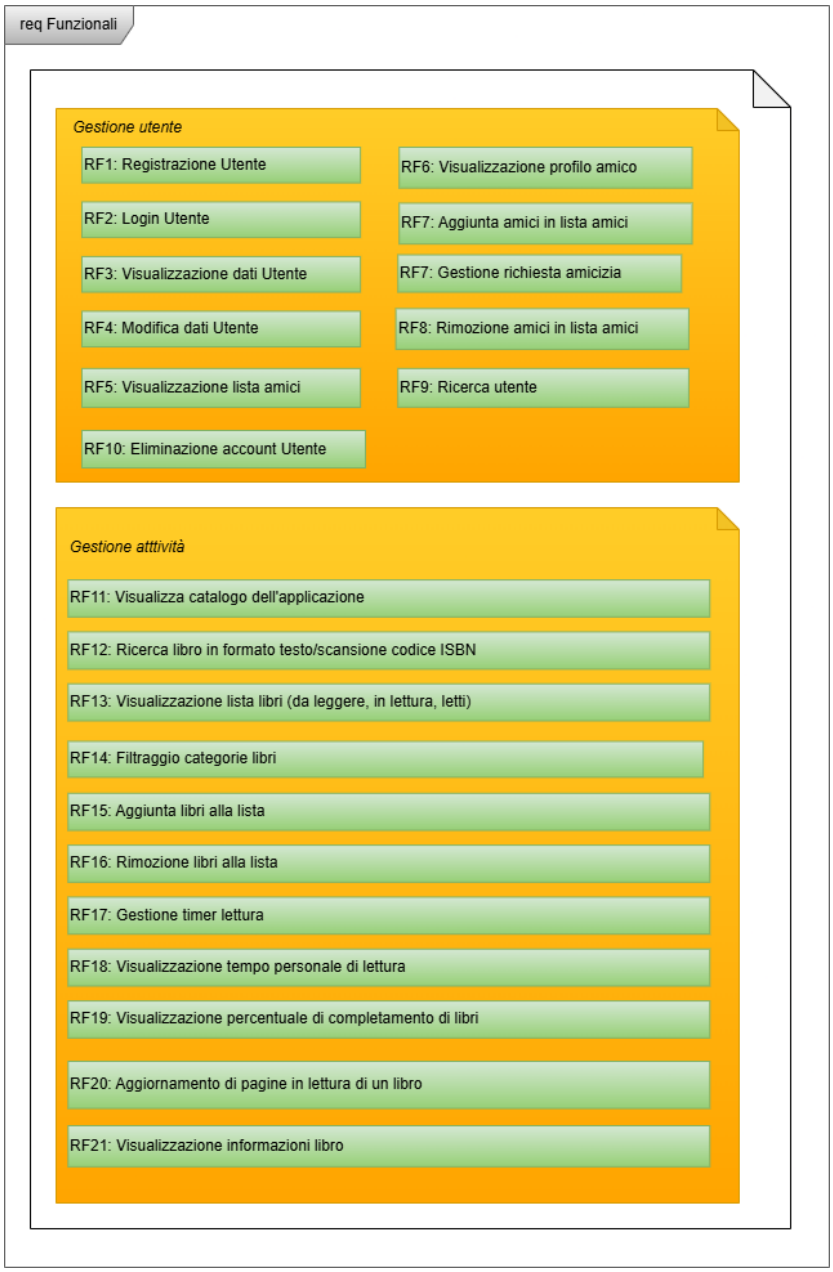

Figure 1: Requisiti funzionali

- *Gestione utente*
  - *RF1: Registrazione utente* - L'applicazione permette di registarsi con le proprie informazioni personali
  - *RF2: Login utente* - L'applicazione permette di autenticare l'utente registrato con le proprie informazioni personali
  - *RF3: Visualizzazione dati utente* - L'applicazione permette di visualizzare i propri dati personali
  - *RF4: Modifica dati utente* - L'applicazione permette di modificare i propri dati personali
  - *RF5: Visualizzazione lista amici* - L'applicazione permette di visualizzare la lista degli amici che hanno accettato la sua richiesta d'amicizia
  - *RF6: Visualizzazione profilo amico* - L'applicazione permette di visualizzare il profilo di un suo amico nella lista
  - *RF7: Aggiunta amici in lista amici* - L'applicazione permette di aggiungere un amico nella sua lista inviando una richiesta
  - *RF8: Gestione richiesta di amicizia* - L'utente può decidere se accettare o rifiutare una richesta di amicizia
  - *RF9: Rimozione amici da lista amici* - L'applicazione permette di rimuovere un amico dalla lista amici
  - *RF10: Ricerca utente* - L'applicazione permette di ricercare un utente digitando il suo username o nome completo
  - *RF11: Eliminazione account utente* - L'applicazione permette all'utente di cancellare il proprio profilo
- *Gestione attività*
  - *RF12: Visualizza catalogo dell'applicazione* - L'applicazione permette di visualizzare il catalogo dei libri distinti in categorie
  - *RF13: Ricerca libro in formato testo o tramite scansione codice a barre* - L'applicazione permette la ricerca del libro digitando testualmente il titolo in una casella di ricerca apposita oppure scansionando il codice a barre di un libro fisico
  - *RF14: Visualizzazione lista libri (da leggere, in lettura, letti)* - L'applicazione permette di visualizzare il contenuto delle liste principali formate dai libri salvati dall'utente
  - *RF15: Filtraggio categorie libri* - L'applicazione permette all'utente di filtrare le categorie dei libri da visualizzare nel catalogo
  - *RF16: Aggiunta libri alla lista* - L'applicazione permette all'utente di aggiungere i libri in una delle apposite liste fornite dall'app
  - *RF17: Rimozione libri dalla lista* - L'applicazione permette la rimozione di libri dalle liste

– *RF18: Gestione timer lettura* - L'applicazione permette di tenere sotto controllo il tempo di lettura di un libro con l'utilizzo di un timer dedicato attivabile e disattivabile dall'utente

– *RF19: Visualizzazione tempo personale di lettura* - L'applicazione permette all'utente di visualizzare per ogni singolo libro il tempo di lettura impiegato con l'utilizzo del timer

– *RF20: Visualizzazione percentuale completamento di libri* - L'applicazione permette di visualizzare la percentuale di completamento di ogni singolo libro, calcolata tramite il numero di pagine lette dall'utente rispetto al numero di pagine totali di un determinato libro

– *RF21: Aggiornamento percentuale di pagine in lettura di un libro* - L'applicazione permette di aggiornare il numero di pagine lette di un libro per calcolare la percentuale di completamento

– *RF22: Visualizzazione informazioni libro* - L'applicazione permette la visualizzazione delle info di un determinato libro

2.1.2 Requisiti non funzionali

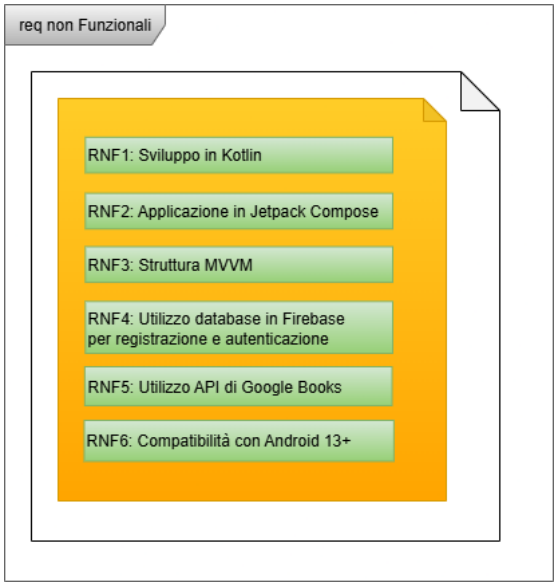

Figure 2: Requisiti non funzionali

– *RNF1: Sviluppo in Kotlin* - L'applicazione è stata sviluppata principalmente con il linguaggio di programmazione Kotlin

– *RNF2: Applicazione in Jetpack Compose* - L'applicazione è stata realizzata con Jetpack Compose per realizzare le interfacce grafiche

- *RNF3: Struttura MVVM* - L'applicazione ha una struttura architetturale basata su Model-View-ViewModel per agevolare la programmazione con sezioni dedicate
- *RNF4: Firebase Firestore per memorizzazione e accesso/autenticazione utente* - L'applicazione fa utilizzo per la memorizzazione di dati e la gestione della registrazione e l'accesso dell'utente di un database tramite la piattaforma Google Firebase
- *RNF5: API di Google Books* - L'applicazione si sincronizza e fa utilizzo delle API Key di Google Books per mostrare i libri nel catalogo e nella ricerca
- *RNF6: compatibilità con Android 13+* - L'applicazione ha una compatibilità con il livello di API 33, permettendone l'utilizzo su android 13 e oltre

2.1.3 Casi d'uso

Si riportano gli attori e i tre casi d'uso principali che descrivono le azioni principali che si possono svolgere nell'applicazione a partire dai requisiti elencati in precedenza.

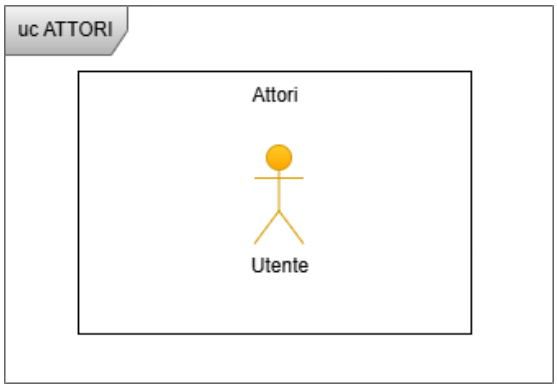

Figure 3: Attori principali

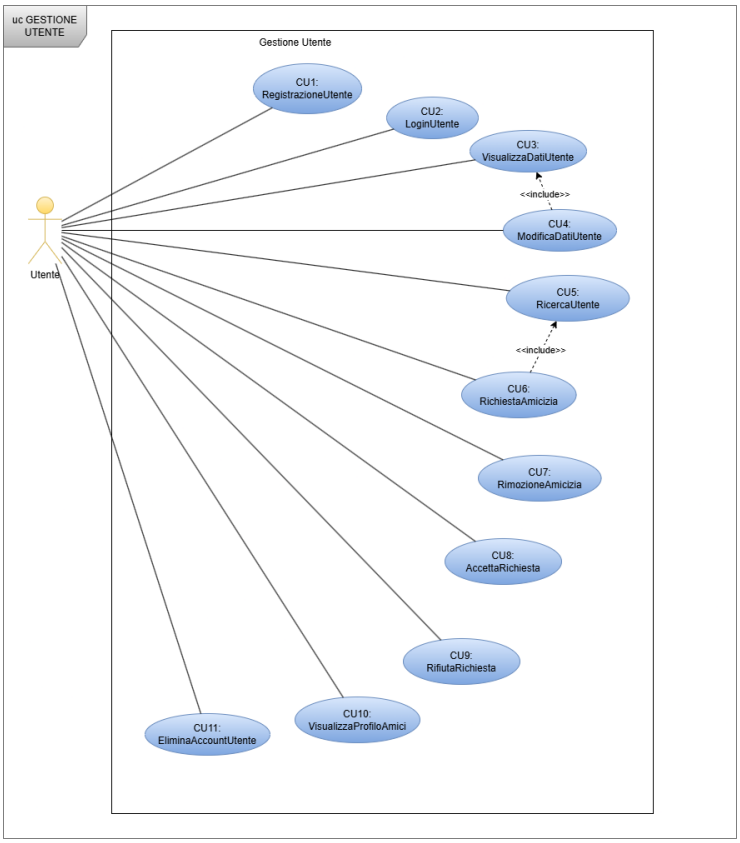

Figure 4: Gestione utente

- *CU1: RegistrazioneUtente* - L'utente, per usufruire dei servizi offerti dall'applicazione, ha bissogno di registarsi con le proprie informazioni personali
- *CU2: LoginUtente* - L'utente può accedere ai servizi forniti inserendo le proprie credenziali (email e password) precedentemente utilizzate per registrarsi
- *CU3: VisualizzaDatiUtente* - L'utente può visualizzare le proprie informazioni personali in una sezione dedicata
- *CU4: ModificaDatiUtente* - L'utente può modificare i propri dati personali
- *CU5: RicercaUtente* - L'utente può ricercare un utente digitando il suo nome utente (usernam) o nome completo (nome e/o cognome)
- *CU6: RichiestaAmicizia* - L'utente può aggiungere amici nella propria lista inviando richieste ad altri utenti

– *CU7: RimozioneAmicizia* - L'utente può rimuovere gli amici dalla propria lista

– *CU8: AccettaRichiesta* - L'utente può accettare la richiesta di amicizia da un altro utente che l'ha inviata

– *CU9: RifiutaRichiesta* - L'utente può rifiutare la richiesta di amicizia da un altro utente che l'ha inviata

– *CU10: VisualizzaProfiloAmici* - L'utente può visualizzare il profilo dei propri amici con le loro statistiche di lettura

– *CU11: EliminaAccountUtente* - L'utente può eliminare il proprio account utente

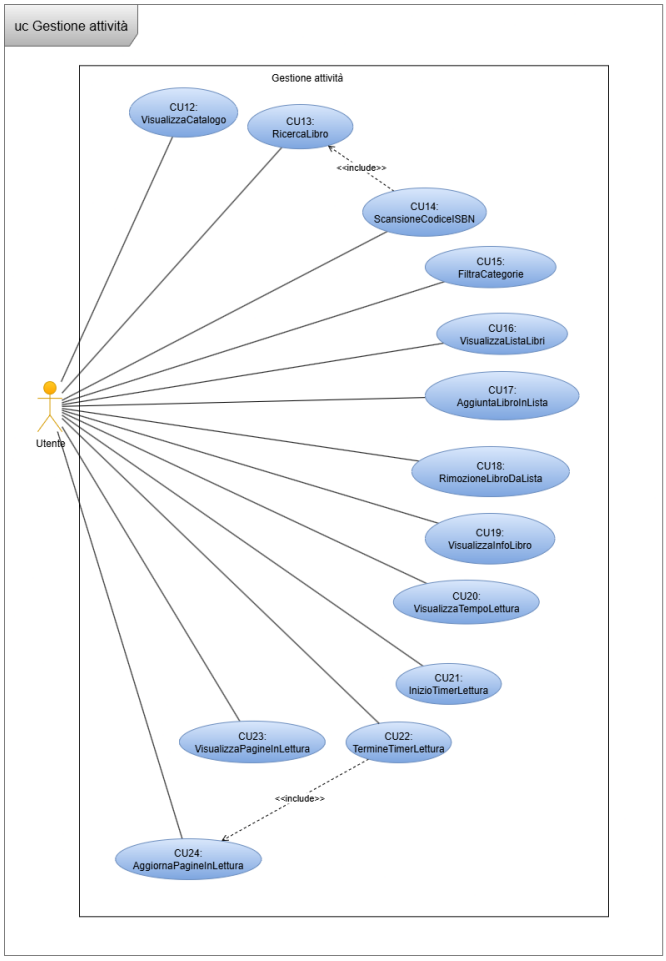

Figure 5: Gestione attività

– *CU12: VisualizzaCatalogo* - L'utente può visualizzare il catalogo di libri separato in categorie

– *CU13: RicercaLibro* - L'utente può ricercare il libro digitando il titolo nella barra di ricerca presente nel catalogo

– *CU14: ScansioneCodiceISBN* - L'utente può effettuare la scansione del codice ISBN di un libro fisico tramite la fotocamera del telefono

– *CU15: FiltroCategorie* - L'utente può filtrare le categorie del catalogo selezionandole da un lista con checkbox

– *CU16: VisualizzaListaLibri* - L'utente può visualizzare una delle tre liste principali (da leggere, in lettura, letti) per memorizzare i libri salvati

– *CU17: AggiuntaLibroInLista* - L'utente può aggiungere un libro selezionato in una delle tre liste principali (da leggere, in lettura, letti)

– *CU18: RimozioneLibroDaLista* - L'utente può rimuovere un libro presente in una delle tre liste principali (da leggere, in lettura, letti)

– *CU19: VisualizzaInfoLibro* - L'utente può visualizzare le informazioni principali che rappresentano il libro selezionandolo dal catalogo o dalle liste

– *CU20: VisualizzaTempoLettura* - L'utente può visualizzare il tempo di lettura di un proprio libro

– *CU21: InizioTimerLettura* - L'utente può iniziare la sessione di lettura di un libro tramite l'inizio di un timer dedicato

– *CU22: TermineTimerLettura* - L'utente può terminare la sessione di lettura di un libro permettendo di fermare il timer

– *CU23: VisualizzaPagineInLettura* - L'utente può visualizzare le pagine in lettura di un libro salvato nell'omonima lista

– *CU24: AggiornaPagineInLettura* - L'utente può aggiornare le pagine

2.2 Architettura applicazione

2.2.1 MVVM

Per la struttura dell'applicazione è stato scelto il pattern architetturale Model-View-ViewModel (MVVM) per ottenere una separazione netta delle responsabilità tale da rendere il codice scalabile e facilmente aggiornabile. Le sue principali componenti sono:

- **Model:** Gestisce i dati dell'applicazione e della loro persistenza, principalmente in remoto con Firebase nel caso specifico di questa applicazione, permettendo operazioni di accesso, modifica e salvataggio dei dati per gli *utenti* che si registrano e dei *libri* che salvano in determinate liste.

- **View:** Rappresenta le interfacce dell'applicazione per permettere all'utente di interagire con la logica dell'applicazione. Vengono definite tramite funzioni con la notazione iniziale *@Composable* e possono essere richiamate nel file contenente il MainActivity o in altre funzioni con la stessa notazione.
- **ViewModel:** Funge da intermediario tra View e Model e permette il cambiamento dello stato della UI con le interazioni dell'utente tramite *LiveData*

2.2.2 Diagramma delle componenti

Il seguente diagramma di componenti rappresenta le principali istanze e le loro interazioni, in particolare tra l'utente e un singolo libro scelto, la lista dei libri e la richiesta di amicizia verso un altro utente. Per il libro l'utente può decidere se inserirlo in una lista o rimuoverlo, mentre per la richiesta l'utente può inviarne una e visualizzarne una ricevuta, per poi decidere se accettarla o rifiutarla.

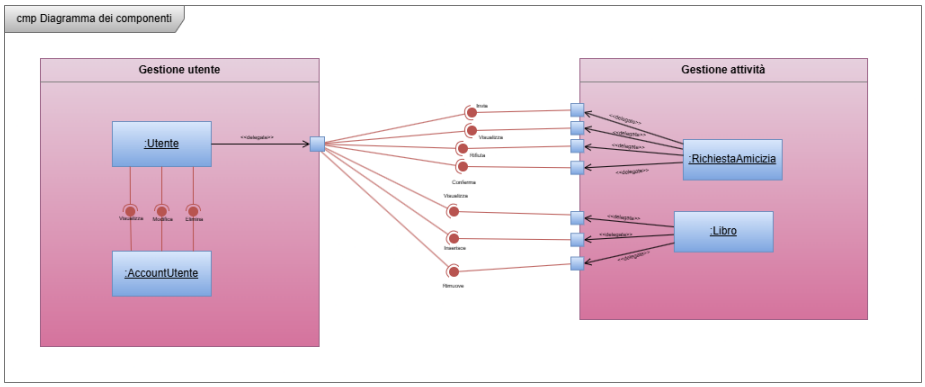

Figure 6: Diagramma delle componenti

2.3 Gestione Database - Firebase

In presenza di un sistema di autenticazione e registrazione a un servizio online, con la possibilità per l'utente di interagire con altri utenti, e permettere il salvataggio personale di libri nel proprio profilo, l'infrastruttura migliore per manipolare questi dati risulta essere il database gestionale Firestore di Google Firebase.

Firestore è un DB NoSQL strutturato a documenti, il quale permette il salvataggio di dati in documenti raggrupati in collezioni e può contenere sottocollezioni e campi annidati. Tali documenti e collezioni vengono generati con la scrittura di dati tramite codice.

Il suo utilizzo si concentra su due componenti della sezione *Creazione*:

- *Authentication:* permette di gestire il metodo di accesso degli utenti tramite diversi servizi. Nel caso di questo progetto abbiamo implementato esclusivamente l'autenticazione semplice tramite email e password.

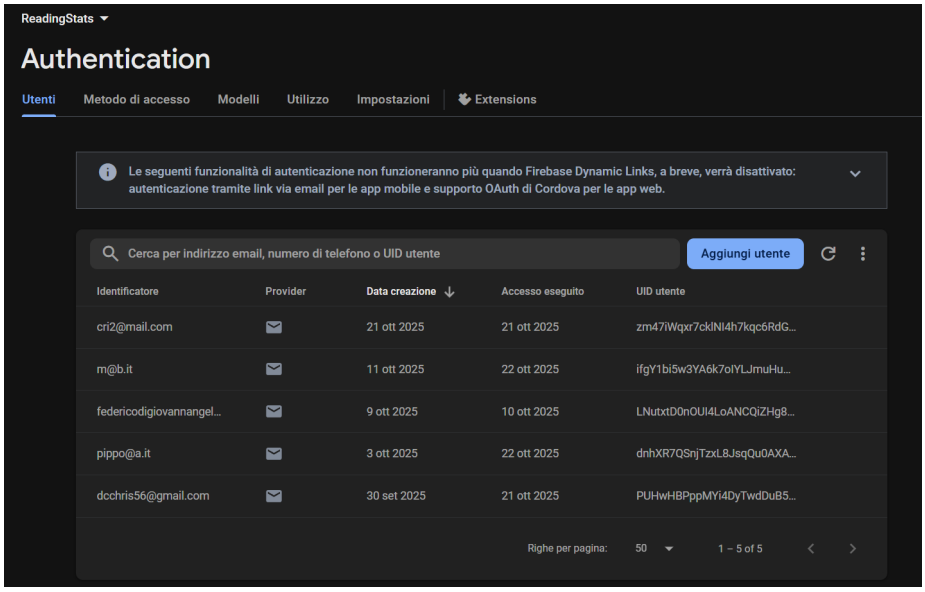

Figure 7: Diagramma delle componenti

- *Firestore Database:* permette la memorizzazione tramite documenti e collezioni descritti in precedenza dei dati dell'utente, delle richieste di amicizia inviate tra utenti e dei libri salvati con ognuno la relativa sessione. Sono state implementate delle regole specifiche per permettere un corretto e coerente accesso ai dati e la loro relativa modifica. Di seguito i documenti e le collezioni generate:

  INSERIRE SCREEN PER OGNI SEZIONE

  - **Salvataggio utenti:** Utenti registrati e salvati nel servizio tramite il form di registrazione, memorizzando nome, cognome, username e mail;
  - **Salvataggio usernames:** Username salvati al momento della registrazione per permettere un controllo di univocità;
  - **Salvataggio di amici tramite richieste:** Lista di utenti che si aggiorna per ogni singolo utente quando accetta una richiesta di amicizia da un altro utente;
  - **Salvataggio di libri dal catalogo:** Raccolta di libri salvati per ogni utente, ognuno con le sue informazioni e lo stato per visualizzarlo in una determinata lista tra quelle fornite;

- **Salvataggio sessione di lettura:** Per ogni libro nello stato "In lettura", viene memorizzata una sessione di lettura contenente il tempo memorizzato in secondi e convertito a schermo in ore:minuti:secondi;
- **Regole di accesso ai documenti definiti:** Le seguenti regole scritte nell'omonima sotto-sezione permettono la definizione di un accesso;

2.4 API utilizzate - Google Books

Le API di Google Books sono state utilizzata per ottenere le informazioni principali dei libri, in modo da permettere all'utente la loro visualizzazione e il conseguente salvataggio. L'API viene fornita tramite un link per le ricerche e una chiave generata da Google Cloud per permetterne l'abilitazione e la limitazione del tipo di dispositivo che la utilizzerà.

Link: https://www.googleapis.com/books/v1/volumes?q=search-terms.

"q" è il parametro da inserire per effettuare la ricerca tramite titolo o autore. Sono presenti altri parametri, i cui principali per l'applicazione sono stati:

- *maxResults*, per mostrare la quantità di libri da visualizzare, utilizzato in particolare nel catalogo e nella ricerca.
- *orderBy*, il quale può essere relevance o newest per mostrare libri di maggiore rilievo tramite parametriche di Google o nuove uscite.
- *langRestrict*, ossia la restrizione dei libri da visualizzare e ricercare con la una lingua specifica.

2.5 Sviluppo

Come anticipato in precedenza, l'applicazione è interamente sviluppata in **Jetpack Compose**, un toolkit dichiarativo di Android che permette, al variare del proprio stato, di ridisegnare le parti necessarie al cambiamento di tale stato.

Permette un approccio dichiarativo tramite la definizione di funzioni con notazione *@Composable*, le quali permettono di definire parti di interfaccia utente esclusivamente con codice Kotlin. In queste funzioni si controlla lo stato per permettere l'aggiornamento di determinate parti coinvolte.

Sono disponibili una serie di layout predefiniti (Column, Row, Box) per strutturare al meglio una componente dell'interfaccia e le funzioni accettano un parametro Modifier che permette di modificarne l'aspetto estetico.

Per la costruzione di un'interfaccia tradizionale è stato utilizzata la funzione *Scaffold* con parametri per definire componenti predefinite come TopAppBar, BottomAppBar, FloatingActionButton e altri ancora.

Si possono distinguere due tipi di funzioni Composable:

- **Stateful**, fa utilizzo della keyword *remember* che permette di memorizzare dati tra diverse recomposition
- **Steless**, ossia un Composable orivo di stato

Le liste scorrevoli che visualizzano molti elementi vengono gestite da componenti *Lazy*, che forniscono un blocco per descrivere i singoli *item* da visualizzare. Ne esistono diverse versioni:

- **LazyRow** per lo scorrimento orizzontale,
- **LazyColumn** per lo scorrimento verticale,
- **LazyVerticalGrid e LazyHorizontalGrid** per lo scorrimento di una griglia rispettivamente verticale e orizzontale

La navigazione viene principalmente gestita tramite un file dedicato, denominato **AppNavHost.kt**, nel quale viene creato un *NavController* e un *NavHost*, il quale fa uso della funzione *composable* che permette di definire le rotte di destinazione all'interno dell'applicazione. Tali rotte vengono definite in formato di stringa in un altro file denominato **Routes.kt**

Le funzioni con notazione *@Composable* possono essere richiamate da un Activity o Fragment, da altre funzioni *@Composable*, all'interno di una destinazione definita in Navigation-Compose e in liste Lazy.

2.5.1 Struttura package

Il progetto, strutturato con architettura MVVM, è stato suddiviso in macro-package al cui interno sono presenti file o altri package caratteristici.

La struttura delle macro-cartelle risulta la seguente:

- **core:** contiene i package con i file di parti di interfaccia *@Composable* riutilizzabili nel codice e i file utilizzati per la personalizzazione globale dell'app.
- **di:** moduli di *Dependency Injection* (Hilt) che descrivono come creare e fornire le dipendenze dell'app. Le classi ricevono gli oggetti necessari tramite iniezione, senza istanziarli al proprio interno. Contiene i file usati a livello globale per integrare le Google Books API e configurare lo stack di rete.
- **features:** è presente il codice principale dell'applicazione scritto in diversi file distinti e gestito in package annidati per la loro funzionalità nell'applicazione.
- **navigation:** contiene i file di gestione della navigazione e di creazione delle route.

2.5.2 File di configurazione

Sono stati modificati i file di configurazione per l'importazione di diverse dipendenze nel progetto. I principali file sono:

- **AndroidManifest.xml:** Utilizzato per la dichiarazione di diversi componenti e permessi per l'app sviluppata su Android che riconosce in runtime. Il principale permesso applicato è stato INTERNET per permettere la richiesta via rete di libri tramite le API fornite.
- **build.gradle.kts (app):** Definisce la modalità di compilazione del modulo *app* e quali librerie collegare per il loro utilizzo. A differenze di AndroidManifest, viene valutato a build-time.
- **build.gradle.kts (root):** Permette l'applicazione di configurazioni e impostazioni condivise a tutti i moduli.

2.6 Spiegazione codice in dettaglio

Di seguito, viene suddivisa in sezioni la spiegazione nel dettaglio dello scopo dei file con le proprie componenti utilizzate.

2.6.1 core: Componenti Compose riutilizzabili

In questo package troviamo componenti *@Compose* riutilizzabili nel codice, elencati e descritti di seguito:

- **AppScaffold.kt:** Struttura principale del componente Scaffold dell'applicazione, con parametri *topBar* e *bottomBar* assegnati in **AppNavHost.kt**. Permette la definizione della struttura principale dell'applicazione.
- **HeaderComponent.kt:** Struttura dell'intestazione dell'app, costruita con *TopAppBar*, contenente il logo dell'applicazione e il titolo "Reading Stats".
- **NavBarComponent.kt:** Struttura della barra di navigazione per permettere di navigare tra le quattro sezioni principali dell'applicazione: **Scaffali, Catalogo, Home e Profilo**. Al richiamo con il parametro assegnato di tipo *NavController*, viene assegnata poi al parametro bottomBar di AppScaffold.
- **SerchBar.kt:** Barra di ricerca costruita e utilizzata nella schermata del **Catalogo** e nella sezione *Amici* del **Profilo**, rispettivamente utilizzata per la ricerca di libri o per la ricerca di amici registrati all'applicazione.
- **VerticalScrollBar.kt:** Barra di scorrimento verticale richiamata nella finestra di dialogo per la selezione dei filtri del **Catalogo**.

2.6.2 di: Dependency Injection

Tale package racchiude i moduli **Hilt** che dichiarano come costruire le dipendenze dell'app a runtime, come *Firebase, Retrofit e OkHttp*. Permette quindi una configurazione centralizzata e una migliore testabiità dell'intera app.

- **AuthModule.kt:** permette la definizione e l'iniziezione nei ViewModel della catena **UseCase → Repository → DataSource → Firebase**. Fornisce servizi di base per l'autenticazione e l'estrazione di dati da database Firestore. Sono presenti degli use case per il login, la registrazione e il controllo dello username. Fa uso di due annotazioni:
  - *@Singleton:* Annotazione di scope per mantenere una sola istanza del binding per il tempo di vita del componente su cui è installato.
  - *@InstallIn:* Annotazione di installazione di modulo il quale comunica con Hilt a quale componente deve essere unito il modulo (con annotazione *@Module*) che contiene le notazioni *@Provides o @Bind*. Definisce quindi il raggio d'azione del modulo e la visibilità dei binding.
- **NetworkModule.kt:** Permette la configurazione di rete dell'applicazione e la possibilità di richiedere le informazioni dei libri tramite l'url fornito dalle API di Google Books.
- **KeysModule.kt:** Definisce un qualifier *@BooksApiKey* per evitare collisioni tra generiche String e la chiave fornita da Google Books quando viene iniettata.

2.6.3 Package interni a features

Di seguito vengono elencati e descritti lo scopo dei singoli package contenuti in **features**, utilizzati per distinguere le diverse sezioni dell'applicazione. Ognuno di loro contiene a sua volta dei package per l'applicazione della struttura con MVVM, quindi con schermate definite con funzioni *@Composable* e model per rappresentare classi i cui oggetti saranno memorizzati nel database Firestore.

- **auth:** contiene ulteriori package con file per la corretta gestione dell'accesso e della registrazione dell'utente tramite Firestore.
- **bookdetail:** contiene i file di schermata delle info del libro selezionato dal catalogo o dalla lista dei libri salvati nel proprio scaffale, con un view model dedicato alla gestione del salvataggio nelle liste e all'aggiornamento delle pagine lette.
- **catalog:** contiene ulteriori package con file per la visualizzazione e gestione del catalogo di libri, richiamando grazie al proprio view-model la possibilità di effettuare query per l'estrazione di libri.
- **home:** contiene ulteriori package con file per la sezione home, la quale permette di visualizzare e interagire con i libri salvati nella lista "In lettura" in modo da permettere l'avvio di un timer dedicato per ciascuno di loro.

- **profile:** contiene ulteriori package con file per la gestione del profilo utente, nel quale è possibile visualizzare le proprie informazioni e visualizzare le informazioni relative agli utenti amici.

- **shelves:** contiene ulteriori package con file per gestione della sezione "scaffali", contenente le liste di libri salvati dall'utente distinte in tre categorie: *da leggere, in lettura e letti*

2.6.4 File di sorgente per gestione autenticazione

I seguenti file sono utilizzati per la gestione dell'accesso e della registrazione dell'utente nel database Firestore.

- **FirebaseAuthDataSource.kt:** Sorgente dati che incapsula *FirebaseAuth* per operazioni di autenticazione. Espone funzioni sospese *createUser(email, password)* e *signIn(email, password)* (ritornano *AuthResult* usando *await()*), oltre a metodi sincroni *signOut()* e *currentUid()* per terminare la sessione e ottenere l'UID corrente.

- **FirestoreUserDataSource.kt:** Sorgente dati che incapsula *Firebase-Firestore* per la gestione dei profili utente, usando le collezioni *users* e *usernames*. Fornisce *isUsernameTaken(username)* e *userExists(uid)*; *createUserAtomically(uid, profile)* esegue una transazione che verifica collisioni sullo username e registra in modo coerente sia il documento utente sia il mapping username $\rightarrow$ uid (merge con *SetOptions*). *getUserProfile(uid)* legge e mappa il documento in *UserModelDto*. *updateUserProfile(uid, username, name, surname, email)* aggiorna i campi (incluso *updatedAt*) e, in transazione, se lo username cambia rimuove il vecchio mapping in *usernames*, verifica che il nuovo non sia occupato e crea il nuovo mapping prima di aggiornare il documento in *users*; eventuali eccezioni sono rilanciate per la gestione a livello ViewModel.

2.6.5 Model: file principali

I Model dell'applicazione vengono qui descritti distinguendo i package interni di **features**, i quali sono:

- **auth:**
  - **data.model.UserModelDto.kt:** descrive le componenti principali dell'utente che si registra e fa uso del servizio. Viene denotato con *DTO* per indicare Data Transfer Object, ossia una classe contenitore utilizzata per trasportare informazioni tra diversi processi. In questo caso viene utilizzata per la gestione delle informazioni da salvare per gli utenti nel database Firestore.
  - **domain.model:**

    - **LoginFormState.kt:** File di rappresentazione dello stato per il login dell'utente che accede, contenente quindi le informazioni essenziali per il suo accesso e la conseguente verifica.
    - **RegisterFormState.kt:** Rispetto a LoginState, questo file contiene più informazioni necessarie per la memorizzazione per la memorizzazione dell'utente nel servizio.

- **catalog:**
  - **data.dto.GoogleBooksDto.kt:** contiene determinati DTO per Google Books, utilizzati per la loro ricerca e l'estrazione delle informazioni principali.
  - **domain.model.Book.kt:** Model principale del libro, utilizzato per definire e raccogliere le informazioni estratte e richiamarle da un oggetto istanziato.

- **home:**
  - **domain.model.UiHomeBook.kt:** Model del libro per la sezione home, contenente le informazioni principali del libro, il numero di pagine lette registrate dall'utente (attributo *pageInReading*) e il totale dei secondi di lettura (attributo *totalReadSeconds*).
  - **domain.model.HomeItemState.kt:** Stato del relativo UiHomeBook per tenere conto dell'inizio e della fine della sessione di lettura avviata, rispettivamente memorizzati negli attributi *sessionStartMillis* e *sessionElapsedSec*. Restituisce il totale dei secondi di lettura tra più sessioni memorizzando in *totalReadSec* i secondi delle sessioni precedenti che vengono sommati a *sessionElapsedSec*. Tale valore viene memorizzato in una funzione get() denominato *totalWithSession*.
  - **domain.model.PagesDialogState.kt:** Stato della finestra di dialogo utilizzato per aggiornare le pagine lette al termine di una sessione di un determinato libro in lettura.
  - **domain.model.HomeUiState.kt:** Stato rappresentante la lista di libri in lettura visualizzabile nella home.

- **profile:**
  - **data.model.FriendModels.kt:** Model relativo all'amico e alla richiesta di amicizia, con relativo stato per definire se un utente è amico o meno di un altro o se è in attesa una richiesta inviata.

- **shelves:**
  - **data.dto.UserBookDto.kt:** Model rappresentante il libro salvato nelle liste negli scaffali, con attributo aggiuntivo *status* per memorizzare lo stato di lettura e il salvataggio nella relativa lista. Viene aggiunta la denominazione DTO per la memorizzazione in Firestore.

- **domain.model:**
  - **ReadingStatus.kt:** Enum rappresentante i tre stati di lettura principali: **TO-READ, READING, READ**
  - **UserBook.kt:** Model da cui UserBookDto si adatta per il salvataggio dei dati.

2.6.6 View: funzioni Composable per l'interfaccia

I seguenti file descritti rappresentano le view dell'applicazione definite in Jetpack Compose, con caratterictiche distintive per ognuno di esse.

- **auth:**
  - **ui.components:**
    - **LoginScreen.kt:** Schermata iniziale necessaria per il login dell'utente registrato. E' presente un *TextButton* per il passaggio alla schermata di registrazione. Fino a quando l'utente non inserisce una mail con il proprio formato e una password da almeno 6 caratteri, non è possibile cliccare il pulsante *Button* per accedere.
    - **RegistrationScreen.kt:** Schermata di registrazione con i campi principali da compilare: nome, cognome, username, email, password e relativa conferma. Come per il login, qui è presente un *TextButton* per il passaggio alla schermata di login. Se tutti i campi non sono compilati correttamente non è possibile registrarsi in quanto il *Button* rimane bloccato. Vengono visualizzati dei messaggi di errore nel caso in cui l'utente inserisce un username utilizzato da un altro utente o quando si registra con una mail presente nel database.
- **profile:**
  - **ui.components:**
    - **ProfileScreen.kt:**
    - **DatiPersonali.kt:**
    - **ModificaDatiPersonali.kt:**
    - **ListaAmici.kt:**
    - **FriendComponents.kt:**
    - **InfoSupporto.kt:**
- **catalog:**
  - **ui.components:**
    - **BookCard.kt:** Componente che rappresenta un libro selezionabile con immagine di copertina renderizzata via *AsyncImage* (modello costruito dalla *thumbnail* del *Book*, con *fallback*/*error* e *crossfade*); sotto è mostrato il titolo troncato su più righe.

L'intera card è cliccabile e invoca *onClick(Book)* per navigare, ad esempio, alla schermata **BookDetailScreen.kt**.

    - **CategoryRow.kt:** Componente che visualizza una riga orizzontale di libri per una categoria. Accetta *title*, *books* e *onBookClick*; mostra l'intestazione della categoria, un placeholder quando la lista è vuota e la lista scorrevole realizzata con *LazyRow* (incapsulata in *Surface* arrotondata), componendo **BookCard.kt** per ciascun elemento e propagando *onBookClick*.
    - **CatalogScreen.kt:** Schermata principale del catalogo che include una *SearchBar* (ricerca istantanea con debounce e ricerca esplicita), una barra *Filtri* per aprire *FiltersDialog* (selezione multipla con checkbox, azzera/conferma), e un corpo che alterna: una griglia *LazyVerticalGrid* a tre colonne per i risultati di ricerca (con indicatore *CircularProgressIndicator* durante il caricamento) oppure una colonna *LazyColumn* di righe categoria composte da **CategoryRow.kt** quando non ci sono risultati. Alla conferma dei filtri vengono mostrate solo le categorie selezionate.

- **bookdetail:**
  - **ui.components:**
    - **BookDetailScreen.kt:** Schermata che mostra i dettagli del libro (copertina con *AsyncImage*, titolo, autori, data, conteggio pagine) con *TopAppBar* e gestione del tasto indietro via *BackHandler*. Integra *BookStatusBar* con tre *IconButton* a selezione esclusiva per impostare lo stato (*Da leggere*, *In lettura*, *Letto*); al click, per *In lettura* avvia un flusso di dialoghi (gestito da *ReadingFlowDialogs* nello stesso file) che chiede eventualmente le pagine totali e poi le pagine lette, mentre per *Da leggere*/*Letto* richiede solo le pagine totali se mancanti, altrimenti applica direttamente lo stato. Usa *SnackbarHost* per notifiche evento e *ExpandableText* per la descrizione, e collega il ViewModel al volume tramite *bindVolume*.
    - **ReadingDialogs.kt:** Descrive due funzioni *@Composable* per finestre di dialogo da richiamare al click di una delle tre componenti *IconButton* in **BookDetailScreen.kt**:
      - *TotalPagesDialog:* Richiamata quando l'attributo *pageCount* è null e quindi non è presente tale valore di riferimento per aggiornare le pagine in lettura del libro selezionato. Permette di impostare il valore numerico in un componente *OutlinedTextField* e confermare o annullare la digitazione con componenti *Button* dedicati.
      - *ReadPagesDialog:* Richiamata per aggiornare le pagine in lettura del libro selezionato.

- **shelves:**
  - **ui.components:**
    - **ShelvesScreen.kt:** Contiene funzioni *@Composable* per la definizione della schermata della sezione "Scaffali", caratterizzata dalla struttura principale definita dalla funzione *ShelvesScreen*, strutturata con un componente *Scaffold*. Questa contiene una lista in colonna di componenti *Card* al cui interno viene richiamata la funzione *ShelfRow* che rappresenta uno scaffale, ossia una lista di libri salvati dal catalogo; la schermata include inoltre un *FloatingActionButton* per l'avvio dello scanner ISBN (*GmsBarcodeScanning*), un *SnackbarHost* per i messaggi, semantiche di accessibilità (*contentDescription*) e la gestione dell'abilitazione dei tap tramite *rememberCanInteract* in base al *Lifecycle*.
    - **SelectedShelfScreen.kt:** Contiene la struttura, costruita in *Scaffold*, della lista di libri selezionata a partire dalle tre liste principali, con *TopAppBar* (icona di ritorno) e dati osservati dal *ShelvesViewModel* tramite *collectAsStateWithLifecycle*. In mancanza di libri salvati, si visualizza in un *Box* centrale il testo "Nessun libro presente in lista". In presenza di libri è possibile visualizzare in ciascun libro anche un indicatore circolare del progresso di lettura, il quale permette di far visualizzare all'utente la percentuale di lettura totale. Ogni riga, definita con la funzione *ShelfBookRow*, contiene l'immagine del libro (thumb con fallback in preview, *BookThumb*) e l'indicatore *ReadingProgressCircle*, renderizzati all'interno di una *LazyColumn*.
- **home:**
  - **ui.components:**
    - **HomeScreen.kt:** Contiene funzioni *@Composable* per la schermata principale *Home*, focalizzata sui libri in lettura e sulla gestione delle sessioni. L'istanza di *HomeViewModel* è ottenuta via Hilt e lo stato è osservato con *collectAsState()*; se non ci sono elementi viene mostrato un placeholder centrale. L'elenco è reso con *LazyColumn* che compone card (*ReadingCard*) con copertina (Coil *AsyncImage* con fallback/errore), titolo, indicatore grafico del progresso (*ReadingProgressCircle*) e tempo totale formattato (*formatSeconds*). Ogni card espone un pulsante contestuale "Riprendi/Termina" che invoca *viewModel.onStart(book)* o *viewModel.onStop(book)*; è garantita un'unica sessione attiva alla volta (*anyRunning*) e, quando in corso, è mostrato anche il timer della sessione. Al termine, se previsto, viene aperta la dialog *PagesDialog* per aggiornare le pagine lette, delegando la conferma a *viewModel.confirmPages(pages)*.

* **ReadingDialogs.kt:** Definisce il composable *PagesDialog* per inserire/aggiornare le *pagine lette totali* alla chiusura di una sessione. L'interfaccia è una *BasicAlertDialog* con *Surface* sagomata, titolo dinamico (mostra l'intervallo consentito (0..*pageCount*) se disponibile), campo numerico con filtro solo cifre e validazioni (non vuoto, valore valido, ≥ 0 e $\leq$ *pageCount* se noto); lo stato dell'input è indicizzato su *book.id* per mantenere correttamente il valore al cambio libro. I pulsanti "Annulla/Salva" invocano rispettivamente *onDismiss* e *onConfirm(Int)*; l'integrazione con *HomeScreen* consente di persistere l'aggiornamento tramite *HomeViewModel*.

2.6.7 ViewModel: file di definizione

- **auth:**
  - **AuthViewModel.kt:** ViewModel di autenticazione annotato *@HiltViewModel* che inietta i casi d'uso *CheckUsernameAvailableUseCase*, *RegisterUserUseCase* e *LoginUserUseCase*. Gestisce due flussi di stato (*StateFlow*): *RegistrationFormState* per la registrazione e *LoginFormState* per il login, esponendo metodi di aggiornamento dei campi (*onNameChange*, *onSurnameChange*, *onEmailChange*, *onPasswordChange*, *onConfirmPasswordChange*, *onUsernameChange*, *onLoginEmailChange*, *onLoginPasswordChange*). La verifica disponibilità dell'username è *debounced* (job cancellabile con *delay(250)* in *viewModelScope*), con normalizzazione (*trim().lowercase()*) e aggiornamento di *usernameAvailable*. Le azioni *submitRegister()* e *submitLogin()* rispettano i guard (*canSubmit*, *isSubmitting*) e, in coroutine, chiamano i rispettivi use case, mappando gli esiti (*Success*/*Error*) in flag di successo e messaggi utente localizzati (es. "Email già registrata.", "Email o Password non corretti.", "Problema di rete, riprova.", "Configurazione Firebase non valida."). L'UI può osservare i due *StateFlow* con *collectAsState()* per abilitare/disabilitare i pulsanti, mostrare errori e reagire al completamento del flusso di autenticazione.
- **catalog:**
  - **CatalogViewModel.kt:** ViewModel del catalogo annotato *@HiltViewModel* che inietta *SearchBookUseCase*, *GetCategoryFeedUseCase*, *UserPreferencesRepository* e *GetCurrentUidUseCase*. Gestisce lo stato principale (*UiState*) con query di ricerca, stato di caricamento, risultati, elenco delle categorie disponibili e selezionate, righe di categoria (*CategoryRowState* { *category, books, isLoading, error* }), visibilità del dialog dei filtri e *currentUid*. All'avvio recupera l'utente corrente, carica le categorie selezionate salvate e popola ogni riga tramite *getCategoryFeed* (con fallback alla pagina successiva se vuota). Espone azioni per ricerca "istantanea" con debounce (*performLiveSearch*),

ricerca esplicita (*performSearch*), pulizia query (*clearSearch*), apertura/chiusura del dialog filtri e gestione del set di categorie (*toggleCategory*, *clearFilters*, *confirmFilters* con persistenza su *UserPreferencesRepository* e ricarica delle righe).

- **bookdetail:**
  - **BookDetailViewModel.kt:** ViewModel dei dettagli libro annotato *@HiltViewModel* che riceve da DI: *SavedStateHandle*, *ObserveBookStatusUseCase*, *ObserveUserBookUseCase*, *SetBookStatusUseCase*, *RemoveBookFromShelfUseCase*, *SetPageCountUseCase* e *UpsertStatusBookUseCase*. Estrae e mantiene il *volumeId* (anche tramite *bindVolume*) e pubblica tre *StateFlow*: *status* (stato di lettura corrente), *savedReadPages* (pagine lette) e *savedTotalPages* (totale pagine), ottenuti osservando il repository utente; espone inoltre un *SharedFlow* di *events* per messaggi UI. Fornisce operazioni atomiche di aggiornamento: *setStatusWithPages* (scrive stato, pagine lette e totali in un'unica chiamata), *onStatusIconClick* (toggle/trasferimento tra liste o rimozione dallo scaffale) e *updatePageCount* (aggiorna il totale pagine).

- **home:**
  - **HomeViewModel.kt:** ViewModel della schermata *Home* annotato *@HiltViewModel* che inietta *HomeRepository*, *StartBookTimerUseCase* e *SetBookTimerUseCase*. Mantiene due stati interni mappa (*MutableStateFlow*) per i timer: *running* (bookId → istante di avvio) e *ticking* (bookId → secondi trascorsi), più lo stato della dialog pagine (*PagesDialogState*). Espone *uiState* come *StateFlow¡HomeUiState¿* ottenuto con *combine* tra i libri in lettura (*repo.observeReadingBooks()*), i timer e la dialog, mappando ciascun libro in *HomeItemState* (inclusi *isRunning*, *sessionElapsedSec* e *totalReadSec*). L'azione *onStart* avvia una singola sessione alla volta (guardando *running*), chiama *startTimer* per ottenere l'istante di avvio e attiva un ticker a 1 Hz che aggiorna *ticking*; *onStop* ferma subito la sessione in UI, apre la *PagesDialog* e salva in background l'intervallo con *setTimer*. Con *confirmPages* aggiorna le pagine lette (*repo.updatePagesRead*) e, se raggiunge il totale, imposta lo stato *READ* tramite *repo.setStatus* passando il *UserBook* completo; *closeDialog* chiude la dialog. Il ticker viene disattivato automaticamente quando non ci sono sessioni attive.

- **profile:**
  - **ProfileViewModel.kt:** ViewModel della sezione profilo che gestisce caricamento e aggiornamento dei dati utente su Firestore.

    All'avvio carica il profilo dell'utente corrente (ricavato da *FirebaseAuth*) tramite *FirestoreUserDataSource.getUserProfile* e lo espone come *StateFlow<UserModelDto?>* (*user*); gestisce flag di caricamento interni

(*_loading*, *updateLoading*) e un esito di aggiornamento (*updateResult* con valori "success", "error_no_user", "error_update_failed"). La funzione *updateUserProfile(username, name, surname, email)* invia le modifiche al datasource (*updateUserProfile*), aggiorna lo stato locale in caso di successo e popola *updateResult*; *clearUpdateResult()* azzera l'esito. Tutte le operazioni avvengono in *viewModelScope* con coroutine.

- **shelves:**
  - **ShelvesViewModel.kt:** ViewModel della sezione "Scaffali" annotato *@HiltViewModel*. Ricava lo stato richiesto dagli argomenti di navigazione tramite *SavedStateHandle* (*shelfStatus*) e lo mappa in un titolo (*Da leggere*, *In lettura*, *Letti*); in caso di valore non valido usa *READ* come default.

    Osserva i libri per stato con *ObserveBooksByStatusUseCase*, trasformandoli in *UiShelfBook* ed esponendoli come *StateFlow<List<UiShelfBook>>* con *stateIn* e *SharingStarted.WhileSubscribed(5000)* (valore iniziale lista vuota). Fornisce inoltre *findBookByScan(raw)*: estrae le sole cifre, costruisce i candidati ISBN (EAN-13 *978/979* e, se *978*, calcola anche l'ISBN-10), prova la ricerca mirata *isbn:<codice>* con *SearchBookUseCase* e, se non è stato trovato nulla, effettua una ricerca libera sul testo grezzo; l'operazione gira su *Dispatchers.IO*.

2.6.8 Repository

I seguenti file implementano il *Repository pattern*, fungendo da strato di astrazione tra *UseCase*/ViewModel e le sorgenti dati concrete (rete e backend). Espongono API di dominio stabili e testabili, nascondendo dettagli tecnologici (Firebase Auth/Firestore, Retrofit verso Google Books) e curando:

- La trasformazione dei DTO remoti in modelli di dominio, inclusa la normalizzazione dei dati (es. estrazione ISBN, forzatura *https* per le immagini);
- La gestione asincrona tramite coroutine e *Flow<T>* per stream in tempo reale (osservazione libri per stato, aggiornamenti profilo, ecc.);
- La coerenza su Firestore con scritture atomiche/transaction o *batch write*, merge controllati (*SetOptions.merge*) e validazioni applicative (vincoli su pagine totali/lette, unicità dello *username*);
- La mappatura degli errori provider–specific in esiti di dominio (sealed result *Success/Error*) così da semplificare la logica di UI. In sintesi, i repository centralizzano l'accesso ai dati (autenticazione, catalogo libri, scaffali e sessioni di lettura), applicano regole funzionali e offrono un contratto pulito e indipendente dall'infrastruttura ai layer superiori.

Di seguito la distinzione per package:

- **auth:**
  - **AuthRepository.kt/AuthRepositoryImpl.kt:** Definisce il contratto e l'implementazione del livello di dominio per l'autenticazione. L'interfaccia espone *isUsernameAvailable*, *register*, *login* e *getCurrentUserProfile*, con risultati modellati come sealed result *RegisterResult* e *LoginResult* (varianti *Success*/*Error* con codice e messaggio). L'implementazione usa *FirebaseAuthDataSource* e *FirestoreUserDataSource*: verifica la disponibilità dello username (negando *isUsernameTaken*); in *register* crea l'utente su Auth, costruisce *UserModelDto* e salva in modo atomico profilo e mapping username → uid, mappando gli errori più comuni (es. *EMAIL_IN_USE*, *USERNAME_TAKEN*, *NETWORK*, *AUTH_ERROR*); in *login* effettua il sign-in, controlla l'esistenza del profilo su Firestore (altrimenti esegue sign-out), e normalizza gli errori frequenti (*USER_NOT_FOUND*, *USER_DISABLED*, *INVALID_EMAIL*, *WRONG_PASSWORD*, *INVALID_CREDENTIALS*, *TOO_MANY_REQUESTS*, *NETWORK*). Tutte le operazioni I/O girano su dispatcher dedicato e ritornano esiti strutturati anziché propagare eccezioni ai layer superiori.
- **catalog:**
  - **CatalogRepository.kt/CatalogRepositoryImpl.kt:** Definisce il contratto del catalogo (*search(query, page, pageSize=21)* e *byCategory(category, page, pageSize=18)*) e la relativa implementazione *@Singleton* basata su *GoogleBooksApi*. L'implementazione normalizza le query con *normalizeQuery*: lascia passare direttamente forme tipo *isbn:...*; in caso di testo che contiene esattamente 10 o 13 cifre (per 13 cifre richiede prefisso 978/979) costruisce una query ibrida *isbn:digits OR raw*, altrimenti usa il testo così com'è. I risultati JSON sono mappati a dominio con *VolumeItem.toDomain()*, che popola *Book* estraendo autore, categorie, data, descrizione, pagina totale e thumbnail (via *bestThumbnailHttps*, che sceglie *thumbnail* o *smallThumbnail* e forza *https*). Gli ISBN sono derivati da *VolumeInfo.extractIsbns()* (per *ISBN_13* filtra solo cifre e valida lunghezza 13; per *ISBN_10* rimuove i trattini, porta in maiuscolo, accetta cifre o *X* e valida lunghezza 10). La ricerca per categoria usa *byCategory*: prova *subject:categoria* in italiano, se vuoto tenta una traduzione interna (mappa *it* → *en*, es. "Avventura" → "Adventure"), e in ultima istanza ripete *subject:category*; in tutti i casi imposta *orderBy=relevance* e *projection=full*, calcolando la paginazione con *startIndex = page * pageSize*.
- **home:**
  - **HomeRepository.kt/HomeRepositoryImpl.kt:** Definisce il contratto del dominio *Home* (*observeReadingBooks*, *addReadingSession*,

updatePagesRead, *setStatus*) e la relativa implementazione *@Singleton* su Firestore/Firebase Auth. *observeReadingBooks()* interroga *users/{uid}/books* filtrando *status=READING*, ordinando per *updatedAt* decrescente e, tramite *callbackFlow*, mappa in tempo reale i documenti in *UiHomeBook*. *addReadingSession(bookId, startMillis, endMillis)* valida i tempi, calcola la durata in secondi (≥ 1) e salva in batch una nuova sessione in *books/{bookId}/sessions* aggiornando anche *totalReadSeconds* con *FieldValue.increment* e *updatedAt*. *updatePagesRead* esegue un *merge* delle chiavi *pageInReading* e *updatedAt*. *setStatus* aggiorna *status*, *updatedAt* e *volumeId*, fondendo opzionalmente dal *payload* i metadati disponibili (titolo, thumbnail, autori, categorie, ISBN, *pageCount* se > 0); tutte le operazioni sono *suspend* con *await()*.

- **shelves:**
  - **ShelvesRepository.kt/ShelvesRepositoryImpl.kt:** Definisce il contratto degli scaffali (*observeBooks*, *observeBookStatus*, *observeUserBook*, *setStatus*, *removeBook*, *setPageCount*, *setPageInReading*, *upsertStatusBook*) e l'implementazione su Firestore/Firebase Auth. La collezione utente è *users/{uid}/books*. *observeBooks(status)* espone aggiornamenti realtime via *callbackFlow* filtrando *status* e ordinando per *updatedAt* decrescente, mappando i documenti in *UserBook* con il mapper *toUserBook* (tipo *Flow<List<UserBook>>*). *observeBookStatus(id)* ascolta il singolo documento e restituisce lo stato come enum oppure *null* se assente/invalidato; *observeUserBook(id)* restituisce l'intero *UserBook* o *null*. *setStatus(id, payload, status)* esegue un merge che imposta *status*, *updatedAt*, *volumeId* e aggiunge metadati dal *payload* solo se mancanti (titolo, thumbnail, autori, categorie, ISBN); *pageCount* è integrato solo se quello esistente è assente o ≤ 0 e il *payload* porta un valore > 0. *removeBook(id)* elimina il documento. *setPageCount(id, n)* valida $n>0$ e salva *pageCount* con *updatedAt*; *setPageInReading(id, n)* valida $n \geq 0$ e, se presente un totale, impone $n \leq$ *pageCount*, quindi salva con merge. *upsertStatusBook(userBook, payload, status, pageCount, pageInReading)* effettua un'unica patch atomica: imposta *status*/*updatedAt*/*volumeId*, fonde i metadati mancanti, decide *pageCount* (priorità al parametro se $>$ 0, altrimenti dal *payload* se l'esistente ≤ 0) e scrive *pageInReading* se fornito. Tutte le operazioni sono *suspend* con *await()* e usano *SetOptions.merge()* per preservare i campi esistenti.

2.6.9 UseCases

Gli *Use Case* incapsulano singole operazioni di dominio (interazioni dell'utente o regole applicative) offrendo un'API coerente ai ViewModel. Sono oggetti leggeri, tipicamente *stateless*, iniettati con **Hilt** e invocati tramite *operator fun invoke(...)* per un uso naturale. Delegano ai *Repository* l'accesso ai dati (rete,

Firebase) e restituiscono risultati di dominio (valori, *Flow<>*, sealed result); in questo modo separano la logica applicativa dai dettagli infrastrutturali e semplificano i test.

- **auth:**
  - **LoginUserUseCase.kt:** invoca *AuthRepository.login(email, password)* e ritorna *LoginResult* (successo/errore normalizzato).
  - **RegisterUserUseCase.kt:** invoca *AuthRepository.register(...)* e ritorna *RegisterResult* (gestione di collisioni email/username e errori di rete).
  - **GetCurrentUidUseCase.kt:** legge l'UID dall'istanza *FirebaseAuth* iniettata e lo ritorna (o *null* se non autenticato).
  - **CheckUsernameAvailableUseCase.kt:** verifica disponibilità username demandando a *AuthRepository.isUsernameAvailable*, con guardia su stringhe vuote.
- **catalog:**
  - **SearchBookUseCase.kt:** normalizza la query: se già in forma *isbn:<codice>* la inoltra; altrimenti, se il testo contiene 10 o 13 cifre (prefisso 978/979 per 13), costruisce la ibrida *isbn:<digits> OR raw* e chiama *CatalogRepository.search*.
  - **GetCategoryFeedUseCase.kt:** recupera l'elenco libri per categoria con *CatalogRepository.byCategory(category, page)* (pagina 0 di default).
- **home:**
  - **StartBookTimerUseCase.kt:** avvia una sessione di lettura restituendo il *startMillis* (usato dal *HomeViewModel* per mostrare un timer live).
  - **SetBookTimerUseCase.kt:** registra una sessione completata delegando a *HomeRepository.addReadingSession(bookId, startMillis, endMillis)* (aggiorna *sessions* e *totalReadSeconds*).
- **shelves:**
  - **ObserveBooksByStatusUseCase.kt:** espone *Flow<List<UserBook>>* da *ShelvesRepository.observeBooks(status)*.
  - **ObserveBookStatusUseCase.kt:** espone *Flow<ReadingStatus?>* per un volume (*null* se assente).
  - **ObserveUserBookUseCase.kt:** espone *Flow<UserBook?>* per osservare i metadati del libro salvato.
  - **RemoveBookFromShelfUseCase.kt:** rimuove il documento libro tramite *ShelvesRepository.removeBook(id)*.

- **SetBookStatusUseCase.kt:** imposta lo stato unico del libro (*TO_READ/READING/READ*) con eventuale *payload* metadati.
- **SetPageCountUseCase.kt:** assegna/aggiorna *pageCount* validando valori > 0.
- **SetPageInReadingUseCase.kt:** aggiorna le pagine lette validando $0 <= n <=$ *pageCount* quando disponibile.
- **UpsertStatusBookUseCase.kt:** scrittura atomica di stato e metadati: decide *pageCount* (parametro prioritario > 0, altrimenti dal *payload* se mancante), imposta opzionalmente *pageInReading*.

2.7 Unit Testing

I test sono organizzati su due livelli complementari: *unit test* (JVM pura) per validare la logica di dominio e di presentazione senza dipendenze Android, e *instrumented test* (dispositivo/emulatore) per verificare integrazioni con il framework e l'interazione UI. Nel progetto sono impiegati *JUnit4*, *Truth* per le asserzioni, *MockK* per il mocking, le librerie di test delle *Coroutines* (scheduler virtuale, *runTest*), *Robolectric* per eseguire componenti Android su JVM, e per i test strumentati *Hilt Android Testing*, *Jetpack Compose Testing* e *Navigation Testing*. L'uso di *Hilt* consente di sostituire dipendenze reali con mock attraverso *@BindValue*, mentre il controllo del dispatcher principale con *Dispatchers.setMain* e gli scheduler di test garantisce determinismo temporale.

2.7.1 Unit Test - HomeViewModelTest.kt

Verifica il comportamento della logica di temporizzazione e aggiornamento pagine del *HomeViewModel* isolandolo da I/O esterni. Le dipendenze *HomeRepository*, *StartBookTimerUseCase* e *SetBookTimerUseCase* sono mockate con *MockK*; il flusso dei libri in lettura è simulato con *MutableStateFlow*. I test adottano *RobolectricTestRunner* (esecuzione su JVM), *StandardTestDispatcher* e *runTest/runCurrent* per pilotare il tempo virtuale e collezionano lo *uiState* per attivare le combinazioni di flusso. In particolare: (i) avvio sessione: l'invocazione di *onStart* imposta *isRunning=true* e valorizza *sessionStartMillis* con il timestamp restituito dall'use case, verificando anche l'unicità della sessione quando un'altra è in corso; (ii) arresto sessione: *onStop* rimuove lo stato di running, apre la dialog di aggiornamento pagine e invoca *SetBookTimerUseCase* con l'intervallo *start/end*; (iii) conferma pagine: *confirmPages* invia *repo.updatePagesRead* e, se il numero confermato raggiunge il totale, imposta lo stato *READ* tramite *repo.setStatus* costruendo un *UserBook* coerente e chiudendo la dialog. Le asserzioni verificano i campi di *HomeItemState*, l'apertura/chiusura della dialog e le interazioni attese con i mock.

2.7.2 Instrumented Test - BookDetailScreenInstrumentedTest.kt

Convalida il flusso di interazione della *BookDetailScreen* in un ambiente Android reale con *Compose*. Il test è annotato *@HiltAndroidTest* e utilizza *HiltAn-*

droidRule più *createAndroidComposeRule* su **HiltTestActivity**. Le dipendenze del *ViewModel* sono sostituite a livello di processo mediante *@BindValue* (mock di *ObserveBookStatusUseCase*, *SetBookStatusUseCase*, *RemoveBookFromShelfUseCase*, *SetPageCountUseCase*, *ObserveUserBookUseCase*, *UpsertStatusBookUseCase*). La schermata è montata dentro un *NavHost* pilotato da *TestNavHostController* con *ComposeNavigator* per simulare la navigazione. I casi coperti:

- Se lo stato corrente è *TO-READ*, il tap sull'icona "Da leggere" esegue la rimozione dallo scaffale verificando l'invocazione del relativo use case;
- In assenza di stato e con totale pagine noto, il tap su "Letto" invoca *UpsertStatusBookUseCase* con *status=READ*, *pageInReading* uguale al totale e *pageCount* non ripetuto (valore *null* perché già noto dal libro).

Il test driver di Compose interagisce con i nodi tramite *contentDescription*, assicurando accessibilità e selettori stabili. L'infrastruttura di test include **HiltTestRunner** (bootstrap di *HiltTestApplication*) e un manifest di test che dichiara **HiltTestActivity** per l'host delle UI.

3 Progettazione e sviluppo in Flutter

3.1 Requisiti e casi d'uso

3.1.1 Requisiti funzionali

3.1.2 Requisiti non funzionali

3.1.3 Casi d'uso

4 UI - Interfaccia applicazione

5 Discussioni di problematiche riscontrate

6 Conclusioni