UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

System for collection, management and processing movement data

by

Athanasios Vakouftsis

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Spiros Skiadopoulos
Professor

Athens, Month 2022

System for collection, management and processing movement data

Athanasios Vakouftsis

MSc. Thesis, MSc. Programme in Data Science

University of the Peloponnese & NCSR “Democritos”, Month 2022

Copyright © 2022 Athanasios Vakouftsis. All Rights Reserved.

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

System for collection, management and processing movement data

by

Athanasios Vakouftsis

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Spiros Skiadopoulos
Professor

Approved by the examination committee on Month, 2022.

(Signature)	(Signature)	(Signature)
.....	.....	.....
Name1 Surname1	Name2 Surname2	Name3 Surname3
Committee1 title	Committee2 title	Committee3 title

Athens, Month 2022

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Declaration of Authorship

(1) I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

(2) I confirm that this thesis presented for the degree of Bachelor of Science in Informatics and Telecommunications, has

(i) been composed entirely by myself

(ii) been solely the result of my own work

(iii) not been submitted for any other degree or professional qualification

(3) I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

(Signature)

.....

Athanasios Vakouftsis

Athens, Month 2022

Acknowledgments

With the completion of this thesis, in the context of the postgraduate program that i attended at the National Centre for Scientific Research Demokritos in collaboration with the University of Peloponnese (MSc Data Science), as a sign of gratitude for his valuable help, i would like to thank my supervising professor Mr. Spiro Skiadopoulo who entrusted me this subject for this thesis. I would also like to warmly thank my family for the psychological support they provided, as well as my friends, the fellow students for their suggestions and ideas, which helped in the implementation of this dissertation and especially my friends from the undergraduate degree for their on point advices. Finally i would like to thank the colleagues who create free (open source) software as some things like getting data from the sensors wouldn't have been possible in this small amount of time of creating my thesis's application.

To my family.

Περίληψη

Σκοπός αυτής της εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός συστήματος για την συλλογή, την διαχείριση και επεξεργασία δεδομένων μετακίνησης. Αρχικά θα υλοποιηθεί εφαρμογή για κινητά τηλέφωνα με την οποία ο χρήστης θα μπορεί να συνδεθεί στο σύστημα και να διαμοιράσει τα δεδομένα των μετακινήσεών του. Τα δεδομένα αυτά θα αποθηκεύονται σε ένα κεντρικό σύστημα το οποίο θα τα επεξεργάζεται και θα τα διαχειρίζεται. Η εφαρμογή λειτουργεί σε φορητές συσκευές που έχουν λειτουργικό σύστημα Android και IOS. Ακόμα γίνεται αναφορά στα εργαλεία που υπάρχουν στο διαδίκτυο για ανάπτυξη εφαρμογών σε διαφορετικά λειτουργικά συστήματα ταυτόχρονα, στα πλεονεκτήματα και στα μειονεκτήματά τους καθώς και για την επιλογή του εργαλείου που αναπτύχθηκε η εφαρμογή. Τέλος το μεγαλύτερο μέρος της αναφοράς αποτελείται από μία εκτενής ανάλυση της ανάπτυξης της εφαρμογής καθώς και των διαφόρων προβλημάτων που προέκυψαν και το πως αυτά αντιμετωπίστηκαν κατά την δημιουργία της εφαρμογής.

Abstract

The aim of this thesis is to design and implement a system for collecting, managing and processing movement data. Initially, a mobile application will be implemented on which the user will be able to connect to the system and share the data of the movement. The data will be stored in a central system that will process and manage them. The application works on portable devices that have Android and IOS operating systems. Furthermore, reference is still made to the tools available on the internet for mobile application development in different operating systems simultaneously, their advantages and disadvantages as well as for the selection of the tool in which the application was developed. Finally the biggest part of the report consists of an extensive analysis of the development of the application and as well as the various problems that arose and how they were addressed when creating the application.

Contents

List of Tables

List of Figures

List of Abbreviations

IDE	Integrated Development Environment
UI	User Interface
PC	Personal Computer
SDK	Software Development Kit

Chapter 1

Introduction

In the modern age where the demands of everyday life and the needs of a user are changing, the need to create applications that will cover all tastes is imperative. Many applications that register the route followed by a user make their appearance daily in the Play Store and the App Store, however it is necessary to create such applications where their personal data and users will be respected and advertisements will be absent.

1.1 Purpose - Problem description

The purpose of this thesis was divided into two goals, the first one was to create a mobile application for Android and IOS operating systems that tracks with accuracy the movement of the user including the sensors of the device and the second goal was to save and send the data that are created from the app into a database in our university. For better understanding of what we had to do was to create an application that the user must connect with some credentials and select a screen where we the user can find a map just like a navigation application. In the screen the user can find himself only by pressing a button and periodically the map will save the current coordinates. Also in this application it must be included a screen where the user can find a screen in which it will be displayed a number of sensors and their data that the user can see which of these sensors are available on the device. Ending, the application must work in the background if the user also wants

to have another application open in order for our app to collect data.

1.2 Thesis structure

Starting with Chapter 2, a reference is made to all the Software Development Kits and the Frameworks that are available for building applications on both Android and IOS, why and on SDK's which we selected to create a demo app. In Chapter 3 the reasons why we chose the Flutter SDK, the advantages and disadvantages of Flutter and Xamarin. In chapter 4 we analyze how the Flutter is built, what are the widgets, how does it rendering the UI and how is connected to native code. In Chapter 5 we take a quick look at the tools we chose and their competitors. In Chapter 6 we have a walkthrough of the app, is like a user's manual of the application. We analyze what are the functionalities of every screen and how the interact with every UI object. Chapter 7 is the biggest section of this thesis because we have a complete breakdown of how we built the app, the packages we used and the various functionalities we created. Finally, in Chapter 8 we have the conclusion and the future extensions of the application.

Chapter 2

Cross-platform mobile applications development

In our days there are many Integrated development environments (IDE) for developing mobile applications for Android and Apple devices simultaneously. The supported languages of each platform is for Android is Java and for IOS is Swift. Developing both for Android and IOS on the same time at native languages can be very challenging, with the help of many existing SDK's the developer can code one script file that can be executed in both platforms simultaneously. Starting with the wide variety of software development kits (SDK) and their selection of programming languages (see Table 2.1).

SDK	Programming Languages
Apache Cordova (*https://cordova.apache.org/*)	HTML, CSS, Javascript
Ionic (`https://ionicframework.com/`)	Javascript, Angular, React, Vue
NativeScript (`https://nativescript.org/`)	Javascript, Typescript, Angular, Vue, React, Svelte, Capacitor, HTML
React Native (`https://reactnative.dev/`)	Javascript
Flutter (`https://flutter.dev/`)	Dart
Xamarin (`https://dotnet.microsoft.com/en-us/apps/xamarin`)	.Net Framework, C#
Felgo (`https://felgo.com/`)	Javascript, C++, QML
Rho Mobile (`https://tau-platform.com/en/products/rhomobile/`)	Ruby, HTML, CSS, Javascript
Sencha (`https://www.sencha.com/`)	ExtJS
Framework7 (`https://framework7.io/`)	HTML, CSS, Javascript, Vue, React, Svelte
Jasonette (`https://jasonette.com/`)	Javascript, Json

Table 2.1: Software development kits with their programming languages

In this thesis we considered to compare the Ionic, Xamarin, Cordova and Flutter SDKs to make a demo application to each one of them as also they were the most popular choices among cross-platform tools for development. We selected to use Windows 10 to install every SDK.

2.1 General information and installation of SDKs

2.1.1 Cordova

Starting from Cordova, it is a mobile application development framework and supports Android, IOS and Web applications created by Nitobi (acquired by Adobe). It was released in 2009 and there aren't many developers that use Cordova for mobile applications development. It uses HTML, CSS, Javascript and the UI and the functionality. To work with Cordova first we had to install the latest Java SDK (`https://www.oracle.com/java/technologies/downloads/#jdk18-windows`) and Android Studio (`https://developer.android.com/studio`) in order to get the command line tools, the android emulator and the gradle. We installed nodejs from `https://nodejs.org/en/` the version 14.17.6 LTS. After that, we installed Git (`https://git-scm.com/`) because it was needed for the following step (the following commands are written on git). To configure the Android SDK we followed the steps on the documentation `https://cordova.apache.org/docs/en/10.x/guide/platforms/android/index.html`. To create an application we executed the installed launcher of Git and entered the command on Git: cordova create "name of the app". The next step is to add a platform which in our case is android/ios and we enter the command cordova platform add android/cordova platform add ios.

2.1.2 Ionic

For Ionic installation it was needed Java SDK, Android Studio for its emulator, nodejs and Git. In command line we entered the command npm install -g @ionic/cli to install ionic. After the download and installation is complete we can start working on the mobile app.

2.1.3 Xamarin

For Xamarin the installation is easier and more straight forward process. I installed visual studio 2019 community and I installed the package Mobile development with .NET.

2.1.4 Flutter

For Flutter the installation was easy. We downloaded the flutter.zip file from here `https://flutter.dev/docs/get-started/install/windows` and followed the simple steps that were documented.

2.2 Developing the demo application

To understand better the capabilities of each platform we decided to make the same app across these four platforms. We chose to built a very simple application on every of these SDKs. We call this application Demo. Demo has a button each time we press it an alert window appears showing the total number of button clicks within the correct Demo execution.

2.2.1 Cordova

Starting with Cordova, we used Visual Studio Code as the IDE. We created a blank page and starting programming. After that we created a button in the homepage which was an HTML file. In order to build the app to manage Cordova, all the commands must be entered in Git window inside the project folder. After completing all the changes in Javascript and HTML we entered the command cordova build. For sending the app to a mobile device the Cordova gives 2 options, by sending the application to run in an Android/IOS device with the command cordova run android/ios or (for android) by building the apk that was made with the command cordova build. The second way of testing the app is by testing it on android emulator with the command cordova emulate android (for the android application we used the Android Studio built-in emulator) and for IOS the testing on emulator must be

done on Mac operating system. Figure 2.1 shows the implementation of the Demo application using Cordova.

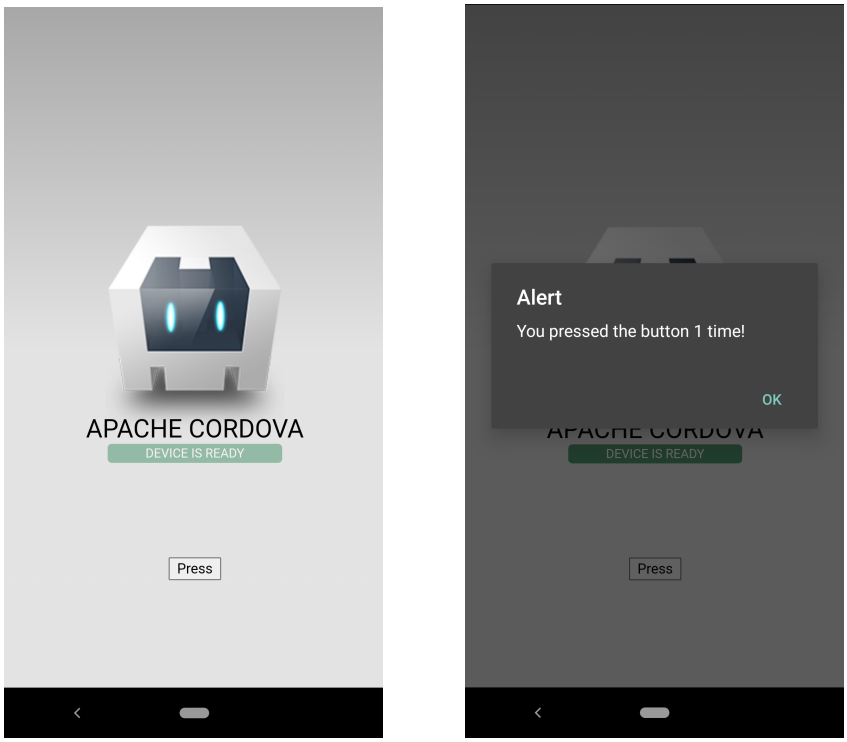

Figure 2.1: Demo build using Cordova

2.2.2 Ionic Framework

Ionic is a complete open-source SDK for hybrid mobile app development and supports Android,IOS and Web applications. We also used Visual Studio Code as our IDE. Ionic uses multiple frameworks for building applications like Angular, React, Vue and vanilla Javascript. We used Angular for building the application which is a TypeScript based free and open-source web application framework. We created a blank project so the home page was an empty HTML file. The function with the button is inside a typescript file an then the function is called in the HTML page with an ion button. The ionic application can be deployed on android/ios device with the command ionic capacitor run android/ios. To see all the connected devices and their IDs on personal computer (pc) we enter the command ionic capacitor run android –list and to select a device we enter ionic capacitor run android/ios –target=”device-id”. Ionic has a deployment local server that shows in webview

how the application would look like on a selected device, by pressing F12 we can select or even add a custom device (the webview isn't emulating the application, just how it looks from webview to a smaller screen). Ionic can also deploy applications on the same emulator as Apache Cordova with the command ionic capacitor emulate android/ios (for android app we used the Android Studio built-in emulator). This is how it looks on an Android 10 device (see figure 2.2).

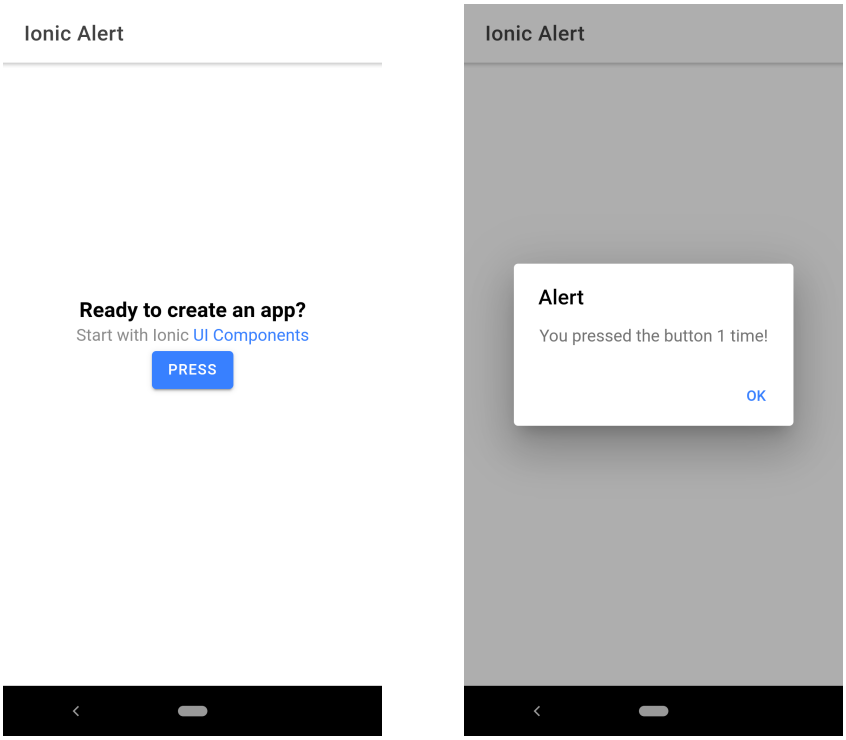

Figure 2.2: Ionic UI

2.2.3 Xamarin

Xamarin is an open-source platform for building modern and performant applications for Android , IOS and Windows with .NET. For development we used Visual Studio 2019 as it is the best IDE for Xamarin development. The language we used was C# and once again we created a blank project. On the same folder Visual Studio creates 3 projects, the main project in which we coded the function and the button and 2 other projects for having specific code on every mobile device (Android project, IOS project). We selected the main project as my startup project and then we selected the .xaml In which it was created the homepage of the app, we added a

button and then we created a function on the .xaml.cs file. For deployment Visual Studio supports deployment on Android and IOS devices or by building and .apk file for testing (only for Android devices). Visual Studio uses the emulator from Android Studio. From here we can select the device or the emulator we want to deploy the application for testing (see figure 2.3).

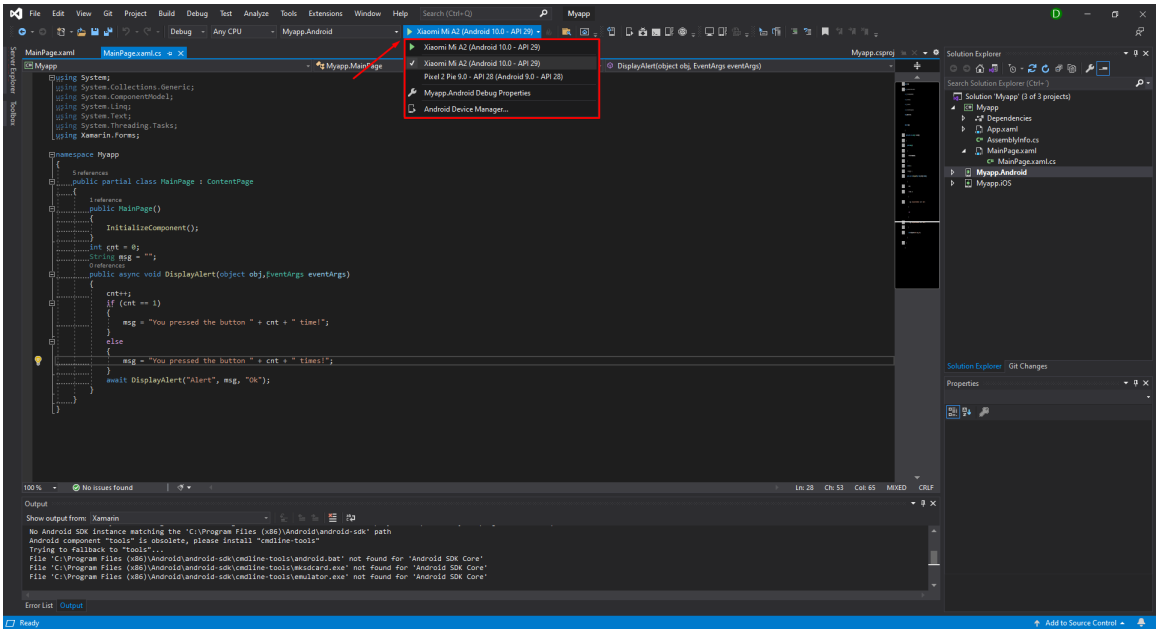

Figure 2.3: Visual Studio interface

This is how it looks on an Android 10 device (see figure 2.4).

2.2.4 Flutter

Flutter is an open-source UI software development kit created by Google and supports Android, IOS, Mac, Windows, Google Fuchsia and Web applications. For development I used Android Studio as my IDE in which we integrated it by following the steps from the documentation of Flutter (`https://flutter.dev/docs/get-started/install/windows`). We used the programming language Dart for creating the application as it is the default programming language of Flutter. We created a blank project and on the same file I created a function for the alert and we called it within a button in the homepage. Flutter supports deployment by building an .apk file(for the Android OS) or by connecting an Android/IOS device. For emulator Android Studio uses a built-in emulator which is used on most of the

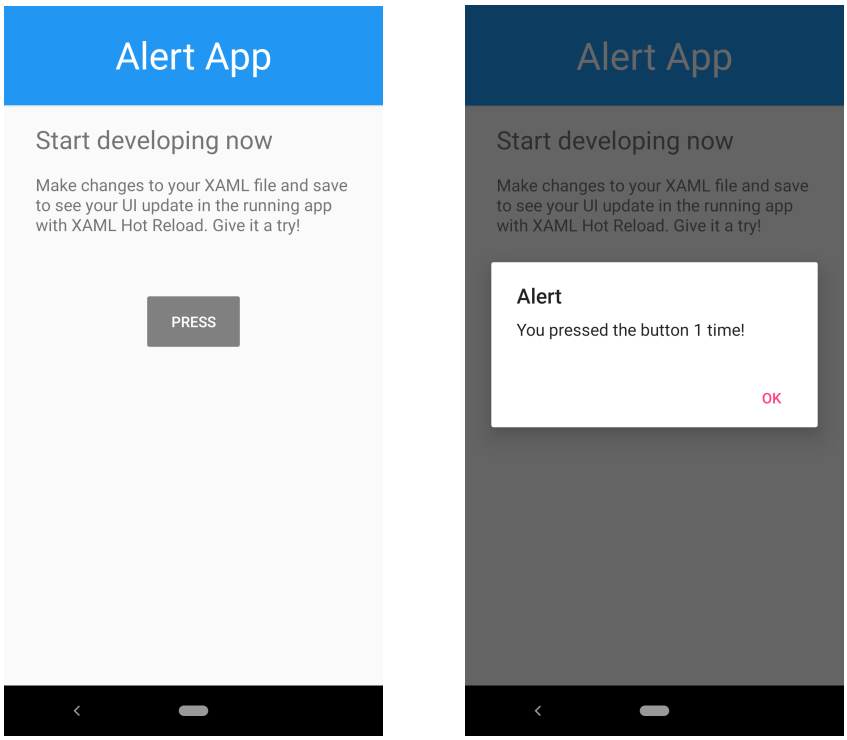

Figure 2.4: Xamarin UI

platforms for testing Android applications. From here we can select the device or the emulator we want to deploy the application for testing (see figure 2.5).

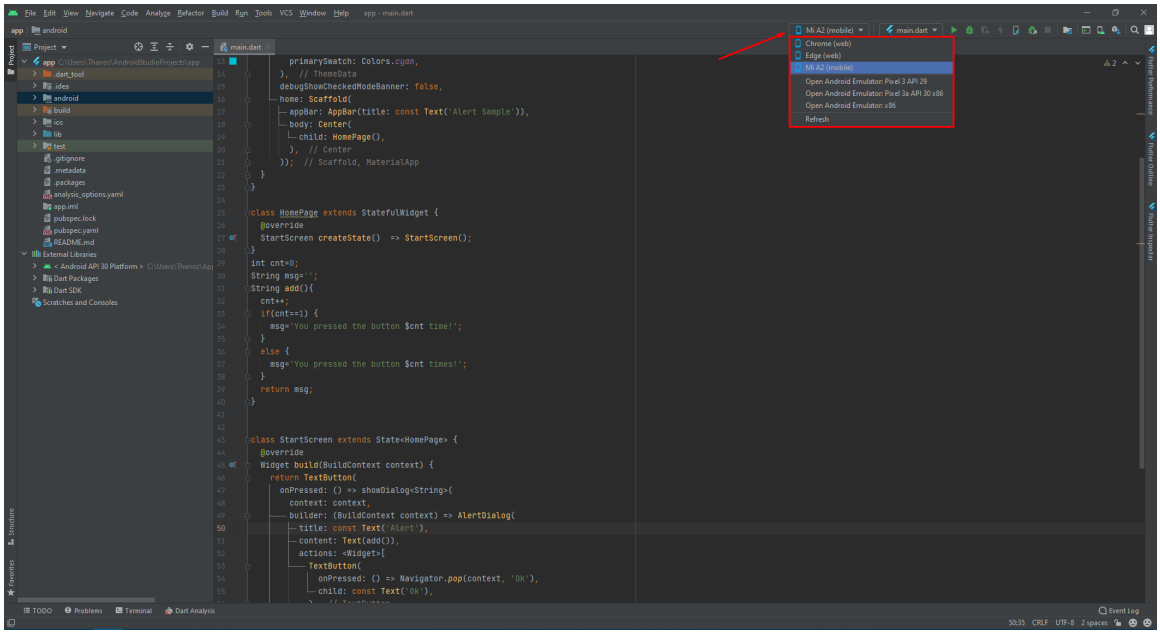

Figure 2.5: Flutter interface

This is how it looks on an Android 10 device (see figure 2.6).

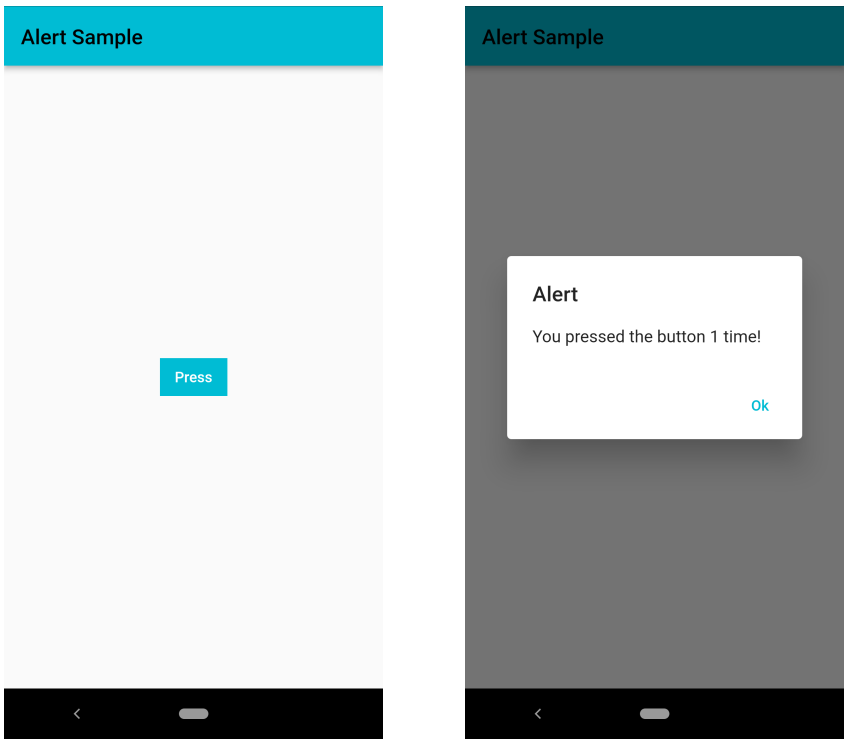

Figure 2.6: Flutter UI

Chapter 3

Chosen framework

As we developed a simple application (app) in each one of the 4 frameworks we realised that for the project we couldn't continue with a web framework (Cordova or Ionic) for the simple reason that they don't have good support for native libraries and it would be much harder to develop functions for getting data from the device sensors or for using what every device manufacturer can provide throught the Operating System (OS). Another reason was the lack of translation to native code through an engine. On both Xamarin and Flutter the frameworks provide an intermediate layer from writing code from Dart (on Flutter) and from C# (on Xamarin) translating both the written scripts to native code for each os, Java for android and swift for IOS. This results of course as the app gets bigger, optimal execution even on low end devices when writing web scripts will be much slower as the app gets more complicated functions. Lastly both Flutter and Xamarin as they translate the written scripts on native code it can have provide native User Interface (ui) for both operating systems just by calling the already built functions on libraries. So it all come down to one of Flutter and Xamarin. Both have excellent documentation for building apps and both have custom engines as intermediate layer both have its pros and cons to consider before proceeding. A few can be found on these tables (see tables 3.1, 3.2).

Advantages	
Flutter	Xamarin
Easy to learn	Complete ecosystem to build many applications due to C#, .Net
One file contains UI and functionality	Specific files for in depth customization
Provides hot reaload	Provides hot reaload
High performance	High performance
Free to use	Free to use

Table 3.1: Advantages of Flutter and Xamarin

Disadvantages	
Flutter	Xamarin
Lack of third-party libraries	Slow updates of the framework
Flawed iOS support	Heavy graphics due to must have platform specific code
Platform specific optimizations must be done sometimes on each platform	Platform specific optimizations must be done sometimes on each platform
Large app size	Large app size

Table 3.2: Disadvantages of Flutter and Xamarin

It would be easier for us to create the whole app on Xamarin as we already knew C# from the bachelor thesis, but we choose to learn a new language which is Dart(it has some common things from Java and Javascript). As Xamarin is an older cross platform framework and more mature which is a benefit on writing software, and as we are am an early adopters (don't know why) we choose Flutter as it has a more community based approach with loads and loads of projects already built and is faster to build a UI as it uses widgets with dart scripts on one file compared to Xamarin which requires one .xaml file for UI and one file for writing the C# code. Flutter also uses packages to use as libraries, in which there are many already built code for run to be used.

Chapter 4

An inside look of the Flutter

4.1 Architectural layers of Flutter

Flutter is designed as an extensible, layered system. It consists of three layers and no layer has privileged access to the layer below, every part of the framework layer is designed to be optional and replaceable. The second layer contains the Flutter engine which is written in C++ and is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics through Skia, text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain. Flutter applications are packaged in the same way as any other native application. The last layer is the embedder which is written in a language that is appopriate to the underlying operating system, Java and C++ for Android, Objective-C/Objective-C++ for iOS and MacOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code may be the entire content of the application. This is how the layers are structured (see figure 4.1).

4.2 Reactive user interfaces

On the surface, Flutter is a reactive, pseudo-declarative User Interface (UI) framework, in which the developer provides a mapping from application state to

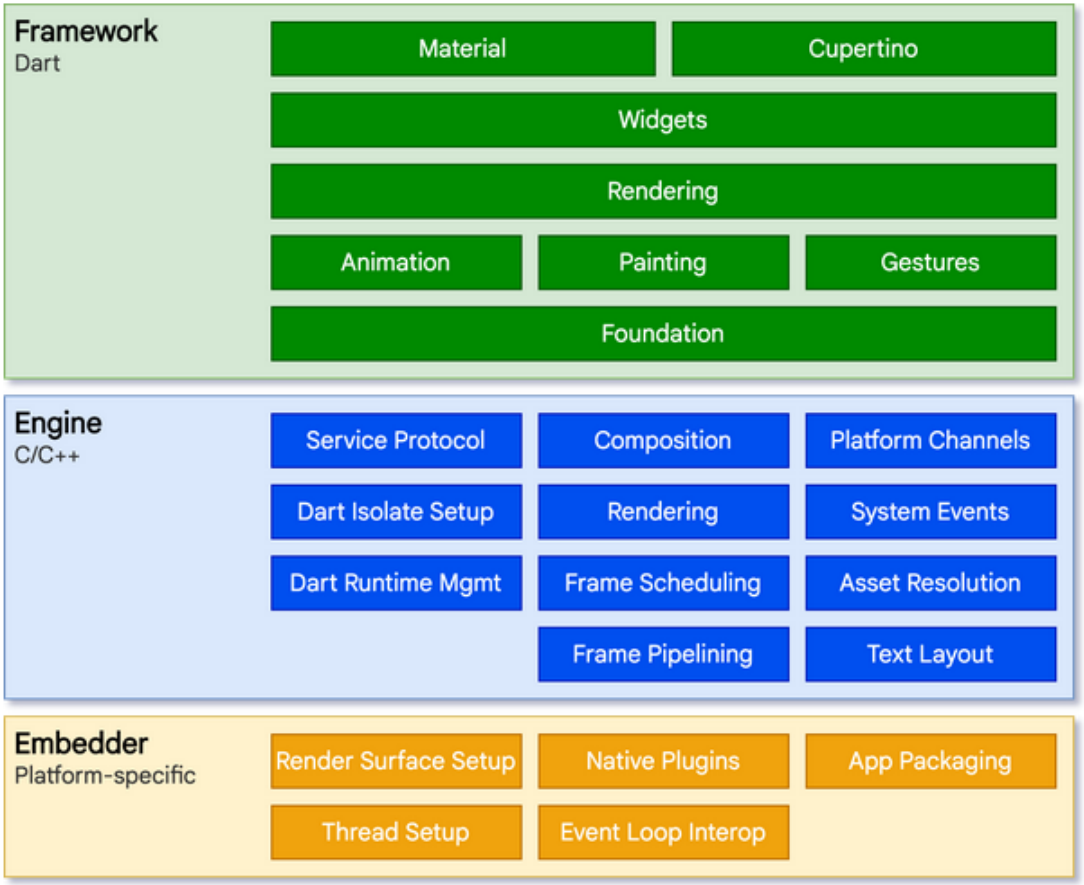

Figure 4.1: Flutter's Layers

interface state, and the framework takes on the task of updating the interface at runtime when the application state changes. Flutter decouples the user interface from its underlying state which results that the user only creates the UI description and the framework takes care of using that configuration to both create and/or update the user interface as appropriate.

4.3 Widgets

Flutter emphasizes widgets as a unit of composition. Widgets are the building blocks of a Flutter app's user interface, and each widget is an immutable declaration of part of the UI. Widgets form a hierarchy based on composition. Each widget nests inside its parent and can receive context from the parent. This structure carries all the way up to the root widget.

4.3.1 Composition

Widgets are typically composed of many other small, single-purpose widgets that combine to produce powerful effects. There is the class hierarchy which is deliberately shallow and broad to maximize the possible number of combinations, focusing on small, composable widgets that each do one thing well. Core features are abstract, with even basic features like padding and alignment being implemented as separate components rather than being built into the core.

4.3.2 Widget State

In Flutter there are two major classes of widgets: stateful and stateless widgets.If widgets doesn't change their state and they don't have any properties that change over time then these widgets are StatelessWidgets, if the unique characteristics of a widget need to change based on user interaction or other factors, that widget is stateful. When that value changes, the widget needs to be rebuilt to update its part of the UI. These widgets subclass StatefulWidgets, and they store mutable state in a separate class that subclasses State. StatefulWidgets don't have a build method; instead, their user interface is built through their State object. Whenever the programmer change a State object the SetState() method must be called to signal the framework to update the UI by calling the State's build method again.

4.3.3 State Management

The state is managed and passed around with a constructor in a widget to initialize its data. The method build() can ensure that any child widget is instantiated with the data it needs. As widget trees get deeper, however, passing state information up and down the tree hierarchy becomes cumbersome. So, a third widget type, InheritedWidget, provides an easy way to grab data from a shared ancestor. InheritedWidget can be used to create a state widget that wraps common ancestor in the widget tree. InheritedWidgets also offer an updateShouldNotify() method, which Flutter calls to determine whether a state change should trigger a rebuild

of child widgets that use it. As applications grow, more advanced state management approaches that reduce the ceremony of creating and using stateful widgets become more attractive. Many Flutter apps use utility packages like provider, which provides a wrapper around InheritedWidget.

4.4 Rendering and layout

Cross-platform frameworks typically work by creating an abstraction layer over the underlying native Android and iOS UI libraries, attempting to smooth out the inconsistencies of each platform representation. App code is often written in an interpreted language like JavaScript, which must in turn interact with the Java-based Android or Objective-C-based IOS system libraries to display UI. All this adds overhead that can be significant, particularly where there is a lot of interaction between the UI and the app logic. The diagram shows the pipeline of how the data flows to the system (see figure 4.2).

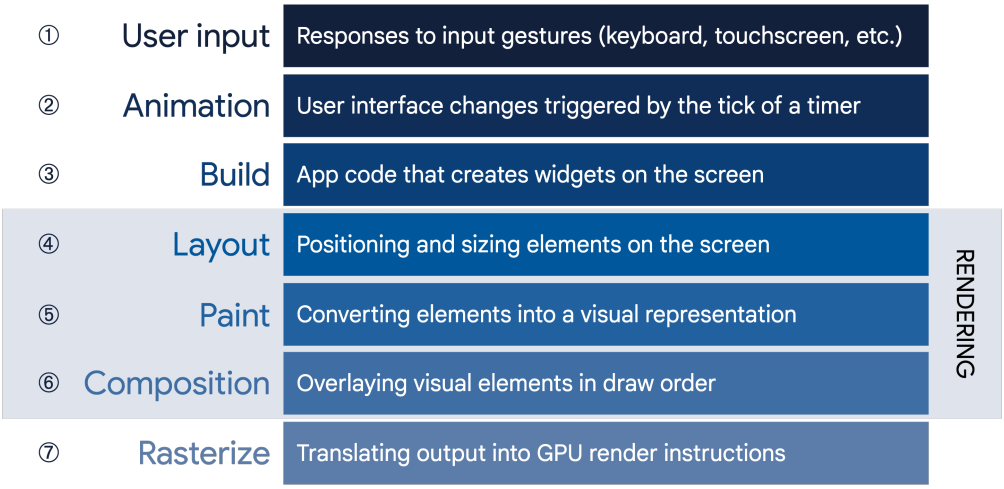

Figure 4.2: Render Pipeline

4.5 Platform Embedding

Flutter user interfaces are built, laid out, composited, and painted by Flutter itself. The mechanism for obtaining the texture and participating in the app lifecycle of the underlying operating system inevitably varies depending on the unique concerns of that platform. The engine is platform-agnostic, presenting a stable Application Binary Interface (ABI) that provides a platform embedder with a way to

set up and use Flutter. The platform embedder is the native OS application that hosts all Flutter content, and acts as the glue between the host operating system and Flutter. When a Flutter app start, the embedder provides the entrypoint, initializes the Flutter engine, obtains threads for UI and rastering, and creates a texture that Flutter can write to. The embedder is also responsible for the app lifecycle, including input gestures (such as mouse, keyboard, touch), window sizing, thread management, and platform messages. Flutter currently includes platform embedders for Android, IOS, Windows, MacOS, and Linux.

4.6 Integrating with other code

Flutter provides a variety of interoperability mechanisms, whether the programmer accessing code or APIs written in a language like Kotlin or Swift, calling a native C-based API, embedding native controls in a Flutter app, or embedding Flutter in an existing application. For mobile and desktop apps, Flutter allows the programmer to call into custom code through a platform channel, which is a mechanism for communicating between the Dart code and the platform-specific code of the host app. By creating a common channel (encapsulating a name and a codec), messages can be send and received between Dart and a platform component written in a language like Kotlin or Swift. Data is serialized from a Dart type like Map into a standard format, and then deserialized into an equivalent representation in Kotlin (such as HashMap) or Swift (such as Dictionary). The diagram shows the exact procedure of integration (see figure 4.3).

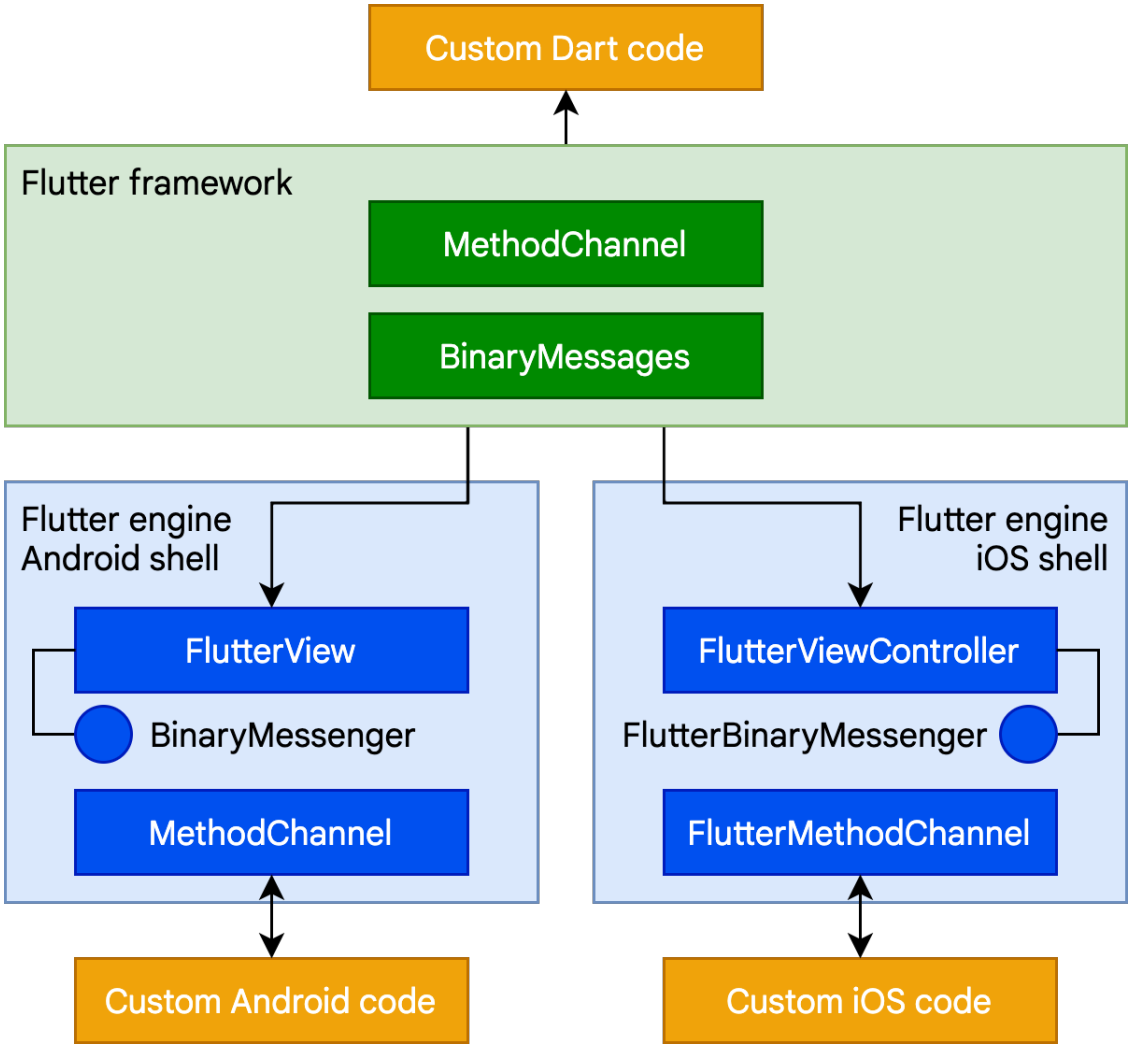

Figure 4.3: Render Pipeline

Chapter 5

Tools

5.1 Choosing the best IDE

When it comes to developing apps for Flutter the are two choices for Integrated Development Environments. The first one is the Visual Studio Code and the Android Studio. The VS code (Visual Studio Code for short) is a more lightweight IDE and comes with all the things a programmer would expect like debugging, breakpoints etc, but it hasn't Android Emulator built-in. The Android Studio on the other hand comes with all the things the VS code has but has Android Emulator built-in with a tool to select the version of Android device the programmer wants to debug the app. We had already used in the past Android Studio to develop some small apps for fun and learning so we selected Android Studio as Flutter app development tool. To integrate flutter we followed the steps that we described on a previous section of the thesis and for Flutter integration on Android Studio we installed two plugins through the Android Studio plugin store, the first one was Dart to be able to write, compile and debug Dart code and the second one was Flutter plugin to connect the installed Framework on the computer with the Android Studio environment. Android Studio combined with Flutter supports 3 types of builds for deployment on a device: debug, profile, release. In more depth: 1) In debug mode, the app is set up for debugging on the physical device, emulator, or simulator. Debug mode for mobile apps mean that:

- Assertions are enabled.
- Service extensions are enabled.
- Compilation is optimized for fast development and run cycles.
- Debugging is enabled, and tools supporting source level debugging can connect to the process.

2) In profile mode, some debugging ability is maintained—enough to profile your app's performance. Profile mode is disabled on the emulator and simulator, because their behavior is not representative of real performance. On mobile, profile mode is similar to release mode, with the following differences:

- Some service extensions, such as the one that enables the performance overlay, are enabled. Tracing is enabled, and tools supporting source-level debugging such as DevTools can connect to the process.
- Tracing is enabled, and tools supporting source-level debugging such as DevTools can connect to the process.

3) In release mode, is used when the programmer wants the maximum optimization and minimal footprint size. For mobile, release mode means (which is not supported on the simulator or emulator) that:

- Assertions are disabled.
- Debugging information is stripped out.
- Debugging is disabled.
- Compilation is optimized for fast startup, fast execution, and small package sizes.
- Service extensions are disabled.

We can see the UI of Android Studio on the image and there are all the available tools provided by the Flutter framework such as performance, outline and inspector tools (see figure 5.1).

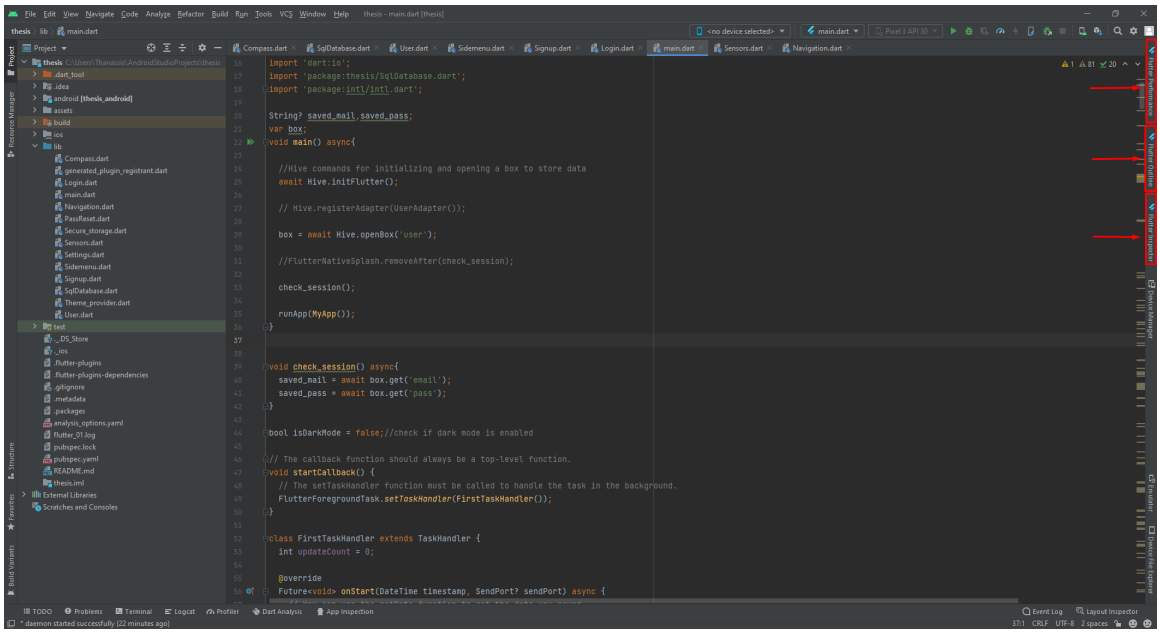

Figure 5.1: Flutter interface

5.2 Virtual Box - VmWare

Because we have installed Windows 10 on the desktop and we don't own an Apple PC we needed a way to test the app also on an iPhone and the only way was to install MacOS on a Virtual Machine. We used Virtual Box for the first try of installing High Sierra, we tried to install Xcode but it didn't support the latest version of it the OS. So we downloaded a newer version of MacOS, the Big Sur, recreating a new Virtual Machine with Virtual Box. After we installed it successfully we downloaded all the essential programs(including Xcode) to be able to build Flutter apps we realized that when we opened the nested virtualization option on Virtual Box, the Virtual Machine wouldn't start no matter what solution we tried (virtualization is necessary for running emulators). Then we tried the program VmWare to a Mac Virtual Machine. After we went through the same procedure as before we finally had a working installation of MacOS we needed.

Chapter 6

Application Tour

6.1 Login Screen

Login screen is the first screen the user sees after he/she opens the app. From there has two options, if user has registered already to the app, he/she enters the e-mail and the password and then presses the button “Login” in order to connect to the app, else he/she must presses the text beneath the “Login” button “Don’t Have an Account? Sign up” to proceed to the registration screen. The e-mail textfield has built-in e-mail in case the user enters an email without the proper form to inform that the e-mail isn’t valid as it is written. The user must be connected to the internet to proceed to the main screen. If the user hasn’t logged out since the last time the app was closed on their device, he/she will be redirected to the main screen without entering their credentials again. On this image we can see the Login screen (see figure 6.1).

6.2 Registration Screen

Registration screen is where a new user can register to the app only by providing username, email and password. Just like the Login screen the email textfield has an email validator in case the user enters a non valid email to warn him. The e-mail must be unique, if the user enter an e-mail that already exist when the button is pressed a small text poping from the bottom will tell the user to enter other e-

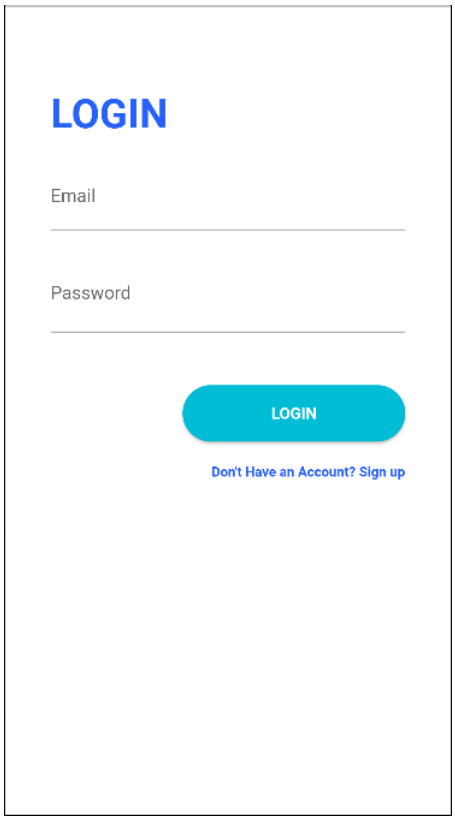

Figure 6.1: Login Screen

mail.The username is only for displaying it inside the app and keep a copy in the database. The user must type a password and type the same password again on the confirmation textfield. If the two password textfields don't match, the user will be warned with a red text under the second textfield. The password must be greater than 10 letters and less than 16 letters while containing at least a special character from these: !, @, #, $, %, ^, &, *. The user just like on the Login screen, must be connected to the internet in order to proceed to the main screen. On this image we can see the Register screen (see figure 6.2).

6.3 Main Screen

This is the main screen of the app. This is where the user gets redirected after the registration or the login. Here we can see an image of the main screen without activity permission (see figure 6.3).

If we want to collect daily steps we must give the permission of activity to the app(if the user has an android device with android version 8 or 9 it's not required to give

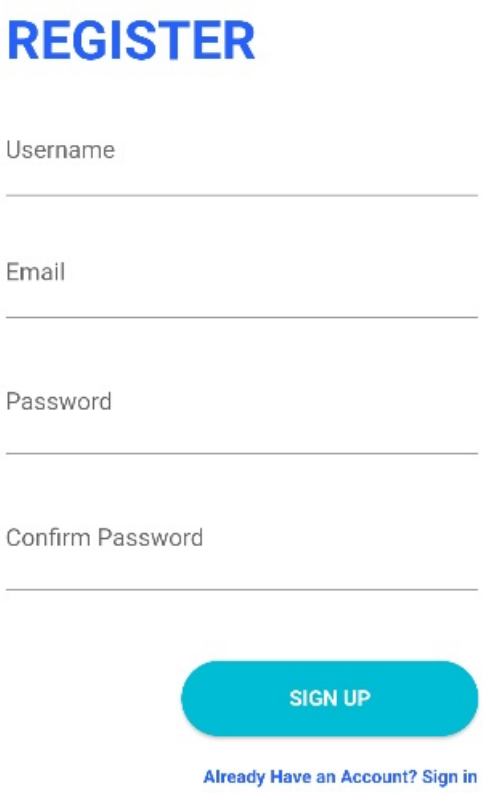

Figure 6.2: Register Screen

permission). After we press the button 'Request Permission' a dialog will appear (see figure 6.4),

telling us to enter the daily steps target the user want, the height of the user(must be between 0-250cm else there will a warning which will not let the user to press the button ok) and the option to select between male or female for accurate tracking of kilometers walked by the user (the activity permission will be asked only the first time of opening the app). After the user presses the button 'ok' the user can see his/hers progress on a circular progress bar(due to using the emulator on pc to take images, the emulator doesn't have pedometer so the message is 'Pedometer not available'). The card (see figure 6.5) which is a widget (card is the widget name on Flutter which has all the text and the circular progress bar of the main screen)

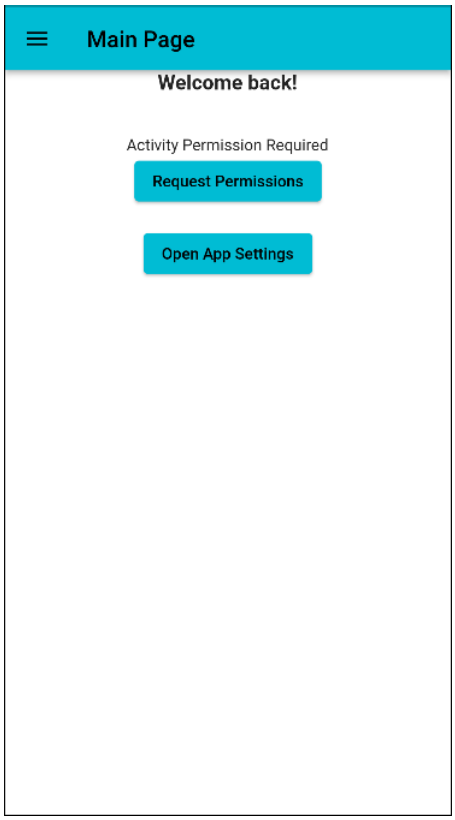

Figure 6.3: Main Screen without Activity Permission

contains all the information of the users daily steps.

The daily steps are saved once per day on SQL database table (locally on device) for later synchronization with a central database. On the top right corner we can see a target icon which is a button. From there we can change the daily steps target of the user and the height of the user(in the case a child uses the app). On this image (see figure 6.6) we can see on the dialog of the target button:

On the top left corner there are 3 parallel lines, if the user presses it or by sliding from the left to the right of the screen the sidemenu will appear on which the user can navigate to the rest of the screens. On this image (see figure 6.7) we can see the sidemenu.

6.4 Sidemenu

This is where the user can navigate through the multiple screens of the app. The Home Screen (or Main Screen) is the main screen of the app, the Navigation screen is where the user can see a map with the current position in it and the path following

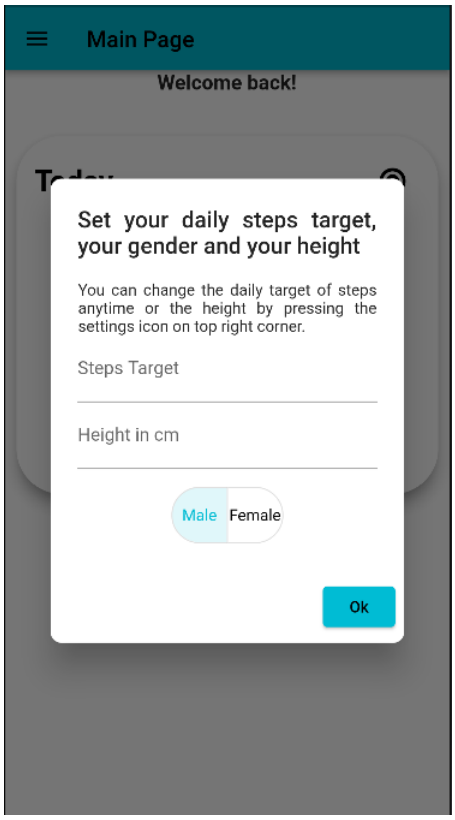

Figure 6.4: Activity Permission Dialog

the user icon with a red line, Compass screen is a screen with compass(which uses the magnetometer of the device), location, address and altitude. Sensors is the screen in which the user can see all the data from the device sensors. Settings is the screen in which the user can change between light and dark theme and the Logout is a button that appears a dialog box in which the app asks the user if he/she wants to logout from the app. If the user presses the button 'Yes', the user will be redirected on the Login screen and he/she will have to enter the credentials in order to login again or the next time the user opens the app. On this image (see figure 6.8) we can see the dialog box of Logout:

6.5 Navigation Screen

This is the screen where the map is. The user can find his/hers current position on the map. In order for the tracking of the user to work properly the user must have enabled the Gps on the device before opening this screen, else the map will be loaded but the coordinates won't update each time the user changes location.

Figure 6.5: Main Screen with activity Permission

After entering this screen for the first time the app will require real time tracking permission in order for the map to be loaded, so two buttons are on the screen, the first one to select without the type of permission the user wants to give to the app and the second one is to open app settings so that the user will give manually the real time tracking permission to the app. On this image (see figure 6.9) we can see how the screen looks without tracking permission:

After the user gives the required permission to the app, the map will be loaded using the internet connection and will show the user's current position on map. On the bottom right corner there is the button which zoom to user current location. If the user navigate to the map losing the current location marker this button also centers the map with the center point the user's current position marker. Every time the user changes location a copy of the timestamp, latitude and longitude is saved on SQL database table (locally on device) for later synchronization with a central database. This is how the screen looks when the user presses the button (see figure 6.10).

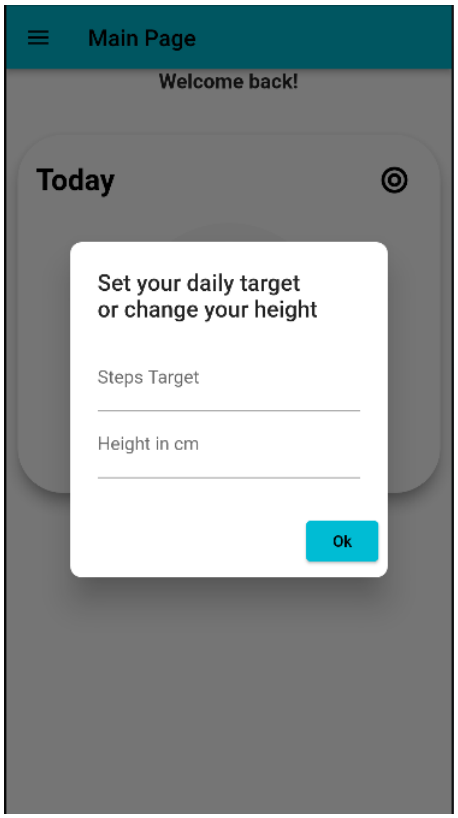

Figure 6.6: Target Button dialog

6.6 Compass Screen

This is the screen where the user can the compass using the magnetometer of the device. This screen also requires from the user to give real time tracking permission, not for using the compass, but for getting the latitude, longitude, address and altitude. This is how the screen look like before getting the permission by the user (see figure 6.11).

In order for the app to show the coordinates (latitude, longitude) the user must enable the gps and press the button on the top right corner (pin icon) to get the current coordinates. If the user also wants the current address and the current altitude the user must connect to the internet by opening the wi-fi or the cellular data and then pressing again the pin button. Every time the user changes location a copy of the timestamp, latitude, longitude and altitude is saved on SQL database table (locally on device) for later synchronization with a central database (in order for the altitude to be saved an internet connection is required). The compass in

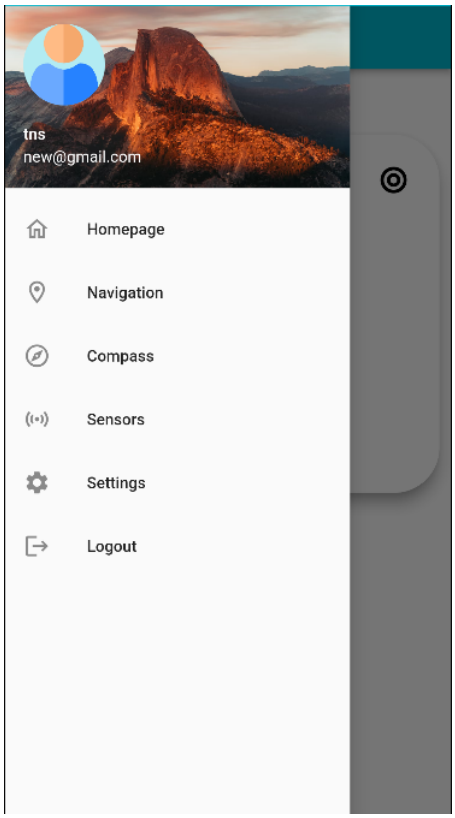

Figure 6.7: Sidemenu

order to work doesn't require enabled gps or internet connection. This figure 6.12 shows the image after the user gives the real time tracking permission (the gps and the internet are disabled).

6.7 Sensors Screen

This is the screen where the user can see some sensors of the device such as pedometer, barometer, accelerometer, gyroscope, magnetometer and proximity. If the device doesn't have a sensor then below its sensor there will be a message of the current sensor saying that the sensor isn't available. The pressure sensor shows the pressure on milli bars. The proximity sensor show if something is close enough to the sensor, under 1 cm the sensor shows yes else is shows no. Every 10 seconds a copy of each sensor data is taken and with a timestamp is saved on SQL database table (locally on device) for later synchronization with a central database(total count of steps is '-' because the image was taken by an emulator). The image 6.13 shows the screen of the sensors.

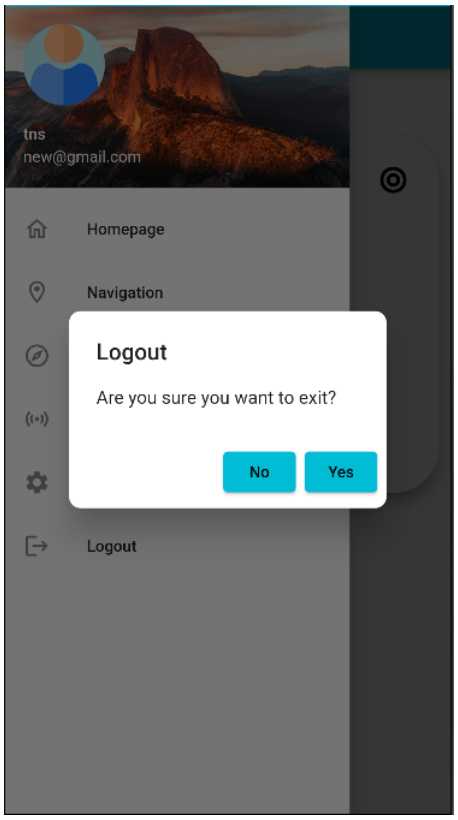

Figure 6.8: Logout Button

6.8 Settings Screen

This is the last screen, for the time being the user have only one setting, to choose between light and dark mode to change while using the app, even though the app takes by default the theme that the device currently use, the user can change it anytime. This is how settings screen look like (see figure 6.14).

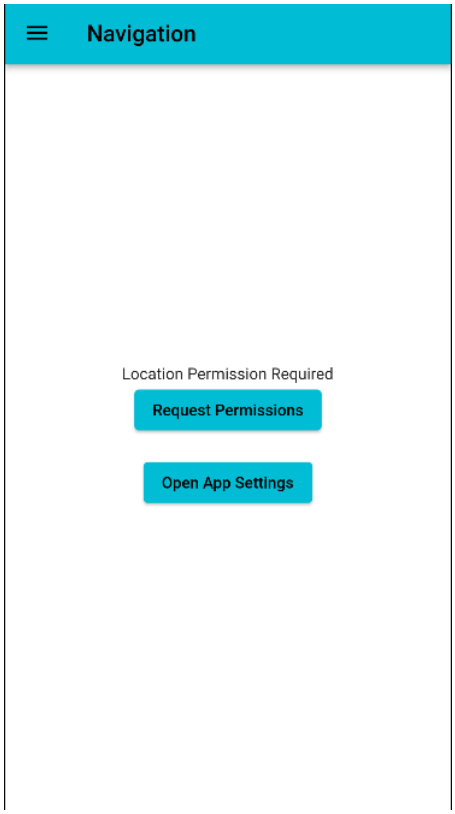

Figure 6.9: Navigation Screen without location permission

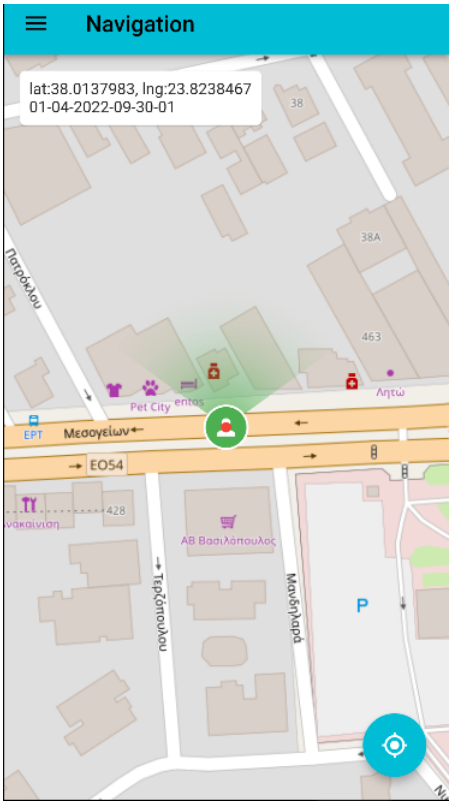

Figure 6.10: Navigation Screen with loaded map

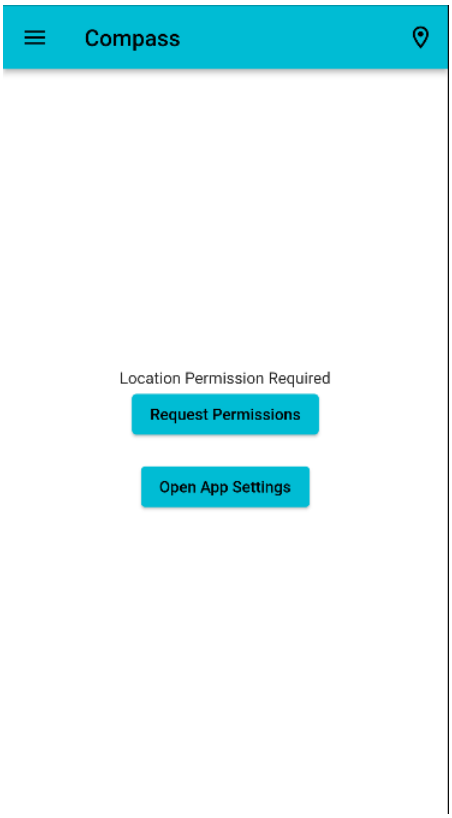

Figure 6.11: Compass Screen without location permission

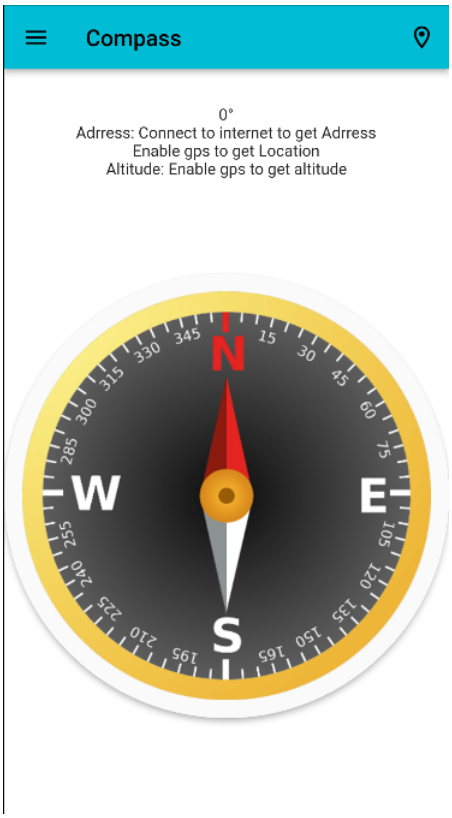

Figure 6.12: Compass Screen

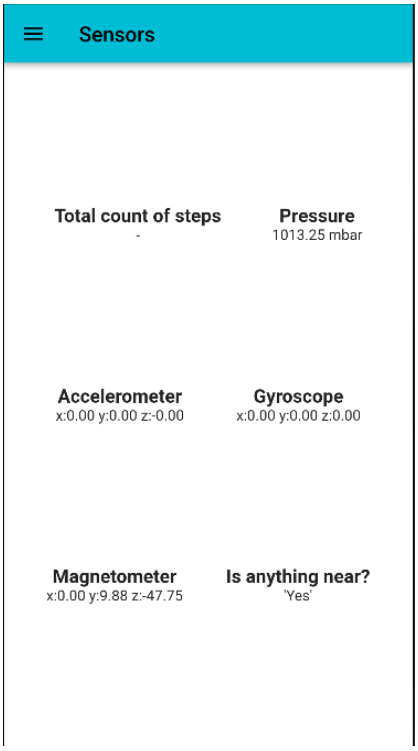

Figure 6.13: Sensors Screen

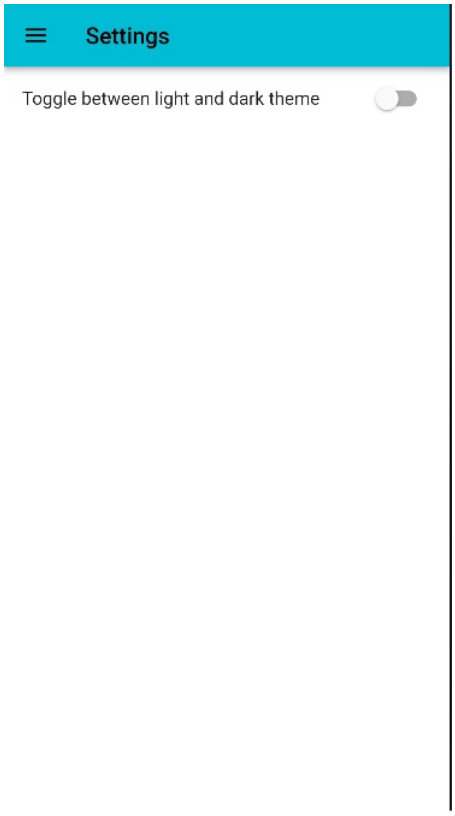

Figure 6.14: Settings Screen

Chapter 7

Development of the application

7.1 User Login

The Login screen (Login.dart) is extended with a stateful widget which means that this screen has states, because it has User Interface (UI) elements that changes if the user interacts with them. Also it has only 4 UI elements, two textfields, one button and one gesture detector. To get the text from each textfield we used a TextEditingController for email textfield and another one for the password textfield. The TextEditingController also requires a listener to work properly, so it must be initialized once every time the screen is opened for the first time the app is launched. To do this we used the function initState which is a core component of a stateful class. Then, when the app terminates the listeners must be disposed (see figure 7.1).

The first textfield in which the user can enter the email has built-in an email validator. We used the package 'email_validator' (`https://pub.dev/packages/\email_validator`) which checks if the value in the current textfield is an email. Basically it checks if the text has the necessary form of an email (for example text@text.text). We built a function to check whether the email textfield is empty or if the email is valid (see figure 7.2).

When a user enters something and the deletes it, then there is an error text (a small red text below the textfield) indicating that the textfield can't be empty but the text won't appear if the textfield doesn't change for the first time (see figure 7.3).

```
@override
void initState(){
  super.initState();

  //Start listening to changes with listeners
  mail_txtController.addListener(mailvalue);
  pass_txtController.addListener(passvalue);
}

@override
void dispose(){
  //Clean controllers when the widget is removed from the widget tree
  //and removes the values of both listeners
  mail_txtController.dispose();
  pass_txtController.dispose();
  super.dispose();
}
```

Figure 7.1: Textfield Listeners

In order to know if the textfield changes state we used the built-in function of Textfield widget called onChanged which monitors every change it happens on the current textfield. Then we have a boolean variable which calls a function returning true or false if the textfield changed for the first time. If it changed it calls the function to check the context of the text (as seen on the image above), if the text is an email, if the text is empty or if the text is a valid email (in which case the error text disappears). The password textfield works with the same logic. We used the same built-in function onChanged of the Textfield widget to know if there is a text entered in the textfield for the first time by setting another boolean variable which calls a function returning true or false. Then a function is called to check if there is a text entered in the textfield or not (see figure 7.4).

If the user types something in the password textfield and then deletes it an error text below the textfield will appear warning the user that the password textfield can't be empty (see figure 7.5).

The password textfield has also the option to hide and show the password as the user is typing it by clicking the eye button at the end of the textfield. This is feasible by using a built-in function of the textfield called obscuredText. We used this function

```
//Function for displaying the correct error message on email textfield
String? Mail_Textfield_check(){
  String mail_msg='';
  if(mail_txtController.text.isEmpty==true){
    mail_msg='Email can\'t be empty';
    print(mail_msg);
    mail_check=false;
    return mail_msg;
  }
  else if(EmailValidator.validate(mail_txtController.text)==false){
    mail_msg='Enter a valid Email';
    print(mail_msg);
    mail_check=false;
    return mail_msg;
  }
  else if(EmailValidator.validate(mail_txtController.text)==true){
    mail_check=true;
    mail_msg='Valid email';
    print(mail_msg);
  }
}
```

Figure 7.2: Mail Textfield function

with a boolean variable that when the user presses the button the variable changes from true to false and vice versa. This is how the button works (see figure 7.6).

If the user doesn't have an account he/she must create one from the registration page by pressing the text 'Don't Have an Account? Sign up' which is a GestureDetector widget. It has a funtion called onTap that triggers when the user touches the text. When the onTap function is triggered we forward the user to the registration page for sign up. Finally the user must press the button Login in order to proceed. When the button is pressed the user must have already enabled the device Wi-fi or cellular data and has a stable connection to the internet or else the user won't be able to login and there will be pop up text warning the user that the device isn't connected to the internet. For checking if the user has the device Wi-fi or cellular data enabled we used the package 'connectivity_plus' (`https://pub.dev/packages/connectivity_plus`) and for checking if the user is connected to the internet the package 'internet_connection_checker' (`https://pub.dev/packages/internet_connectio`

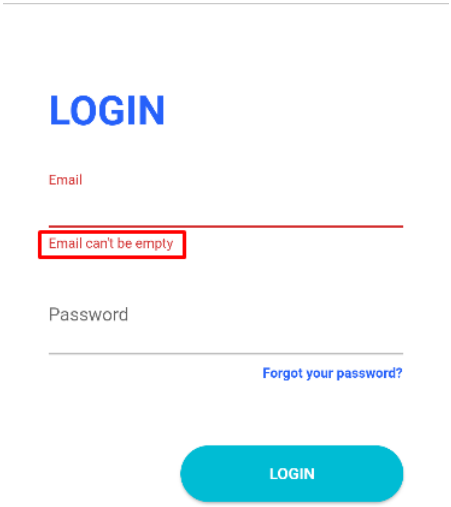

Figure 7.3: Email Textfield error text

`n_checker`). The pop up text is called Toast and we used the package 'fluttertoast' (`https://pub.dev/packages/fluttertoast`) in order to display a message with an animation to the user. On this image (see figure 7.7) we can see the toast message that is displayed when the user is connected to the internet.

Also the two textfields must be valid in order to login. If the above conditions are met then the user will be redirected to the main page of the app.

7.2 User Registration

Registration page (Signup.dart) is extended with a stateful widget. It consists by 6 UI elements, four textfields, one button and one gesture detector. To get the text from each textfield we used the same logic just like on the login screen, one TextEditingController with its listener for each textfield. Inside the initState function we have the initializations of the listeners and the dispose of them inside the dispose function (see figure 7.8).

The first textfield is a simple textfield for only saving and showing the username of

```
//Function for displaying the correct error message on password textfield
String? Pass_Textfield_check(){
  String pass_msg='';
  if(pass_txtController.text.isEmpty==true){
    pass_msg='Password can\'t be empty';
    print(pass_msg);
    print(pass_check);
    pass_check=false;
    return pass_msg;
  }
  else if(pass_txtController.text.isEmpty==false){
    pass_check=true;
    pass_msg='Valid password';
    print(pass_msg);
  }
}
```

Figure 7.4: Password Textfield function

the user. It shows an error message below the textfield in case that the user writes a username and then delete it to warn the user that the textfield can't be empty. We used the function onChanged from the Textfield widget to know when the user types on the textfield and then we used the function below to warn the user if it is necessary (see figure 7.9).

The email textfield is built the exact same way as in the login screen using the function onChanged to use a function for checking if the textfield is empty or if the email is valid. The password textfield again uses the onChanged function of textfield widget with a function to check if the criteria for a strong password are met which are: the length of the password must be between 10 and 16 letters and use at least a special character like !, @, #, $, %, ˆ, &, . The same function also checks if the textfield is empty (see figure 7.10).

The functionality for the hide and show button at the end of the textfield is the same as in the Login screen. The confirm textfield also uses the function onChanged to call a function to check if the text that is written on the password textfield is the same as the one written on the confirm password textfield. The function also checks if the textfield is empty and warns the user (see figure 7.11).

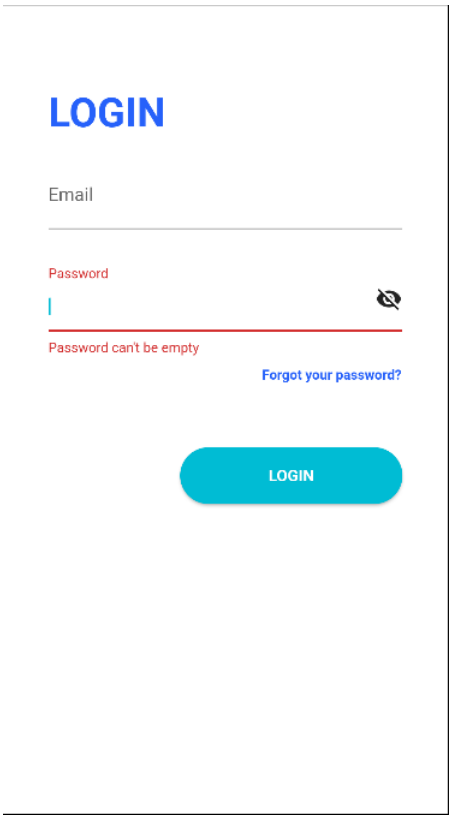

Figure 7.5: Password Textfield error text

The user finally must press the button 'Sign up' in order to create an account. The user must have enabled the Wi-fi or the cellular data of the device and a stable connection to the internet before pressing the button just like the login button on the Login screen. Of course the four textfields must have a valid context to proceed to get redirected to the Main screen or else a Toast message will appear to warn the user that some credentials may be missing or to connect to the internet. The text below the button named 'Already Have an Account? Sign in' is GestureDetector widget with a onTap function that when it's triggered by touch the user will be redirected to the Login screen.

7.3 Foreground Functionality

When the user closes the app, the app by default will stop all the functionality it has and will shut down. Before searching for the best package that does exactly what we wanted we had to search what does it take to keep an app running if the user changes to another app or if the user locks the device. We searched first on

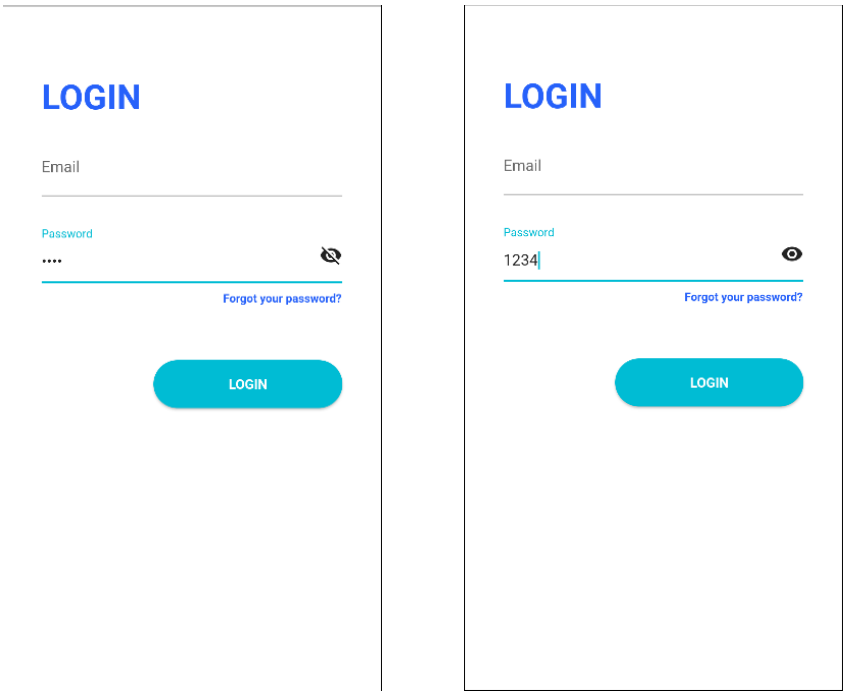

Figure 7.6: Password Textfield show/hide text button

Android what is needed to make it work (writing native code) and we found out that the app must become a service. In the Android documents it states that a service is an application component that can perform long-running operations in the background and it does not provide a user interface. If it starts a service might continue running for some time even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). The service has three types: Foreground, Background, Bound.

- The foreground service performs some operation that is noticeable to the user and it must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

- The background service performs an operation that isn't directly noticed by the user and it isn't necessary to display a Notification.

- A service is bound when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do

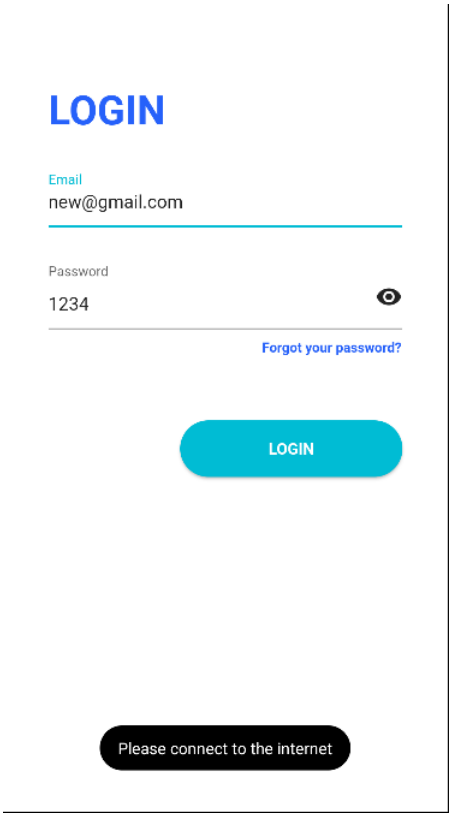

Figure 7.7: Toast pop up message

so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

So we had to find a way to make the app a background service or a foreground service. After some search we found that an app like this which uses real time tracking must be a foreground service as Android and Apple documents suggest. But before we found this suggestion we installed two packages (both of them didn't work as intended) that make the app a background service which can be found `https://pub.dev/packages/flutter_background`, `https://pub.dev/packages/flutter_background_service`. After the mentioned suggestion we tried some packages that make the app a foreground service. We used these four packages, `https://pub.dev/packages/foreground_service`, `https://pub.dev/packages/flutter_foreground_plugin`, `https://pub.dev/packages/flutter_foreground_service`, `https://pub.dev/packages/flutter_foreground_service_plugin`.

```dart
@override
void initState(){
  super.initState();

  //Start listening to changes with listeners
  user_txtController.addListener(uservalue);
  mail_txtController.addListener(mailvalue);
  pass_txtController.addListener(passvalue);
  confpass_txtController.addListener(confvalue);
}

@override
void dispose(){

  //Clean controllers when the widget is removed from the
  //widget tree and removes the values of listeners
  user_txtController.dispose();
  mail_txtController.dispose();
  pass_txtController.dispose();
  confpass_txtController.dispose();
  super.dispose();
}
```

Figure 7.8: Textfield listeners

```dart
//Function for displaying the correct error message on username textfield
String? User_Textfield_check(){
  String user_msg='';
  if(user_txtController.text.isEmpty==true){
    user_msg='Username can\'t be empty';
    print(user_msg);
    user_check=false;
    return user_msg;
  }
  else if(user_txtController.text.isEmpty==false){
    user_check=true;
    user_msg='Valid username';
    print(user_msg);
  }
}
```

Figure 7.9: User Textfield function

None of these worked as we would like to. As a last resort we thought of writing native code using platform channels for both Android and IOS but that would be difficult and we had to make it ourselves because we didn't found anything similar. We found a last package called 'flutter_foreground_task' (`https://pub.dev/packages/flutter_foreground_task`). Fortunately it worked as we would like to and we integrate it in the app. The whole code of the foreground functionality must be inside the main file which is the main screen and so the app will begin running in the foreground after the user enters the main screen for the first time. To achieve this

```
//Function for displaying the correct error message on password textfield
String? Pass_Textfield_check(){
  String pass_msg='';
  if(pass_txtController.text.isEmpty==true){
    pass_msg='Password can\'t be empty';
    print(pass_msg);
    pass_check=false;
    return pass_msg;
  }
  else if(pass_txtController.text.isEmpty==false){
    if(pass_txtController.text.length < 10 || pass_txtController.text.contains(new RegExp(r'(?=.*[!@#$%^&*])')) == false){
      pass_msg='Password must be at least 10 letters\nand contain special characters';
      pass_check=false;
      return pass_msg;
    }
    else if(pass_txtController.text.length > 16 || pass_txtController.text.contains(new RegExp(r'(?=.*[!@#$%^&*])')) == false){
      pass_msg='Password must be maximum 16 letters\nand contain special characters';
      pass_check=false;
      return pass_msg;
    }
    else{
        pass_check=true;
        pass_msg='Valid password';
        print(pass_msg);
    }
  }
}
```

Figure 7.10: Password Textfield function

functionality the home of the app (which is the first screen the user will encounter by default as Flutter provides) must be wrapped by the widget WithForegroundTask that the foreground package provides. The code below is provided by the package and it must be included to initiate the functionality (see figure 7.12, 7.13).

The code below is used for configurating in each OS the Notifications options the programmer wants to have. On IOS the options are very limited compared to Android but we kept the notification simple on both operating systems (see figure 7.14, 7.15).

As always the functions initForegroundTask and StartForegroundTask which are for notifications options and starting the notification service respectively are stated inside the initState function on Main screen class. This is how the notification looks like (see figure 7.16).

```dart
//Function for displaying the correct error message on confirmation password textfield
String? Conf_Textfield_check(){
  String conf_msg='';
  if(confpass_txtController.text.isEmpty==true){
    conf_msg='Password can\'t be empty';
    print(conf_msg);
    conf_check=false;
    return conf_msg;
  }
  else if(confpass_txtController.text.isEmpty==false){
    if(confpass_txtController.text.compareTo(pass_txtController.text) != 0){
      conf_check=false;
      conf_msg='Password isn\'t same as the one above';
      return conf_msg;
    }
    else{
      conf_check=true;
      conf_msg='Valid username';
      print(conf_msg);
    }
  }
}
```

Figure 7.11: Confirmation Textfield function

```dart
// The callback function should always be a top-level function.
void startCallback() {
  // The setTaskHandler function must be called to handle the task in the background.
  FlutterForegroundTask.setTaskHandler(FirstTaskHandler());
}

class FirstTaskHandler extends TaskHandler {
  int updateCount = 0;

  @override
  Future<void> onStart(DateTime timestamp, SendPort? sendPort) async {
    // You can use the getData function to get the data you saved.
    final customData = await FlutterForegroundTask.getData<String>(key: 'customData');
    print('customData: $customData');
  }

  @override
  Future<void> onEvent(DateTime timestamp, SendPort? sendPort) async {
    FlutterForegroundTask.updateService(notificationTitle: 'FirstTaskHandler',notificationText: timestamp.toString(),callback: updateCount >= 10 ? updateCallback : null);
    // Send data to the main isolate.
    sendPort?.send(timestamp);
    sendPort?.send(updateCount);

    updateCount++;
  }

  @override
  Future<void> onDestroy(DateTime timestamp) async {
    // You can use the clearAllData function to clear all the stored data.
    await FlutterForegroundTask.clearAllData();
  }

  @override
  void onButtonPressed(String id) {
    // Called when the notification button on the Android platform is pressed.
    print('onButtonPressed >> $id');
  }
}
```

Figure 7.12: Basic Handler of Foreground functionality

```dart
void updateCallback() {
  FlutterForegroundTask.setTaskHandler(SecondTaskHandler());
}

class SecondTaskHandler extends TaskHandler {
  @override
  Future<void> onStart(DateTime timestamp, SendPort? sendPort) async {

  }

  @override
  Future<void> onEvent(DateTime timestamp, SendPort? sendPort) async {
    FlutterForegroundTask.updateService(notificationTitle: 'SecondTaskHandler', notificationText: timestamp.toString());
    // Send data to the main isolate.
    sendPort?.send(timestamp);
  }

  @override
  Future<void> onDestroy(DateTime timestamp) async {

  }
}
```

Figure 7.13: Assistant Handler of Foreground functionality

```dart
Future<void> initForegroundTask() async {
  await FlutterForegroundTask.init(
    androidNotificationOptions: AndroidNotificationOptions(
      channelId: 'notification_channel_id',
      channelName: 'Foreground Notification',
      channelDescription:'This notification appears when the foreground service is running.',
      channelImportance: NotificationChannelImportance.LOW,
      priority: NotificationPriority.LOW,
      iconData: const NotificationIconData(
        resType: ResourceType.mipmap,
        resPrefix: ResourcePrefix.ic,
        name: 'launcher',
      ),  // NotificationIconData
    ),  // AndroidNotificationOptions
    iosNotificationOptions: const IOSNotificationOptions(
      showNotification: true,
      playSound: true
    ),
    foregroundTaskOptions: const ForegroundTaskOptions(
      interval: 1000,
      autoRunOnBoot: false,
      allowWifiLock: true,
    ),
    printDevLog: true,


  );
}
```

Figure 7.14: Foreground functionality options

```dart
Future<bool> startForegroundTask() async {
  // You can save data using the saveData function.
  await FlutterForegroundTask.saveData(key: 'customData', value: 'hello');

  ReceivePort? receivePort;
  if (await FlutterForegroundTask.isRunningService) {
    receivePort = await FlutterForegroundTask.restartService();
  } else {
    receivePort = await FlutterForegroundTask.startService(
      notificationTitle: 'App is running on the background',
      notificationText: 'Tap to return to the app',
      callback: startCallback,
    );
  }

  if (receivePort != null) {
    _receivePort = receivePort;
    _receivePort?.listen((message) {
      if (message is DateTime) {
        print('receive timestamp: $message');
      } else if (message is int) {
        print('receive updateCount: $message');
      }
    });

    return true;
  }

  return false;
}

Future<bool> stopForegroundTask() async {
  return await FlutterForegroundTask.stopService();
}
```

Figure 7.15: Foreground start and stop function

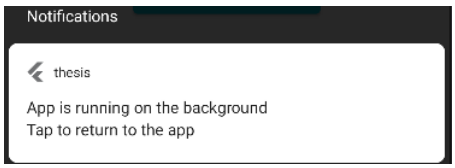

Figure 7.16: Foreground Notification

7.4 App lifecycle

Flutter provides a way of observing when the app changes state with the function didChangeAppLifecycleState. With this function the programmer can observe the states of the app:

- inactive - The application is in an inactive state and is not receiving user input (iOS only)
- paused - The application is not currently visible to the user, not responding to user input, and running in the background
- resumed - The application is visible and responding to user input
- suspending - The application will be suspended momentarily (Android only)
- detached – The application is closing

In order for this function to work the class must have an Observer by adding with WidgetsBindingObserver after the name of the class. Then we initiated an Observer on initState with 'WidgetsBinding.instance?.addObserver(this)' and dispose it on dispose function with 'WidgetsBinding.instance?.removeObserver(this)'. It wasn't needed by default on every Flutter app to observe the state of the app but we wanted to know when the app go to the background or when the user is interacting with the app to know when we must enable or disable the background location functionality. Also we wanted to stop foreground functionality of the app when when user close the app.

7.5 Permissions

If the programmer wants to ask the user to collect data from the sensors of the device it is necessary to include permissions. Flutter doesn't provide how to ask permissions so we used the package 'permission_handler' (`https://pub.dev/packages/permission_handler`). We used this package for getting permissions for activity tracking (getting daily steps) and real time location (getting longitude and

latitude). We added activity permission in my main file (Main Screen) in which the user must accept the permission in order for the daily steps to be counted. We added two buttons with two different options (see figure 7.17).

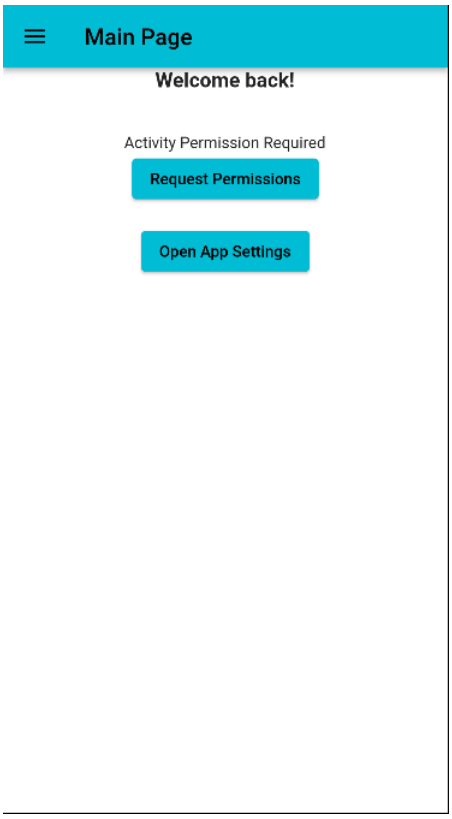

Figure 7.17: Activity Permission buttons

the first one to ask the user if he/she wants to give permission for activity tracking with a system pop up directly (see figure 7.18)

and the second button is for redirecting the user on the app settings in order for the user to manually accept the settings he/she wants. The permission for the real time location is located in the navigation screen and in the compass screen. When the user selects for the first time the navigation screen the permission for the location will show up with a system pop up and the user have 3 options, to use location when the app is in use, to give one time permission of location to the app or to deny to give permission (see figure 7.19).

If the user deny on the pop up two buttons will appear on the screen, one for requesting permission for the location appearing the same pop up again and the other button will open the app's settings. Only on Android 10 and above in the

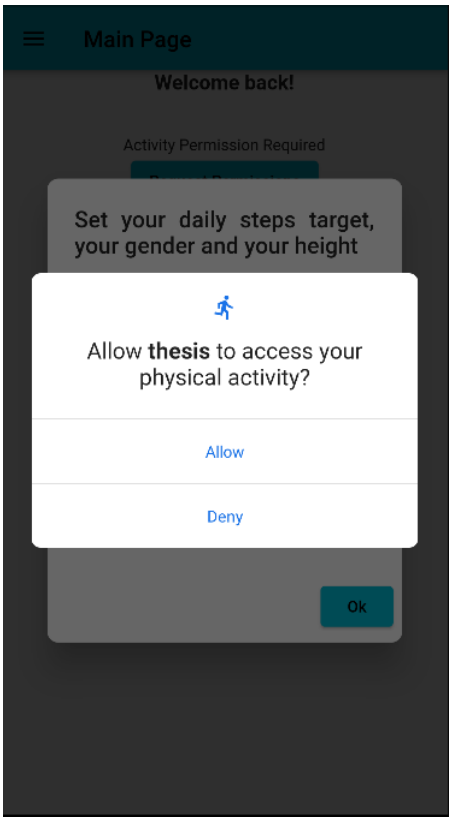

Figure 7.18: Activity Permission pop up

options for the location permission there is a fourth option (Allow all the time) that the user must select through app's settings in order for the background location to working, if the user didn't select this option the next time the user enters the navigation screen the app will redirect the user to the permissions options of the app to select it (see figure 7.20).

If the user gives location permission for the first time to the app through navigation screen the app won't request the permission again on compass screen and if the user gives location permission on the first time on compass screen the app won't request the permission again on navigation screen.

7.6 Sensors

7.6.1 Pressure Sensor

To get data from the pressure sensor we got through a mini odyssey. We first searched if there was available a package that gets the pressure data from the sensor

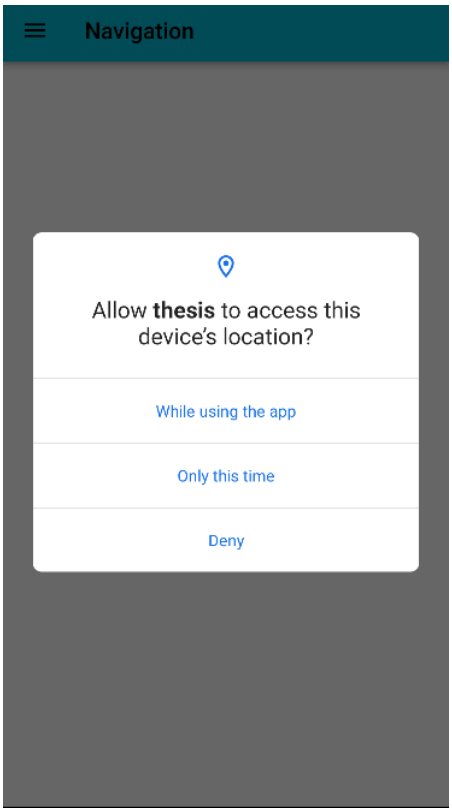

Figure 7.19: Location Permission pop up

on both Operating Systems (Android and IOS) but there wasn't anything on the time we were building the functionality. So the next worst solution would be to write native code on Kotlin for Android and Swift for IOS and connect the native code with Flutter using platform channels. Thankfully we found exactly one github project on which we were able to support some code and build on top of that on github (`https://github.com/nhandrew/platformcode`). So on Android side we created a main file with name MainActivity.kt and we created a variable a method channel and an event channel (In short both method channel and event channel exist for communicating with dart, method channel is for sending information like status and event channel is for sending values even if they change dynamically). To connect two same types of channels between native and Dart the channels must have the same name (for example the variable for availability of pressure sensor is called pressure_sensor on Android, IOS and Dart file). The method channel purpose is for sending if the device has a pressure sensor or not. We can see the code (see figure 7.21) on Android for the availability of the sensor,

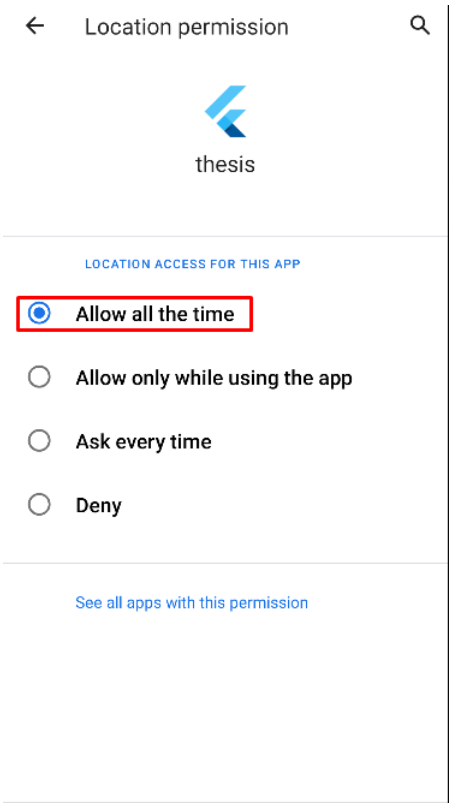

Figure 7.20: App Settings Location Permission on Android 10 and above

```
//Channel for pressure
presschannel = MethodChannel(messenger, press_channel)
presschannel!!.setMethodCallHandler{
    call,result ->
    if (call.method == "isSensorAvailable") {
        result.success(sensorManager!!.getSensorList(Sensor.TYPE_PRESSURE).isNotEmpty())
    } else {
        result.notImplemented()
    }
}
```

Figure 7.21: Method channel for pressure availability

the code for the data from the sensor we made a class Stream Handler in another file which basically registers a listener to get the events from the sensor (see figure 7.22)

and then we call the StreamHandler class in the main file (see figure 7.23).

For IOS we followed the same steps by making a .swift file having the main function inside. Then we made a variable for the method channel (pressure_sensor) and a variable for event channel (pressure_channel). For checking if the pressure sensor is available on the current device we called the function (see figure 7.24)

and we followed the same methodology for getting data from the sensor making

```
class StreamHandler(private val sensorManager: SensorManager, sensorType: Int, private var interval: Int = SensorManager.SENSOR_DELAY_NORMAL):
        EventChannel.StreamHandler, SensorEventListener {
    private val sensor = sensorManager.getDefaultSensor(sensorType)
    private var eventSink: EventChannel.EventSink? = null

    override fun onListen(arguments: Any?, events: EventChannel.EventSink?) {
        if (sensor != null){
            eventSink = events
            sensorManager.registerListener( listener: this, sensor, interval)
        }
    }

    override fun onCancel(arguments: Any?) {
        sensorManager.unregisterListener( listener: this)
        eventSink = null
    }

    override fun onSensorChanged(event: SensorEvent?) {
        val sensorValues = event!!.values[0]
        eventSink?.success(sensorValues)
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {

    }
}
```

Figure 7.22: Stream Handler for observation of the data on Android

```
pressureChannel = EventChannel(messenger, pressure_channel)
pressureStreamHandler = StreamHandler(sensorManager!!, Sensor.TYPE_PRESSURE)
pressureChannel!!.setStreamHandler(pressureStreamHandler)
```

Figure 7.23: Pressure Stream Handler initialization on Android

another class inside the file to handle the events of the sensor (see figure 7.25)

and then we called the function inside the main file (see figure 7.26).

After this procedure the native code on both Android and IOS is done and we moved to connect the native code with Dart. The pressure functionality is inside the Sensors screen in which also the user can see if the device has pressure sensor or not and the data it gets from the sensor. First we created the corresponding method channel and event channel and then we made a function to check if the device has a pressure sensor (see figure 7.27).

Then we called the function in initState to be called once every time the app opens this screen to check if there is pressure sensor on the device and also listen to the StreamHandler to get data from the sensor in case there is with the code below (see figure 7.28).

7.6.2 Proximity Sensor

To get data from proximity sensor it was much easier because there was a package called 'proximity_sensor' (`https://pub.dev/packages/proximity_sensor`)

```
let presschannel = FlutterMethodChannel(name: press_channel, binaryMessenger: controller.binaryMessenger)

presschannel.setMethodCallHandler({
    (call: FlutterMethodCall, result: @escaping FlutterResult) -> Void in
    switch call.method {
    case "isSensorAvailable":
        result(CMAltimeter.isRelativeAltitudeAvailable())
    default:
        result(FlutterMethodNotImplemented)
    }
})
```

Figure 7.24: Pressure Method Handler on IOS

```
class PressureStreamHandler: NSObject, FlutterStreamHandler {
    let altimeter = CMAltimeter()
    private let queue = OperationQueue()

    func onListen(withArguments arguments: Any?, eventSink events: @escaping FlutterEventSink) -> FlutterError? {

        if CMAltimeter.isRelativeAltitudeAvailable() {
            altimeter.startRelativeAltitudeUpdates(to: queue) { (data,error) in
                if data != nil {
                //Get pressure
                let pressurePascals = data?.pressure
                    events(pressurePascals!.doubleValue * 10.0)
             }
            }
        }
        return nil
    }

    func onCancel(withArguments arguments:Any?) -> FlutterError? {
        altimeter.stopRelativeAltitudeUpdates()
        return nil
    }

}
```

Figure 7.25: Pressure Stream Handler on IOS

which has pre-build functions to get data from the sensor. On Android to know if there is a proximity sensor on the device (or at least a virtual sensor) we made a method channel writing native code in the same file we made for pressure sensor (MainActivity.kt) to send the status of the sensor (see figure 7.29).

Then in the Sensors screen (Sensors.dart) we created a function for getting the availability of the sensor from the method channel (see figure 7.30).

```
let pressurechannel = FlutterEventChannel(name: pressure_channel, binaryMessenger: controller.binaryMessenger)
pressurechannel.setStreamHandler(pressureStreamHandler)
```

Figure 7.26: Pressure Stream Handler initialization on IOS

```
//Future for checking the availability of pressure sensor
Future<void> check_pressure_availability() async {
  try {
    var available = await press_channel.invokeMethod('isSensorAvailable');
    setState(() {
      press_check = available;
    });
  } on PlatformException catch (e) {
    print(e);
  }
}
```

Figure 7.27: Function for getting the availability of the pressure sensor

```
//pressure initialization event
pressureSubscription = press_channel.receiveBroadcastStream().listen((event) {
  setState(() {
    if(press_check == true){
      pressure=event;
      pmsg = '${pressure.toStringAsFixed(2)} mbar';
      if(press_check == false)
      {
        pmsg = 'Pressure not available';
      }
    }
    else{
      pmsg = 'Pressure not available';
    }
  });
});
```

Figure 7.28: Pressure Subscription to get the stream of the pressure sensor

```
//Channel for proximity
proxchannel = MethodChannel(messenger, prox_channel)
proxchannel!!.setMethodCallHandler{
    call,result ->
    if (call.method == "isSensorAvailable") {
        result.success(sensorManager!!.getSensorList(Sensor.TYPE_PROXIMITY).isNotEmpty())
    } else {
        result.notImplemented()
    }
}
```

Figure 7.29: Proximity Method Channel

```
//Future for checking the availability of proximity sensor
Future<void> check_proximity_availability() async {
  try {
    var available = await prox_channel.invokeMethod('isSensorAvailable');
    setState(() {
      prox_check = available;
    });
  } on PlatformException catch (e) {
    print(e);
  }
}
```

Figure 7.30: Proximity check function

and a function to get the data from the sensor (see figure 7.31).

```
//Future for getting data from proximity sensor
Future<void> listenSensor() async {
  FlutterError.onError = (FlutterErrorDetails details) {
    if (foundation.kDebugMode) {
      FlutterError.dumpErrorToConsole(details);
    }
  };
  _streamSubscription = ProximitySensor.events.listen((int event) {
    setState(() {
      if(prox_check == true) {
        _isNear = (event > 0) ? true : false;
        if (_isNear == true) {
          nmsg = "'Yes'";
        }
        else {
          nmsg = "'No'";
        }
      }
      else{
        nmsg = 'Proximity not available';
      }
    });
  });
}
```

Figure 7.31: Proximity Subscription to get the stream of the pressure sensor

On IOS there is no need to check for proximity sensor because every supported device to run this app has the sensor.

7.6.3 Accelerometer-Magnetometer–Gyroscope Sensors

To get data from acceleration sensor, magnetometer sensor and gyroscope sensor we used the package 'sensors_plus' (`https://pub.dev/packages/sensors_plus`) which has pre-build functions to get data from the sensors but not the availability of them in the device. To get the availability of the sensors we followed the same strategy as in proximity and pressure sensors by writing native code and using method channels for communicating with the Dart files. Below we can see the native code used for checking availability (see figure 7.32).

```
//Channel for accelerometer
accchannel = MethodChannel(messenger, acc_channel)
accchannel!!.setMethodCallHandler{
    call,result ->
    if (call.method == "isSensorAvailable") {
        result.success(sensorManager!!.getSensorList(Sensor.TYPE_ACCELEROMETER).isNotEmpty())
    } else {
        result.notImplemented()
    }
}

//Channel for gyroscope
gyrochannel = MethodChannel(messenger, gyro_channel)
gyrochannel!!.setMethodCallHandler{
    call,result ->
    if (call.method == "isSensorAvailable") {
        result.success(sensorManager!!.getSensorList(Sensor.TYPE_GYROSCOPE).isNotEmpty())
    } else {
        result.notImplemented()
    }
}

//Channel for magnetometer
magnchannel = MethodChannel(messenger, magn_channel)
magnchannel!!.setMethodCallHandler{
    call,result ->
    if (call.method == "isSensorAvailable") {
        result.success(sensorManager!!.getSensorList(Sensor.TYPE_MAGNETIC_FIELD).isNotEmpty())
    } else {
        result.notImplemented()
    }
}
```

Figure 7.32: Method Channels for checking the availability of accelerometer, gyroscope and magnetometer

Then in the Sensors screen after we created three Future functions to get the availability of each sensor from the native code (see figure 7.33),

```
//Future for checking the availability of accelerometer sensor
Future<void> check_acc_availability() async {
  try {
    var available = await acc_channel.invokeMethod('isSensorAvailable');
    setState(() {
      acc_check = available;
    });
  } on PlatformException catch (e) {
    print(e);
  }
}

//Future for checking the availability of gyroscope sensor
Future<void> check_gyro_availability() async {
  try {
    var available = await gyro_channel.invokeMethod('isSensorAvailable');
    setState(() {
      gyro_check = available;
    });
  } on PlatformException catch (e) {
    print(e);
  }
}

//Future for checking the availability of magnetometer
Future<void> check_magn_availability() async {
  try {
    var available = await magn_channel.invokeMethod('isSensorAvailable');
    setState(() {
      magn_check = available;
    });
  } on PlatformException catch (e) {
    print(e);
  }
}
```

Figure 7.33: Future functions for getting the availability

called them inside the initState function and to get data from each sensor we also initiated inside the initState the events listeners for the data streams (see figure 7.34, 7.35).

Again for IOS there is no need to check for accelerometer, gyroscope and magnetometer sensors because every supported device to run this app has all these sensor.

7.6.4 Pedometer Sensor

For counting steps we had to get data from the pedometer sensor and once again we were lucky because there is a package named 'pedometer' (`https://pub.dev/packages/pedometer`) that does this exact functionality. We created the functions

```
//accelerometer initialization event
userAccelerometerEvents.listen((UserAccelerometerEvent event) {
  setState(() {
    if(acc_check == true){
      ax = event.x;
      ay = event.y;
      az = event.z;
      amsg='x:${ax.toStringAsFixed(2)} y:${ay.toStringAsFixed(2)} z:${az.toStringAsFixed(2)}';
    }
    else{
      amsg='Accelerometer not available';
    }
  });
});

//gyroscope initialization event
gyroscopeEvents.listen((GyroscopeEvent event) {
  setState(() {
    if(gyro_check == true) {
      gx = event.x;
      gy = event.y;
      gz = event.z;
      gmsg = 'x:${gx.toStringAsFixed(2)} y:${gy.toStringAsFixed(2)} z:${gz.toStringAsFixed(2)}';
    }
    else{
      gmsg='Gyroscope not available';
    }
  });
});
```

Figure 7.34: Accelerometer and gyroscope functions to get the streams of data

```
//magnetometer initialization event
magnetometerEvents.listen((MagnetometerEvent event) {
  setState(() {
    if(magn_check == true){
      mx = event.x;
      my = event.y;
      mz = event.z;
      mmsg='x:${mx.toStringAsFixed(2)} y:${my.toStringAsFixed(2)} z:${mz.toStringAsFixed(2)}';
    }
    else{
      mmsg='Magnetometer not available';
    }
  });
});
```

Figure 7.35: Magnetometer function to get the stream of data

below to get the number of steps, to show the availability and to initialize them (see figure 7.36).

The onStepCount count every step the user does and saves it to a local database (more on database section), the onStepCountError function checks if the device has a pedometer sensor and the initPlatformState function initializes the pedometer and registers the changes, it is called in the initState function.

```
void onStepCount(StepCount event) {
  print(event);
  setState(() {
    numsteps++;
    if(box.get('today_steps') == null){
      sum_steps = numsteps;
      box.put('today_steps',sum_steps);
    }
    else{
      sum_steps = box.get('today_steps') + numsteps;
      box.put('today_steps',sum_steps);
    }

  });

  box.put('date',date_once);
}

void onStepCountError(error) {
  print('onStepCountError: $error');
  setState(() {
    steps = 'Pedometer not\navailable';
  });
}

void initPlatformState() {

  _stepCountStream = Pedometer.stepCountStream;
  _stepCountStream.listen(onStepCount).onError(onStepCountError);

  if (!mounted) return;
}
```

Figure 7.36: Pedometer functionality

7.7 Main Screen

Main screen (main.dart) is extended with a stateful widget. It consists by a text widget and two button at first. If the user's device is Android 10 and above the screen will show two buttons in which the first one is requesting the permission to register physical activity in order to count the daily steps and the second button to open the app settings to manually give the physical activity permission (if the user's device is android 9 or 8 the two buttons won't appear). After the user gives the required permission an Alert Dialog will appear asking for the user to enter his/hers height, daily steps target and the gender. As seen on previous textfields, each textfield requires a TextEditingController with a listener, a function to know if the user interacts with the current textfield for the first time and a function to check the context of the textfield. On the second textfield for example the user must enter a height between 0 and 250cm and requires also the textfield to not be empty. For that we have created the function (see figure 7.37).

```
//Function for displaying the correct error message on height textfield
String? Height_Textfield_check(){
  String height_msg='';
  if(heightController.text.isEmpty==true){
    height_msg='Height can\'t be empty';
    print(height_msg);
    height_check=false;
    return height_msg;
  }
  else if(int.parse(heightController.text) > 250){
    height_msg='Height must be less than 250cm';
    print(height_msg);
    height_check=false;
    return height_msg;
  }
  else if(int.parse(heightController.text) <= 250){
    height_check =true;
    height_msg='Valid height';
    print(height_msg);
  }
}
```

Figure 7.37: Height textfield function

After entering all the requirements a widget called Card will appear on the screen with the daily target of steps and the progress of them in a circular progress bar. The Card uses an elevation option inside the widget to be give the user a more 3d aspect of the widget (the daily steps are saved in a database, more on that topic on database section). On the top right corner there is a target icon which is a button,

when the user presses the button another Alert Dialog will show up to give the option to the user to change the height and the daily steps target. On the bottom of the Card the user can see the kilometers by foot calculated by the number of steps and the height that the user has given. If the device doesn't have a pedometer sensor the package we used named 'pedometer' will try and calculate the number of steps by the motion of the device using the accelerometer and the gyroscope of the device, not very accurate though.

7.8 Navigation Screen

Navigation screen (navigation.dart) is also extended with a stateful widget. When the user enters the screen for the first time if he/she hasn't already given the permission to track real time location before showing the map two buttons will appear, the first one to request permission for real time location and the second one to open app's settings to manually give permission. After giving the permission the map will appear with a button on the left bottom corner. It's important that the user must have enabled the Gps if opening the screen for the first after a full shutdown and be connected to the internet in order for the map to load, due to a bug in initialization of positioning the marker won't show the correct location of the user if he/she enters the screen with the Gps disabled and enable it after he/she enters. For maps there aren't many options if the programmer doesn't want to implement native maps. The first and the most popular choice is to use the package 'google_maps_flutter' (`https://pub.dev/packages/google_maps_flutter`) which is the best package for implementing maps on flutter due to having official support from Google and a special functionality which is camera animation. The biggest flaw of this package is that it requires to get an API key from google maps platform which is free but has specific amount of API calls which can easily be reached, after the limit it is not free. So the next best choice is the package 'flutter_map' (`https://pub.dev/packages/flutter_map`) which is the one that we have used and it is free. It is community based which means that is less supported and the updates are less often. It is a Dart implementation of Leaflet Maps with tiles from open street maps which are also free. This plugin by itself

doesn't provide other functionality other than showing with a small dot where the user is on the map for the first time opening the map. If for example the user moved from the point he/she was before he/she had to select another screen in the app and select again the navigation screen. In order to show on the map every time the user changes location we used the package 'flutter_map_location_marker' (`https://pub.dev/packages/flutter_map_location_marker`) which also has a marker with the heading of the device. To get the changes of position we integrated the package in the FlutterMap widget with a LocationMarkerLayer widget as seen (see figure 7.38).

```
LocationMarkerLayerWidget(
  plugin: LocationMarkerPlugin(
    centerCurrentLocationStream:center_current_location_StreamController.stream,
    centerOnLocationUpdate: center_on_location_update
  ),  // LocationMarkerPlugin
  options: LocationMarkerLayerOptions(
    marker: DefaultLocationMarker(
      color: Colors.green,
      child: Icon(
        Icons.person,
        color: Colors.white,
      ),  // Icon
    ),  // DefaultLocationMarker
    markerSize: const Size(40, 40),
    accuracyCircleColor: Colors.green.withOpacity(0.1),
    headingSectorColor: Colors.green.withOpacity(0.8),
    headingSectorRadius: 120,
    markerAnimationDuration: Duration(milliseconds: Duration.millisecondsPerSecond),
  ),  // LocationMarkerLayerOptions
),  // LocationMarkerLayerWidget
```

Figure 7.38: Location Market Widget

We also used the package 'location' (`https://pub.dev/packages/location`) to get the latitude and longitude as value because the flutter_map_location_marker didn't provide the information we needed but only the representation on the map. To get the location in the foreground or in the background the location package provides a function and all we had to do was to enable it through the lifecycle (as mentioned in the Lifecycle section above) when the app goes to the background and disable it when the app is resumed. We initialiazed the function listen in initState to get the location changes and called a function to save the cordinates and check for the gps status as seen (see figure 7.39, 7.40).

Also we added a button on the right bottom corner in case that the user wanted to navigate the map without losing where he/she is on the map, when the user

```
location.onLocationChanged.listen((loc.LocationData cLoc) {

  currentLocation = cLoc;

  setState(() {
    setpoint(cLoc.latitude, cLoc.longitude);
  });

  insert_toDb();

});
```

Figure 7.39: Function for getting the coordinates

```
void setpoint(latitude,longitude) async{
  serviceEnabled = await geo.Geolocator.isLocationServiceEnabled();

  lat=latitude;
  lng=longitude;
}
```

Figure 7.40: Setter for coordinates

presses the button the screen will show the user current location with the marker in the center of the screen. We also wanted to have a way for the user to see his/hers route on map with a line (called polyline), so we used the package 'flutter_map_tappable_polyline' (`https://pub.dev/packages/flutter_map_tappable_polyline`) which shows with a red line the route that the user has travelled since he/she opened the navigation screen. We draw this line with a list, adding every time the new latitude and longitude every time the user changes one of them. Then we showed the line using the list with coordinates with the package as a widget inside the FlutterMap widget (see figure 7.41).

This package has also a function called onTap in which the user can tap a polyline and in our case print the position of the polyline on map. The user can use the functionalities of the map without connecting to the internet and can see a part of the map that was saved the last time the user used the map with an internet connection. We used the package 'cached _network_image' (`https://pub.dev/packages/cached_network_image`). So we built a function to get an image of what the user sees on screen and we save it based

```
TappablePolylineLayerWidget(
  options: TappablePolylineLayerOptions(
    polylineCulling: true,
    pointerDistanceTolerance: 20,
    polylines: [
      // getP()
      TaggedPolyline(
        tag: 'My Polyline',
        // An optional tag to distinguish polylines in callback
        points: polylineCoordinates,
        color: Colors.red,
        strokeWidth: 9.0,
      ),  // TaggedPolyline
    ],
    onTap: (polylines, tapPosition) => print('Tapped: ' + polylines.map((polyline) => polyline.tag).join(',') +
          'at' + tapPosition.globalPosition.toString()),
    onMiss: (tapPosition){
      print('No polyline was tapped at position' + tapPosition.globalPosition.toString());
    }
  ),  // TappablePolylineLayerOptions
),  // TappablePolylineLayerWidget
```

Figure 7.41: Tappable Polyline Widget

on the coordinates and the options we have set on the map widget (see figure 7.42).

```
class CachedTileProvider extends TileProvider {
  const CachedTileProvider({customCacheManager});
  @override
  ImageProvider getImage(Coords<num> coords, TileLayerOptions options) {
    return CachedNetworkImageProvider(
      getTileUrl(coords, options),
      //Now you can set options that determine how the image gets cached via whichever plugin you use.
    );
  }
}
```

Figure 7.42: Cached Tile Provider function

Basically when the user uses the map with the device connected to the internet this package saves a number of tiles around the user and the map is saved as an image inside the package of the app and the image is kept for 30 days, after that the image gets deleted automatically. This method in order to work requires to built a Cache Manager to save the image, so we used the package 'flutter_cache_manager' (`https://pub.dev/packages/flutter_cache_manager`) in which we can configure the number of objects we want to save and the duration of the data we want to be kept as seen (see figure 7.43).

7.9 Compass Screen

Compass screen (Compass.dart) is extended with a stateful widget due to having changing values. In order to use this screen the user must give the location

```
static final customCacheManager = CacheManager(
  Config(
      'customCacheKey',
      stalePeriod: Duration(days:30),
      maxNrOfCacheObjects: 200
  ),  // Config
);  // CacheManager
```

Figure 7.43: Cache Manager function

permission to the app (we used the permission_handler package mentioned on the permissions section above). As in the navigation screen, when the user enters for the first time on this screen he/she will see two buttons, the first one to ask permission with a native pop up message to give directly the location permission and the second button to direct the user to the app's settings to give the location permission manually. If the user has already given the location permission to the app through the navigation screen then on the two buttons won't show up. After the user gives the location permission four textfields show up and an image of a compass. The first textfield shows the angles from the north, the second textfield show the address that the user is currently, the third one shows the latitude and longitude and the fourth one shows the altitude. In order to get the latitude and longitude the user must enable the Gps on the device. If the user enters the screen without the Gps enabled he/she has to press the button on the top right corner to get the coordinates. In order to get the address and the altitude the user must have an internet connection. Below we can see the screen, on the left when the device has internet connection and an enabled gps and on the right when the device hasn't an internet connection and the Gps is disabled (see figure 7.44).

For getting the coordinates we used the package 'geolocator' (`https://pub.dev/packages/geolocator`). We created a function to get the status of the Gps and also the current position of the user (see figure 7.45).

The function returns a variable of type position which contains information of the current latitude, longitude and altitude. Then to find the current address from the coordinates we used the package 'geocoding' (`https://pub.dev/packages/geocoding`) and then we created the function below to get the address given the

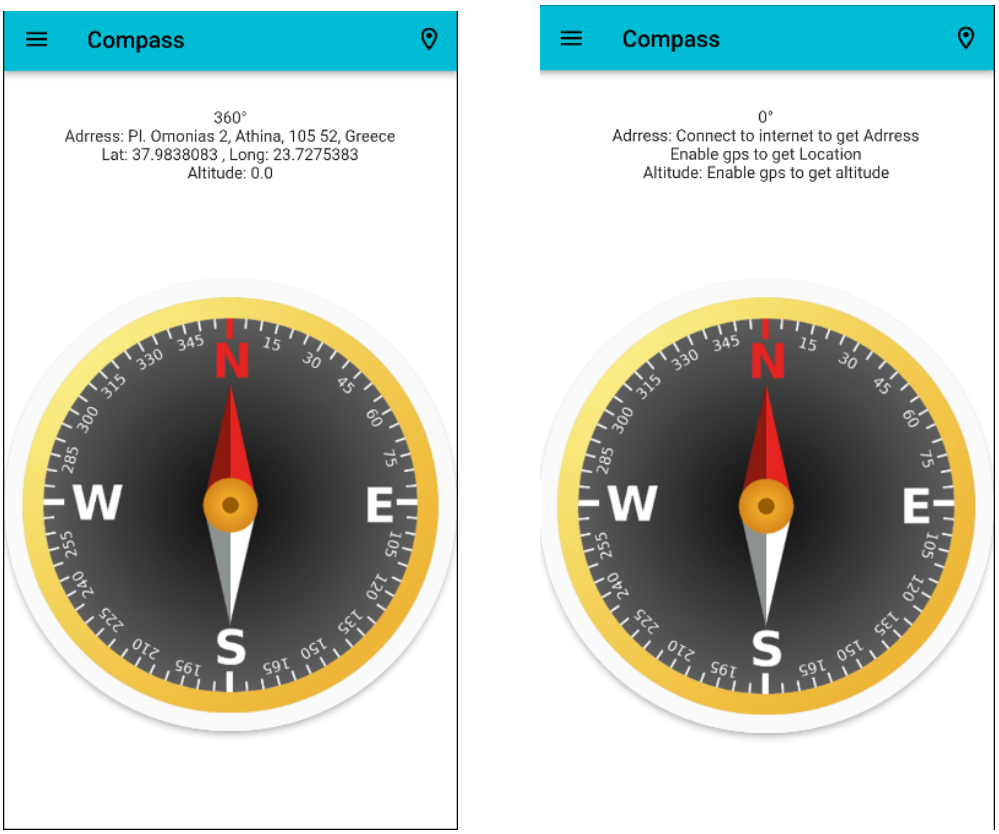

Figure 7.44: Compass Screen with enabled Gps and Internet connection and without

```dart
//Function for getting the status of Gps
Future<Position> getGeoLocationPosition() async {

  serviceEnabled = await Geolocator.isLocationServiceEnabled();

  return await Geolocator.getCurrentPosition(desiredAccuracy: LocationAccuracy.best);
}
```

Figure 7.45: Future for getting the Gps status

coordinates we found with the previous function (we print on the screen not only the address but also the country and the postal code) (see figure 7.46).

```dart
//Function for getting lat lng and
Future<void> GetAddressFromLatLong(Position position)async {
  List<Placemark> placemarks = await placemarkFromCoordinates(position.latitude, position.longitude);
  print(placemarks);
  Placemark place = placemarks[0];
  setState(() {
    Address = '${place.street}, ${place.locality}, ${place.postalCode}, ${place.country}';
  });
}
```

Figure 7.46: Function for finding Address

Then we created the following function to call the two functions mentioned above

and initialized it in initState in order every time the user enters the screen to show the corresponding messages (see figure 7.47).

```
//Function for setting address and location
void getData() async {
  Position position = await getGeoLocationPosition();
  location ='Lat: ${position.latitude} , Long: ${position.longitude}';
  Altitude = position.altitude;

  print('$location');
  GetAddressFromLatLong(position);
}
```

Figure 7.47: Function for getting the coordinates, altitude and Address

For the compass we used the package 'flutter_compass' (`https://pub.dev/packages/flutter_compass`). Just like a physical compass this package uses the magnetometer of the device and shows the angles from north that the device is heading to. The compass is a single image of a compass and the package uses the gyroscope of the device to rotate the image. The package provides a stream of the angles and sets as 0 when the device is heading to north,0-179 when the device is to the right of the north and -180-0 when the device is heading to the left of the north. We didn't like this approach so we created a simple function to show to the user the angles from 0-359 (see figure 7.48).

```
//Function for getting the angles of compass
void get_angle(event) {
  setState(() {
    if(event.heading>0){
      angle = event.heading;
    }
    else{
      angle = event.heading + 360;
    }
  });
}
```

Figure 7.48: Function for showing the angles from 0-360

We integrated the package as widget inside the screen that is build every time the user enters the screen and is initialized in initState function.

7.10 Sensors Screen

Sensors screen (Sensors.dart) is extended with a stateful widget. It consists by 12 texts and in order to align them properly we have 3 rows with 2 columns each. Below each sensor, the user can see the availability of the sensor or if the device has a sensor the data it creates (the functionality of each sensor is mentioned on the Sensors section above). This screen exists only for showing the user the available sensors on the device and their values when an event is occurred (for example when the user spins the device), so the user can't interact with this screen. Below we can see the sensors screen (see figure 7.49).

7.11 Settings Screen

Settings screen (Settings.dart) is extended with a stateful widget. For now it only has a SwitchListTile widget which changes the theme of the whole app from light to dark and vice versa. It has a field which holds a value, that the user can change between the two values by pressing the toggle button (SwitchListTile wiget). To manage the theme, we created the file Theme_provider.dart in which we have the functionality to change the theme of the app. To easily use this file we used the package 'provider' (`https://pub.dev/packages/provider`) which is a wrapper around InheritedWidget. In Theme_provider.dart file we have two classes, in the first one we have a function to set a boolean variable to change between light and dark theme so we can use it in the settings screen and the second class is where we have the light and the dark theme colors for everything we use in the app. The color of the background change from white to dark grey and the color of the text changes from black to white. By default the selected theme is the selected theme that the device is using, for example if a user's device is set to dark theme then when the user opens the app, the app's theme will be also dark. This is done by getting the system's theme in the Theme_provider file and setting it in the main file when we build the whole app, before the user encounters the first screen when opening the app (see figure 7.50).

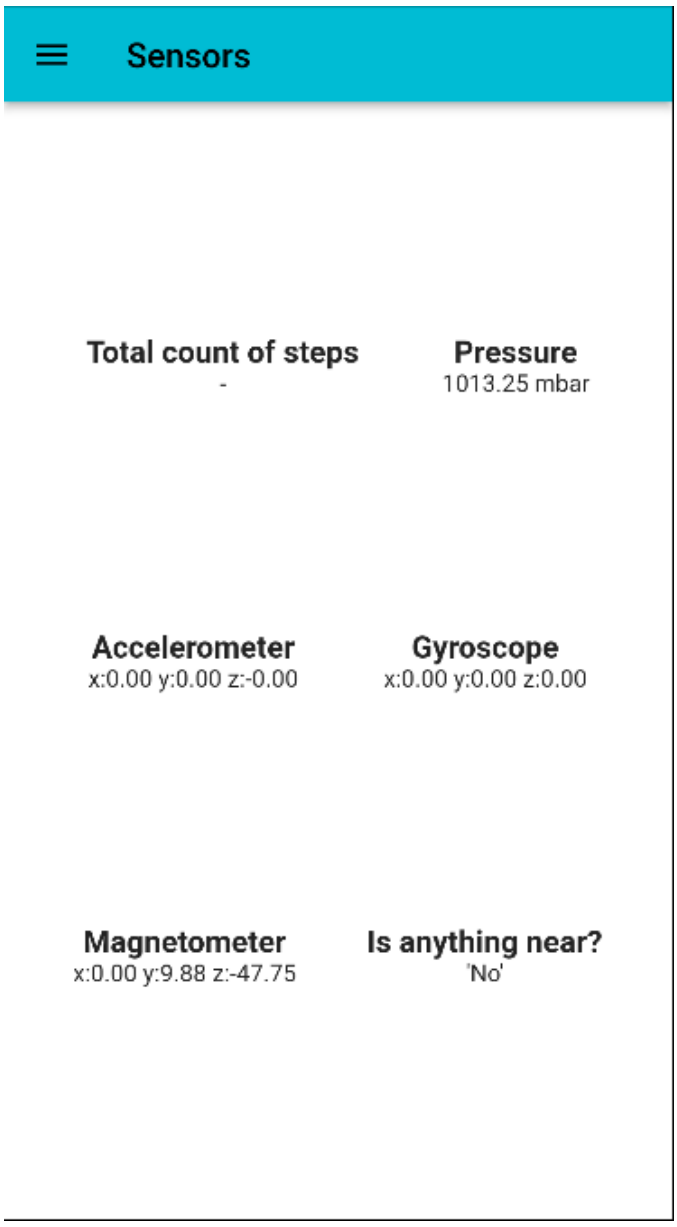

Figure 7.49: Sensors Screen

```
return MaterialApp(
  title: 'Flutter Demo',
  themeMode: themeProvider.themeMode,
  theme: MyThemes.lightTheme,
  darkTheme: MyThemes.darkTheme,
  debugShowCheckedModeBanner: false,
  home:WithForegroundTask(
    child: (saved_mail !=null && saved_pass!=null)? MyHomePage() : Login()
  )  // WithForegroundTask
);  // MaterialApp
```

Figure 7.50: Theme Modes initializing before showing the first page

On these images, on the left we can see the app using light mode and on the right using the dark mode (see figure 7.51, 7.52).

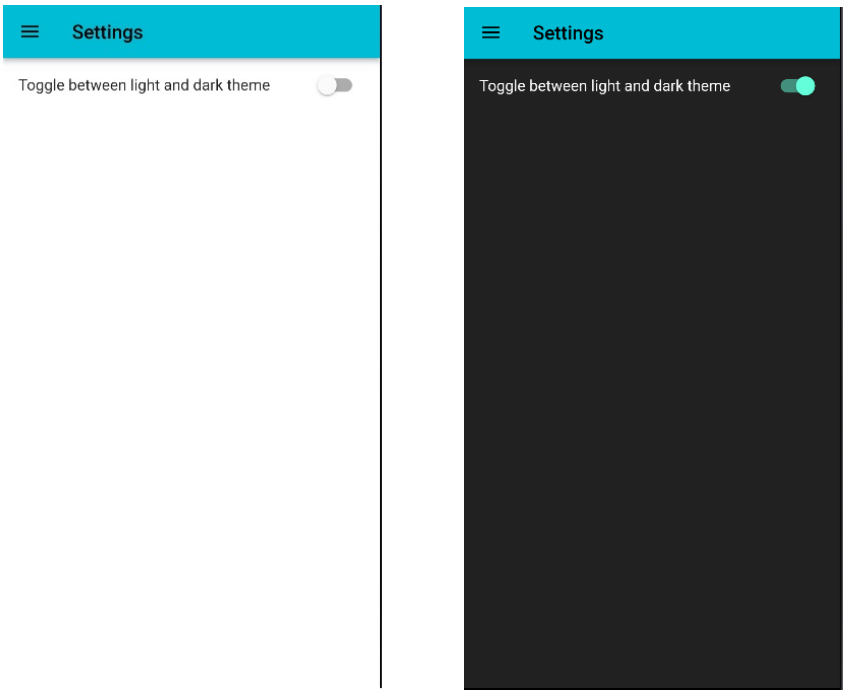

Figure 7.51: Settings Screen with light and dark theme

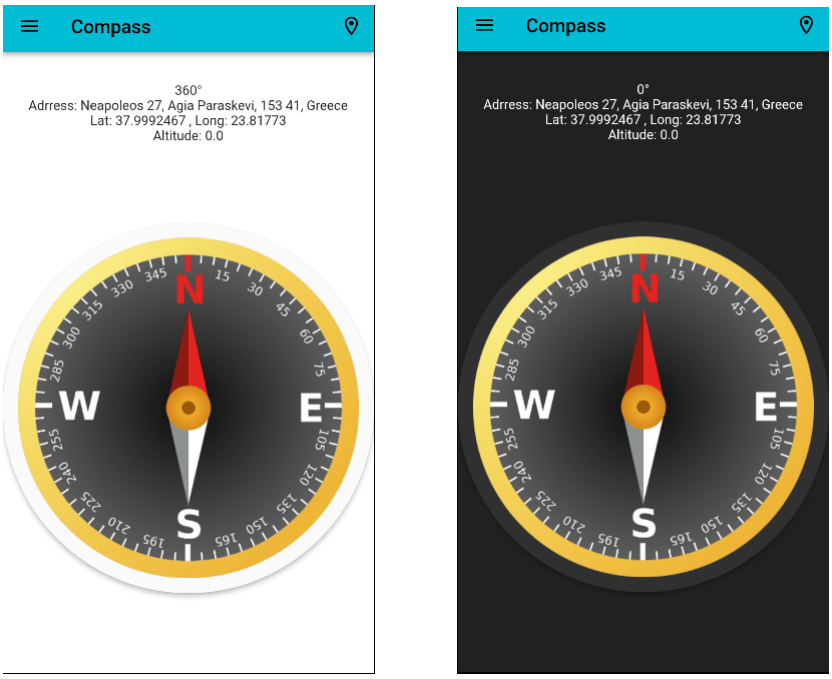

Figure 7.52: Sensors Screen with light and dark theme

7.12 Sidemenu

Sidemenu (Sidemenu.dart) is the only "screen" that is extended with a stateless widget because we don't have a state to change. We chose to use Drawer widget to navigate through our screens as we though it was more convenient and more beautiful for the user. The next option was tabs but because we have multiple screens on the app it would be more difficult for the user to select a tab. To open the sidemenu the user must press the three parallel lines on the top left corner of each screen or slide from the left of the screen to the right to show up (see figure 7.53).

It consists by 2 Texts widgets and 6 ListTiles widgets. In the 2 texts we show the username and the email that the user has used to log in, and the first 5 ListTiles are used to navigate to each one of our screens (Homepage, Navigation, Compass, Sensors, Settings). The last ListTile is used to log out the user from the app and also the database with a pop up dialog, it also terminates the background location functionality of the app. In order for the user to log out he/she must press the log out ListTile and select "Yes" in the dialog, else the "No" option will close the dialog and won't disconnect the user (see figure 7.54).

7.13 Databases

For saving the data that the app creates was the most difficult task we encountered on the creation of the app. We wanted the app to save the data locally on device and when the device is online to synchronize the data to a server that is hosted to our university. We though to use a NoSQL database so we searched for a solution (a package) that provides offline and online database self hosted (not on cloud). The most popular choice that provides the above requirements is ObjectBox (`https://pub.dev/packages/objectbox`). ObjectBox is a database that the programmer can create objects and store them to the database. Objectbox for the local database is easy to install, all we had to do was to install the package and start working. But to synchronize the data to our hosted database we had to request a key with the program from the company in order to work. After requesting the key,

Figure 7.53: Button to open the Sidemenu

we answered to a quick Google Forms questionnaire for the reasons we selected the ObjectBox and why we wanted to use a hosted database and after this form we had answer another form about some personal data and billing address even though the hosted database was free only to realize that this database didn't have built-in authorization and authentication functions. So we searched for other solutions and we found Couchbase lite which is also a NoSQL database that offers the same functionalities but not authentication and authorization. So we decided to take a different approach and have a separate database for saving the data locally and a separate

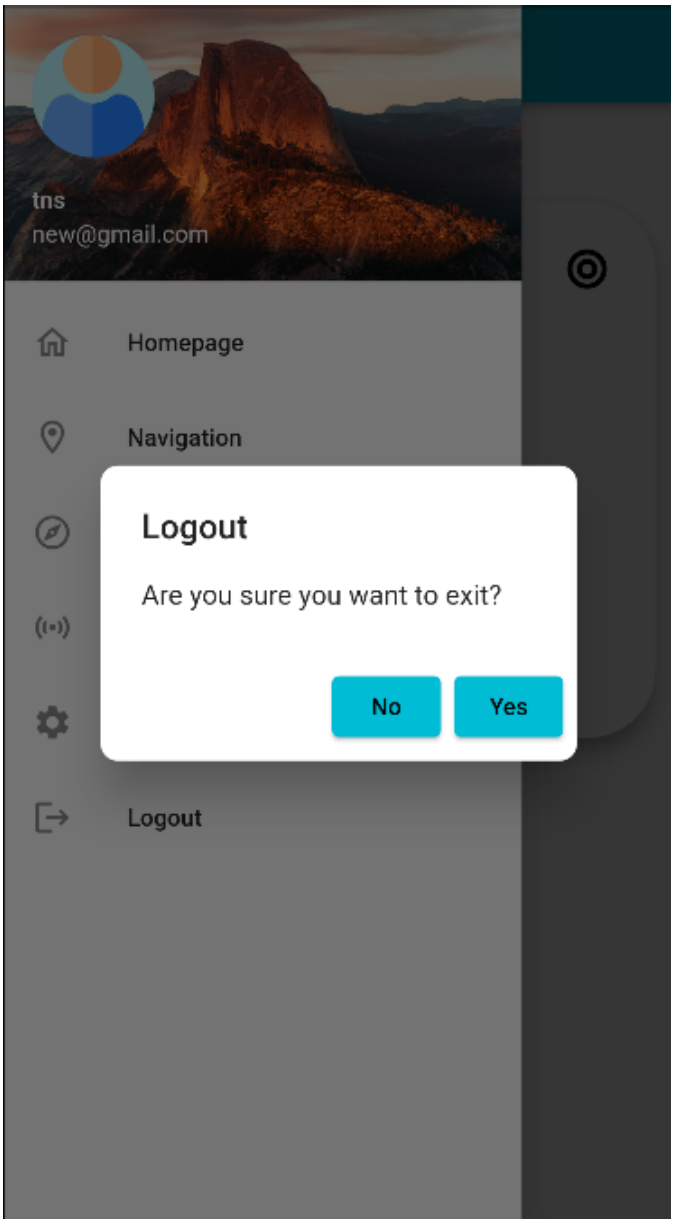

Figure 7.54: Dialog message when the user presses the Log out button

database hosted on the server of the university. So we decided to host a Mongodb database on the server of the university and for the local database we searched for the best solution. First we tried NoSQL databases and we had multiple packages to choose:

- ObjectBox
- Hive
- SharedPreferences

- Sembast
- Couchbase Lite
- Realm

ObjectBox, Hive, Sembast, Couchbase Lite and Realm work similar and support creating objects to save them in their database and all them support CRUD operations. SharePreferences wraps platform-specific persistent storage for simple data (NSUserDefaults on iOS and macOS, SharedPreferences on Android). Data may be persisted to disk asynchronously, and there is no guarantee that writes will be persisted to disk after returning. So we decided to use Hive as it is the most lightweight and is written in pure Dart. The idea of the Hive database is simple, the programmer creates a box with a name and put data to the box with the function put(). Then in order to retrieve the data the programmer must use the function get(). The data can be either variables, list, maps and objects. In order to save objects to the database programmer must generate a type adapter. The update operation is only available if the programmer extends the object with HiveObject. When we started using the Hive database we noticed that the update operation of the database isn't the same like an SQL database, because if for example we added a list to a box, the new list we added overwrites the previous one and we wanted to keep the data from the first time the user registers to the app until every time the user enters the app again. So we used Hive only for some basic tasks, like saving the email and the password of the user to keep a session, so the user every time he/she closes the app and opens it again to be redirected to the main screen without the need of re-typing the credentials, for saving the target of steps and the daily steps and for saving the user's height and gender. To check if the user has already logged in to the app without logging out by pressing the "Logout" button in the Sidemenu, we checked before running the main function of the app if the "email" and "pass" fields in the box are empty to redirect the user to appropriate screen. Below we can see the function to check if the is an open session (see figure 7.55, 7.56).

So we searched for available SQL packages and we found Drift, Floor and Sqflite. Drift is built on top of Sqlite, Floor is an abstraction of Sqlite and Sqflite is Sqlite

```
void main() async{

  //Hive commands for initializing and opening a box to store data
  await Hive.initFlutter();

  box = await Hive.openBox('user');

  check_session();

  runApp(MyApp());
}


void check_session() async{
  saved_mail = await box.get('email');
  saved_pass = await box.get('pass');
}
```

Figure 7.55: Function for checking session

```
return MaterialApp(
  title: 'Flutter Demo',
  themeMode: themeProvider.themeMode,
  theme: MyThemes.lightTheme,
  darkTheme: MyThemes.darkTheme,
  debugShowCheckedModeBanner: false,
  home:WithForegroundTask(
    child: (saved_mail !=null && saved_pass!=null)? MyHomePage() : Login()
  )  // WithForegroundTask
```

Figure 7.56: Checking for credentials before redirecting the user to the appropriate screen

package for Flutter. We ended up using Sqflite as it is the most stable package. We used SQL database for saving the coordinates of the user, for saving data from the sensors (pressure, acceleration, gyroscope, magnetometer, proximity) every 10 seconds, altitude every 5 seconds and pedometer once per day. We have created a file (SqlDatabase.dart) containing all the SQL statements (including creation of tables, insertion to tables and selection from tables) and the required functions for the database like the creation of the database file (db.db) and the initialization of it. We have 8 tables for saving the data we need (coordinates, altitude, pressure, acceleration, gyroscope, magnetometer, proximity, daily_steps). The daily steps are

inserted into the SQL database once per different day, we save the date of the last date and we check if the date of the last step is the different than the current day, if it is we insert the steps into the database with the current date. Of course in order for this function to work properly we execute it before we start counting the current's date steps. Below we can see the function of checking the dates which is in initState function and the function to insert into the database table (these functions are inside the main.dart file)(see figure 7.57, 7.58).

```
if((box.get('date') != date_once) && (box.get('date') != null)){
  insert_toDb();
  box.put('today_steps',0);
}
```

Figure 7.57: Function for checking the date in comparison to the registered date

```
insert_toDb() async{
  int stp = box.get('today_steps');
  await SqlDatabase.instance.insert_daily_steps(date,stp,0);
  List<Map> lista = await SqlDatabase.instance.select_daily_steps();
  print(lista);
}
```

Figure 7.58: Insertion of daily steps to SQL database

For saving the coordinates the strategy was much simpler, we decided to save the coordinates to the database every time the user changes position, so this was done in the function we have the listener for changing position as mentioned on the navigation screen above (the function are inside the Navigation.dart file)(see figure 7.59).

For saving the altitude again the strategy was simple, we decided to save the altitude every 5 seconds to the database using a timer (this was done inside the Compass.dart file). We followed the same thinking with saving the data from the sensors, we checked for the availability of them and then saved the data every 10 seconds (the function used on the image below is inside the Sensors.dart file in initState function)(see figure 7.60).

```
location.onLocationChanged.listen((loc.LocationData cLoc) {

  currentLocation = cLoc;

  setState(() {
    setpoint(cLoc.latitude, cLoc.longitude);
  });

  insert_toDb();

});
```

Figure 7.59: Insertion of coordinates to SQL database

```
timer = Timer.periodic(Duration(seconds: 10), (Timer t) {

  if(acc_check == true){
    insert_acc_toDb();
  }
  if(gyro_check == true){
    insert_gyro_toDb();
  }
  if(magn_check == true){
    insert_magn_toDb();
  }
  if(press_check == true){
    insert_pressure_toDb();
  }
  if(prox_check == true){
    insert_prox_toDb();
  }
  //check();

});  // Timer.periodic
```

Figure 7.60: Insertion of the sensors data to SQL database

Chapter 8

Conclusions and Future Work

8.1 Conclusion

In our days, a high percentage of the people have a smartphone both Android and IOS and want to know the their daily movement. In the present thesis, a complete responsive user's tracking application was developed, which includes all the necessary functions that a user needs to register all of the movement on a daily basis. The reality is that we didn't give much attention to the design of the User's Interface, we did what we could to be as nice as possible, given that we aren't User's Interface Designers but we emphasized on the convenience and usability. During the development of the app we encountered some major issues which were solved like making the app work when the user doesn't interact with it or finding the best solution for storing data both locally and on a server. So the first goal of the thesis which was developing an app by scratch was achieved and the second goal which was connecting the app with a database was also achieved. There are some things that could have been better solved but given the time that we had the results are pretty good. Even though the thesis journey has come to an end the app will grow as seen on the section below.

8.2 Future extensions

I am happy to announce that the thesis will be funded for a short period of time to be extended adding more functionalities and making it more polished. Below we can see some some rough thoughts for future expansion:

- Add functionality to count the daily user's route distance
- Add animations to the app
- Fill Settings Screen with more options for example to change the user's username or password
- Find and fix bugs that may or may not be present in the future due to the use of the app by a large number of users, as its extensive use will lead to the detection of any bugs.

References

[1] https://en.wikipedia.org/wiki/Apache_Cordova

[2] https://en.wikipedia.org/wiki/Ionic_(mobile_app_framework)

[3] https://en.wikipedia.org/wiki/Flutter_(software)

[4] https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin

[5] https://blog.logrocket.com/xamarin-vs-flutter/

[6] https://docs.flutter.dev/resources/architectural-overview#architectural-layers

[7] https://developer.android.com/guide/components/services

[8] https://www.flaticon.com

[9] https://github.com/flutter-moum/flutter_all_sensors/blob/master/lib/all_sensors.dart

[10] https://protocoderspoint.com/flutter-sensors-plugin-accelerometer-and-gyroscope-sensors-library/

[11] https://www.youtube.com/watch?v=PMcXhYmFFN4

[12] https://www.youtube.com/watch?v=voARoVV_EDI

[13] https://codewithandrea.com/articles/flutter-text-field-form-validation/

[14] https://flutter.dev/docs/cookbook/forms/text-field-changes

[15] https://fluttercorner.com/textfield-validation-in-flutter/

[16] https://flutter-examples.com/check-textfield-text-input-is-empty-or-not/

[17] https://www.youtube.com/watch?v=CUM5OIw9zDI

[18] https://github.com/nhandrew/platformcode

[19] https://brightmarbles.io/blog/platform-channel-in-flutter-benefits-and-limitations/

[20] https://medium.com/flutter/flutter-platform-channels-ce7f540a104e

[21] https://protocoderspoint.com/flutter-get-current-location-address-from-latitude-longitude/

[22] https://www.codegrepper.com/code-examples/whatever/geolocation+street+address+flutter

[23] https://morioh.com/p/832f968ed090

[24] https://www.youtube.com/watch?v=pDnfJvh7uzc

[25] https://www.youtube.com/watch?v=XIxahpXU_QE

[26] https://www.youtube.com/watch?v=y4_x6Ss8gOE

[27] https://www.youtube.com/watch?v=v2QGS4UqaqA

[28] https://api.flutter.dev/flutter/material/ToggleButtons-class.html

[29] https://www.lewisgavin.co.uk/Step-Tracker-Android/

[30] https://www.youtube.com/watch?v=ttSuRvs36Bc

[31] https://www.youtube.com/watch?v=HSAa9yi0OMA

[32] https://api.flutter.dev/flutter/material/SwitchListTile-class.html

[33] https://github.com/fleaflet/flutter_map

[34] https://github.com/transistorsoft/flutter_background_fetch/blob/master/help/INSTALL-ANDROID.md

[35] https://www.youtube.com/watch?v=6wvD-Z-9ZRM

[36] https://github.com/tlserver/flutter_map_location_marker

[37] https://www.youtube.com/watch?v=otWy4QhMJMo

[38] https://objectbox.io/flutter-databases-sqflite-hive-objectbox-and-moor/

[39] https://greenrobot.org/news/flutter-databases-a-comprehensive-comparison/

[40] https://blog.codemagic.io/choosing-the-right-database-for-your-flutter-app/

[41] https://levelup.gitconnected.com/top-5-local-database-solutions-for-flutter-development-6351cd494070

[42] https://www.youtube.com/watch?v=ckXSR79AACg

[43] https://www.youtube.com/watch?v=w8cZKm9s228

[44] https://github.com/fleaflet/flutter_map

[45] https://www.youtube.com/watch?v=JyapvlrmM24

[46] https://www.youtube.com/watch?v=dB0dOnc2k10

[47] https://www.youtube.com/watch?v=HloDTrNH37c

[48] https://www.youtube.com/watch?v=rHJxHSJY-8k

[49] https://pub.dev/packages/proximity_sensor/example

[50] https://pub.dev/packages/email_validator/install

[51] https://pub.dev/packages/flutter_compass

[52] https://pub.dev/packages/cupertino_icons

[53] https://pub.dev/packages/font_awesome_flutter

[54] https://pub.dev/packages/flutter_map_tile_caching

[55] https://pub.dev/packages/background_fetch

[56] https://pub.dev/packages/crypto

[57] https://pub.dev/packages/cached_network_image

[58] https://pub.dev/packages/provider

[59] https://pub.dev/packages/connectivity_plus

[60] https://pub.dev/packages/internet_connection_checker/example

[61] https://pub.dev/packages/sqflite

[62] https://pub.dev/packages/flutter_foreground_task

[63] https://pub.dev/packages/flutter_map

[64] https://pub.dev/packages/mongo_dart/install