University of Piraeus

Department of Digital Systems

Postgraduate Program 'Information Systems & Services'

Πληροφοριακό σύστημα επεξεργασίας δεδομένων τροφίμων από κοινωνικά δίκτυα σε περιβάλλοντα νεφοϋπολογιστικής

Information system processing social media food data on cloud environments

ME1916

Tsatsaronakis Emmanouil

Supervising Professor: Dimosthenis Kiriazis

PIRAEUS 2021

Abstract

The aim of this paper is to present a complete Mobile application using the Dart and Flutter languages for the User Interface. The application uses Firebase for Authentication, Fire Store for data storage, Firebase Storage for storing images and Video as well as Firebase Functions. The application is of the Social Media type, ie users are allowed to upload videos or pictures, create friends and followers who will have access to the user's profile. In addition, it contains recipes that the user will be able to access through search from the application. Finally, the application will be able to recognize the content of a user's food image and suggest corresponding recipes through machine learning.

Table of contents

1. Introduction

Mobile applications are becoming more and more present in our daily lives, allowing users to perform many tasks through the use of mobile devices. As of November 2016, there is more network traffic from mobile devices (48.19%) compared to desktops or laptops (47%). There are different mobile operating systems with different programming languages and tools.

To be distributed to most users, a mobile application must be adapted to two separate platforms, namely Android and IOS. Obviously, the differences between the two are large and often require different skill sets to develop, such as Java / Kotlin for Android only and Object-C / Swift for iOS only. Thus, developers and companies often struggle to cope with the complexity of developing multi-platform applications.

An application is usually developed with a software development kit (SDK) provided by the mobile operating system developer. Applications can also be developed using cross-platform development approaches. This approach allows developers to use a code base that can be run on multiple platforms. Growth costs are often reduced and offer other benefits such as faster growth and release of overhead costs.

Flutter is a new multi-platform technology that promises high performance applications. Provides widgets for Android and iOS thus offering high user satisfaction. This study explores the development of a Flutter application. The programming language behind Flutter is Dart and a combination of these will be used in this work to develop the application. The Dart programming language was developed by Google and can also be used to build server and desktop applications.

The power of social media is growing enormously and developing a social media app is a hot trend these days. The thesis aimed to create a social media application for general purpose goal accomplishment. Its purpose was to explore how to design and create a social media application to improve the users intent to achieve their goals.

1.1 Social media need

Millions of people use social media to connect, meet and share. Online social networks are configuration of relationships to facilitate social interactions which are offered by new web or Mobile applications, ranging communications among people from casual friendships to professional networks found in businesses. By the appearance of the Internet, geographical limits are disappeared by diversified social ties whereas people are enabled to connect to the world through making new connections and using specific services of social media applications. In fact, the new virtual communities response on social media enable people to form globalized relationships [1].

In most of social media platforms, users are easily interested to the open-networking because of their friends and/or their peers' membership. Each individual can lose or refuse relationships/connections when find them no more useful, an action which is less frequent in the real world. Users of social media are able to make countless friendships through joining social groups or adding someone to their friends or connection list. Almost all of social units, professionals, organizations and groups of traditional society have noticed the power of social media and are aware of the values added to their activities by joining this phenomenon.

1.2 Social media importance

Today there exist a lot of social media applications. These usually provide a specific set of services to their users. The most apparent example is Facebook, which is a social network for everybody. Some others are Linked In, Twitter and Instagram. However, do these social networks help the users themselves, or are they just a waste of time? Regardless of the answer, people often have goals they want to achieve. Yet, distraction is everywhere and without a solid habit, it is easy to forget about a personal goal. Ironically distractions quite often come in the form of notifications from social media, or various push notifications of minor importance [2].

The important role of social media for scientific research becomes evident by the sheer number of papers analyzing data from social media platforms [3]. Various review studies and meta-analyses provide an overview of how data extracted from different social media platforms are analyzed, and how social media apps are used in different contexts and environments. For example, one paper reviews literature on the use of social media in academia [4]. It distinguishes between several categories of social media use, including social networking, social data sharing, video, blogging, microblogging, wikis, rating, and reviewing. It reports that the percentage of scholars who use social networking apps (Facebook, LinkedIn) for professional purposes is much lower than the percentage of scholars who use it for personal reasons. It also points towards a large variability in usage of different platforms among scholars, with numbers ranging between 10% for Twitter, 46% for ResearchGate, and 55% for YouTube. Another systematic review analyzes social media use for public health communication among the general public, patients, and health professionals based on 98 original research studies [5]. These studies included a range of social media tools and apps, with Facebook, blogs, Twitter, and YouTube being the most often reported tools.

1.3 Related works

As mentioned before, some examples of worldwide known social media apps are Linked In, Twitter and Instagram and of course Facebook.

1.3.1 Facebook

Launched in a Harvard dormitory in 2004 by Mark Zuckerberg, Eduardo Saverin, Chris Hughes, and Dustin Moskovitz, the project initially had high school and college students in mind. It attracted seven million users in its first two years of existence. By February 2010 Facebook was reported to have more than 400 million active members [7].

Facebook users can create profiles with photos, lists of personal interests, contact information, and other personal information. Communicating with friends and other users can be done through private or public messages or a chat feature. Also, users can create and join personal interest and fan groups, some of which are maintained by organizations as a means of advertising. Unlike other social network sites, Facebook clearly distinguishes between brands and regular members. Instead of profiles, brands establish communities using Facebook pages.

For researchers, Facebook constitutes a rich site for those interested in studying the phenomena of social networks due to its heavy usage patterns and technological capacities that bridge online and offline connections.

Facebook is at the top of the social media game as its platform caters to a wide variety of people, incorporating many different media aspects, from photos to messenger to text. It is not as limited as LinkedIn and Twitter, which typically cater to a specific demographic. Because of its wide appeal, Facebook has attracted a significant user base, which translates to ad revenue, since companies desire to spend their ad budgets on platforms that receive the most viewership, and with 2.74 billion active users monthly, it's hard to top Facebook [8].

1.3.2 Instagram

Instagram is a photo sharing application (Instagram 2014) in which the user can share the world through their eyes to showcase what they find interesting or important. The majority of pictures range from mundane events, such as what the user is eating, reading or listening to, to the events attended by the user; parties, concerts, get together, etc. Instagram allows users to edit and upload photos and short videos through a mobile app. Users can add a caption to each of their posts and use hashtags and location-based tags to index these posts and make them searchable by other users within the app. Each post by a user appears on their followers Instagram feeds and can also be viewed by the public when tagged using hashtags or tags. Users also have the option of making their profile private so that only their followers can view their posts [9].

As with other social networking platforms, Instagram users can like, comment on and bookmark others posts, as well as send private messages to their friends via the Instagram Direct feature. Photos can be shared on one or several other social media sites including Twitter, Facebook and Tumblr with a single click.

Instagram is not only a tool for individuals, but also for businesses. The photo-sharing app offers companies the opportunity to start a free business account to promote their brand and products. Companies with business accounts have access to free engagement and impression metrics. According to Instagram's website, more than 1 million advertisers worldwide use Instagram to share their stories and drive business results. Additionally, 60% of people say they discover new products through the app. In addition, Instagram provides a wide range of digital filters that can be applied to users' photos [10].

1.3.3 Twitter

Twitter has become increasingly popular with academics as well as students, policymakers, politicians and the general public. Many users struggled to understand what Twitter is and how they could use it, but it has now become the social media platform of choice for many.

Twitter is a 'micro-blogging' system that allows you to send and receive short posts called tweets. Tweets can be up to 140 characters long and can include links to relevant websites and resources. Twitter users follow other users. If users follow someone, they can see their tweets in their twitter 'timeline' [11]. Users can choose to follow people and organizations with similar academic and personal interests to them. Users can create their own tweets or they can retweet information that has been tweeted by others. Retweeting means that information can be shared quickly and efficiently with a large number of people [12].

Twitter allows you to:

- easily promote your research, for example by providing links to your blog stories, journal articles and news items.
- reach a large number of people quickly through tweets and retweets.
- follow the work of other experts in their field.

- build relationships with experts and other followers.
- keep up-to-date with the latest news and developments, and share it with others instantly.
- reach new audiences.
- seek feedback about your work and give feedback to others.
- follow and contribute to discussions on events, for example conferences that you can't attend in person.
- express who you are as a person.

1.3.4 LinkedIn

LinkedIn is the world's largest professional network on the internet. Users can use LinkedIn to find the right job or internship, connect and strengthen professional relationships, and learn the skills they need to succeed in their career. Users can access LinkedIn from a desktop, LinkedIn mobile app, mobile web experience, or the LinkedIn Lite Android mobile app [13].

A complete LinkedIn profile can help users connect with opportunities by showcasing their unique professional story through experience, skills, and education. Users can also use LinkedIn to organize offline events, join groups, write articles, post photos and videos, and more.

LinkedIn is a platform for anyone who is looking to advance their career. This can include people from various professional backgrounds, such as small business owners, students, and job seekers. LinkedIn members can use LinkedIn to tap into a network of professionals, companies, and groups within and beyond their industry [14].

1.4 Added value

Most of social media applications focus on just that, making users to socialize with each other and use the application more often for that reason. In our application, one of the goals was for our users to interact with each other by making posts, comments and likes but also use our application for searching different recipes that provide a method and the list of ingredients. We hope our users not only use the application just to socialize with each other but to be provided with recipes information as well and for that reason use the application way more often, even when they want to cook food. For these reasons, our application differs from many others in the same category.

2. Technologies Used

Many different technologies were used in the application development process such as Firebase and the Services it offers, the Dart programming language and the Flutter Framework which will be analyzed in this section.

2.1 Dart & Flutter

It is important to introduce both Flutter and the Dart programming language. A common problem with native mobile development is that usually both an iOS app and an Android app need to be developed. This can be a costly and time consuming process. The application has been created in Flutter which allows us to have a unique code base for the front-end of the application [15].

Dart is a language with many similarities to Java, C # and Javascript. Its combination with Flutter makes application development much easier. Both Flutter and Dart are open-source projects from Google, and updates to the Dart language take Flutter into account. Dart supports both named and required parameters. Especially when designing layouts named parameters are frequently used. This feels like less of a cognitive load as opposed to using only required parameters, and more can be inferred by only looking at the code. Dart allows trailing commas after parameters, which is convenient when creating layouts in Flutter [16].

2.2 Introducing Flutter

Flutter uses Dart to create the user interface, removing the need to use separate languages. Flutter is declarative meaning that it builds the UI to reflect the state of the app. When the state changes, the UI is redrawn, and Flutter constructs a new instance of the widget. Widgets will be analyzed in the next section.

Flutter is fast, rendering runs at 60 frames per second (fps) and 120fps for capable devices. The higher the fps, the smoother the animations and transitions.

Another advantage of Flutter is that applications are built from a single code base. Flutter provides the developers with tools to create beautiful and professional looking applications, with the ability to customize any aspect of the application. Developers are able to add smooth animations, gesture detection and splash feedback to the User Interface. Furthermore, Flutter uses hot reload to refresh the running application in milliseconds when developers change the source code to add new features or modify existing ones. Using hot reload is a great way to see changes made to the code on the emulator or mobile device while keeping the application's state on the screen [17].

2.2.1 Flutter Widgets

The Flutter User Interface is implemented by using widgets from a modern reactive framework. Flutter uses its own rendering engine to draw widgets.

Widgets are similar to HTML elements. Adding the elements together, we can create an object and by adding different kinds of elements, we can alter the behavior of the object. Widgets are the building blocks of a Flutter app, and each widget is an immutable declaration of the User Interface. Widgets, in other words, are instructions for different parts of the UI. Placing the widgets together the widget tree is created.

A brief look to different widgets is presented below:

- Widgets with structuring elements such as text, grid, list and button.
- Widgets with input elements such as form fields and keyboard listeners.
- Widgets with styling elements such as font type, size, color, weight, border and shadow.
- Widgets to lay out the UI such as row, column, stack, centering, padding and list view.

- Widgets with interactive elements that respond to touch, gestures, dragging and dismissible.
- Widgets with animation and motion elements such as hero animation, animated container, rotation, scale, size, slide and opacity.
- Widgets with elements like assets, images and icons.
- Widgets that can be nested together to create the UI needed.
- Custom widgets that developers can create.

2.2.2 Stateless and Stateful widgets

In flutter there are two types of widgets, StatelessWidgets and StatefulWidgets to build the UI. A stateless widget is used when the state do not change and the stateful widget is used when the state changes during running of the application. The layout is defined using the functional composition within the build method found in both of these widgets. Instead of using a compelling style to define the user interface, Flutter uses a declarative programming style to define the user interface [18].

A simple example of using Flutter and Dart is shown below.

```
1  void main() => runApp(MyApp());
2
3  class MyApp extends StatelessWidget {
4          @override
5           Widget build(BuildContext context) {
6                    return MaterialApp(
7                          title: 'Widget Example',
8                          theme: ThemeData(
9                                    primarySwatch: Colors.blue,
10                          ),
11                         home: BasicWidgetExample("Hello World ", title: 'Material App'),
12                   );
13          }
14    }
```

2.2.3 Widget Lifecycle Events

There are in-built methods that developers can call if needed. Some of these functions have a specific time they are called when running the application. Every class have these methods, so depending of what is needed these methods can be called many times in different classes, resulting differently.

The life cycle is based on the state and how it changes. A statedul widget has a state so we can explain the life cycle of Flutter based on it. The stage of the life cycle is the following:

1. CreateState(): When we create a stateful widget, Flutter framework will instruct to create this method in override. This method will return an instance of a state associated with it.

```
class MyHomePage extends StatefulWidget {
      // This widget is the home page of your application. It is stateful, meaning
      // that it has a State object (defined below) that contains fields that affect
      // how it looks.

      // This class is the configuration for the state. It holds the values (in this
      // case the title) provided by the parent (in this case the App widget) and
      // used by the build method of the State.

  final String title;

   @override
  _MyHomePageState createState() {
    print('1. createState()');
    return _MyHomePageState();
  }
}
```

Figure 2.1 createState method

2. InitState(): This is a very important method, used almost in every single class, because is the method that is called first after the widget is created. It is called only once. It is used to initialize the data that can change on the widget.

```
@override
void initState() {
  print('1a. mounted :$mounted');
  print('2. initState()');
  super.initState();
}
```

Figure 2.2 initState method

3. DidChangeDependencies(): This method is called immediately after initState() on the first time the widget is built.

```
@override
void didChangeDependencies() {
  print('3. didChangeWidget()');
  super.didChangeDependencies();
}
```

Figure 2.3 didChangeDependencies method

4. Build(): This method is the most important one. It is called after didChangeDependencies(). The entire tree of widgets to be rendered relays on this method. This will be called every single time the UI needs to be rendered.

```
@override
Widget build(BuildContext context) {
  print('4. build');
  return ();
}
```

Figure 2.4 build method

5. DidUpdateWidget(): If the parent widget changes configuration and has to rebuild the widget the this method is called. When parent widget made a change and needs to redraw the UI to get the oldWidget parameter and so that compare it with the current widget to do some extra logic right there. It works every single time when we reload (hot reload) the app.

```
@override
void didUpdateWidget(MyHomePage oldWidget) {
  print('5. didUpdateWidget');
  super.didUpdateWidget(oldWidget);
}
```

Figure 2.5 didUpdateWidget method

6. SetState(): Another important method. This method is used to change the state when it is called. It is called inside a method and tells Flutter framework that something has changed in its state, which causes it to rerun the build method so that the display can reflect the updated values. If we want to change without calling setState(), then the build method would not be called again and so nothing would appear to happen.

```
setState(() {
  print('6.home :setState()');
});
```

Figure 2.6 setState method

7. Deactivate(): This is called when the state is removed from the tree, but it might be reinserted into another part of the tree before the current frame change is finished.

Figure 2.7 deactivate method

8. Dispose(): This one is important as well. It is called when this object and its state is removed from the tree permanently and will never build again which is the equivalent opposite of initState(). In this method usually we unsubscribe streams, dispose controllers and more.

Figure 2.8 dispose method

2.3 Firebase

Firebase is a development platform released in April 2012 and adopted by Google in 2014 as a back-end programming solution. The combination of features in Firebase automatically speeds up database integration in both Web and mobile applications. Firebase is divided into 3 sections: code development, profit and development. Depending on the properties of the application, the client could develop one of these 3 pillars or integrate all the functions of Firebase to synchronize the necessary data, resulting in a hassle-free experience for end users from the database to real Real-Time Database, Google Analytics, Messaging and even dynamic Links. Firebase users can upgrade their subscription to access its advanced features [19].

Firebase was chosen, among others, because it offers several services and is supported by Flutter. In addition, the policy followed for its price is pay-as-you-go, i.e. it results depending on its use, something that satisfied our wishes for the data of our application [20].

2.4 Firebase Services

Firebase services are Firebase Authentication, Cloud Firestore, Cloud Storage, Cloud Functions and Firebase Cloud Messaging, some of which will be discussed below.

2.4.1 Firebase Authentication

Most applications must have some form of user identification. This allows them to set their preferences, store data and provide personalized experiences that are consistent across all user devices. In order to provide this, they must enable new users to sign up, enable existing users to log in, manage their account information, and

keep all that data secure. It is a very difficult and time consuming process. Users tend to be reluctant to provide information that is personal data, such as username, password, security questions, and anything else that might be leaked. As a result, they often prefer to use credentials that they have already provided to a service, and that service manages the connection for them. So, for example, if they have a Facebook account, they would like to use Facebook to verify their data in another application that receives this type of verification, without having to give you further information [21].

Firebase Authentication offers the ability to use different providers as a connection method. From May 2019, it offers connection via Email / Password, Phone, Google, Play Games, Game Center, Facebook, Twitter, GitHub, Yahoo, Microsoft and Anonymous login. Not everything was available at the beginning of the project, however there is some preparation involved with adding each of them to the application. Thus, only a subset of them was selected, initially only a Google and Facebook connection.

In the development of our application, some of the most popular providers were used, such as Google and Facebook, as well as a connection with Email / Password.

2.4.2 Cloud Firestore

Cloud Firestore is a NoSQL database. Some of the advantages of this is that it does not need management infrastructure unlike SQL database applications. For example, Firestore scales horizontally, adding more machines to meet the size and requirements of the database.

However, data modeling in Firestore is different from data modeling in SQL databases. Demodulation is perceived in SQL databases, but in NoSQL databases, this is critical to its performance as it can reduce the number of queries required to complete a task.

2.4.2.1 Querying Firestore

Firestore imposes some restrictions on database documents. Documents have a limit of 1MB in size, in addition, it can not have more than 40,000 index fields. Firestore uses indexes for everything in documents to provide excellent read speeds through a binary search algorithm. Writing speed is a bit slower, but readings are usually more frequent. The queries are quite easy. It is important to note that writing a query varies slightly depending on the language, such as whether it is written in the Dart programming language for Flutter clients or in TypeScript for Cloud Functions. Equality searches can become quite specific when creating a query. Figure 2.9 shows an example query in Firestore.

```
static Stream<QuerySnapshot> getReminderStream(String goalId) {
  return collectionAccount()
      .document(_account.accountId)
      .collection("reminders")
      .where("goalId", isEqualTo: goalId)
      .where("canceled", isEqualTo: false)
      .orderBy("timeToRemind", descending: true)
      .snapshots();
}
```

Figure 2.9 Query example

However, there are some drawbacks to this approach, because queries can not have "and", "or" or "not equal to" functions in them when searching for a single field. This is because queries must return a set of documents between a start index and an end index. It is not possible with the binary search algorithm used behind the scenes to return multiple sets of documents to a query. If required, the results of many queries should be processed instead [22].

Additionally, Firestore can generate advanced indexes if there is a need for advanced queries. Advanced indexes are generated when needed and managed automatically by Firestore. Data in Firestore is protected by security rules. They specify who is allowed to access them in the database. An example of such security rules is shown in Figure 2.10 where database entries and data readings are allowed until the date shown.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2021, 9, 18);
    }
  }
}
```

Figure 2.10 Security Rules

2.4.2.2 Firestore Transactions

Firestore supports transactions. Transactions are a safe way for a user to read or write data to the database with accuracy even when another user is writing data at the same time or the connection is lost. For a transaction to succeed, the documents retrieved by its read operations must remain unmodified by operations outside the transaction. If another operation attempts to change one of those documents, that operations enters a state of data contention with the transaction. For example, one transaction might require a document to remain consistent while a concurrent operation tries to update that document's field values.

Cloud Firestore resolves data contention by delaying or failing one of the operations. The Cloud Firestore client libraries automatically retry transactions that fail due to data contention. After a finite number of retries, the transaction operation fails and returns an error message.

2.4.2.3 Firestore compared to MongoDB

Both engines are NoSQL databases because of the document-object data structure they use to store data. This means that the way that the data is stored is the same as the way JavaScript handles objects. For example, MySQL uses tables as an entity, while MongoDB and Firestore use objects called documents as entities. One thing to keep in mind is that MongoDB accepts query language to retrieve data, but Firestore has its own method and API calls to do so [23].

2.4.2.3.1 Firestore and MongoDB Similarities

Both MongoDB and Firestore are managed services. They offer the developer the liberty to spend more time working with data and less time configuring the server and infrastructure. Google data centers are distributed on every continent. MongoDB also has its own global clusters around the world. MongoDB has its own metrics and alerting service. Firestore utilizes Stackdriver to accomplish the same functionality. Event-driven triggers in MongoDB can be related to cloud Pub/Sub in Firestore. This means that in MongoDB, you can create triggers that start working when some rules are fulfilled in the database and respond in real-time with server-side logic,

while cloud Pub/Sub can be configured to listen to specific events in the whole cloud to execute Cloud Functions logic similar to the event-driven triggers in MongoDB [24].

2.4.2.3.2 Firestore and MongoDB Differences

In order to understand the differences between these two technologies, we have to emphasize the most important features that make each unique.

- The Mongo service offers automated daily backups to the databases managed by the Atlas service. It also offers incremental data recovery and consistent, cluster-wide snapshots of shared clusters. Firestore's similar service, "Scheduled export", has to be configured and implemented manually by the administrator. In the other hand, Firestore can be configured to export all its data to a cloud storage bucket, almost like a backup but harder to recover in a case of a failure or the need to roll back the data because you have to manually update all your database, in this case, MongoDB feature is better for backing up.

- Mongo has a unique level of security for its Atlas service. Firestore has its own security rules that work slightly differently:

    1. Rules by document: Firestore can assign user-specific data and protect it from other users from reading. This functionality can be accomplished by Mongo, creating as many database users as final users, which can be a struggle in many cases.

    2. Connection: Mongo uses its own port, which can be configured to use SSL/TLS. Firestore uses https request to its API.

    3. Private Link: While Firestore works over the Google Cloud Platform and all its services are connected internally inside the Google Cloud, Mongo uses Private Link, which provides exclusive access to AWS.

- One of the most important features that distinguishes Firestore from other services is real time updates. The capability to update and listen to changes in the database in real-time is a pretty good feature when working with web/mobile applications, giving all the users the feeling they are updated all the time. Mongo can be integrated with other technologies to make this possible, but it requires a lot of work and a very specific infrastructure. For example, an application that keeps a user's to-do list and syncs between the mobile and web versions of the app would best utilize Firebase because of its ability to provide real-time updates.

- Working with Firestore gives the developer the ability to enable offline support to save data locally and then sync it with the cloud whenever the connection is back. Offline data is not new to web and mobile applications; the difference here is that this feature already comes out of the box using the Firestore library. Let's imagine that you have to create a web application for local businesses to manage their inventory, sales, and providers. This sounds like a more robust application, so for this multi-purpose approach you should be using MongoDB, since it can be installed in your own server and doesn't need to have internet access to store or process data. MongoDB can also be installed in Linux, Windows and OS X, which means a server application can be developed locally without needing any remote connections.

- Firestore supports transactions while Mongo does not.

2.4.2.3.3 Conclusion

Firestore offers scalability, offline support, real time updates, transactions and lots of features while it is easy to use and assimilated in a Flutter mobile application. Firebase in general, offers a lot of services combined in a single platform making it easy for a programmer to focus on the development of the app rather than the infrastructure. Finally, the pay-as-you-go pricing for the platform suits well for our needs in the development of our application. That said, Firestore and Firebase was an obvious and clear choice for our mobile application.

2.4.3 Cloud Storage

As the image resources are quite large, they would be unsuitable for storage in the Cloud Firestore, as the documents are limited to 1MB. Fortunately, there is Cloud Storage. Cloud Storage is a simple file system where various files can be stored, commonly used for pictures or videos. Cloud Storage is as secure as Cloud Firestore, i.e it uses security rules.

Developers use the Firebase SDK for Cloud Storage to upload and download files directly from clients. If the network connection is poor, the client can resume the operation where it left off, saving users time. Cloud Storage stores files in a Google Cloud Storage bucket, making them accessible via Firebase and Google Cloud. This allows the user the flexibility to upload and download files from mobile clients via the Firebase SDK. Cloud Storage scales automatically, which means there is no need to relocate to any other provider [25].

2.4.4 Cloud Functions

Cloud Functions are triggers that occur in response to an event. Some examples are when CRUD (Create, Read, Update, Delete) functions occur in documents, when a URL receives an https call, or when a new user is created. The advantage of using Cloud Functions is that we can have an application without a server (Server-less). Cloud functions are written in either TypeScript, Python or Go. Note that in this application TypeScript was used to create the Cloud Functions which will be presented later [26].

The life cycle of a Cloud Function is shown below.

1. We write the code for a new function by selecting a provider (such as the Cloud Firestore) and specifying the conditions under which the function is to be executed.

2. After the deploy of the Function:

    - Firebase CLI creates a .zip file of the operation code, which is then downloaded to a Cloud Storage bucket.

    - Cloud Build retrieves the passcode and creates the source of the function.

3. When the event provider creates an event that matches the conditions of the function, the code is called.

4. If Function is busy handling multiple events, Google generates more impressions to handle the process faster. If the Function is inactive, the instances are deleted.

5. When a Function is deleted, the zip files are cleared. The connection between the Function and the event provider is removed.

An example of a Cloud Function is shown in Figure 2.11.

```
export function onNewChallenge() {
    return functions
        .region('europe-west1')
        .firestore
        .document('/challenge/{challengeId}')
        .onCreate(async (snapshot, context) => {
            const data = snapshot.data();
            if (data !== undefined) {
                const accountId = data.creatorId;
                await rewardPoints(accountId, PointType.Creativity, 5, null);
            }

        })
}
```

Figure 2.11 Cloud Function

2.5 Web-Scraping

Plenty of useful data can be found on the internet and specifically on websites which are accessible to the public at no cost. However, in order for this data to be considered useful and for it to be reusable it must be exported in some way. The process of extracting this data from websites is known in programming as Web Scraping and is becoming more and more useful and important as the information available on the internet is growing rapidly.

It should be noted that Web Scraping is not always legal. If the exported data is for personal use then there is no restriction. However, if this data were to be reused and made available to the public then it is very important the type of data and whether it exposes personal data and information of either companies or individuals as units.

In any case, the process of extracting data from websites should be done in a polite way as the developer is a visitor of the website. One way to check if a website owner allows or not to run data from their website is the robots.txt file in which the websites that own it define the information and data that developers can get [27].

2.5.1 Web-Scraping in Python

Python has several libraries for the Web Scraping process. It is worth noting some of them such as Requests, Beautiful Soup 4 (known as BS4), lxml, Selenium and Scrapy. The Beautiful Soup 4 library and Selenium were used in this project.

2.5.2 Beautiful Soup 4

The Beautiful Soup library strives to beautify, give meaning to, and organize the cluttered HTML that governs web pages. This library is not a Python key which means it needs to be installed. Installing libraries in Python is usually done with pip. The installation command is -> pip install BS4

An example of reading a web page is shown in Figure 2.12.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
print(bs.h1)
```

Figure 2.12 Website Reading with BS4

2.5.3 Selenium

Selenium Library is a set of tools for automating browsers. It is mainly used for application control but its use is not limited to control. It can also be used to export data by automating some processes in a Browser window. Supports automation in all known Browsers including Firefox, Internet Explorer, Google Chrome, Safari and Opera. In the present project the use of the library was made with the aim of the automated pressing of the buttons for page change (where necessary) so that the recipe data can be carried out without the constant supervision of the process by us. This was achieved with Selenium WebDriver, a tool provided by the library.

Similar to BS4, the Selenium library should be installed with the command -> pip install selenium

A simple example of opening a window in the Firefox browser using the Selenium Library is shown in Figure 2.13.

```
# create a new Firefox session
driver = webdriver.Firefox()
driver.implicitly_wait(30)
driver.maximize_window()
```

Figure 2.13 Selenium-Firefox

2.6 Algolia Search Engine

The Algolia model provides search as a service, offering web search across a client's website using an externally hosted search engine. Algolia provides software and tools to help us implement efficient, flexible, and insightful search on our application.

Algolia provides users with a fast and rich search experience. Algolia search interface can contain a search bar, filters, infinite scrolling, query suggestions, sorting, refinements, etc. These help our users find what they're looking for and discover new searches.

Algolia provides a set of tools that simplify the process of making and integrating a full search experience into applications. These include:

- Back-end API clients, in many different languages, to index, configure, and manage data.
- Front-end widgets to build web and mobile search experiences.
- Integrations with popular frameworks and platforms, to further simplify the integration of Algolia.

- A secure, distributed search network that hosts content and serves it to customers quickly.
- A transparent, customizable relevance algorithm.
- A heavily optimized search engine built, from scratch, in C++.
- Extensive documentation, implementation guides, and code examples.

Algolia is used in our application so that our users can search for other users and search for recipes based on the ingredients a user selects.

3. Application Analysis

3.1 Requirements

This section will analyze the system requirements. Application requirements can be divided into functional and non-functional requirements. Functional requirements define the capabilities and functions that the system should be able to perform successfully. Non-functional requirements are defined as criteria and properties that can be used to evaluate system performance.

3.1.1 Functional Requirements

Below are the functional requirements of the application.

1. Users will be able to create an account in the application.
2. Users will be able to sign in using either their Gmail or Facebook account.
3. Users will be able to log out of their account.
4. Users are requested to verify their email after creating an account.
5. Users will be able to retrieve the forgotten password.
6. Users will be able to state a name that represents them in the application and will be able to change it through the application after logging in.
7. Users will be able to follow other users of the application as well as stop following them respectively.
8. Users will be able to upload a post in either image or video format.
9. Users will be able to report a Post or user account for unwanted content/behavior.
10. Users will be able to edit the uploaded image or video during the Post.
11. During the Post, users will be able to state any location they wish as well as add a caption and tag recipe.
12. Users will be able to delete / edit their Posts and disable comments on them.
13. Users will be able to comment to posts.

14. Users will be able to search for recipes in the application’s database and access them with all the necessary information.
15. Users will be able to search for other users.
16. Users will receive notifications within the application regarding comments, likes, mentions and follows.
17. Users will receive notifications according to the app’s life cycle state (background, foreground, terminated), in all different cases.
18. Users will be able to add a recipe to their favorites and share it as a pdf file.
19. Users will have access to other users' profiles and their information (posts, followers, bio).
20. The application should check the user's connectivity and inform him with a message in case of internet interruption.
21. The application should provide messages (toasts) to the user for various functions(name length, password).

3.1.2 Non Functional Requirements

Listed below are the non-functional requirements based on their importance:

- Usability: is the ease with which a user can learn to operate, prepare inputs and interpret system output.
- Modification / Scalability: The ability of the system to change easily to meet new requirements.
- Portability: The ease with which a system or component can be transferred from one environment to another.
- Flexibility: The ability of the system to easily exchange information with the user.
- Reliability: is the ability of a system to perform its required functions according to the conditions for a specific period of time.
- Security: login, passwords.
- Performance: refers to the operating speed of a system.
- Total cost: The total cost of the project in terms of price and time.

The non-functional requirements of our application are:

- The graphical interface of the application should be designed to provide a pleasant feeling to the user and will be easy to use.
- Alerts should be clear and restricted so as not to repel the user.

- The application should have no errors in the code and the user should be informed of any errors and the reason behind it.
- Database operations and interactions should be performed in the best possible way so as not to burden the performance and speed of the application.

3.2 Design

All the basic functions of the application and the data that have been collected or will be stored by the users happen with the help of Firebase. Recipe information is stored in the Fire Store while new users, their Posts, their notifications and their reports are stored. Images and videos of users are stored in Storage. Notifications are made with the help of Cloud Functions which are triggered when a user interacts with them. A basic structure of the application is shown in Figure 3.1.

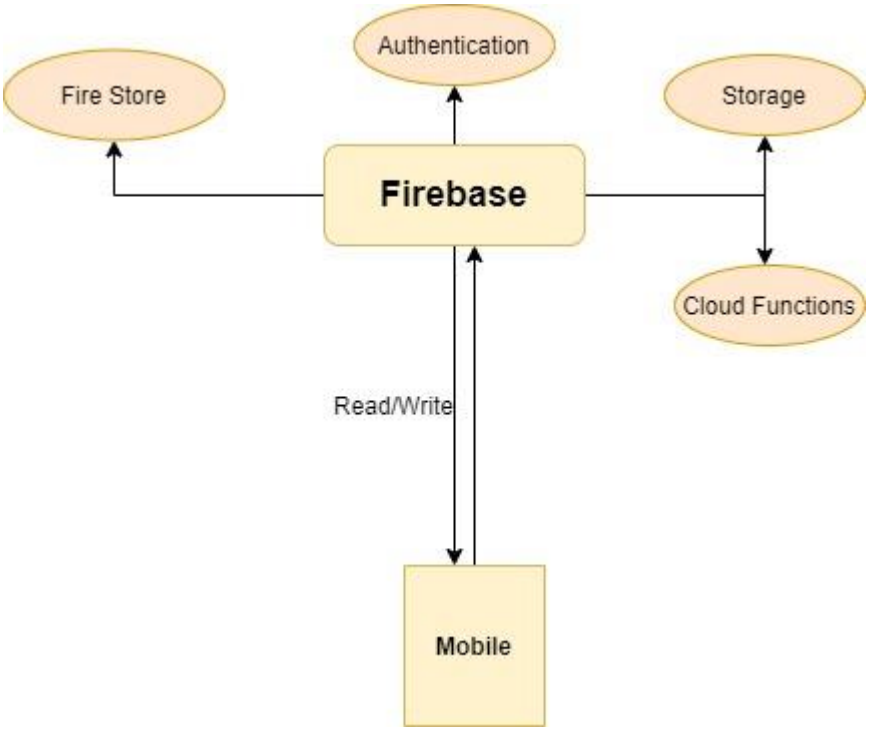


Figure 3.1 Basic Design

3.3 Database Structure

The database consists of collections. Its structure is presented below.

- Collection Users. Here the users information is stored.

Users(Collection)

Uid(Document)

{

activity: array,

androidNotificationToken: String,

displayName: String

email: String,

favouriteRecipes: array,

followers: array,

following: array,

id: String,

photoUrl: String,

provider: String

}

Each field and its description is shown in Figure 3.2.

activity: array	A list of all notification id's in order to display to the user
androidNotificationToken: String	The token needed for displaying notifications
displayName: String	User's in-app name
email: String	User's email
favouriteRecipes: array	List of the favorite recipes a user has selected
followers: array	List of users followed by
following: array	List of users following
id: String	Unique identification code
PhotoUrl: String	Photo url of the user's profile picture
provider: String	Refers to the registration provider(Google, Facebook, Email/Password)

Figure 3.2 users collection

- Collection user_posts. Posts information that users upload is stored in this collection.

User_posts(Collection)

Post_id(Document)

{

commentsAllowed: bool,

description: String,

likes: array,

location: String,

mediaType: String,

mediaUrl: String,

ownerId: String,

postId: String,

refferedRecipeInfo: map,

thumbNailUrl: String,

timestamp: Timestamp,

username: String

}

commentsAllowed: bool	A boolean field which determines whether the comments should be allowed or not
description: String,	The description of the post
likes: array	A list of user id's that liked the post
location: String	The location of the post
mediaType: String	Whether the post contains a picture or a video
mediaUrl: String	The url of the picture or video that is stored in Storage
ownerId: String	The unique identification number of the post owner
postId: String	Unique identification code of the post
refferedRecipeInfo: map	Map that contains recipe info tagged in the post
thumbNailUrl: String	Url of the picture in case of a video in post
timestamp: Timestamp	The time that the post is shared
username: String	The display name of the owner of the post

Figure 3.3 user_posts collection

- Collection recipes. The recipes information is stored here.

recipes(Collection)

chef(Document)

category(collection)

recipe_title(Document)

{

cooking_method: array,

cooking_time: array,

ingredients: array,

image_url: String,

rating: number,

recipe_title: String,

recipe_url: String,

users_rating: array,

}

cooking_method: array	The method described for the recipe
cooking_time: array	The cooking time fro the recipe
ingredients: array	A list of ingredients of the recipe
image_url: String	The image url for the recipe
rating: number	The total rating of the users for the recipe
recipe_title: String	The title of the recipe
recipe_url: String	The url that leads to the original site the recipe was taken from
users_rating: array	A list that contains all user id's that rated the recipe

Figure 3.4 recipes collection

- Collection cookit_comments. Comments in users Posts are stored here.

Cookit_comments(Collection)

id(Document)

comments(collection)

commetn_id(Document)

{

comment: String,

comment_id: String,

likes: array,

post_id: String,

user_id: String,

timestamp: Timestamp,

}

comment: String	The comment's content
comment_id: String	The identification number of the comment
likes: array	A list of user id's that liked the comment
post_id: String	The identification number of the post
user_id: String	The identification number of the user that created the comment
timestamp: Timestamp	The time the comment uploaded

Figure 3.5 cookit_comments collection

- Collection cookit_a_feed. Notifications that users receive are stored here.

Cookit_comments(Collection)

id(Document)

items(collection)

notification_id(Document)

{

mediaUrl: String,

post_id: String,

likes: array,

thumbNailUrl: String,

type: String,

timestamp: Timestamp,

userId: String,

userProfileImage: String,

username: String

}

mediaUrl: String	The url of the image or video that got commented or liked

post_id: String	The identification number of the post that is liked or commented
likes: array	A list of user id's that liked the post
thumbNailUrl: String	The image url of the picture in case a video is posted
type: String	Video or picture
timestamp: Timestamp	The time the video is uploaded
userId: String	The identification number of the user that created the notification
userProfileImage: String	The url of the user's profile picture
username: String	The display name of the user that created the notification

Figure 3.6 cookit_a_feed collection

- Collection user_reports. Reports that users make in a post or a comment are stored here.

user_reports (Collection)

user_id(Document)

post_reports (collection)

post_id(Document)

{

type: String,

post_id: String,

postOwnerId: String,

reportingUserId: String,

timestamp: Timestamp,

comment: String,

commentId: String,

commentOwnerId: String

}

type: String	The reporting reason

post_id: String	The identification number of the post
postOwnerId: String	The identification number of the post's owner
reportingUserId: String	The identification number of the reporting user
timestamp: Timestamp	The time that the report happened
comment: String	The description of the comment in case a comment report
commentId: String	The identification number of the comment
commentOwnerId: String	The identification number of the comment owner's id

Figure 3.7 user_reports collection

4. Application Development

This section will present the application development process, code snippets and the techniques used.

4.1 Create Firebase Project

After installing Android Studio, the Firebase Project called FoodApp was created. At https://console.firebase.google.com/ and after logging in to your Google Account, we add a new Project as shown in Figure 4.1.

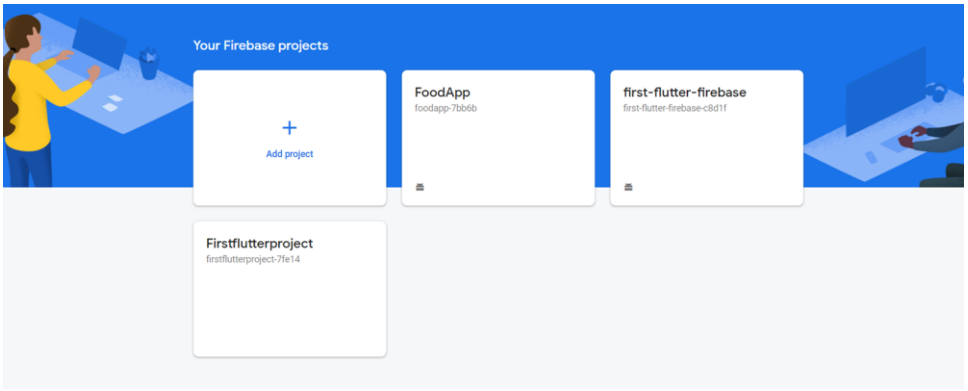


Figure 4.1 Creating Project

Then, after selecting the account to save the Project and having the Google Analytics option selected, which will provide charts for using the database, the Project is created after some time. To add an application to Project, select the Android icon as shown in Figure 4.2.

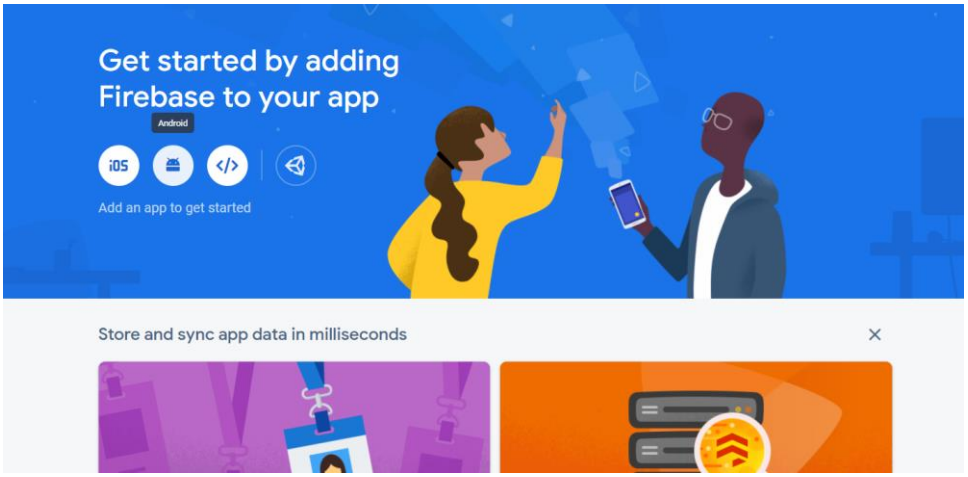


Figure 4.2 Add application

Next, select the application and package name as shown in Figure 4.3.

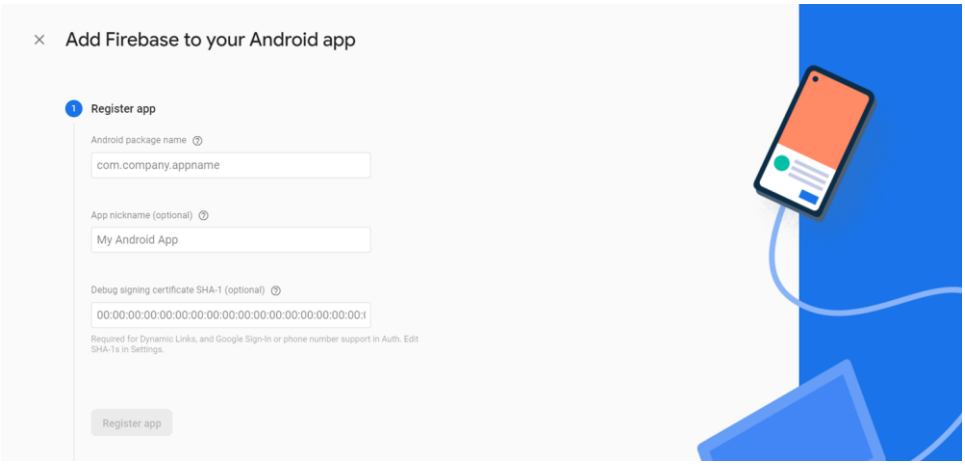


Figure 4.3 Selecting an application name

After this step, a google-services.json file is generated which has all the necessary information to connect Android Studio to Firebase and should be placed in the path suggested as shown in Figure 4.4.

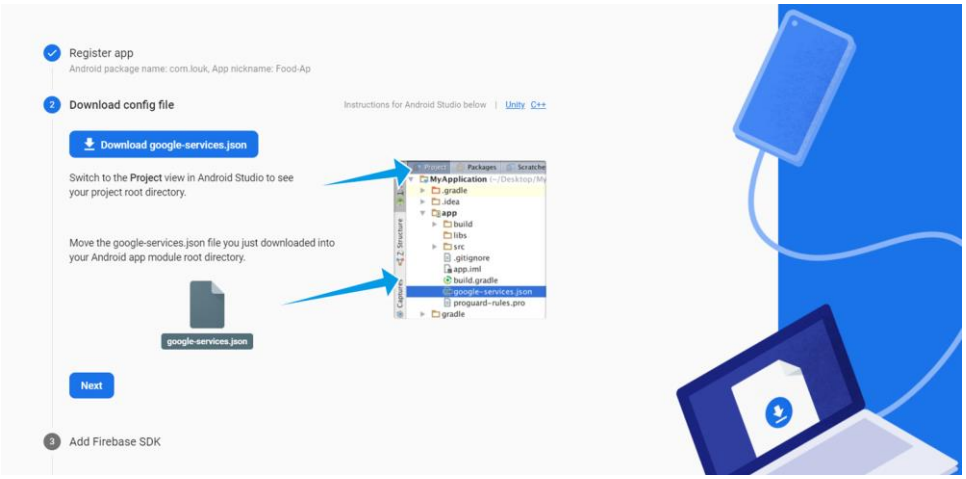


Figure 4.4 The google-services.json file

Finally, the code series should be added to the Android Studio Project in the files suggested in Figures 4.5, 4.6.

3 Add Firebase SDK

Instructions for Gradle | Unity C++

The Google services plugin for Gradle loads the `google-services.json` file you just downloaded. Modify your build.gradle files to use the plugin.

Project-level build.gradle (`<project>/build.gradle`):

```
buildscript {
  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
  }
  dependencies {
    ...
    // Add this line
    classpath 'com.google.gms:google-services:4.3.8'
  }
}

allprojects {
  ...
  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
    ...
  }
}
```

Figure 4.5

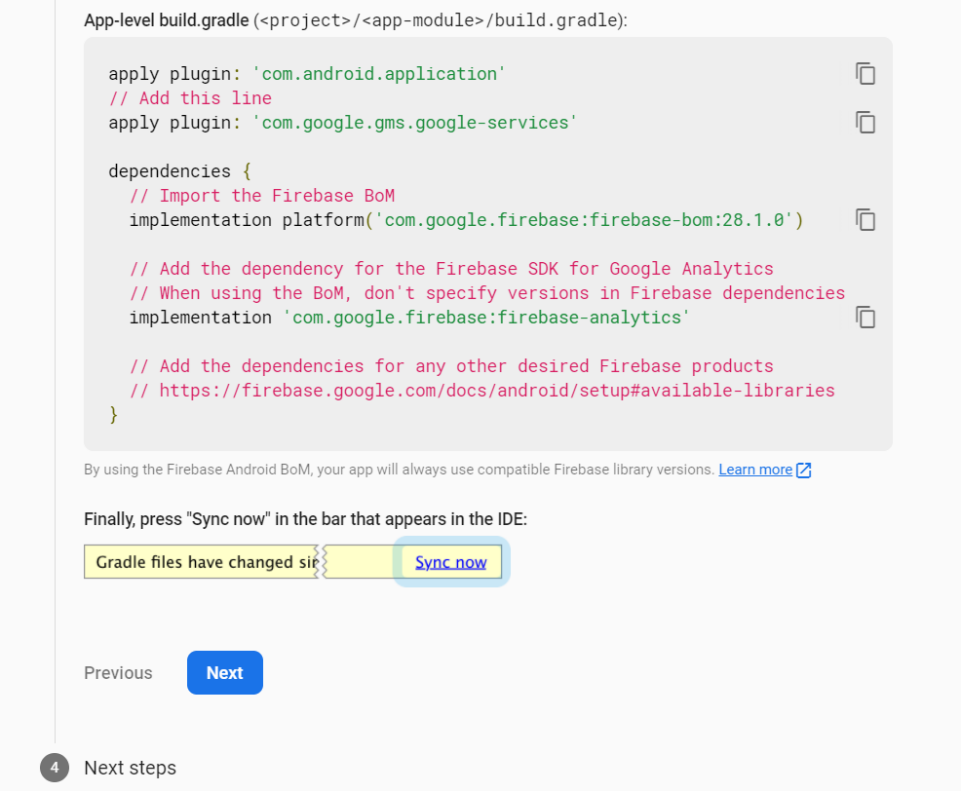


Figure 4.6

After adding the code, the Firebase Project and that of Android Studio are now connected.

4.2 Developing Process

As mentioned in the previous chapter, in order to fulfill the requirements, the application was splitted in different components. These are the authentication which includes the log in and sign up screen, the home screen that includes the socializing part of the application, the recipes search as well as any in app and system notifications.

4.2.1 Authentication Component

The general idea was to build a screen that includes different ways of authentication in order to log in in our application as shown in Figure 4.7.

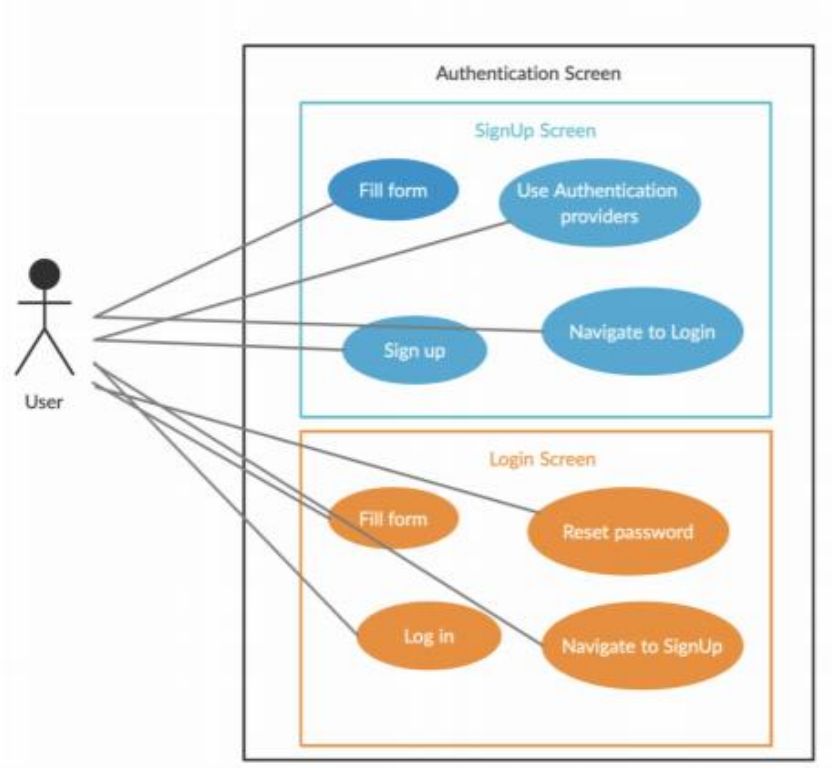


Figure 4.7

Three different ways that a user can sign up or sign in was selected. These are the following:

- Sign Up/ Sign In with Google.
- Sign Up/Sign In with Facebook.
- Sign Up/Sign In with Email/Password.

4.2.1.1 Sign Up/ Sign In with Google

To achieve google sign up with Firebase, a pop up window displays after users select this sign up option. After choosing the preferred email, a different page is presented and users are asked to fill a display name and pick a profile picture (if they want to). After the completion of the form, some actions are triggered.

The first action refers to the upload of user's photo for his profile picture in Firebase Storage with the _uploadGoogleImageToFirebaseStorage function. The content of this function is shown below:

```
var firebaseStorageRef=FirebaseStorage.instance
  .ref()
  .child('Google/${widget.gMail}/ProfilePic.png');
var uploadTask = firebaseStorageRef.putFile(utils.imageFileToFireStoreGoogle);
await uploadTask.then((res) async {url = await res.ref.getDownloadURL();
});
```

When this function completes, the url that refers to the path of the photo is saved in order to be used afterwards.

After the completion of the previous function the user's data are uploaded in Firestore. To achieve that, the _addGoogleUser function is triggered. The different fields such as id, email, displayName, as well as the photoUrl that is included in the url variable we saved with the _uploadGoogleImageToFirebaseStorage function, and others, are added to the users collection, inside a document that belongs to the specific user. The code sample is shown below:

```
FirebaseFirestore.instance.collection('users')
   .doc("${utils.auth.currentUser.uid}")
   .set({
  "id": utils.auth.currentUser.uid,
  'email': widget.gMail,
  'displayName': _googleDisplayName,
  'photoUrl': url,
  'provider': "google",
  'followers':[],
  'following':[],
  "favouriteRecipes":[],
  "activity":[],
  "androidNotificationToken":tokenRef

})
```

After this step, the user is navigated to the home screen.

In order for the user to sign in with Google they have to choose an existing email that they have already filled the form and completed the registration. The googleSignIn method is triggered at this occasion. The content of the method is shown after:

```
final GoogleSignInAccount googleSignInAccount =
await utils.googleSignIn.signIn();

final GoogleSignInAuthentication googleSignInAuthentication =
await googleSignInAccount.authentication;

final AuthCredential _credential = GoogleAuthProvider.credential(
  accessToken: googleSignInAuthentication.accessToken,
  idToken: googleSignInAuthentication.idToken,
)
```

This function also includes checks for whether a user has already an account for the specific email by a different method, meaning either Facebook or Email/Password. To achieve that, a call in Firestore is performed. In case the user owns an existing account for that email, a message appears on the screen to inform him. The code sample of this process is described below:

```
final search = await FirebaseFirestore.instance
   .collection('users')
   .where('email', isEqualTo: '$_gMail')
   .get();
var provider;
search.docs.forEach((res) {
```

```
  provider = res.data()["provider"];
});


if (provider == "email" || provider == "facebook") {
  toastMessages("Email already in use for another account.");
  utils.googleSignIn.signOut();
  _forkProgressIndicator().hide();
  this._loginForkController.dispose();
} else {
  if (search.docs.isEmpty) {
    return _goToGoogleScreen(
        context, _gMail, _gName, _gPhotoUrl, _credential);
  } else {
    try {

      Navigator.push(context,PageTransition(type:PageTransitionType.fade,alignment:Alignment.bottomCenter,
child:splash.SplashScreenMain()));
      await FirebaseAuth.instance
          .signInWithCredential(_credential);

    } catch (e) {

      toastMessages("Something went wrong");

    }
  }
}
```

4.2.1.2 Sign Up/ Sign In with Facebook

A very similar process to Google Sign Up/Sign In is followed for this case also. The main difference in order to accomplish Facebook sign up in Flutter is that developers have to create an account in https://developers.facebook..com and link the specific project to this account. After we created the account and linked the project we were able to add the Facebook sign up process in the application.

To achieve Facebook sign up with Firebase, a pop up window displays after users select this sign up option. After choosing their Facebook linked email, a different page is presented and users are asked to fill a display name and pick a profile picture (if they want to). After the completion of the form, the same actions as Google sign up are triggered.

The first action refers to the upload of user's photo for his profile picture in Firebase Storage with the _uploadFacebookImageToFirebaseStorage function. The content of this function is shown below:

```
var firebaseStorageRef = FirebaseStorage.instance
    .ref()
    .child('Facebook/${widget.fbProfile["email"]}/ProfilePic.png');
var uploadTask = firebaseStorageRef.putFile(utils.imageFileToFireStoreFB);
await uploadTask.then((res) async {
  url = await res.ref.getDownloadURL();
  print(url);
});
```

After the completion of the previous function the user's data are uploaded in Firestore. To achieve that, the _addFacebookUser function is triggered. The different fields such as id, email, displayName, as well as the photoUrl that is included in the url variable we saved with the _uploadFacebookImageToFirebaseStorage function, and others, are added to the users collection, inside a document that belongs to the specific user. The code sample is shown below:

```
utils.users
 .doc("${utils.auth.currentUser.uid}")
 .set({
 "id": utils.auth.currentUser.uid,
 'email': widget.fbProfile["email"], // John Doe
 'displayName': _fbDisplayName, // Stokes and Sons
 'photoUrl': url, //widget.fbProfile["picture"]["data"]["url"] // 42
 'provider': "facebook",
 'followers':[],
 'following':[],
 "favouriteRecipes":[],
 "activity":[],
 "androidNotificationToken":tokenRef
})
```

After this step, the user is navigated to the home screen.

In order for the user to sign in with Facebook they have to choose an existing email that they have already filled the form and completed the registration. The facebookSignIn method is triggered at this occasion. The content of the method is shown after:

```
final FacebookLoginResult result = await utils.fbLogin.logIn(["email"]);
switch (result.status) {
  case FacebookLoginStatus.loggedIn:
    final token = result.accessToken.token;
    final response=await
http.get(Uri.parse('https://graph.facebook.com/v2.12/me?fields=name,picture.width(800).height(800),first_name,last_
name,email&access_token=${token}'));
    final profile = jsonDecode(response.body);
    print(profile);
    final search = await FirebaseFirestore.instance
        .collection('users')
        .where('email', isEqualTo: '${profile["email"]}')
        .get();
    var provider;
    search.docs.forEach((res) {
      provider = res.data()["provider"];
    });
    if (provider == "email" || provider == "google") {
      toastMessages("Email already in use for another account.");
      _forkProgressIndicator().hide();
      this._loginForkController.dispose();
    } else {
      if (search.docs.isEmpty) {
        _goToFbScreen(context, profile, token);
```

```
    } else {
      try {

        final AuthCredential credentials =
        FacebookAuthProvider.credential(token);
        _forkProgressIndicator().hide();
        _loginForkController.dispose();
        Navigator.push(context, PageTransition(type: PageTransitionType.fade,alignment:Alignment.bottomCenter, child:
splash.SplashScreenMain()));
        await FirebaseAuth.instance
            .signInWithCredential(credentials);

      } catch (e) {
        print(e);
        toastMessages(e.code);
      }
    }
  }
  break;
```

4.2.1.3 Sign Up/ Sign In with Email/Password

The last sign up method is sign up with Email and Password. If a user select this option, a different screen appears and the user has to fill a form providing his personal email, a password and the preferred display name. After the completion of the form the user is created in Firebase and data uploaded in Firestore similar to the other sign up options. After this step, a verification email is sent and users must follow the provided link in that email. In addition, in order to send an email verification, user must be signed in. This was a problem for the application because we check for the state of the user, if he is signed in or not, in order to either push him in the home screen or the log in screen. To get over such a problem we sign in the user, send the verification email, upload user's data in Firestore and then sign out. Also checks if the password is too weak or the email already exists or a field is empty, are provided and messages appear in any case. This process is shown below:

```
await FirebaseAuth.instance
    .createUserWithEmailAndPassword(
    email: _signUpEmailController.text,
    password: _signUpPassController.text)
    .then((value) async{
  User user = FirebaseAuth.instance.currentUser;


  await _uploadEmailImageToFirebaseStorage();
  if (!user.emailVerified) {
    utils.auth.currentUser.sendEmailVerification();
    await _addEmailUser();
    utils.auth.signOut();


    Navigator.of(context).pop();
    FocusScope.of(context).requestFocus(new FocusNode());
    _displaySnackBar(context);
```

```
    }

  });
} on FirebaseAuthException catch (e) {
  if (e.code == 'weak-password') {

    _changeTextFieldBoarderColor("red");
    toastMessages('The password provided is too weak.');

    FocusScope.of(context).requestFocus(_signUpPassNode);
  } else if (e.code == 'email-already-in-use') {

    _changeTextFieldBoarderColor("red");
    toastMessages('The account already exists for that email.');
    FocusScope.of(context).requestFocus(_signUpEmailNode);
  } else {
    Future.delayed(Duration(milliseconds: 500)).then((value) {
      _forkProgressIndicator().hide();
      _signUpForkController.dispose();
    }).whenComplete(() {
      toastMessages(e.code);

    });
  }
```

Users can sign in providing their email and password only if their email is verified. After a period of time, the verification email expires, so users can resend the email again through the application. If a user forget the password, there is the option to reset it by sending an email to their email through the application that includes the steps to achieve that.

4.2.2 Authentication State

This chapter explains the connection between log in screen and main screen. In order for the user to sign in and navigate to the home screen a stream has been used. A stream in dart listens whenever something changes its state. In our case we had to listen to a stream and know whether a user is signed in or signed out from an account. The stream we used listens to authStateChanges() function and has the following form:

```
Stream<User> get _onAuthStateChanged => FirebaseAuth.instance.authStateChanges();
```

This stream is provided to a StreamBuilder widget of Flutter and in any case returns the appropriate screen to user based on the state the user is in.

The Stream Builder is shown below:

```
StreamBuilder(
    stream: _onAuthStateChanged,
    builder: (context, AsyncSnapshot snapshot) {
      if (utils.auth.currentUser != null) {
        if (snapshot.connectionState == ConnectionState.active &&
            utils.auth.currentUser.emailVerified ||
            FirebaseAuth
                .instance.currentUser.providerData[0].providerId ==
                'facebook.com') {
```

```
        return MainScreen();
       }
       else {
        return Login();
       }
      } else {
       return Login();
      }
     },
    ),
```

Before navigating to the Main Screen, we always check if the email is verified. Using this stream is easier to navigate users to the appropriate screens without needed to perform different checks of user's current state since the stream triggers immediately after the state changes.

4.2.3 Socialization Component

The next step after a user sign in with any provider is the home screen of the app. The functionality of the app is dedicated in socializing and offers options to users in order to achieve that. The basic idea is to give users the option to post an image or video, make comments, add location and like any comment or post as well as deleting or editing either a post or comment.

To achieve these functionalities and make them work together, we chose to add a bottom navigation bar in the home screen in order to navigate into different screens. Apart from that, users can navigate to screens by pressing different buttons. In the next section we will describe the way each component developed.

4.2.3.1 Users Posts

To begin with, users can see posts in the feed page only if he follows a different user and that user has posted anything, or his own posts. The different posts that appear to a user are called in our application using a cloud function. The reason we chose to use a cloud function was because it's a heavy operation to happen by the application itself and users would have to wait some time for their posts to appear.

The idea of the function is to check the user's document in Firestore and specifically the field ''following'' which is a list that contains all user Id's of people the user follows. After receiving this list, an operation that iterates through the collection userPosts in Firestore takes place producing a different list with every detail of each post and is sent to our application.

The call for this function happens in dart and Is shown below:

```
var url =
   'https://<<Function Url>>/getFeed?uid=' + userId;
var httpClient = HttpClient();

var request = await httpClient.getUrl(Uri.parse(url));
var response = await request.close();

if (response.statusCode == HttpStatus.ok) {
 String json = await response.transform(utf8.decoder).join();
 // prefs.setString('${utils.auth.currentUser.uid}_feed', json);
 List<Map<String, dynamic>> data =
```

```
 jsonDecode(json).cast<Map<String, dynamic>>();
 listOfPosts = _generateFeed(data);
 result = "Success in http request for feed";
}
```

We provide the url of the function combined with the user Id. After that a call to the function happens. If everything went as planned the we get the response. The response of the function contains all the information we need to show the posts to the user screen.

The cloud function, as described in a previous chapter, is developed in TypeScript and saved in our Firebase project. A code sample of the function is shown below:

```
export const getFeed = functions.https.onRequest((req, res) => {
   getFeedModule(req, res);
});
export const getFeedModule = function (req: any, res: any) {
 const uid = String(req.query.uid);
 const latestDocument = new Date(Number(req.query.latestDocument));
 console.log(latestDocument);
 async function compileFeedPost() {
  let listOfPosts = await getAllPosts(uid, latestDocument, res,);
  listOfPosts = [].concat.apply([], listOfPosts); // flattens list
  res.send(listOfPosts);
 }
 compileFeedPost().then().catch();
}
function getAllPosts(userId: any, latestDocument: any, res: any) {

Const posts=admin.firestore().collection("user_posts")
.where("ownerFollowers","array-contains", userId).orderBy("timestamp", 'desc').startAfter(latestDocument || 0).limit(20);
 return posts.get()
  .then(function (querySnapshot) {
   const listOfPosts: any = [];
   querySnapshot.forEach(function (doc) {
    listOfPosts.push(doc.data());
   });
   return listOfPosts;
  })
```

```
}
```

In case a user is following many users, the feed page would take too long to fully load. For that reason pagination has been used in the cloud function as shown in the code sample, with the addition of the limit(20). What this means is that we load the first twenty posts, based on timestamp, until the user scrolls at the end of the page. Then we load the next twenty posts and so on. To achieve that iteration in Firestore, the last known document of the batch of twenty must be saved, so the next time we take the other twenty posts, firebase will know from which document will continue the iteration. The way this logic is implemented in the application is by saving to a variable the last known document, so the moment the user scrolls to the end of page, the next batch of twenty posts will appear. In case a user reloads the feed page, this variable is set to zero so this process starts again from the beginning.

After the cloud function completes, all the different posts are presented to the users interface. Users can like each post, comment, report or even edit or delete only in case they own the post.

The like functionality happens with a double tap on the image or just a single tap on the spoon icon which is what we selected for the application. After liking the post, a spoon animation appears in the middle of the photo. This animation was built in Rive, which is an online tool for building animations. This tool, after the completion of the animation, produces a .rive file which we read in Flutter with the help of the Rive library.

By tapping in the location of the posts users can open Google Maps and see the location there. If a user delete a post, the collections on comments, posts and reports are different in firestore which means an iteration to all these collections is happening in order to delete everything that is related to this post. This operation is presented below:

```
await FirebaseFirestore.instance
   .collection("user_posts")
   .doc(widget.postId).delete();
Navigator.pop(context);

//remove comments contained in this post, from fireStore
await FirebaseFirestore.instance
   .collection("cookit_comments")
   .doc(widget.postId).collection("comments").get().then((value) {
  value.docs.forEach((element) {
   //remove comment_likes from commentOwner's activityFeed
   var commentLikes = element.data()["likes"];

   for (var i=0; i<commentLikes.length;i++){
     FirebaseFirestore.instance
        .collection("cookit_a_feed").doc(element.data()["userId"])
        .collection("items").where("commentId",isEqualTo:element.data()["commentId"]).where("userId",
isEqualTo:commentLikes[i])
        .where("type", isEqualTo:"comment_like").get().then((snapshot){
      snapshot.docs.first.reference.delete();
     });
   }
   //remove comment from fireStore
   FirebaseFirestore.instance.collection("cookit_comments").doc(widget.postId).
   collection("comments")
      .doc(element.id)
      .delete()
```

```
      .then((value) => print("success"));

  });
});
//remove comments and post_likes from postOwner's activityFeed
FirebaseFirestore.instance
   .collection("cookit_a_feed").doc(widget.ownerId)
   .collection("items").where("postId", isEqualTo: widget.postId)
   .get().then((snapshot)async{
 for(DocumentSnapshot ds in snapshot.docs)  {
   await ds.reference.delete();
 }
 // snapshot.docs.first.reference.delete();
});
//remove this' post image/video from firebaseStorage
await FirebaseStorage.instance.refFromURL(widget.mediaUrl).delete().then((_) => print('Successfully deleted storage
item' ));
//remove this' post videoThumbnail from firebaseStorage if it's a video
if(widget.thumbNailUrl!=null && widget.thumbNailUrl.isNotEmpty){
await FirebaseStorage.instance.refFromURL(widget.thumbNailUrl).delete().then((_) => print('Successfully deleted
storage item' ));
}
```

Every user can enter another's profile by tapping in their display name anywhere in the application. Each profile contains a grid view of user's posts as well as their followers and a list of who they follow. This way we re providing an easy way for a user to search for more users from their friends and contribute to the large network we hope to exist in the application.

4.2.3.2 Users Comments

The addition of comments was important because we believe is a more direct way for our users to communicate and socialize more. Each comment page belongs to a single post. It contains every comment that has been made for the particular post. The comments are paginated as well for the same reason we paginated the cloud function for posts, speed. In case the comments were too many, users would have to wait a long time for the page to load, so a pagination of twenty comments was selected. Users are able to add emojis and icons in their comment, pictures taken from the camera or the gallery of their device, or even gifs.

Gif's weren't in-built in Flutter, so the library giphy_picker had to be used. Furthermore, an account in giphy.com for developers had to be made in order to own a giphy Api code and have access to the vast majority of the gifs in the site.

Users can either report or delete a comment. Different checks are made when a user wants to do an operation in comments. These checks refer to whether the user owns the post or the particular comment they have selected. If they re not the owner of the comment this means that they will be provided with the option only to report the comment. In case they own the comment they will have the option to delete the comment. In case they own the post, they will be able to delete any comment whether its their own or not. This process is happening since we always know the user id of each comment, meaning the person who made the comment, as well as the user id that uses the application and is logged in. With a few words, if a user is signed in the application, the information of the user's id is simply taken with *FirebaseAuth.instance.currentUser.uid* command in dart.

4.2.3.3 Creating Posts

Users can create a post by simply pressing the middle button in bottom navigation bar. By doing that, a modal sheet appears and gives the option to upload a video or image either from gallery or camera.

The creation of the post is a new page that contains different fields such as the description of the post, the location and the related recipe. Since the application is a food based application we added this feature so in case the picture a user wants to upload is related to a recipe from our app he can simply link the recipe to the post. To do so, during making the post, users have access to every recipe in our application that he can then tag to the post.

Users can also add location as mentioned. A field which they can search the preferred location is provided. To get the different locations. In addition, before even search for a location, under the text field there will appear suggested nearby places based on the user's device location. The code example is shown below:

```
LocationData currentLocation;
String error;
Location location = Location();
bool _serviceEnabled;


_serviceEnabled = await location.serviceEnabled();
if (!_serviceEnabled) {
 _serviceEnabled = await location.requestService();
 if (!_serviceEnabled) {//not necessary cause we use try catch in initplatformstate if currentlocation=null...
  return;
 }
}

currentLocation = await location.getLocation()

final coordinates = Coordinates(
   currentLocation.latitude, currentLocation.longitude);
var addresses =
await Geocoder.local.findAddressesFromCoordinates(coordinates);
var first = addresses.first;
```

In the above sample code, is shown that we first show a dialog in case location services aren't enabled and wait for user to respond. After the user accepts to give access to the application for the device's location, we get the details of the user's location such as longitude and latitude. After this operation, we search for nearby places to suggest to the user as shown in the code below:

```
var dio = Dio();
var url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json";
var parameters = {
 "key":utils.kGoogleApiKey,
 "location":"${currentLocation["latitude"]},${currentLocation["longitude"]}",
 "radius":"500",
};
var response = await dio.get(url,queryParameters: parameters);
```

```
return response.data["results"]
 .map<String>((result)=> result["name"].toString())
 .toList();
```

The list that the previous function returns, contains every name of the nearby places and the first five places are presented to the user as suggestions under the location field. After the completion of the form the user can finally upload the post. At this point, every information of the post is uploaded to Firestore and the image or video is uploaded to Storage. The upload to Firestore operation is shown in the code sample below:

```
var reference = await FirebaseFirestore.instance.collection('user_posts').doc().get();
String docId;

await reference.reference.set({
  "username": utils.name,
  "location": location,
  "likes": [],
  "mediaUrl": mediaUrl,
  "description": description,
  "ownerId": utils.auth.currentUser.uid,
  "timestamp": DateTime.now(),
  "referredRecipeInfo":referredRecipeInfo,
  "postId":reference.reference.id,
  "commentsAllowed":true,
  "mediaType":mediaType,
});
```

In case of a video, we upload an image of a fraction of the video like ''thumbnail'' that helps us in other parts of the application, as for example the image we show to a user before the video fully loads and starts playing.

4.2.3.4 Search for Users

The ability to search for other users in our application was crucial because it provides an easy way for the users to find other users using the application. The way we accomplished the search functionality was through Algolia. Algolia is a search engine and has been described in a previous chapter. The main reason Algolia was selected for the search functionality was the factor of how fast it will return the results. The first attempt was through Firestore with queries each time a user writes a different letter in the search field. This was a heavy operation for Firebase and it took a long time to return the results. Algolia works differently than Firestore because it focuses in searches. To achieve the search with Algolia functionality we first created cloud functions that executes whenever a user is created in the application, updated or deleted. The reason behind this, is that the collection of Users must be created in Algolia in order to implement the search in this collection and send the appropriate data to the application.

A code sample of the cloud functions in TypeScript when a user is created, updated or deleted is shown below:

```
const ALGOLIA_APP_ID = "xxx";

const ALGOLIA_ADMIN_KEY = "xxx";

const ALGOLIA_USERS_INDEX_NAME = "Users";


exports.createUser = functions.firestore.document('users/{userId}').onCreate(async (snap, context) => {
```

```
    const newValue = snap.data();
    newValue.objectID = snap.id;
    var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);
    var index = client.initIndex(ALGOLIA_USERS_INDEX_NAME);
    index.saveObject(newValue);
    console.log("finished");
});

exports.updateUser = functions.firestore.document('users/{userId}').onUpdate(async (snap, context) => {
    const afterUdpate = snap.after.data();
    afterUdpate.objectID = snap.after.id;
    var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);
    var index = client.initIndex(ALGOLIA_USERS_INDEX_NAME);
    index.saveObject(afterUdpate);
});

exports.deleteUser = functions.firestore.document('users/{userId}').onDelete(async (snap, context) => {
    const oldId = snap.id;
    var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);
    var index = client.initIndex(ALGOLIA_USERS_INDEX_NAME);
    index.deleteObject(oldId);
});
```

The functions are triggered whenever one of the above events happen.

4.2.4 Recipes Component

In this section, the way all recipes were retrieved and how these assimilated in the application will be described. Recipes are a part of the home page since they can be reached through the search button in bottom navigation bar of the home page.

4.2.4.1 Scrape the Recipes

The first step was to scrape all these recipes from three different web sites. These sites are *www.akispetretzikis.com*, *www.recipetineats.com* and *www.jamieoliver.com*. To begin with, web sraping was developed with Python programming language. The main proccess was to first identify all the elements needed in order to get the recipe's title, photo url, ingredients, method and cooking time. One example of the code is shown below and describes the proccess followed to get the list of ingredients.

```
cookingIngredients= []
ingredients = page_soup4.find('div', {'class': "wprm-recipe-ingredients-container"})

if ingredients:
 for ultag in ingredients.find_all('ul'):
  for litag in ultag.find_all("li","wprm-recipe-ingredient"):
    cookingIngredients = []
    for span in litag.find_all("span"):
                cookingIngredients.append((" ".join(span.text.replace("□", "").split())))
    cookingIngredients2.append(' '.join(cookingIngredients).lstrip())


else:
  stringIngredients = "Cooking ingredients undefined"
```

At first an empty list called cookingIngredients is claimed. Then with the help of the Python library BeautifulSoup4 we find all the elements in the page, that we declared earlier, where the element class is equal to *wprm-recipe-ingredients-container.* After that we iterate through these elements to get the information we want which is the list of ingredients inside a recipe. The final step is to take the list cookingIngredients and upload it in Firestore. The code below shows exactly that process.

```
for element in cookingIngredients:

  doc_ref = db.collection('cookeat_recipes').document(recipeUrl[counter].replace("/", ""))

  doc_ref.set({

    'ingredients': element,

  })
```

This way the ingredients are saved in the document inside cookeat_recipes collection in Firestore. During the whole process every field that was needed as information was taken and as described in the above code, all fields were saved at the same time. The process continued for all three web sites with some modification in code, mostly in html elements that had to be used, but the main idea remained the same.

4.2.4.2 Cloud Functions for Algolia

The main goal was that users should have access to all these recipes by developing an easy way to search for each one. Another goal was to give the ability to search for a recipe based on the ingredients the user selects.

Cloud functions were developed in order to save every recipe from Firestore to Algolia as well, during the web scraping phase. This way the implementation of search by ingredients can happen with Algolia. The code referring in cloud functions, and developed in Typescript, is provided below:

```
const ALGOLIA_RECIPE_INDEX_NAME = "Recipes";

exports.createRecipe = functions.firestore.document('cookeat_recipes/{recipeId}').onCreate(async (snap, context) => {

  const newValue = snap.data();

  newValue.objectID = snap.id;

  var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);

  var index = client.initIndex(ALGOLIA_RECIPE_INDEX_NAME);
```

```
    index.saveObject(newValue);
    console.log("finished");
});

exports.updateRecipe = functions.firestore.document('cookeat_recipes/{recipeId}').onUpdate(async (snap, context) => {
    const afterUdpate = snap.after.data();
    afterUdpate.objectID = snap.after.id;
    var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);
    var index = client.initIndex(ALGOLIA_RECIPE_INDEX_NAME);
    index.saveObject(afterUdpate);
});

exports.deleteRecipe = functions.firestore.document('cookeat_recipes/{recipeId}').onDelete(async (snap, context) => {
    const oldId = snap.id;
    var client = algoliasearch(ALGOLIA_APP_ID, ALGOLIA_ADMIN_KEY);
    var index = client.initIndex(ALGOLIA_RECIPE_INDEX_NAME);
    index.deleteObject(oldId);
});
```

As shown, there are three different functions. The first function createRecipe is triggered whenever a recipe is stored in Firestore and send this information in Algolia. The next function updateRecipe triggers whenever an update happens in a recipe so Algolia's recipes are up to date as well. This function triggers for example whenever a user rates a recipe. The field ''rating'' in the document is Firestore updates then and so is Algolia's field for this recipe. The last function deleteRecipe triggers when a recipe is deleted.

4.2.5 Notifications

There are two ways for a user to be notified in app whenever an event happens such us another user commented to his post, liked it, followed or liked a comment. If the application is used by the user, which means it runs in the foreground, after an event happens user will be able to check any notification inside the app from the bell button in the home screen. Another way a user can be notified is by system notifications.

4.2.5.1 In-app Notifications

Every notification will be displayed, from the most recent to the oldest. In order to make a user friendly page for notifications, pagination was implemented in case there were many notifications for a user. There are two main methods that operate. The first method is called getActivityFeed and its purpose is to get user's notifications when user opens the activity page. Pagination of fifteen results was implemented. The code below describes this function:

```
getActivityFeed(var latestDocument, bool loadRefresh ) async {
```

```
List<ActivityFeedItem> items = [];
var snap = await FirebaseFirestore.instance
    .collection('cookit_a_feed')
    .doc(utils.auth.currentUser.uid)
    .collection("items")
    .orderBy("timestamp",descending: true).startAfter([latestDocument]).limit(15)
    .get();
 for (var doc1 in snap.docs) {
  var userInfo = await utils.users.doc(doc1.data()["userId"]).get();
  List followers = userInfo["followers"];
  bool isFollowing;
followers.contains(utils.auth.currentUser.uid)?isFollowing=true:isFollowing=false;
items.add(ActivityFeedItem(userId:doc1.data()["userId"],username:userInfo["displayName"],type:doc1.data()["type"],
    mediaUrl: doc1.data()["mediaUrl"]??null,
     commentData: doc1.data()["commentData"]??null, timestamp:DateTime.parse(
doc1.data()["timestamp"].toDate().toString())??null,postId:doc1.data()["postId"]??null,commentId:doc1.data()["comme
ntId"]??null,followedOwnerId:doc1.data()["ownerId"]??null,userProfileImage:userInfo["photoUrl"]??null,isFollowing:is
Following,activityId:doc1.id,thumbNailUrl:doc1.data()["thumbNailUrl"]??null
    )
    );
  }
  if(mounted) {
   if(loadRefresh){
    activityItems=[];
   }
   try {
    if(mounted)
    setState(() {
     hasMorePressed = false;
     activityItems.addAll(items);
     if(items.length<15){
      hasMore=false;
     }else{
      hasMore=true;
      latestDoc = snap.docs.last["timestamp"].toDate();
     }
```

```
      });
    }catch (e){
    }
  return activityItems;
 }
```

This function is called immediately when user opens the activity page, to get the first fifteen results, but is also called when user scroll at the end of page to get the next fifteen and so on. By storing the value of the last document of each batch in a variable, we are able to request the next batch of fifteen, by using the startAfter(lastDocument) function when requesting the data from Firestore. With every request, all the necessary information is taken, such as the type of the notification, the user id of the owner, the photo url and more, in order to show to users the appropriate notifications in each case.

The second important function is called updateToNewestFeedInFirestore and its purpose is to update a field in Firestore with the notification id. Whenever a notification is received to a user a stream builder detects that and this function is triggered. A badge appears in the notification icon, to notify the user that a new activity has appeared. The field that is updated in Firestore and is called ‘’activity’’ is used in order to detect whether a user has opened the activity page, so the badge has to be removed, or not. The code sample of this function is shown below:

```
updateToNewestFeedInFireStore() async{
  if(utils.newNotification){
    var activity = [];
Var result=await
FirebaseFirestore.instance.collection("cookit_a_feed").doc(utils.auth.currentUser.uid).collection("items").
orderBy("timestamp",descending: true).get();
    for (var i in result.docs){
      activity.add(i.id);
    }
    utils.users.doc(utils.auth.currentUser.uid).update(
       {"activity":activity});
      setState(() {
        utils.badgeStreamController.add(false);
      });
  }
 }
```

4.2.5.2 System Notifications

System notifications only appear in users device if an event happens and the application is running in the background or is terminated.

The implementation of these kind of notifications, happened with cloud functions. The basic idea was to create a document with the information of the notification whenever a user makes an action. This creation of the notification document in Firestore, under the collection cookeat_a_feed, that contains the notification documents for each user, triggers a function that sends the notification in user's device along with the data needed. Specifically, the code developed in Typescript for the cloud functions is shown below:

```
export const notificationHandler = functions.firestore.
  document("/cookit_a_feed/{userId}/items/{activityFeedItem}")
  .onCreate(async (snapshot, context) => {
    await notificationHandlerModule(snapshot, context);
  });

export const notificationHandlerModule = async function (snapshot: DocumentSnapshot, context: any) {
  console.log(snapshot.data())
  const ownerDoc = admin.firestore().collection("users").doc(context.params.userId)//.doc("users/" + context.params.userId)
  const ownerData: any = await ownerDoc.get()
  activityId = context.params.activityFeedItem
  const androidNotificationToken = ownerData.data()["androidNotificationToken"];
  if (androidNotificationToken) {
    sendNotification(androidNotificationToken, snapshot.data()!);
  } else {
    console.log("No token for User, not sending a notification");
  }
 return 0;
};
function sendNotification(androidNotificationToken: string, activityItem: FirebaseFirestore.DocumentData) {

  let title: string = "";
  let body: string = "";
  let userId: string = "";
  let userProfileImage: string = "";
  let mediaUrl: string = "";
  let type: string = "";
  let commentId: string = "";
  let postId: string = "";
  let photoInCommentUrl: string = "";
```

```
let gifUrl: string = "";
let userName: string = "";
let bio: string = "";

if (activityItem['type'] === "comment") {
    title = activityItem['username'] + " commented on your post"
    body = activityItem["commentData"]
    userId = activityItem["userId"]
    if (activityItem["photoInCommentUrl"] !== null) {
        photoInCommentUrl = activityItem["photoInCommentUrl"]
    } else if (activityItem["gifUrl"] !== null) {
        gifUrl = activityItem["gifUrl"]
    }
    userProfileImage = activityItem['userProfileImage']
    if (activityItem["thumbNailUrl"] === null) {
        mediaUrl = activityItem['mediaUrl']
    } else {
        mediaUrl = activityItem['thumbNailUrl']
    }


    commentId = activityItem['commentId']
    postId = activityItem['postId']


} else if (activityItem["type"] === "like") {
    title = activityItem['username'] + " liked your post"
    userId = activityItem["userId"]
    userProfileImage = activityItem['userProfileImage']
    if (activityItem["thumbNailUrl"] === null) {
        mediaUrl = activityItem['mediaUrl']
    } else {
        mediaUrl = activityItem['thumbNailUrl']
    }
    postId = activityItem['postId']
```

```
} else if (activityItem["type"] === "comment_like") {
    title = activityItem["username"] + " liked your comment"
    body = activityItem["commentData"]
    userId = activityItem["userId"]
    userProfileImage = activityItem['userProfileImage']
    commentId = activityItem['commentId']
    postId = activityItem['postId']
    if (activityItem["photoInCommentUrl"] !== null) {
        photoInCommentUrl = activityItem["photoInCommentUrl"]
    } else if (activityItem["gifUrl"] !== null) {
        gifUrl = activityItem["gifUrl"]
    }
}
else if (activityItem["type"] === "follow") {
    title = activityItem['username'] + " started following you"
    userId = activityItem["userId"]
    userProfileImage = activityItem['userProfileImage']
    userName = activityItem['username']
    bio = activityItem['bio']
}
type = activityItem["type"]
const message = {
    data: {
        userName: userName,
        bio: bio,
        title: title,
        body: body,
        userId: userId,
        userProfileImage: userProfileImage,
        mediaUrl: mediaUrl,
        type: type,
        commentId: commentId,
        postId: postId,
```

```
        activityId: activityId,
        photoInCommentUrl: photoInCommentUrl,
        gifUrl: gifUrl,
      },
      token: androidNotificationToken
    }


  admin.messaging().send(message)
      .then((response) => {
        // Response is a message ID string.
        console.log('Successfully sent message:', response);
      })
      .catch((error) => {
        console.log('Error sending message:', error);
      });
}
```

As shown in the above code, every information of the notification is stored in the variable ‘’message’’. This variable and its content is sent in the application in order to show the notification to the user. In order to listen whenever a notification is sent to the user in Flutter we use the code provided below:

FirebaseMessaging.onBackgroundMessage(firebaseMessagingBackgroundHandler);

```
Future<void> firebaseMessagingBackgroundHandler(RemoteMessage message) async {

 if(message.data["type"]=="like"){
  AwesomeNotifications().createNotification(
    content: NotificationContent(
     payload: {"type":message.data["type"], "postId":message.data["postId"],"activityId":message.data["activityId"]},
     hideLargeIconOnExpand: false,
     color: Colors.lightBlue,
     notificationLayout: NotificationLayout.BigPicture,
     id: 10,
     channelKey: 'basic_channel',
     title: 'CookIt',
     body: '${message.notification.title}.',
     largeIcon:message.data["userProfileImage"],
     bigPicture: message.data["mediaUrl"],
    )
  );
 }else if (message.data["type"]=="comment"){
  await AwesomeNotifications().createNotification(
    content: NotificationContent(
     payload: {"type":message.data["type"], "commentId":message.data["commentId"],
```

```
        "postId":message.data["postId"],

        "activityId":message.data["activityId"]},
      hideLargeIconOnExpand: false,

      color: Colors.lightBlue,
      notificationLayout: NotificationLayout.BigText,
      id: 10,
      channelKey: 'basic_channel',
      title: 'CookIt',
      body: '${message.notification.title} : "${message.notification.body}"',
      largeIcon:message.data["mediaUrl"],

    )
  );
}else if (message.data["type"]=="comment_like"){
  print("its a comment_like");
  AwesomeNotifications().createNotification(

    content: NotificationContent(
      payload: {"type":message.data["type"], "commentId":message.data["commentId"],
        "postId":message.data["postId"], "activityId":message.data["activityId"],},
      hideLargeIconOnExpand: false,

      color: Colors.lightBlue,
      notificationLayout: NotificationLayout.BigText,
      id: 10,
      channelKey: 'basic_channel',
      title: 'CookIt',
      body: '${message.notification.title} : "${message.notification.body}"',
      largeIcon:message.data["userProfileImage"],

    )
  );
}else if (message.data["type"]=="follow"){
  print("its a follow");
  AwesomeNotifications().createNotification(

    content: NotificationContent(
      payload: {"type":message.data["type"],"activityId":message.data["activityId"]},

      color: Colors.lightBlue,
      notificationLayout: NotificationLayout.Default,
      id: 10,
      channelKey: 'basic_channel',
      title: 'CookIt',
      body: '${message.notification.title}.',
      largeIcon:message.data["userProfileImage"],

    )
  );
```

```
 }
 else if(message.data["type"]==null){

   AwesomeNotifications().createNotification(
      actionButtons:[NotificationActionButton(key:"Dismiss",autoCancel:
true,buttonType:ActionButtonType.KeepOnTop,enabled: true,label: "Dismiss" )],
      content: NotificationContent(
        payload: {"type":message.data["type"],},
        hideLargeIconOnExpand: false,

        color: Colors.lightBlue,
        id: 10,
        channelKey: 'basic_channel',
        title: message.notification.title,
        body: message.notification.body,

      )
   );
 }
```

The package used in flutter is Awesome Notifications. The function *FirebaseMessaging.onBackgroundMessage triggers* whenever a notification is received. Based on the event, a notification is produced and sent to the user's device, with different appearance in each case. The different attributes such as color, body, title e.t.c are referring to the appearance of the notification.

5. Evaluation & Testing

This chapter presents the evaluation and testing of the project. The first two sections present the testing of the functional requirements using the Android framework and the non-functional requirements evaluation respectively. The third section presents the application's UI testing using the Android tools, followed by the application's evaluation. The last section is a summary of all the results of the testing that has taken place.

5.1 Functional Requirements Evaluation

In order to test if the application meets the defined functional requirements in Section 3.1.1, the Android Studio emulator was used as well as some different android mobile devices. We tested all aspects of the application during and after the implementation.

Below the test cases are presented to evaluate the correct functionality of the application.

- Checks that the application allows users to sign up.
- Checks that the application allows users to sign in. At first an indicator is displayed to the user and the user is redirected to the main screen.
- Checks that the application allows users to change the password or resend a verification email after a user sign up.
- Checks that the application allows users to change their name and password after they are logged in.

- Checks that the application is presenting the correct information inside a recipe.
- Checks that the application is uploading data when needed to Firestore.
- Checks that the application is uploading pictures or videos in Firebase Storage.
- Checks that the application is reading the correct information of users posts.
- Checks that the application is reading the correct data from Shared Preferences, to present users history.
- Checks that the applications is showing the correct messages to users in all different error cases.
- Checks that the data shown to the users are the correct ones even if the user switch profiles from the same device.
- Checks that the application's data saved in the temporary directory of the device are deleted when users uninstall the application or clear cache.
- Checks that the refreshes in every screen of the application is indeed refreshing the data needed.
- Checks that the application is displaying the correct system notifications to users, according to the app's life cycle state (background, foreground, terminated), in all different cases.
- Checks that in app notifications are displayed to users when the application is in foreground and not displaying system notifications in that case.
- Checks at video trimming and image processing when users upload a photo or video.

All the functional requirements were met during the implementation and thus all test cases passed. Tests were made during the development of the application as well as when the application was finished.

5.2 Non Functional Requirements Evaluation

In this section, it is checked whether the non-functional requirements of the application which were defined in Section 3.1.2 were successfully fulfilled.

- The application offers the same user interface in all screens using the same style for menus, buttons and layouts.
- The application should show clear and detailed notification messages to the user. This was tested by navigating through all the application's screens and using all the functions which have been implemented. For every action a user performs, the system shows a detailed Toast message.
- The application must have a lack of bugs and must inform the user of any wrong operation. This was tested by doing all the possible mistakes in text fields that user has to fill information such as password or username. Furthermore, running the application and performing all possible actions in every screen, made it clear that the code is properly built.
- The application should be able to run on all Android devices. This was tested apart from the emulator of Android Studio, in many different devices of different brands.

- The application should request a password for each user account. Nobody can have access to the main screen of the application without passing through the sign in operation. If the user adds blank data to the password field, the system will display an error recognition message, as it was tested in the test cases in the previous section. The same goes for the sign up process which displays the same messages when the user leaves the password fields empty.

5.3 Testing the Application's UI

In addition to unit testing the individual components that make up the application (such as activities, services), it is also important to test the behavior of the application's user interface (UI) when it is running on a device. UI testing ensures that the application returns the correct UI output in response to a sequence of user actions on a device.

The approach for testing the User Interface was through the Android Studio emulator and android mobile devices during the application's implementation. Another approach, was to share the application's APK to a group of people and report every problem occurred during a period of time using the application.

Below UI test cases are presented that evaluate the correct UI output of the application. Each test case has been tested following both approaches.

- Checks that the application UI shows the keyboard input in all EditText objects (such as name, password).
- Checks that the application UI shows the ListView objects where needed(Posts, recipes, followers etc).
- Checks that the application UI shows the post photo or video dialog in all screens.
- Check that the application UI shows all buttons correctly in all screens.
- Checks that all screens have lack of crashing UI errors.
- Checks that the application UI shows the error toast messages correctly.
- Checks that the UI is showing the correct information in posts or comments.

All the User Interface test cases completed with almost no errors. Some errors were reported by the users using the application but were minor problems, such us toast message display position in the screen, but they were fixed immediately when reported.

5.4 Bad Internet connection

A problem with the login occurred when users were on an unstable internet connection. When the user is registering through sign in with google, Facebook or email, after they have completed the registration and they return to the app, the state does not update properly. When this problem happened, the users were stuck seeing a progress indicator. There is no such way to avoid this problem so the user must find a way to improve the internet connection and try to sign in again to the application. Unfortunately, these kind of problems are mostly concern the user and not the developer of an application. The implementation that is done, is the use of a package called connectivity which tracks whether a user is connected or not to an internet.

When the change of the connection state is happening, our application is displaying a message and informs the user if there is an internet connection or not.

Apart from the sign in process with a bad internet connection, there is an implementation of what should happen when no connection is available to almost every widget in code. That means that if there is no connection, the users Posts for example will display a message in the middle saying ''No internet Connection'' instead of the image or video, but the user in any case will not face a red error screen.

5.5 Database Reading Problems

During the application development we found out that the number of reads made from the database were unexpectedly high. After a period of time testing the application, we found that the source of this problem was the use of a widget in Flutter called FutureBuilder. What FutureBuilder does is that while building a page, for example the Comment Page, it waits for its future function to be completed meaning that it will wait for the function to read all the data from Firestore and display them after. The combined use of a FutureBuilder with a setState function(that refreshes the UI if its needed) causes the future function of the FutureBuilder to run again meaning more reads from Firestore.

To avoid these kind of problems, our first action was to initialize the future function inside the initState function of the class. Doing that decreased the number of reads made but did not entirely solve this problem unfortunately. This is presented in Figure 4.1 and Figure 4.2.

```
Future getActivity;
@override
void initState() {
  getActivity = getActivityFeed();
  super.initState();
}
```

Figure 5.1 Future in initState Function

Figure 5.2 FutureBuilder with future Function

The main reason this had to be fixed was that if a lot of users were using the application the number of reads from the database would be a very big number and that would make the cost go up dramatically.

The more drastic action that took place, was the implementation of pagination, meaning we didn't bring all the results when needed but we brought batches of results, for example by 20, when the users scrolls to the end of a list. This method implemented in many screens such as the Post screen, Comment screen, Recipes screen and User Profile screen.

In order to implement the pagination of lists in a ListView widget in Flutter, we used a library called Lazy Load Scrollview. The library allows the use of a LazyLoadScrollview widget that ‘’does’’ something when the users scrolls to the end of a list. Using variables that are incremented when user scrolls to the end of the list, we managed to accomplish what we wanted. In case of refreshing the list, the variables value is returning to its original desired value.

As shown in Figure 4.3, LazyLoadScrollView widget has and attribute called onEndOfPage. This attribute allows us to implement the code for what will happen when a user scroll to the end of the list. The two variables present and perPage are the variables that we increment to get the next batch of the results to the list.

Figure 5.3 Lazy Load ScrollView widget

5.6 Summary

The goal of this Chapter was to present all the tests and evaluations that took place during the implementation phase and afterwards to evaluate the correct functionality of the application. All test cases for the functional requirements were passed which means that the application has no functional error. The non-functional requirements were met too. In order to test the UI beside on our own by the use of Android Studio emulator, we asked a group of people to use the application for a period of time and report any problem they met during that period. Last but not least, in order to solve the increased number of reads from Firestore pagination was implemented to almost all the lists of information used in the application. There will always be some room for errors or bugs in the app but they will be resolved as long as they appear in the future.

6. Conclusion

This chapter summarizes the work done in this project, followed by the project's contribution and future work.

6.1 Work Summary

The goal of the project was to create a social media mobile application called Cook Eat. In addition, the use of Firebase and its services was another goal for the project. Solving common or more complex development

problems and trying to make a friendly and interesting user interface brought us closer to that goal. The application, as mentioned in chapter 5, has passed almost all the tests we have implemented as a result we hope to give an excellent experience to our users.

6.2 Project contribution

The contribution to the mobile developers' community is described in this section. The contribution is addressed below:

- Design principles and patterns of mobile interface design were analyzed. Instructions were given, which can be followed by new mobile developers for designing and developing good applications. This analysis may also help developers understand and consider a few important things when they want to design a new mobile application or transfer a full-sized computer application to a mobile environment.

- Tactics for solving common problems in Android application development to help new developers were provided.

- A fully functional Android application was created. The main advantages of creating this application are:

    - It is designed to make people interact with each other and have access to instructions of recipes they want to cook.

    - As it is an Android application, it can be installed for free in all Android devices (mobile phones and tablets), by being shared in the Google Store.

    - The source can be used by other developers to extend the current functions or add new ones that will enhance design of social media applications and tactics with Firebase.

    - Well design and strict login or sign up development can contribute to other developers future work in applications.

- Evaluation of the application was made. The evaluation was based on unit testing for the functional requirements and UI. We took into consideration the feedback of a group of people using the applications for a period of time.

6.3 Future work

This section discusses the possible improvements that can be made in the future to improve the application and add new functions.

→ **Improvement in Navigation.** The GUI of the application offers a minimal and easy navigation through the application's screens. As a result, the user needs to go to the main screen to navigate from the main to other screens. To avoid this, future developers can implement a navigation drawer. The navigation drawer is a panel that transitions in from the left edge of the screen and displays the application's main navigation options. It has the ability to access any top level content from anywhere in the application, no matter how deep the level a user is in.

→ **Improvement in Sign In method.** Currently, the check if the user is logged in or not is done through a StreamBuilder widget. There are some problems using a stream builder, especially when the internet connection of the user is bad. To avoid this, developers can implement many different ways such as SharedPrederences, Provider widget and more.

→ **Add charts.** As a feature, developers can add charts, giving information to the users such as average time users used the application the past week or how often the upload posts in the app and more.

→ **Implement data analysis.** Information about a users average age or gender or preferred recipes and categories of food could be an excellent work for future developers. The implementation can be done by developers by adding a form for users to fill, then extract the information and analyze it. A recommendation system of recipes can be added as well, by analyzing users preferences and most visited recipes or recipes they add to their favorite collection. For this to happen, developers must add extra collections to FireStore and save all the important information needed for such a system.

→ **Use more Cloud Functions.** Many transactions with Firestore can be done with Cloud Functions. Reading data from Firestore and displaying that data to users, even with the use of paginators, is sometimes a heavy process for the mobile device. As a future work, developers could use Cloud Functions to retrieve data for recipes,in-app notifications, personal posts, followers and more.

→ **Add Chat.** This feature is pretty common to most social media applications. Future developers could add messaging for more socialization between users. Chat rooms, for a group of people or one to one conversations, could be implemented giving a better experience to users.

→ **Identify inappropriate content**. The image or video of a post can have inappropriate content because there is no check that is happening now. With the help of Machine Learning, a model that identifies this kind of content can be built in order to check the content of each post or comment, making it impossible for a user to make an upload with inappropriate content in the application.

→ **More exciting features.** These kind of applications that are referring to socializing between users, have more room for many features and trends that a developer can add to the application. Instagram stories, for example, was something new and exciting for users. As an interesting feature, developers could add videos in recipes so users can watch the cooking process and make it easier for them to cook the recipe. Another feature could be giving the option to users to upload their own recipes in Firestore and other users could have access to these recipes, as well as rate the recipe or add it to their favorite collection.

6.4 Concluding remarks

Having mentioned both the theoretical and the practical aspects of mobile application development, as well as its primary uses and benefits, we can provide an overall account of our study. To begin with, in our project we analyzed the development process followed for Android applications, as well as the design principles, patterns and tactics which are useful for mobile application development. Having taken all these factors into consideration, we put them into practice and created a social media Android application called Cook Eat. To conclude, we believe that, with further improvement from future developers, the already useful application we created could become as well known as other applications that offers even more features to users.

References

1. Mayfield, Antony. "What is social media." (2008).

2. Bradley P. Be where the conversations are: The critical importance of social media. Business Information Review. 2010 Dec;27(4):248-52.

3. Vinerean S. Importance of strategic social media marketing.

4. Miller D, Sinanan J, Wang X, McDonald T, Haynes N, Costa E, Spyer J, Venkatraman S, Nicolescu R. How the world changed social media. UCL press; 2016.

5. Perrin A. Social media usage. Pew research center. 2015 Oct 8;125:52-68.

6. Ruths D, Pfeffer J. Social media for large studies of behavior. Science. 2014 Nov 28;346(6213):1063-4.

7. Wilson RE, Gosling SD, Graham LT. A review of Facebook research in the social sciences. Perspectives on psychological science. 2012 May;7(3):203-20.

8. Miller D. Tales from facebook. Polity; 2011 Apr 11.

9. Kwak H, Lee C, Park H, Moon S. What is Twitter, a social network or a news media?. InProceedings of the 19th international conference on World wide web 2010 Apr 26 (pp. 591-600).

10. Myers SA, Sharma A, Gupta P, Lin J. Information network or social network? The structure of the Twitter follow graph. InProceedings of the 23rd International Conference on World Wide Web 2014 Apr 7 (pp. 493-498).

11. Skeels MM, Grudin J. When social networks cross boundaries: a case study of workplace use of facebook and linkedin. InProceedings of the ACM 2009 international conference on Supporting group work 2009 May 10 (pp. 95-104).

12. Sumbaly R, Kreps J, Shah S. The big data ecosystem at linkedin. InProceedings of the 2013 acm sigmod international conference on management of data 2013 Jun 22 (pp. 1125-1134).

13. Miles J. Instagram power. McGraw-Hill Publishing; 2013.

14. Moreau E. What is Instagram, anyway. Here's what Instagram is all about and how people are using it [online]. 2018.

15. Hartmann G, Stead G, DeGani A. Cross-platform mobile development. Mobile Learning Environment, Cambridge. 2011 Mar;16(9):158-71.

16. Payne R. Developing in Flutter. InBeginning App Development with Flutter 2019 (pp. 9-27). Apress, Berkeley, CA.

17. Napoli ML. Beginning Flutter: A Hands on Guide to App Development. John Wiley & Sons; 2019 Oct 8.

18. Meiller D. Modern App Development with Dart and Flutter 2. InModern App Development with Dart and Flutter 2 2021 Jun 21. De Gruyter Oldenbourg.

19. Khawas C, Shah P. Application of firebase in android app development-a study. International Journal of Computer Applications. 2018 Jun;179(46):49-53.

20. Payne R. Using Firebase with Flutter. InBeginning App Development with Flutter 2019 (pp. 255-285). Apress, Berkeley, CA.

21. Moroney L. Using authentication in firebase. InThe Definitive Guide to Firebase 2017 (pp. 25-50). Apress, Berkeley, CA.

22. Moroney L. Firebase cloud messaging. InThe Definitive Guide to Firebase 2017 (pp. 163-188). Apress, Berkeley, CA.

23. Stonehem B. Google Android Firebase: Learning the Basics. First Rank Publishing; 2016 Jun 29.

24. Delia L, Pesado P. Performance Analysis in NoSQL Databases, Relational Databases and NoSQL Databases as a Service in the Cloud. InComputer Science–CACIC 2020: 26th Argentine Congress, CACIC 2020, San Justo, Buenos Aires, Argentina, October 5–9, 2020, Revised Selected Papers 2021 (p. 157). Springer Nature.

25. Moroney L. Cloud storage for firebase. InThe Definitive Guide to Firebase 2017 (pp. 73-92). Apress, Berkeley, CA.

26. Moroney L. Cloud functions for firebase. InThe Definitive Guide to Firebase 2017 (pp. 139-161). Apress, Berkeley, CA.

27. Mitchell, R. (2018). *Web scraping with Python: Collecting more data from the modern web*. " O'Reilly Media, Inc.".

28. Kumar A. Mastering Firebase for Android Development: Build real-time, scalable, and cloud-enabled Android apps with Firebase. Packt Publishing Ltd; 2018 Jun 29.

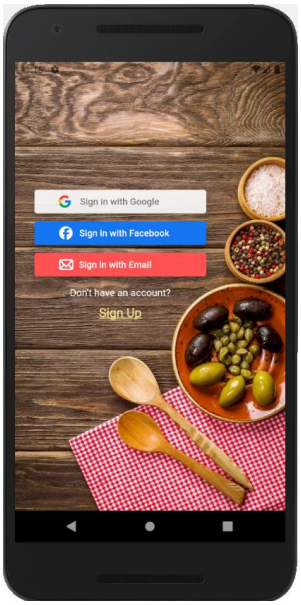
29. B. Ballard, Designing the Mobile User Experience, Wiley, 2007

30. Ian Sommerville, Software Engineering, 8th ed. Essex, England: Pearson Education Limited, 2007.

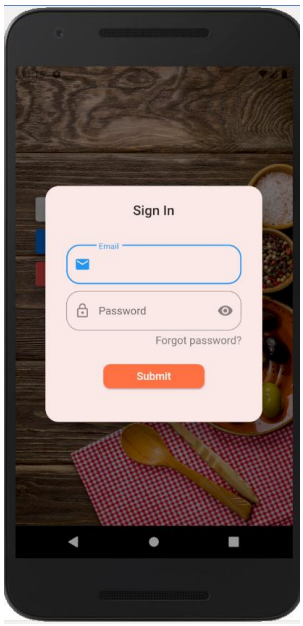
Annex - User Interface

This chapter will provide all different screens of the application and some key widgets that used in Flutter code.

Users are able to sign in with three different methods, Google, Facebook and Email/Password.



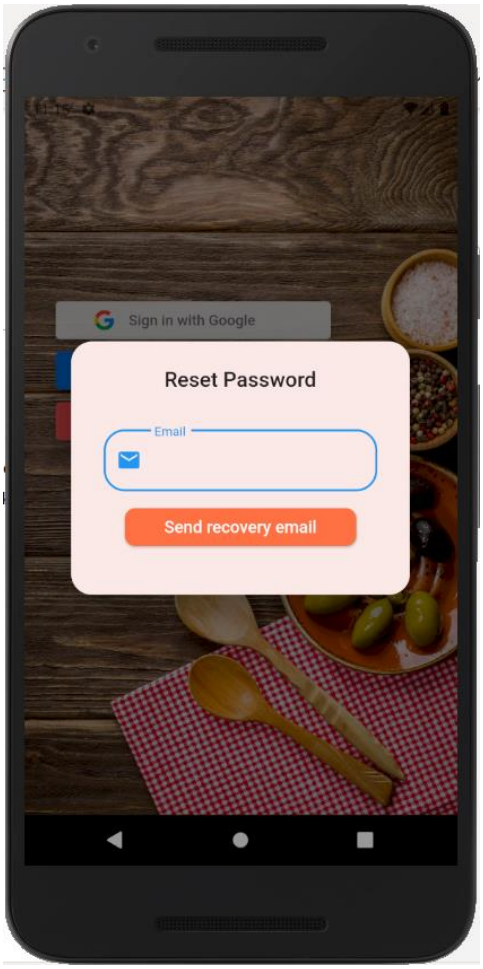
1. Login Screen



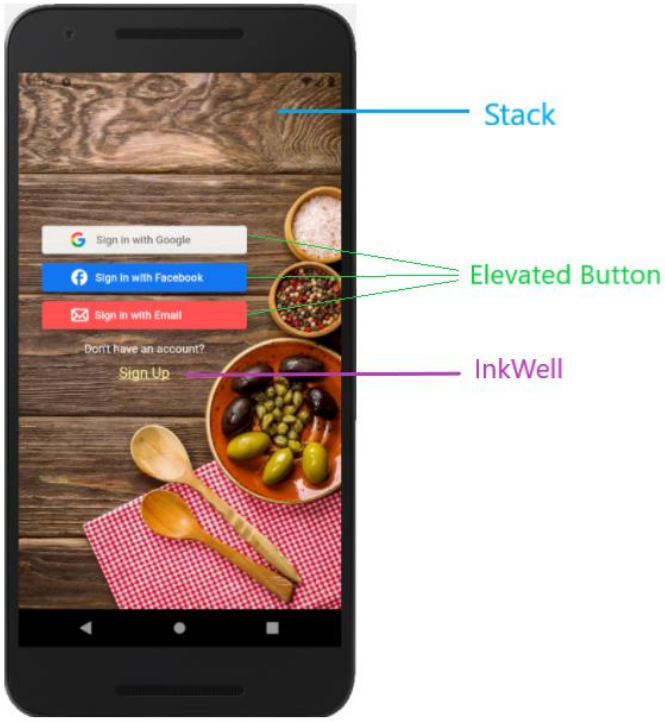
2. Sign In with Email

In Sign in with Email dialog, users are requested to fill their email and provide the password.

Users are able to reset their password in case they forgot it by entering their email address of the email they want to recover the password.

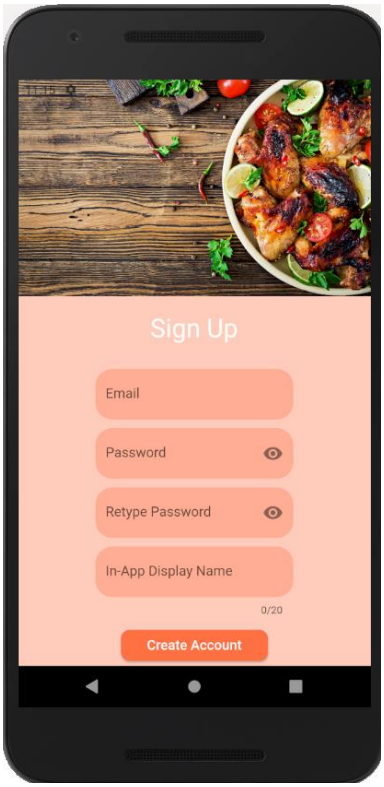


3.Reset Password

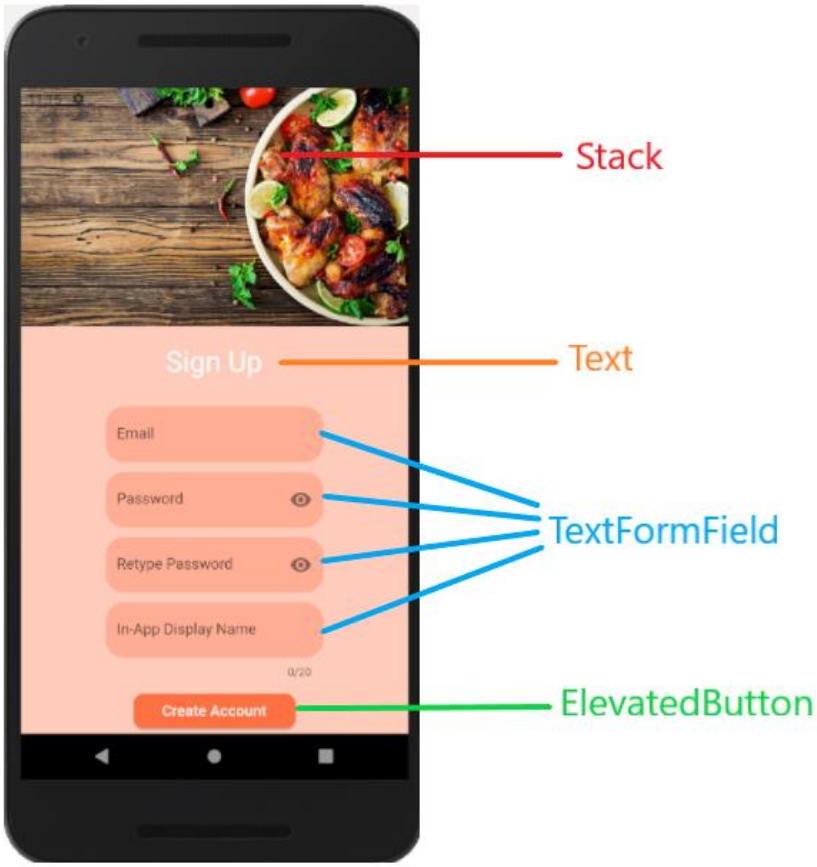


4. Login Screen Widgets

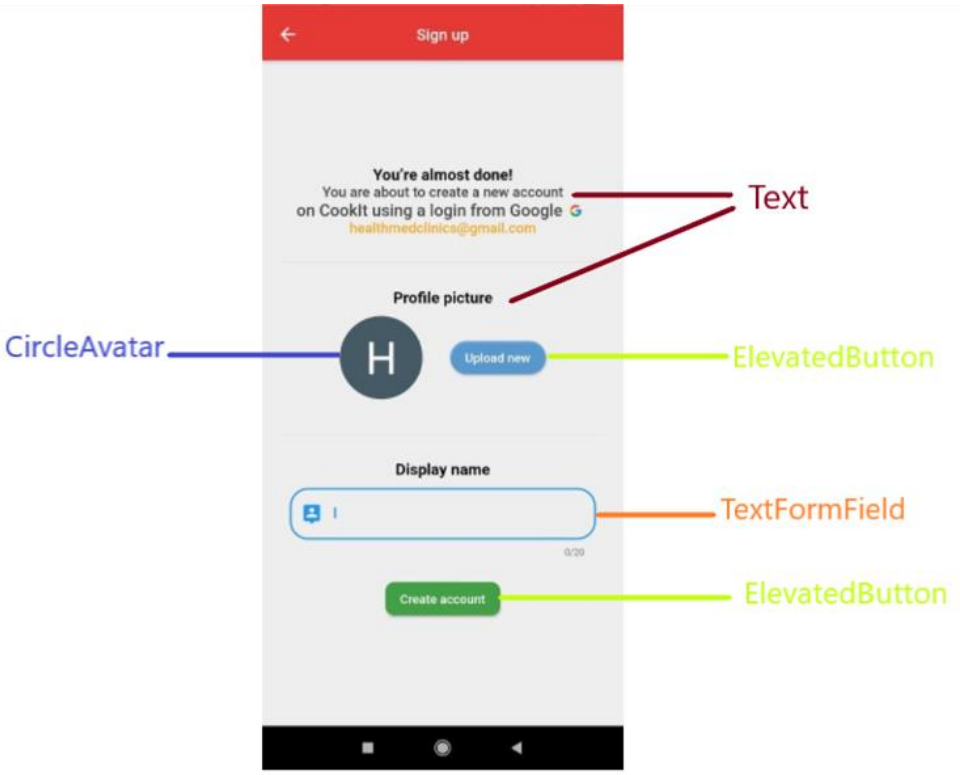
In sign up screens users are requested to fill a form and provide data such as their email, password and the display name of their choice.



5. Sign Up Screen

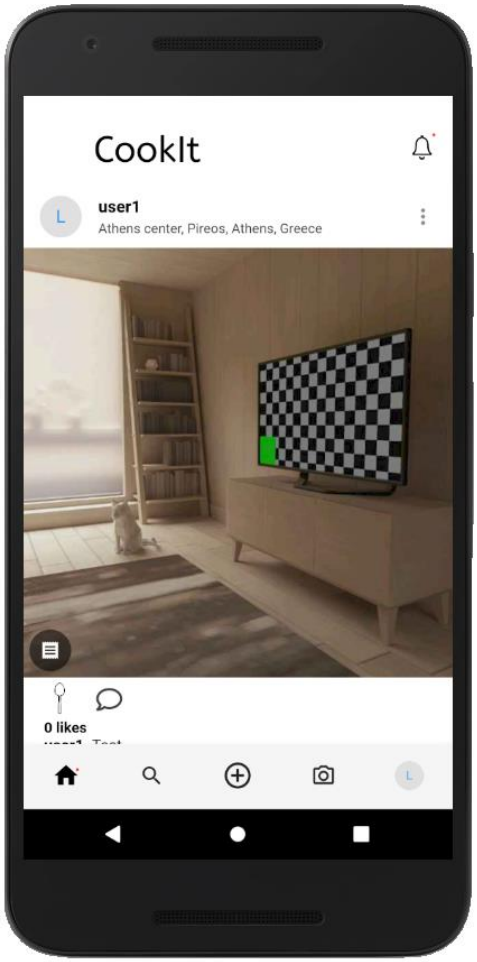


6. Sign Up Screen Widgets

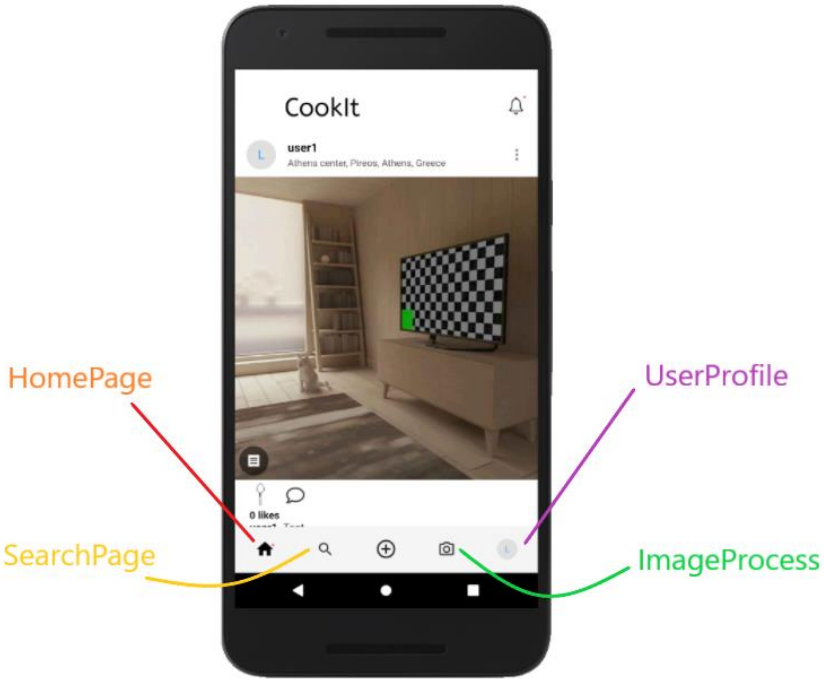


7. Google Sign Up Screen

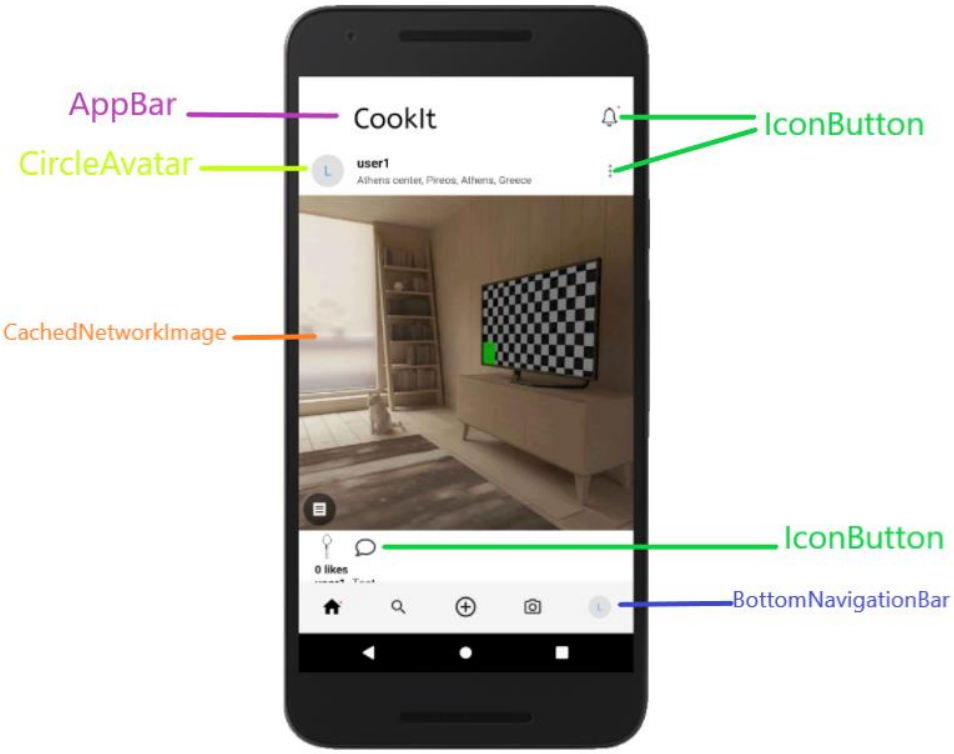
The home screen that contains the users posts, the bottom navigation bar and the bell icon button that leads to user's notifications.



8. Home Screen

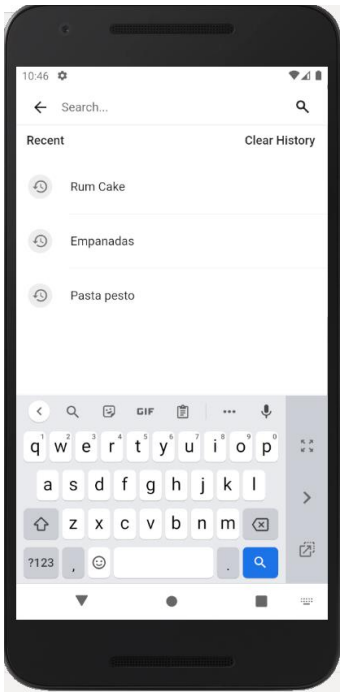


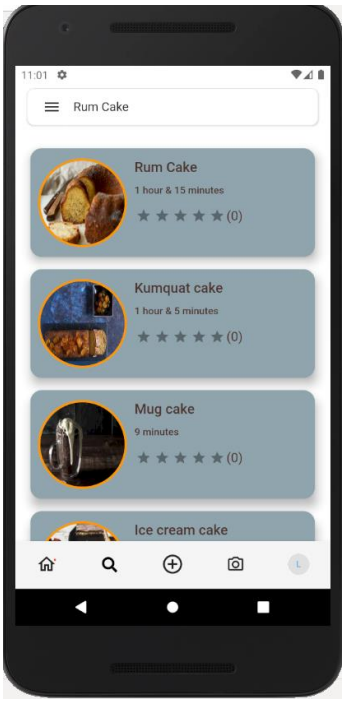
9. Home Screen Navigation Bar



10. Home Screen Widgets

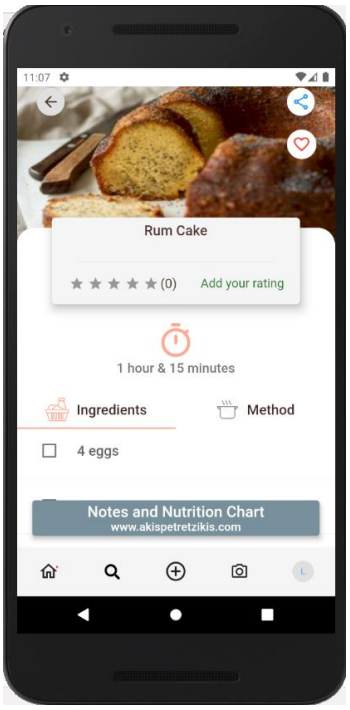
Users can search for different recipes based on what they write in the search field and get the result of all related recipes to that search.



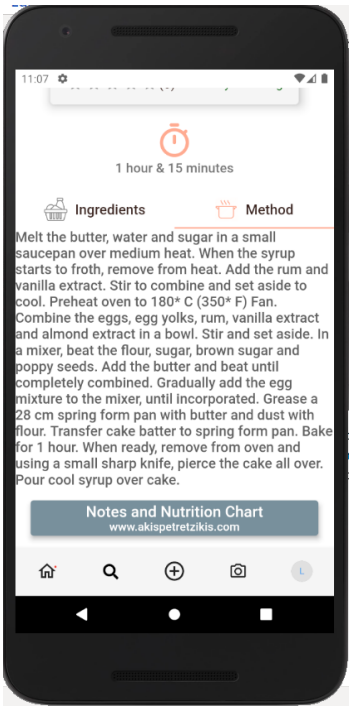


11. Search recipes Page **12. Recipe results Page**

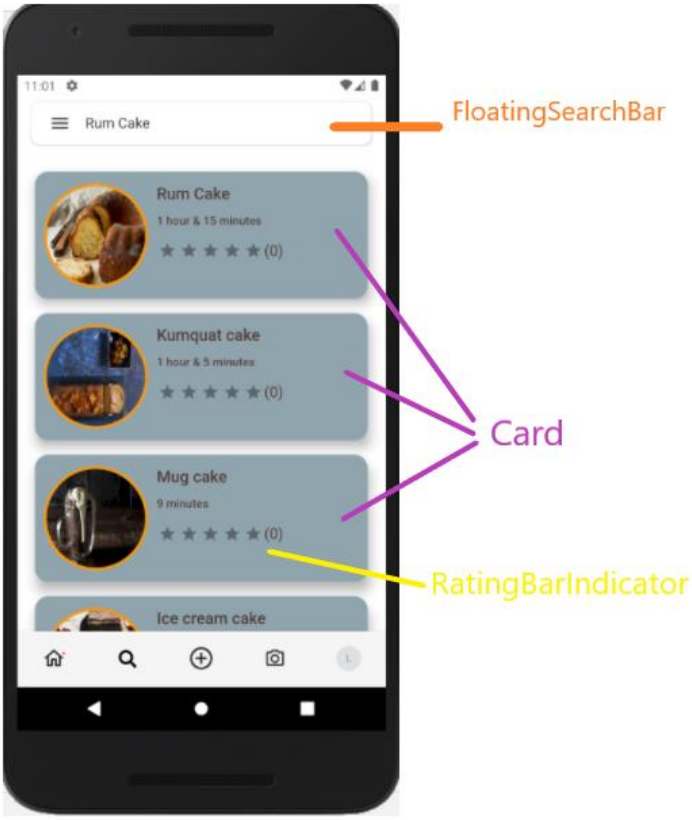
Each recipe contains all the information a user needs such as the list of ingredients, the cooking time of the recipe, the method and a way to be transfered in the original site the recipe was taken from.



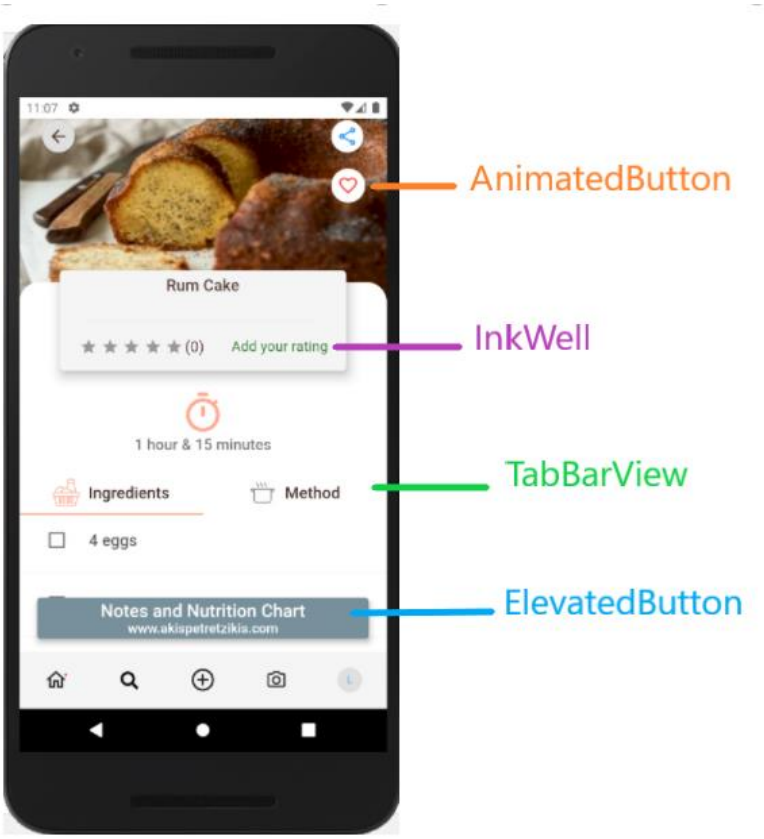
13.Recipe Page ingredients



14. Cooking method and tabs in Recipe Page

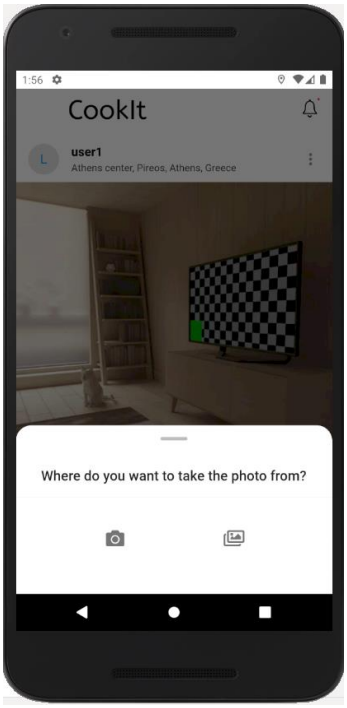


15. Recipe results page widgets

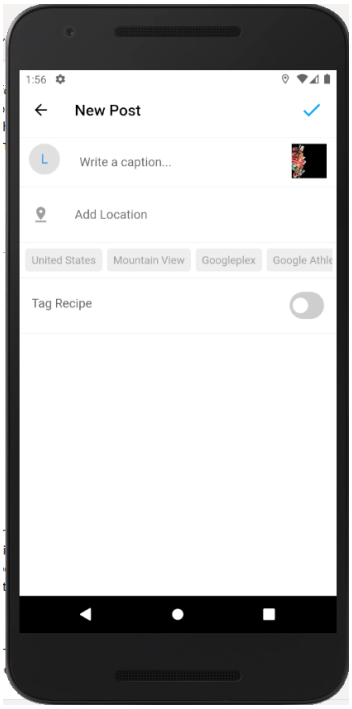


16. Recipe Page Widgets

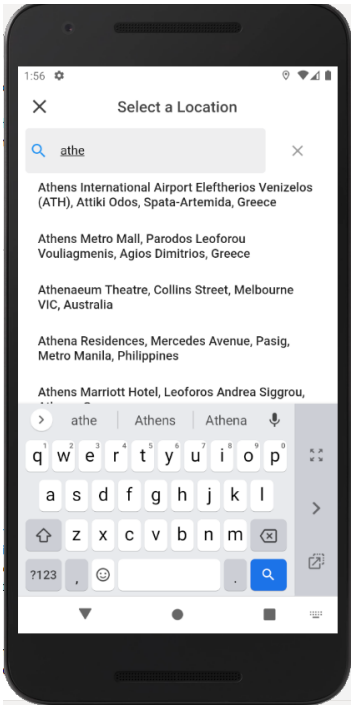
A modal sheet appears whenever a user presses the middle plus button of the bottom navigation bar. There a user can choose the type of content they want to upload (picture or video) as well as where they want to get it from (camera or gallery). After choosing, users are now able to fill a form refering to their post with fields such as a description, location and related recipe.



17. Uploading post modal sheet

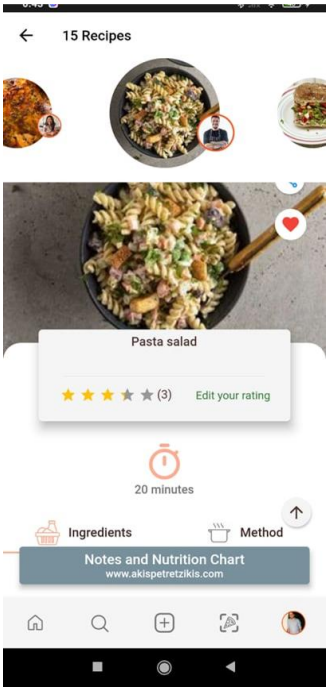


18. Create Post Page



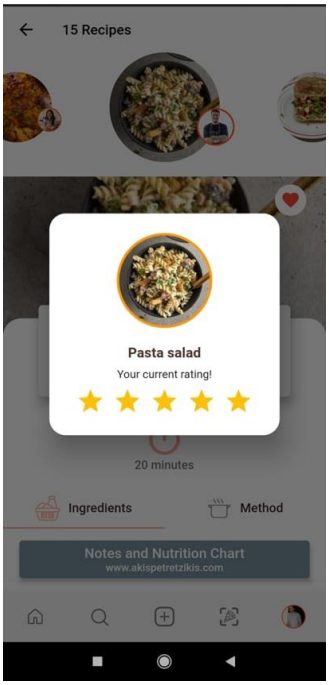
19. Search Location Search bar Page

Each user can see their favorite recipes listed horizontally and have access to all the information provided for each recipe.



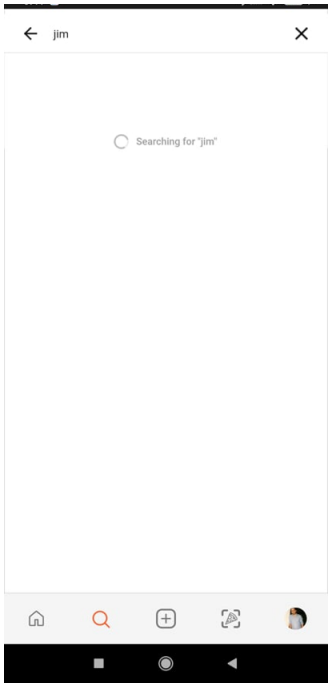
20. Favorite Recipes Page

Users can rate each recipe in a dialog that appears, showing their initial rating if they have provided one.



21. Rate recipe dialog

Users can search for other users. For each letter a user writes in the search field, a drop down list appears with all the related users based on that search.



22. Search Users Page