Better Than Yesterday: The first social network for volunteering

By Jacopo Minniti
(jacopo.minniti004@gmail.com)

Abstract

This paper aims to explain the thought processes, preparation, reasoning and realization of a mobile application called *Better Than Yesterday,* entirely conceived and developed by Jacopo Minniti.

Better Than Yesterday is a social network application which highlights the realities of volunteering and civil commitment, and enables users to organize and share their activities and initiatives.

The project is no-profit and the code is completely open-source. To view the repository on GitHub, click here, or search for 'Better Than Yesterday'.

At present,the beta version of the application is available in the Play Store. For security reasons, the application can be downloaded only if the user is registered to the official Google Group for beta-testers. To become a member of the group, you can simply click here, or search on Google Group 'Better Than Yesterday'. Afterwards, click here to download the beta from Play Store.

Otherwise, click here to download the APK. Please note that by downloading the APK, you will not be able to receive updates with new features and bug corrections.

Table of Contents

1 The idea

1.1 Many in need, few participants

Due to the pandemic, wars, and all the significant financial and social problems these events have caused, the last few years have been the hardest of the 21st century for many countries around the world. Although the nations that suffer the most are the poorest, even the wealthiest countries have been negatively impacted by the current global situation.

Here in Italy, my own country, poverty is dramatically rising, with about 9.4% of households in absolute poverty in the southern regions. This issue is accompanied by a higher unemployment rate, lower levels of education, worsening immigrant conditions, and acute social tension[1].

The incapacity and instability of the last governments prevented any successful social action aimed at mitigating these issues emerging, and instead created discontent among the population. While such severe and complex problems require a decisive intervention from the state and other "big players," civil society can play a decisive role through volunteering and civic participation.

In the last years, numerous Italian citizens began similar activities to cope with difficult times. In 2013, nearly one Italian over eight was engaged in non-compensated work somehow[2].

Although these data are reassuring, there are two major problems with civic participation in this country:

1. *Few Young people*: According to ISTAT, "the participation curve [for volunteering] reaches its maximum between the ages of 40 and 64, settling on values around 15%". Now, it is essential to consider that Italy is an old country, so it is expected that in many surveys, people who are aged above 40 are in the majority. However, it is also true that volunteering, and civic participation in general, have always been the prerogative of young people, so it would not be unrealistic to hope for a rise in participation. In addition, it is fundamental to consider not only pure participation but which roles are occupied by the youth. In most sector organizations, the average age is probably higher in executive positions.

[1] See Absolute poverty growing again by ISTAT 2020
[2] See Unpaid activities to benefit others by ISTAT 2014

2. *Little Organization*: In general,many realities of volunteering and civic commitment are born to meet a range of needs under challenging situations. Even if every new organization carries out an essential task in the city community, the lack of organization, the poor management of functions, and the obvious economic difficulties that the small, poorly financed realities encounter cause a dispersion of the work carried out and its effectiveness.

1.2 The need for progress

Without an efficient organizational structure, many of these realities encounter difficulties reaching their objectives or even surviving. From my personal experience, I met groups of volunteers who had to pay significant sums from their own pockets because the funds they received from calls were insufficient. In other situations, there were people whose hard work produced a poor result compared to their commitment, because of the difficulties in grasping specific issues involving stakeholders and funds or, more simply, not being able to advertise their services.

To tackle all the highlighted (and other) problems of this sector, it is essential to develop new tools and a new way of thinking. There are undoubtedly many ways this development could bedriven, and a single idea cannot solve complex matters alone; however, it can be a starting point and inspire other proposals.

Considering my interests in computer programming, and knowing the immense power of new technologies, if correctly used, I thought of a way that could help solve some of the problems mentioned above, at least partially.

The idea is a social network called *Better Than Yesterday*, which aims to increment the quantity and quality of volunteering and civil commitment.

1.3 Social network as a platform of reorganization

The project Better Than Yesterday has the objective of both addressing the issues listed above and being a tool for creating new opportunities. First, a social network is a way of organizing information understood by the youth, which finds itself comfortable with the idea of custom feed, direct interaction with the author, or real-time interaction between multiple users. As for *Figure 1*, about 80% of Americans[3] aged between 18 and 60 will use social media in 2022. Today, social networks are used and understood everywhere by every age group.

[3] While the survey is based on the American population, most European countries have similar percentages.

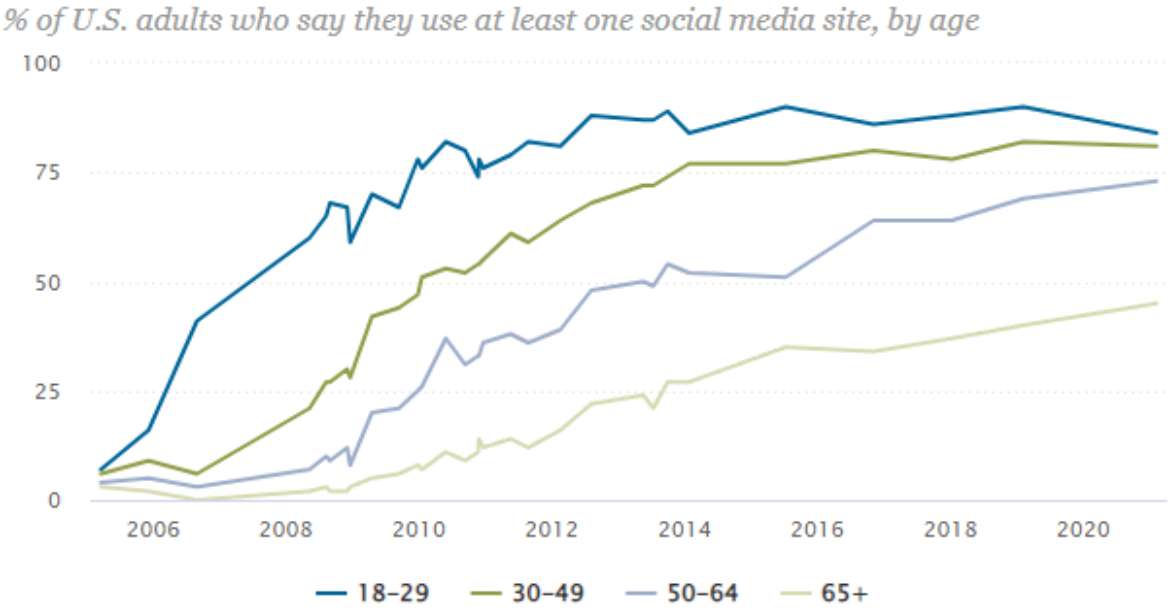

FIgure 1, usage of social media in the U.S. population.
Source: Social Media Fact Sheet by Pew Research Center

Second, a social network, if specially designed, permits a fast, detailed, and simple exchange of information. With "word of mouth" (a method widely used in my area), you can reach relatively few people, whereas a *post* on a social network allows you to reach several people and pass on a lot of information in the form of photos, text, dates, or links.
Third, it is possible to have far more data on how many and who participates. Participants can now ask for information about the activity and share the contents they like with their followers. With simple data analysis, volunteers can also understand their communities' needs.
Fourth, everyone is a *creator*. In Italy, you often need to be part of an association, cooperative, or parish to participate in public life, in order to support the organizational and monetary structure. However, creating a post to advertise and share an event is free, and having the opportunity to contact people with similar interests facilitates organizing in groups.

ADVANTAGES OF A SOCIAL NETWORK APPLICATION FOR VOLUNTEERING
Possibility of reaching more people, with particular attention to younger ones, who are less prone to volunteering
Fast, and simple content sharing
Better control over participants

Everyone can simply start an event or activity, without the need for a larger organization
Thanks to filters and recommendations, volunteers will easily find out and apply to activities which interest them
A post contains a lot of different information. In this way, a potential participant will have all the data he or she needs to decide if that specific activity appeals to them

Table 1, advantages of a new organization based on a social network

1.4 Preparation and planning

A good idea does not make a good project. A social network could reveal a valuable tool to reach my objectives.

A similar project requires the following skills:

1. *Full-stack development*: a social network is an application that requires knowledge of front-end development and, above all, back-end development, as interactions with a database in a server are continuous.
2. *Design: Modern understanding of economy and sociology reveals the importance of appearance. A user interface that can be understood by everyone (even 50+ users) and, at the same time, attracts attention is fundamental for building a successful social network.*
3. *Cost management: Manage costs efficiently and industriously, and secure funding for future improvements.*
4. *Project management: Find closed-beta participants, use possible future funding diligently, and popularise the app on the stores.*

It is important to remember that I, the author of this paper, am entirely alone in the preparation and realization of this project. For this reason, I am having to hone all the skills listed above.

I completed a beta version of the software in about a month and a half of work, plus about a month of learning the programming languages and others. The project is still young, and many features are to be completed.

However, considering the amount of work, other commitments, such as school and college preparation, and my obvious inexperience of such ambitious projects, I opted for a step-by-step realization. As I expand my knowledge and understanding, I will update the application. Of course, *software isconstantly evolving* , so this approach is actually standard among much bigger projects.

You will read the progress I have made until 1 October, 2022. By the time you are reading this, the application will probably be in beta testing, and improvements will have been made. I plan to release the first version of the application on the Play Store by March 2023.

2 Front-end

2.1 Introduction to Dart and Flutter

The first issue to solve is how to reach the most relevant sector of the public with the least work. Big companies have enough money, workforce, and time to invest in two codebases for Android users and one for IOS users. They create two different apps which are published on the respective stores. If, as is the case for social networks, the application can also be found in the form of a website, another codebase has to be developed. The programming languages used for Android are Java or Kotlin[4], for IOS Swift or Objective-C[5], and for web Javascript and CSS.

For this project to be completed, I should have developed, maintained, and debugged three different codes. Considering the few resources and time I had, this was impractical, at least initially.

The best solution to this problem is using Flutter or a Progressive Web App. Both frameworks allow writing a single code usable for Android, IOS, and the Web. While there is no better alternative, I opted for Flutter.

Flutter is an SDK developed by Google and released in 2017. It is used in combination with Dart, an object-oriented programming language similar to C# and Java in syntax[6]. Exactly like Java, it presents a VM. While it was designed for Web development, today, it is mainly used with Flutter for mobile applications.

Dart is heavily object oriented, as everything other than *null*, is an object. Even functions are objects, and can be passed through classes' constructors, making callbacks. One thing Dart does well is *asynchronous operations*; they are intuitive and effective, and they permit the usage of `await` and `async`.

Another interesting feature is *Optional dynamic typing,* which combines the obvious advantages of type safety[7], with the possibility to declare a generic type dynamic or to simply use the keywords var and final when the type is obvious from context[8].

This hybrid functionality is particularly handy when you have to interact with a third party API. An example is `http` requests with json type, for which you do not have to create your own type.

---

[4] Recently Kotlin, expressly created for mobile development, is preferred by most developers. However, the discussion on which language to use is open and much debated. For further information see the official *Kotlin documentation* Comparison to Java | Kotlin

[5] In this case, the comparison is easier. Objective-C is quite old and most developers, in particular less experienced ones, prefer Swift. See Swift Vs. Objective C: Which One is the Best Option For iOS App Development in 2022.

[6] One important difference however, is that in Dart not everything has to be written inside a class, like Java. Moreover, more recent updates have introduced features that distance it from the aforementioned language (es. Null safety, the removal of the keyword *new, ecc.*.)

[7] See Type safety - Wikipedia

[8] An example is when the value of the variable is declared. In this case, convention wants *var* over the specific type. For example, var a = false is preferred over bool a = false. In fact, in the first case we get more information without being verbose. For further information, see Effective Dart: Usage

```dart
Map<String, dynamic> getData(int currentUserId) async {
  final url = Uri.https('YOUR_SITE.com',
        'file.php');
  final response = await http.post(url, body: json.encode({
          'userId': currentUserId,
        })); //get data from database
  dynamic data = json.decode(response.body); //dynamic is a generic type
  //...
  // if at some point you are pretty sure of the type of data,
  // dart makes casting incredibly easy.
   try {
    final dataMap = data as Map<String, dynamic>;
    return dataMap;
    } catch (e) {
        rethrow;
    }
}
```

Hot reload is a functionality specific to the dart language when combined with Flutter. Hot reload allows the developer to experiment easily with changes in code by quickly rebuilding the widget tree. Contrary to hot restart, hot reload injects just the updated code into the DVM, which means that the app's state (and current screen) is not lost. Buffering time varies according to the vastness of the changes, but for most cases, it is under three seconds.
In Flutter, most classes extend a *Stateless* or *Stateful* Widget. The fundamental difference between the two is that, while the *state*[9] of a stateless widget can not be altered after the widget was built, a stateful widget can dynamically change its state.

The state changes, most of the time, when the `setState(VoidCallback fn)` method[10] is called.

2.2 Thinking through the main features

One of this application's aims is to involve more young people, so it has to replicate some key features of social networks. However, a mindless copy, without understanding the specific needs of the potentially interested user, would only result in a bland imitation. For this reason, BTY preserves elements drawn from various popular social networks and, at the same time, introduces (and will introduce) new features specific to the application's purpose.

Some of these features, which I will deepen later, are listed below:

[9] With state we refer to any data contained in the memory of the app.
[10] See setState method - State class - widgets library - Dart API

- *Likes System: Every site or application that allows interaction among users has a like system. It can be used for many reasons, such as personalizing feeds, ordering posts by an objective indicator, permitting the quantification of popularity for a given post, or giving the user a tool to express appreciation.*
- *Shares: The user has the opportunity to share posts he or she particularly likes, so that followers will see them more easily.*
- *Creation of post:* Every user can create a post representing an activity or project he or she is organizing. Every post is characterized by a *title, thumbnail, description, requirements*[11], *date of the activity, time of the day, location, the maximum number of* participants[12]*,* and *additional images.* Every post can later be deleted.
- *Participation*: *In the particular context of the app, liking or sharing is not enough. A post represents an activity, so every user can decide to participate.*
- *Details screen*: In most social media, the post, graphically, is represented as a card that contains the post's information. However, an activity is characterized by much information, so loading it all by default would have meant worse performance and impracticable scrolling[13]. On BTY, to see additional information, a click on the postcard is enough; a detail page will open and reveal all the information.
- *Personalized Feed*: Different screens for the different pages are needed to guarantee the user an optimal experience. By just accessing the pages on the bottom navigation bar, the user has the possibility to view different posts, all related to his or her interests.
- *Following system*: You can follow a user by simply clicking a button. The user visualizes all up-to-date posts by his or her followings.
- *Search for users*: Through a search bar, it is possible to search for users with similar usernames to that typed.
- *Trend posts*: The top ten trend posts of the moment are visualized by every user on the explore page.
- *Categories*: To permit a more personalized experience, every user and post is characterized by some filters. These filters correspond to the most popular categories of volunteering. Every post has up to three filters, and every user has up to ten.
- *Verified user*: It is possible to request "verified," which certifies that the user in question truly represents the organization or public figure in question.
- *Edit profile*: Some decisions made during the user's profile creation can be later modified. For example, the profile picture.
- *Comments*: On every post it is possible to view, like and write comments.

[11] If left blank, "no requirements needed" will be considered
[12] If left blank, 1000 will be considered
[13] To scroll just a single post with a long description could take many seconds.

2.3 The main screens

All screens are accessible through the five main pages. These five pages are, in turn, accessible via the bottom navigation bar and are in order: home page, explore page, new post page, messages page[14], and personal profile page.

The *home page is* the default page when the user opens the app and is already logged in. It is formed by an app bar in which the title is displayed. Next to it, in the right-upper corner, is the *setting button*. The user can ask for verification or delete his or her account by accessing the setting screen. The *posts list*[15] occupies the vast majority of the page. When no post has to be visualized, the message "No posts to see" appears in the center.

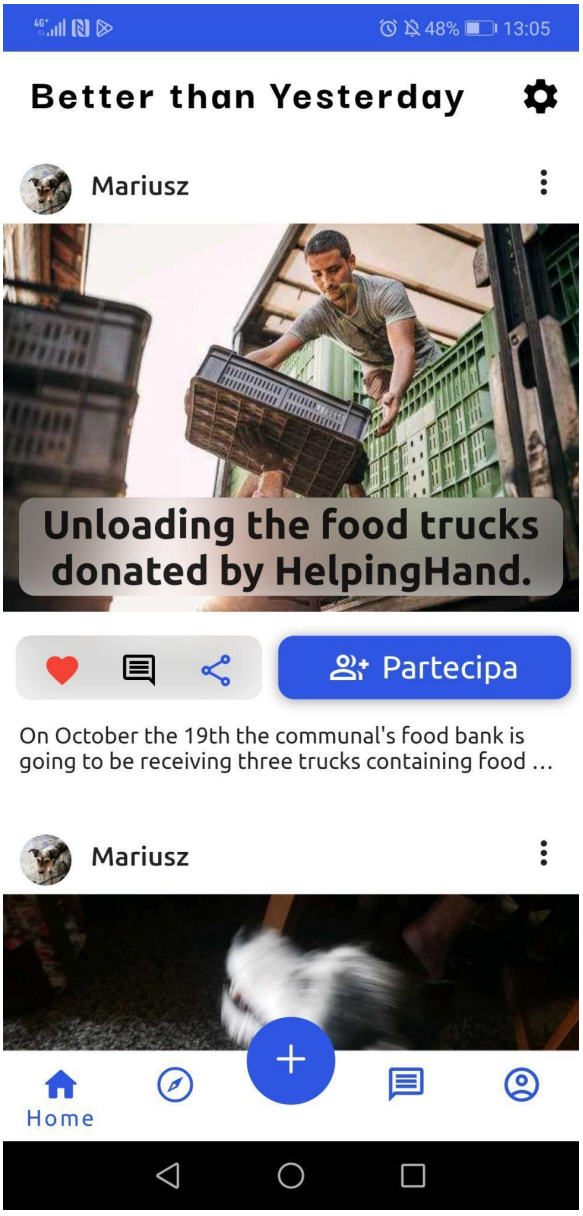

Figure 2, The home page

[14] This is still not implemented as for the current version of the Beta.
[15] The specific feed of every page is topic for the next paragraph.

The *explore page* is divided into three parts. In the top, a search bar for users can be clicked to access the *search users screen.* This screen extends a stateless widget, and to implement suggestions and other classic search bar features, the `showSearch`[16] method is used. It accepts two parameters: the `context` of type `BuildContext`, and a `delegate` of type `SearchDelegate`. `SeachDelegate` is a built-in abstract class, so to actually implement it, it is necessary to override its methods. There are four methods:

- `buildActions`: It returns a list of widgets. Most times these widgets are `IconButtons`, which perform some useful actions. I personally used this method to implement a "clear research" button.
- `buildResults`: It is mainly used to return the list of suggested users based on what the user typed.
- `buildLeading`: A method returning a widget displayed at the start of the search bar. Built a simple back-arrow button to return to the explore page.
- `appBarTheme`: The aesthetic theme of the appbar.

When one of the suggestions is clicked, the user profile page is returned.

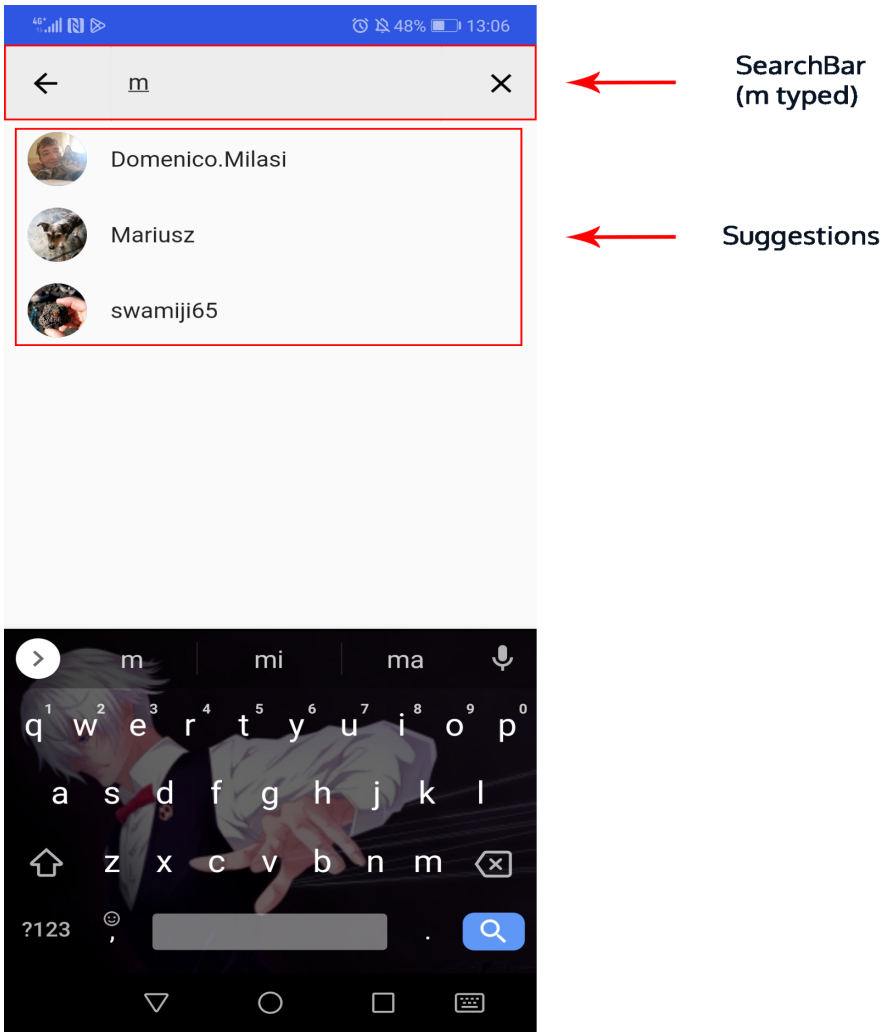

Figure 3, Screen when the search bar is clicked.

[16] See showSearch function - material library - Dart API

Below the search bar, are displayed the trend posts and categories. When one of the categories is clicked, a page is opened with the title of the category as appbar, and the list of posts of that category. These trend posts and categories are organized inside a `TabBarView`. `TabBar` and `TabBarView`[17] are two built-in widgets. The `TabBar` represents the text (i.e. trend posts or categories), and when clicked it visualizes under it the corresponding `TabBarView`; this is simply a horizontal scrollable list. By default, the `TabBar` is set to display trend posts.

Finally, tre rest of the page is occupied by the *post list*. A `Divider` separates posts from the other elements.

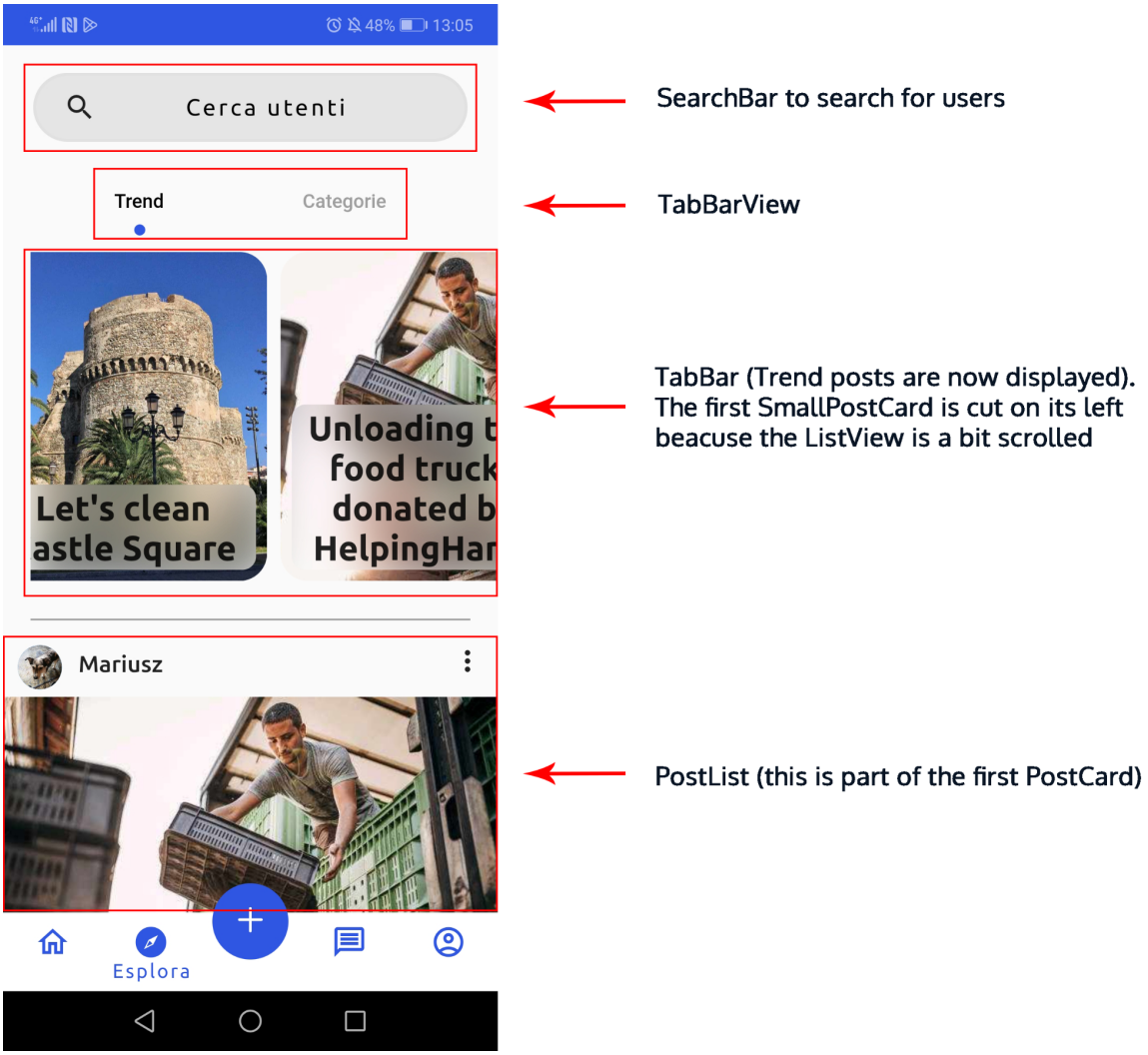

Figure 4, Explore screen.

The central page, accessed through the plus sign button, is the *new post page*. The app bar is in this case a `TextButton`, which creates the post when clicked, and if all data was

[17] See Work with tabs | Flutter

provided. Starting from above, the user inserts the thumbnail. Then he or she picks a date through a date picker, and a time of day through a time picker.

To select the location, the user clicks on the corresponding button, which opens a new screen where they can write down activity's location and pick one of the suggestions given by Google Maps[18].

After this, the user has to insert a description of at least 500 characters. Now, there are two optional parameters: requirements and maximum number of participants; if the `Chip` is activated a `TextField` is shown. If not inserted, the default values of "No requirements requested" and 100 respectively, are considered.

The user has to select from his or her gallery or camera to insert one or more additional images. All these images are displayed in a horizontal scrollable list. Finally, the user has to select from 1 to 3 categories.

The fourth and last page is the *personal profile page*. The app bar contains the username and three buttons. One is for logging out, one makes the user navigate to the *participations screen*, and one is for the edit profile screen.

In the *participations screen* the user is presented with three lists of posts:

1. *Participations:* Under this title are listed all the posts the user participates in. Past participations are excluded. By clicking on the post, the details screen of the post will be displayed.
2. *Waitlist:* Here are grouped all the posts to which the user requested to participate, but has not yet been accepted . By clicking on the post, the details screen of the post will be displayed.
3. *Requests:* The list of all the user's posts to which at least one other user has requested to participate. By clicking on the post, the *accepts participants screen is* displayed. All the users who requested to participate are grouped here. By clicking on "Accept", you authorize the user to participate in the activity.

[18] For further information on this topic, see paragraph 2.9 *Google Maps API*

Figure 5, Participations screen.

The *edit profile screen* allows the user to modify some of the information provided during the creation of the profile. In particular, the user can edit: profile picture, bio, location and filters. After clicking the save button, the app is restarted[19] and changes immediately take place.

[19] This restart is performed through the package `restart_app` and its static function `Restart.restartApp()`. See restart_app | Flutter Package

Figure 6, Edit Screen. In order, the widgets to appear are: Text Field for the bio, ChooseLocation, Filters Grid (the filters in blue are the ones previously selected)

Immediately below the personal profile page appbar, the profile picture is visualized; next to it there are some stats like the number of posts, followers and followings. Below all of this, there is a TabBar with two elements: user's posts and shared posts.

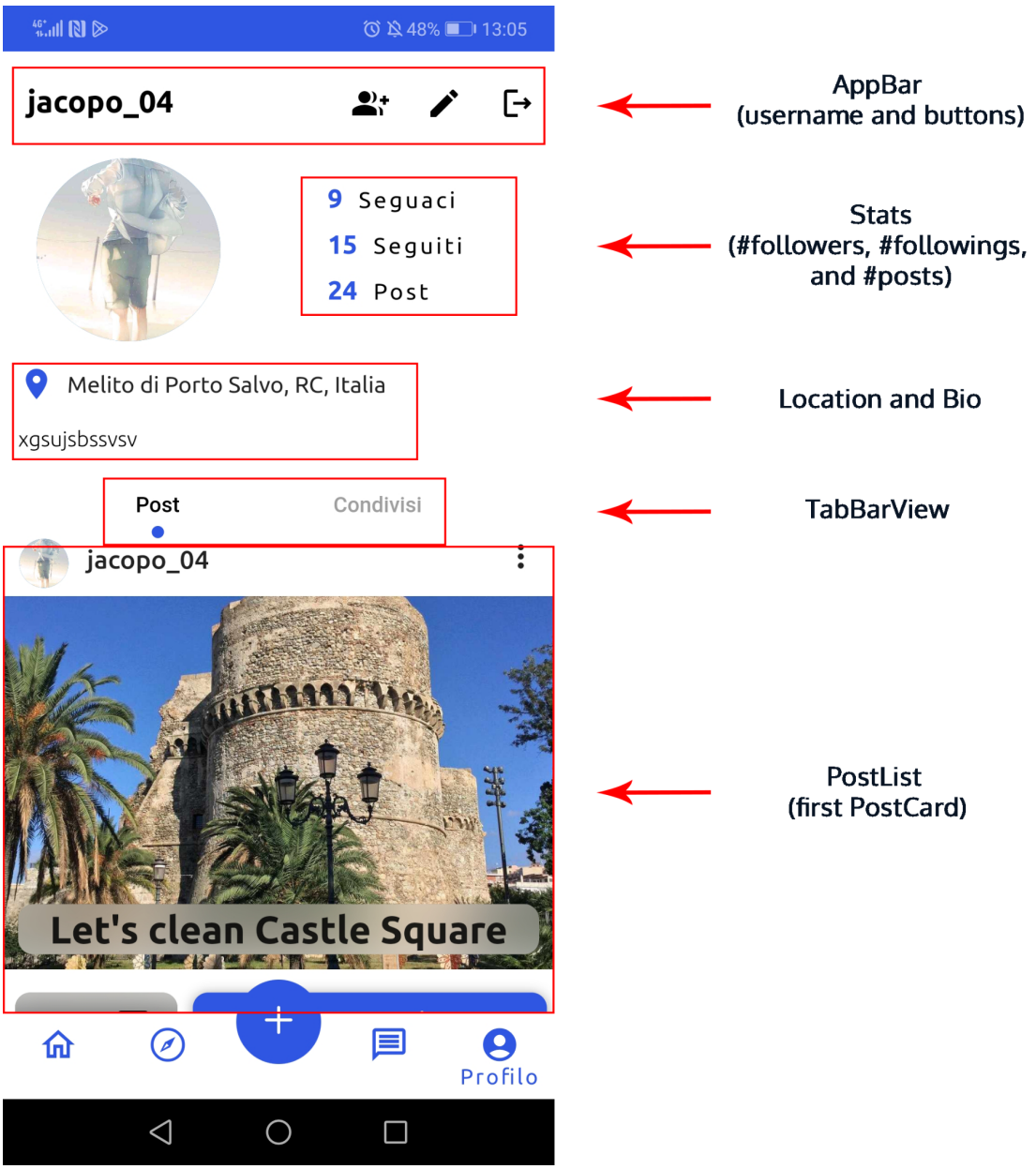

Figure 7, Personal profile Screen. If the profile was not of the user logged, in place of the three buttons in the app bar, the FollowingButton would be displayed

2.4 The different feeds

Three of the four main pages present one or more *posts lists.* Each one of them has a different objective, as it contains different posts, but it is graphically equal. For this reason, instead of doing boiler plate code, throughout the code I use the same widget which I named `PostList`. Given the importance of this widget, it is interesting to analyze it.

```
import 'package:Flutter/material.dart';

import 'package:scrollable_positioned_list/scrollable_positioned_list.dart';
```

```dart
import 'package:uuid/uuid.dart'; // package for generating UUID in Flutter


import 'post_card.dart'; // the widget which represents the post card


class PostList extends StatelessWidget {
  final List posts; //List of type <Post>
  final int currentUserId;
  PostList({
    Key? key,
    required this.posts,
    required this.currentUserId,
  }) : super(key: key);
  final itemScrollController = ItemScrollController();
  final itemPositionsListener = ItemPositionsListener.create();


  @override
  Widget build(BuildContext context) {
    return posts.isEmpty
        ? const Center(child: Text('Nessun Post da visualizzare al momento.'))
        : ScrollablePositionedList.builder(
            shrinkWrap: true,
            itemCount: posts.length,
            itemBuilder: (context, index) {
              return Padding(
                padding: const EdgeInsets.only(bottom: 20),
                child: PostCard(
                    post: posts[index],
                    currentUserId: currentUserId,
```

```
                    heroTag: const Uuid().v1()),
              );
            },
            itemScrollController: itemScrollController,
            itemPositionsListener: itemPositionsListener);
  }
}
```

While Flutter Material package provides a widget called `ListView`[20], which create a scrollable list, numerous github discussions and stackoverflow questions, and report lagging problems[21]. Multiple solutions are possible, and one which perfectly fits my situation is the usage of a package called scrollable_positioned_list[22]. Other than presenting various useful functionalities, this package solved the lagging problem.

The `ScrollablePositionedList.builder()` works almost identically to Flutter's native `ListView.builder()`, which is the recommended constructor to use for longer lists. The shrinkWrap property, when set to true, automatically shrinks the list as much as possible, and the itemCount property accepts the list's length as value. The itemBuilder is where we define the element of the list. In this case a PostCard widget, which accepts as arguments a post, the currentUserId, and the heroTag, which is an ID used for animation purposes[23]. Lastly, we pass the controllers we had instatiated before the build method.

As stated earlier, every feed displays posts which meet certain conditions. In the following table are listed all feeds, with the respective conditions[24].

Name	Screen where displayed	Conditions
Home Feed	Home Page	Posts where the author is followed by the user, or the

[20] See ListView class - widgets library - Dart API
[21] Search on web *Flutter listview laggy* for further information. Even Flutter documentation itself provides possible debugging, however many users report problems even after following Flutter team instructions. Another interesting package I recently found (which I will test in future) is keyframe; see keframe | Flutter Package
[22] See scrollable_positioned_list | Flutter Package
[23] See Hero animations | Flutter
[24] In the backend section the MySql queries necessary to achieve these feeds, will be explained in details

		post is shared by one of the followed users. Posts' date have to be greater than the current date.
Explore Feed	Explore Page	Posts where one of the filters is in common with the user's filters, and are localized at a radius of maximum 65 Km from the user's location. Posts' date have to be greater than the current date.
User Feed	Personal Profile Page or (another) User Profile Page	Posts where the author is the user
Shares Feed	Personal Profile Page or (another) User Profile Page	Posts which are shared by the user
Trend Feed	Explore Page	Posts whose trendPoints[25] are the ten highests. Posts' date have to be greater than the current date.
Category Feed	Explore Page	Posts which have as category the category selected, and are localized at a radius of maximum 65 km from the user's location. Posts' date have to be greater than the current date.

Table 2, Feeds used throughout the application

2.5 Postcard

[25] As for now, trend points are simply given by the number of participants. In the future, I will assign a more complex expression where other factors such as likes, and distance in time from the event are considered.

The elements which are built by PostList are of the class PostCard. Contrary to PostList, PostCard is a more complex class, so for details I suggest you visit the github link to the code. The PostCard is where all the essential information of every post is visualized.

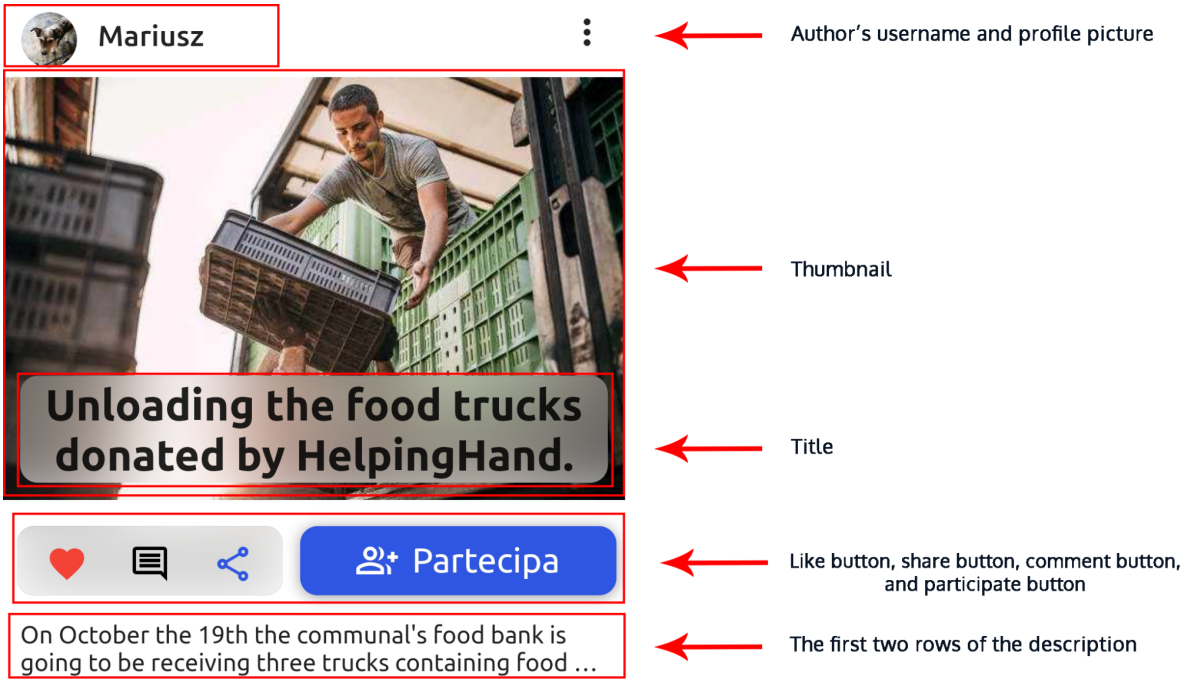

Figure 8, PostCard change communal's

Starting from the top, the two elements are the profile picture and username of the author. By clicking one of those, it navigates to the profile page of the author. Below, it is found the thumbnail, and over it the title. Below the thumbnail, are positioned all the buttons. The like, share and participate buttons have to interact locally and with the database. Here, we are interested in the first.

The *like button*, when clicked changes its color: filled in red when the post is liked by the user, empty otherwise. It also increments or decrements the `likes` property of the post, which measures the number of likes the post has received.

The *share button*, when the post is shared is blue, otherwise is black. When clicked, the *Shares Feed* of the user is immediately updated.

The *participate button* is blue with "Participate" written above in white. When clicked the colors reverse and the text changes in "Cancel". The consequences of clicking this button are fundamentally related to the backend, so they will be treated in the next chapter.

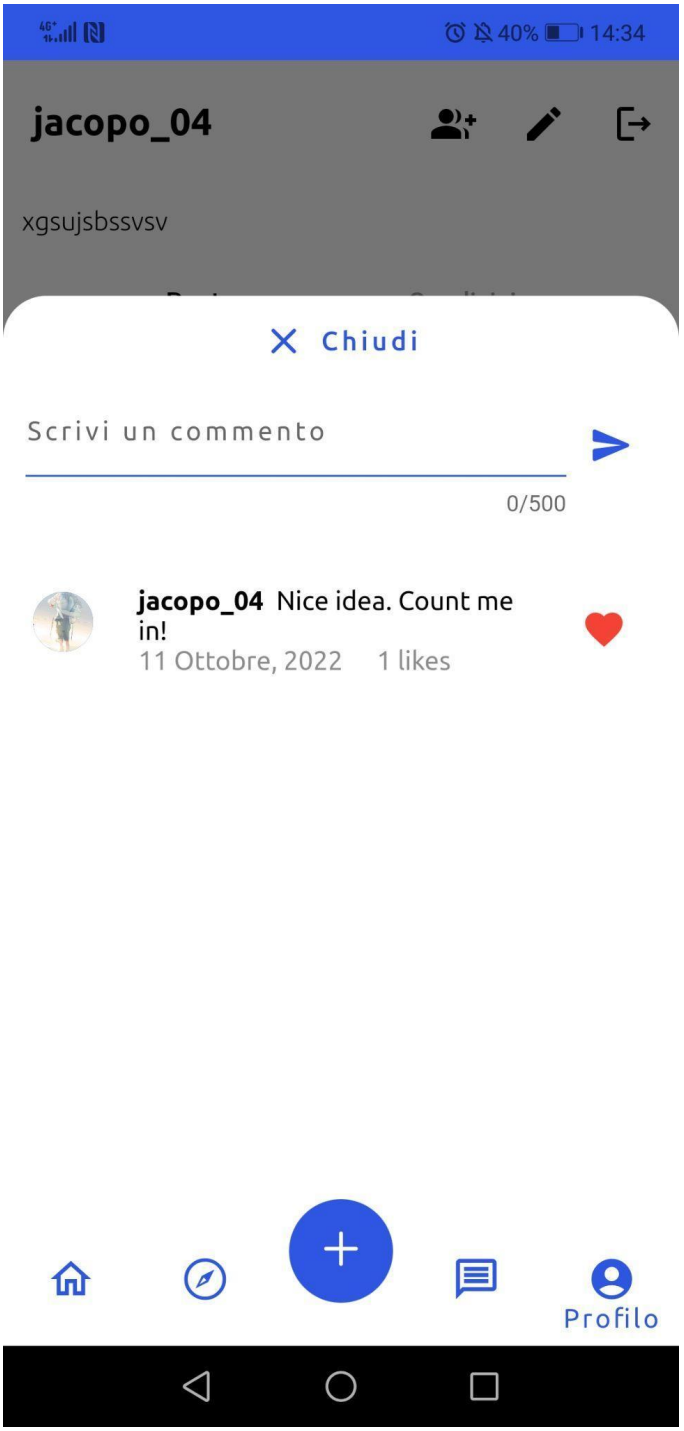

Figure 9, Comment ScrollableSheet. In this case, only one user commented on the post

The *comment button*, when clicked, opens a bottom scrollable sheet, which occupies about 50% of the page. Here, all the comments of the posts are displayed. Every comment is defined by a model in the *Comment.dart* file.

```
class Comment {

  final int postId;

  final int commentId;
```

```
final int userId;
final String username;
final String body;
final String profilePictureUrl;
int likes;
final String createdAt;
bool isLiked;

Comment(
    {required this.username,
    required this.postId,
    required this.commentId,
    required this.userId,
    required this.body,
    required this.profilePictureUrl,
    required this.createdAt,
    required this.isLiked,
    required this.likes});

Map<String, dynamic> toJson() {
  return {
    'username': username,
    'body': body,
    'commentId': commentId,
    'postId': postId,
    'likes': likes,
    'userId': userId,
    'isLiked': isLiked,
```

```
      'profilePictureUrl': profilePictureUrl,
      'createdAt': createdAt,
    };
  }

  static Comment fromMap(Map<String, dynamic> data) {
    return Comment(
      likes: int.parse(data['likes']),
      createdAt: data['createdAt'],
      username: data['username'],
      userId: int.parse(data['userId']),
      profilePictureUrl: data['profilePictureUrl'],
      isLiked: data['isLiked'] == '1' ? true : false,
      commentId: int.parse(data['commentId']),
      postId: int.parse(data['postId']),
      body: data['body'],
    );
  }
}
```

The actual information displayed is: the username and profile picture of the author, the body of the comment, the number of likes and the date of publication. Next to every comment, there is a like button which works exactly as the one described above.

To write a comment for the post, it is enough to write a body and click the IconButton next to it. The comment is immediately added to the post so that the user can visualize it and like it if requested.

2.6 Details Screen

When the postCard is clicked, the user accesses the details screen for the given post. In this screen, the thumbnail goes to the background, and a new scrollable sheet partially covers it.

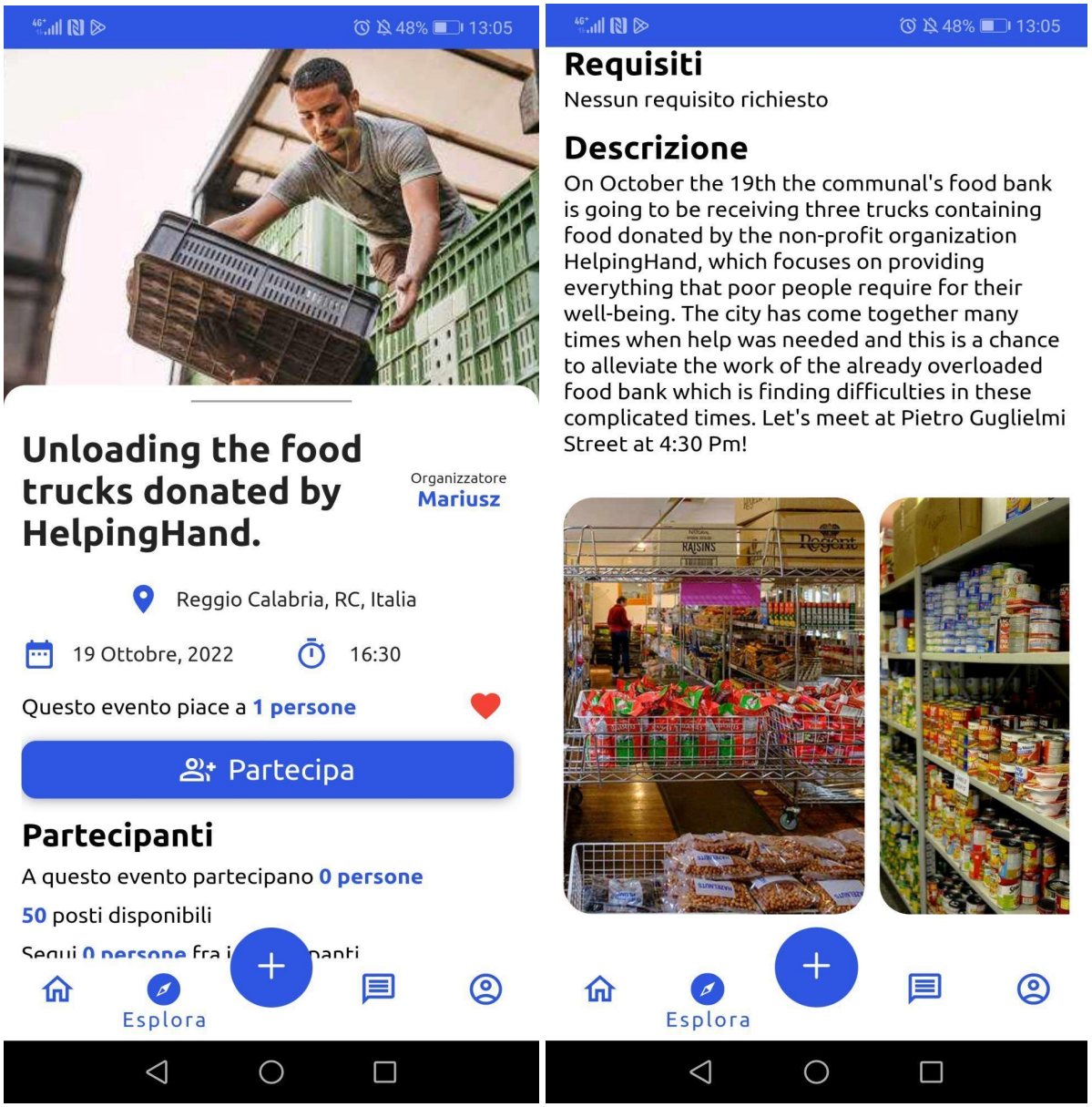

Figure 10, Details Screen. When scrolled, the scrollable bottom sheet occupies the whole screen. Notice how the corners are rounded in the first screenshot, but not in the second.

The details screen contains all the information about the post. The data which was already given, is passed as parameter into the constructor of the object DetailsScreen(), while a new query to the database requests the data needed. When the data is retrieved, and converted from json to a map, it is displayed onto the scrollable sheet. In this case, one interesting Flutter material widget called FutureBuilder, is used.

FutureBuilder requires two arguments: a future, and a builder. When the future passed is completed, so the query is completed and data retrieved, the ConnectionState of the snapshot object changes to isActive. In this way, it is possible to have different behaviors based on the state of the future. Consider the following simple example:

```
import 'dart:convert';

import 'package:http/http.dart' as http;
```

```dart
class FutureBuilderExample extends StatelessWidget {
  const FutureBuilderExample({Key? key}) : super(key: key);


  Future<Map<String, dynamic>> getData() async {
    final url = Uri.https('YOUR_SITE.com', 'file.php');
    final response = await http.get(url);


    try {
      final data = json.decode(response.body) as Map<String, dynamic>;
      return data;
    } catch (e) {
      return {};
    }
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
        backgroundColor: Colors.white,
        body: FutureBuilder(
          future: getData(),
          builder: (context, snapshot) {
            if (snapshot.connectionState == ConnectionState.waiting) {
              return const Center(
                child: CircularProgressIndicator(
                  color: Colors.blue,
                ),
              );
```

```
            }
            final data = snapshot.data! as Map<String, dynamic>;
            return Center(
              child: Text(data['title']),
            );
          },
        ));
  }
}
```

If the data is not ready yet, users will view a buffering. Other than connectionState and data, the object snapshot has other useful properties, such as hasError or hasData.

Considering the continuous connections a social network app has to establish with the database, a widget like FutureBuilder(), which awaits results without causing errors, is fundamental, and thus extensively used throughout the code.

One of the pieces of data which is retrieved is the list of participants who happen to be followers of the user too. They are displayed as a horizontal scrollable list of their profile pictures. If clicked, the user navigates to the profile page of the selected user.

At the bottom of the page are displayed the requirements and additional images for the post.

2.7 State management

One of the most complex topics on mobile development is state management. While frameworks such as Android SDK and iOS UIKit are *imperative*, Flutter is *declarative*. This, simply put, means that the data management changes drastically[26].

Flutter, to deal with state management, offers two options depending on the type of state. If the state is *ephemeral*, which means that it is used only in a single widget, using something like a StatefulWidget, is sufficient. Instead, for more complex situations, such as for data which is used in many different screens and situations, the state takes the name of *app state*. With a declarative framework, without using advanced techniques to manage this app state, it would be necessary to use an extraordinary amount of parameters in constructors and callbacks[27].

[26] See State management | Flutter
[27] See Callback (computer programming) - Wikipedia

To manage the app state, multiple architectures and patterns exist, but the most used and simple is using the `Provider` package[28]. When using it, it is fundamental to create some classes which implement `changeNotifier`[29]. These classes are where all the data used in other parts of the code is stored and controlled through method.

In BTY, there are four providers. They are:

- *Posts*: Here are declared all the lists of posts for the various feeds (`homePosts`, `personalPosts`, ecc.). In this class are also written the functions for the `http` requests to query database tables related to posts.
- *Users*: Here is stored all the data related to the current user, such as its ID, username and other. In this class are also written the functions for the `http` requests to query database tables related to users.
- *Comments*: Here is the list of comments, which is updated every time a user clicks on the comment button to view the comments of the selected post. In this class are also written the functions for the `http` requests to query database tables related to comments.
- *AuthenticationService*: Here is stored the authentication state, the user's firebaseUserId, and email. Above all, here are written all those methods related to the authentication process through the Firebase Authentication Service[30].

As you notice, these classes do not only manage the state locally, but they also host the functions to query the corresponding tables of the database too. This means that with a single class it is possible to globally manage posts users, etcetera.

While the state management state can be improved (and I will certainly do so while acquiring more experience), it is true that BTY does not require, as of now, a highly complex state management to produce efficient and solid results.

2.8 UX and launcher

As stated earlier, the UX and UI of an application is as important as functionality. The user interface has to be easily understandable and capable of attracting users.

Better Than Yesterday has, overall, a fresh design which, however, does not reflect most modern experiments in the matter of mobile applications. Given the variable and vast user base this app is aimed at, a design orientated too much in one direction could have discouraged one of the segments. I preferred a kind of mix. The main colors are white, mainly used for background, and electric blue. To be more specific, these are the colors which are used the most:

- White: [r, g, b] = [255, 255, 255]
- Dark Blue: [r, g, b] = [8, 37, 97]
- Electric Blue: [r, g, b] = [47, 86, 226]

---

[28] See provider | Flutter Package. This package is among the most liked package on the whole platform of pub.dev

[29] In dart to implement a class, the keyword `with` `is used`

[30] For further information see paragraph 3.1 *Understanding the authentication process*.

- Opaque Blue: [r, g, b] = [145, 169, 215]
- Light Blue: [r, g, b] = [255, 217, 229]

In the vast majority of cases, these colors are used with opacity 1 and alpha 255, but there are some exceptions. The most used blue is *electric blue,* and it is also the most iconic. Every social network has a color with which it is associated.

The blue color appearsalso in the *logo* (or *launcher*) of the application. This launcher was made through an artificial intelligence called *Midjourney*[31]. By using some keywords like *logo, minimal, blue color, design and deer* the AI produced an amazing result. Afterwards, some changes and improvements were applied through photoshop.

Figure 11, Logo

I chose an animal for two reasons. First, it is simpler to think of a logo through a distinct subject like an animal; second, for the average user, a stylized animal is at an optimum level of complexity to be remembered. A too complex figure (i. e. a non stylized animal), has too many details, while a highly abstract logo is too anonymous. Finally, I chose a deer because it is a neutral animal that in common language inspires majesty and confidence.

[31] See Midjourney

From the launcher, a splash screen displayed during the initial loading was created using photoshop.

Figure 12, Splash Screen

Many features in the main pages are inspired by various social networks (mostly Instagram). By doing this, the user does not need to learn all the functionalities from zero, but will understand most of them immediately as they visually recall applications they already know.

Much originality and effort is put into the bottom part of the *postcard*, where all the buttons are located. Explaining the way all those effects were obtained would be too verbose, so I will directly put in the corresponding code lines:

```
      Row(
        mainAxisAlignment: MainAxisAlignment.start,
        children: [
          Padding(
            padding: const EdgeInsets.only(
                bottom: 15, top: 15, right: 10, left: 8),
            child: Container(
              decoration: BoxDecoration(
                  color: Colors.black26,
                  borderRadius: BorderRadius.circular(10)),
              child: ClipRRect(
                borderRadius: BorderRadius.circular(10),
                child: BackdropFilter(
                  filter: ImageFilter.blur(sigmaX: 15, sigmaY: 15),
                  child: Container(
                    alignment: Alignment.topCenter,
                    height: 40,
                    decoration: BoxDecoration(
                      color: Colors.white.withOpacity(0.3),
                      borderRadius: BorderRadius.circular(10),
                    ),
                    child: Padding(
                      padding: const EdgeInsets.symmetric(
                          horizontal: 5, vertical: 2),
                      child: Row(
```

```
                    mainAxisAlignment: MainAxisAlignment.center,
                    crossAxisAlignment: CrossAxisAlignment.start,
                    children: [
                      Consumer<Posts>(
                          builder: (context, postsProvider, child) {
                        return NewLikeButton(
                          currentUserId: currentUserId,
                          isLiked: post.isLiked,
                          postId: post.postId,
                        );
                      }),
                      CommentButton(postId: post.postId),
                      if (!isCurrentUser)
                        Consumer<Posts>(
                            builder: (context, postsProvider, child) {
                          return NewSharesButton(
                            currentUserId: currentUserId,
                            isShared: post.isShared,
                            postId: post.postId,
                          );
                        })
                    ],
                  ),
                ),
              ),
            ),
          ),
        ),
```

```
          ),
          Expanded(child: Consumer<Posts>(builder: (context, value, child) {
            return NewPartecipateButton(
                currentUserId: currentUserId,
                postId: post.postId,
                isPartecipable:
                    post.maxPartecipants - post.partecipations > 0,
                partecipationStatus: post.partecipationStatus);
          })),
        ],
      ),
```

Other than the Consumer class, which is part of the Provider package, and the parameters passed to the buttons, everything above is about extetics.

To improve the visual experience, some animations are present. Most of them are transitions between screens. For these, a *slide right animation* similar to IOS native animations is used. For the transition to the DetailsScreen two animations are used: the first is the hero animation for the image, the second a slide bottom-top for the bottom sheet.

Other than the hero one, these animations are imported from the persistent_bottom_nav_bar_v2 package[32]. This package, as the name suggests, is the one used to recreate a modern and functional *bottom navigation bar.* The style used is the number 15, probably the most adept for the NewPostPage button (the plus sign), which creates a greater depth by being in a different position.

2.9 Google Maps API

Location is a piece of data which characterizes both users and posts. It is fundamental for BTY as, for multiple feeds, posts are shown based on the distance from the user: if it is superior to 65 Km, posts are not shown.

When the user has to select a location, the screen ChooseLocationScreen is displayed. This screen simply contains a search bar. When something is typed in it, suggestions start to appear about locations with similar names.

[32] See persistent_bottom_nav_bar_v2 | Flutter Package

When the user picks one of these suggestions, the name of the location and the corresponding coordinates are sent to the database and stored. To extract the coordinates from the name of the location, a library called *geocoding* with the function `locationFromAddress` is used[33].

Of course, to use Google Maps functionalities an API key is necessary,along with a Google Maps package[34]. This key is obtained through Google Cloud Console and it is different for Android and apple users[35]. Actually, for the usage in the app, just three components of the Google Maps API are needed: *Maps SDK for Android*, *Maps SDK for IOS* and *Places API*[36].

The following code shows an example of how it is possible to use Google Maps in Flutter:

```
import 'package:Flutter/material.dart';

import 'package:google_place/google_place.dart';


class ChooseLocationScreen extends StatefulWidget {

  const ChooseLocationScreen({Key? key}) : super(key: key);


  @override

  State<ChooseLocationScreen> createState() =>

      _ChooseLocationScreenState();

}


class _ChooseLocationScreenState extends State<ChooseLocationScreen> {

  final _startSearchFieldController = TextEditingController();


  late GooglePlace googlePlace;

  List<AutocompletePrediction> predictions = [];


  @override

  void initState() {
```

[33] See geocoding | Flutter Package
[34] See google_place | Dart Package
[35] It is not always in this way, but it is the preferable strategy as it is more secure.
[36] For all information about pricing see paragraph 4.1 *Costs and fundings*

```
  super.initState();
  String apiKey = 'YOUR_GOOGLE_MAPS_API_KEY';
  googlePlace = GooglePlace(apiKey);
}

void autoCompleteSearch(String value) async {
  //you have to select a language for locations. In my case is italian
  var result = await googlePlace.autocomplete.get(value, language: 'it');
  if (result != null && result.predictions != null && mounted) {
    setState(() {
      predictions = result.predictions!;
    });
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Colors.white,
      leading: const BackButton(color: Colors.blue),
      elevation: 0,
    ),
    body: Padding(
      padding: const EdgeInsets.all(15.0),
      child: SingleChildScrollView(
        child: Column(
          children: [
```

```
            TextField(
              controller: _startSearchFieldController,
              autofocus: false,
              style: const TextStyle(fontSize: 24),
              decoration: InputDecoration(
                  hintText: 'A place of your choice',
                  hintStyle: const TextStyle(
                      fontWeight: FontWeight.w500, fontSize: 24),
                  filled: true,
                  fillColor: Colors.grey[200],
                  border: InputBorder.none),
              onChanged: (value) {
                if (value.isNotEmpty) {
                  //places api
                  autoCompleteSearch(value);
                } else {
                  //clear out the results
                }
              },
            ),
            ListView.builder(
                shrinkWrap: true,
                itemCount: predictions.length,
                itemBuilder: (context, index) {
                  return ListTile(
                    leading: const CircleAvatar(
                      child: Icon(
                        Icons.pin_drop,
```

```
                        color: Colors.white,
                      ),
                    ),
                    title: Text(predictions[index].description.toString()),
                    onTap: () {
                      Navigator.pop(
                  context, predictions[index].description.toString()
                      );
                    },
                  );
                })
            ],
          ),
        ),
      ),
    );
  }
}
```

3 Back-end

3.1 Understanding the authentication process

Every social network requires an account to associate to the user. There are multiple methods of authentication: authentication with email, Google, Facebook and others. As for now, BTY requires you to create an account through an email and password to access the actual content[37].

[37] In the near future, authentication through Google will be introduced .

To authenticate users, store their passwords, emails and access tokens BTY uses the *Firebase Authentication Service*[38]. Firebase is a group of services, deeply integrated with Google Cloud Console, developed by Google and widely used in modern mobile applications, in particular in those developed with Flutter[39].

Firebase Authentication is simple, secure, and provides a number of built-in methods and functionalities to recover password, verify email, and manage the access token[40]. In combo with *Firebase App Check*[41], Firebase Authentication assures a great level of security. In addition, using this service means that there is no need to store users' emails and passwords in the database. This prevents a priori any attempt, through *sql injections*, to obtain this sensitive data (the only actual sensitive data for this app).

On the client side, the authentication is made possible by two classes: `AuthenticationStream` and `AuthenticationService`. The former chooses the current page based on the authentication status. This *authentication status* is an `enum` value which is controlled by the latter. This is the `enum`:

```
enum Status {
   uninitialized,
   authenticated,
   authenticating,
   unauthenticated,
   noVerification
}
```

The state updates in real time, as the Firebase Authentication Service is connected to the client through a `stream`, which is instantly sent the current *Firebase User Token*[42].

```
User? _user;

Status _status = Status.uninitialized;

AuthenticationService.instance() : _auth = FirebaseAuth.instance {

  _auth.authStateChanges().listen((firebaseUser) {

    if (firebaseUser == null) {
```

[38] See Firebase Authentication
[39] The combination Flutter-Firebase is so common that in Flutter documentation there is a section dedicated to Firebase.
[40] Thanks to this, users do not have to login every time they open the application.
[41] See Firebase App Check
[42] This token stores various information about the user. For example its email or if the user previously verified his or her email.

```
      _status = Status.unauthenticated;
    } else if (firebaseUser.emailVerified == false) {
      _user = firebaseUser;
      _status = Status.noVerification;
    } else {
      _user = firebaseUser;
      _status = Status.authenticated;
    }
    notifyListeners();
  });
}
```

`notifyListeners()` is fundamental in the above code. When called all *listeners* change based on the new values of the provider. The only listener to the Authentication Service is the AuthenticationStream class.

In the `build()` method of the AuthenticationStream class, a `switch case` changes screen by referring to the `_status` variable defined in the above code:

```
      switch (auth.status) {
        case Status.uninitialized:
          return const SplashScreen();
        case Status.unauthenticated:
        case Status.authenticating:
          return const AuthenticateUserScreen();
        case Status.noVerification:
          return const VerifyScreen();
        case Status.authenticated:
          return FutureBuilder(
            future: userExist(auth.user!.uid),
            builder: (context, snapshot) {
```

```
            if (snapshot.connectionState == ConnectionState.waiting) {
              return const SplashScreen();
            }
            final userExist = snapshot.data as bool;
            if (userExist) {
              final posts = Posts();
              final users = Users();
              return MultiProvider(providers: [
                ChangeNotifierProvider(create: (context) => posts),
                ChangeNotifierProvider(create: (context) => users),
                ChangeNotifierProvider(create: (context) => Comments()),
              ], child: PagesController(posts: posts, users: users));
            } else {
              return const UserInformationScreen();
            }
          },
        );
      }
```

As you can see, If the user is *unauthenticated*, the `AuthenticateUserScreen()` is shown; this corresponds to the screen where the sign up or sign in takes place. If the user is authenticated but his or her email is not verified, the `VerifyScreen()` is shown (and an email is sent to the user's email).

Finally, if authenticated it enters a `FutureBuilder()`. The future `userExist()` simply checks if the `firebaseUserId` is already present in the database, which means the user went through account creation; it returns true, if a row in the `users_profiles` table with the same `firebaseUserId` exists, false otherwise. If true, it finally shows the `PagesController()`[43], which is the screen where the bottom navigation bar is created and all its children instantiated.

[43] PagesController() is a child of the Widget MultiProviders, through which the provider services discussed earlier are instantiated, and the corresponding listener is set up.

If the user eventually logs out, the Firebase Token updates, and the authentication status changes to unauthenticated.

3.2 Firebase Storage

Another service marked Firebase is *Firebase Storage*. While it is possible to store images on the database, it would have resulted in two problems:

1. *Performance Reduction:* An image is a far more complex data type than an integer or string. Storing an image in the database would have meant retrieving all that data for each post and user. Considering the large amount of posts and users' profile pictures BTY needs to load when the app is opened, performance would have dropped a lot.
2. *Cost issues:* While Firebase is famous for being pricey, the storage of images, in the particular context of BTY, ends up being cheaper than consuming the (relatively) few gigabytes offered by the database. Images are much heavier, so to avoid problems, at least for the first time, Firebase Storage is preferable.

To actually use the images stored, it is sufficient to first upload them on Firebase Storage, and immediately get the download link using the function `getDownloadURL()`[44]. This string will be uploaded into the database.

To load the images from the web using its link, the Flutter Widget `NetworkImage(String imageURL)`[45] is used.

To make things clear, this method uploads all the images of a post into Firebase Storage, and returns the links for each one:

```
static Future<List<String>> uploadPostImages(
    {required List<File?> photos, required String userId}) async {
  var imagesUrl = <String>[];
  for (var photo in photos) {
    var imageId = const Uuid().v1();
    final path = 'posts/$userId/$imageId';
    final reference = FirebaseStorage.instance.ref().child(path);
    final filePath = photo!.absolute.path;
    final lastIndex = filePath.lastIndexOf(RegExp(r'.jp'));
    final splitted = filePath.substring(0, (lastIndex));
    final outPath = "${splitted}_out${filePath.substring(lastIndex)}";
```

[44] This function is part of firebase_storage | Flutter Package
[45] See Display images from the internet | Flutter

```
      final resizedPhoto = await FlutterImageCompress.compressAndGetFile(
        filePath,
        outPath,
        quality: 45,
      );
      final uploadTask = reference.putFile(resizedPhoto!);
      final snapshot = await uploadTask.whenComplete(() => null);
      final url = await snapshot.ref.getDownloadURL();
      imagesUrl.add(url);
    }
    return imagesUrl;
  }
```

To optimize performance, before being uploaded all images are *compressed*[46], so that loading them will be faster and more efficient. Better Than Yesterday is not a social network focused on images, so having perfect quality is not necessary. However, providing too low resolution would have defeated the very sense of uploading images. So I have experimented with various trade-offs, and found a balance that seems to respect needs.

3.3 Firestore vs MySql

For a social network application, one of the most important decisions is the choice of the database. Today there are countless valid services: some more accessible, others cheaper, others more performing. To meet my needs, I needed a solid, stable, inexpensive database, with a lot of documentation to consult and that could satisfy a certain complexity required by an app like BTY.

Initially, I had opted for another Firebase service, called *Firestore*[47], which is a no-sql database very popular among Flutter developers. However, after a lot of work, and a considerable amount of problems, I decided to change to *MySql*[48]. Firestore was not able to satisfy the needs of a social network app, or at least, not the particular needs of this project.

---

[46] Compression is realized through an external package. See https://pub.dev/packages/Flutter_image_compress. To choose among the various packages with this functionality see Flutter the best way to compress image - Stack Overflow

[47] See Firestore | Firebase

[48] With InnoDB as storage engine

The following table compares the two databases on multiple subjects and fields important for BTY.

	Firestore	MySql
Database Type	Firestore is a no-sql database, which means there are no relations between tables.	MySql is the most used relational database which makes use of the Structured Query Language (SQL)
Database Structure	Firestore uses an original structure. The database contains collections, and each collection groups numerous documents. Each document has a number of fields[49].	In Mysql, a database contains multiple tables. Each table is composed of columns, which represent the fields of that table, and rows. Each row represents a set of values.
Documentation and resources	Firestore documentation, complete as it may be, seems to be lacking in detail. Furthermore, in some cases, the organization of the pages does not give an easy overview of all the pros and cons. However, the Firestore community, which has grown particularly over the past few years, helps a lot where documentation seems lacking. There are numerous discussions on GitHub, questions on Stackoverflow, or specialized sites that deal with the topic.	MySql documentation is extensive, incredibly accessible, and full of examples. The organization is very linear, so that every developer can learn the basics step by step. In addition to the huge community, noteworthy is the presence of numerous sites, which allow you to try MySql on the web.
Speed and scalability	Firestore is incredibly fast, both in reading and writing. Moreover, it scales pretty well, as the dimensions of the collection are not	MySql is highly optimized. Every table can contain up to 700 million rows, and before the first hundred million, size does not give

[49] An example of this structure is: Database->Posts->postId->{title, thumbnail, numLikes, …}

	relevant[50].	performance issues. However, speed and scalability are incredibly complex topics, which go far beyond sizes[51].
Geo-queries[52]	Firebase geo-queries are problematic. Due to some constraints, it is not possible to query both on altitude and longitude, which means that to perform these kinds of queries, a mechanic called *geohash*[53] is used. While for less complex geo queries this solution is sufficient, when other conditions are needed in the query (filters, date, etc), this approach is not efficient and requires the usage of multiple iterations.	MySql queries do not present constraints in this context. Moreover, geo-queries were optimized with the introduction of the *Spatial Index*[54] and other functionalities.
Queries complexity	Firebase presents numerous limitations, which prevent more complex queries. While the situation has been improving in the recent years, Firestore is still not able to perform many queries which other databases can do[55].	MySql grants a number of keywords and features which allow the developer to perform incredibly complex queries.
Join	Join queries are not possible on Firestore. There is a workaround, but it requires the usage of cloud	In MySql three different types of joins are possible. A single query can retrieve data from multiple tables

[50] Using Frank van Puffelen's words, engineer at Firebase Google, "in Firestore the amount of data in the collection has no effect on the query performance. So if you get 10Mb of data from 10Gb of data in the collection, it will take the same amount of time as when you that the same 10Mb from a 10Tb collection". See Firebase Firestore Query Performance - Stack Overflow

[51] To start to deepen the topic, I suggest reading How big can a MySQL database get before performance starts to degrade - Stack Overflow

[52] "Geo-query" refers to those queries where spatial distance is a one of the factors to consider.

[53] See Geo queries | Firestore | Firebase

[54] See MySQL 8.0 Reference Manual :: 11.4.10 Creating Spatial Indexes and Spatial queries with MySQL | End Point Dev

[55] See Perform simple and compound queries in Cloud Firestore | Firebase (Scroll to the end of the page)

	functions, which means they are less efficient and more expensive[56]	and even the same table multiple times.
Data types	Firestore includes all the basic data types[57].	MySql supports all basic data types too, but permits a greater handling. For example, it is possible to indicate how many characters a *varchar* can contain; or specify the type of integer (small int, big int ,etc.) to not waste memory.
Features	Firestore supports batch writes, transactions, caching and other useful features.	MySql supports most of Firestore features (excluding caching).
Cost	Firestore has two plans. The blaze plan is *pay as you go*. Which means that costs are in proportion to how many read and write operations take place. While for smaller applications this is a great offer, many users complain about its low scalability.	Fundamentally, the costs of maintaining a MySql database are all included in the server host you chose. For smaller applications, these costs are extremely low.
Code	All code for managing Firestore is written client side. This is done by importing the Firestore package in your project and connecting it to your database	All code for managing a MySql database is usually written in php[58]. This means that to access your database, an http request is required.
Security	All firebase services use *security rules*. These rules are written in a custom DS language. Incredibly simple to write, they provide an effective security level, as they are completely checked server side.	Other than all the good practices to avoid SQL injections (as to use named parameters in php), other layers of security can be added in your php file.

[56] See SQL-like joins in Cloud Firestore #1: Getting the sample app up and running
[57] See Supported data types | Firestore | Firebase
[58] Php is used to perform the query. The query itself is, of course, written in SQL.

Table 3, Comparison between Firestore and MySql

Both databases are incredibly powerful in their own ways. While at a first glance Firestore is more palatable, in particular for a developer with little experience like me, its queries have too many limitations. Better Than Yesterday, requires a pretty complex logic on server side. Just the absence of relations between collections (i.e. tables) and the difficulties associated with joins and geo-queries are valid reasons to prefer MySql, for my specific project.

Managing a MySql database meant learning a new programming language and, in general, is far less accessible, since to optimize queries and database structure great experience is required. Considering the little time I have had since I started with this new database, there are certainly many adjustments, corrections and improvements to make, but I already find the new application, with the new database, faster and cleaner than the previous one.

3.4 Database structure

All the data on the database is organized in seven tables:

- *users_profiles:* Here is stored all the data regarding the user. Its `primary key` is a `bigint` called `userId`. It has also a field called `firebaseUserId`, used to connect the credentials and token of the Firebase Authentication Service, to the user in the database. Other than when the user just opened the app, this second id is rarely used, as the userId is an int and a `primary key`, both characteristics which make it far more efficient for read operations. Mysql does not support arrays, so the best solution to represent the filters selected by the user, is to create a column of type `boolean` for each filter; if the value is 1, then the filter was selected. To support geo-queries, one column is the `GeoPoint` of type `geometry`, to which is associated a `spatial index`.
- *users_followers*: This table is composed of just two rows plus the `primary key.` These rows contain the `followingUserId` and the `followerUserId`, and they are both `foreign keys` connected to the userId of the users_profile table.
- *posts:* In this table is stored all the data related to posts. The `primary key` is an `auto increment bigint` called `postId`. The author of the post is stored as a `foreign key` connected to the users__profile table. The logic behind filters and the `GeoPoint` is equal to the one used in the users_profile table. To field `description` is of type `text`, the only case in the whole database, as description's length is virtually limitless.
- *posts_photos:* This table contains three columns. The `primary key`, the link to the image stored in Firebase Storage under the name of `photo`, and a `foreign key` to the posts table called `postId`.
- *posts_status*: This table is used to store information about likes, participations and shared posts. More information in the following paragraph.

- *comments*: Every comment is stored here. Other than the usual `primary key`, a comment is characterized by the author's `userId`, the `postId`, the date of publication, and, of course, the `body`.
- *comments_status*: This table is used to store information about likes for comments.

This structure was thought to be scalable and efficient. Although the posts and users_profile tables have respectively 33 and 31 columns (not excessive, but small neither), most of these are simple booleans[59] which are incredibly light to store, and thus do not invalidate performances.

Not all `indexes` have been created, and there is still plenty of space for optimization, but the structure, in its fundamentals, was conceived after extensive research on mysql database structures for social networks.

3.5 Likes, participations and shared posts

To organize all the logic behind likes, participations and shares, one table called *posts_status,* was created. These are its fields:

1. *codeInt:* This is the primary key. It is of type bigint auto-increment.
2. *userId:* This field, of type bigint, represents the userId of the user who has liked, shared or requested the participation to the post.
3. *postId:* This is the only foreign key of the table. It is linked to the primary key postId in the table *posts*.
4. *isLiked*: This boolean value is equal to 1 if the post is currently liked by the user, 0 otherwise. It is equal to 0 by default.
5. *isShared:* This boolean value is equal to 1 if the post is currently shared by the user, 0 otherwise. It is equal to 0 by default.
6. *participationStatus:* This field is of type tinyint(2), so it assumes three values. It is equal to 0 by default, it is equal to 1 if the user is waiting to be accepted, and 2 if the user was accepted and thus now participates. When the user clicks on the participate button, the participationStatus changes to 1.; if he or she clicks again the participationStatus changes back to 0.

Every time posts are retrieved, the query presents at least one left join to the posts_status table. On client side this means that every post object is also described by three variables from posts_status: *isLiked*, *isShared, participationStatus*.

The table *comments_status,* fundamentally works in the same way, but for comments[60].

3.6 Notifications and Firebase Cloud functions

[59] Actually, boolean type does not exist in MySql. It is technically correct to talk about tinyint(1).
[60] Given that a comment can only be liked, the table lacks the fields isShared and participationStatus.

Notifications, when not abused, are an excellent method to let users know about an event, even when the app is closed or in background. To implement notification in a Flutter app, two services are often used: Firebase Cloud Functions[61] and Flutter Local Notification[62].

Firebase Cloud Functions is one of the many Firebase Services and is closely connected to the Google Cloud Console. Cloud Functions are simply functions which are executed on a server. The language used is Javascript or typescript and they can be written directly from your Flutter project, by connecting it to the Firebase Project. They are used for many different purposes, one of which is notification.

In your `index.js` file, you first define the followings:

```
const functions = require('firebase-functions');

const admin = require('firebase-admin');

admin.initializeApp();
```

The admin library is just "a set of server libraries that lets you interact with Firebase from privileged environments"[63].

Next, a function of type `onCall`[64] is defined in the region which is more convenient[65]. This function takes two parameters: `data` and `context`. The first is a map of parameters passed by the client.

```
exports.fcm = functions

    .region('europe-west1')

    .https

    .onCall((data, context) => {

        const username = data.username;

        const title = data.title;

        const token = data.token;

        const receiverId = data.receiverId;

        const payload = {

            token: token,

            notification: {
```

[61] See Cloud Functions for Firebase
[62] See Flutter_local_notifications | Flutter Package
[63] See Add the Firebase Admin SDK to your server
[64] See Call functions from your app | Cloud Functions for Firebase
[65] To see all the regions available Cloud Functions locations

```
            title: `${username} ha richiesto
                    la partecipazione a ${title}`,
            body: 'Visualizza le tue richieste',
            },
            data: {
                receiverId: receiverId,
            },
        };
    admin.messaging().send(payload).then((response) => {
        return {success: true};
    }).catch((error) => {
        return {error: error.code};
    });
});
```

After the initialization of all the necessary constants, a `payload` is defined. It contains all the necessary information for a notification. The `receiverId` is the `userId` of the user to which this notification is destined. Through the method admin.messaging().send(payload) the notification is sent.

The token is unique to every device, and it is retrieved and uploaded to the database when the user accesses the application for the first time, or after he or she has deleted the cache of the application. The code to retrieve the token is something like this:

```
_firebaseMessaging.onTokenRefresh.listen((token) {
  //this stream makes this code execute only when the token is refreshed
  //setToken(String token) uploads the token on the database
  Provider.of<Users>(context, listen: false).setToken(token);
});
```

This code is located in the *initState()* of the *PagesController* class.

If the user receives the notification when the app is closed or in background, Firebase automatically displays the notification to the user. If, instead, the user is currently online,

firebase default behavior is to silent the notification. Consequently, another service which can replace Firebase is needed. This job is perfectly executed by the *Flutter Local Notification* package.

The way this is achieved is quite complex, so I invite you to read the commented code[66].

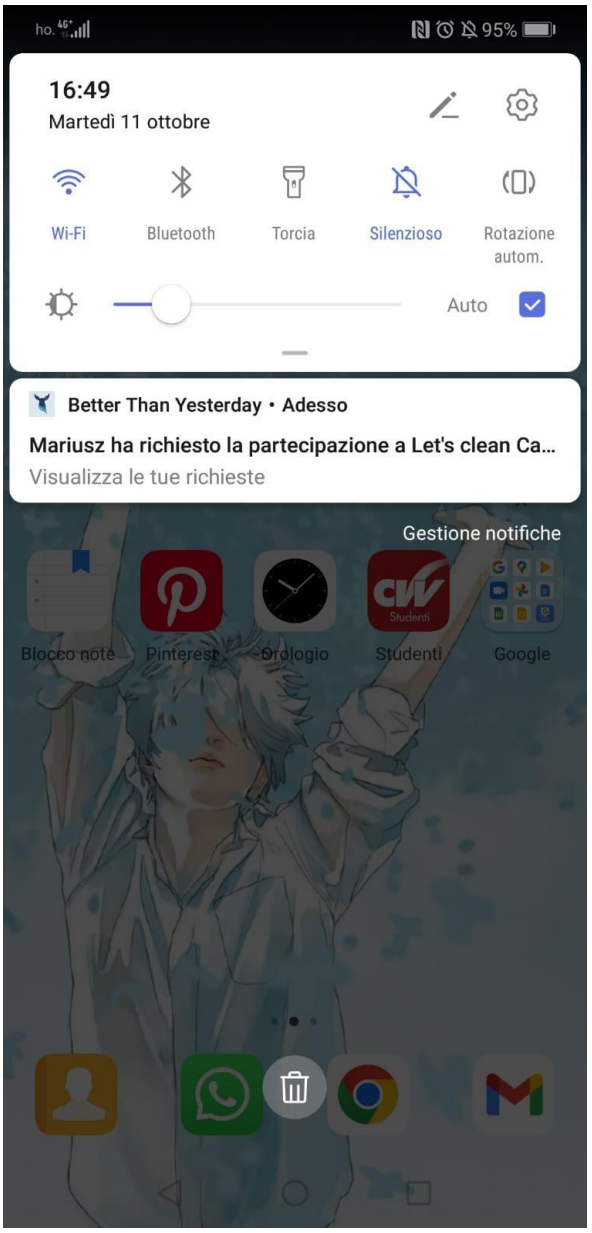

Figure 13, the user 'Mariusz' requested permission to participate in 'Let's clean Ca…'.

3.7 Best Web Hosting Providers

[66] See the *LocalNotificationService* class, under the *services* folder. The initialization, instead, is written in the *initState()* of the *PagesController* class

At least at first, maintaining a server alone would result in excessive and useless costs, as the user base would be pretty insignificant. Therefore, a hosting provider to maintain the database on a server is required.

There are plenty of valid solutions, among both Italian and international services. For now[67], given that the open beta is still not available, I am using a free host called 000WebHost[68]. Before releasing the open beta version[69], this is just a temporary solution to avoid any unneccesary payment.

My choice for hosting service fell on *Hostinger*. On many specialized sites Hostinger is rated among the best (if not the best) overall[70]. The *Business Web Hosting*[71] plan offers:

1. 200 Gb of SSD, a lot more than other competitors
2. Unlimited Bandwidth and Databases
3. 99.9% Uptime
4. Free domain
5. Daily backups
6. Optimized speed for accessing databases
7. Unlimited Cronjobs

According to WebsiteSetup[72], the average time for loading a page was about *300ms*.

4 Entrepreneurial aspects and the future

4.1 Costs and fundings

Better Than Yesterday is a non-profit project which, for the moment, has no ads nor need for any kind of payments to be used. Considering these important restrictions, it is important to minimize costs and search for funding.

I, the only developer until now, have no intention of receiving money for my work, and so, the costs are derived from the services used. These costs are:

- *Firebase Cloud Functions*: They cost $0.40/million, but up to 2 million every month they are free[73]. As for the moment, cloud functions are used for notifications only,

[67] This is written on the 9 October 2022.
[68] See 000WebHost
[69] In a few weeks, probably in the first days of November.
[70] A few example: Top 10 Best Web Hosting Providers of 2022 – In-Depth Reviews, The Best Web Hosting Services for 2022 | PCMag
[71] See Web Hosting | With 1-Click Installer, 24/7 Support, and More. I will not choose this offer from the start, ,as it is for medium-sized projects. However, if and when, this threshold is reached, I will probably switch to this offer.
[72] See Hostinger Review: Is This European Web Host Any Good?.
[73] Actually, there are other pricings, but considering the functions used, they are quite irrelevant. For further information see Cloud Functions pricing

which means that their total pricing per month is 0, at least until a consistent user base is formed.

- *Cloud storage:* Two prices have to be considered: $0.026/GB for the storage, and $0.05/10k for uploading operations. Even in this case, costs are irrelevant, until a certain point is reached.
- *Web hosting provider*: As discussed earlier[74], for the first period, the web hosting service will be the *Premium Shared Hosting* offer by *Hostinger*. It costs $7 per month.
- *Google maps API*: The only API used, actually, is the *Places API,* to find places. The geocoding part (coordinates from place), is realized using a free third party package. The method used is called *Autocomplete - Per request*, which costs $2.27 per 1000 calls[75]. However every month, $200 from the total money spent with the Google Maps api, are subtracted. Which means that to actually pay something for the usage of this api, there should be about one hundred thousand requests. Furthermore, in the near future, I will probably request the additional discounts offered to non-profit projects by Google[76]. If accepted, costs for this API will fall even more.

Firebase Authentication Service and Firebase Cloud Messaging are completely free. Moreover, Google Cloud Console offers the first $300 spent. In future, to realize messaging functionality, I will probably use *Realtime Database*, another Firebase service. But for now, the costs for the services are just the ones listed above.

Another category of costs is given by “external” and optional things like ads or contacting a professional developer to help me to some extent. While these are certainly possibilities to keep in mind, they remain, for now, just ideas for the future.

To be able to realize these and similar ideas, and to pay for the services above in case the application reaches a certain level of success, and thus they begin to represent a significant expense, some *funding* would prove necessary.

In order to achieve this goal, I have already contacted a famous entrepreneur in my city, who is known to often dedicate himself to volunteering. He has given me advice and has already suggested various funds, associations and companies, which annually organize calls for which this project would be eligible.

Before trying any of these, however, it is necessary to test the application extensively, to ensure that it turns out to be an attractive product.

4.2 Alpha and beta testing

To test the application’s features, and correct the most obvious bugs, most software goes through a period of alpha and beta testing. The original alpha version for BTY was released on 5 September 2022. Alpha testers were just a small group of four friends, who simpy needed to test if core functionalities worked, and test the app performances.

---

[74] See note n. 71
[75] See Places API Usage and Billing | Google Developers
[76] See Activate Google Maps Platform credits - Google for Nonprofits Help

While I was preparing for the beta, I decided to change database, and so, after some weeks of work, on 2 October, I released a new alpha which used MySql instead of Firestore. Other than a new back-end, this "second alpha" introduced certain new features. This time, alpha testers were a group of seven friends. For the moment, this alpha did not produce bugs, and while performance still has a lot of space for optimization, this second version grants a far better scalability. Advanced optimization problems will be treated starting from the beta, when more users have access to the app, and similar matters will be easier to analyze.

The beta should start in the second half of November, and it will probably involve hundreds of people. Among them, actual volunteering realities are counted. Through the beta, possibilities, major problems and the appreciation rate of the project will be finally estimated more concretely.

While the code is written and planned for IOS users too, for now, I just tested it on Android. This was done for two reasons:

1. *Simplicity*: My family, my close friends and myself all have Android devices, and thus a fast testing was not possible. Moreover, to actual deploy the app for IOS, some modifications have to be done using Xcode, which is just available for Mac users[77].
2. *Progressivity:* Many big projects are first tried on one operating system, and then on the other. While it is true that Flutter was invented to break this hard division, some functionalities and bugs are still platform specific. It is preferable, also considering the limited workforce available, to focus first on one platform, and then, when assured that major problems are solved, pass to the other one.

4.3 Effective Release

Better Than Yesterday should be released in the play store during March 2023. The release on the app store is still unknown, as there are many variables which are impossible to know at the moment. However, if no particular problems are present, BTY will be available for IOS users not long after March.

Thanks to my acquaintances in the world of city volunteering, I will be able to promote the application in the city with relative ease. Although small, a city already represents a quite heterogeneous sample to be able to carry out market segmentation, and above all, to understand what further needs people in the sector could have.

If I am recognized as a *non-profit* organization, another advantage I could use would be with Google Ads[78]. In this way, BTY would be increasingly suggested to a more suitable audience. Although I still have to start the request, the probability of being accepted is fairly high, as the requirements for Italy are:

[77] To solve this problem, I will probably install a virtual machine.
[78] See Google Ad Grants - Free Google Ads for Nonprofits

1. Organizations must be currently registered with SocialTechno, TechSoup Global's regional arm[79].
2. Organizations must be: (1) associations (associazioni), foundations (fondazioni), and social cooperatives (società cooperativi) registered as non-profit social utility organizations (organizzazioni non lucrative di utilità sociale – ONLUS); (2) international NGOs (organizzazioni non governativa – ONG) registered with the Ministry of Foreign Affairs and International Cooperation; (3) religious organizations (istituzioni religiose); or (4) other organizations operating on a non-profit basis for the public benefit[80].

Of course, this means that I should collaborate with an organization or fund one on my own. In both cases, the eligibility requirements are compatible with the idea which is the basis of the project.

4.4 What to do now

Below, is the list of *to does* for the next few months[81]. This is just something to help the reader visualize which points of the project are probably going to be realized.

In all likelihood, some points on this list will change: some will be deleted, others added, others modified, and others slipped. However the general form will not change, as most of the following points are mandatory for the success of the project.

1. Improve performances by modifying some queries on the backend.
2. Implement Firebase App Check.
3. Secure Google Maps API keys.
4. Implement authentication with Google Account.
5. Make the analytical expression of *trend points* more complex.
6. Apply for non-profits organization at Google (for Google Ads and Google Maps).
7. Buy the web hosting service.
8. Release a closed-beta with a considerable number of users.
9. Implement Messages in App (probably with Realtime Database).
10. Implement other notifications for other events.
11. Make the project available for IOS users.
12. Obtain license for publishing in app store.
13. Apply to calls for funds
14. Add the English language

[79] See Social Techno
[80] See Eligibility guidelines - Google for Nonprofits Help
[81] Consider approximately March 2023 as deadline.

Conclusion

Better Than Yesterday is a project which started just two months ago, and considering the limited workforce and limited resources available, there is still a long way to go. About 95% of the beta version of the code has been written, and will be released as soon as possible.

Software is never finished, and the project will continue to grow in complexity. My little experience does not allow me to make concrete estimates on the potential success of this application. Nor if it turns out to be capable of managing a large number of users. There are many unknowns, but the starting point is solid.

There is an incredible number of people who could benefit from this project, both volunteers and, of course, people in need.