VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Ilia Maslovskii

SENNO SMART RING APPLICATION

ANDROID APPLICATION DEVELOPMENT FOR WEARABLE DEVICES UTILIZING BLE TECHNOLOGY

Information Technology
2019

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Ilia Maslovskii
Title	Senno Smart Ring Application
Year	2019
Language	English
Pages	95
Name of Supervisor	Jukka Matila

In recent years, with increasing sophistication of smartphones and the advent of wearable electronics, it has become easier to have a healthier lifestyle by being conscious of constant changes in your body. However, despite a plethora of devices which provide limited health monitoring capabilities, few allow for more than a few of relevant readings, and the devices are either imprecise, obtrusive or not easy to use.
The objective of the thesis was to develop a companion application to the Senno Ring, a device which provides non-intrusive health monitoring using LED technologies. The end goal of the application was to avoid common pitfalls of the competitors, while having modern design, providing ease of use and allowing for clear visualization and interpretation of the data transmitted by the ring.

The application was implemented primarily by using Java and Android-specific frameworks, excluding BLE data transfer, which was built on RxJava2. The data is collected in the Firebase Cloud services for storage, ease of access and further analysis, using NoSQL Database Structure.

The thesis was done for Delektre Ltd., to coincide with the release of the Senno ring prototype. During the thesis work, prototype application with core functionality was successfully developed, with it being ready for deployment on devices running Android 4.0 and higher.

Keywords Reactive, Java, Android, Firebase, Bluetooth, Wearables

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

XML	Extensible Markup Language
BLE	Bluetooth Low Energy
GATT	Generic Attribute Profile
UI	User Interface
UX	User Experience
RX	Reactive
UUID	Universally Unique Identifier
LED	Light-Emitting Diode
IDE	Integrated Development Environment
NoSQL	Non-Structured Query Language
SQL	Structured Query Language

1 INTRODUCTION

With the advancements of wearable technology, it has been expected that wellness-oriented self-monitoring devices would become readily available. Indeed, while devices such as fitness trackers and sleep monitors are abundant on the market, most of the solutions provided are limited in their scope or do not provide comprehensive analysis of the client's wellbeing, despite there being no technological roadblocks for more sophisticated devices.

Delektre Ltd., the main supervisor of the thesis, has taken upon themselves to develop a new type of non-intrusive measurement system called Senno Ring, which measures body vitals, and the companion application, which provides analysis for the data received from the ring, and presents it to the customer in a clear, concise way, showing the client most important reading, giving it clear representation in terms of current metabolic state and providing notifications if vitals deviate from the norm.

Usages of the device and application may vary greatly – from personal day-to-day usage to maintain a healthy lifestyle and a way to monitor changes in the body during physical activities, to being an additional monitoring tool for the doctor or being given to a retired relative for automated check-ups. As no concrete use-case was selected, the application should have contained code and layouts that could be reused, or tweaked for more concrete use cases.

It was decided that the application was to be designed in a way to be scalable, to have cloud storage capabilities and to have modern UI in order to provide solid user experience and to be competitive on the market. This thesis demonstrates development of Android-based Java application which was supposed to be demonstrated alongside the Senno Ring in the late 2019/early 2020. The project must contain all core application functionality in order to be considered successful.

My personal goals to be achieved during thesis work are learning Android development processes, building experience by participating in a real working life project and reinforcing knowledge received during my studies at VAMK.

1.1 Delektre Ltd.

The following information is taken from the company's LinkedIn page:

"DELEKTRE LTD offers consultancy, research and development services for business and institutional customers. We have broad experience on software, hardware and marketing of enhanced and completely new products. We build bridges between research ideas and commercial world.

We cooperate with leading European research institutions and we are networked around the world from SMEs to conglomerates. Our own products include devices, such as a smart ring with exceptional measurement capabilities.

Our employees are experienced professionals who are always glad to try new challenges." /1/

1.2 Possible Competitors

This section contains commercial products available that, at least partially, share the market space and purpose that Senno Ring might occupy. Although Delektre Ltd. aims to create a comprehensive personal wellness monitoring device, some of the products available contain functionalities that might be implemented in Senno Ring, either fully or as parts of a larger solution.

1.2.1 Motiv Ring

Founded in 2013, Motiv is a breakthrough wearable technology company focused on designing products that fit seamlessly into people's lives and keep them living better. Motiv creates products people want to wear, are easy to use and deliver meaningful experiences. Motiv's first product, Motiv Ring, is sold internationally, in Europe, Asian, the Middle East, and Australia and New Zealand. The Motiv App is available in both the App Store and on Google Play. /2/

Motiv Ring is an equivalent of a fitness bracelet in a ring form. In addition to tracking fitness progress, it measures the quality of sleep. Having the ability to evaluate the ring

in person, it was discovered that the measurements were imprecise, and that measurements were not sent in a real-time mode – instead, the application and the device were synchronised either manually, which required flipping the ring over on your finger, or at fixed time intervals when the phone application was opened. Furthermore, at the moment of writing of this thesis, the mobile application was barely usable, as it was almost impossible to pair the ring and the application. However, the popularity of the Motiv Ring indicated that there is a real market demand for this type of device.

1.2.2 Oura Ring

Oura Health Ltd. is a Finnish health technology company was founded in 2013. Oura is the world's first wellness ring and an app that shows how your body responds to your lifestyle by analyzing your sleep, activity levels, daily rhythms and the physiological responses in your body. Personalized for you, Oura guides you towards better sleep, recovery and readiness to perform. Oura has users in over 70 countries, and several top universities, research organizations, sleep clinics, and companies are utilizing the data and insights Oura provides. Oura Health Ltd.'s HQ and major manufacturing facilities are located in Oulu, Finland. Other locations include Helsinki and San Francisco. /3/

Oura Ring, while being capable of fitness tracking, focuses on keeping track of sleep health. It demonstrates how technologies that enable fitness tracking, with proper research, could be used for different purposes. After using Oura Ring for a week for market research, it was noted that it can track sleep phases semi-reliably – there were some false-positives during periods of low physical activities of the user. The mobile application provided a wealth of sleep-related data, giving access to user's sleep history, highlighting different phases of sleep and sleep disturbances. Like Motiv Ring, it only synchs periodically and when the application is opened. User interface and user experience can be considered outstanding. Research into Oura Ring shows the importance of visual presentation, design coherence and benefits of a clear product vision, as indicated by the company's success.

2 RELEVANT TECHNOLOGIES AND TOOLS

2.1 Java

Being initially developed for interactive television by James Gosling, Java is a general-purpose computer language that is concurrent, class-based and object-oriented. It is designed to be able to run on all platforms which support Java Virtual Machine (JVM), which allows for Java programs compiled to bytecode to run without recompilation. It allows for greater cross-platform development. The Java syntax is similar to C and C++, it is one of the most used programming languages despite its age and is free under GNU General Public License. Java is the primary language used to develop Android applications. /4/

2.2 RxJava

RxJava is a Reactive Extensions implementation for JVM. It is a library which provides developers with the ability to handle events and asynchronous fluidly by providing additional high level of abstraction. It implements observable/observer pattern, meaning that an object called observable keeps the list of recipients (observers), which it notifies on the state changes as specified by the developer's implementation. RxJava allows holding easily maintainable one-to-many relationships by delegating responsibility of updating to the observers. In other words, observable is not aware of its observers.

2.2.1 RxAndroid

RxAndroid is platform-specific addition to the RxJava library. It enables easier development by providing schedulers which account for Android application lifecycle and threading. More specifically, it allows for observing on main thread for effective scheduling and event handling, replacing traditional AsyncTask. /5/

2.2.2 RxAndroidBle

RxAndroidBle is a Java library implementing RxJava and RxAndroid. Its main usage is to handle Bluetooth Low Energy discovery, management and data transfer inside the ap-

plication by using reactive patterns and event-based programming, as well as error-handling and threading. RxAndroidBle makes Bluetooth Low Energy applications clean to read and easy to implement and maintain by reducing the amount of boilerplate code necessary. /6/

2.3 Firebase

Firebase is a platform and a framework provided by Google for swift application development, desktop, web and mobile alike. It provides unified login system, data storage, database and introduce cohesion and enable cross-platform development. More importantly, it reduces amount of work done by developer by providing ready-made back end solutions and eliminating the security concerns. The core services provided by Google Firebase are Storage – secure online cloud storage with flexible plans for emerging entrepreneurs and established businesses alike, Firebase Auth, a secure way to enable login and sign-up that could be easily integrated into Google ecosystem, online databases, of which there are Firestore and Realtime Database, which could be chosen upon particular enterprise or personal needs, and Firebase Hosting for web applications. /7/

2.4 Android Studio

Android Studio is a primary IDE for creating Android application. It is made by Google and is based on JetBrains' IntelliJ IDEA IDE and was designed specifically for Android development. Like most modern IDEs, it provides syntax-specific autocompletion and error detection (dependent on the language of programmer's choosing), as well as Android specific refactoring and implementation of typical function overrides and templates. It simplifies UI development by providing rich XML editor and integrates Google Cloud and Firebase platforms. Android Studio supports Gradle build system out-of-the-box and supports Kotlin, Java and C++ languages on installation, with plugins available to extent its capabilities. Android Studio implements Android device emulation and supports various methods of version control, including GIT. /8/

2.5 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a Bluetooth-based technology intended for use with application and devices for healthcare, fitness, security and others, encompassing a wide array of devices which require low maintenance and little to none interference from the user. Its communication range is close to Bluetooth, although bandwidth and power consumptions are severely reduced. Despite it being a relatively new technology, 90% of smart devices have hardware necessary to support it. Bluetooth Low Energy operates on Generic Attribute Profile (GATT), which means that there is always a client, typically a smart device which starts GAT commands, a server, device receiving commands and requests, characteristic, which is a data transferred between client and a server. BLE implementations have their own Universally Unique Identifier (UUID). They should be chosen semi-randomly. The list of reserved ones is available online at Bluetooth website /9/

2.6 Vital Signs Measurement

Vital signs (also known as vitals) are a group of the four to six most important signs that indicate the status of the body's vital (life-sustaining) functions. These measurements are taken to help assess the general physical health of a person, give clues to possible diseases, and show progress toward recovery. /10/ Senno Ring should, ideally, be capable of precise vital signal measurement. It is supposed to use different sensors for different vitals. During the thesis implementation and the device development, only temperature sensor and in-house built pulse oximeter, a device which illuminates the skin and measure differences in light absorption. /11/ While measured temperature was easily convertible to Celsius degree, due to the availability of public drivers for the sensors used and because of the straightforward nature of the measurement, data from the assembled pulse oximeter, which contained four LEDs of green, red, yellow and blue color, as well as one LED of infrared variety, lacked any way of meaningful interpretation. This is because of the lack of research done on pulse oximeter data analysis at the time of development by Delektre. Ideally, once the devices have been calibrated and big enough data sample has been gathered, it would be possible to measure breathing rate, pulse and hydration level. This is possible by taking LED reflected light value of an individuals with normal vitals and comparing them to the actual reflected light value.

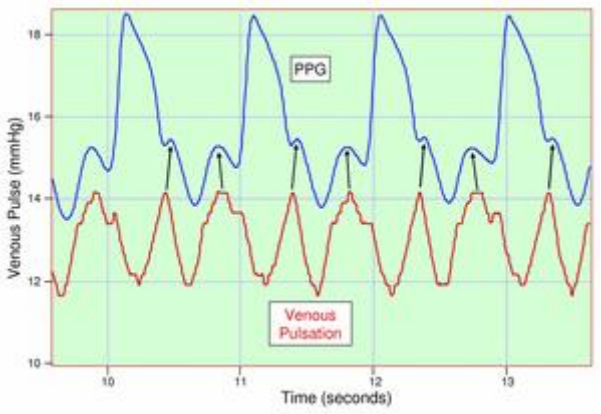


Figure 1. Example of Venous Pulsation Measurement Obtained Using Pulse Oximeter

During the development it was discussed that by storing large quantities of such data, it is possible to display personalized health trends for the user. If the development ever reaches that stage, contextualized health-related notifications could be send to the ring user, or their healthcare provider. Similar technology is utilized in, for example, Apple Watches.

3 APPLICATION REQUIREMENTS

The main purpose of the application is to provide user with a clear and concise way to see the data received from the Senno ring. At this stage of the development, and as far as this project is concerned, the only non-user generated data that is shown is raw data, meaning the only non-user generated data displayed is readings of various LED and sensors which are present on the ring itself. The user generated data takes form of various personal data, such as height, weight, and age, which, in the next versions of the application, will be used to calculate various biometric and vital readings, including metabolic state and hydration.

The application UI was inspired by Google Material Design, in particular the Google Fit Android app, as well as the closest similar products available on the Play Store, which are Motiv, Oura and Samsung PEAR. The design is made to be minimalistic but with personality touches which serves to distinguish it from the competitors.

The principle idea is to have one main screen which serves as a dashboard for the most important data, while also having additional screens which provide information which is not considered vital and immediate for the user. The screens with additional information are accessed through the sliding panel.

3.1 Application Overview

The structure of the application is depicted in the use case diagram. It showcases relationships between different parts of the application and how outside actors interact with it. Use case diagram depicts actions, not necessarily functions and methods and therefore, each user case will be examined in detail in order to facilitate the design of UI. It helps with dividing codebase into the classes and methods which implement the cases and provides a roadmap for the development.

The user case diagram displays how the user enters the application – via the login activity which requires Firebase connection in order to function – and how he/she accesses various application parts. The data transfer between the wearable device and the application is shown, along with the cloud data storage. Actions such as viewing user profile are demonstrated. This section of the thesis expands on the user case diagram by showing the UI which was developed based on the diagram.

The structure of the application is depicted in the use case diagram. It showcases relationships between different parts of the application and how outside actors interact with it. Use case diagram depicts actions, not necessarily functions and methods – therefore, each user case will be examined in detail in order to facilitate the development of UI.

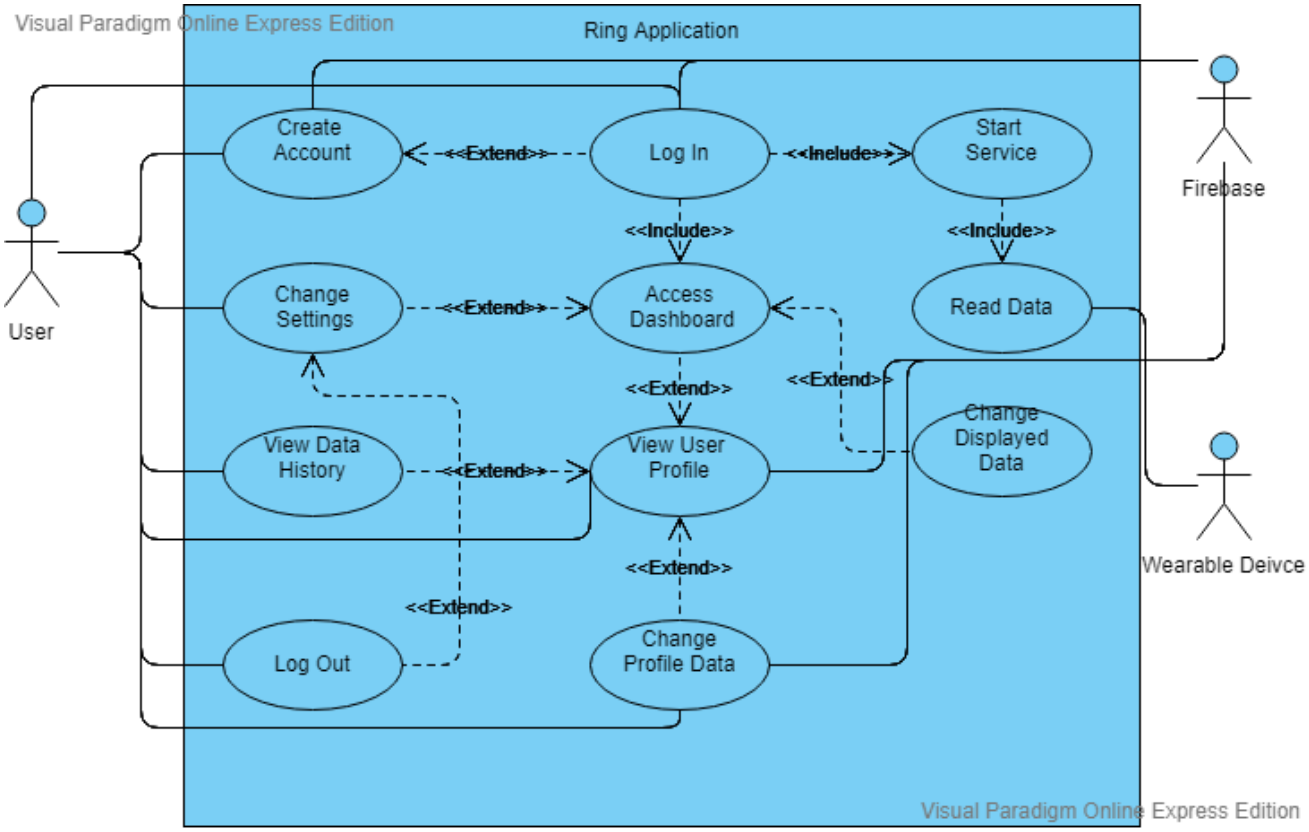


Figure 2. Use Case Diagram.

3.2 Application Functions

3.2.1 Account Creation

The main purpose of the account creation screen is to create unique user credentials in order to enable Firebase capabilities and to gather the user's biometric data in order to create a swift user experience. As this project serves as a backbone for further develop-

ment, the data is not used in a meaningful way (e.g. to enhance the analysis and interpretation of values received from the wearable device), however, the implementation is necessary to show this application's potential capabilities. Account creation should include multiple screens, such as the introductory screen (which is shared with the login part of the application) that collects the desired email and password, and, depending on satisfying the conditions set during development, either moves the user forward or displays the error message on pressing "Create Profile" button. It also includes the application logo.

Following that, the application requests first name, surname and gender from the user. Two buttons should be present – "Back", which would allow the user to cancel account creation, and "Next", used to move forward in the account creation process. The button also allows for the user to upload a profile picture from their phone. It will not allow the user to proceed further unless all fields are filled, excluding profile picture. Lastly, the creation screen asks for the weight, height and age. The screen should change the measurement units based on user input. Again, two buttons should be present – "Back", to allow user to change any of the data and to move out of account creation process if necessary, and "Create", which will complete the account creation and will automatically log in the user and enter the main dashboard of the app. User data is then stored in the Firebase Database and could be changed by the user in other activities. Their measurement unit preference is stored there. The application should not allow the user to complete the registration if all of the data has not been entered and it should display error message in such cases.

3.2.2 Login Screen

Login screen should serve the purpose of letting the user enter the main body of the application. It should be the first screen that the user sees, and it will provide clear information on what is required of the user, and how to proceed next. Login screen is also used to instigate the account creation process. It must contain two editable text fields, "Email" and "Password" with hints, and two buttons, "Login" and "Create". Upon pressing either of the buttons, if any of the input information is absent or not sufficient, e.g. the password strength is low or the email is not correct, the visual prompt corresponding to the error is displayed. Otherwise, the application should launch activity which corresponds to one of the buttons and store the current authorization token which is maintained between all of the application activities and fragments. Data persistence should be enabled, and if the

user has logged in once, the login screen should be skipped, and the first screen that the user will access must be the dashboard.

3.2.3 Dashboard

On the dashboard initialization, the background service should be launched. The service is responsible for sharing the data between different activities, looking for available BLE devices and BLE communication between the phone and the ring, sending the measurements to the application. The measurements are stored in the Firebase Database under the unique user ID

The dashboard is the application's main page, which should be the primary screen which is used when interacting with the application. It should have the most important information clearly displayed. In case of the target version of the application, it must have a button which should be used for the navigation between the secondary and tertiary activities, a top bar with the application logo and (if present) the user's profile picture, buttons to change the most commonly used settings, and the ability to see the data received from the ring in a real-time fashion. It should be presented as one view with a graph displayed that show fluctuations between values and which cuts off at both maximum and minimum values, and which will erase past measurements after a certain point to increase performance and improve readability. The metrics received should not be displayed at the same time – for each type of measurement, a dedicated graph should be created, with the ability to change between them on a button press. A carousel-style view shall be used for those buttons, and, as the current version of hardware provides limited measurements, only buttons for temperature, water balance, pulse and blood pressure should be present. When pressed, the graph view is repopulated with the corresponding data type measurements. Depending on the hardware developments, a button which connects/disconnects/scans for the device might be present.

3.2.4 Setting Screen

The Settings screen, this being a tertiary activity, should be accessed through the dashboard. It is represented as a standard Android-type settings screen, being a scrollable list. At this stage, the options menu should be limited to placeholder buttons for sending the

commands to the wearable device and to the option menu which would allow logging out of the current user account. Android software button “Back” should take the user to the previous activity, which, according to the application structure, is always the dashboard.

3.2.5 User History

User history is a secondary screen which aggregates historical data for the user. It should take measurements from the Firebase Database, sort them according to the selected option, and display them as a graph. It should be able to display the measurements for the current day, last week, and last month. To display this data in s clear manner, this should be implemented by using three tabs corresponding to different periods of measurements. Additionally, a way to change between different type of data must be present. Daily measurements use accumulated data from the last 24 hours and aggregates them to hourly averages. Weekly and monthly graphs display the average daily value over a week and moth, correspondingly. When the Android software “Back” button is pressed, the user should be taken to the dashboard.

3.2.6 User Profile

User profile is a secondary screen which lets the user adjust the personal data that was provided during the account creation. It should contain editable surname, first name, gender, weight, birthday and height fields. Moreover, the user should be able to change his profile picture, either by taking a new photo, or by selecting a picture from the picture gallery. The profile screen should allow the user to change the measurement units for the relevant fields, and to display them alongside with the values to avoid confusion. If any of the field are empty upon user trying to get back to the previous screen, the corresponding error message should be displayed, and the user should be forbidden to proceed. The picture taken by the user should be stored in the cloud and on the device to increase load time and to enhance offline capabilities.

3.3 Application Mock-Ups

The following section displays mock-ups of the application’s activities. The mock-ups were created with consideration for the application requirements., and have been used to create views for the Activities and Fragments.

3.3.1 Dashboard

The application's dashboard is the main view with which a user interacts with the application. Figure 2 demonstrates that the dashboard is a view that contains access to other application sections via the hamburger icon, top bar which includes the current Senno logo and a user's profile picture, and a real-time graph display, controlled by buttons which prescribe the type of information to be displayed via the graph.

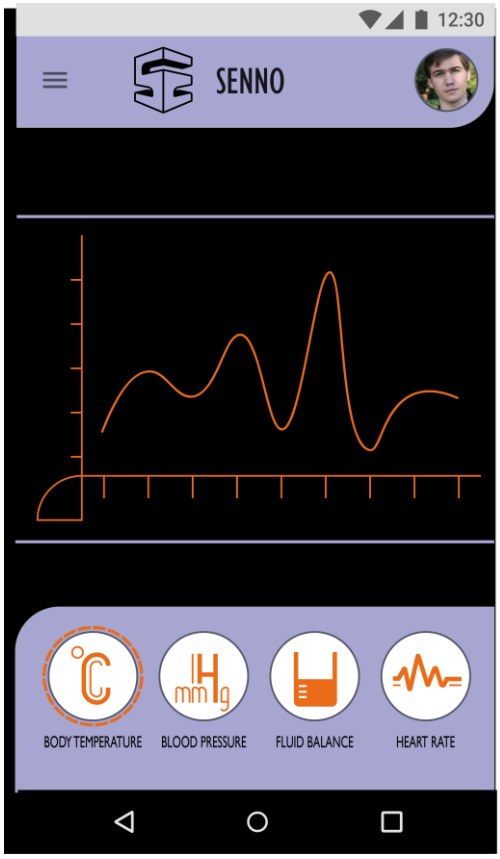


Figure 3. Dashboard Mock-Up

3.3.2 Navigation Panel

The navigation panel, as shows in Figure 3, should contain a sliding view with a list of available views. It is accessed via the hamburger icon on the top left of the dashboard. Most other views should not have access to the navigation panel, as at most the perform tertiary functions. Upon pressing currently active one, no view change should occur, and the sliding panel should move back. Currently active view, as long as the navigation panel is active, should be inaccessible and have gray transparent overlay, to signify that navigation panel is in focus. Optionally, correlated icons could be displayed along the views present in the list.

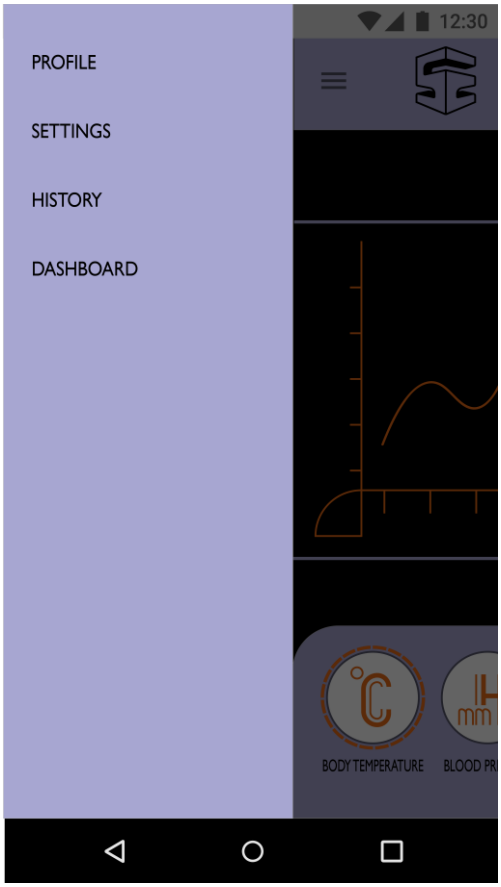


Figure 4. Navigation Bar Mockup.

3.3.3 Profile View

Shown in Figure 4, the profile view consists of profile picture, and field to see and edit name, surname, as well as height, weight, age and birthday. As the final product must rely

on the user's body metrics, they should be easy to access and readily available to be edited. This view should not contain any unnecessary information and be as clear as possible, as shown in Figure 4.

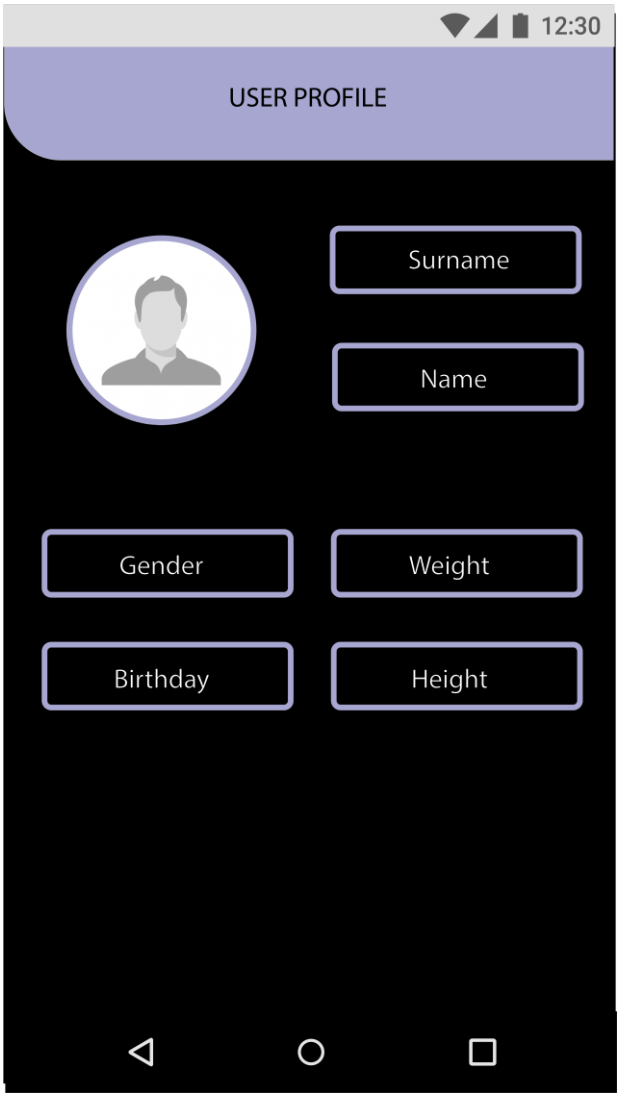


Figure 5. Profile View Mockup.

4 APPLICATION DESIGN

4.1 Project Structure

Senno app is a native Android application, and it follows the common project structure. Each Android application consists of a package with folders for each type of file extension. The root package path contains build-tools related files with“.graddle” extension, IDE settings folder, build-tools wrapper folder “gradle”, and the application folder. In the application folder, the files are stored according to the default Android Studio parameters, with customization necessitated by Senno application specifics. As such, Java class files are organized based on their functionality and uses, as illustrated by Figure 5.

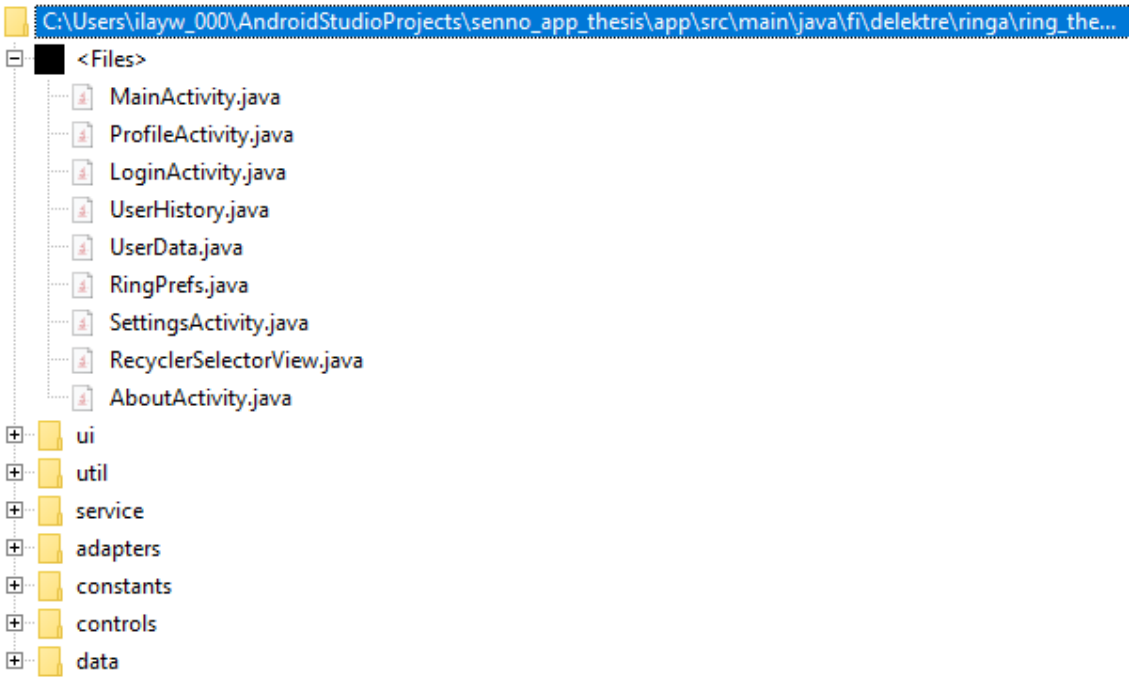


Figure 6. Senno Project Composition

The application uses Activities, which are single, focused user interaction contained within an abstraction in Android /12/, as a main tool of implementation, and uses Fragments (pieces of application interface and behavior that work concurrently with Activities and maintain their own lifecycle /13/) sparingly, when applicable. Usage of constantly-running Android Services /14/ is required in order to receive data even when user leaves the application environment.

All Java files representing Activities are stored within the root folder. Majority of accessible Activities implement Firebase API either for data storage or for user authentication.

The Activities follow their own lifecycle, which is demonstrated in Figure 6. Lifecycle represents callbacks performed on different Activity stages. In Senno App, actions such as Firebase instance declaration and variable assignments, along with view setup, is done in “onCreate()” callback. Scanning for the wearable device is tracked through the whole lifecycle.

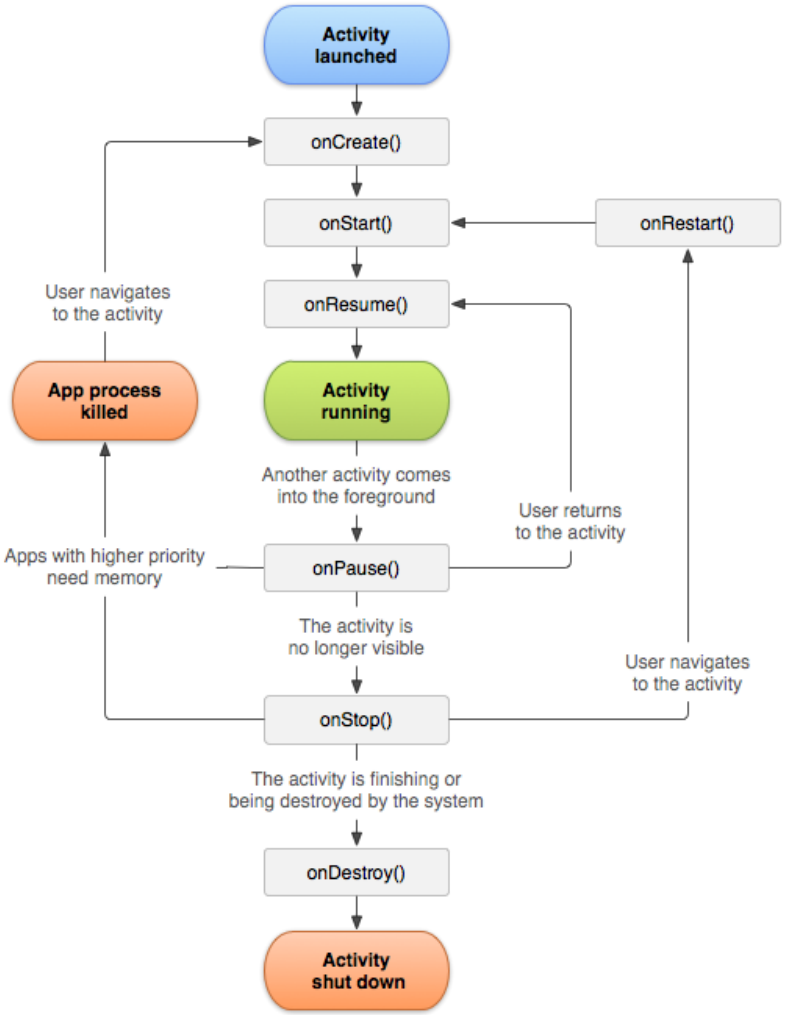


Figure 7. Activity Lifecycle

4.1.1 Fragment Usage

While Fragments are used sparingly to reduce application bloat and complexity, their presence allows to provide the user with expected modern UI flow. Fragments are applied when the app performs small, focused action which is contained on itself but provides data for the larger parts of application (Activities). In particular, Fragments are used in the account creation activity and in user profile activity, to set and read user values, store them temporarily and send them back to the parent Activity and/or Firebase Database. Fragments follow their own lifecycle that is dependent on the Activity lifecycle, which is shown in Figure 7.

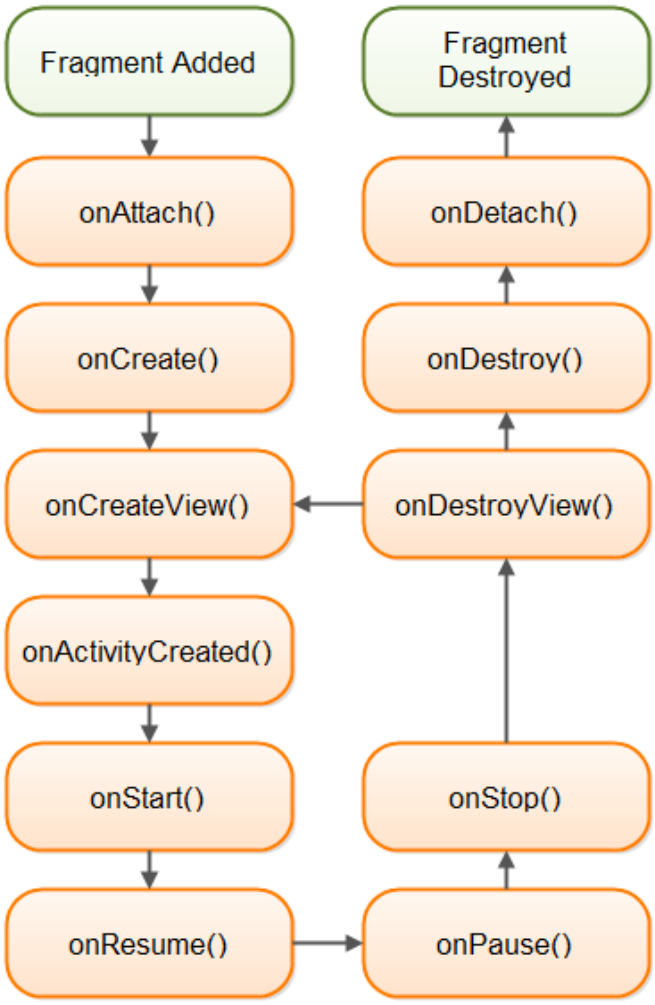


Figure 8. Fragment Lifecycle

4.2 Class Diagram

Unified Modelling Language (UML) class diagram demonstrates project structure. It shows class structure and provides an outlook on how these classes interact with each. The differences between a typical Java project and an Android application can be seen in Figure 10, where there is comparitevely few classes in relation to the number of methods in each of them. This is because in Android development, Activities and Fragments are tied to the classes, and majority of the development consists of invoking and instantiating Android system methods that are not included in the package file. Hence, they are not displayed in the UML diagram. Furthermore, when the diagram was created, because of the View declarations and system object instances, the number of variables present has

increased dramatically. This is why class variables have been ommited during the making of UML diagram.

Each of the arrows going from and to a class signifies one of the relationships that they have. An explanation on how to read the diagram is provided in Figure 9.

- Inheritance show classes that inherit from each other, the arrow direction represents the superclass.
- Aggregation shows which classes are contained within which class.

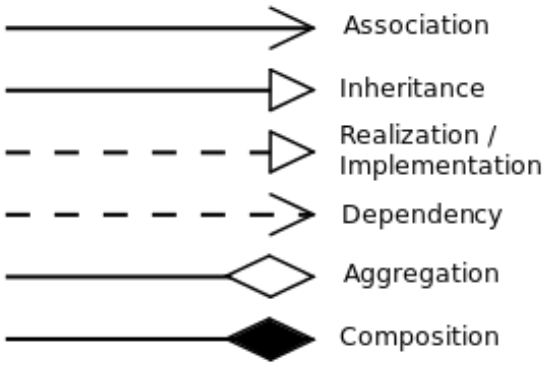


Figure 9. UML Diagram Legend

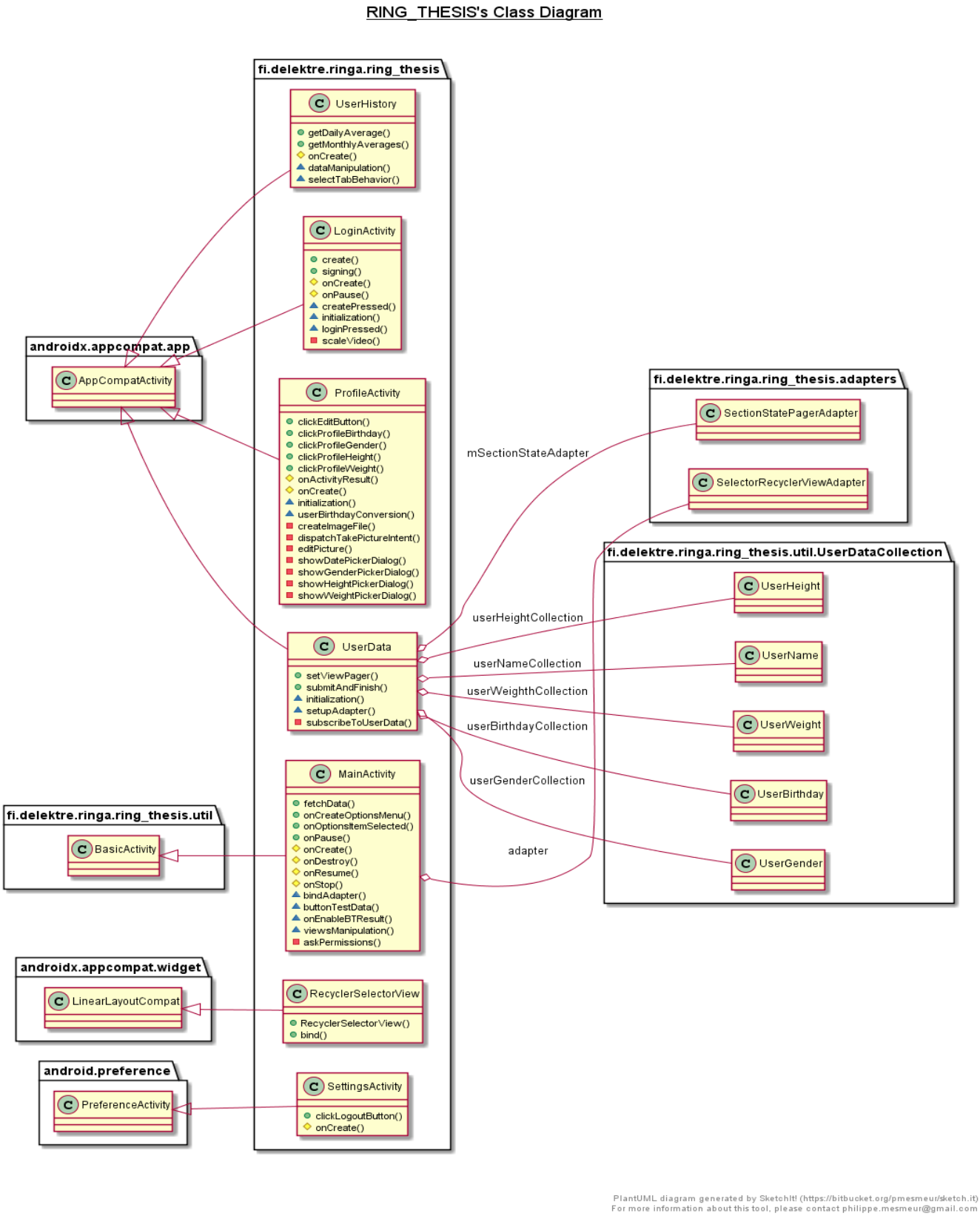


Figure 10. Project UML Diagram

4.3 Data Management

Due to the application complexity, data management and inter-application communication is split into three parts – Cross-Activity communication, used for transmitting data between different Activities that allow user to interact with the said data, Service communication with the hardware over BLE connection using RxJava, to relief the application from the strain of two-way real-time communication with the hardware and to handle the data received in orderly fashion, and Google Firebase, which handles the back-end and stores the received data.

4.3.1 Cross-Activity Communication

As the project implements RxJava as a RxAndroidBle dependency, it was decided to create an event-based system for sharing data. It shall implement Observable-Subscriber model, with all Fragments, Services and Activities which emit data that needs to be transferred acting as Observable. Other Observables and functions which return data would push it onto a single channel, while Subscribers will be able to subscribe to certain types of data, and, whenever the data of certain type would be emitted by the channel, necessary methods would be executed. Figure 11, taken from the official ReactiveX website /15/, demonstrates typical implementation of Observable-Subscriber method.

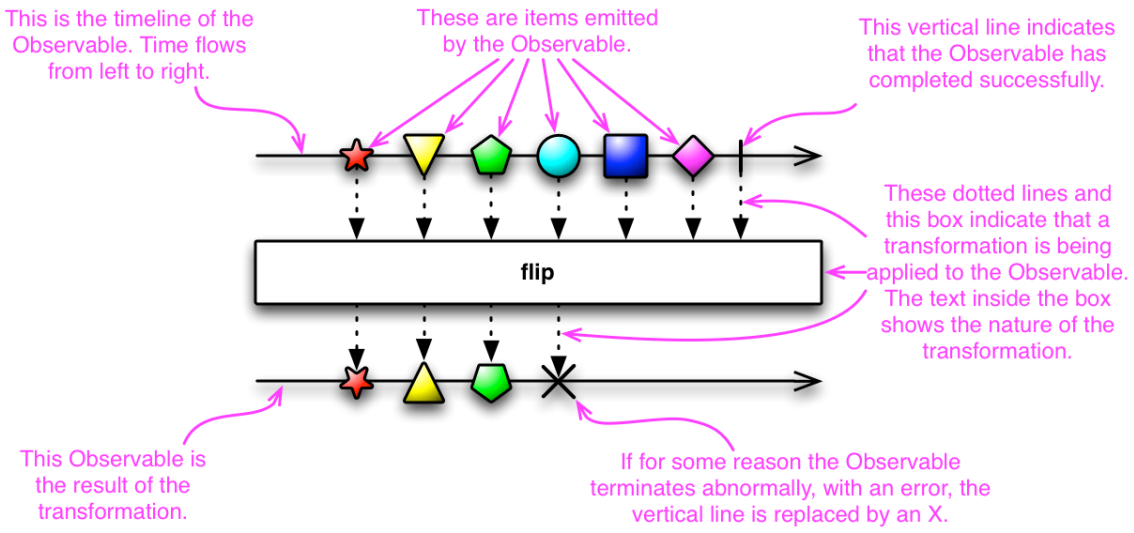


Figure 11. Observable-Subscriber Model

4.3.2 Service Communication

Service communication in Senno app, in a similar fashion to RxJava-based bus system for cross-activity communication, is based on Observable-Subscriber model. Service is required for real-time continuous data receival from the wearable device, in the form of byte arrays. Inside the byte array, wearable commands, identifier and sensor values are encoded. The data model has been designed beforehand, and its Java representation is demonstrated in Figure 12. RxAndroidBle library will be used for the implementation. This helps to reduce boilerplate code and helps to manage proper responses to different byte packages. Once the connection has been established, the RxAndroidBle instance acts as Subscriber, watching over BLE connection, and as Observable, emitting the processed data to different methods and Activities.

```
//Takes every second byte in the array
//And if it is less then 0, shift it to a positive value
//Add all those values to the array
int shiftedBytes[] = new int[8];
for (int i = 0; i < 14; i++){
    if (i%2 == 0){
        shiftedBytes[i/2] = bytes[i];
        if (shiftedBytes[i/2] < 0){
            shiftedBytes[i/2] = shiftedBytes[i/2] + 255;
        }
    }
}
int zero1 = (int) bytes[0];
int redS = (int) bytes[1] << 8| shiftedBytes[1];
int greenS = (int) bytes[3] << 8| shiftedBytes[2];
int blueS = (int) bytes[5] << 8| shiftedBytes[3];
int nirS = (int) bytes[7] << 8| shiftedBytes[4];
int yellowS = (int) bytes[9] << 8| shiftedBytes[5];
int tempByte12 = bytes[12];
if (tempByte12 < 0){
    tempByte12 = tempByte12 + 255;
}
int temp1S = (int) bytes[11] << 8| shiftedBytes[6];
int temp2S = (int) bytes[13] << 8| shiftedBytes[7];
int steps = bytes[15] | (bytes[16] << 8) | (bytes[17] << 16) | (bytes[18] << 24);
int zero2 = bytes[19];

int byte11 = bytes[11] << 8;
int byte12 = tempByte12;
int byte12shift = byte12 << 8;
int byte13 = bytes[13];
int byte14 = bytes[14];
int byte14shift = bytes[14] << 8;
```

Figure 12. Code Snippet for Byte Array Data Mapping.

Additionally, once a certain time has elapsed or specific amount of measurements have been received, Service should communicate with Firebase Database and send the data to it, for safekeeping and further analysis. Service should be responsible for scanning for BLE device, reconnection and handling BLE-related errors.

4.3.3 Google Firebase

Google Firebase has been chosen as the method for backend handling, for its accessibility for prototyping, ease of use, and integration into Android ecosystem. In addition, Firebase allows for user identification using email or social media. As Firebase Database is a NoSQL-type database, it is preferable to the SQL-based databases because of the scalability – with the projected amount of data sent by each of the users for the duration of connection between smartphone and the wearable, loosely-coupled nature of NoSQL da-

tabases allows to only get relevant data from it, mitigating backend cost, user traffic expenditure and Android system resources. Constant data uploading ensures data persistence and will eliminate necessity to sync the wearable device with the backend using a premade schedule. The Firebase Database and Firebase Authentication setup is done through Firebase Console. The database should contain two collections – one for user personal data and preferences, and one for data received and decoded from the wearable device. Both would create entries which would use the Firebase Authentication UUID, ensuring weak data linking.

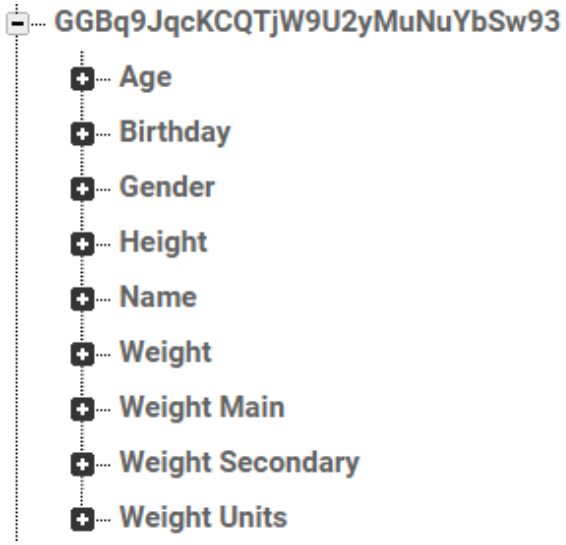


Figure 13. User Information Database Entry

The reading-type entries contain interpreted data from the device, stored as entries where the main key is the timestamp of the device at the moment of data transmission.

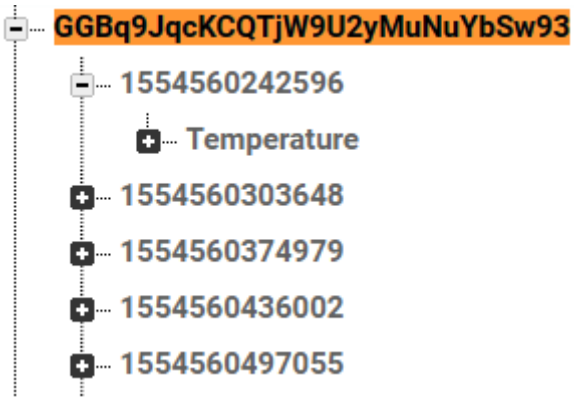


Figure 14. Readings Database Entry

4.4 Layout Design

The layouts define the structure for a user interface for an application component, such as Activity. It is done by creating XML files, which contain a hierarchy of View and Viewgroup objects /16/. Android Studio provides the developer with ability to use drag-and-drop tools for creating the layouts, along with a preview of how a layout would look on a mobile device. Using the Login Activity layout as an example, Figure 15 and Figure 16 demonstrate the actual code for the Login button along with its representation in the drag-and-drop constructor and preview.

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <com.google.android.material.button.MaterialButton
        android:id="@+id/login_create"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Create"
        app:cornerRadius="90dp"/>

    <com.google.android.material.button.MaterialButton
        android:id="@+id/login_enter"
        android:layout_width="match_parent"
        android:layout_marginTop="8dp"
        android:layout_height="wrap_content"
        android:layout_below="@+id/login_create"
        android:text="Login"
        app:cornerRadius="90dp"/>

</RelativeLayout>
```

Figure 15. Code Snippet of Login Activity “Login” Button Representation

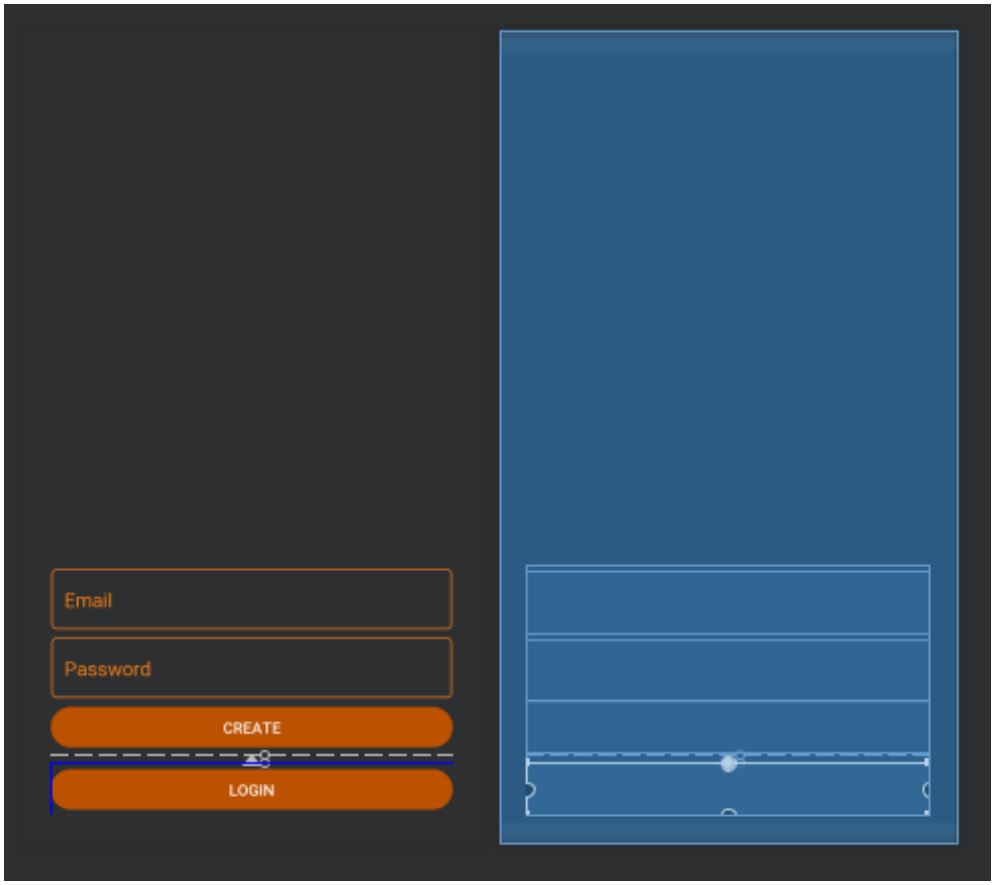


Figure 16. Layout Representation in Android Studio Editor, along with a Preview.

Layouts could be imported into one another. This property is relevant for this thesis project, as several Activity classes will extend from a superclass and have shared UI elements and similar hierarchy. Furthermore, this property, if utilized, lets keep the code more readable and manageable. The import of one layout into another is done using "<include>" tag, as demonstrated in Figure 17.

```
<!-- include main content -->
<include layout="@layout/layout_main_screen"
    android:fitsSystemWindows="true"
    />

<!-- include appbar -->
<include
    layout="@layout/layout_main_appbarlayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Figure 17. Demonstration of "<include>" tag, using Main Activity Layout.

4.5 Data Visualizing

For data visualizing, MpAndroidChart is used. MpAndroidChart is an Android library for efficient charting. It supports all commonly used chart types and allows extensive customization with writing boilerplate code. Smooth rendering of up to 10 000 data points and ease of use, as well as its light size and small impact on performance has been deciding factors upon selecting the library. It accommodates most of the graphing needs in the application. /17/

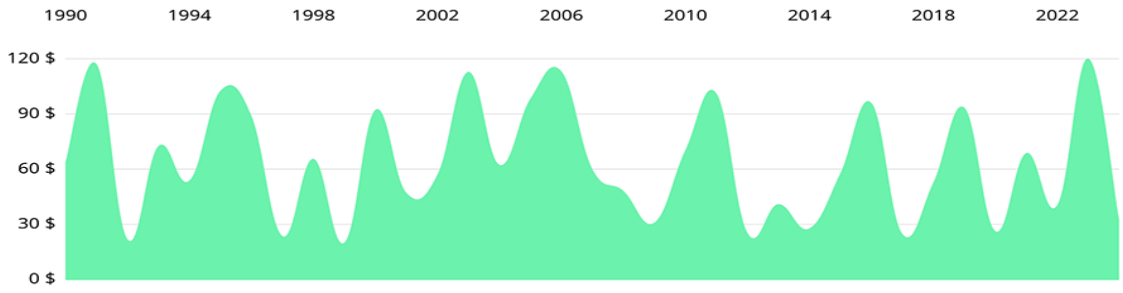


Figure 18. Example of Chart Rendered with MpAndroidChart.

4.6 Dependency Injection

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A "dependency" is an object that can be used, for example as a service. Instead of a client specifying which service it will use, something tells the client what service to use. The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it. The service is made part of the client's state. /18/ In Senno app, AndroidAnnotions library is used for dependency injection in order to reduce boilerplate code, make it readable and to better manage Activity lifecycle. AndroidAnnotations achieves it by generating classes based on bindings on compile. The typical usage of AndroidAnnotations is show in Figure 19.

```
@ViewById(R.id.cardview_main_1)
protected CardView cardView;
```

Figure 19. Example of AndroidAnnotations Usage Replacing Native Android View Declaration.

5 IMPLEMENTATION

5.1 Setup

The development process requires related tools. Android Studio has been installed and all needed SDKs and official libraries have been added after the installation has been complete. Following that, a Firebase account has been created. Once Senno app Android project was created, it was added to the Firebase Console. The process entails creating Firebase Project on the overview, entering application ID (package name), which is “fi.delektre.ringa.ring_thesis” and clicking “Register”. It generates “google-services.json” once the application has been registered, which is then imported into Android Studio project. Following that, the application appears in the Firebase dashboard. From there, Firebase Database and Firebase Login are enabled, as shown in Figure 20.

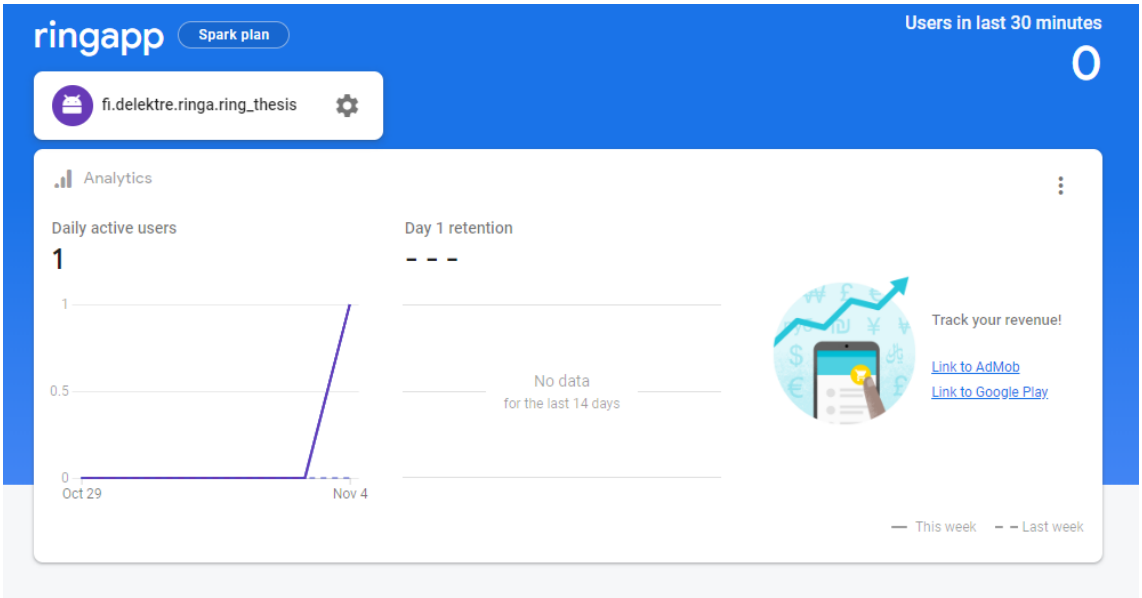


Figure 20. Application Setup in Firebase.

5.2 RxBus

RxBus is a public final class that is used as a basis for communication between activities. It replaces Interfaces, is more robust than SharedPreferences, does not hang up application because every task is processed asynchronously via Subscriber-Observable model, and does not depend on a specific data or object type. It is implemented using RxJava

library. Whenever a part of the application needs to transfer data to any other part, the method “RxBus.publish()” is called. Then, asynchronous listeners are set up on a thread. Those listeners, once initialized, are waiting for a specific type of data or object of specific type to be pushed onto the RxBus. Once it is observed, the correct method is called.

```
public final class RxBus {
    private static PublishSubject<Object> bus = PublishSubject.create();

    public RxBus(){

    }

    public static Disposable subscribe(@NonNull Consumer<Object> action){
        return bus.subscribe(action);
    }

    public static void publish(@NonNull Object message) { bus.onNext(message); }
}
```

Figure 21. RxBus Implementation.

5.3 Utility Classes

Utility class section covers all the classes that are not directly tied to their corresponding activities. In Senno application, they are used either as a superclass for other classes, for data encapsulation or for code snippets that are repeatably used throughout the application. Whenever a class is presented, package and library imports are omitted, as well as View bindings done with AndroidAnnotations. Sparsely used functions which overload parts of Activity lifecycle are verbally described. Due to the scope of the projects, not all parts of the implementation are highlighted.

5.3.1 BasicActivity.java

BasicActivity.java class is used as a superclass (parent class from which children classes inherit constructors, methods and variables) for the MainActivity.java class. It was implemented as a proof of concept for having Navigation Drawer persistent between different activities without using Android Fragments. The class contains a list of activities which user can access through the sidebar, and overloads for typical Navigation Drawer actions. Due to usage of Android Annotations, all Activity classes should be addressed

by calling the classes generated by Android Annotations. Once Navigation Drawer is accessed. Active View is moved to the side proportionally to the length of Navigation Drawer. Initialization of the Navigation Drawer is show in Figure 22.

```
drawerToggle = new ActionBarDrawerToggle((Activity) this, drawerLayout,  openDrawerContentDescRes: 0,  closeDrawerContentDescRes: 0)
{
    @Override
    public void onDrawerSlide(View drawerView, float slideOffset) {
        super.onDrawerSlide(drawerView, slideOffset);
        float moveFactor = (drawerList.getWidth() * slideOffset);
        frameLayout.setTranslationX(moveFactor);
    }

    public void onDrawerClosed(View view)
    {
        getSupportActionBar().setTitle("Ring App Thesis");
    }

    public void onDrawerOpened(View drawerView)
    {
        getSupportActionBar().setTitle("Ring App Thesis");
    }
};
drawerToggle.setDrawerIndicatorEnabled(true);
drawerLayout.addDrawerListener(drawerToggle);

getSupportActionBar().setDisplayHomeAsUpEnabled(true);
getSupportActionBar().setHomeButtonEnabled(true);
```

Figure 22. Code Snippet for Navigation Bar Initialization.

The following code snippet, Figure 23, demonstrates how onNavigationItemSelected events are bound to the Activities. In this case, and subsequentially throughout the thesis, the option button (text label) is addressed using its “android:id” property, which is assigned in the corresponding Layout .xml file. Then the Activity is started using Intent and startActivity(). The current Activity name is set upon Activity launch. This is possible because of Java polymorphism, as all overrides are inherited by the child classes. Activity names are cleared up after they are cleared. If the current Activity is chosen, Navigation Drawer is closed instead.

```
drawerList.setNavigationItemSelectedListener(new NavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem menuItem) {

        switch (menuItem.getItemId()){
            case R.id.nav_history:
                if (currentClassName.equals("UserHistory_")){
                    drawerLayout.closeDrawers();
                    break;
                } else {
                    Intent intent = new Intent(getApplicationContext(), UserHistory_.class);
                    startActivity(intent);
                    break;
                }
```

Figure 23. Binding of startActivity() Method to an Navigation Drawer Item.

5.3.2 AppBarTransparentScrollingViewBehavior.java

This .java file is responsible for Main Activity's AppBar transparency. It was needed because of the Android issue with non-rectangular source images used in the AppBars in combination with ScrollViews, which resulted in transparent parts of source image being either colored black or overlapping the content of scroll view. This is fixed by changing vertical offset of the ScrollView content manually, by setting it to be equal to the width of the AppBar. The code shown in Figure 24 achieves this by extending native CoordinatorLayout behavior. It converts display pixels (dp) to actual pixels (px), and sets the offset to proper amount of pixels.

```
    private boolean updateOffset(CoordinatorLayout parent, View child,
                                 View dependency) {
        final CoordinatorLayout.Behavior behavior = ((CoordinatorLayout.LayoutParams) dependency
                .getLayoutParams()).getBehavior();
        if (behavior instanceof CoordinatorLayout.Behavior) {
            // Offset the child so that it is below the app-bar (with any
            // overlap)

            final int offset = -(int)dpToPx(100);

            setTopAndBottomOffset(offset);
            return true;
        }
        return false;
    }

    public float dpToPx(float dp){
        float px;
        Resources r = ctx.getResources();
        px = TypedValue.applyDimension(
                TypedValue.COMPLEX_UNIT_DIP,
                dp,
                r.getDisplayMetrics()
        );
        return px;
    }

}
```

Figure 24. Code Utilized to Fix Android UI Glitch.

5.3.3 FirebasePersistence.java

Firebase persistence is a simple class that should extends Application class and replaces it inside the Senno app. It is used to enable the FirebasePersistance, as it has to be done before first instance of Firebase is created, otherwise, fatal crash occurs. FirebasePersistance allows to store FirebaseDatabase data locally until the smartphone successfully connects to the Internet. Figure 25 shows Firebase persistence initialization.

```
public class FirebasePersistence extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        /* Enable disk persistence  */
        FirebaseDatabase.getInstance().setPersistenceEnabled(true);
    }

}
```

Figure 25. Setting Firebase Persistence.

5.3.4 ChartUtils.Java

A separate class dedicated to methods that manipulate data charts. During the course of development of the project, only one method was included into the class – removeOutdatedEntries(), which handles popping data points from the data sets. It is necessary to make the application faster - due to large size of the data being added when the measurements get sent back to the smartphone, it is unreasonable to store all of them in memory. It also makes data shown to the user clearer to process.

```
public class ChartUtils {
    private static final int MAX_ENTRIES = 50;
    public static void removeOutdatedEntries(DataSet... dataSets){
        for (DataSet ds : dataSets){
            while (ds.getEntryCount() > MAX_ENTRIES){
                ds.removeFirst();
            }
        }

    }
}
```

Figure 26. ChartUtils Implementation.

5.3.5 DataType.Java

This class is used to check the type of user data that is passed from Fragments corresponding to user input to Activities and Firebase Database instance. Whenever an instance of UserDataCollection object is passed through the RxBus, switch conditional is used to

check its DataType variable against DataType.java static integers. Based on that, appropriate code is executed.

```
public class DataType {

    public static final int HEIGHT = 0;
    public static final int WEIGHT = 1;
    public static final int BIRTHDAY = 2;
    public static final int GENDER = 3;
    public static final int NAME = 4;

    private int correspondingInt;

    public static final String DataLabel[] = new String[]{"Height", "Weight", "Birthday", "Gender", "Name"};

    DataType(int i) { this.correspondingInt = i; }

    public int getCorrespondingInt() { return correspondingInt; }
}
```

Figure 27. DataType.Java Class.

```
RxBus.subscribe((userDataCollection) -> {
    if (userDataCollection instanceof UserDataCollection) {
        UserDataCollection userData = (UserDataCollection) userDataCollection;
        mReference.child(UserID).child(DataType.DataLabel[userData.getDataType()]).setValue(userData.getUserData());
        switch (((UserDataCollection) userDataCollection).getDataType()){
            case DataType.HEIGHT: {
                UserDataCollection.UserHeight newUserDataCollection = (UserDataCollection.UserHeight) userDataCollection;
                userHeight = newUserDataCollection.getUserData();
                profileHeight.setText(newUserDataCollection.getDisplayString());
                break;
            }
```

Figure 28. Use Case of DataType Class.

5.3.6 LineDataCollection.java

LineDataCollection class is an object that encapsulates instances of LineData objects that are created on receival of data packages from the wearable device. A separate class is required in order to differentiate between different types of object being send over RxBus, using “instanceof” method. This class is shown in Figure 29 below.

```java
package fi.delektre.ringa.ring_thesis.util;

import com.github.mikephil.charting.data.LineData;
public class LineDataCollection {

    private LineData temp;
    private LineData pressure;

    public LineDataCollection(){
        this.temp = new LineData();
        this.pressure = new LineData();
    }

    public LineDataCollection(LineData temp, LineData pressure){
        this.temp = temp;
        this.pressure = pressure;
    }

    public void setTemp(LineData temp) { this.temp = temp; }

    public void setPressure(LineData pressure) { this.pressure = pressure; }


    public LineData getPressure() { return pressure; }

    public LineData getTemp() { return temp; }
}
```

Figure 29. LineDataCollection.Java. Only temperature-related methods have been used during the project.

5.3.7 SelectorOption.java

SelectorOption class is used to create instances of buttons that are used to change the type of data displayed, as shown in the Figure 30.

```
public class SelectorOption {
    public final int imageDrawableResource;
    public final String optionName;

    public SelectorOption(int imageDrawableResource, String optionName) {
        this.imageDrawableResource = imageDrawableResource;
        this.optionName = optionName;
    }
}
```

Figure 30. SelectorOption Class.

5.3.8 UserDataCollection.java

This class is necessary for storing, sending and retrieving different types of personal data. It contains superclass called “UserDataCollection”, and five children classes corresponding to personal data collected by the application: “UserHeight”, “UserWeight”, “UserBirthday”, “UserGender” and “UserName”. As Figure 31 demonstrates, superclass has “DataType” and “UserData” properties. Figure 32 show how all children classes inherit from the “UserDataCollection”. Each of the children contains a method for presenting a user-readable string based on the received data.

```
public class UserDataCollection {
    protected String userData;
    protected int dataType;

    public UserDataCollection(){

    }

    public UserDataCollection(String userData, int dataType){
        this.dataType = dataType;
        this.userData = userData;

    }

    public int getDataType() { return dataType; }

    public String getUserData() { return userData; }
```

Figure 31. UserDataCollection Superclass.

```
public class UserHeight extends UserDataCollection {
    public UserHeight(int dataType, String userData) {
        this.dataType = dataType;
        this.userData = userData;
    }

    public String getDisplayString(){
        String receivedData[] = this.getUserData().split( regex: ":");
        if (receivedData.length == 3) {
            String user_height = receivedData[0] + "' " + receivedData[1] + "''";
            return user_height;
        } else if (receivedData.length == 2){
            String user_height = receivedData[0] + " " + receivedData[1];
            return user_height;
        }
        return null;
    }
}
```

Figure 32. Example of Class Inheriting from UserDataCollection. Note “getDisplayString()” Method.

5.4 Adapter Classes

In Android, adapter classes are classes which connect data with adapter views. They are used for generating views based on a give data set. Adapter views help to minimize boilerplate code and improve performance. RecyclerView Adapters, which are used in the

project, have to be implemented in a specific way because of the usage of AndroidAnnotations, and two following classes are implementations taken from the AndroidAnnotations guidelines. /19/

5.4.1 ViewWrapper.Java

A universal wrapper class which can wrap any View into a ViewHolder, shown in Figure 33.

```
public class ViewWrapper<V extends View> extends RecyclerView.ViewHolder {

    private V view;

    public ViewWrapper(V itemView) {
        super(itemView);
        view = itemView;
    }

    public V getView() { return view; }
}
```

Figure 33. ViewWrapper Class.

5.4.2 RecyclerViewAdapterBase.Jjava

A shared class used for all RecycleView adapters, shown in Figure 34.

```
public abstract class RecyclerViewAdapterBase<T, V extends View> extends RecyclerView.Adapter<ViewWrapper<V>> {
    public List<T> items = new ArrayList<T>();

    @Override
    public int getItemCount() { return items.size(); }

    @Override
    public final ViewWrapper<V> onCreateViewHolder(ViewGroup parent, int viewType) {
        return new ViewWrapper<V>(onCreateItemView(parent,viewType));
    }

    protected abstract V onCreateItemView(ViewGroup parent, int ViewType);

}
```

Figure 34. RecyclerViewAdapterBase Class.

5.4.3 SelectorRecycleViewAdapter.java

This class is responsible for creating buttons that switch between measurement types displayed in the GraphView. It sets listeners to the buttons inside the RecyclerSelector view.

In addition, SelectorRecycleViewAdapter.java holds measurement data that is used for graph rendering. It contains a method called “initGraphDataSets()”, shown in Figure 35,

which instantiates LineData objects for each measurement type, as displayed in Figure 36, and sets the settings accordingly. A singleton instantiation of the class is required for proper usage, hence, the instance of the class is created in “@AfterViews” callback in the MainActivity. New buttons are created using “.items.add” method, which is a native method for adding new objects to a List<>.

```
public void initGraphsDataSets(){
    lineDataSetBloodPressure = new LineDataSet(yAxesPressure,  label: "Blood Pressure");
    lineDataSetBloodPressure.setDrawCircles(false);
    lineDataSetBloodPressure.setColor(Color.BLUE);
    lineDataSetBloodPressure.setMode(LineDataSet.Mode.CUBIC_BEZIER);
```

Figure 35. Example of LineDataSet Creation Inside the SelectorRecycleViewAdapter class.

```
    public void onBindViewHolder(ViewWrapper<RecyclerSelectorView> viewHolder, int position) {
        RecyclerSelectorView view = viewHolder.getView();
        SelectorOption option = items.get(position);
        view.bind(option);
        int pos = position;

        view.setOnClickListener(view1 -> {
            if (pos == 0){
                buttonPosition = 0;
                lineChart.setData(new LineData(lineDataSetsTemperatures));
            }
            if (pos == 1){
                buttonPosition = 1;
                lineChart.setData(new LineData(lineDataSetFluidBalance));
            }
            if (pos == 2){
                buttonPosition = 2;
                lineChart.setData(new LineData(lineDataSetPulse));
            }
            if (pos == 3){
                buttonPosition = 3;
                lineChart.setData(new LineData(lineDataSetBloodPressure));
            }
            lineChart.notifyDataSetChanged();
            lineChart.invalidate();

        });
    }
}
```

Figure 36. OnBindViewHolder Override, Which Adds “onClick()” Listeners for Items in Different Positions Inside RecyclerView.

5.4.4 SectionStatePagerAdapter.java

Demonstrated in Figure 37, SectionStatePagerAdapter class is similar in function to other classes presented in this section of the thesis. The difference comes from it being an adapter for Fragments instead of regular views. Core implementation is similar to SelectorRecycleView class: there are methods present for adding Fragments to the adapter, getting a Fragment and for counting the number of Fragments added to the List<> of type Fragment.

```
public class SectionStatePagerAdapter extends FragmentStatePagerAdapter {

    private final List<Fragment> mFragmentList = new ArrayList<>();
    private final List<String> mFragmentTitleList = new ArrayList<>();

    public SectionStatePagerAdapter(FragmentManager fm){
        super(fm);
    }

    public void addFragment(Fragment fragment, String title){
        mFragmentList.add(fragment);
        mFragmentTitleList.add(title);
    }

    @Override
    public Fragment getItem(int position) {
        return mFragmentList.get(position);
    }

    @Override
    public int getCount() {
        return mFragmentList.size();
    }
}
```

Figure 37. SectionStatePagerAdapter Class.

5.5 Service

Classes under “Service” folder are used for initialization of the “BLE_Service” which manages retrieval and handling of data send from the wearable device, as well as “BLE_ConnectionCheck”, that displays a message if a Bluetooth device on connection state change.

5.5.1 BLE_ConnectionCheck.java

BLE_ConnectionCheck class, as demonstrated in Figure 38, extends from the superclass BroadcastReceiver, which is native Android method used to transmit data or result codes whenever a system event occurs. In the Senno application, this is used to detect Bluetooth devices connection state changes. The class overrides “.onReceive” method, which subscribes to system events, and shows appropriate toast messages on “BluetoothAdaper.STATE_*” callbacks. It is instantiated in the “BLE_Service” class.

```java
public class BLE_ConnectionCheck extends BroadcastReceiver {
    Context activityContext;
    int duration = Toast.LENGTH_SHORT;
    public BLE_ConnectionCheck(Context activityContext){
        this.activityContext = activityContext;
    }

    @Override
    public void onReceive(Context context, Intent intent){
        final String action = intent.getAction();

        if (action.equals(BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED)){
            final int state = intent.getIntExtra(BluetoothAdapter.EXTRA_CONNECTION_STATE, BluetoothAdapter.ERROR);
            switch (state) {
                case BluetoothAdapter.STATE_DISCONNECTED:
                    Toast toast1 = Toast.makeText(activityContext,  text: "Device Disconnected!", duration);
                    toast1.show();
                    break;
                case BluetoothAdapter.STATE_CONNECTED:
                    Toast toast2 = Toast.makeText(activityContext,  text: "Device Connected!", duration);
                    toast2.show();
                    break;
                }
            }
        }
    }
```

Figure 38. BLE_ConnectionCheck Class.

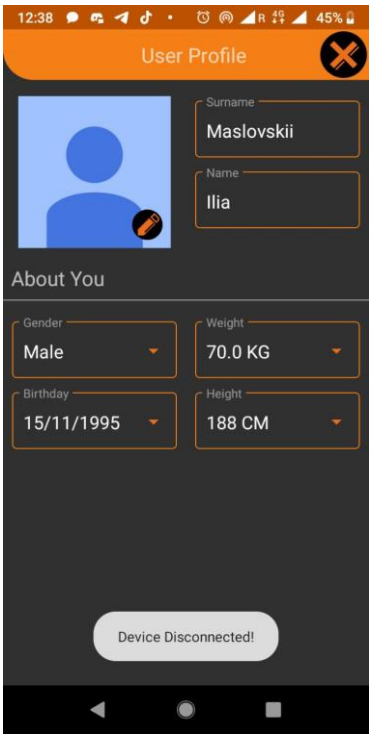


Figure 39. Bluetooth Device Disconnected.

5.5.2 BLE_Service.java

This class handles the received data management and BLE communication. “BLE_Services” utilize AndroidRxBle library to enhance and simplify BLE connection management on Android. It listens for BLE observables, which can be seen in Figure 40, and subscribes to them if any are found, using “onScanSuccess” handler by assigning the device discovered to “connectionObservable” variable, as demonstrated in Figure 41. If none are found or an error is encountered, an exception is thrown using the “onScanFailure” handler, as shown in Figure 42. If scanning fails, the application stops listening to “connectionObservable”. Whenever possible, the operations are threaded to avoid the application freezing.

```
protected void findDevice() {
    if (rxBleClient == null) {
        Log.e(TAG,  msg: "Missing BLE client, creating");
        rxBleClient = RxBleClient.create(this);
    }

    scanSubscription = rxBleClient.scanBleDevices(
            new ScanSettings.Builder()
                    //.setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                    //.setCallbackType(ScanSettings.CALLBACK_TYPE_FIRST_MATCH)
                    .build(),
            new ScanFilter.Builder()
                    .setDeviceName("RINGA")
                    .build()
    )
            .observeOn(AndroidSchedulers.mainThread())
            .doOnDispose(this::clearSubscription)
            .take(1)
            .subscribe(this::onScanSuccess, this::onScanFailure);
}
```

Figure 40. "findDevice()" Method Implementation. Note the Handlers for Successful and Failed Scan in "subscribe()" Method.

```
@UiThread
public void onScanSuccess(ScanResult scanResult) {
    Log.e(TAG,  msg: "Found: " + scanResult);
    rxBleDevice = scanResult.getBleDevice();
    String macAddr = scanResult.getBleDevice().getMacAddress();

    connectionObservable = prepareConnectionObservable();
    if (scanSubscription != null)
        scanSubscription.dispose();
}



private Observable<RxBleConnection> prepareConnectionObservable() {
    return rxBleDevice
            .establishConnection( autoConnect: false)
            .takeUntil(disconnectTriggerSubject)
            .doOnDispose(this::clearSubscription)
            .compose(new ConnectionSharingAdapter());
}
```

Figure 41. Assigning rxBleDevice Connection to "connectionObservable" Variable.

```
private void onScanFailure(Throwable throwable) {

    if (throwable instanceof BleScanException) {
        handleBleScanException((BleScanException) throwable);
    }
    connectionObservable.unsubscribeOn(AndroidSchedulers.mainThread());
}
```

Figure 42. Scanning Failure Handling. Not included – various error messages depending on the issue type.

Following device discovery, “connectionObservable” emits data received over the Bluetooth connection is used in “fetchData()” method that is called during “onCreate()” Android callback. It maps all data that is emitted using specified UUID to a flat map, observes on an Android thread, utilizes custom “RetryWithDelay” class whenever the connection is disrupted, and subscribes function “showData()” to the emitted bytes, meaning it is called whenever new bytes enter the data stream and are observed.

```
public void fetchData() {
    if (connectionObservable != null) {
        connectionObservable
                .flatMapSingle(rxBleConnection -> rxBleConnection.readCharacteristic(AppConst.UUID_DATA_CHARACTERISTIC))
                .observeOn(AndroidSchedulers.mainThread())
                .retryWhen(new RetryWithDelay( maxRetries: 3,  retryDelayMillis: 20000 ))
                .subscribe(bytes -> {
                            showData(bytes);
                        },
                        this::onReadFailure);
    }

}
```

Figure 43.“fetchData()” Method Implementation.

Shown in Figure 41, “fetchData()” method handles further data mapping and pipeline management. An example of data mapping is shown in chapter “4.2.2. Service Communication” of this thesis. “fetchData()” is called every 300 milliseconds using “fetchDataTimer()” method, as demonstrated in Figure 44.

```
void fetchDataTimer(){
    Observable disposable = Observable.interval( period: 300, TimeUnit.MILLISECONDS);
    disposable.subscribeOn(Schedulers.from(Executors.newFixedThreadPool( nThreads: 10))).subscribe(t -> {
        fetchData();
    });
}
```

Figure 44. "fetchDataTimer()" Method.

Certain types of measurement data, as is the case with temperature measurements, are published over RxBus to be displayed using GraphView, as shown in Figure 45. Other data, due to the constraints of the project, are sent directly to the Firebase Database for storage. Custom class "ChartUtils" is utilized in this method, to reduce the size of data sets that are transferred to the "MainActivity" for data visualization.

```
removeOutdatedEntries(lineDataSetTemperature1);
removeOutdatedEntries(lineDataSetTemperature2);

LineData tempData = new LineData(lineDataSetsTemperatures);
parseData.setTemp(tempData);

RxBus.publish(parseData);
```

Figure 45. Publishing data over RxBus and Utilizing "CharUtils" to Remove Outdated Entries.

To check the integrity of data received, both the wearable device and the mobile application implement CRC8 integrity check. It is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents. CRC8 implementation is demonstrated in Figure 46. On retrieval, the calculation is repeated and, in the event the check values do not match, corrective action can be taken against data corruption. CRCs can be used for error correction. /20/

```
public static byte CRC8(byte[] bytes) {
    byte generator = (byte) 0x07;
    byte crc = 0;
    for (byte currByte : bytes) {
        crc ^= currByte;

        for (int i = 0; i < 8; i++) {
            if ((crc & 0x80) != 0) {
                crc = (byte) ((crc << 1) ^ generator);
            } else {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

Figure 46. CRC8 Implementation. It is Applied to the Received Bytes Array.

In the current version of the application, temperature data is concatenated and sent to the Firebase Database only once per minute. This mitigates database overflow. Figure 47 demonstrates how the data is sent to the database.

During the testing phase, measurements from LED-based spectrometer have been gathered. Those measurements have processed to the Firebase Database immediately upon read. Timestamps are used as parents for the batch of readings. Example is shown in Figure 48.

```
if (tempArrayList.size() < 60) {
    tempArrayList.add((int) temperature1);
} else {
    int avgTemp = 0;
    for (int temp : tempArrayList){
        avgTemp += temp;
    }
    avgTemp = avgTemp / 60;
    currentTimeMillis = System.currentTimeMillis();

    mReferecence.child("Readings").child(UserID).child(String.valueOf(currentTimeMillis))
            .child("Temperature").setValue(avgTemp);
    tempArrayList.clear();
    tempArrayList = new ArrayList<>();
}
long timeSnapshot = System.currentTimeMillis();
mReferecence.child("LED_Test").child(UserID)
        .child(String.valueOf(timeSnapshot)).child("RED_S").setValue(redS);
mReferecence.child("LED_Test").child(UserID)
        .child(String.valueOf(timeSnapshot)).child("GREEN_S").setValue(greenS);
mReferecence.child("LED_Test").child(UserID)
        .child(String.valueOf(timeSnapshot)).child("BLUE_S").setValue(blueS);
mReferecence.child("LED_Test").child(UserID)
        .child(String.valueOf(timeSnapshot)).child("NIR_S").setValue(nirS);
mReferecence.child("LED_Test").child(UserID).
        child(String.valueOf(timeSnapshot)).child("YELLOW_S").setValue(yellowS);
```

Figure 47. Example of Raw and Concatenated Data Send to Firebase Database.

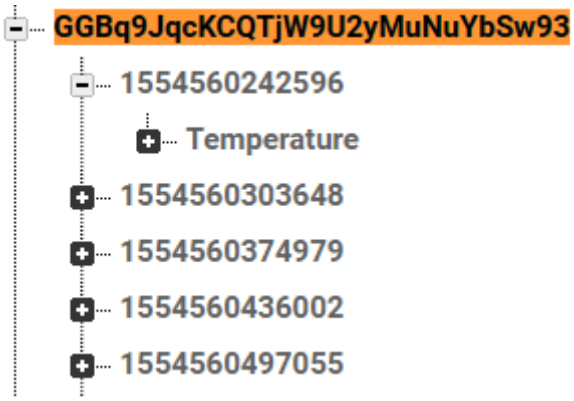


Figure 48. Temperature Measurement in Firebase Database.

5.6 User Interface

User Interface folder holds classes which implement Android Fragments related to user interface, specifically, those pertaining to user personal data. These classes, once instantiated, are the primary way of setting and changing the personal data, excluding direct database manipulation. Once their user has set the data one way or another, it is published

onto RxBus. Following that, the data is retrieved by the Fragment's parent activity and from there on acted upon accordingly, e.g., it is send to the Firebase Database or displayed inside the Activity. Two buttons with implemented "onClick()" listeners are present in all "*PickerDialog.java" Fragments – they all implement "declineChanges()", shown in Figure 49, method, which resets the values to the original ones, and "acceptChanges()" method, which pushes the changes to RxBus.

```
@Click(R.id.gender_decline)
void declineChanges() { dismiss(); }

@Click(R.id.gender_accept)
void acceptChanges(){
    int i = genderPicker.getValue();
    String data = genderPickerVars[i];
    RxBus.publish(new UserDataCollection().new UserGender(DataType.GENDER, data));
    dismiss();
}
```

Figure 49. "declineChanges()" Method.

5.6.1 GenderPickerDialog.java

"GenderPickerDialog" class is an implementation of "DialogFragment" class, as shown in Figure 50. Once instantiated, it sets two values, 0 and 1, to the "NumberPicker" UI element that it implements and assigns string labels to be displayed whenever one of the values is picked using "NumberPicker". The "NumberPicker" is set not to roll over. A Toast message notification with currently selected value in the text form is shown when the user access "DialogFragment".

```
@AfterViews
void initialization(){
    userGenderString = getArguments().getString( key: "gender");

    genderPickerVars = new String[]{"Male", "Female"};
    genderPicker.setValue(0);

    genderPicker.setMinValue(0);
    genderPicker.setMaxValue(genderPickerVars.length-1);
    genderPicker.setDisplayedValues(genderPickerVars);

    if(userGenderString != null){
        int i = 0;
        for (String s : genderPickerVars){
            if (s.equals(userGenderString)){
                Toast.makeText(this.getContext(), s, Toast.LENGTH_SHORT).show();
                genderPicker.setValue(i);
            }
            i++;
        }
    }
}
```

Figure 50. GenderPickerDialog Implementation of Initialization.

Figure 51. GenderPickerDialog in Application.

5.6.2 DatePickerDialog.java

In addition to methods shared between all “DialogFragment” under the UI folder of Android Project, two additional methods are implemented. They are used to switch between View that allows picking year directly, and a View that lets user choose day and month by using Calendar widget. “changeYear()” enabled “NumberPicker” element that is responsible for choosing year, and “changeMonth()” disable the “NumberPicker” and enables the Calendar. Those methods’ implementations are shown in Figure 52. It also sets the default birthday date if none is present in the database. Whenever a day of the month is chosen, a Toast notification with the current selection is shown to the user. In both cases, listeners are responsible for detecting changes in data and saving them to the temporary values. As demonstrated in Figure 53, this is achieved by overriding “onValueChange” and “onSelectedDayChange” callbacks.

```
@Click(R.id.date_dialog_year)
void changeYear() {
    dateYear.setText(String.valueOf(pickerYear.getValue()));
    calendarView.setVisibility(View.GONE);
    pickerYear.setVisibility(View.VISIBLE);

}

@Click(R.id.date_dialog_date)
void changeMonth(){
    SimpleDateFormat sdf =  new SimpleDateFormat( pattern: "dd:MM:yyyy");
        try {
            if (newUserBirthdayString == null){
                newUserBirthdayString = "01:01";
            }
            dateInstance = sdf.parse( source: newUserBirthdayString + ":" + year);
        } catch (ParseException e) {


            e.printStackTrace();
        }
    mCalendar.setTime(dateInstance);
    mCalendar.set(Calendar.YEAR, pickerYear.getValue());

    dateInstance = mCalendar.getTime();
    calendarView.setDate(dateInstance.getTime());
    dateYear.setText(Integer.toString(pickerYear.getValue()));
    calendarView.setVisibility(View.VISIBLE);
    pickerYear.setVisibility(View.GONE);
}
```

Figure 52. “changeYear()” and “changeMonth()” Implementation.

```
calendarView.setOnDateChangeListener(new CalendarView.OnDateChangeListener() {
    @Override
    public void onSelectedDayChange(@NonNull CalendarView calendarView, int i, int i1, int i2) {
        year = i;

        i1++;
        pickerYear.setValue(year);
        dateYear.setText(String.valueOf(year));
        newUserBirthdayString = i2 + ":" + i1;
        SimpleDateFormat sdf =  new SimpleDateFormat( pattern: "dd:MM");
        try {
            dateInstance = sdf.parse(newUserBirthdayString);

        } catch (ParseException e) {
            e.printStackTrace();
        }
        displayDateString = displayDateFormat.format(dateInstance);
        dateMonth.setText(displayDateString);

        Toast toast = Toast.makeText(mContext, displayDateString, Toast.LENGTH_LONG);
        toast.show();


    }
});
```

Figure 53. Example of Listener Implementation. Note “newUserBirthdayString” Format.

From here on now, whenever a data string is formatted to a type of “xx:xx:xx”, it is done to store it in Firebase Database with the ability of easy parsing when retrieved.

Figure 54. Date Picker in Application.

5.6.3 HeightPickerDialog.java and WeightPickerDialog.java

These two Fragments are used by the users to pick their weight and height. Due to similarities in implementation, only one implementation is shown in detail.

Both dialogs are designed to be used by people who prefer either the metric or the imperial measurement system. Hence, both implementations have a switch which allows to transition between the two measurement systems. This required keeping the values in temporary storage variables (Figure 55), implementing methods for unit conversion (Figure 57, Figure 58), and storing the measurement unit preferred by the user in the database (Figure 56). Boolean is used to distinguish between two measurement modes in-app. In "HeightPickerDialog", string length is used in to determine which type of measurement unity is selected at this moment and has been passed to the Fragment. In "WeightPickerDialog", as the strings are of the same length, the last piece of split data array is used to establish selected measurement unit.

```
userHeightString = getArguments().getString( key: "height");

/*Receiving string from the parent activity, splitting it.
Then, if the array size is equal to 3, it assumes that user's height is received in imperial units
Otherwise, it is recieved in SI. The correct initialization is applied.
 */
if (userHeightString != null) {
    String dataSplit[] = userHeightString.split( regex: ":");
    if (dataSplit.length == 2) {
        measUnit = dataSplit[1];
        heightUnits.setText(measUnit);
        unitSwitch.setText(measUnit);
        mainPicker.setFormatter(new NumberPicker.Formatter() {
            @Override
            public String format(int value) { return String.format("%d CM", value); }
        });
        decimalPicker.setVisibility(View.GONE);
    } else if (dataSplit.length == 3){
        measUnit = dataSplit[2];
        heightUnits.setText(measUnit);
        unitSwitch.setText(measUnit);
        userDecimalHeight = Double.parseDouble(dataSplit[1]);
        mainPicker.setFormatter(new NumberPicker.Formatter() {
            @Override
            public String format(int value) { return String.format("%d ft", value); }
        });
        View firstItem = mainPicker.getChildAt( index: 0);
        if (firstItem != null) {
            firstItem.setVisibility(View.INVISIBLE);
        }
```

Figure 55. Example of Parsing Data String, “HeightPickerDialog” Class Implementation.

```
@Click(R.id.height_accept)
void acceptChanges(){
    if (measUnit.equals("CM")) {
        String data = (String.valueOf(mainPicker.getValue()) + ":" + String.valueOf(measUnit));
        RxBus.publish(new UserDataCollection().new UserHeight(DataType.HEIGHT, data));
    } else {
        String data = (String.valueOf(mainPicker.getValue()) + ":" + String.valueOf(decimalPicker
                .getValue()) + ":" + String.valueOf(measUnit));
        RxBus.publish(new UserDataCollection().new UserHeight(DataType.HEIGHT, data));
    }
    dismiss();
}
```

Figure 56. Conditional to Send Different Height with Different Units.

```
newCheckListener();

if (measUnit.equals("FT")){
    unitSwitch.setOnCheckedChangeListener(null);
    unitSwitch.setChecked(true);
    newCheckListener();
} else {
    unitSwitch.setOnCheckedChangeListener(null);
    unitSwitch.setChecked(false);
    newCheckListener();
}
```

Figure 57. Setting the Switch on Initialization Based on Current Measurement Unit.

```
public void newCheckListener(){
    unitSwitch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
        @Override
        public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
            if (isChecked){
                measUnit = "LBS";
                userWeightMain = (mainPicker.getValue() + (decimalPicker.getValue() * 0.1)) * 2.205;
                userDecimalWeight = Math.round(userWeightMain % 1 * 10);
                mainPicker.setMaxValue(weightPoundMax);
                mainPicker.setValue((int) userWeightMain);
                decimalPicker.setValue((int) userDecimalWeight);
                //decimalPicker.setValue((int) userDecimalWeight);
                weightUnits.setText(measUnit);
                unitSwitch.setText("LBS");
            } else {
                measUnit = "KG";
                userWeightMain = (mainPicker.getValue() + (decimalPicker.getValue() * 0.1)) / 2.205;
                //userDecimalWeight = userWeightMain % 1 * 10;
                userDecimalWeight = Math.round(userWeightMain % 1 * 10);
                mainPicker.setMaxValue(weightKilosMax);
                mainPicker.setValue((int) userWeightMain);
                decimalPicker.setValue((int) userDecimalWeight);
                weightUnits.setText(measUnit);
                unitSwitch.setText("KG");
            }
        }
    });
}
```

Figure 58. Example of Measurement Unit Conversion Method, “WeightPickerDialog” Class.

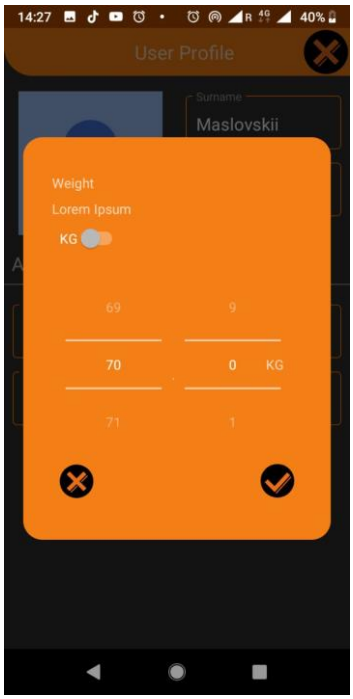


Figure 59. Weight Picker Dialog in Application.

5.6.4 CreateProfileNameFragment.java

“CreateProfileNameFragment” class is the implementation of Fragment that is part of the user account creation flow. It contains input methods for name and surname, which are implemented via adding listeners using “addTextChangedListener()” method on input field objects, as shown in Figure 60. From there, in “afterTextChanged” callback, text from input fields is set to temporary variables. In addition, Radio Buttons have been implemented for choosing the user’s gender. Upon pressing one of them, the gender value is set according to the label of the button. Lastly, this class contains methods “createBackButton()”, showcased in Figure 61, which redirects user to the login screen, and next, which publishes data that has been input over RxBus and switches to the next Fragment in the registration flow, which is “CreateProfileDataFragment”.

```
createFirstName.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence charSequence, int i, int i1, int i2) {

    }

    @Override
    public void onTextChanged(CharSequence charSequence, int i, int i1, int i2) {
    }

    @Override
    public void afterTextChanged(Editable editable) {
        userFirstname = String.valueOf(editable);
    }
});
```

Figure 60. Implementation of Input Field Listener.

```
@Click(R.id.create_back)
void createBackButton(){
    startActivity(new Intent(getActivity(), LoginActivity_.class));
    ((UserData_)getActivity()).overridePendingTransition( enterAnim: 0,  exitAnim: 0);
}

@Click(R.id.create_next)
void createNextButton(){
    if (userFirstname != null && userSurname != null && userGender!= null){
        userFullname = userFirstname + ":" + userSurname;
        RxBus.publish(new UserDataCollection().new UserName(DataType.NAME, userFullname));
        RxBus.publish(new UserDataCollection().new UserGender(DataType.GENDER, userGender));
        ((UserData_)getActivity()).setViewPager(1);
    }
}
```

Figure 61. “createBackButton()” and “createNextButton()” Implementation.

5.6.5 CreateProfileDataFragment.java

This Fragment implements both “HightPickerDialogFragment” and “WeightPicker-DialogFragment”. It is used during the user registration application flow, to set up the user’s personal weight and height. On initialization, this Fragments subscribes to the object types that are pushed onto RxBus by the implemented “DialogFragment” classes. It contains methods which would inflate the “DialogFragment” classes by utilizing Fragment Manager. Fragment instantiates on the user interacting with input fields. As shown in Figure 62, CreateProfileDataFragment class contains “finishSignup()” method, which publishes the data collected from “DialogFragment” instances to RxBus and call method

"submitAndFinish()" from the parent activity, and a back button, which returns the user to previous signup stage.

```
private void showBirthdayPickerDialog() {
    FragmentManager fm = getActivity().getSupportFragmentManager();
    FragmentTransaction ft = fm.beginTransaction();
    ft.addToBackStack(null);

    DatePickerDialog datePicker = DatePickerDialog_.builder().build();
    Bundle dateBundle = new Bundle();

    dateBundle.putString("birthday", userBirthday);
    datePicker.setArguments(dateBundle);
    datePicker.show(ft,  tag: "date_picker_name");
}

private void finishSignup(){
    RxBus.publish(userBirthdayCollection);
    RxBus.publish(userWeigthCollection);
    RxBus.publish(userBirthdayCollection);
    ((UserData_)getActivity()).submitAndFinish();
}
```

Figure 62. Example of "DialogFragment" Class Inflation Method and "finishSignup" Method Implementation.

5.7 Activities

This section contains implementations of Activities. Due to their special role in Android application structure, the sections of this chapter will not be called by the name of the file that implements it, but by a human readable name, to properly describe its function. All activities instantiate Firebase variables, specifically, the database reference and authorization instance. Some of them use a Firebase Storage reference, to send or retrieve files to the Firebase Storage.

5.7.1 Login Activity

Login Activity is implemented by LoginActivity.java. Login Activity class extends the "AppCompatActivity" class in order to be able to properly use Fragments and to ensure backward compatibility.

Login Activity contains implementations and declarations for "MediaPlayer"-type object, which plays a video on a loop in the background of the login screen and "Login" and "Create" buttons, which, respectively, login the user if the information provided has been

correct and create new user account, given that the credentials provided has not been taken. If the user has already logged in, he/she is redirected to application dashboard, as shown below in Figure 63.

```
FirebaseUser currentUser = mAuth.getCurrentUser();
if (currentUser != null) {
    mReference.child(currentUser.getUid()).addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            userName = ((String) dataSnapshot.child("Name").getValue());
            Toast.makeText( context: LoginActivity.this, userName, Toast.LENGTH_SHORT).show();
            if (userName != null){
                Intent intent = new Intent( packageContext: LoginActivity.this, MainActivity_.class);
                startActivity(intent);
            }
        }

        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {

        }
    });
}
```

Figure 63. User Redirection.

The “MediaPlayer” object is set up once the Views are ready. It needs a Surface-type View to project its contents. The link to a placeholder video is provided, “MediaPlayer” is started, and set to looping. Then, a media source is set to the placeholder video, in Surface View listener, the player is bound to the surface via “setSurface()” method. The process is described in Figure 64 and Figure 65.

```
mediaPlayerUri = Uri.parse("android.resource://"+getPackageName()+"/"+R.raw.clock_spin);
Ml_Player = new MediaPlayer();
Ml_Player.setOnPreparedListener(new MediaPlayer.OnPreparedListener() {
    @Override
    public void onPrepared(MediaPlayer mediaPlayer) {
        mediaPlayer.setVolume( leftVolume: 0,  rightVolume: 0);
        mediaPlayer.start();
        mediaPlayer.setLooping(true);
    }
});
Tl_View.setSurfaceTextureListener(new TextureView.SurfaceTextureListener() {
    @Override
    public void onSurfaceTextureAvailable(SurfaceTexture surfaceTexture, int i, int il) {
        Sl_Surface = new Surface(surfaceTexture);

        try {
            Ml_Player.setDataSource( context: LoginActivity.this, mediaPlayerUri);
        } catch (IOException e) {
            e.printStackTrace();
        }
        Ml_Player.setSurface(Sl_Surface);
        Ml_Player.setVideoScalingMode(2);
        scaleVideo(Ml_Player);
        Ml_Player.prepareAsync();

    }
```

Figure 64. Setting up “MediaPlayer”.

```
private void scaleVideo(MediaPlayer mediaPlayer) {

    LayoutParams videoParams = (LayoutParams) Tl_View
            .getLayoutParams();
    DisplayMetrics dm = new DisplayMetrics();
    LoginActivity.this.getWindowManager().getDefaultDisplay()
            .getMetrics(dm);

    final int height = dm.heightPixels;
    final int width = dm.widthPixels;
    int videoHeight = mediaPlayer.getVideoHeight();
    int videoWidth = mediaPlayer.getVideoWidth();
    double hRatio = 1;

    hRatio = (height * 1.0 / videoHeight) / (width * 1.0 / videoWidth);
    int videoParam_X = (int) (hRatio <= 1 ? 0 : Math.round((-(hRatio - 1) / 2)
            * width));
    int videoParam_Y = (int) (hRatio >= 1 ? 0 : Math
            .round((((-1 / hRatio) + 1) / 2) * height));
    videoParams.width = width - videoParam_X - videoParam_X;
    videoParams.height = height - videoParam_Y - videoParam_Y;
    Log.e(TAG,  msg: "x:" + videoParam_X + " y:" + videoParam_Y);
    Tl_View.setScaleX(1.00001f);//<-- this line enables smoothing of the picture in TextureView.
    Tl_View.requestLayout();
    Tl_View.invalidate();
}
```

Figure 65. Video Scaling Method.

The methods for signing in and account creation are similar in implementation. The difference comes from calling different Firebase Auth methods. One implements the

“signInWithEmailAndPassword()” method, and if callback is returned successfully it redirects user to the main dashboard, and the other implements “createUser-WithEmailAndPassword”, attempting to create a new Firebase User with credentials given, and, if successful, starts the account creation flow. The “create()” method, shown in Figure 66, sets up an empty field for user personal data in the Firebase Database, under the user’s unique ID that is returned in “onComplete” callback. Toast messages are shown if both methods fail at their task.

```
public void create(String Email, String Password){
    mAuth.createUserWithEmailAndPassword(Email, Password)
            .addOnCompleteListener( activity: this, new OnCompleteListener<AuthResult>() {
                @Override
                public void onComplete(@NonNull Task<AuthResult> task) {
                    if(task.isSuccessful()){
                        Toast.makeText( context: LoginActivity.this, text: "Account Created", Toast.LENGTH_SHORT).show();
                        Toast.makeText( context: LoginActivity.this, text: "Signing In", Toast.LENGTH_SHORT).show();
                        FirebaseUser user = mAuth.getCurrentUser();
                        if (user!= null) {
                            String UserID = user.getUid();
                            mReference.child(UserID).setValue("true");
                            mReference.child(UserID).child("Name").setValue(null);
                            mReference.child(UserID).child("Age").setValue(null);
                            mReference.child(UserID).child("Height").setValue(null);
                            mReference.child(UserID).child("Weight").setValue(null);
                            Intent intent = new Intent( packageContext: LoginActivity.this, UserData_.class);
                            startActivity(intent);
                        }
                    } else
                        Toast.makeText( context: LoginActivity.this, text: "Error, Account not created", Toast
                                .LENGTH_SHORT).show();
                }
            }
    );
}
```

Figure 66. Create Account Method Implementation.

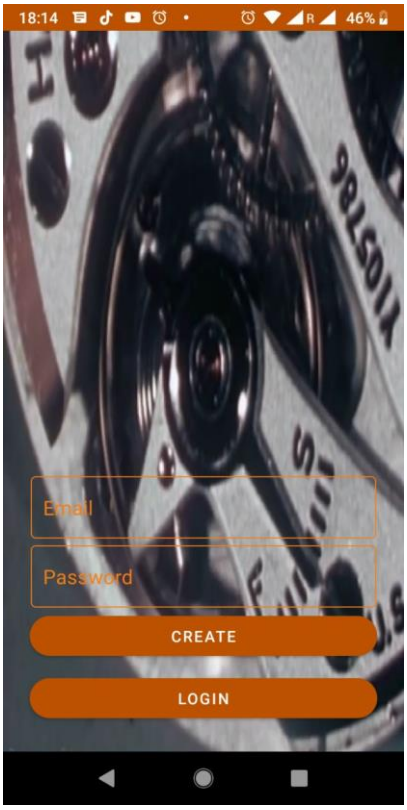


Figure 67. Login Screen in Application.

5.7.2 Register User Activity

Implemented by UserData.java. This Activity is used during the account creation process. First, it asks the user to provide first name, last name and gender, and to proceed further.

If the information provided, the button “Next”, which is present on the screen, will take the user to a view which asks to pick a birthday date and weight. Then, once everything has been provided and user presses “Next” once again, the registration is complete, and the user is taken to the dashboard. In this Activity, switching between multiple views is achieved by using separate fragments. They have been described in chapters 5.6.4. CreateProfileNameFragment.java and 5.6.5. CreateProfileDataFragment.java of this thesis. They are instantiated and manipulated using “SectionStatePagerAdapter” object, which is described in chapter 5.4.4. SectionStatePagerAdapter.java, and which setup is shown in Figure 68. Data that has been pushed by Fragments over RxBus is being listened for. The subscription is done after Views has been initialized, using “subscribeToUserData()” method (Figure 69).

```
void setupAdapter(ViewPager viewPager) {
    SectionStatePagerAdapter adapter = new SectionStatePagerAdapter(getSupportFragmentManager());
    adapter.addFragment(new CreateProfileNameFragment_(),  title: "ProfileNameFragment");
    adapter.addFragment(new CreateProfileDataFragment_(),  title: "ProfileDataFragment");
    viewPager.setAdapter(adapter);
}
```

Figure 68. Setting Up ViewPager for Fragment Usage.

```
private void subscribeToUserData() {
    RxBus.subscribe((userDataCollection) -> {
        if (userDataCollection instanceof UserDataCollection) {
            UserDataCollection userData = (UserDataCollection) userDataCollection;
            mReference.child(UserID).child(DataType.DataLabel[userData.getDataType()]).setValue(userData.getUserData());
            switch (((UserDataCollection) userDataCollection).getDataType()) {
                case DataType.HEIGHT: {
                    userHeightCollection = (UserDataCollection.UserHeight) userDataCollection;
                    userHeight = userHeightCollection.getUserData();
                    dataCollectionArray[DataType.HEIGHT] = userHeightCollection;
                    break;
                }
```

Figure 69. Example of Subscribing to User Data Published from Fragments.

```
public void submitAndFinish(){
    for (int i=0; i < dataCollectionArray.length; i++){
        if (dataCollectionArray[i] == null){
            break;
        }
    }
    Intent intent = new Intent(getApplicationContext(), MainActivity_.class);
    startActivity(intent);
}
```

Figure 70. Method Responsible for Launching Main Activity.

5.7.3 Main Activity

Main Activity is implemented by MainActivity.java, extends “BasicActivity” class. This Activity is the main point of interaction for the user and allows the user to access secondary Activities. It contains an app bar, which holds the Bluetooth connection indicator, app name, and user profile picture. User profile picture is loaded from Firebase Database. As it extends “BasicActivity” class, it contains a navigation bar, accessible by pressing the hamburger icon or on slide, and a graph view, which is the main way for user to visualize the data received over the BLE from the wearable device. For the development purposes, as not all data types were available for transfer over BLE, a Floating Action Bar is present. It generates random values that are added to “LineData” objects that are then displayed with GraphView, as demonstrated in Figure 71.

```
@Click(R.id.fabTest)
void buttonTestData() {
    //graphIndex = adapter.graphIndex;
    graphIndex++;
    Log.d( tag: "TEST",  msg: "TEST INDEX:" + graphIndex);
    //adapter.graphIndex = graphIndex;
    int min = 0;
    int max = 30;
    float temperature1 = (float) Math.random() * ((max - min) + 1);
    float temperature2 = (float) Math.random() * ((max - min) + 1);
    adapter.lineDataSetTemperature1.addEntry(new Entry(graphIndex, temperature1));
    Log.d( tag: "TEST",  msg: "TEST TEMP 1:" + temperature1);
    Log.d( tag: "TEST",  msg: "TEST TEMP 2:" + temperature2);
    adapter.lineDataSetTemperature2.addEntry(new Entry(graphIndex, temperature2));
    float test1 = (float) Math.random() * ((max - min) + 1);
    float test2 = (float) Math.random() * ((max - min) + 1);
    float test3 = (float) Math.random() * ((max - min) + 1);
    adapter.lineDataSetBloodPressure.addEntry(new Entry(graphIndex, test1));
    adapter.lineDataSetPulse.addEntry(new Entry(graphIndex, test2));
    adapter.lineDataSetFluidBalance.addEntry(new Entry(graphIndex, test3));
    adapter.lineDataSetsTemperatures.set(0, adapter.lineDataSetTemperature1);
    adapter.lineDataSetsTemperatures.set(1, adapter.lineDataSetTemperature2);
    lineChart.setData(new LineData(adapter.lineDataSetsTemperatures));
    lineChart.notifyDataSetChanged();
    lineChart.invalidate();
}
```

Figure 71. Test Button Implementation. Generates Random Data Point to Add to Data Sets.

The code shown in Figure 72 instantiates adapter that holds buttons dedicated to selecting preferred data graph, as well as creating the said buttons.

```
void bindAdapter(){

    LinearLayoutManager layoutManager = new LinearLayoutManager( context: this, LinearLayoutManager
            .HORIZONTAL,  reverseLayout: false);
    recycler_view_graph_options.setLayoutManager(layoutManager);
    adapter.items.add(new SelectorOption(R.drawable.ic_body_temp,  optionName: "Body Temperature"));
    adapter.items.add(new SelectorOption(R.drawable.ic_water_balance,  optionName: "Water Balance"));
    adapter.items.add(new SelectorOption(R.drawable.ic_pulse,  optionName: "Pulse"));
    adapter.items.add(new SelectorOption(R.drawable.ic_blood_pressure,  optionName: "Blood Pressure"));
    adapter.lineChart = lineChart;
    adapter.initGraphsDataSets();
    adapter.buttonPosition = 0;
    recycler_view_graph_options.setAdapter(adapter);
}
```

Figure 72. Instantiating Adapter Inside Main Activity.

To view real life data, it necessary to create a subscription to the RxBus. It is achieved via the “fetchData()” method (Figure 73), which subscribes for the objects of type “lineDataCollection”. It extracts temperature data, applies it to a “LineData” type property of instantiated adapter class, and refreshes the GraphView.

```
public void fetchData() {
    RxBus.subscribe((parseData) -> {
        if (parseData instanceof LineDataCollection){
            LineDataCollection chartData = (LineDataCollection) parseData;
            if (adapter.buttonPosition == 0){
                adapter.tempData = (chartData.getTemp());
                lineChart.setData(chartData.getTemp());
            }
        }
        lineChart.notifyDataSetChanged(); // let the chart know it's data changed
        lineChart.invalidate();
    });
}
```

Figure 73. “fetchData()” Implementation.

Main Activity needs to ask for the Bluetooth permission in order for the application to function. To achieve this, “askPermissions()” function is called on initialization (Figure 74).

```
private void askPermissions() {
    Log.d(TAG,  msg: "askPermissions()");
    ActivityCompat.requestPermissions( activity: this,
            new String[]{
                    Manifest.permission.BLUETOOTH,
            },
            PERMISSION_REQUEST_ALL);
}
```

Figure 74. “askPermission()” Implementation

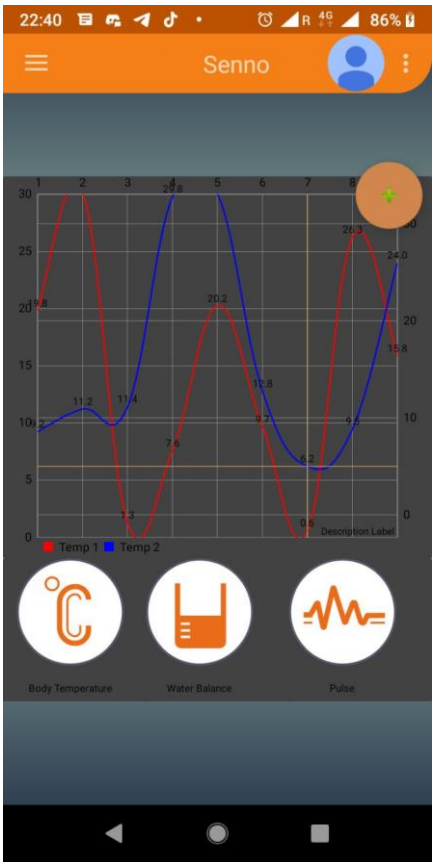


Figure 75. Main Activity View with Sample Data Present.

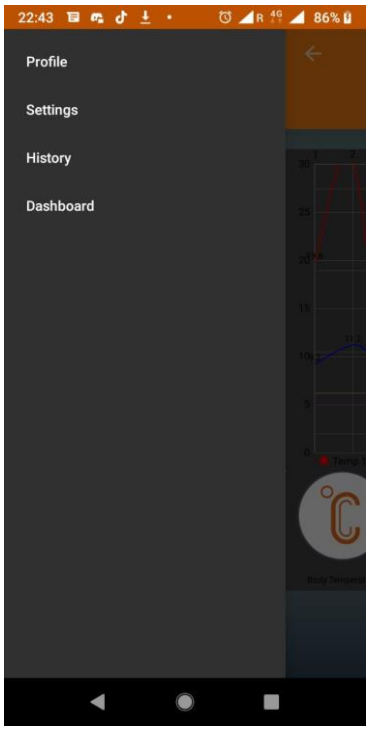


Figure 76. Navigation Bar Present in Main Activity.

5.7.4 Profile Activity

Profile Activity is implemented by ProfileActivity.java. This class is responsible for adjusting the user data after the sign up has been done. It implements all “DialogFragments” related to the personal user information. Each “DialogFragment” has its own inflator method tied to the clickable input field, as showcased in Figure 77.

```
private void showDatePickerDialog(){
    FragmentManager fm = getSupportFragmentManager();
    FragmentTransaction ft = fm.beginTransaction();
    ft.addToBackStack(null);

    DatePickerDialog datePicker = DatePickerDialog_.builder().build();
    Bundle dateBundle = new Bundle();

    dateBundle.putString("birthday", userBirthday);
    datePicker.setArguments(dateBundle);
    datePicker.show(ft,  tag: "date_picker_name");
}
```

Figure 77. Inflator Method for “showDatePickerDialog()” .

Profile Activity uses RxBus “subscribe()” method to update all visual labels inside the activity and to send updated personal data to the Firebase Database (Figure 78). The

proper Firebase Field and Activity View is inferred from the UserDataCollection object DataType field.

```
RxBus.subscribe((userDataCollection) -> {
    if (userDataCollection instanceof UserDataCollection) {
        UserDataCollection userData = (UserDataCollection) userDataCollection;
        mReference.child(UserID).child(DataType.DataLabel[userData.getDataType()]).setValue(userData.getUserData());
        switch (((UserDataCollection) userDataCollection).getDataType()){
            case DataType.HEIGHT: {
                UserDataCollection.UserHeight newUserDataCollection = (UserDataCollection.UserHeight) userDataCollection;
                userHeight = newUserDataCollection.getUserData();
                profileHeight.setText(newUserDataCollection.getDisplayString());
                break;
            }
```

Figure 78. Implementation of Rx.Bus “subscribe()” in Profile Activity.

Profile Activity has a method for taking a user picture and uploading it to the Firebase Storage. This is implemented by launching “activityForRelsult” with correct arguments, which launches the phone’s default camera, then, on callback, if the result is successful, e.g. a picture is taken and the photo is saved, it creates a new File object at the specified path, uses Picasso library to compress it, then, using file stream, upload it to Firebase Storage using Storage Reference. The process is described in Figure 79 and Figure 80.

Then, it sets the image to the ImageView that displays the profile picture.

```
@Override
public void onSuccess() {
        OutputStream fOut = null;
        Bitmap imageBitmap = ((BitmapDrawable) profilePicture.getDrawable()).getBitmap();
        RoundedBitmapDrawable imageDrawable = RoundedBitmapDrawableFactory.create(getResources(), imageBitmap);
        imageDrawable.setCircular(true);
        try {
            ProfileActivity.this.deleteFile(imageFileName);
            profilePictureFile = new File(picturePath);
            try {
                InputStream stream = new FileInputStream(profilePictureFile);
                uploadTask = userFolder.putStream(stream);
                uploadTask.addOnFailureListener(new OnFailureListener() {
                    @Override
                    public void onFailure(@NonNull Exception e) {
                        Toast toast = Toast.makeText( context: ProfileActivity.this,
                                text: "Picture not uploaded", Toast.LENGTH_LONG);
                        toast.show();
                    }
                });
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
            fOut = new FileOutputStream(profilePictureFile);
            imageBitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 100, fOut);
            fOut.flush();
            fOut.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    profilePicture.setImageDrawable(imageDrawable);
```

Figure 79. Part of Implementation of “onActivityResult()” Method Used for Picture Dispatch.

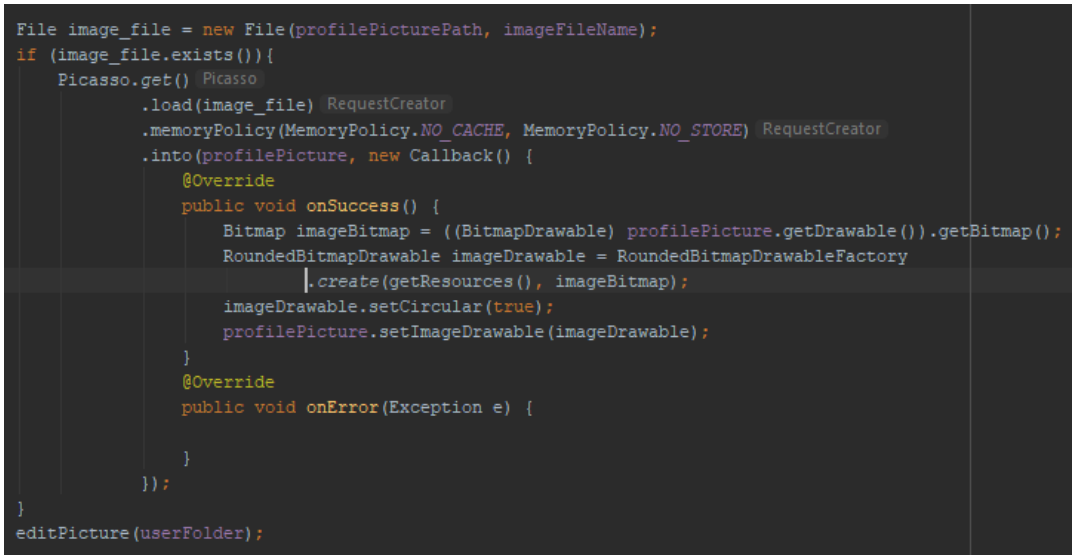
```
private File createImageFile(){
    if (!profilePicturePath.exists()){
        profilePicturePath.mkdirs();
    }
    File image_file;

    image_file = new File(profilePicturePath, imageFileName);
    if (image_file.exists()){
        ProfileActivity.this.deleteFile(imageFileName);
        image_file = new File(profilePicturePath, imageFileName);
    }
    picturePath = image_file.getAbsolutePath();
    return image_file;
}
```

Figure 80. Method Used to Create Empty Image Files.

After views have been initialized, Profile Activity implementation checks if the default profile picture exists on the device (Figure 81). If it does, it sets it up at the ImageView.

After that, it tries to retrieve a profile picture from the Firebase Storage. If one exists, it downloads it and sets it to be displayed instead (Figure 82).



```
File image_file = new File(profilePicturePath, imageFileName);
if (image_file.exists()){
    Picasso.get() Picasso
            .load(image_file) RequestCreator
            .memoryPolicy(MemoryPolicy.NO_CACHE, MemoryPolicy.NO_STORE) RequestCreator
            .into(profilePicture, new Callback() {
                @Override
                public void onSuccess() {
                    Bitmap imageBitmap = ((BitmapDrawable) profilePicture.getDrawable()).getBitmap();
                    RoundedBitmapDrawable imageDrawable = RoundedBitmapDrawableFactory
                            .create(getResources(), imageBitmap);
                    imageDrawable.setCircular(true);
                    profilePicture.setImageDrawable(imageDrawable);
                }
                @Override
                public void onError(Exception e) {

                }
            });
}
editPicture(userFolder);
```

Figure 81. Check for Default Profile Picture.

```
private void editPicture(StorageReference pictureReference){
    pictureReference.getDownloadUrl().addOnSuccessListener(new OnSuccessListener<Uri>() {
        @Override
        public void onSuccess(Uri uri) {
            Picasso.get().load(uri.toString()).into(profilePicture, new Callback() {
                @Override
                public void onSuccess() {
                    Bitmap imageBitmap = ((BitmapDrawable) profilePicture.getDrawable()).getBitmap();
                    RoundedBitmapDrawable imageDrawable = RoundedBitmapDrawableFactory
                            .create(getResources(), imageBitmap);
                    imageDrawable.setCircular(true);
                    profilePicture.setImageDrawable(imageDrawable);
                }
                @Override
                public void onError(Exception e) {
                }
            });
        }
    });
}
```

Figure 82. “editPicture()” Method for Downloading Image from Firebase Storage.

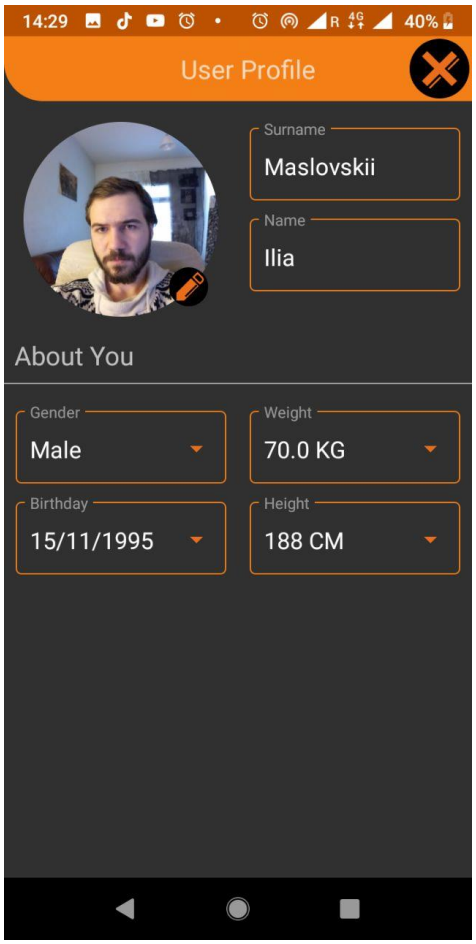


Figure 83. Profile Activity in Application.

5.7.5 User History Activity

User History Activity is implemented by UserHistory.java. User History Activity takes values stored in the Firebase Database and sets them up to be used in a historical graph view. This is achieved by taking average values over a specified timespan. The values are retrieved by subtracting said timespan representation in milliseconds from the current system time. All values higher than the resulting time representation are saved. The process is demonstrated in Figure 84 and Figure 85, where different parts of data aggregation implementation are shown.

```
public void getMonthlyAverages(long currentDayMillis, DataSnapshot dataSnapshot){
    long newDayMillis = currentDayMillis;
    for (int i = 0; i < 30; i++){
        int newDailyAverage = getDailyAverage(newDayMillis, dataSnapshot);
        if (i < 7){
            dataWeek.addEntry(new Entry(i, newDailyAverage));
        }
        dataMonth.addEntry(new Entry(i, newDailyAverage));
        newDayMillis -=dayToMillis;
        dailyAverage = 0;
    }
}
```

Figure 84. Function "getMonthlyAverages()" uses "getDailyAverage()" Method to Get Values from Data Snapshot.

```
    long newDayMillis = currentDayMillis;
    for(DataSnapshot snapshot : dataSnapshot.getChildren()){
        timestampMillis = Long.parseLong(snapshot.getKey());
        if(timestampMillis < newDayMillis && timestampMillis > (newDayMillis - dayToMillis)){
            Calendar sortCal = Calendar.getInstance();
            sortCal.setTimeInMillis(timestampMillis);
            int hour = sortCal.get(Calendar.HOUR_OF_DAY);
            hourMeasurementsInDay[hour].add(Integer.parseInt(snapshot.child("Temperature").getValue().toString()));
        }
    }
    int nonZeroHourlyAverages = 0;
    for (int i = 0; i < 24; i++){
        for (int j = 0; j < hourMeasurementsInDay[i].size(); j++){
            hourlyAverages[i] += hourMeasurementsInDay[i].get(j);
        }
        if (hourMeasurementsInDay[i].size() > 0) {
            hourlyAverages[i] = hourlyAverages[i] / hourMeasurementsInDay[i].size();
            if (newDayMillis == currentDayReference) {
                dataToday.addEntry(new Entry(i, hourlyAverages[i]));
            }
            if (hourlyAverages[i] > 0) {
                nonZeroHourlyAverages++;
            }
            dailyAverage += hourlyAverages[i];
        }
    }
    if(nonZeroHourlyAverages == 0){
        nonZeroHourlyAverages = 1;
    }
    dailyAverage = dailyAverage/nonZeroHourlyAverages;
    hourlyAverages = new int [24];
    hourMeasurementsInDay = new ArrayList[24];
    for (int i = 0; i < 24; i++) {
        hourMeasurementsInDay[i] = new ArrayList<Integer>();
    }
    dailyAverage = dailyAverage/nonZeroHourlyAverages;
    hourlyAverages = new int [24];
    hourMeasurementsInDay = new ArrayList[24];
    for (int i = 0; i < 24; i++) {
        hourMeasurementsInDay[i] = new ArrayList<Integer>();
    }
    return dailyAverage;
}
```

Figure 85. "getDailyAverages()" Function Implementation.

The Application uses Tab layout to allow the user to switch between the daily, weekly and monthly data display (Figure 86).

```
@AfterViews
void selectTabBehavior(){
    tabLayout.addOnTabSelectedListener(new TabLayout.BaseOnTabSelectedListener<TabLayout.Tab>() {
        @Override
        public void onTabSelected(TabLayout.Tab tab) {
            if (tabLayout.getSelectedTabPosition() == 0){
                historyChart.setData(new LineData(dataToday));
                historyChart.notifyDataSetChanged();
                historyChart.invalidate();
            } else if (tabLayout.getSelectedTabPosition() == 1){
                historyChart.setData(new LineData(dataWeek));
                historyChart.notifyDataSetChanged();
                historyChart.invalidate();
            } else if (tabLayout.getSelectedTabPosition() == 2){
                historyChart.setData(new LineData(dataMonth));
                historyChart.notifyDataSetChanged();
                historyChart.invalidate();
            }
        }
        @Override
        public void onTabUnselected(TabLayout.Tab tab) {
        }
        @Override
        public void onTabReselected(TabLayout.Tab tab) {
        }
    });
}
```

Figure 86. “selectTabBehavior()” Method that Adds Listeners to Tab Button in a View.

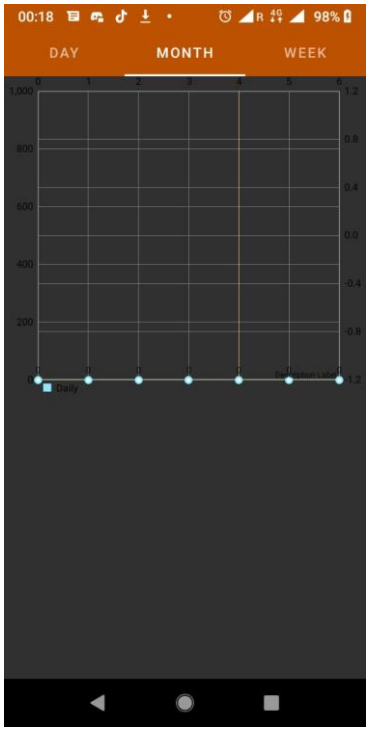


Figure 87. User History Activity in Application.

5.7.6 Settings Activity

Settings Activity is implemented by SettingsActivity.java. Is used to set up methods that interact with the Settings View. Due to the application limitation, the only active button with a method attached to it is “clickLogoutButton()”. It is used to change the application user. Upon clicking it, the user is logged out and the application launches Login Activity.

The implementation of activity is displayed in Figure 88.

```
@PreferenceScreen(R.xml.preferences)
@EActivity
public class SettingsActivity extends PreferenceActivity{
    private final String TAG = "SettingsActivity";
    public static String PREF_NAME = "RingPrefs";
    private FirebaseAuth mAuth;
    private FirebaseAuth.AuthStateListener mAuthStateListener;
    private FirebaseUser currentUser;

    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        mAuth = FirebaseAuth.getInstance();
        currentUser = mAuth.getCurrentUser();
    }


    public void clickLogoutButton (View v){
        mAuth.signOut();
        currentUser = mAuth.getCurrentUser();
        Toast.makeText( context: SettingsActivity.this,  text: "Clicked", Toast.LENGTH_SHORT).show();
        if (currentUser == null){
            Intent intent = new Intent( packageContext: SettingsActivity.this, LoginActivity_.class);
            startActivity(intent);
        }
    }

}
```

Figure 88. Settings Activity Implementation.

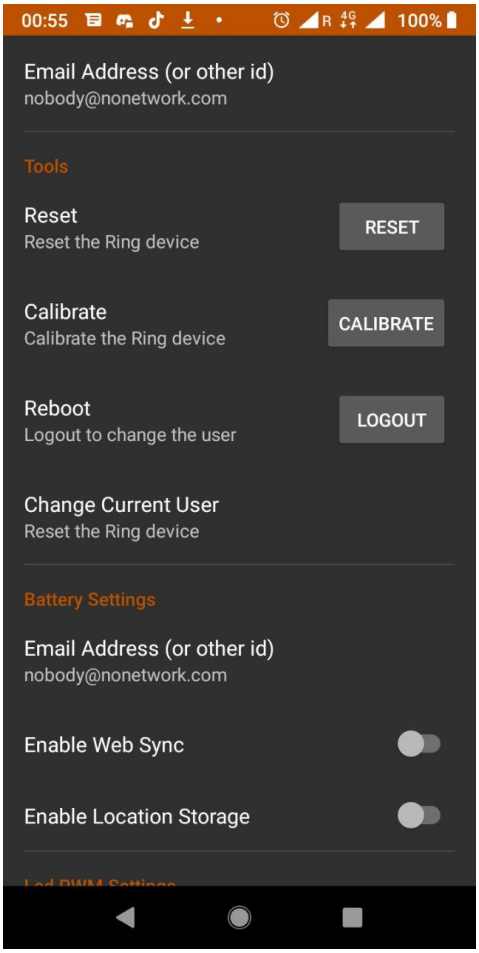


Figure 89. Settings Activity in Application.

6 TESTING

Testing has been conducted during multiple stages of the development. Whenever a new Activity has been implemented, the application has been compiled and run on the personal Android smartphone. Most of the debugging has been done either by observations of the developer, compiler messages, and, most importantly, using Android Debug Bridge (ADB) in conjunction with Android Studio. Once the smartphone is connected over USB, ADB allows to transfer Android application log over to Android Studio console, to run code changes without recompiling, and to use breakpoints. ADB testing is shown in Figure 90.

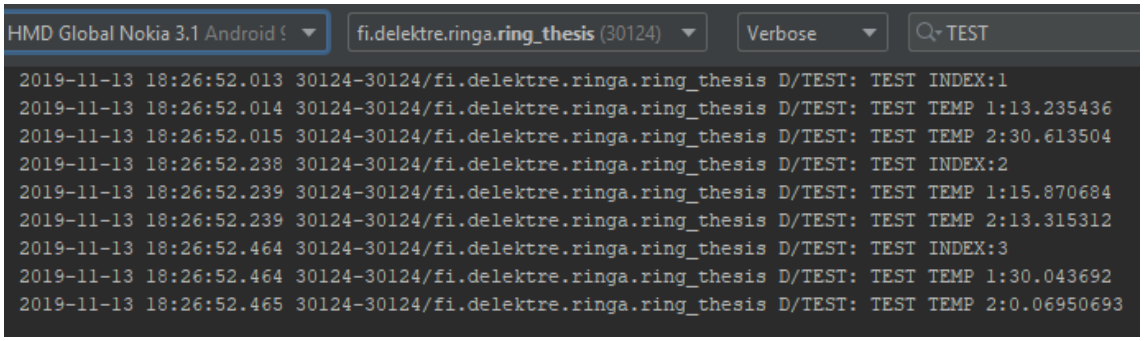


Figure 90. Testing Using Logcat and ADB

For testing the Firebase Database connection, multiple test users have been created during the development (Figure 91). At various stages of the development, the database has been manipulated directly from the Firebase Console, to check if the data is retrieved correctly when the client-side methods that use such data have been implemented. Image files have been uploaded to storage for testing purposes.

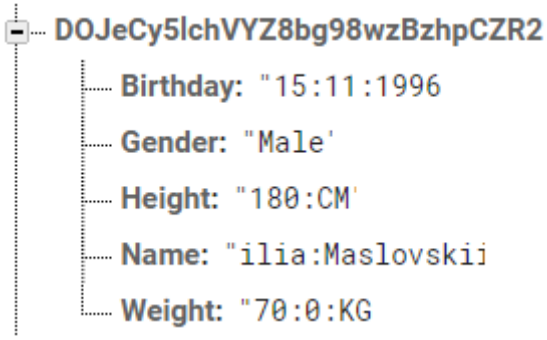


Figure 91. Mock User in Firebase Database

To test Bluetooth connectivity, a “KW41Z: Kinetis” development board manufactured by NXP has been used. It had sensors connected to it and was emitting the readings using BLE. The readings have been saved to a Firebase Database. LED readouts were exported to Excel spreadsheet for evaluation (Figure 92).

A	B	C	D	E	F
	BLUE_S	GREEN	NIR_S	RED_S	YELLOW_S
1.56E+12	184	201	180	172	245
1.56E+12	184	201	180	172	245
1.56E+12	184	201	180	172	245
1.56E+12	184	201	180	172	245
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	183	201	178	173	243
1.56E+12	184	201	180	173	244

Figure 92. LED Reading During Testing

Because of the project scope and nature of the thesis project (rapid prototyping), no unit testing was conducted. Only integration testing was done, which is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements. /21/

7 CONCLUSION

The Senno Application satisfies all the criteria set up for its development. It succeeds in being a demo that showcases the possibilities of Reactive Java and allows to receive, store and visualize data send to an Android phone over Bluetooth Low Energy connection. Senno Application uses Google Firebase to its full extent, whenever its usage is necessitated. The application runs a persistent background task which handles data management.

Major challenges during the thesis project included learning Android development and the application structure, subsequentially, learning frameworks and libraries related to the project and understanding how reactive programming works in a manner sufficient for completing the thesis.

The application could be improved extensively. Due to time constraints and the intended scope of application, not all features have been implemented to their fullest extent. Data processing remains rudimentary, and most classes require additional code that is necessary for edge use cases.

Personal benefits for myself are mainly learning about a new development platform, applying embedded concepts (asynchronous programming) to mobile development and about learning commonly used Android libraries. I have gained invaluable experience on developer applications with real-world uses.

8 REFERENCES

/1/ About Delektre Ltd. Accessed 12.11.2019

https://www.linkedin.com/company/delektre-oy/about/

/2/ About Motiv. Accessed 13.06.2019

https://www.linkedin.com/company/motiv-inc/about/

/3/ About Oura. Accessed 13.06.2019

https://ouraring.com/forpress/

/4/ Java (Programming Language). Accessed 25.05.2019

https://en.wikipedia.org/wiki/Java_(programming_language)

/5/ RxAndroid. Accessed 12.06.2019

https://github.com/ReactiveX/RxAndroid

/6/ Android Studion. Acceesed 14.06.2019

https://en.wikipedia.org/wiki/Android_Studio

/7/ Introduction to Firebase. Accessed 15.06.2019

https://hackernoon.com/introduction-to-firebase-218a23186cd7

/8/ Introduction to Android Studio. Accessed 15.06.2019

https://developer.android.com/studio/intro

/9/ Gatt Services Description. Accesed 30.08.2019

https://www.bluetooth.com/specifications/gatt/services/

/10/ Vitals Measurement. Accessed 12.11.2019

https://my.clevelandclinic.org/health/articles/10881-vital-signs

/11/ Pulse Oximeter. Accessed 12.11.2019

K. Shelley and S. Shelley, Pulse Oximeter Waveform: Photoelectric Plethysmography, in Clinical Monitoring, Carol Lake, R. Hines, and C. Blitt, Eds.: W.B. Saunders Company, 2001, pp. 420-42

/12/ Android, Activity. Accessed 5.09.2019

https://developer.android.com/reference/android/app/Activity

/13/ Android, Fragments. Accessed 13.10.2019

https://developer.android.com/reference/android/app/Fragment.html

/14/ Android, Services Implementation Guide. Accessed 9.09.2019

https://developer.android.com/guide/components/services

/15/ ReactiveX, Observable Pattern. Accessed 8.10.2019

http://reactivex.io/documentation/observable.html

/16/ Android, Declaring Layout. Accessed 5.11.2019

https://developer.android.com/guide/topics/ui/declaring-layout

/17/ MpAndroidChart Core Features. Accessed 6.11.2019

https://weeklycoding.com/mpandroidchart/core-features/

/18/ I.T., Titanium. "James Shore: Dependency Injection Demystified. Accessed 27.10.2019.

www.jamesshore.com

/19/ Android Annotations Cookbook. Accessed 10.11.2019

https://github.com/androidannotations/androidannotations/wiki/Adapters-and-lists

/20/ An Algorithm for Error Correcting Cyclic Redundance Checks. Accessed 11.11.2019

https://web.archive.org/web/20170720165847/http://www.drdobbs.com/an-algorithm-for-error-correcting-cyclic/184401662

/21/ ISO/IEC/IEEE International Standard - Systems and software engineering. ISO/IEC/IEEE 24765:2010(E). 2010. pp. vol., no., pp.1–418 Accessed 13.11.2019