



POLITECNICO
MILANO 1863

Book Finder

Software Design Document

Angel Chelaru

Contents

1	Introduction	3
1.1	Idea	3
1.2	General Purpose	3
1.3	Goals	3
1.4	Platform	4
1.5	Language and Framework	4
1.6	Scope	4
2	Overall Description	5
2.1	Product Description	5
2.2	Application Features	5
2.2.1	Core Features	5
2.2.2	Marginal features	6
2.3	Assumptions	6
3	Data Design	7
3.1	NoSQL Database	7
3.2	Firestore Cloud	7
3.3	Structure	8
4	Architectural Design	10
4.1	System Architecture	10
4.1.1	Frontend Development	10
4.1.2	Middleware Development	10
4.1.3	Backend Development	10
4.2	Package Organization	11
4.2.1	Main Folder	11
4.2.2	Application Organization	12
4.2.3	Scenes	13
4.2.4	Components	13
4.2.5	Actions	14
4.2.6	Utils	14
4.2.7	Styles	15
4.2.8	Types	15
4.2.9	Assets	16
4.2.10	Tests	16
4.3	UML Diagram	17
4.4	API Description	17
4.5	Security	19

5	User Interfaces	20
5.1	Splash screen	20
5.2	Login Screen	20
5.3	SignUp Screen	21
5.4	Homepage Screen	22
5.5	AddBook Screen	23
5.6	Search Screen	24
5.7	Results Screen	24
5.8	Messages Screen	25
5.9	Chat Screen	26
5.10	WishList Screen	26
5.11	Add to WishList Screen	27
5.12	Push Notification	28
5.13	Settings Screen	29
6	External Services and Libraries	30
6.1	Firebase	30
6.1.1	Firebase Authentication	30
6.1.2	Cloud Firestore	30
6.1.3	Functions	30
6.2	Google Book API	31
6.3	Other libraries and frameworks	31
7	Test cases	32
7.1	JEST	32
7.2	Manual tests	32
8	Cost Estimation	37
8.1	Time	37
8.2	Price	38

1 Introduction

1.1 Idea

The idea of this project was originated by noticing how old university books were sold by college students among each other.

Nowadays, the main procedure used by the people to sell their schoolbooks was through the use of Facebook groups of which most students are members. I, in the first place, have made several posts to sell some of my old books on multiple groups of this kind.

Nonetheless, not all the times I have managed to sell them. I have noticed that, given the Facebook policy to show most relevant posts first, if nobody immediately needed the book I was planning on selling, my advert would get lost among newer posts. This was decreasing my chances of selling the book. In fact, even if somebody was looking for the book I was still selling, they were not able to find it in the group.

From this, I decided to create a mobile application to facilitate this exchange of schoolbooks.

1.2 General Purpose

BookFinder is a mobile application for Android devices that allows its users to be put in contact to exchange books. To be able to access it, the user needs to be registered and to have stated where their university is located and what faculty they are attending, in order to make it easier for the algorithms to show books relevant to them.

1.3 Goals

The goals of this mobile applications are:

1. To create a way of posting the books you want to sell.
2. To put in contact with each other the seller and the buyer.
3. To be able to create a wish list of books you need and receive a notification as soon as somebody posts that book.

1.4 Platform

The application was implemented for Android only. The main reasons were for this choice were:

- I do not own a computer that runs macOS.
- I do not own an iPhone to test it on an actual device.
- It is illegal to develop the application on a device not made by Apple if you are implementing an iOS App.

1.5 Language and Framework

I decided to implement BookFinder using JavaScript and React Native as its framework. The nice thing about React Native is that it allows you to write applications for Android and iOS devices by using the same coding language.

Furthermore, I decided not to use plain JavaScript, but to use TypeScript, which is an open-source programming language developed and maintained by Microsoft, which brings types in the JavaScript world. The main reasons for this choice were that TypeScript improves the readability of the code and it is a best-practise to use it in the software industry.

1.6 Scope

BookFinder is an application meant to be used by all university students throughout Italy. It may also be used by ex-students who are looking to sell their old books, or any other person who wishes to sell or buy a schoolbook.

2 Overall Description

2.1 Product Description

BookFinder is a mobile application written in JavaScript with the use of React Native as its framework. It provides the users the possibility to sell their old books, or to find used books that they might need.

The application allows only registered users to log-in and be able to take advantage of its functionalities. Since it is thought for only students selling school books, at the moment of registration the user is asked what university they belong to and what is their faculty.

As soon as the user is logged in, they can see all the books that are being sold from students of their university.

Users now can either add a new book that they want to sell, or search for a book they are in need for. Whenever they find it, they can contact the seller and arrange where to meet to do the actual deal.

2.2 Application Features

The features of this application have been divided in two categories: core features and marginal features.

2.2.1 Core Features

The core features are:

- Register a new user with all their information.
- Log-in a user only with their correct credentials.
- Add a book you want to sell in the database, so it can be seen by the other users.
- Search for a book you are interested in by inputting its information (title, author, isbn, etc..). Seeing all the results of the people selling that specific book.
- Get in contact with the person who is selling a book that the user is searching for, so that they can arrange where to meet and exchange the item.

2.2.2 Marginal features

The marginal features are those features that are not essential for the correct use of the application, but they are add-on features that simplify the use of it.

- Add a new book through the use of the camera. Scan the ISBN.
- Insert a book in the user's WishList
- Get a notification whenever somebody inserts a book with the characteristics of one of the books in their WishList.

2.3 Assumptions

When implementing this mobile application, I have made the following assumptions.

- Users do not lie about their university.
- Users do not post books that they do not have.
- Users will not use inappropriate or harmful language so that another user needs to block them.
- Users will only use this app for its original purpose.

3 Data Design

In this section it will be explained how the what kind of database was chosen, how it was structured, and the reason behind those choices.

3.1 NoSQL Database

The type of database chosen for the making of BookFinder has been a NoSQL database. A NoSQL database is a nonrelational database that is flexible with data models and schemas. Its key points are its functionality and its high scalability.

Among the several reasons for this choice, the main one was that Firebase was offering only a NoSQL database for free. Furthermore, I had never had the chance to work with a NoSQL database, and I thought this was a great opportunity to learn. Furthermore, no complicated query need to be performed so a relational database was not a must.

3.2 Firestore Cloud

The platform chosen to host our database was Cloud Firestore. It is a cloud-hosted, NoSQL database that can be accessed through native SDKs. The data is stored in documents that contain fields mapping to values. These documents are stored in collections, which are the containers of the documents, and they can be used to organize the data and build queries.

The key capabilities are:

Flexibility: data model supports flexible, hierarchical data structures.

Scalability: automatic multi-region data replication, strong consistency guarantees, atomic batch operations, and real transaction support.

Realtime updates: it uses data synchronization to update data on any connected device.

3.3 Structure

The database is made of five collections. Each collection either contains documents or it contains subcollections.

- Books: this collection contains a document for each book inserted by the users. The structure of the documents is the following.

```
{
  title: string,
  author: string,
  editor: string
  isbn: string,
  price: string
  inputTime: timestamp,
  sellerEmail: string,
  sellerUniversity: string,
  sellerFaculty: string,
  sellerName: string,
  sold: boolean
}
```

- Users: this collection contains a document for each user. The structure of the documents is the following.

```
{
  firstname: string,
  lastname: string,
  email: string,
  birthdate: date,
  university: string,
  faculty: string
}
```

- Wishlist: this collection contains a document for each book of a user wishlist. The structure of the documents is the following.

```
{
  title: string,
  author: string,
  editor: string,
```

```

    isbn: string,
    email: string
  }

```

- Messages: this collection contains a sub-collection for each conversation between two users. The id of the sub-collection is the merge of the emails of the two users *the emails are merged in alphabetical order*. The structure of the documents is the following.

```

{
  _id: string,
  createdAt: timestamp,
  text: string
  user: {
    _id: string,
    name: string
  }
}

```

- Last messages: this collection contains a sub-collection for each user. Under that sub-collection there is a document for each last conversation they have had. The structure of the documents is the following.

```

{
  conversationWith: string,
  lastSender: string,
  name: string,
  message: string,
  read: boolean,
  timestamp: timestamp
}

```

4 Architectural Design

4.1 System Architecture

The application has been divided in three parts: screens, components, and the middleware. The back-end has been implemented in the cloud thanks to Firebase Functions and Firestore Cloud.

4.1.1 Frontend Development

The frontend development has been done dividing the application in two parts: the screens and the components.

The screens are the part related to the various scenes we see in the application. We have the Login screen, the Homepage, the Chat, and so on. Each has its own file in the scenes folder.

The components are objects used by the screens. They represent parts of a screen. The most important example is the BarcodeScanner component, which is used in the AddBook screen, and it is used to obtain the information about a book just by scanning the barcode of the physical book with the camera of the phone.

All the elements in the frontend use the middleware to interact with the database or to make HTTP requests in order to fetch data from Google Books API.

4.1.2 Middleware Development

The middleware implements the connection between the Firebase platform and the application. It handles the authentication of the users (login, registration, forgotten password email submission) and the data fetching from the database.

It is implemented in the file inside the Actions folder. The API can be checked in the API Description chapter.

4.1.3 Backend Development

I decided to use Firebase, a Backend-as-a-Service product, so that I did not have to worry about where to host my database and about scalability. In fact, I used the Firestore Cloud, the NoSQL database offered by Firebase.

Furthermore, I used the Cloud Functions in order to trigger the push notifications on the users' application whenever someone adds a book with the same information as a book they have in their WishList.

4.2 Package Organization

4.2.1 Main Folder

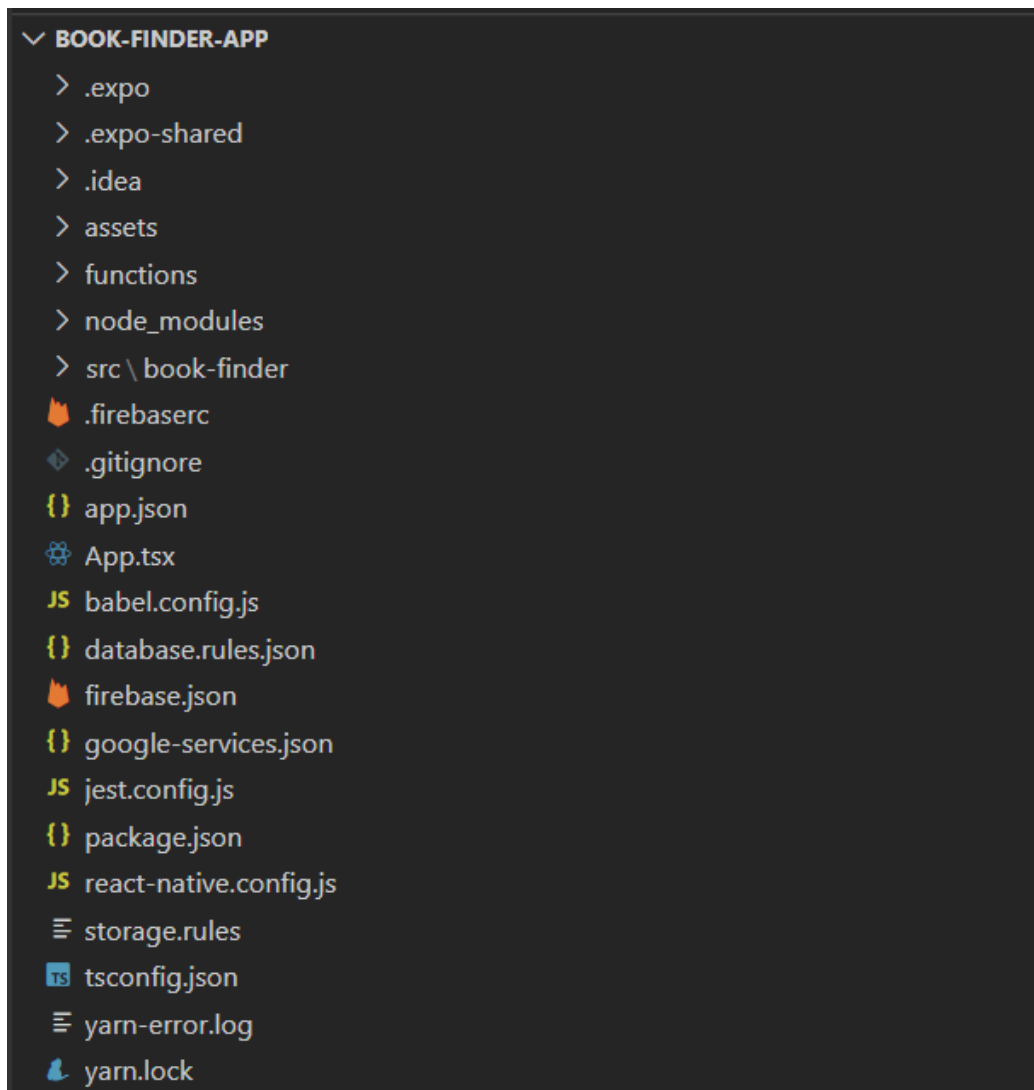


Figure 1: Main folder

In the picture above it is possible to see the classical organization of a React Native application.

The functions folder it is used for the cloud functions implementation.
The src\book-finder folder contains the various files used for the actual implementation of the application.
The App.tsx file is the file that will be first used when we run our application.
In it there is the implementation of the initialisation of the app.
The rest of the files are needed to configure the various frameworks used.

4.2.2 Application Organization

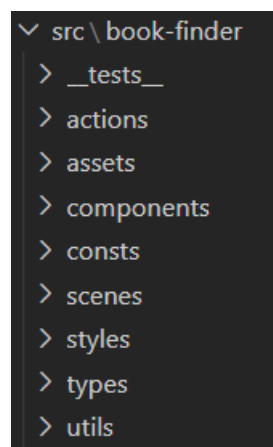


Figure 2: Application folder

In the picture above we can see the structure of the folders for the various files of the implementation of the application. Furthermore, there is also a folder for the tests made using Jest.

4.2.3 Scenes

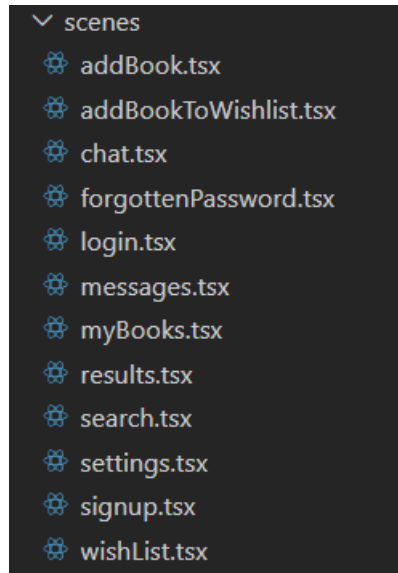


Figure 3: Scenes folder

In the picture above we can see the various screens of the application.

4.2.4 Components

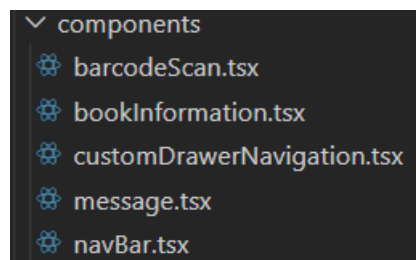


Figure 4: Components folder

In the picture above we can see the various components used in the screens of the application.

4.2.5 Actions

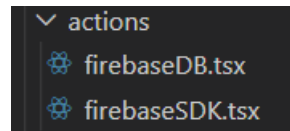


Figure 5: Actions folder

In the picture above we can see the files that are used to interact with the Firebase. The first file manages the database queries, whereas the second file manages the user authentication.

4.2.6 Utils

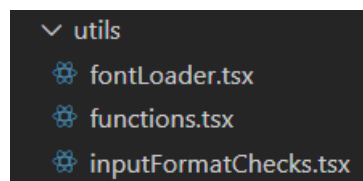


Figure 6: Util functions folder

In the picture above we can see the files that will contain all the utility functions used in the screens and components of the application. All the functions in these files are tested with Jest in the tests folder.

4.2.7 Styles

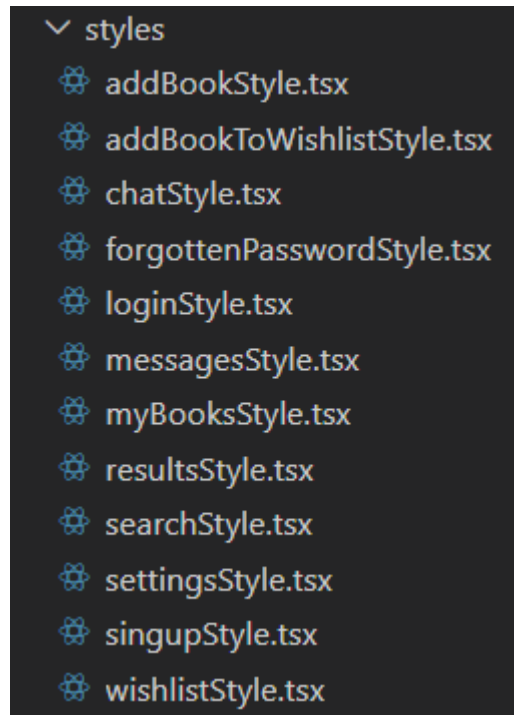


Figure 7: Styles folder

In the picture above we can see the structure of the files in the styles folder. There is one for every screen of the application.

4.2.8 Types

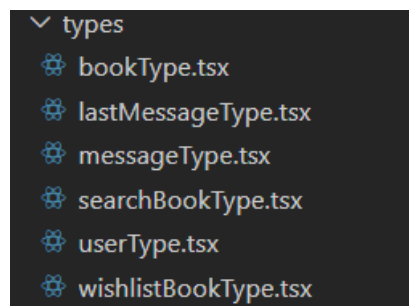


Figure 8: Types folder

In the picture above we can see the files for all the types that I had to implement since I am using TypeScript.

4.2.9 Assets

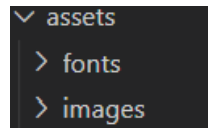


Figure 9: Assets folder

In this folder there are the fonts and images needed in the application.

4.2.10 Tests

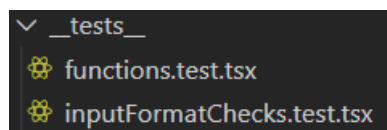
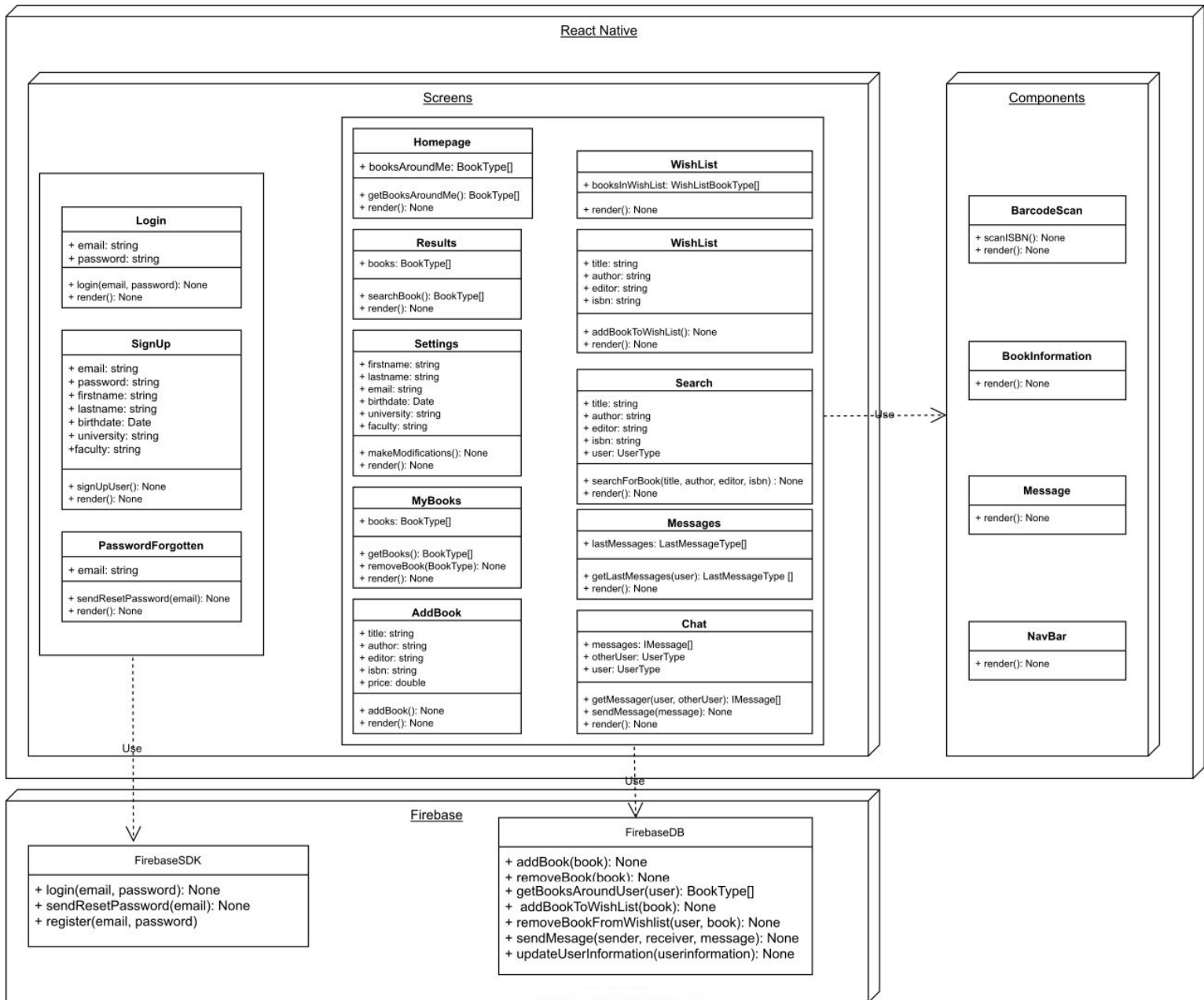


Figure 10: Tests folder

In the folder there are the tests written using Jest for the utility functions that are in the utils folder.

4.3 UML Diagram



4.4 API Description

The following functions are the functions available for the front-end part of the project to interact with the database. The implementations can be found

in the "FirebaseDB" file.

- *getBooksAroundUser()*: this function takes as an input the user information and a callback function to set the books that are being sold by student of his university and his faculty.
- *getSellingBook()*: this function takes as input the email of the user and a callback function that updates the application with the books currently being on sale by the user.
- *addBook()*: this function takes as input the user, the title, author, editor and isbn of the book the user wants to sell, and the price he wants the book to be sold at, and it adds it in the database.
- *getWishList()*: this function takes as input the user's email and a callback function to update the value in the screen with all the user's books in his WishList.
- *addBookToWishList()*: this function takes as input the user's email, the title, author, editor and isbn of the book we want to add in the user's WishList, and it adds it in the database.
- *removeBook()*: this function takes as in input the email of the user, the title, author, editor and isbn of the book we want to delete, and it removes it from the database.
- *removeBookFromWishList()*: this function takes as in input the email of the user, the title, author, editor and isbn of the book we want to delete from the user's WishList, and it removes it from the database.
- *searchBook()*: this function takes as input the user information, the title, the author, the editor and the isbn of the book the user is looking for, it searches it in the database, and through a callback function it returns it.
- *getUser()*: this function takes as input the user's email and a callback function that will be used to load the user's information in the application.
- *addUser()*: this function takes as input the new user's email, his first-name, his lastname, his birthdate, his university and his faculty, and it creates a new document in the database with these values.

- *updateUser()*: this function takes as input the user's email, his first-name, his lastname, his birthdate, his university and his faculty, and it updates the document in the database with the new values.
- *getMessages()*: this function takes as input the sender email, the receiver email and a callback functions to update the application with the messages exchanged between the two users.
- *addMessage()*: this function takes as input the sender's email and receiver's email, and the message, and it adds it in the database.
- *getLastMessages()*: this function takes as input the user's email and a callback function to set the last messages in the application.
- *setLastMessageRead()*: this function takes as input the email of the current user and the user he has open the chat with, and it sets to true that the last message has been seen.
- *updateLastMessage()*: this function takes as input the sender email, the receiver email and the message, and it update the last message sent between the two users.

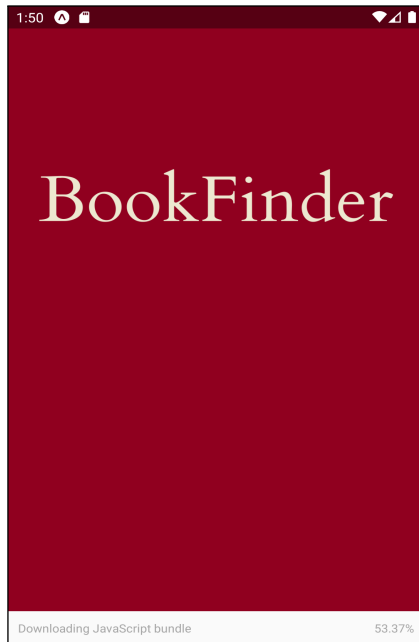
4.5 Security

This application interacts with a NoSQL database. Therefore, SQL injections are not a threat. Nevertheless, there exist NoSQL injection attacks.

For this reason, all the data has been sanitized before being sent to the database. I was careful to escape even nested elements in an object and rejected all the data that contained the characters "--" used for comments.

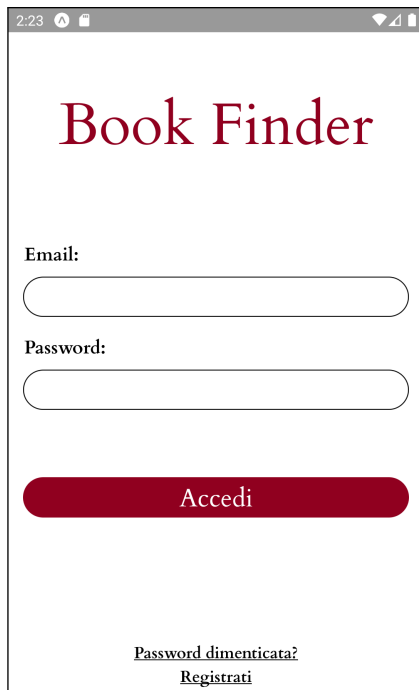
5 User Interfaces

5.1 Splash screen



This is splash screen shown when the app is loading. The screenshot was taken in development mode. When the app will be released, the white banner at the bottom will not be shown.

5.2 Login Screen



This is the screen where the user can login by inserting its credentials

5.3 SignUp Screen



A mobile app interface for 'Book Finder' showing a sign-up form. The form includes fields for Name (Marco), Surname (Rossi), Birthdate (01-01-1995), Email (test@gmail.com), University (dropdown menu), Faculty (dropdown menu), and Password. The title 'Book Finder' is at the top in a large, red, serif font.

This is the signup screen.

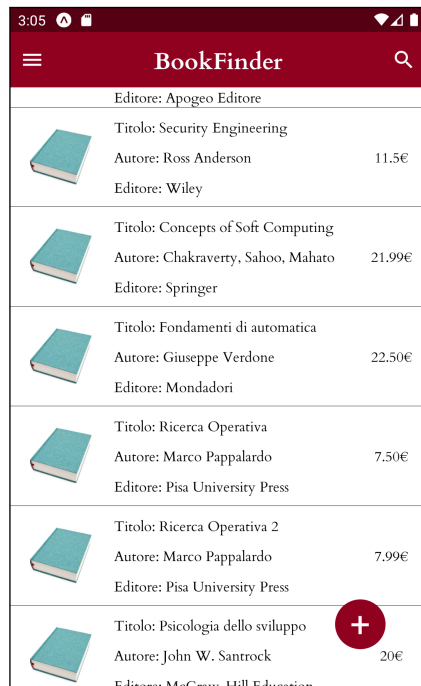
In order to sign up, the user needs to enter their name, their lastname their birthdate, their email...



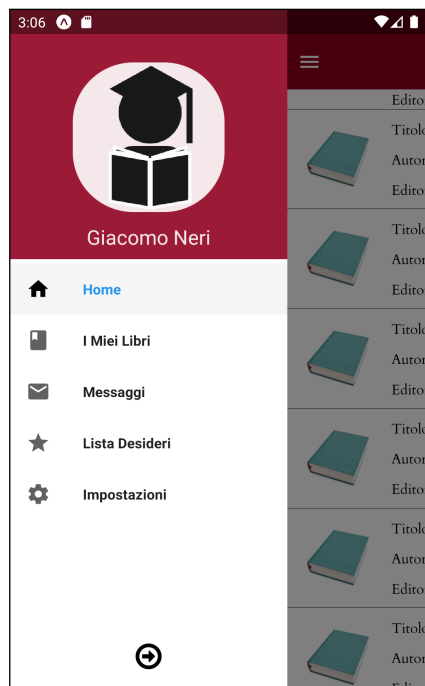
A mobile app interface for 'Book Finder' showing the continuation of the sign-up form. It includes fields for University (dropdown menu), Faculty (dropdown menu), Password, and Verifica Password. At the bottom, there is a red button labeled 'Registrati' and a link 'Accedi con le tue credenziali'.

...the university they are attending, their faculty, and a password twice.

5.4 Homepage Screen



In the homepage of the mobile application we can see the latest books added by students of our university.



When we press the hamburger button or swipe from left to right, the side menu opens. In the side menu, we can navigate in the various screens of the application.

5.5 AddBook Screen



3:06

Usa il codice a barre per ottenere le info

Titolo

Autore

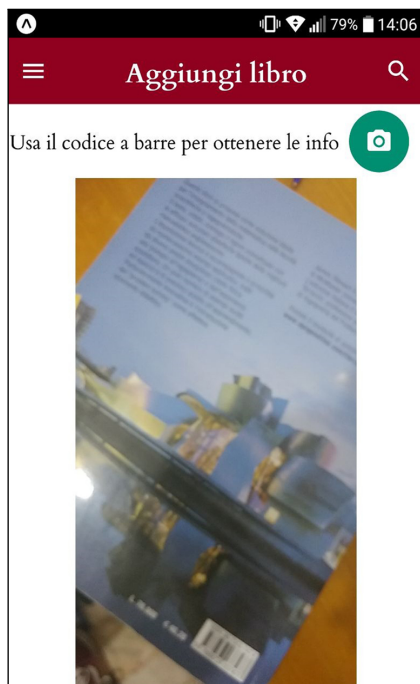
Editore

ISBN

Prezzo

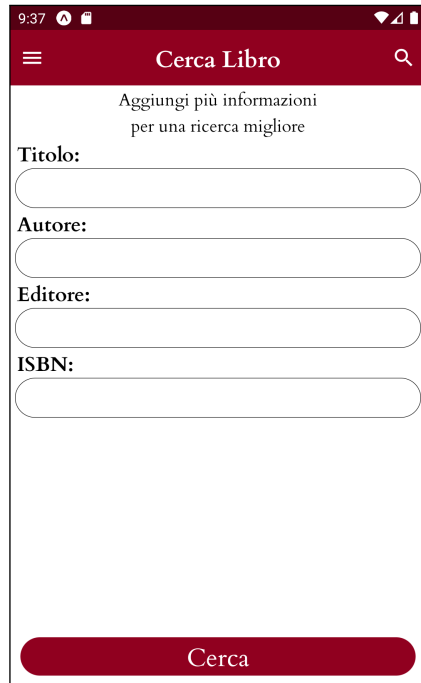
Aggiungi

This is the screen in which we input all the information to add a new book so that people can get in touch to buy it.



By clicking on the camera icon, we reach this screen. By focusing on the bar code of a book we want to add, we can fill the information needed automatically.

5.6 Search Screen



9:37

Aggiungi più informazioni
per una ricerca migliore

Titolo:

Autore:

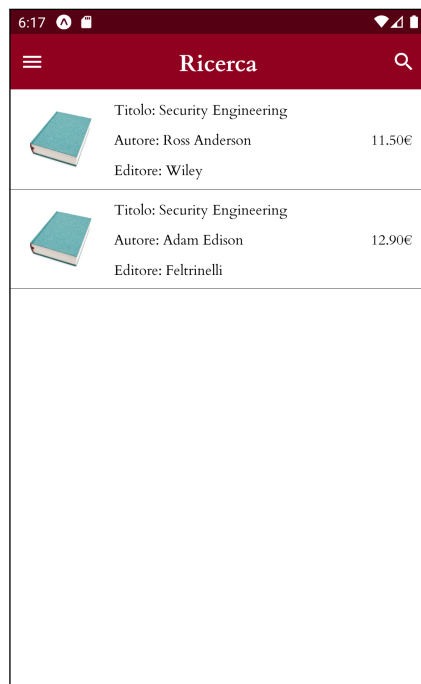
Editore:

ISBN:

Cerca



By clicking on the camera icon, we reach this screen. By focusing on the bar code of a book we want to add, we can fill the information needed automatically.

5.7 Results Screen



6:17

Ricerca

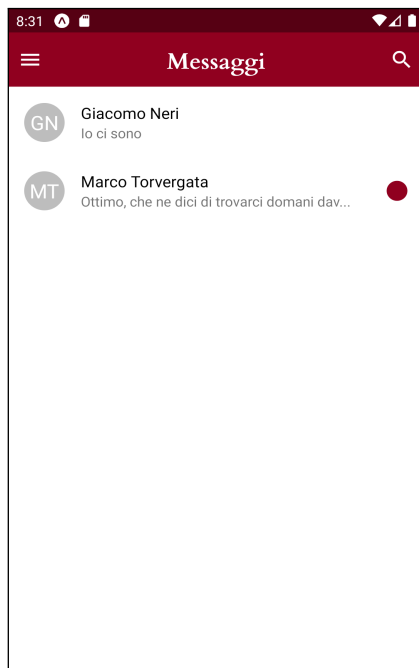
	Titolo: Security Engineering Autore: Ross Anderson Editore: Wiley	11.50€
	Titolo: Security Engineering Autore: Adam Edison Editore: Feltrinelli	12.90€

This is the screen where the results of our research will appear



This test the screen that will appear if there are not matches of the searched book in our database.

5.8 Messages Screen



In this screen we can see all our old conversations. If we have an unread message, then a red dot will appear at the right of the message preview.

5.9 Chat Screen



In this screen the users can communicate and agree on where to meet to sell the book.

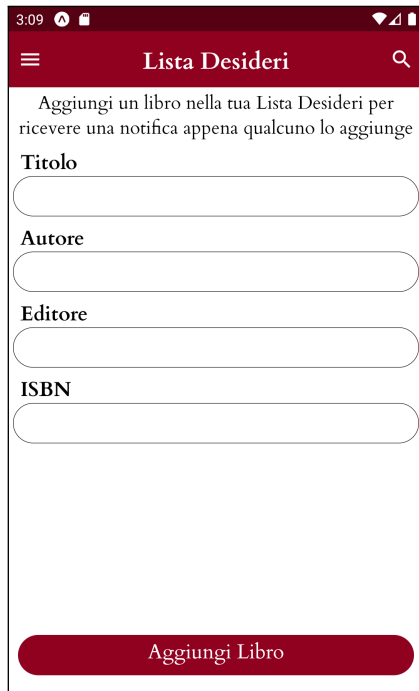
Figure 11: Chat screen

5.10 WishList Screen



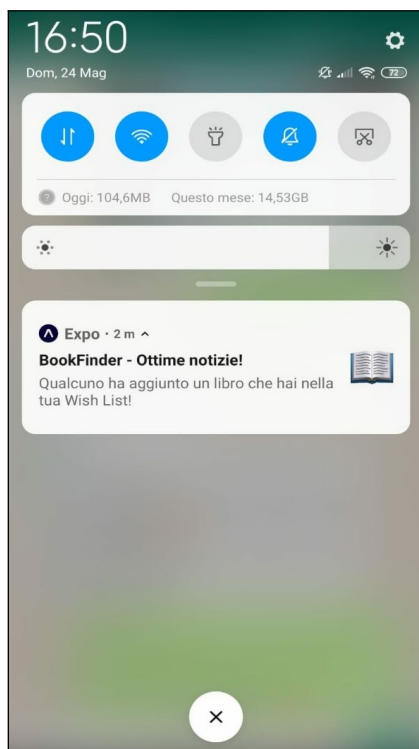
This is the screen we all our WishList books are shown.

5.11 Add to WishList Screen



This is the screen in which we need to put all the book information so that we can get a notification as soon as somebody starts selling this book.

5.12 Push Notification



This is the screen of a push notification to alert the user that someone has input a book that they have in their Wish List.

5.13 Settings Screen

Email	angel.chelaru@hotmail.it
Nome	Angel
Cognome	Chelaru
Data di nascita	19-02-1996
Università	Politecnico di Milano
Facoltà	Ingegneria
Modifica Informazioni	

This is our settings screen. Here, we can change our basic information, with the exception of our email, which cannot change.

6 External Services and Libraries

6.1 Firebase

Firebase is a Backend-as-a-Service (BaaS) platform that gives several tools to develop the backend of a mobile application in the cloud.

There are three reasons for the choice of using this service.

First of all, it is free up to a certain level of usage. This has given us the chance of implementing and testing the app without having to pay anything. Second, it has a very intuitive interface with an online console that allows us to check the changes in the database, the new registrations of the users, and many other analytics.

Third, the documentation is well kept and easy to understand.

During the research of the best BaaS to use, Amazon Web Services was also taken into consideration. Nevertheless, there is bug that has not been solved yet with Amplify for Windows.

6.1.1 Firebase Authentication

Firebase Authentication provides backend services to authenticate users to this app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more. Nevertheless, I decided to only use the classic email-password authentication. This service is provided through some SDKs, and it handles the registration of a new user, its signing-in and the request to change one's password in case the user has forgotten it.

6.1.2 Cloud Firestore

Cloud Firestore is a flexible, scalable database for mobile development (and not only). It keeps the data in sync across client apps through real-time listeners. It uses a flexible, scalable NoSQL cloud database to store and sync data.

6.1.3 Functions

Cloud Functions for Firebase is a serverless framework that lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests. The TypeScript code is stored in Google's cloud and runs in a managed environment. There's no need to manage and scale your

own servers.

They are used to trigger the push notifications on the users' device whenever a book the have in the WishList is triggered.

6.2 Google Book API

For the feature that retrieves the book information by scanning the bar code with the phone camera, Google Book API was being used. In fact, after having scanned the ISBN number through the bar code, a request is sent to retrieve the eventual data from its database. In case it is found, the various field get filled automatically.

6.3 Other libraries and frameworks

During the implementation of the application there have been used several libraries and frameworks to avoid reimplementing features that have been already implemented and tested. Here it is a list of them:

- Expo: it is used handle the push notifications;
- React Navigation Hooks: it is used to implement the hooks, like the Context, the state management;
- GiftedChat: it is used to implement the chat;
- Formik: it is used to make the implementation of forms easier;
- Awesome Alerts: it is used to make nice pop-ups;
- React Keyboard Aware Scroll View: it is used to avoid the keyboard covering the input fields when typing.

7 Test cases

7.1 JEST

JEST is the most used JavaScript Testing Framework. It is open source and maintained by Facebook. It was used to test the utility functions implemented for the application to check all the edge cases.

The tests can be found in the tests folder, and they are related to the utility functions used throughout the application.

To run all the tests, it is only necessary to run the following command in the shell from the base directory:

```
>> npm test
```

7.2 Manual tests

Since there does not exist a consolidated framework used to check the interface, several tests were done manually on the final product. The main ones were:

Test Case	Register new user
Goal	Sign-in a new user
Input	Click on "Registrati", fill up all the information with valid data and press "Registrati"
Expected outcome	The new user is registered.
Actual outcome	CORRECT: if the user enters an email that has not been registered yet, he successfully gets signed-in and he is redirected to the login page.

Test Case	Reset password
Goal	Reset the password
Input	Click on "Password dimenticata?" in the login page and insert your email. Then click on "Invia Email".
Expected outcome	The user receives an email with a link to change the password.
Actual outcome	CORRECT: The email is received at the inserted email address, and by filling the change password form, the password is actually changed.

Test Case	Login
Goal	Log a user in.
Input	In the Login page, insert the email and the right password of a user, and then press "Accedi".
Expected outcome	Get redirected to the user homepage.
Actual outcome	CORRECT: We get redirected to the user homepage.

Test Case	Login with a wrong password
Goal	Try to login with a wrong password and get notified
Input	In the login page insert the email of a user and a wrong password. Then press "Accedi".
Expected outcome	Not get directed to the homepage, but receive a pop-up saying that the password is not correct.
Actual outcome	CORRECT: We are not redirected to the homepage, but we get notified that the password inserted is not correct.

Test Case	Search for a book
Goal	Look up a book we know is in the database.
Input	From the homepage click on the magnifying glass icon on the top right side of the screen. Insert the data of a book and press "Cerca".
Expected outcome	Get to the result page and see the book you are looking for.
Actual outcome	CORRECT: We got to the Result page and the only book showing was the one we were looking for.

Test Case	Search for a book (2)
Goal	Look up a book we know is not in the database.
Input	From the homepage click on the magnifying glass icon on the top right side of the screen. Insert the data of a book we know is not in the database and press "Cerca".
Expected outcome	Get to the result page and get the screen saying that no book has been found.
Actual outcome	CORRECT: We got to the Result page and the screen had a message saying that the book is not there.

Test Case	Add a new book
Goal	Add a new book to be sold
Input	From the homepage click on the "+" button at the bottom right part of the screen. Insert the data of the new book to be added and click "Aggiungi".
Expected outcome	A new book is added in "I miei libri"
Actual outcome	CORRECT: The book that was inserted is now shown in the "I miei libri" screen.

Test Case	Add a new book using the camera
Goal	Add a new book to be sold by inserting the data through its ISBN number.
Input	From the homepage click on the "+" button at the bottom right part of the screen. Click on the camera button and scan the barcode of the physical book. Then input a price and press "Aggiungi".
Expected outcome	The information about the book should be inserted automatically and then the book should be added in "I miei libri".
Actual outcome	CORRECT: The information about the book is correctly added automatically. The book is then shown in the "I miei libri" screen.

Test Case	Delete a book we are selling
Goal	Remove a book from "I miei libri".
Input	Go to the screen "I miei libri" from the side menu. Swipe to the left the book we want to remove and then press the red trash button.
Expected outcome	The selected book should be removed from the list.
Actual outcome	CORRECT: The selected book is correctly removed from the list.

Test Case	Add a book to the Wishlist
Goal	Add a book in the Wishlist
Input	Go to the Wishlist screen. Press "Aggiungi un libro", and then insert the data about a book. Finally, press "Aggiungi".
Expected outcome	A new book should be added in the Wishlist.
Actual outcome	CORRECT: A pop-up appears saying that the book is added. As soon as we click outside of the pop-up we are redirected to the Wishlist screen, and we see the new added book.

Test Case	Remove a book from Wishlist
Goal	Remove a book from the Wishlist
Input	Go to the Wishlist screen. Swipe left on the book we want to remove. Click on the red trash bin icon.
Expected outcome	The book should disappear from the Wishlist.
Actual outcome	CORRECT: The book disappears from the Wishlist.

Test Case	Check Push Notifications
Goal	Check the correct receiving of push notifications.
Input	After having input a book in your wish list, use another account to check if you are receiving the push notifications. Note: you need to use a physical device in order to receive push notifications.
Expected outcome	Receive a push notification on the phone.
Actual outcome	CORRECT: Correctly received the push notification on the phone.

Test Case	Contact a seller from the homepage
Goal	Contact a seller through the book information we see in our homepage.
Input	Click on one book shown in the homepage. The book information should expand. Then click on the "Contatta" button.
Expected outcome	Get to the chat of the person selling the book.
Actual outcome	CORRECT: We get redirected to the chat with the person selling the selected book.

Test Case	Send a message
Goal	Send a message to another user.
Input	When in the chat with another user, type a message and press "Send".
Expected outcome	The message should show up in the chat and the other user should receive it.
Actual outcome	CORRECT: The message does show up. The other user receives the message with a delay of a few seconds.

Test Case	Change university and faculty from Settings
Goal	Change the user university and faculty.
Input	Go to the "Impostazioni" screen. Click on "Modifica informazioni". Change the university and the faculty. Click "Salva modifiche".
Expected outcome	The new information should be displayed.
Actual outcome	CORRECT: The new information is displayed correctly.

Test Case	Change birthdate from Settings
Goal	Change the birthdate of the user.
Input	Go to the "Impostazioni" screen. Click on "Modifica informazioni". Change the birthday. Click "Salva modifiche".
Expected outcome	The new information should be displayed.
Actual outcome	CORRECT: The new information is displayed correctly.

Test Case	Logout
Goal	Logout of the user.
Input	Open the side menu by swiping right or by clicking on the hamburger button. Then click on the circled right arrow icon at the bottom of the menu.
Expected outcome	Logout and go to the Login screen.
Actual outcome	CORRECT: Correctly logged out and redirected to the Login screen.

8 Cost Estimation

8.1 Time

To estimate the time needed for this project, I decided to use the CoCoMo (Constructive Cost Model) model, which is a regression model base on the number of lines of code.

The basic equation for the CoCoMo model is:

$$EFFORT = a * (KLOC)^b * EAF$$

where:

- EFFORT: amount of labor that will be required to complete a task. It is measured in person-months units.
- KLOC: the estimated number of thousands of lines of code.
- EAF: the Effort Adjustment Factor is composed of the cost drivers that take into account various other factors such as reliability, experience, capability.
- a & b: two coefficients that depend on the type of project.

In the CoCoMo model, projects are classified in three classes:

1. Organic: A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.
2. Semi-detached : A software project is said to be a Semi-detached type if the vital characteristics such as team-size, experience, knowledge of the various programming environment lie in between that of organic and Embedded. The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience and better guidance and creativity.
3. Embedded: A software project with requiring the highest level of complexity, creativity, and experience requirement fall under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

Since BookFinder is a small project, and I was familiar with React, I decided to consider it a organic project. Therefore, the coefficients that are used in our formula are $a = 3.2$ and $b = 1.05$.

In oder to calculate our EAF value, I have taken into consideration the standard cost drivers that were relevant to this project. Below we can find the table with the paramenters and their related rating and cost driver value.

Attributes	Rating	Cost Driver Value
Required Software Reliability	Nominal	1.00
Size of Application Database	Low	0.84
Complexity of The Product	Low	0.85
Applications experience	High	0.95
Software engineer capability	Nominal	1.00
Programming language experience	High	0.95
Application of software engineering methods	Nominal	1.00
Use of software tools	High	0.91
All others	Nominal	1.00

We can now estimate the effort that it will be needed to implement this project:

$$EFFORT = 3.2 * (3.5)^{1.05} * (1.00 * 0.84 * 0.85 * 0.95 * 1.00 * 0.95 * 1.00 * 0.91 * 1.00) \approx 5.95$$

From the effort value we can calculate the estimation of the duration of the project by using the formula:

$$DURATION = \frac{EFFORT}{PEOPLE}$$

Since I will implement the project by myself, the duration value will match the effort value. Nevertheless, the estimation is:

$$DURATION = \frac{5.95 \text{ person-months}}{1 \text{ person}} = 5.95 \text{ months}$$

The time estimated for the development of BookFinder is around 6 months.

8.2 Price

When I chose what services to use for the implementation of this project, I have searched for only free services.

Nevertheless, Firebase is free only up to a certain point. In fact, the plan chosen is the Spark Plan. It has the following restrictions:

- Authentication: free up to 10k/month
- Cloud Firestore:
 - Stored data: 1GiB total
 - Document writes: 20K/day
 - Document reads: 50K/day
 - Document deletes: 20K/day
- Cloud Functions: it only allows 125K invocations per month

In case we need more freedom, we could use the Blaze Plan, which has the following prices:

- Authentication: \$0.01/verification
- Cloud Firestore:
 - Stored data: \$0.18/GiB
 - Document writes: \$0.18/100K
 - Document reads: \$0.06/100K
 - Document deletes: \$0.02/100K
- Cloud Functions: \$0.40/million invocations

Given the following assumptions:

- 300 authentications per day
- Each user will either add a new book or add a new book to its wishlist. Furthermore, it will send on average 5 messages. So, there will be 10 write per logged-in user per day.
- Each user will make several writes (new books around the user, searching for a new book, chat messages, etc...), so on average there will be 50 document reads per logged-in user per day.
- Only 5% of the logged-in users will delete a book per day.
- The cloud functions would be invocated each time a new book is added and each time a message is sent. Therefore at least 10000 times per day.

Given these assumptions, the total price per month will still be \$0.00.