

IZMIR UNIVERSITY OF ECONOMICS  
FACULTY OF ENGINEERING  
DEPT. OF SOFTWARE ENGINEERING

# FENG 497 (498) PROJECT REPORT



## **READIFY: A Social Library**

Author(s): Emre Bekmezci

Deniz Dumbak

Gökberk Ömer Çoban

Supervisor: Dr. Kaan Kurtel

## Version History

Version	Date	Commend
1.0	01.01.2022	The objective was written. The introduction was written.
1.1	16.11.2022	The problem statement and motivation were written. The scope of the work was written. A literature survey was written.
1.2	24.11.2022	The methodology was specified. Technologies were written. Gantt Chart was drawn. Similar Applications were researched. The database was specified. The platform was explained.
1.3	25.11.2022	Vocabulary was added. Risks were discussed. References were added.
1.4	28.12.2022	Requirements were specified. Diagrams were drawn. Test cases were specified.
1.5	15.01.2023	Conclusion was explained.
2.0	19.05.2023	Introduction was updated. Motivation was updated. Methodology was updated. Literature review was updated.
2.1	27.05.2023	Database was updated. Risks was updated. Technologies was updated. Requirements were updated.
2.2	01.06.2023	Implementation and design were added. Conclusion was updated.

## Vocabulary

**Z39.50:** Platform-independent open communication protocol, specifies a standard way of communication between two systems for searching databases and retrieving information.

**API:** Application programming interface, a way for two or more computer programs to communicate with each other.

**Dart:** A programming language designed for client development, such as for web and mobile applications.

**Dublin Core:** A set of fifteen "core" elements (properties) for describing resources.

**Firebase:** Set of hosting services for any type of application.

**Flutter:** An open-source UI software development kit created by Google, used to develop cross-platform applications.

**FPS:** Frame rate, the frequency at which consecutive frames are captured or displayed.

**Hot Reload:** A functionality present in Flutter.

**Ionic:** A complete open-source SDK for hybrid mobile application development.

**MARC:** Machine-Readable Cataloging, standards are a set of digital formats for the description of items cataloged by libraries, such as books, DVDs, and digital resources.

**Microblogging:** A form of social network that permits only short posts.

**React Native:** An open-source UI software framework created by Meta Platforms, Inc.

**SQL:** Structured query language, a domain-specific language used in programming and designed for managing data held in a relational database management system.

**UI:** User Interface, the point of human-computer interaction and communication in a device.

**UWP:** Universal Windows Platform, a computing platform created by Microsoft and first introduced in Windows 10.

**Widget:** An immutable description of part of a user interface.

**Firebase – Firestore:** Firebase Firestore is a scalable, flexible, and real-time synchronization NoSQL cloud database provided by Google.

# CONTENTS

İZMİR UNIVERSITY OF ECONOMICS FACULTY OF ENGINEERING .....	i
Vocabulary .....	iii
CONTENTS .....	<b>Hata! Yer işareti tanımlanmamış.</b>
1.Objective .....	1
2.Introduction.....	1
2.1.Motivation.....	2
3.Scope of The Work .....	2
4.Methodology .....	4
5.Literature Review .....	5
5.1. Similar Applications .....	5
5.2 Database .....	7
5.3 Platform .....	7
6. Project Management and Governness .....	8
6.1. Risk .....	8
6.2. Gantt Chart .....	10
6.3 Technologies.....	10
7. Requirements .....	11
7.1 Functional Requirements .....	11
7.2 Non-Functional Requirements .....	12
7.3 Test Cases .....	14
7.4 Diagrams.....	15
8. System Desing – Graphical User Interface .....	17
9. Implementations .....	40
10. Test .....	68
11. Conclusion .....	69
References.....	70

# 1. Objective

- The aim of Readify is to provide a social platform to book readers, authors, and everyone who will use the application to share their ideas, offer and receive suggestions about books.
- Readify aims to provide such an environment that Readify users can taste the experiences of other users about books and life and build up their self-improvement.

# 2. Introduction

A social platform defines as generally a web-based technology that enables the development, deployment, and management of social media solutions and services. It provides the ability to create social media websites and services with complete social media network functionality.[1]

The project's title is Readify, and Readify serves to meet people who want to read books but cannot decide, meet people with similar reading tastes, discover different genres, and create an archive. The decision process will be shorter, and users might be interested in different categories. Moreover, users are able to use reading books as a tool to be more social.

As mentioned above, Readify expedites the decision process and provides discovering genres because it includes many comments and suggestions from other readers about many different categories of books. Furthermore, Readify users can exchange experiences, inspirations, and opinions between them and share emotions. As well as exchanging inspirations and sharing emotions, Readify users benefit from these experiences or opinions and can build up them personally, moreover, acquire their empathy skills. Moreover, users can create their book history. In addition, Readify users can access other users' book histories. Furthermore, users can see each other's when they read the same book at the same time so they can find someone with similar reading tastes and create a book history.

In short, there are many social media platforms today where people can meet and communicate with each other, to be able to share their ideas and add value to each other such as Instagram, Twitter, and Facebook. In addition to these features, Readify has some features that can make it different from other social media platforms. For instance, since users can see other users's favorite books, currently reading and have read before so communicating easier with other users. Most importantly, Readify gets users to adopt real-life habits.

Finally, Readify is a mobile application that allows its users to exchange ideas about the book, suggest a book. We will use Flutter technology to design and build this mobile app. At the same time,

we need to have a database so that we can create the library it contains. We need Firebase technology to create and use this database. In the following chapters, this project presents some technical backgrounds.

### Problem Statement

Most readers cannot decide what to read; from our perspective, this is one of the main reasons not to read. In addition, people need to socialize, and we want people to associate while reading. In real life, this is impossible. There are chances to socialize in libraries or some coffee shops but the conditions in these places are restricted, so we think we need to solve this problem by creating a social media platform that forces users to read and socialize.

## **2.1. Motivation**

There are many popular and accepted social media platforms in this age such as Twitter, Goodreads, and LibraryThing. Moreover, each of those platforms responds to almost all requests of users. Users can share and exchange their ideas and feelings and create their fields. Moreover, users can gain knowledge and communicate with other users. Furthermore, and most importantly, users want to have a good time and meet someone who can have fun and improve with. However, despite the existence of many social media platforms, users often fail to establish respectful, beneficial, and solid relationships with other users that allow them to get to know and develop themselves. Especially users generally cannot do this quickly, meeting with good faces, through the help of just one book.

In addition, Readify does what most social media platforms generally cannot allow the user to meet other users thanks to just one book. Moreover, Readify allows users to communicate with each other by seeing who read the book they are reading, thanks to the users they follow and the users who are followers. Furthermore, whereas the previous book social media platforms have aimed to meet with other users in the community, Readify, also allows for quick communication between users by showing who has read the books.

## **3. Scope of The Work**

Readify is a mobile application that allows writers, readers, and all other people to interact with each other about books, and can be used anytime, anywhere. Readify allows people who want to read a book but can't decide, those who want to know about books, those who want to comment on books, to exchange ideas with each other, users to make book archives for themselves and to discover different types of books; It aims to enable people to have prior knowledge about books, to meet new people and to accelerate their decision-making processes. We will collect and classify the

requirements of our project. We will present our project to people and get feedback. According to this feedback, we will add or remove the project scope. [2] We will make and check the project plans, follow-ups, and scope via Trello or Github. We will use Flutter technology to design and build Readify mobile app, as well as Firebase technology to build and use the library within the app and the database that allows people to archive theirs.

## 4. Methodology

Figure 4.1 shows the methodology of the project and the main sections with their sub-sections. The boxes with black color have done in the first semester and the boxes with the color white will be done in the second semester. The flag represents the proposal and the sections above the flag are submitted at the proposal deadline.

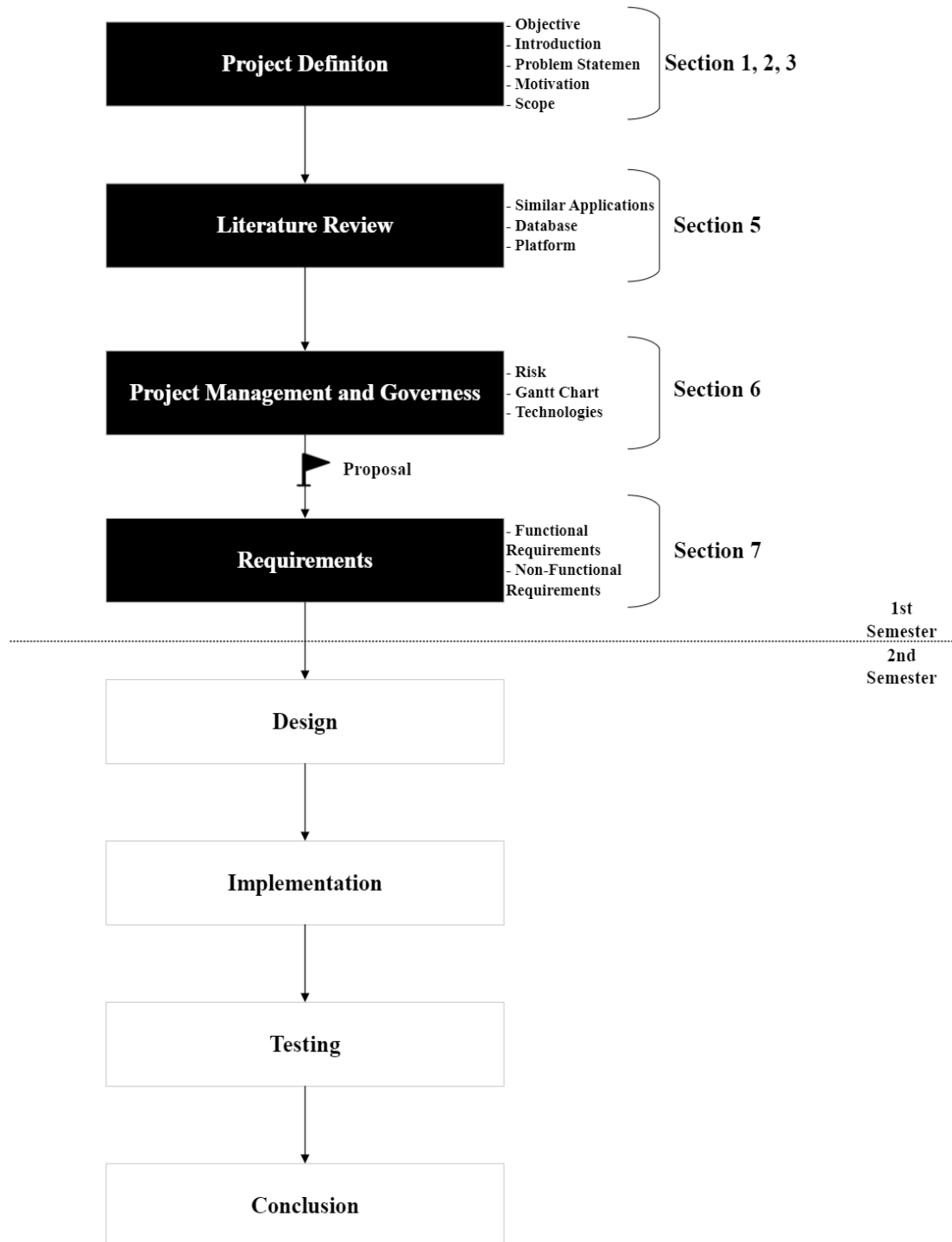


Figure 4.1



## 5. Literature Review

Day by day, many applications are offered to users. Different applications respond to almost all requests of users. When this is the case, it is close to impossible to create an application that has not existed before. Therefore, we now think that creating an application that contains the features of several different applications rather than coming up with an original idea, will make our application successful.

Readify aims to serve the user by combining the features of a social media platform and a library application. A social media platform is based on sharing and, this includes elements such as sharing ideas, commenting, liking the expressed opinion, and messaging. A library application sorts books by author and by title. Moreover Readify, allows users to like, commend and share these books and provides basic information about the books to the user.

There are some similar applications that Readify is inspired by. As we mentioned, Readify is fundamentally a social network application, and Twitter and Facebook are two of the best examples of social network applications. Otherwise, Readify can be defined as a social library so, there are also some applications from that Readify originates. Goodreads LibraryThing and Litsy are three of them and appropriate to compare.

### 5.1. Similar Applications

Twitter is a microblogging and social networking service owned by American company Twitter, Inc., on which users post and interact with messages known as "tweets". Registered users can post, like, and retweet tweets, while unregistered users only have a limited ability to read public tweets. Users interact with Twitter through browser or mobile frontend software, or programmatically via its APIs. [3]

Facebook is an online social media and social networking service owned by American company Meta Platforms. Facebook can be accessed from devices with Internet connectivity, such as personal computers, tablets, and smartphones. After registering, users can create a profile revealing information about themselves. They can post text, photos and multimedia which are shared with any other users who have agreed to be their "friend" or, with different privacy settings, publicly. Users can also communicate directly with each other with Facebook Messenger, join common-interest groups, and receive notifications on the activities of their Facebook friends and the pages they follow. [4]

Goodreads is an American social cataloging website and a subsidiary of Amazon that allows individuals to search its database of books, annotations, quotes, and reviews. Users can sign up and register books to generate library catalogs and reading lists. They can also create their own groups of book suggestions, surveys, polls, blogs, and discussions. [5]

LibraryThing is a social cataloging web application for storing and sharing book catalogs and various types of book metadata. It is used by authors, individuals, libraries, and publishers. The primary feature of LibraryThing (LT) is the cataloging of books, movies, music, and other media by importing data from libraries through Z39.50 connections and from six Amazon.com stores. Library sources supply Dublin Core and MARC records to LT; users can import information from over 2000 libraries, including the British Library, Canadian National Catalogue, Library of Congress, National Library of Australia, and Yale University. Should a record not be available from any of these sources, it is also possible to input the book information manually via a blank form.[6]

Litsy is an iOS and Android social media application and website that is based on reading books. It was launched in April 2016. Some book releasers have used it for their marketing. Users can make three types of posts: a quick blurb, a quote, or a review and all posts can have a maximum of 300 characters. Posts can be marked as having spoilers when they are submitted. They can also "like" books, upload photos, use emojis, and create virtual stacks of books. Every user has a "Litfluence" score which shows how influential each one of them is on Litsy. [7]

In table 5.1, there are the major features that are critical for Readify from our perspective and some other features that make a difference from other applications.

Table 5.1 – Comparison Table

FEATURES	TWITTER	FACEBOOK	GOODREADS	LIBRARYTHING	LITSY	READIFY
Price	FREE	FREE	FREE	FREE	FREE	FREE
Comment	✓	✓	✓	✓	✓	✓
Like	✓	✓	✓	✓	✓	✓
Direct Message	✓	✓	✓	✓	✓	✓
Authentication	✓	✓	✓			✓
Domain Specified			✓	✓	✓	✓
Current Users	450 Million	2.96 Billion	125 Million	2.1 Million	20.000	N/A

## 5.2 Database

The two main actors of Readify which are Readify users and the books in the application, and various relationships that may be related to them and their specific information are called data. Moreover, we need to store this data somewhere. We will use the Firebase Firestore database to store our data. Furthermore, we access and manage the data we store. We chose the Firebase Firestore database over other databases because it is easy to use and compatible with Flutter, which we will use as our programming language. Moreover, it is an ideal database for a social platform. We can analyze users' information and verify their identity information.

Moreover, we can instantly communicate with users and send notifications to them. In addition, we believe that the Firebase Firestore database has advantages in creating a social media platform compared to other databases in terms of storage and performance. Moreover, we are also able to obtain some analysis about users through Firebase Analytics. Furthermore, we were encouraged by Spotify, HSBC, Home Depot, Snapchat, Philips use of the Firebase Firestore database.

## 5.3 Platform

There are many mobile application development platforms that people use or can use to develop Mobile applications. Some of these platforms; Powered by Facebook, React Native is a mobile software development tool used to build cross-platform apps capable of delivering native platform-like performance across iOS, Android, and UWP platforms. It is used by Facebook, Airbnb, Walmart, and Tesla. React uses JavaScript as its native programming language. Another application of ours; is Ionic, an application development framework for cross-platforms. Mobile application development can be done on multiple platforms using a single code base. As a programming language, Ionic makes the most of web technologies such as HTML, CSS, and JavaScript. Here we refer to Ionic as a cross-development application development tool. This is because you can use Ionic to build hybrid cross-platform apps for the web, native iOS, and Android platforms. Finally, Flutter, also powered by Google, is a comprehensive mobile software development framework that delivers world-class native experiences on Android and iOS platforms. Flutter uses the Dart language developed by Google as its programming language. [8] We are working to create and develop our Readify mobile application, which is our project; We will

use the flutter mobile application development platform, which we have the know-how of and use before.

## **6. Project Management and Governness**

### **6.1. Risk**

There are risk types that we can gather under 2 different headings that we foresee in the Readify mobile application project we are doing. One of these headings is the risks related to the operation of our application, and the other is the risks in the technical structure of our application.

Readify is a mobile application that serves to meet people with similar reading tastes, discover different genres and create archives, comment on books, and send message to people with the same book tastes but people can use this application for other purposes. The risks are explained below.

- The purpose of people communicating with other people may not be booked.
- People have the right to comment on books in our Readify application. But people can use our app for another purpose by commenting on different topics.
- They can show all books as read or reading so misleading occurs side of other users.

There may also be risks posed by the technologies we use while developing our Readify mobile application. We will use Flutter as the mobile application development platform. There may be risks posed by this platform, for example:

- Applications prepared with Flutter are mobile-oriented. Therefore, the limited storage space that mobile devices have should be considered. However, applications prepared with Flutter may have high file sizes. For example, A comfortable 500 KB application built with Java has a file size of 4.7 MB to 6.7 MB when prepared in Flutter. [9]
- Since Flutter is a new technology, we can say that it still has some shortcomings. one of them is the lack of a third-party library. No matter how much you customize the widgets offered by Flutter, things can get a little slow as no third-party libraries are offered internally. [9]
- Flutter says we need to learn a new language. Although the Dart programming language is much more useful than other programming languages in many ways, we do not know how much it is worth the time we spend at the end of the learning period, as it is used in a single tool. [9]

Besides, we will use Firebase as the database and there are risks in using it.

- In case of not providing much income and a low number of users; When we install our application, which has such a structure, we stay connected to this platform, and it will be a waste of time and money to move our application to large servers in case of later negativities. i.e., Unless our App is running on a single central database updated by many users, that would be a huge overkill. [10]
- Firebase's storage format is completely different from SQL format (Firebase uses JSON), so we can't migrate it easily. [11]
- The cost of Firebase can be quite high according to the number of users. Because: Limited to 100 Connections and 1GB Storage. [11]

There may be risks that arise after we develop our Readify mobile app, for example:

- The full names and e-mail addresses of the users registered in the system are visible in their profiles by other users.
- The users can receive messages from all other users in the system, users cannot block any user.

<b>Risk Driver</b>	<b>Probability in Percentage (%)</b>	<b>Impact (in levels 1 to 5)</b>
<b>User Caused</b>		
- Out-of-purpose communication	<b>50</b>	<b>5</b>
- Commenting on Different Topics	<b>40</b>	<b>4</b>
- Showing all books as read or reading	<b>40</b>	<b>3</b>
<b>Development Platform Caused</b>		
- High file sizes	<b>30</b>	<b>4</b>
- Some shortcomings due to being a new technology	<b>10</b>	<b>2</b>
- Uncertainty about how much time spent at the end of the learning period is worth	<b>10</b>	<b>1</b>
<b>Database Caused</b>		

- Not running on a single central database updated by many users	<b>30</b>	<b>3</b>
- Storage format difference compared to SQL	<b>30</b>	<b>2</b>
- High cost according to the number of users	<b>20</b>	<b>2</b>

Table 6.1.1 demonstrates the probability of the risks and their impact on the project.

Table 6.1.1 – Risk Table

## 6.2. Gantt Chart

Figure 6.2.1 illustrates the main sections of the project and their time period of implementation.

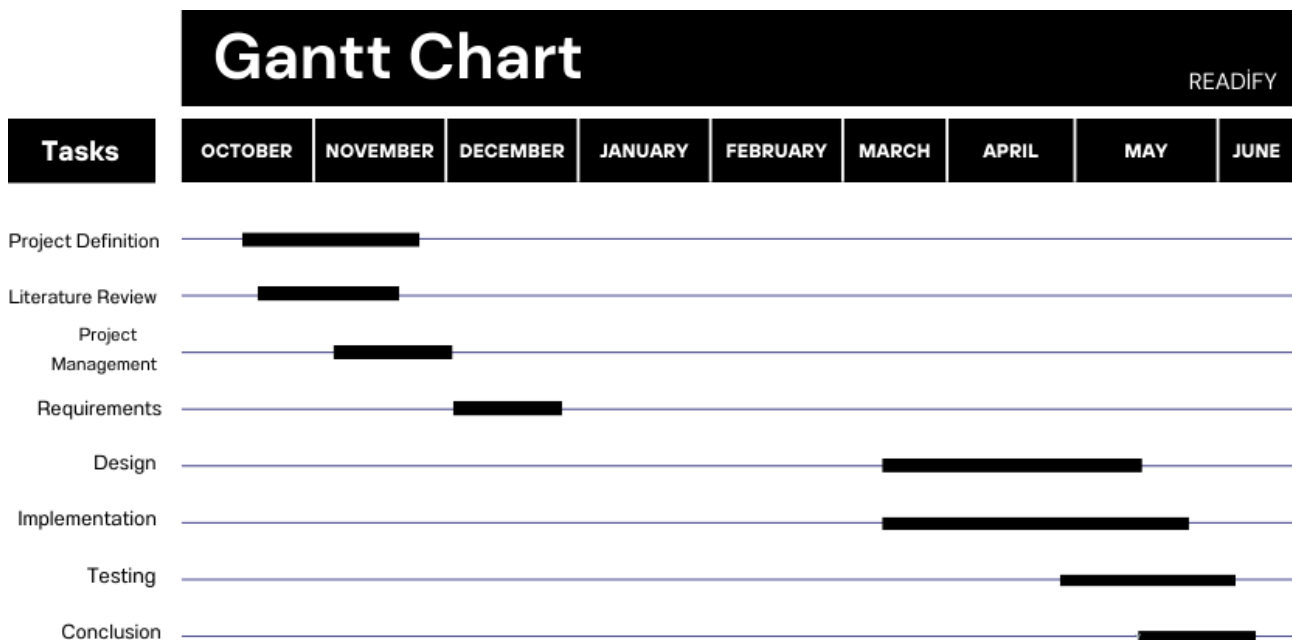


Figure 6.2.1 – Gantt Chart

## 6.3 Technologies

There are also certain reasons why we use flutter technology, which is a mobile application development platform, in our Readify application other than our prior knowledge. a few of these reasons are:

- Flutter's use of Dart as a programming language, Flutter implementations are written in Dart code, which is compiled in native code for each platform. This allows us to use a single set of Dart

code for all platforms without having to write separate apps for each platform, thus allowing us to share code between our iOS and Android apps and maximize efficiency using a single language across all platforms. [12]

- Flutter offers a rich Material Design, Cupertino, and Widget toolset so that we can make our app look and feel like it belongs to every platform. [12]

- Hot Reload, Flutter's reactive framework, and statefulness allow us to create beautiful UIs that run at 60 FPS. [12]

- Flutter gives us detailed control over how your user interface interacts with the underlying hardware on each platform. [12]

At the same time, there are certain reasons why we use Firebase as a database.

- Apps to analyze usage data and send notifications to users to make new announcements.[13]

- It offers some important facilities such as testing the application before publishing it. [13]

- Readify uses Google Book API to fetch book data.

## **7. Requirements**

### **7.1 Functional Requirements**

**F1.** Readify provides its users with an in-app search button; Authors should allow searches for book titles.

**F2.** Readify should allow users to access their book archives by all other users.

**F3.** Readify should allow users to archive their own books.

**F4.** Readify should allow users to comment.

**F5.** Readify should allow users to follow each other.

**F6.** Readify should allow users to make a wish list.

**F7.** Readify should allow users to like books.

**F8.** Readify should allow users to see details of books.

**F9.** Readify should allow users to send messages to each other.

**F10.** Readify should allow users to view book categories.

- F11.** Readify should allow users to view other users' profiles.
- F12.** The system should allow users to create and edit their own profiles.
- F13.** The system should allow users to log in.
- F14.** The system should allow users to sign up.
- F15.** The system should allow users to be able to change the language of the application from the settings section.
- F16.** The system should allow users to be able to exit the application by pressing the logout icon on the profile page.
- F17.** The system should allow users to display the user's full name and e-mail address registered to the system on the user's profile page.
- F18.** The system should allow users to show the books that users like on their profile page.
- F19.** The system should allow users to show the books that users are reading instantly on their profile pages and home page.
- F20.** Readify allows users to see all other users registered to the system.
- F21.** The system allows users to follow and unfollow other users.
- F22.** The System allow user to like another user's comment.
- F23.** The system allows the users to view the relevant book comments.

## **7.2 Non-Functional Requirements**

- NF1.** Readify must allow users to log in with their unique E-mail addresses and passwords.
- NF2.** Readify must allow users to authentication their accounts using their mail accounts.
- NF3.** Readify must have a library database that has categorized books according to their authors and titles.
- NF4.** Readify must not allow the user's name to appear in the user's password.
- NF5.** Readify must allow users to reset their password by clicking on "I forgot my password" and receiving a link to their verified mail address.
- NF6.** The system must be available 24/7.
- NF7.** The system must include Turkish and English language choices.



- NF8.** The system must load at most in 10 seconds.
- NF9.** The system must run in all mobile devices and tablets.
- NF10.** The system should use Firebase Firestore Database for database operation.
- NF11.** Readify must uses Google Book API to fetch book data.
- NF12.** The system must be compatible with Android, IOS, Linux, MacOS, Windows operating systems.
- NF13.** The system must display the dates of the messages and comments in the system as DD-MM-YYYY with hour and minutes.
- NF14.** The system must display a warning message thanks to the snackbar when the user enters an invalid value to the system registration screen or the system login screen.
- NF15.** The system must display "Have a problem" when the user enters an invalid input in the search box.

## 7.3 Test Cases

Figure 7.3.1 illustrates the steps while test cases are specified. These steps are included the needed stages before creating test cases and the stages that test cases created.

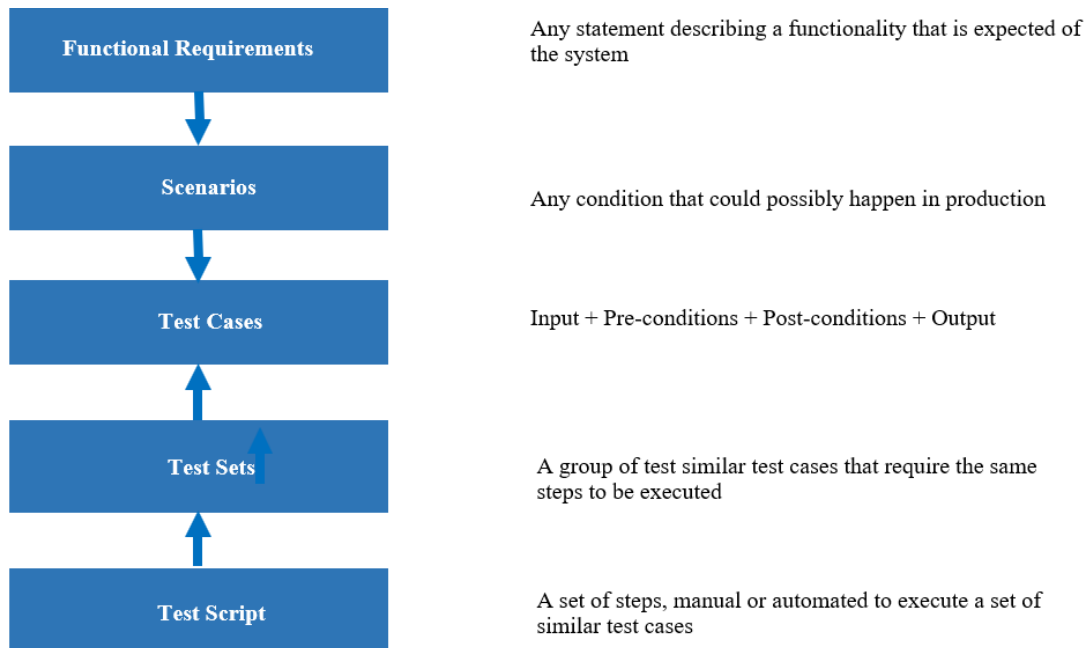


Figure 7.3.1 – Test Case Creation Stages

### F1- Test Case

Test Case	Pre-Condition	Test Steps	Test Data	Expected Result	Actual Result	Status
App launch	App is installed	Trigger <code>MyApp()</code> widget	MyApp0 function	App is launched and initial frame is created	To be filled after test execution	To be filled after test execution
Initial counter state	App is launched	Check if the counter starts at '0'	Counter's state	Counter starts at '0'	To be filled after test execution	To be filled after test execution
'+' button click	Counter is at '0'	Find and tap the '+' icon	'+' icon	'+' button click is successful	To be filled after test execution	To be filled after test execution
Counter increment	'+' button is clicked	Check if the counter is '1'	Counter's state	Counter is at '1'	To be filled after test execution	To be filled after test execution

## 7.4 Diagrams

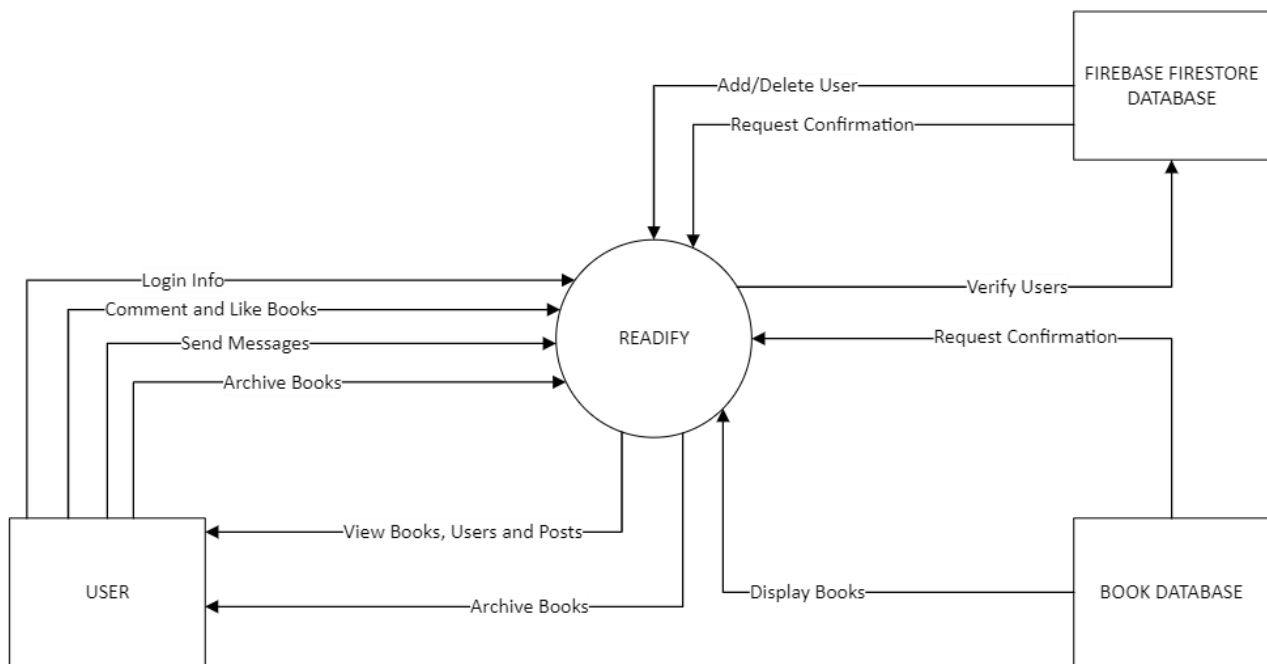


Diagram 1 – Context Diagram

Diagram 1 shows the relationships between the system and its subsystems. The user has some operations that are sent to the system and the system has functions that are sent to the user. Firebase can add and delete user and verify them. Finally, Book database display books to the user.

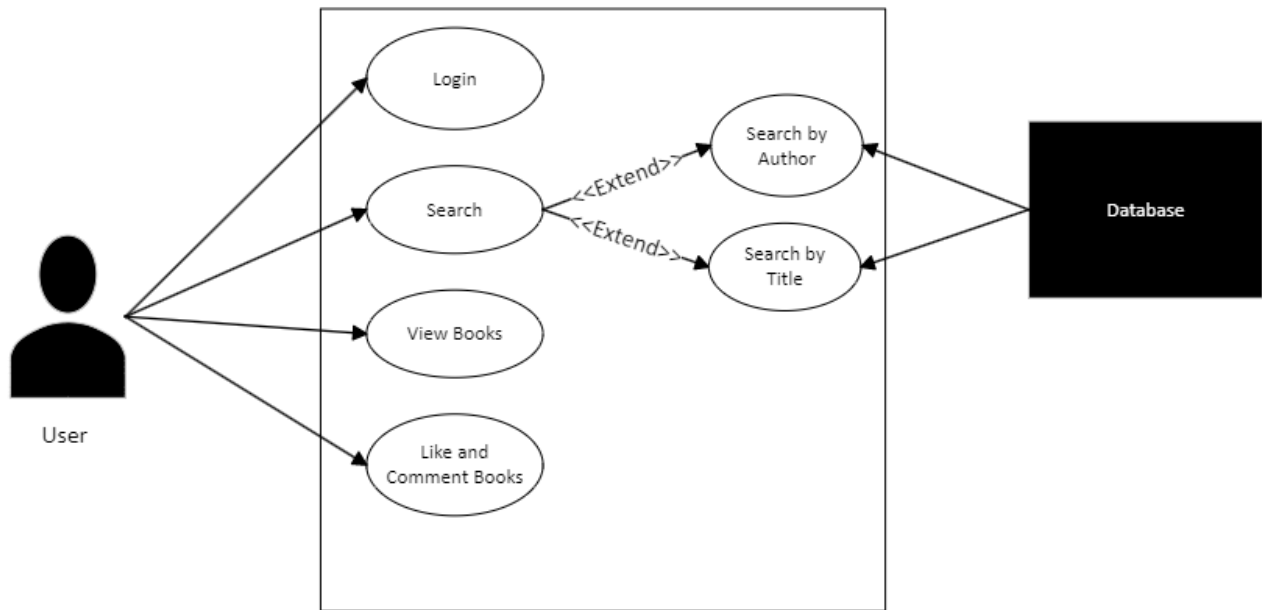


Diagram 2 – Use Case Diagram

Diagram 2 shows the use case of the searching function of Readify. The user needs to log in first and to login. After that, they can search book by author or book title. Then, they will see the list of books according to keywords that they search and they can comment, archive, view and like these books.

## 8. System Desing – Graphical User Interface

### Log in Page

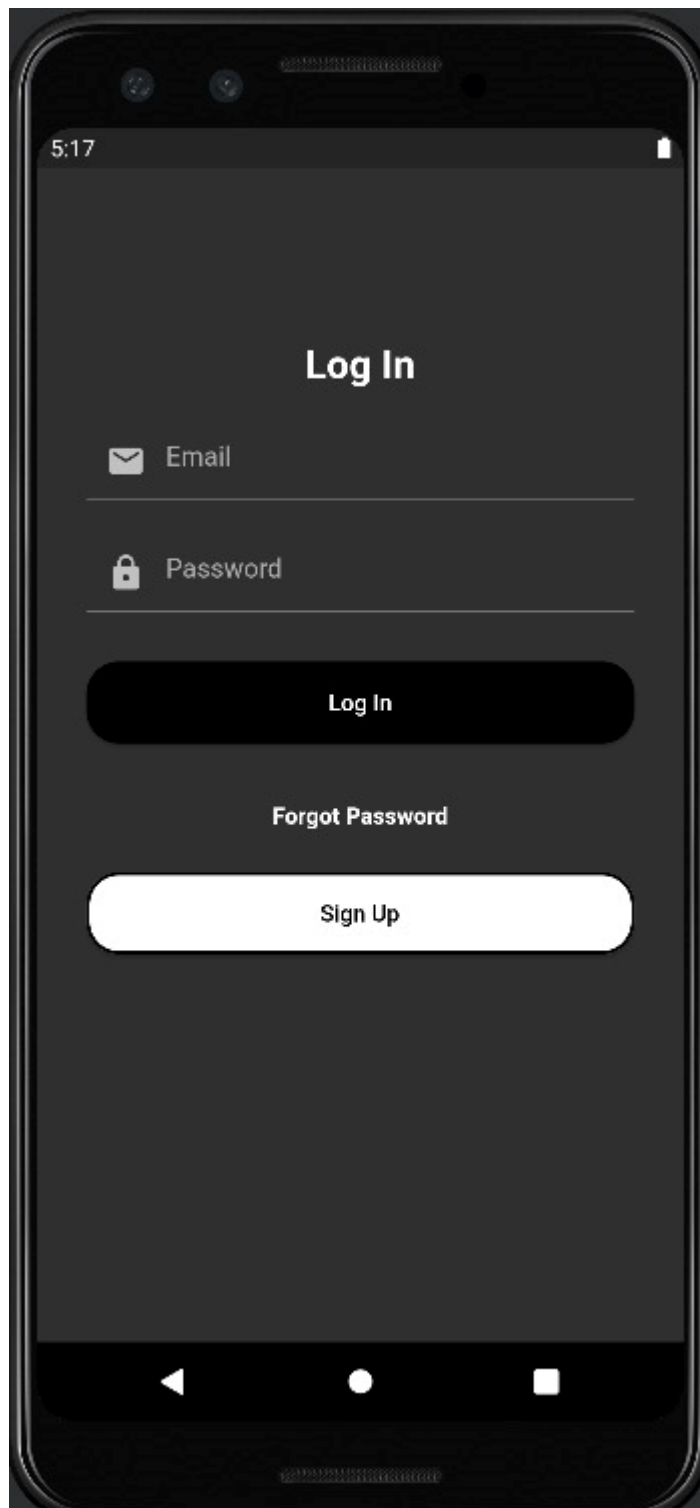


Image1: This page is log in page for the users.

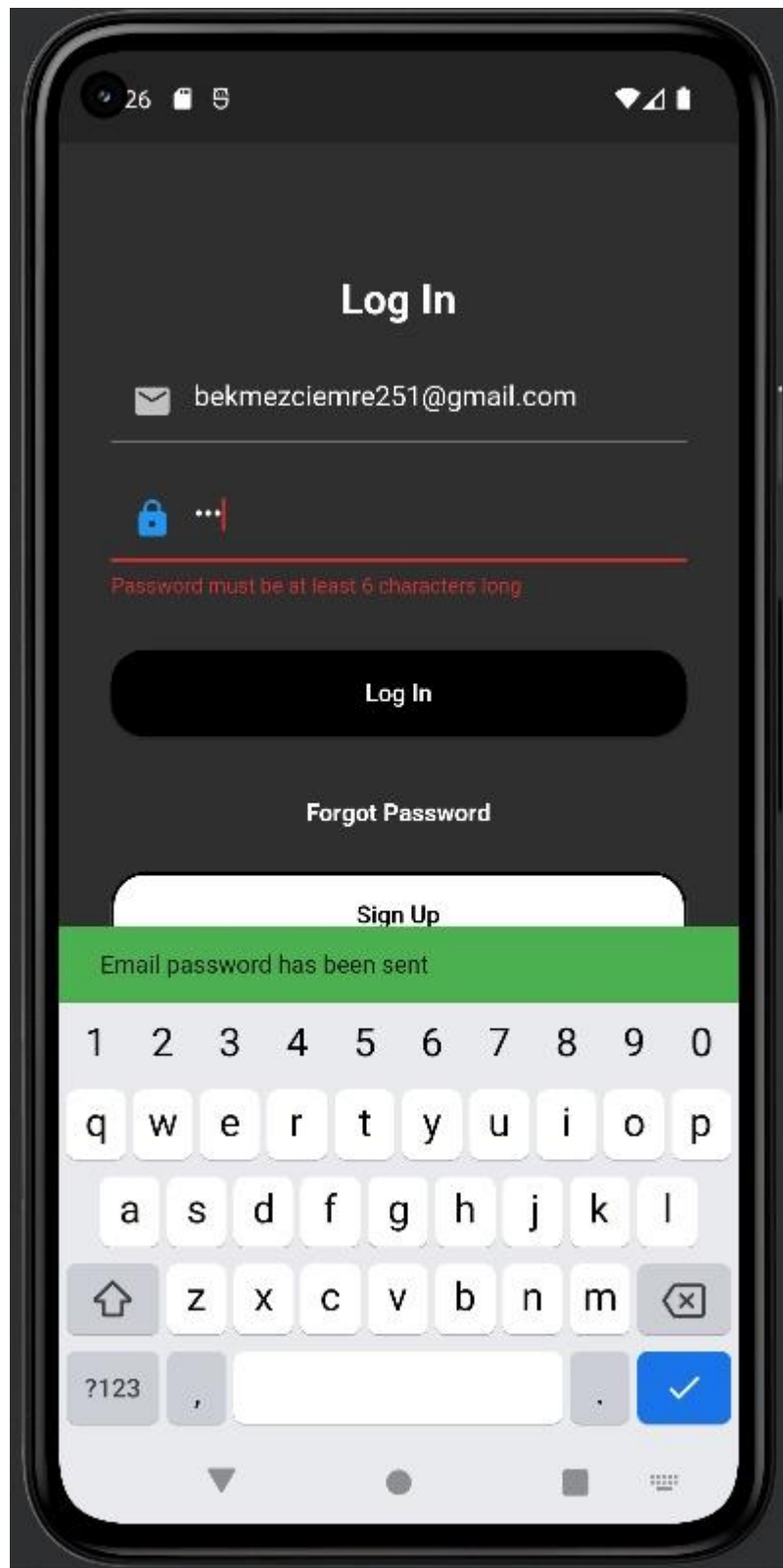


Image2: This page is after entering the valid mail address for forget password and gives the user a warning via snackbar.

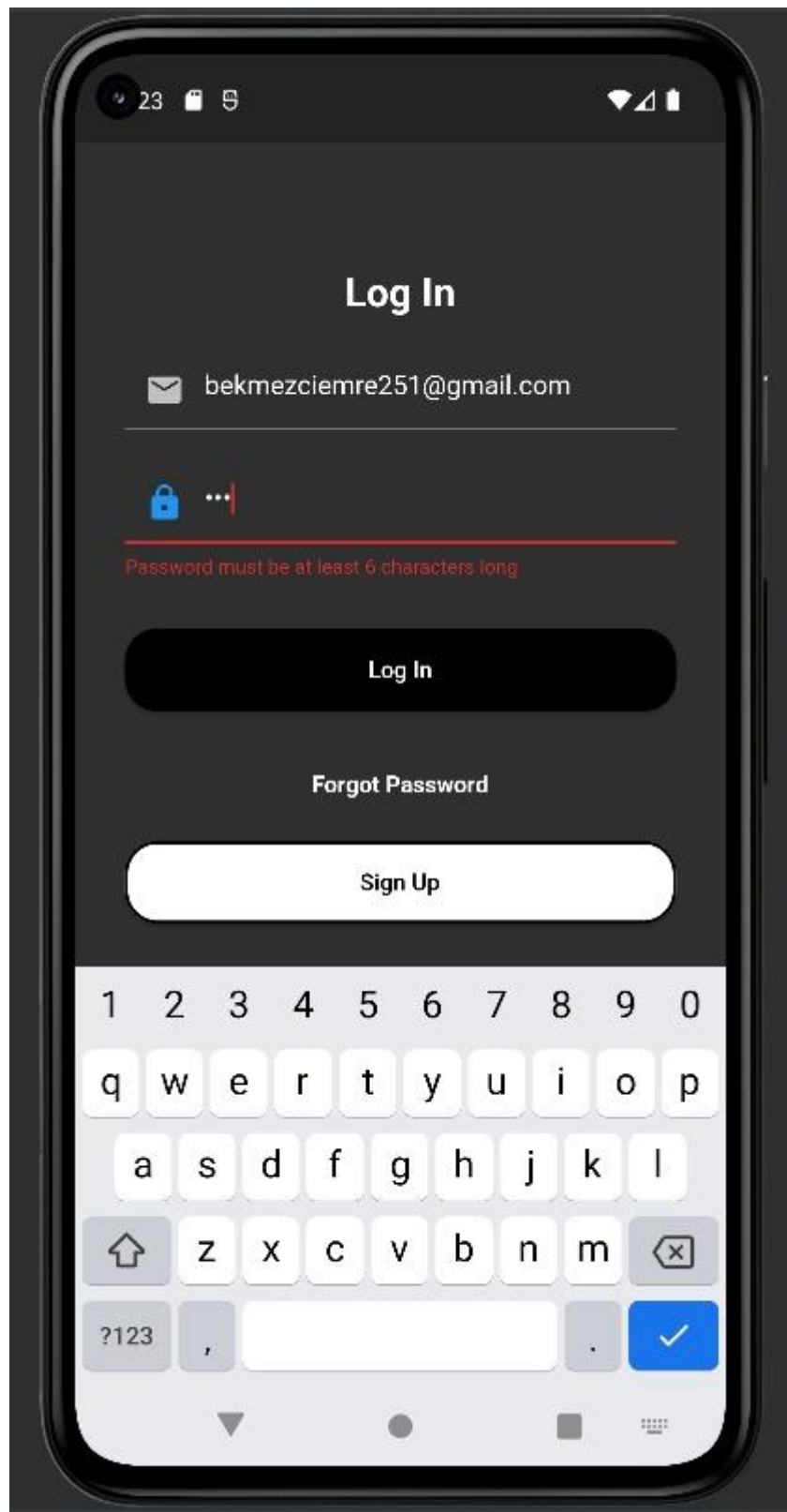


Image3: This page constraints on log in for the user.

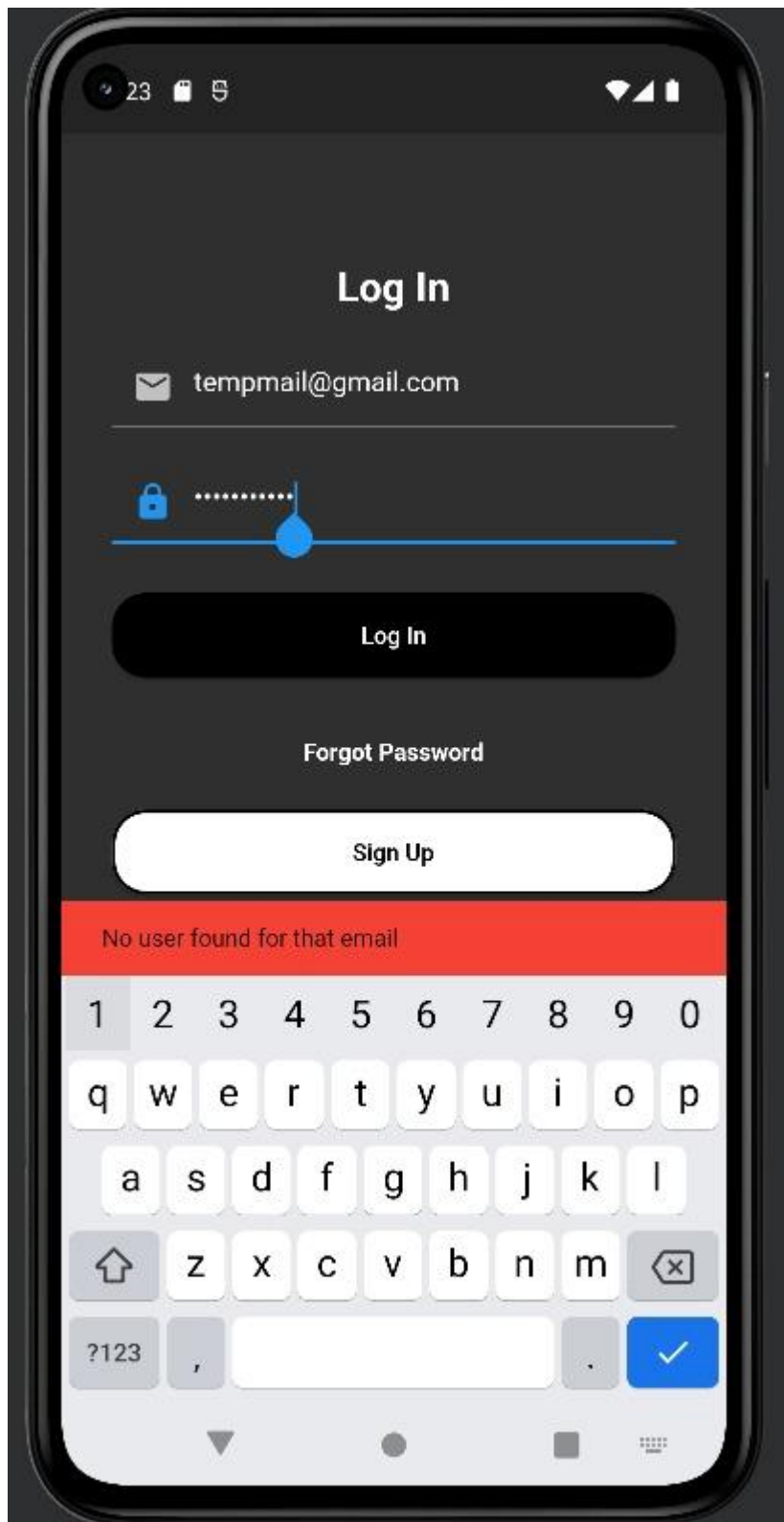


Image4: This page constraints on registering for the user and gives the user a warning via snackbar.



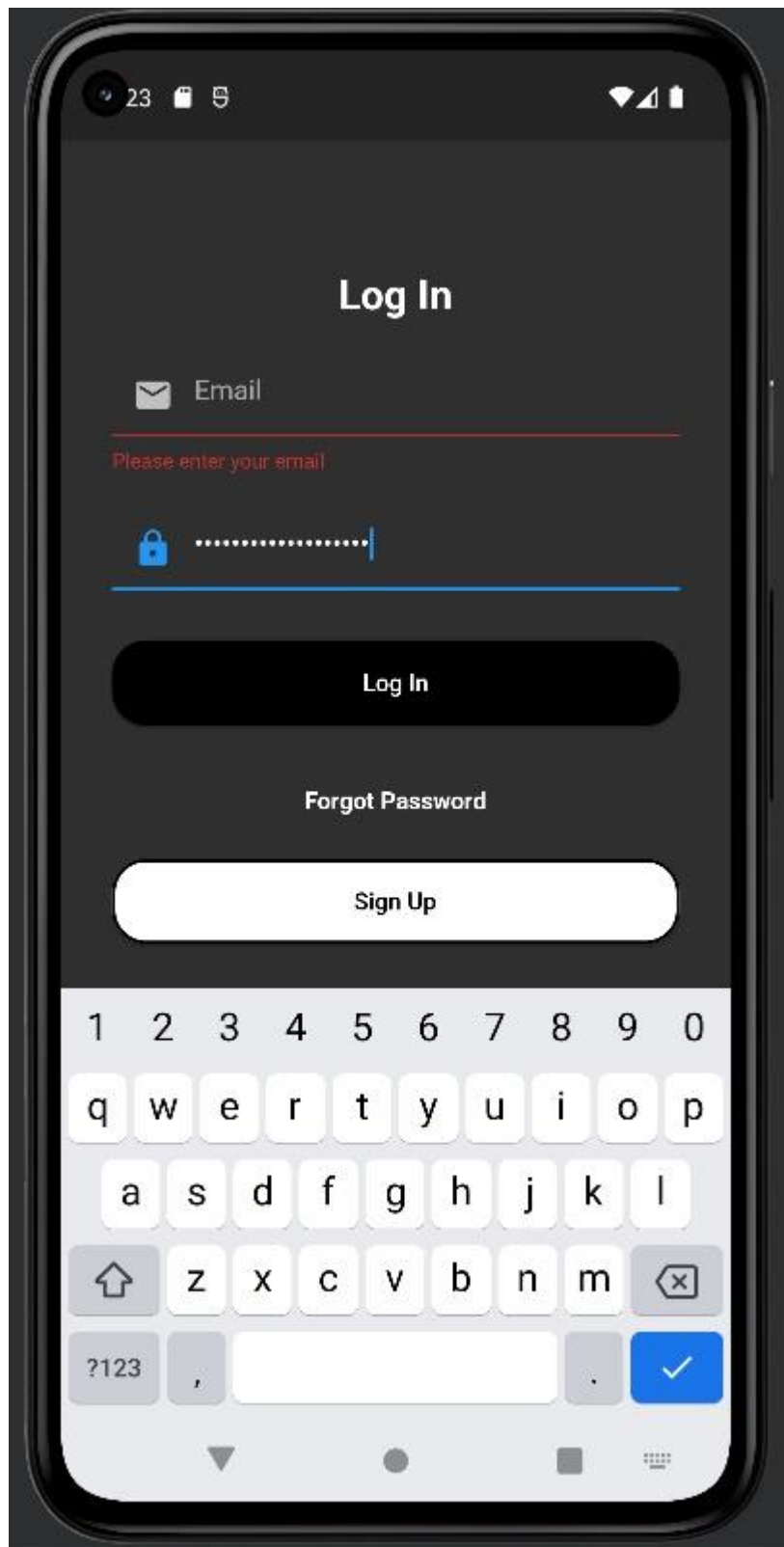


Image5: This page constraints on log in for the user.

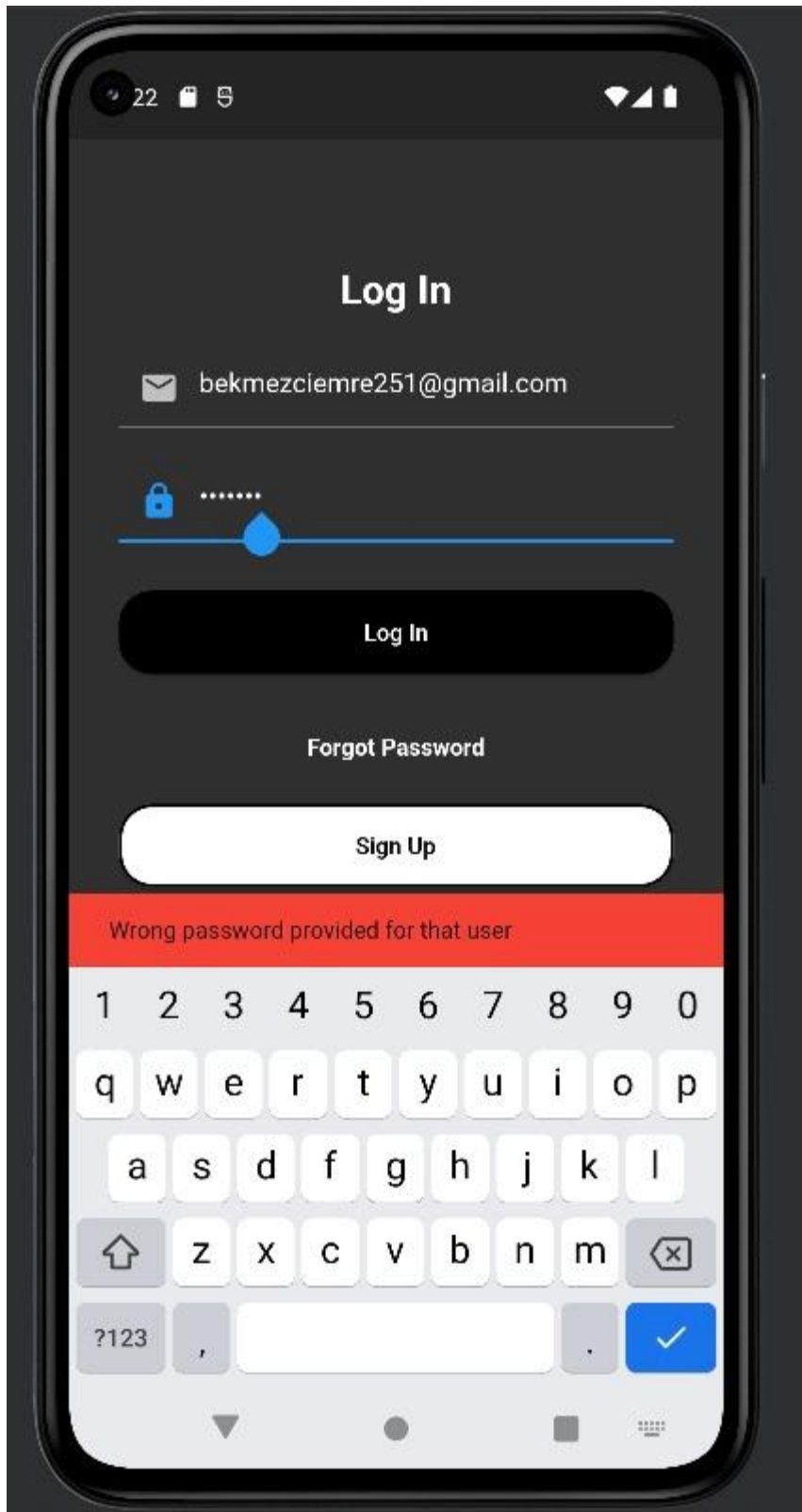


Image6: This page constraints on log in for the user and gives the user a warning via snackbar.

## Sign up Page

The image shows a mobile application interface for a sign-up page. The background is a dark gray. At the top, there is a status bar with the number '28' and icons for signal, Wi-Fi, and battery. Below the status bar, the title 'Sign Up' is centered in a white, sans-serif font. Underneath the title, there are three input fields, each with a small icon to its left: a person icon for 'Full Name', an envelope icon for 'Email', and a lock icon for 'Password'. Each input field has a horizontal line below it. Below the input fields, there is a dark, rounded rectangular button with the text 'Sign Up' in white. Underneath the button, the text 'Already have an account?' is centered in a smaller white font. Below this text, there is a light gray, rounded rectangular button with the text 'Log In' in dark gray. At the bottom of the screen, there is a dark navigation bar with three white icons: a back arrow, a circle, and a square.

Image7: This page is Sign up page for the users.

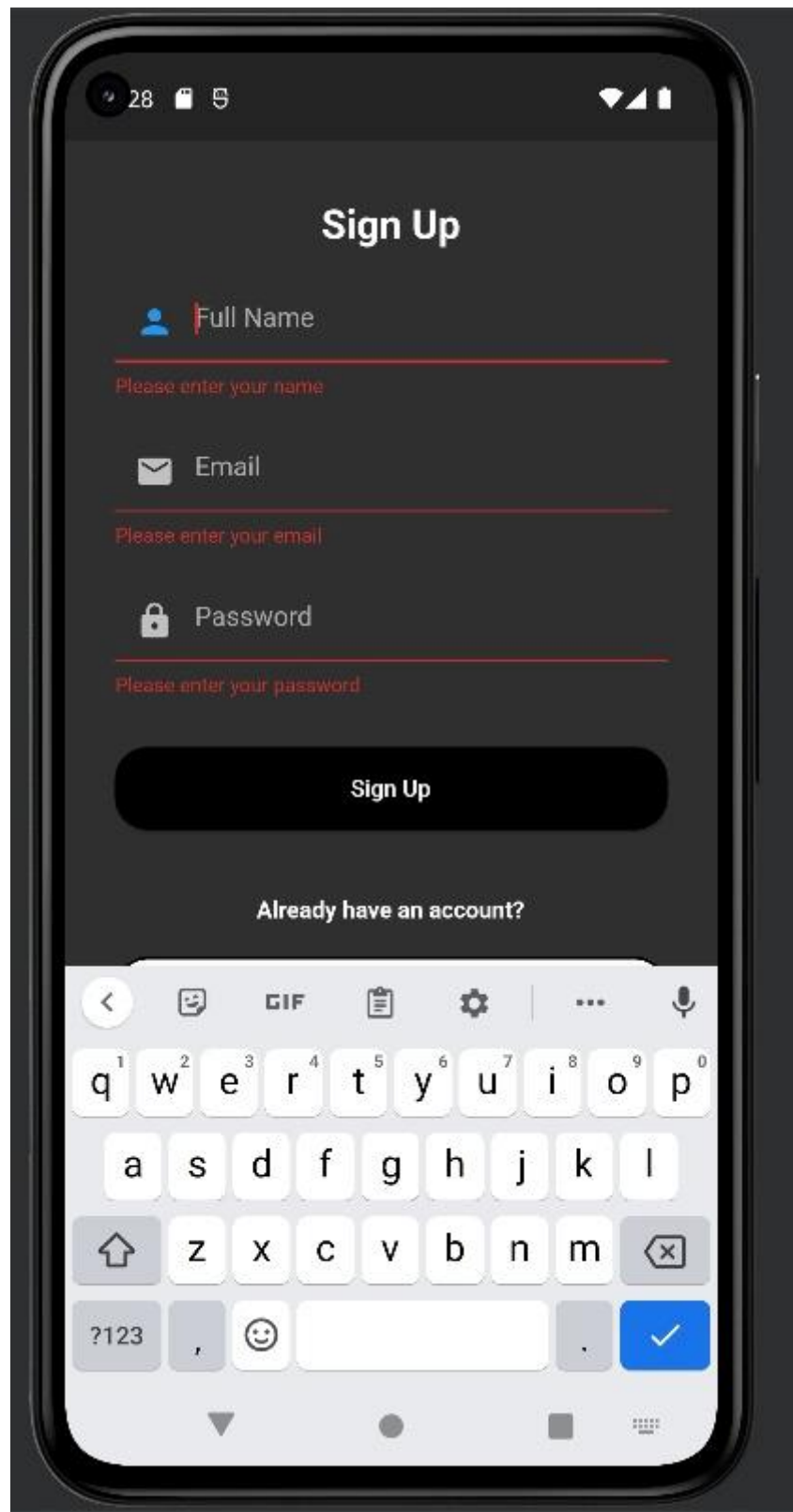


Image8: This page constraints on registering for the user.



Image9: This page is forgetting password page for the users.

# Firestore

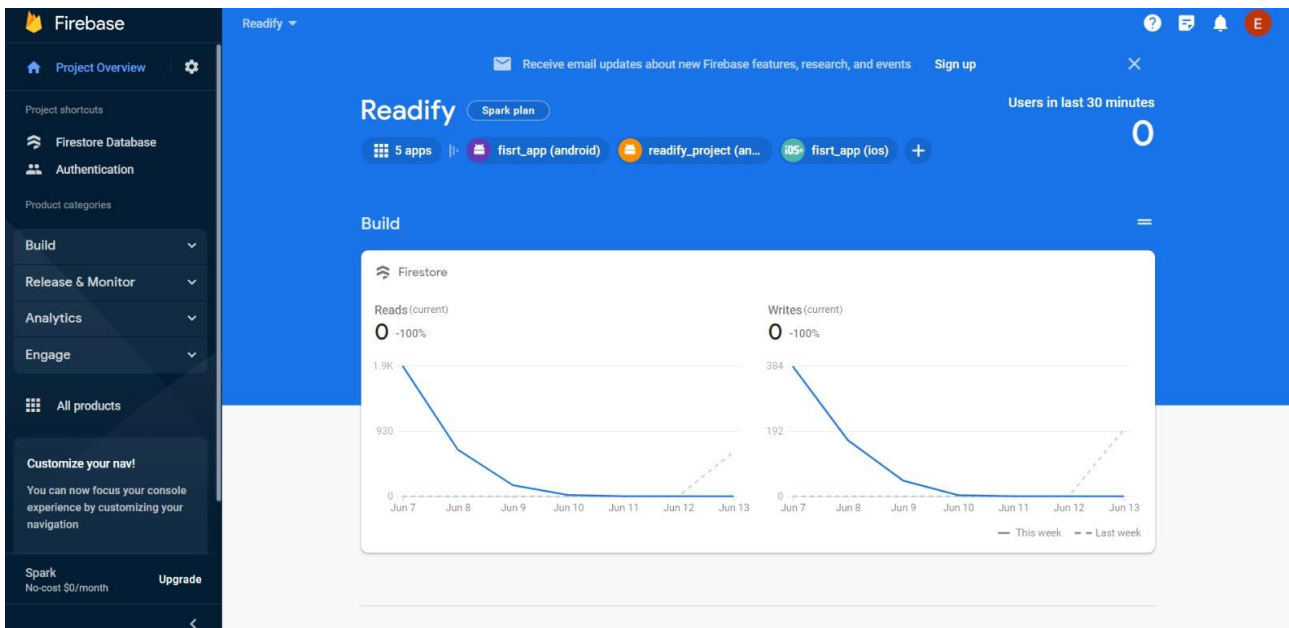


Image10: Readify Firebase project.

The screenshot shows a table of users in the Firestore database. The table has five columns: Identifier, Providers, Created, Signed In, and User UID. There are five rows of user data. The table is part of a larger interface with a search bar and an 'Add user' button.

Identifier	Providers	Created	Signed In	User UID
gokberk_1985@hotmail.com	✉	Jun 14, 2023		kRqGTpnxCCcjyOCxh9itlosW6oZ2
denizdmbk19@gmail.com	✉	Jun 10, 2023	Jun 10, 2023	oxgciuTBkAfWhZ0iY3H69JRM5yn1
bekmezciemre251@gmail.com	✉	Jun 10, 2023	Jun 10, 2023	Svfe0qOAVtXGcNY3lvSNLSE6sFE2
tom@gmail.com	✉	Jun 8, 2023	Jun 8, 2023	3tcW1icoYZXSYAZfCbW39Jes3TE3
daniel@gmail.com	✉	Jun 8, 2023	Jun 9, 2023	ZBnUxo0c0BUwzHPKdqrSkZnfFH3

Image11: Readify users in Firestore database.

The screenshot shows the 'bookComments' collection in the Firestore database. The collection is named 'bookComments' and is located under the 'booksComments' collection. The collection contains three documents: 'DFF9AwAAQBAJ', '0728oQEACAAJ', and 'rQm8x8yvv5G4C'. The collection is part of a larger interface with a search bar and an 'Add document' button.

Collection	Document
booksComments	DFF9AwAAQBAJ
booksComments	0728oQEACAAJ
booksComments	rQm8x8yvv5G4C

Image12: bookComments collection in Firestore database.

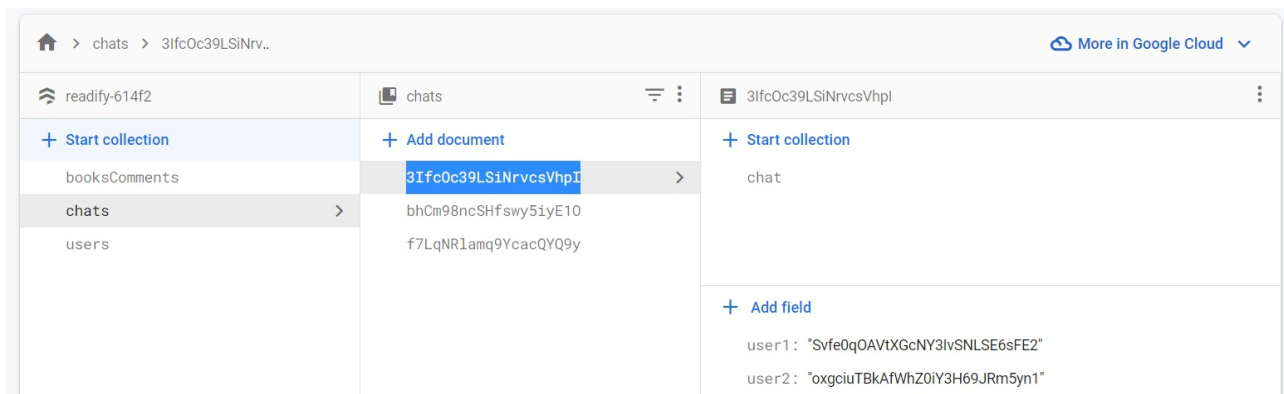


Image13: chats collection in Firestore database.

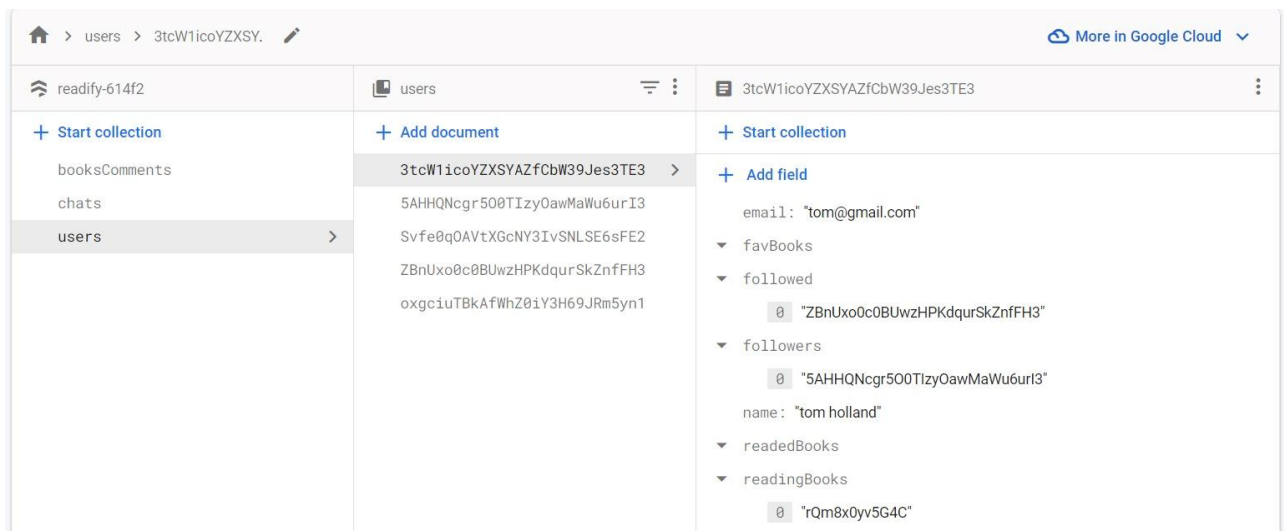


Image14: users collection in Firestore database.

## Home Page

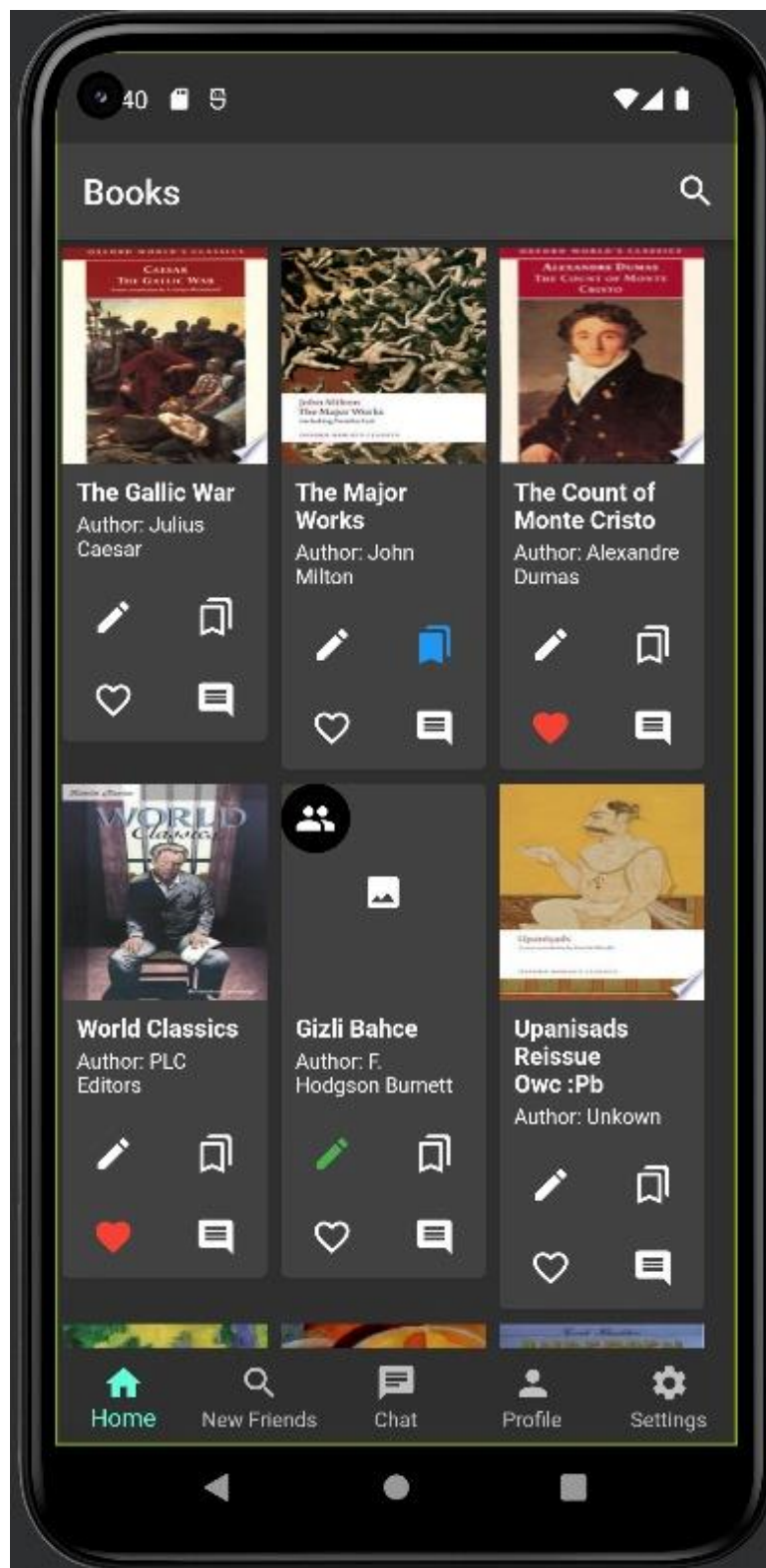


Image15: This page represents the users adding books to their favourites, adding them to the list of books they are read and adding them to the list of books they are reading currently and so that show to user which users have read the book currently.



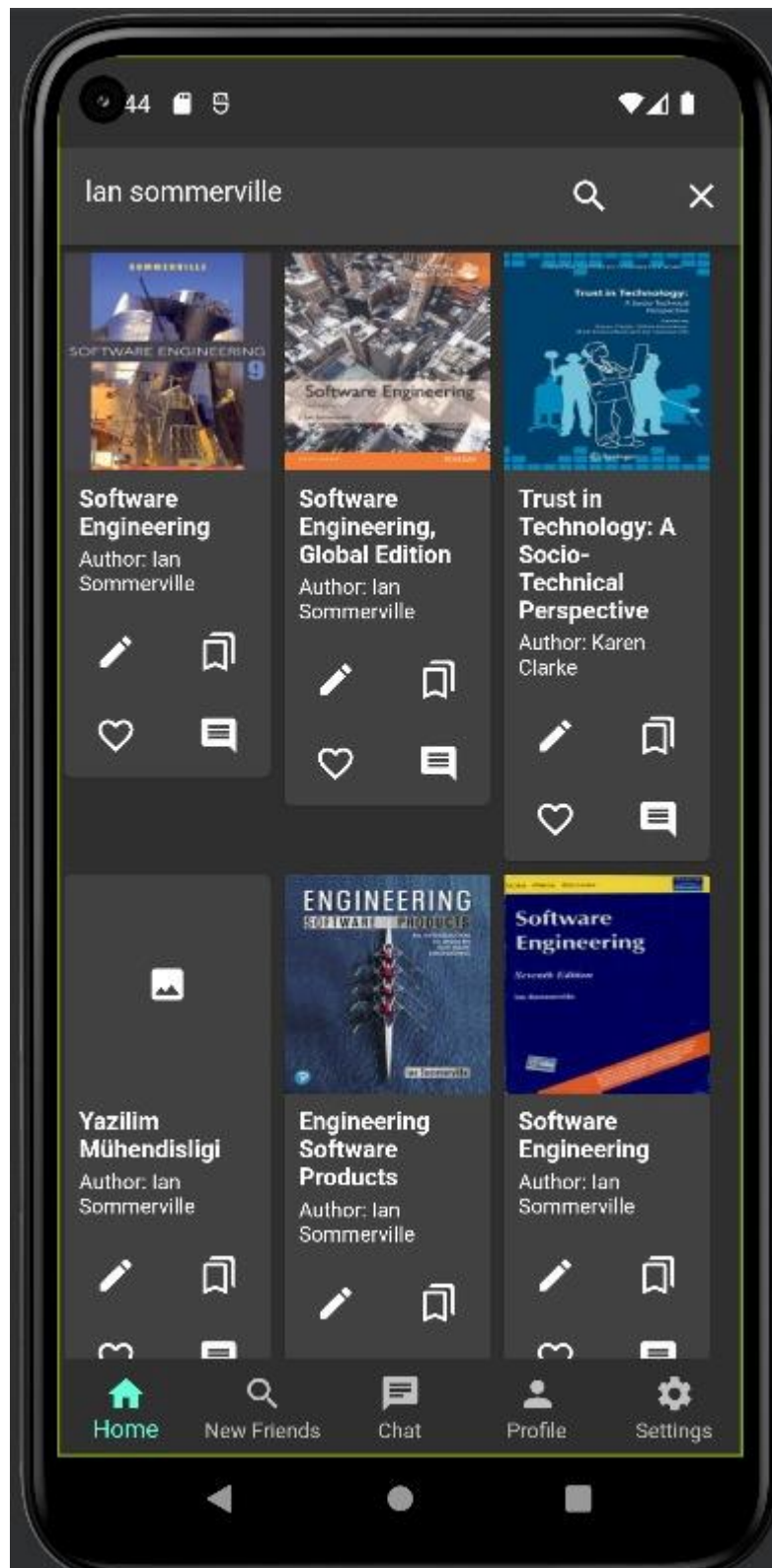


Image16: This page represents after user search a book according to its author through the search button for the users.



Image17: This page shows the option to view and optionally like comments on the selected book.



Image18: This page shows the currently readers of specified book to user.

## Profile Page

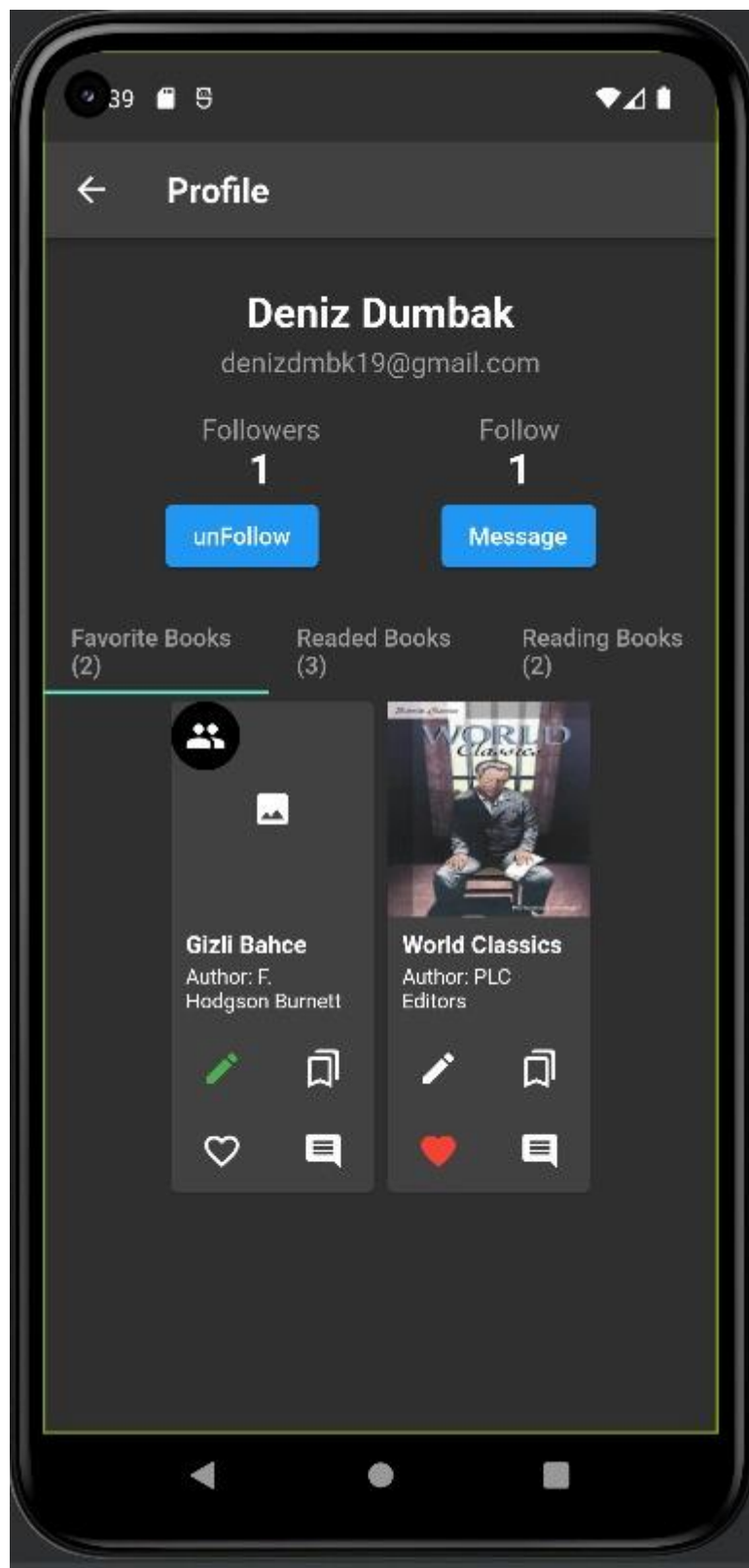


Image19: This page represents Profile page for the other users.

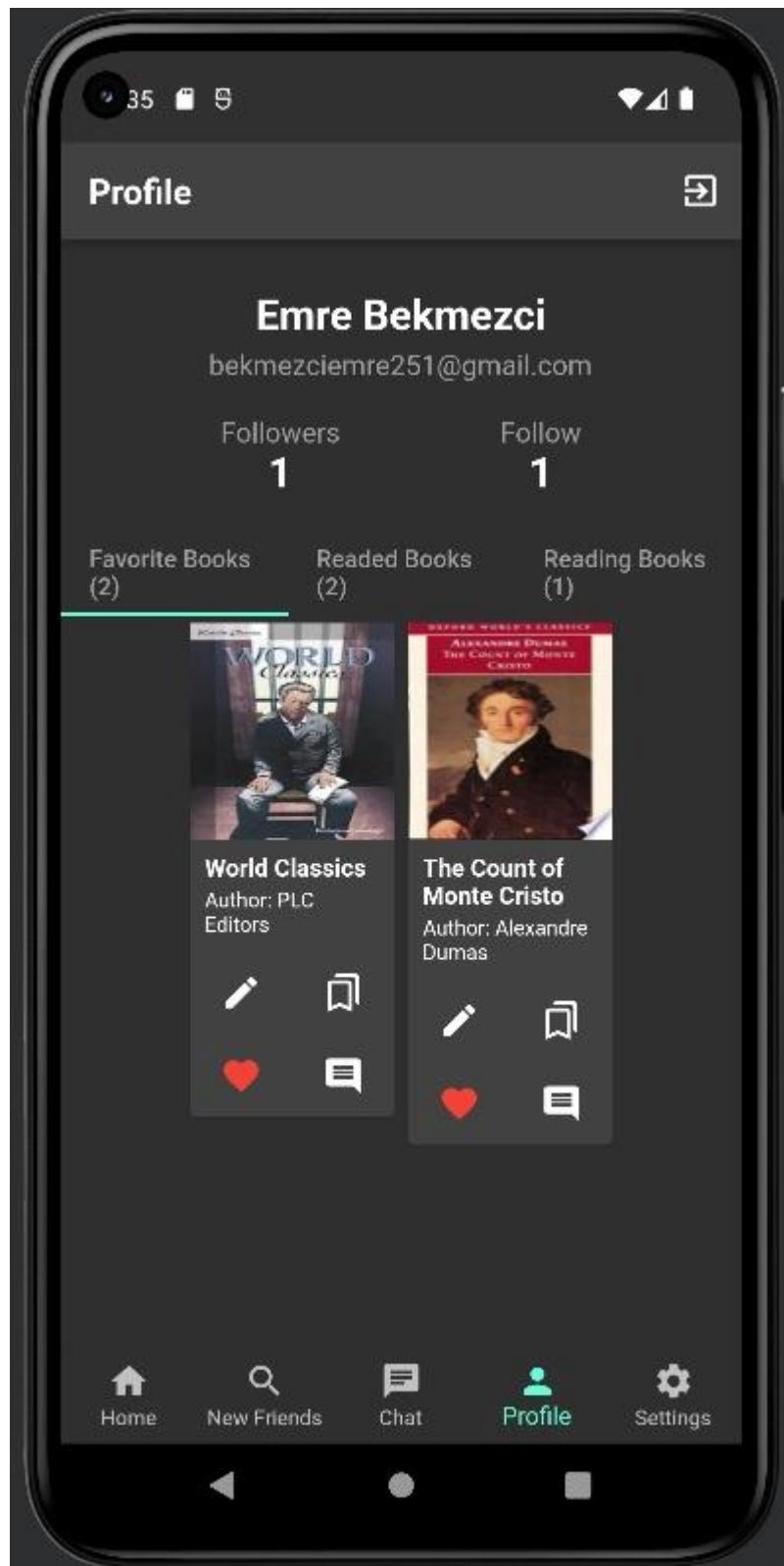


Image20: This page represents Favorite Book list in Profile Page.

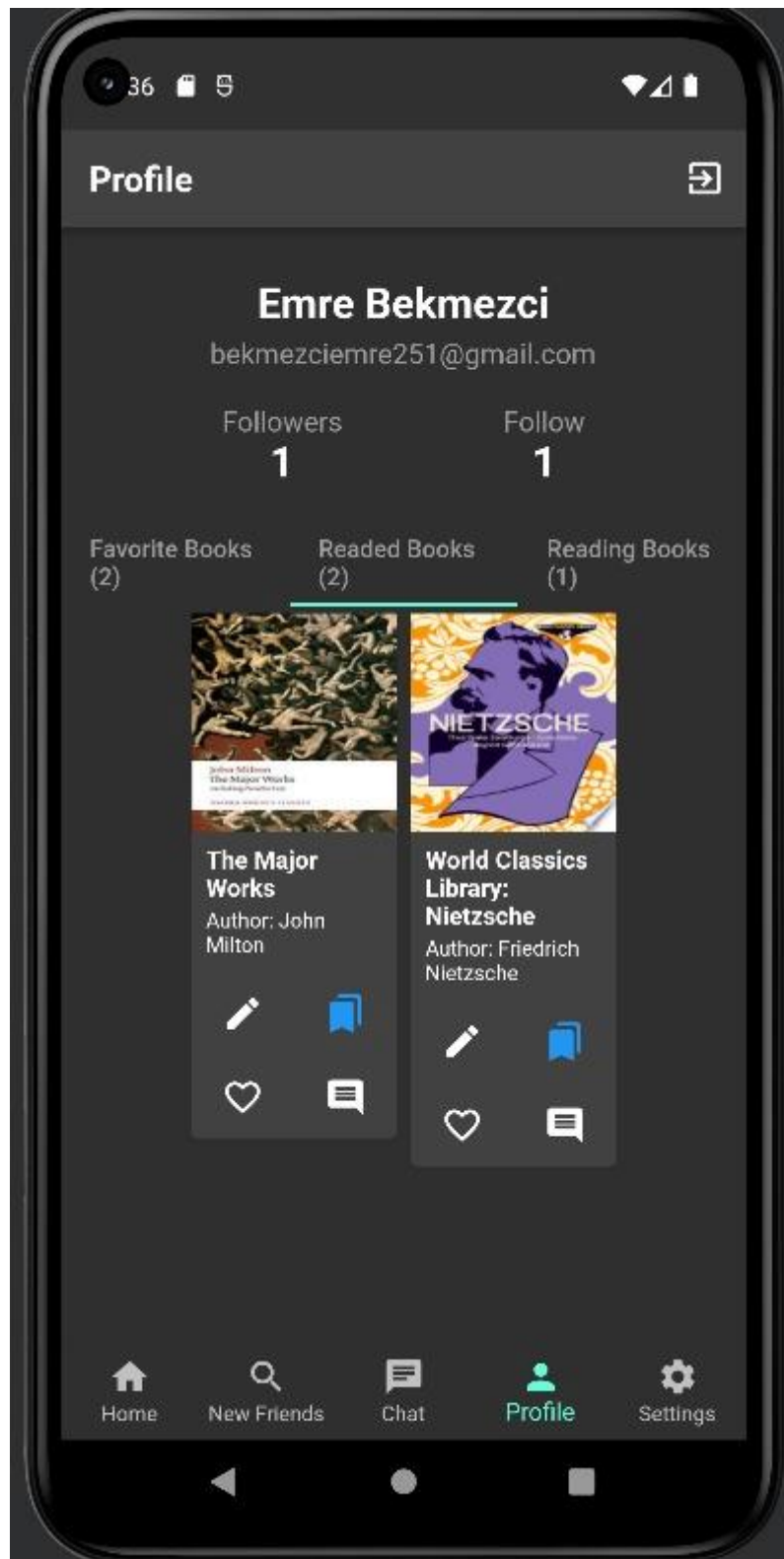


Image21: This page represents Readed Book list in Profile Page.

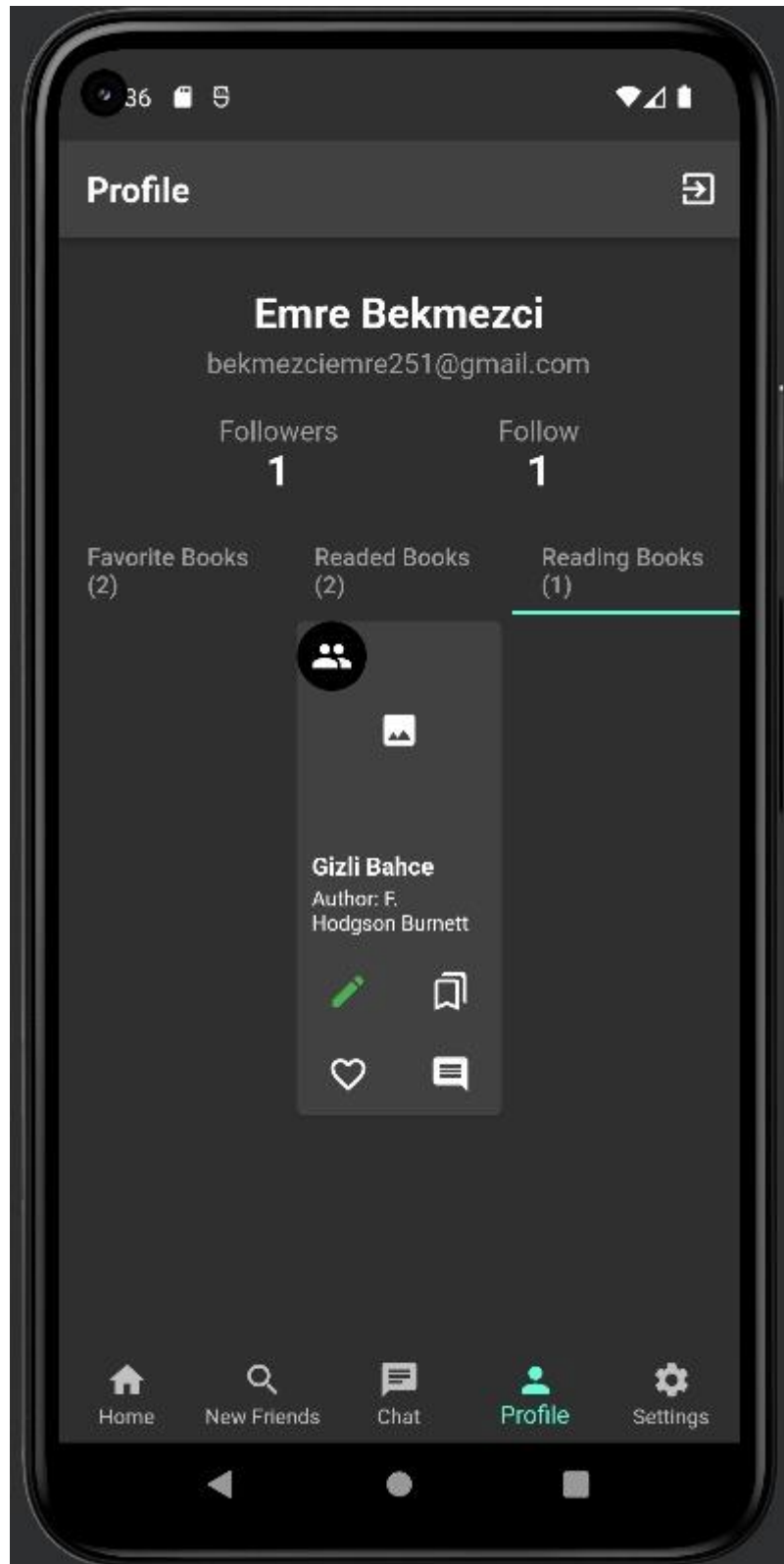


Image22: This page represents Reading Books list in Profile Page.

## New Friends Page

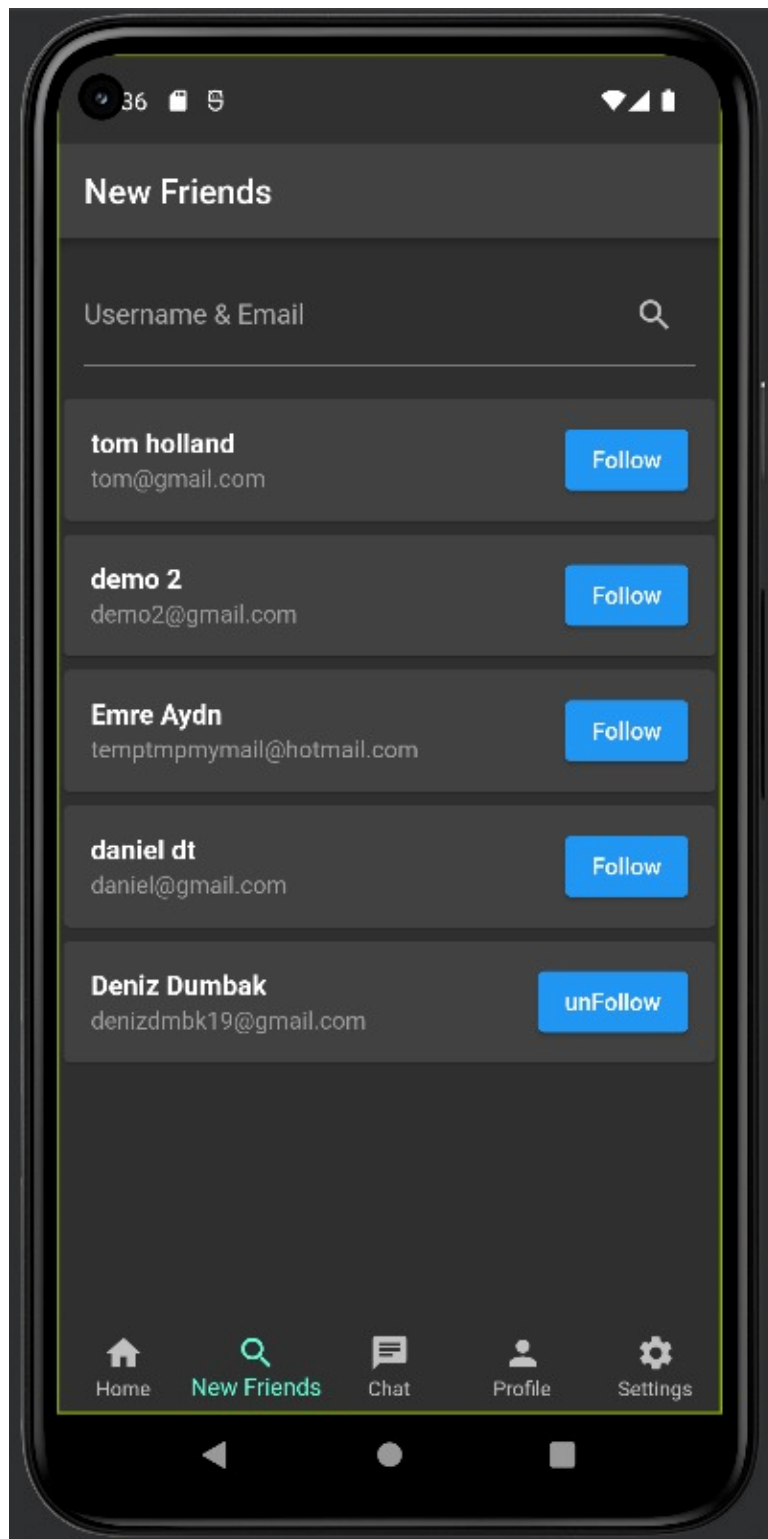


Image23: This page shows all users registered in the system and Follow or Unfollow features of user in New Friends Page.



## Chat Page



Image24: This Page shows chats with other users along with time information.



Image25: This Page shows the chat screen with specified user.

## Settings Page



Image26: This Page shows the Settings screen to change the language from Turkish to English or from English to Turkish.

## 9. Implementations

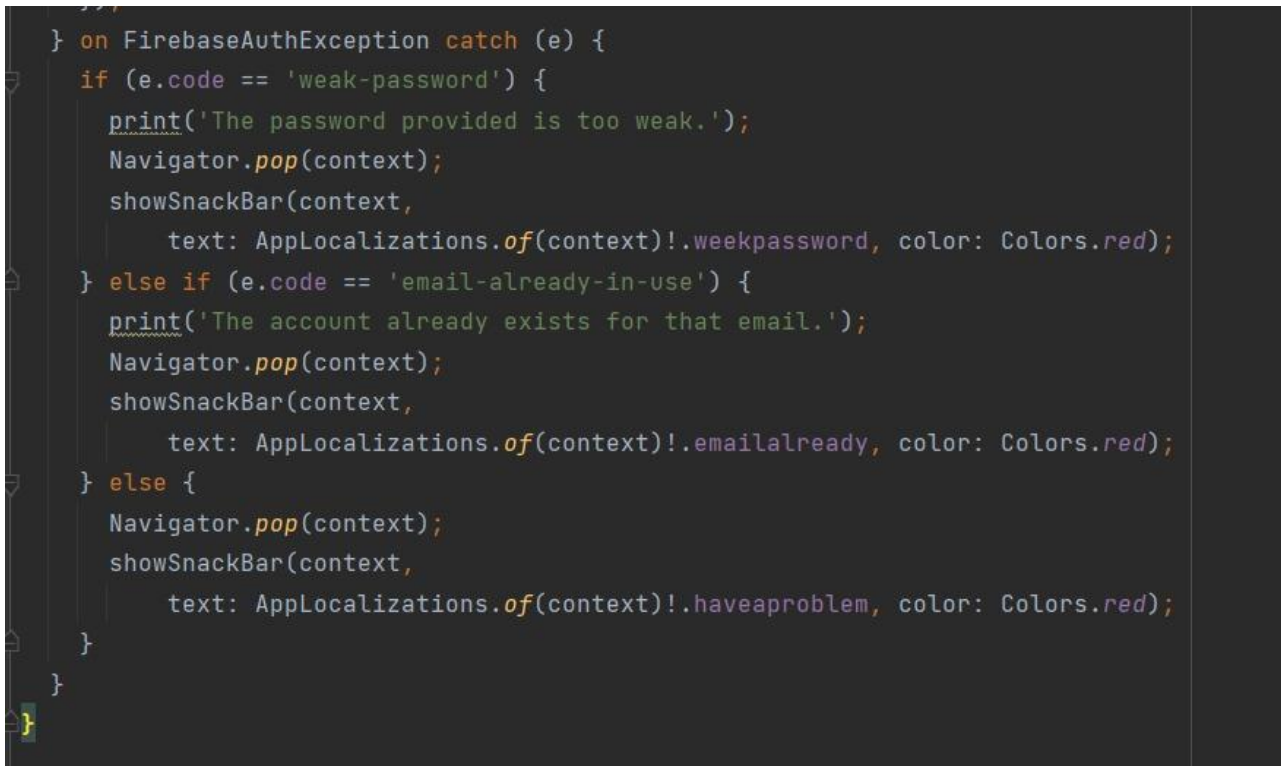
```
registerControl(context,
  {required String name,
   required String email,
   required String password}) async {
  loadingWidget(context);
  try {
    final credential = await FirebaseAuth.instance
      .createUserWithEmailAndPassword(
        email: email,
        password: password,
      )
      .then((value) async {
        await FirebaseAuth.instance.currentUser?.updateDisplayName(name);
        await FirebaseFirestore.instance
          .collection('users')
          .doc(value.user!.uid)
          .set({
            'name': name,
            'email': email,
            'favBooks': [],
            'readedBooks': [],
            'readingBooks': [],
            'followed': [],
            'followers': [],
          });
      }).then((value) {
        Navigator.pushAndRemoveUntil(
          context,
          MaterialPageRoute(builder: (context) => PageNavigationBottom()),
          (route) => false);
      });
  };
```

*figure 9. 1*

The function `registerControl` takes four arguments: `context` (required), `name` (required), `email` (required), and `password` (required). It is marked as `async` since it uses asynchronous operations. `loadingWidget(context)` is likely a function that displays a loading indicator or widget on the screen to indicate that the registration process is in progress. It is called before the registration process starts. The code block inside the `try` statement is where the registration process takes place. `FirebaseAuth.instance.createUserWithEmailAndPassword` is a method provided by the Firebase Authentication library. It attempts to create a new user account using the provided email and password. The resulting credential from the registration process is stored for further use. Inside the `then` callback of the `createUserWithEmailAndPassword` method, the user's display name is updated using `FirebaseAuth.instance.currentUser?.updateDisplayName(name)`. It sets the provided name as the display name for the newly created user. The user's data is stored in Firestore using `FirebaseFirestore.instance.collection('users').doc(value.user!.uid).set({...})`. It creates a new

document in the "users" collection with the user's unique ID (value.user!.uid) and sets various fields like name, email, favBooks, readedBooks, readingBooks, followed, and followers. If the Firestore data storage operation is successful, Navigator.pushAndRemoveUntil is called to navigate to the PageNavigationBottom screen, removing all previous routes. It effectively takes the user to the main screen after successful registration. If any errors occur during the registration process (e.g., network issues, invalid input, etc.), they can be caught in the catch block, allowing for error handling.

```
    } on FirebaseAuthException catch (e) {
      if (e.code == 'weak-password') {
        print('The password provided is too weak.');
```



```
        Navigator.pop(context);
        showSnackBar(context,
          text: AppLocalizations.of(context)!.weekpassword, color: Colors.red);
      } else if (e.code == 'email-already-in-use') {
        print('The account already exists for that email.');
```

```
        Navigator.pop(context);
        showSnackBar(context,
          text: AppLocalizations.of(context)!.emailalready, color: Colors.red);
      } else {
        Navigator.pop(context);
        showSnackBar(context,
          text: AppLocalizations.of(context)!.haveaproblem, color: Colors.red);
      }
    }
  }
```

*figure 9. 2*

The code is wrapped in a try-catch block specifically for handling exceptions of type FirebaseAuthException. This means it catches exceptions related to Firebase Authentication operations. If an exception of type FirebaseAuthException is caught, the code inside the catch block is executed. The if-else statements check the e.code property of the caught exception to determine the specific error that occurred. If the exception's code is 'weak-password', it means the provided password is considered weak. In this case, a message is printed to the console (print('The password provided is too weak.')), the current screen is popped from the navigation stack (Navigator.pop(context)), and a Snackbar is shown to the user with a localized message (showSnackBar(context, text: AppLocalizations.of(context)!.weekpassword, color: Colors.red)). If the exception's code is 'email-already-in-use', it indicates that the email address provided during

registration is already associated with an existing account. Similar to the previous case, it prints a message to the console, pops the current screen from the navigation stack, and shows a SnackBar with a localized message indicating that the email is already in use. If the exception does not match any of the predefined codes, it means there was a different or unexpected error. In this case, the current screen is popped from the navigation stack, and a SnackBar with a generic error message is displayed to the user.

```
loginControl(context, {required String email, required String password}) async {  
  loadingWidget(context);  
  try {  
    final credential = await FirebaseAuth.instance  
      .signInWithEmailAndPassword(email: email, password: password)  
      .then((value) {  
        Navigator.pushAndRemoveUntil(  
          context,  
          MaterialPageRoute(builder: (context) => PageNavigationBottom()),  
          (route) => false);  
      });  
  } on FirebaseAuthException catch (e) {  
    if (e.code == 'user-not-found') {  
      print('No user found for that email.');
```

```
      Navigator.pop(context);  
      showSnackBar(context,  
        text: AppLocalizations.of(context)!.usernotfound, color: Colors.red);  
    } else if (e.code == 'wrong-password') {  
      print('Wrong password provided for that user.');
```

```
      Navigator.pop(context);  
      showSnackBar(context,  
        text: AppLocalizations.of(context)!.wrongpassword, color: Colors.red);  
    } else {  
      Navigator.pop(context);  
      showSnackBar(context,  
        text: AppLocalizations.of(context)!.haveaproblem, color: Colors.red);  
    }  
  }  
}
```

figure 9. 3

The function loginControl takes three arguments: context (required), email (required), and password (required). It is marked as async since it uses asynchronous operations. loadingWidget(context) is likely a function that displays a loading indicator or widget on the screen to indicate that the login process is in progress. It is called before the login process starts. The code block inside the try statement is where the login process takes place.

`FirebaseAuth.instance.signInWithEmailAndPassword` is a method provided by the Firebase Authentication library. It attempts to sign in the user with the provided email and password. The resulting credential from the login process is stored for further use. Inside the then callback of the `signInWithEmailAndPassword` method, the user login is considered successful. It navigates to the main screen using `Navigator.pushAndRemoveUntil`, which removes all previous routes and replaces them with the `PageNavigationBottom` screen. If an exception of type `FirebaseAuthException` is caught, the code inside the catch block is executed to handle specific login exceptions. The if-else statements check the `e.code` property of the caught exception to determine the specific error that occurred. If the exception's code is `'user-not-found'`, it means no user account was found with the provided email. In this case, a message is printed to the console, the current screen is popped from the navigation stack, and a `Snackbar` is shown to the user with a localized message indicating that no user was found. If the exception's code is `'wrong-password'`, it indicates that the provided password is incorrect for the given email. Similar to the previous case, it prints a message to the console, pops the current screen from the navigation stack, and shows a `Snackbar` with a localized message indicating that the password is incorrect. If the exception does not match any of the predefined codes, it means there was a different or unexpected error. In this case, the current screen is popped from the navigation stack, and a `Snackbar` with a generic error message is displayed to the user.

```

forgotPasswordControl(context, {required String email}) async {
  loadingWidget(context);
  try {
    await FirebaseAuth.instance
      .sendPasswordResetEmail(email: email)
      .then((value) {
        Navigator.pop(context);
        Navigator.pop(context);
        showSnackBar(context,
          text: AppLocalizations.of(context)!.forgotemailsend,
          color: Colors.green);
      });
  } catch (e) {
    Navigator.pop(context);
    showSnackBar(context,
      text: AppLocalizations.of(context)!.haveaproblem, color: Colors.red);
  }
}
}

```

figure 9. 4

The function `forgotPasswordControl` takes two arguments: `context` (required) and `email` (required). It is marked as `async` since it uses asynchronous operations. `loadingWidget(context)` is likely a function that displays a loading indicator or widget on the screen to indicate that the password reset process is in progress. It is called before the password reset process starts. The code block inside the `try` statement is where the password reset process takes place. `FirebaseAuth.instance.sendPasswordResetEmail` is a method provided by the Firebase Authentication library. It sends a password reset email to the provided email address. The `await` keyword is used to wait for the password reset email to be sent before proceeding. Inside the `then` callback of the `sendPasswordResetEmail` method, it is assumed that the password reset email was sent successfully. The code handles the UI feedback by popping the current and previous screens from the navigation stack using `Navigator.pop(context)`, effectively going back to the previous screen before the "forgot password" screen. It then shows a `SnackBar` with a localized message indicating that the password reset email was sent successfully. If any errors occur during the password reset process, they are caught in the `catch` block. The code inside the `catch` block is executed to handle the errors. In case of an error, the loading indicator is removed by popping the current screen from the navigation stack using `Navigator.pop(context)`. Then, a `SnackBar` with a generic error message is displayed to the user.



```

factory BookModel.fromJson(Map<String, dynamic> json) => BookModel(
    kind: json["kind"],
    totalItems: json["totalItems"],
    items: List<Item>.from(json["items"].map((x) => Item.fromJson(x))),
); // BookModel

Map<String, dynamic> toJson() => {
    "kind": kind,
    "totalItems": totalItems,
    "items": List<dynamic>.from(items.map((x) => x.toJson())),
};

```

*figure 9.5*

The `fromJson` factory constructor takes a `Map<String, dynamic>` named `json` as input and returns a new instance of the `BookModel` class. Inside the factory constructor, the values from the `json` map are extracted using their corresponding keys. `json["kind"]` retrieves the value associated with the key "kind" and assigns it to the `kind` property of the `BookModel`. `json["totalItems"]` retrieves the value associated with the key "totalItems" and assigns it to the `totalItems` property of the `BookModel`. `json["items"].map((x) => Item.fromJson(x))` retrieves the value associated with the key "items" from the `json` map, and maps it to a list of `Item` objects using the `fromJson` method of the `Item` class. The resulting list is assigned to the `items` property of the `BookModel`. The `toJson` method converts the `BookModel` object to a JSON format and returns a `Map<String, dynamic>` representation of the object. Inside the `toJson` method, the properties of the `BookModel` object are assigned to the corresponding keys in the JSON map. `kind` is assigned to the key "kind", `totalItems` is assigned to the key "totalItems", and `items.map((x) => x.toJson())` converts the list of `Item` objects to JSON format using the `toJson` method of the `Item` class. The resulting list is assigned to the key "items".

```

factory Item.fromJson(Map<String, dynamic> json) => Item(
  kind: kindValues.map[json["kind"]]!,
  id: json["id"],
  etag: json["etag"],
  selfLink: json["selfLink"],
  volumeInfo: VolumeInfo.fromJson(json["volumeInfo"]),
  saleInfo: SaleInfo.fromJson(json["saleInfo"]),
  accessInfo: AccessInfo.fromJson(json["accessInfo"]),
  searchInfo: json["searchInfo"] == null
    ? null
    : SearchInfo.fromJson(json["searchInfo"]),
); // Item

Map<String, dynamic> toJson() => {
  "kind": kindValues.reverse[kind],
  "id": id,
  "etag": etag,
  "selfLink": selfLink,
  "volumeInfo": volumeInfo.toJson(),
  "saleInfo": saleInfo.toJson(),
  "accessInfo": accessInfo.toJson(),
  "searchInfo": searchInfo?.toJson(),
};

```

figure 9. 6

The fromJson factory constructor takes a Map<String, dynamic> named json as input and returns a new instance of the Item class. Inside the factory constructor, the values from the json map are extracted using their corresponding keys. kindValues.map[json["kind"]]! retrieves the value associated with the key "kind" from the json map and maps it to the Kind enum using the kindValues map. The resulting value is assigned to the kind property of the Item object. json["id"], json["etag"], and json["selfLink"] retrieve the values associated with the respective keys and assign them to the corresponding properties of the Item object. VolumeInfo.fromJson(json["volumeInfo"]), SaleInfo.fromJson(json["saleInfo"]), AccessInfo.fromJson(json["accessInfo"]), and SearchInfo.fromJson(json["searchInfo"]) convert the JSON data for the respective nested objects (VolumeInfo, SaleInfo, AccessInfo, and SearchInfo) into Dart objects using their respective fromJson methods. The resulting objects are assigned to the corresponding properties of the Item object. The toJson method converts the Item object to a JSON format and returns a Map<String, dynamic>

representation of the object. Inside the toJson method, the properties of the Item object are assigned to the corresponding keys in the JSON map. kindValues.reverse[kind] retrieves the key associated with the kind property value from the kindValues map. The resulting value is assigned to the key "kind" in the JSON map. The remaining properties (id, etag, selfLink, volumeInfo, saleInfo, accessInfo, and searchInfo) are directly assigned to their corresponding keys in the JSON map. The nested objects are converted to JSON format using their respective toJson methods.

```
factory SaleInfo.fromJson(Map<String, dynamic> json) => SaleInfo(  
    country: countryValues.map[json["country"]],  
    saleability: saleabilityValues.map[json["saleability"]],  
    isEbook: json["isEbook"],  
);  
  
Map<String, dynamic> toJson() => {  
    "country": countryValues.reverse[country],  
    "saleability": saleabilityValues.reverse[saleability],  
    "isEbook": isEbook,  
};  
}
```

*figure 9. 7*

The fromJson factory constructor takes a Map<String, dynamic> named json as input and returns a new instance of the SaleInfo class. Inside the factory constructor, the values from the json map are extracted using their corresponding keys. countryValues.map[json["country"]] retrieves the value associated with the key "country" from the json map and maps it to an enum value using the map property of the countryValues enum. The resulting value is assigned to the country property of the SaleInfo. saleabilityValues.map[json["saleability"]] retrieves the value associated with the key "saleability" from the json map and maps it to an enum value using the map property of the saleabilityValues enum. The resulting value is assigned to the saleability property of the SaleInfo. json["isEbook"] retrieves the value associated with the key "isEbook" from the json map and assigns it to the isEbook property of the SaleInfo. The toJson method converts the SaleInfo object to a JSON format and returns a Map<String, dynamic> representation of the object. Inside the toJson method, the properties of the SaleInfo object are assigned to the corresponding keys in the JSON map. countryValues.reverse[country] retrieves the enum value associated with the country property and maps it back to its original value using the reverse property of the countryValues enum. The resulting value is assigned to the key "country". saleabilityValues.reverse[saleability] retrieves the enum value associated with the saleability property and maps it back to its original value using the reverse

property of the `saleabilityValues` enum. The resulting value is assigned to the key "saleability". The `isEbook` property is assigned to the key "isEbook".

```
factory SearchInfo.fromJson(Map<String, dynamic> json) => SearchInfo(  
    textSnippet: json["textSnippet"],  
);  
  
Map<String, dynamic> toJson() => {  
    "textSnippet": textSnippet,  
};  
}
```

*figure 9. 8*

The `fromJson` factory constructor takes a `Map<String, dynamic>` named `json` as input and returns a new instance of the `SearchInfo` class. Inside the factory constructor, the value from the `json` map is extracted using the key "textSnippet". `json["textSnippet"]` retrieves the value associated with the key "textSnippet" from the `json` map and assigns it to the `textSnippet` property of the `SearchInfo`. The `toJson` method converts the `SearchInfo` object to a JSON format and returns a `Map<String, dynamic>` representation of the object. Inside the `toJson` method, the `textSnippet` property of the `SearchInfo` object is assigned to the key "textSnippet" in the JSON map.

```

factory VolumeInfo.fromJson(Map<String, dynamic> json) => VolumeInfo(
  title: json["title"],
  authors: List<String>.from(json["authors"].map((x) => x)),
  publishedDate: json["publishedDate"],
  description: json["description"],
  industryIdentifiers: List<IndustryIdentifier>.from(
    json["industryIdentifiers"]
      .map((x) => IndustryIdentifier.fromJson(x))), // List.from
  readingModes: ReadingModes.fromJson(json["readingModes"]),
  pageCount: json["pageCount"],
  printType: printTypeValues.map[json["printType"]]!,
  categories: json["categories"] == null
    ? []
    : List<String>.from(json["categories"]!.map((x) => x)),
  maturityRating: maturityRatingValues.map[json["maturityRating"]]!,
  allowAnonLogging: json["allowAnonLogging"],
  contentVersion: contentVersionValues.map[json["contentVersion"]]!,
  imageLinks: json["imageLinks"] == null
    ? null
    : ImageLinks.fromJson(json["imageLinks"]),
  language: languageValues.map[json["language"]]!,
  previewLink: json["previewLink"],
  infoLink: json["infoLink"],
  canonicalVolumeLink: json["canonicalVolumeLink"],
  publisher: json["publisher"],
  panelizationSummary: json["panelizationSummary"] == null
    ? null
    : PanelizationSummary.fromJson(json["panelizationSummary"]),
  subtitle: json["subtitle"],
); // VolumeInfo

```

figure 9.9

The `fromJson` factory constructor takes a `Map<String, dynamic>` named `json` as input and returns a new instance of the `VolumeInfo` class. Inside the factory constructor, the values from the `json` map are extracted using their corresponding keys. The properties of the `VolumeInfo` object are assigned values based on the extracted data from the `json` map. Some properties, such as `authors`, `industryIdentifiers`, `readingModes`, `categories`, `imageLinks`, and `panelizationSummary`, require additional conversion from JSON objects to Dart objects. These conversions are performed using their respective `fromJson` methods. The `toJson` method converts the `VolumeInfo` object to a JSON format and returns a `Map<String, dynamic>` representation of the object. Inside the `toJson` method, the properties of the `VolumeInfo` object are assigned to the corresponding keys in the JSON map. Some properties, such as `authors`, `industryIdentifiers`, `readingModes`, `categories`, `imageLinks`, and `panelizationSummary`, require additional conversion from Dart objects to JSON objects. These conversions are performed using their respective `toJson` methods.

```

factory ImageLinks.fromJson(Map<String, dynamic> json) => ImageLinks(
    smallThumbnail: json["smallThumbnail"],
    thumbnail: json["thumbnail"],
);

Map<String, dynamic> toJson() => {
    "smallThumbnail": smallThumbnail,
    "thumbnail": thumbnail,
};

```

figure 9. 10

The fromJson factory constructor takes a Map<String, dynamic> named json as input and returns a new instance of the ImageLinks class. Inside the factory constructor, the values from the json map are extracted using their corresponding keys. The properties of the ImageLinks object, namely smallThumbnail and thumbnail, are assigned values based on the extracted data from the json map. The toJson method converts the ImageLinks object to a JSON format and returns a Map<String, dynamic> representation of the object. Inside the toJson method, the properties of the ImageLinks object are assigned to the corresponding keys in the JSON map.

```

factory IndustryIdentifier.fromJson(Map<String, dynamic> json) =>
    IndustryIdentifier(
        type: typeValues.map[json["type"]!],
        identifier: json["identifier"],
    );

Map<String, dynamic> toJson() => {
    "type": typeValues.reverse[type],
    "identifier": identifier,
};

```

figure 9. 11

The fromJson factory constructor takes a Map<String, dynamic> named json as input and returns a new instance of the IndustryIdentifier class. Inside the factory constructor, the values from the json map are extracted using their corresponding keys. The properties of the IndustryIdentifier object, namely type and identifier, are assigned values based on the extracted data from the json map. The type value is retrieved using the map getter of the typeValues enum, which maps the string value from the json map to the corresponding enum value. The toJson method converts the IndustryIdentifier object to a JSON format and returns a Map<String, dynamic> representation of the object. Inside the toJson method, the properties of the IndustryIdentifier object are assigned to the



corresponding keys in the JSON map. The type value is retrieved using the reverse getter of the `typeValues` enum, which maps the enum value back to its string representation.

```
factory ReadingModes.fromJson(Map<String, dynamic> json) => ReadingModes(  
    text: json["text"],  
    image: json["image"],  
);  
  
Map<String, dynamic> toJson() => {  
    "text": text,  
    "image": image,  
};
```

*figure 9.12*

The `fromJson` factory constructor takes a `Map<String, dynamic>` named `json` as input and returns a new instance of the `ReadingModes` class. Inside the factory constructor, the values from the `json` map are extracted using their corresponding keys. The properties of the `ReadingModes` object, namely `text` and `image`, are assigned values based on the extracted data from the `json` map. The `toJson` method converts the `ReadingModes` object to a JSON format and returns a `Map<String, dynamic>` representation of the object. Inside the `toJson` method, the properties of the `ReadingModes` object are assigned to the corresponding keys in the JSON map.

```

void main() {
  testWidgets('Counter increments smoke test', (WidgetTester tester) async {
    // Build our app and trigger a frame.
    await tester.pumpWidget(const MyApp());

    // Verify that our counter starts at 0.
    expect(find.text('0'), findsOneWidget);
    expect(find.text('1'), findsNothing);

    // Tap the '+' icon and trigger a frame.
    await tester.tap(find.byIcon(Icons.add));
    await tester.pump();

    // Verify that our counter has incremented.
    expect(find.text('0'), findsNothing);
    expect(find.text('1'), findsOneWidget);
  });
}

```

figure 9. 13

The main function is the entry point of the test. It will be executed when running the test file. The testWidgets function is provided by the Flutter testing framework and is used to define a test case for a widget. It takes a description of the test case as a string and a callback function that contains the actual test logic. Inside the callback function, the test scenario is described step by step. await tester.pumpWidget(const MyApp()) builds the app and triggers a frame update. The MyApp widget is the widget under test. pumpWidget allows the framework to perform an initial build and layout of the widget hierarchy. The expect statements are used to assert the expected behavior of the counter widget. expect(find.text('0'), findsOneWidget) verifies that the text widget displaying '0' is present in the widget tree. expect(find.text('1'), findsNothing) verifies that the text widget displaying '1' is not present in the widget tree. await tester.tap(find.byIcon(Icons.add)) simulates a tap on the widget identified by the Icons.add icon. This triggers the increment action of the counter widget. await tester.pump() triggers a frame update after the tap action. The subsequent expect statements verify the updated state of the counter widget. expect(find.text('0'), findsNothing) asserts that the text widget displaying '0' is no longer present. expect(find.text('1'), findsOneWidget) asserts that the text widget displaying '1' is now present in the widget tree.



```
Future<void> main() async {  
  WidgetsFlutterBinding.ensureInitialized();  
  
  await Firebase.initializeApp();  
  await localeGet();  
  runApp(MyApp());  
}
```

*figure 9. 14*

The main function is the entry point of the Flutter application. It is an asynchronous function that returns a `Future<void>`. `WidgetsFlutterBinding.ensureInitialized()` ensures that the Flutter framework is properly initialized before running the app. It sets up the necessary bindings and initializes the Flutter engine. `await Firebase.initializeApp()` initializes the Firebase services used in the application. This step is important to establish a connection with Firebase and enable Firebase features. `await localeGet()` is an asynchronous function that is likely responsible for retrieving the locale or language settings for the application. It is awaited to ensure that the locale settings are obtained before running the app. `runApp(MyApp())` starts the Flutter application by running the `MyApp` widget as the root widget of the app. The `MyApp` widget is responsible for defining the overall structure and behavior of the application.

```

String? _validateName(String? value) {
    if (value == null || value.isEmpty) {
        return AppLocalizations.of(context)!.fullnameempty;
    }
    return null;
}

String? _validateEmail(String? value) {
    const Pattern pattern = r'^[a-zA-Z0-9.]+@[a-zA-Z0-9]+\.[a-zA-Z]+';
    final RegExp regex = RegExp(pattern.toString());
    if (value == null || value.isEmpty) {
        return AppLocalizations.of(context)!.emailempty;
    } else if (!regex.hasMatch(value)) {
        return AppLocalizations.of(context)!.emailcorrect;
    }
    return null;
}

String? _validatePassword(String? value) {
    if (value == null || value.isEmpty) {
        return AppLocalizations.of(context)!.passwordempty;
    } else if (value.length < 6) {
        return AppLocalizations.of(context)!.passwordcorrect;
    }
    return null;
}

```

figure 9. 15

validateName is a method that takes a String? value as input and returns a String? as the validation result. It checks if the value is null or empty. If it is, it returns the localized error message for an empty full name field. \_validateEmail is a method that takes a String? value as input and returns a String? as the validation result. It first defines a regular expression pattern to validate email format. It then checks if the value is null or empty. If it is, it returns the localized error message for an empty email field. If the value is not empty, it checks if it matches the email pattern. If it doesn't, it returns the localized error message for an incorrect email format. \_validatePassword is a method that takes a String? value as input and returns a String? as the validation result. It checks if the value is null or empty. If it is, it returns the localized error message for an empty password field. If the value is not

empty, it checks if its length is less than 6 characters. If it is, it returns the localized error message for an incorrect password length.

```
if (_formKey.currentState!.validate()) {  
  forgotPasswordControl(context, email: email.text);  
}
```

figure 9. 16

The given code snippet checks if the current state of the form, represented by the `_formKey.currentState`, is valid using the `validate()` method. If the form is valid, it proceeds to call the `forgotPasswordControl` function, passing the context and the value of the email text field obtained from `email.text` as arguments.

```
if (comment.text.isNotEmpty) {  
  FirebaseFirestore.instance  
    .collection("booksComments")  
    .doc(widget.id)  
    .collection('comments')  
    .add({  
      'likedUsers': [],  
      'senderComment': comment.text,  
      'senderName':  
        FirebaseAuth.instance.currentUser!.displayName,  
      'senderid': FirebaseAuth.instance.currentUser!.uid,  
      'time': DateTime.now()  
    }).then((value) {  
      setState(() {  
        comment.text = "";  
      });  
    });  
}
```

figure 9. 17

The provided code snippet is an if statement that checks if the `comment.text` is not empty. If the condition is true, the code proceeds to execute the following actions: It accesses the `FirebaseFirestore` instance to interact with the Firestore database. It specifies the collection path as `"booksComments"` and then creates a document with the `widget.id` as its identifier. It further specifies

a subcollection path called "comments". It uses the add method to add a new document to the "comments" subcollection. This document contains the following fields:

'likedUsers': an empty list (presumably to store user IDs who liked the comment).

'senderComment': the content of the comment entered by the user.

'senderName': the display name of the currently authenticated user (retrieved from `FirebaseAuth.instance.currentUser!`).

'senderid': the user ID of the currently authenticated user (retrieved from `FirebaseAuth.instance.currentUser!.uid`).

'time': the current date and time, obtained using `DateTime.now()`.

If the document addition is successful, the code calls `setState` to update the state and clears the `comment.text` by setting it to an empty string.

```

List newLikedUsers = likedUsers;

newLikedUsers.remove(
    FirebaseAuth.instance.currentUser!.uid);

FirebaseFirestore.instance
    .collection('booksComments')
    .doc(widget.bookid)
    .collection('comments')
    .doc(document.id)
    .update({'likedUsers': newLikedUsers});
} else {
List newLikedUsers = likedUsers;

newLikedUsers.add(
    FirebaseAuth.instance.currentUser!.uid);
FirebaseFirestore.instance
    .collection('booksComments')
    .doc(widget.bookid)
    .collection('comments')
    .doc(document.id)
    .update({'likedUsers': newLikedUsers});
}

```

figure 9. 18

Initialize a new list `newLikedUsers` and assign it the value of the existing `likedUsers` list. Check if the current user's ID (`FirebaseAuth.instance.currentUser!.uid`) is already present in the `newLikedUsers` list. If it is, it means that the user has already liked the comment and wants to remove their like. Inside the `if` block, remove the current user's ID from the `newLikedUsers` list using the `remove` method. Use the `FirebaseFirestore` instance to access the Firestore database. Specify the collection path as "booksComments". Use the `doc` method to specify the document ID of the comment you want to update, based on `document.id`. Further specify a subcollection path called "comments". Use the `update` method to update the specific document with the new value of `likedUsers` by providing

a map with the field name ("likedUsers") as the key and the updated newlikedUsers list as the value. If the current user's ID is not present in the newlikedUsers list, it means they want to like the comment. Inside the else block, add the current user's ID to the newlikedUsers list using the add method. Use the update method to update the Firestore document with the updated likedUsers list.

```
if (message.text.isNotEmpty) {
  DateTime time = await NTP.now();

  await FirebaseFirestore.instance
    .collection('chats')
    .doc(widget.roomID)
    .collection('chat')
    .add({
      'senderID': FirebaseAuth.instance.currentUser!.uid,
      'senderMsg': message.text,
      'time': time
    });
  setState(() {
    message.text = '';
  });
}
```

*figure 9. 19*

Initialize a new list newlikedUsers and assign it the value of the existing likedUsers list. Check if the current user's ID (FirebaseAuth.instance.currentUser!.uid) is already present in the newlikedUsers list. If it is, it means that the user has already liked the comment and wants to remove their like. Inside the if block, remove the current user's ID from the newlikedUsers list using the remove method. Use the FirebaseFirestore instance to access the Firestore database. Specify the collection path as "booksComments". Use the doc method to specify the document ID of the comment you want to update, based on document.id. Further specify a subcollection path called "comments". Use the update method to update the specific document with the new value of likedUsers by providing a map with the field name ("likedUsers") as the key and the updated newlikedUsers list as the value. If the current user's ID is not present in the newlikedUsers list, it means they want to like the comment.

Inside the else block, add the current user's ID to the newLikedUsers list using the add method. Use the update method to update the Firestore document with the updated likedUsers list.

```
DocumentReference reference = FirebaseFirestore.instance
    .collection('users')
    .doc(FirebaseAuth.instance.currentUser!.uid);
reference.snapshots().listen((querySnapshot) {
    if (mounted) {
        setState(() {
            favBooks = querySnapshot.get("favBooks");
            readedBooks = querySnapshot.get("readedBooks");
            readingBooks = querySnapshot.get("readingBooks");
        });
    }
});
```

*figure 9. 20*

A DocumentReference object named reference is created, pointing to a specific document in the Firestore database. The document is determined based on the current user's ID obtained from FirebaseAuth.instance.currentUser!.uid. The reference.snapshots().listen() method is called, which sets up a listener to receive updates whenever the referenced document changes. Inside the listener, the received querySnapshot is processed. The listener function is triggered whenever the referenced document is modified. If the widget is still mounted (i.e., it has not been disposed of), the setState() method is called to update the widget's state with the new data obtained from the querySnapshot. This ensures that any changes in the database are reflected in the widget. The specific fields (favBooks, readedBooks, readingBooks) are updated with the corresponding values obtained from the querySnapshot.



```

if (readingBooks.contains(widget.id) == true) {
    List newreadingBooks = readingBooks;

    newreadingBooks.remove(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'readingBooks': newreadingBooks});
} else {
    List newreadingBooks = readingBooks;

    newreadingBooks.add(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'readingBooks': newreadingBooks});
}

```

*figure 9. 21*

The condition `readingBooks.contains(widget.id) == true` checks if the `readingBooks` list contains the value of `widget.id`. If the condition evaluates to true, it means that the book ID is already present in the `readingBooks` list. In this case, the following steps are performed: A new list, `newreadingBooks`, is created as a copy of the original `readingBooks` list. The book ID (`widget.id`) is removed from the `newreadingBooks` list using the `remove` method. The updated `newreadingBooks` list is saved back to the Firestore database under the `readingBooks` field for the current user. If the condition evaluates to false, it means that the book ID is not present in the `readingBooks` list. In this case, the following steps are performed: A new list, `newreadingBooks`, is created as a copy of the original `readingBooks` list. The book ID (`widget.id`) is added to the `newreadingBooks` list using the `add` method. The updated `newreadingBooks` list is saved back to the Firestore database under the `readingBooks` field for the current user.



```

if (readedBooks.contains(widget.id) == true) {
    List newreadedBooks = readedBooks;

    newreadedBooks.remove(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'readedBooks': newreadedBooks});
} else {
    List newreadedBooks = readedBooks;

    newreadedBooks.add(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'readedBooks': newreadedBooks});
}

```

*figure 9. 22*

The condition `readedBooks.contains(widget.id) == true` checks if the `readedBooks` list contains the value of `widget.id`. If the condition evaluates to true, it means that the book ID is already present in the `readedBooks` list. In this case, the following steps are performed: A new list, `newreadedBooks`, is created as a copy of the original `readedBooks` list. The book ID (`widget.id`) is removed from the `newreadedBooks` list using the `remove` method. The updated `newreadedBooks` list is saved back to the Firestore database under the `readedBooks` field for the current user. If the condition evaluates to false, it means that the book ID is not present in the `readedBooks` list. In this case, the following steps are performed: A new list, `newreadedBooks`, is created as a copy of the original `readedBooks` list. The book ID (`widget.id`) is added to the `newreadedBooks` list using the `add` method. The updated `newreadedBooks` list is saved back to the Firestore database under the `readedBooks` field for the current user.

```

if (favBooks.contains(widget.id) == true) {
    List newBooks = favBooks;

    newBooks.remove(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'favBooks': newBooks});
} else {
    List newBooks = favBooks;

    newBooks.add(widget.id);

    FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser!.uid)
        .update({'favBooks': newBooks});
}

```

*figure 9. 23*

The condition `favBooks.contains(widget.id) == true` checks if the `favBooks` list contains the value of `widget.id`. If the condition evaluates to true, it means that the book ID is already present in the `favBooks` list. In this case, the following steps are performed: A new list, `newBooks`, is created as a copy of the original `favBooks` list. The book ID (`widget.id`) is removed from the `newBooks` list using the `remove` method. The updated `newBooks` list is saved back to the Firestore database under the `favBooks` field for the current user. If the condition evaluates to false, it means that the book ID is not present in the `favBooks` list. In this case, the following steps are performed: A new list, `newBooks`, is created as a copy of the original `favBooks` list. The book ID (`widget.id`) is added to the `newBooks` list using the `add` method. The updated `newBooks` list is saved back to the Firestore database under the `favBooks` field for the current user.

```

void initState() {
  // TODO: implement initState
  super.initState();
  CollectionReference<Map<String, dynamic>> reference = FirebaseFirestore
    .instance
    .collection('chats')
    .doc(widget.roomid)
    .collection("chat");
  reference
    .orderBy("time", descending: false)
    .snapshots()
    .listen((querySnapshot) {
  if (querySnapshot != null) {
    if (mounted) {
      setState(() {
        Timestamp millis = querySnapshot.docs.last['time'];
        var dt = DateTime.fromMillisecondsSinceEpoch(
          millis.millisecondsSinceEpoch);

```

figure 9. 24

```

    var d24 = DateFormat('dd/MM/yyyy, HH:mm').format(dt);

    time = d24;
    if (FirebaseAuth.instance.currentUser!.uid !=|
      querySnapshot.docs.last['senderID']) {
      lastMessage = 'You: ${querySnapshot.docs.last['senderMsg']}';
    } else {
      lastMessage = querySnapshot.docs.last['senderMsg'];
    }
  });
}
});

FirebaseFirestore.instance
  .collection("users")
  .doc(widget.userID)
  .get()
  .then((value) {
    if (mounted) {
      setState(() {
        userName = value['name'];
        load = false;
      });
    }
  });
}

```

figure 9. 25

It begins by calling the `initState()` method of the superclass using `super.initState()` to ensure that the parent class's `initState()` method is executed. Inside the `initState()` method, a `CollectionReference` object is created for the 'chats' collection under the document with the ID `widget.roomid`. The `snapshots()` method is called on the `CollectionReference` to listen for changes in the collection. The `listen()` method is used to attach a callback function that is executed when there are changes in the collection. The callback function receives a `querySnapshot` parameter containing the latest snapshot of the queried documents. Within the callback function, the `querySnapshot` is checked for nullity. If the `querySnapshot` is not null, the callback proceeds to update the state using `setState()`. The last document in the `querySnapshot` is accessed to retrieve the timestamp and message details. The timestamp is converted to a `DateTime` object and formatted using the `DateFormat` class from the `intl` package to display it in a specific format. The time state variable is updated with the formatted timestamp. Depending on whether the last message was sent by the current user or another user, the `lastMessage` state variable is set accordingly. Next, a `get()` method is called on the 'users' collection with the document ID `widget.userID` to retrieve the user's data. The `then()` method is attached to the `get()` method to handle the result of the asynchronous operation. Inside the `then()` callback, the user's name is retrieved from the value snapshot and stored in the `userName` state variable. Finally, the `load` state variable is set to `false` to indicate that the necessary data has been loaded.

```
Future searchBooks({required String searchWord}) async {  
  final response = await http.get(Uri.parse(  
    'https://www.googleapis.com/books/v1/volumes?q=${searchWord}&maxResults=40'));  
  if (response.statusCode == 200) {  
    String jsonResponse = response.body;  
    Map<String, dynamic> data = json.decode(jsonResponse);  
  
    return data;  
  } else {  
    print('API isteđi başarısız oldu: ${response.statusCode}');  
    throw 'hata';  
  }  
}
```

figure 9. 26

The function takes a required parameter `searchWord`, which represents the search query for books. The API request is made using the `http.get()` method, passing in the API endpoint URL constructed with the `searchWord` parameter and a maximum result limit of 40. The `await` keyword is

used to wait for the response from the API. If the response status code is 200 (indicating a successful request), the response body is extracted as a string using `response.body`. The `json.decode()` function is used to convert the JSON response string into a `Map<String, dynamic>`. The resulting data map is returned from the function. If the response status code is not 200, an error message is printed, and an exception is thrown.

```
Future getBooks() async {
  final response = await http.get(Uri.parse(
    'https://www.googleapis.com/books/v1/volumes?q=world+classics&maxResults=40'));
  if (response.statusCode == 200) {
    String jsonResponse = response.body;
    Map<String, dynamic> data = json.decode(jsonResponse);

    return data;
  } else {
    print('API isteđi başarısız oldu: ${response.statusCode}');
    throw 'hata';
  }
}
```

*figure 9. 27*

The function uses the `http.get()` method to send a GET request to the specified API endpoint. The endpoint URL is constructed using `Uri.parse()` with the base URL and query parameters. The query parameters include the search query "world classics" and a maximum result limit of 40 books. The `await` keyword is used to wait for the response from the API. If the response status code is 200 (indicating a successful request), the response body is extracted as a string using `response.body`. The `json.decode()` function is used to convert the JSON response string into a `Map<String, dynamic>`. The resulting data map is returned from the function. If the response status code is not 200, an error message is printed, and an exception is thrown.

```

class _WhoReadingState extends State<WhoReading> {
  TextEditingController comment = TextEditingController();
  final _formKey = GlobalKey<FormState>();

  List followersData = [];
  List followedData = [];
  @override
  void initState() {
    // TODO: implement initState
    super.initState();

    DocumentReference reference = FirebaseFirestore.instance
      .collection('users')
      .doc(FirebaseAuth.instance.currentUser!.uid);
    reference.snapshots().listen((querySnapshot) {
      if (mounted) {
        setState(() {
          followersData = querySnapshot.get("followers");

          followedData = querySnapshot.get("followed");
        });
      }
    });
  }
}

```

figure 9. 28

The state class `_WhoReadingState` extends the `State` class and is associated with the `WhoReading` widget. The state class contains a `TextEditingController` named `comment`, which is used to manage the text entered in a text field. It also includes a `_formKey`, which is a `GlobalKey<FormState>` used to validate and manage the form state. There are two lists, `followersData` and `followedData`, which are initially empty. In the `initState` method, the overridden method from the `State` class, a `DocumentReference` is created to reference the current user's document in the "users" collection. The `reference.snapshots().listen()` method is called to listen for changes in the referenced document. When a change occurs, the associated callback function is executed. Inside the callback function, the state is updated using `setState()`. The `followersData` and `followedData` lists are assigned the values obtained from the document snapshot using the `querySnapshot.get()` method. The `setState()` method triggers a rebuild of the widget tree to reflect the updated state.



```

Future getBookInfo({required String id}) async {
  final response = await http
    .get(Uri.parse('https://www.googleapis.com/books/v1/volumes/$id'));
  if (response.statusCode == 200) {
    String jsonResponse = response.body;
    Map<String, dynamic> data = json.decode(jsonResponse);

    return data;
  } else {
    print('API isteği başarısız oldu: ${response.statusCode}');
    throw 'hata';
  }
}

```

figure 9. 29

The function `getBookInfo` takes a required parameter `id`, which represents the ID of the book. Inside the function, an HTTP GET request is made to the Google Books API using the provided book ID. The URL is constructed with the format `'https://www.googleapis.com/books/v1/volumes/$id'`. The response from the API is received and checked for a status code of 200, indicating a successful request. If the response status code is 200, the response body is extracted as a string. The string response is decoded using `json.decode` to convert it into a `Map<String, dynamic>`. The decoded data is returned from the function. If the response status code is not 200, an error message is printed, and an exception is thrown.

## 10. Test

```
void main() {  
  testWidgets('Counter increments smoke test', (WidgetTester tester) async {  
    // Build our app and trigger a frame.  
    await tester.pumpWidget(const MyApp());  
  
    // Verify that our counter starts at 0.  
    expect(find.text('0'), findsOneWidget);  
    expect(find.text('1'), findsNothing);  
  
    // Tap the '+' icon and trigger a frame.  
    await tester.tap(find.byIcon(Icons.add));  
    await tester.pump();  
  
    // Verify that our counter has incremented.  
    expect(find.text('0'), findsNothing);  
    expect(find.text('1'), findsOneWidget);  
  });  
}
```

*figure 10.1*

It tests the behavior of a counter widget in a Flutter app. The `testWidgets` function is used to define a test case. It takes a description of the test and a callback function as arguments. The callback function is executed when the test is run. Inside the callback function, the first step is



## 11. Conclusion

The results show that Readify social platform offers a unique solution for book enthusiasts looking to make reading a more social and interactive experience. With features such as genre discovery, comments and suggestions from other readers, and the ability to create and share book history, Readify makes it easy for users to make informed decisions about what to read next and connect with like-minded individuals. Additionally, the platform has the ability to show who has read the book in real time, allows users to find others with similar reading tastes and communicate with them more easily. Overall, Readify is a valuable tool for anyone looking to enhance their reading experience and expand their social circle. Readify is built with Flutter technology, and Firebase - Firestore technology is used for the database.

On the other hand, Readify poses various risks related to the operation application and the technical structure of the application. These risks include the potential for users to use the application for purposes other than discovering new books and making connections with other readers for example; the full names and e-mail addresses of the users registered in the system are visible in their profiles by other users. In addition, the user can receive messages from all other users in the system, users cannot block any user. Also, user can make comments on the books other than the purpose of the application. As well as potential limitations and drawbacks of the technologies used to develop the app, such as Flutter and Firebase. While doing the project, we extend the knowledge we learned during the software engineering program.

## References

Web Site: “What does social platform mean”, as appears on

[www.techopedia.com/definition/23759/social-platform](http://www.techopedia.com/definition/23759/social-platform), accessed on November 16th, 2022. [1]

Web Site: <https://cevap-bul.com/proje-yonetimi-kapsam-nedir/>, accessed on November 16th, 2022.

[2]

Web Site: “Twitter”, as appears on [en.wikipedia.org/wiki/Twitter](https://en.wikipedia.org/wiki/Twitter), accessed on November 22nd, 2022. [3]

Web Site: “Facebook”, as appears on [en.wikipedia.org/wiki/Facebook](https://en.wikipedia.org/wiki/Facebook), accessed on November 22nd, 2022. [4]

Web Site: “Goodreads”, as appears on [en.wikipedia.org/wiki/Goodreads](https://en.wikipedia.org/wiki/Goodreads), accessed on November 22nd, 2022. [5]

Web Site: “LibraryThing”, as appears on [en.wikipedia.org/wiki/LibraryThing](https://en.wikipedia.org/wiki/LibraryThing), accessed on November 22nd, 2022. [6]

Web Site: “Litsy”, as appears on [en.wikipedia.org/wiki/Litsy](https://en.wikipedia.org/wiki/Litsy), accessed on November 22nd, 2022. [7]

Web Site: “React Native, Ionic Ve Flutter Karşılaştırması”, as appears

[webmobilyazilim.com/mobil-uygulama-haber/flutter-react-native-ionic-mobil-uygulama](http://webmobilyazilim.com/mobil-uygulama-haber/flutter-react-native-ionic-mobil-uygulama), accessed on November 22nd, 2022. [8]

Web Site: “Flutter kullanmanın dezavantajları:”, as appears [www.webtekno.com/flutter-nedir-nasil-kullanilir-h115673.html](http://www.webtekno.com/flutter-nedir-nasil-kullanilir-h115673.html), accessed on November 23rd, 2022. [9]

Web Site: “Firebase nedir?”, as appears [kenanatmaca.com/firebase-nedir/](http://kenanatmaca.com/firebase-nedir/), accessed on November 22nd, 2022. [10]

Web Site: “Bir veritabanı için Firebase’in avantajları ve dezavantajları nelerdir?”, as appears [www.web-development-kb-eu.site/tr/database-recommendation/bir-veritabani-icin-firebase-avantajlari-ve-dezavantajlari-nelerdir/1958292777/amp/](http://www.web-development-kb-eu.site/tr/database-recommendation/bir-veritabani-icin-firebase-avantajlari-ve-dezavantajlari-nelerdir/1958292777/amp/), accessed on November 23rd, 2022. [11]

Web Site: “Dart altyapısı”, “Flutter Kullanmanın Avantajları Nelerdir?” as appears

[bluemarkacademy.com/flutter-nedir-ne-icin-kullanilir-ve-avantajlari-nelerdir](http://bluemarkacademy.com/flutter-nedir-ne-icin-kullanilir-ve-avantajlari-nelerdir), accessed on November 22nd, 2022. [12]

Web Site: “Firebase’in avantajları nelerdir?”, as appears [www.milliyet.com.tr/egitim/firebase-nedir-nasil-kullanilir-firebase-avantajlari-nelerdir-6465331](http://www.milliyet.com.tr/egitim/firebase-nedir-nasil-kullanilir-firebase-avantajlari-nelerdir-6465331), accessed on November 22nd, 2022. [13]