Software Engineering 2

Implementation and Testing Document

Version 1.0

Andrei Constantin Scutariu	833370
Carlo Pulvirenti	828459
Sergio Piermario Placanica	916702

POLITECNICO
MILANO 1863

Academic Year 2018-2019

Link to github repository
https://github.com/xnand/ScutariuPlacanicaPulvirenti

Link to sourcecode
https://github.com/xnand/ScutariuPlacanicaPulvirenti/tree/master/ITD

Contents

1 Introduction **3**
1.1 Scope . . . 3

2 Requirements **4**
2.1 Implemented goals and requirements . . . 4
2.2 Partially or not implemented funcionalities . . . 5

3 Development Frameworks **6**
3.1 Frameworks and programming languages . . . 6
3.1.1 Web server, Application Servers and Database Server . . . 6
3.1.2 Mobile client . . . 6
3.2 Middleware software . . . 7
3.3 Dart Plugin . . . 7

4 Structure of the Source Code **8**
4.1 TrackMeServer . . . 8
4.1.1 DatabaseServer . . . 9
4.1.2 ApplicationServerMobileClient & ApplicationServerData4Help . . . 9
4.1.3 WebServer . . . 9
4.2 MobileApp & WearOS app . . . 9
4.2.1 Flutter code . . . 9
4.2.2 Platform specific code (Java) . . . 10
4.2.3 TrackmeWear . . . 11
4.3 Notable algorithms . . . 11
4.3.1 Blood pressure . . . 12

5 Testing **13**
5.1 Backend server . . . 13
5.2 Mobile App . . . 13

6 Installation Instruction **14**
6.1 Backend server . . . 14
6.1.1 Automated tests . . . 15
6.1.2 Support files for evaluation . . . 15
6.1.3 Email credentials . . . 16
6.2 MobileApp TrackmeMobile . . . 16
6.3 MobileApp TrackmeWear . . . 17

1 Introduction

1.1 Scope

This document is a follow up on the previous two, "Requirement Analysis and Specification Document" and "Design Document": it will describe the prototype for TrackMe's system that has been developed in accordance to them.

The document is structured in the following way:

Chapther 2 Will present the goals and requirements implemented in the working prototype.

Chapther 3 Will present the frameworks adopted for the development and testing process.

Chapther 4 Will present and explain the structure of the source code.

Chapther 5 Will give information on the testing process.

Chapther 6 Will give instructions on how to compile and run the application components for evaluation.

2 Requirements

2.1 Implemented goals and requirements

All goals and requirements of Data4Help and AutomatedSOS have been implemented, at least partially, in the prototype.

Data4Help

G1 Third parties shall be able to make requests for accessing single customer data.

G2 Third parties shall be able to make requests for accessing aggregate anonymous data specifiying filters.

G3 Third parties shall be able to access stored data, for which a request has been approved.

G4 Third parties shall be able to subscribe to new data, for which a request has been approved.

G5 Users shall be able to accept or refuse requests for their data from third parties.

G6 Access to customers data from a third party, for which a request does not exist, or has been rejected, shall be denied.

R1 The system must forward any request from a company to single customer's data to the corresponding user.

R2 The system must reject any request for data regarding a group of customers when it cannot guarantee the anonimity of its components.

R3 The system must periodically collect and store customer's data.

R4 The system must periodically update the accessible data with the newly collected one

R5 * The system must notify the user of the request and permit him to accept or refuse it.

R6 The system must update the request status with the answer provided by the user.

R7 The system must be able to identify which data can be accessed for each third party company.

AutomatedSOS

G7 * From the time a customer's parameters indicate a health emergency status, an ambulance shall be dispatched to his location in less than 5 seconds.

G8 When an ambulance is dispatched, the customer shall be notified of its arrival.

R8 The system must continuously check the data read from customers subscribed to AutomatedSOS.

R9 * In case the data indicate an emergency for a customer, the system must dispatch the closest ambulance to his location.

R10 After an ambulance is dispatched, the system must notify the customer of its arrival.

***** Partial implementation

2.2 Partially or not implemented funcionalities

Being this a prototype, not all funcionalities have been fully implemented. Following are the most notable ones:

- All features of Track4Run are not implemented.
- Even if the Mobile App component was developed with Flutter, a cross platform mobile framework for iOS and Android, native code was required for the communication with the wearable device; only the Android platform is supported at this stage.
- The Mobile App component works and was tested only with the android Wear OS emulator. Being that on the emulator the sensors are not accessible, a simulator is provided with sliders to adjust the values sent to the mobile application.
- AutomatedSOS dispatching feature only works with the provided ambulance dispatcher simulator.
- The activation of the AutomatedSOS service is not implemented; instead, the service is assumed to be already activated on all the registered devices.
- The subscription request within the request for data (specific or group request) has not been implemented; companies can currently only subscribe to a request after it has been made.
- Notification manager components are not implemented. This is not a problem for this protype though, as each resource that was to be notified is automatically synced through components accessing the proper funcionality (for example, new requests for data can be viewed by querying the system, and that's automatically done when opening the requests page on the Mobile App).
- There is no proper address or geolocation resolution and validation.
- Optimizations for large amount of data transfers, such as when sending user's data for a group request that includes many users, have not yet been implemented.
- Some minor funcionalities are left as todo in the source code, intended for completion on further development, and should not significantly impact the behaviour of the system, if at all.

3 Development Frameworks

3.1 Frameworks and programming languages

With regard to the Component View presented in the Design Document, various frameworks and programming languages were adopted for development, depending on the component and the platform on which it runs on.

3.1.1 Web server, Application Servers and Database Server

These components form the back-end of the system.

Node JS with express and Javascript were used for development, and all the exchanged data that forms the model of the system is stored in a database. This combination performs very well in asynchronous and event-driven operations, and is exceptionally suited for applications where a large number of users operate simultaneously with short data payloads (eg. sending user's body parameters), while also having very good performance with larger data payloads (eg. company accessing a large group of customer's data). The two Application server and the Database server components expose a REST API that allows them to communicate with each other and with the Mobile client component. Also, this allows the service to be used with different interfaces without any problem. For example, the Application Server Data4Help API that exposes functions for companies, could be used with any software or framework and/or programming language capable of communicating through standard HTTP protocol, or even with any modern web browser. This holds true for the Application Server Mobile Client, though the functionality that handles user's parameters gathered from a wearable device limits the usage of certain functions to an interface developed specifically for TrackMe's service.

All data is exchanged in JSON format, being platform independent and well supported by all frameworks.

Furthermore, this approach does not require a constantly open network connection between a user's device and the server for the entire interaction process, thus partially avoiding problems with limited simultaneous open connections and allowing a fair queueing mechanism in case of overloading.

3.1.2 Mobile client

TrackMe mobile client was developed for the Android Platform with a combination of the new Google's mobile app SDK "Flutter 1.0" and Platform specific code. Flutter is an app SDK for crafting high-quality native experiences on iOS and Android using the Dart programming language. Thanks to flutter, TrackMe could be shipped also on IOS with minimum effort being almost all the code written in Dart sharable between platforms. Java was used for the WearOS application and for the interaction between TrackMe smartphone app and WearOS device. In particular, the reception of infopackets and the subroutine that send infopacket to the Mobile Client were necessarily developed using Android specific code. The former because currently there's no Dart library specifically developed for the communication between a smartdevice and the handleheld counterpart, the latter because during development it has been noticed that

Flutter was not well suited to handle continuos tasks in background.

3.2 Middleware software

Various middleware platforms were used:

Knex.js Query builder for managing the DBMS, used in the Database Server for creating database tables, creating, deleting, updating columns and querying the database. Also allows changing DBMS software editing just its connection settings.

PostgresQL DataBase Management System.

Mocha, Chai Testing frameworks integrating well with Node js and express.

Swagger For generating interactive documentation of the REST APIs.

3.3 Dart Plugin

Shared preferences 0.4.3 Wraps NSUserDefaults (on iOS) and SharedPreferences (on Android), providing a persistent store for simple data.

http 0.12.0 A set of high-level functions and classes that make it easy to consume HTTP resources.

datetime_picker_formfield 0.1.7 Two Flutter widgets that wrap a TextFormField and integrates the date and/or time picker dialogs.

country_code_picker 1.1.1 A flutter package for showing a country code selector.

4 Structure of the Source Code

The root directory of the source code is "ITD". Inside of it there are three folders:

ITD doc Source code of this document.

TrackMeServer Source code of the back-end server: Application Server Mobile Client, Application Server Data4Help, Database Server and Web Server components.

MobileApp Source code of the mobile application and wearable module.

4.1 TrackMeServer

The macro components presented in the Design Document, except for the Mobile App, are the four different modules found in this directory, while the inner components take the form of functions and REST endpoints. All the four modules share the same basic structure: inside the component's directory (eg. ApplicationServerMobileClient) there is the main javascript file to be launched with node (eg. ApplicationServerMobileClient.js), the documentation source for the component if available (doc.yml), and a routes directory that futher defines funcionalities provided by the component.

The directory named "common" contains two files used by the components:

config.json Contains configuration variables that can or must be set for the components to work, and can be overridden by environment variables (see chapter 6).

common.js Contains functions often used by all other components; they are defined here to avoid code repetition and ease the development process.

One more directory named "support" contains files useful for development and evaluation. Three python scripts and a standalone node (javascript) file. The python scripts are written and tested for python 3 version 3.7.2 and require the "requests" package:

cleanDatabase.py Simply performs a GET to /dropALL on the DatabaseServer component to reset the database tables (see the DatabaseServer component description below).

populateDatabase.py This script uses the ApplicationServerMobileClient and ApplicationServerData4Help components to register random model data in the system as fast as possible, running on multiple threads.

subscriptionHandler.py A simple server to listen and write data to simulate a company subscription's funcionality.

AmulanceDispatcherSimulator/dispatcher.js Node JS server that simulates the external ambulance dispatcher simulator, required for AutomatedSOS funcionality.

Every main component javascript file sets up the environment and binds the service to the address provided by the config file (or environment variable).

4.1.1 DatabaseServer

This component has one more file, "knex.js", that sets up the connection with the DBMS.

Inside the main file the tables of the database are created, if they don't exists, then the files in "routes" are sourced for the various funcionalities. Also, a temporary endpoint is defined (GET to /dropALL), only for development and evaluation purposes, that can be used to drop all current tables and recreate them empty.

The three files in "routes" directory contain the endpoints relative to their name, for example "dbs_user.js" contains all the endpoints to handle user informations.

4.1.2 ApplicationServerMobileClient & ApplicationServerData4Help

As the DatabaseServer component, the main file set up the environment and binds the service to the provided address; these two components also provide an endpoint (GET to /docs) to the documentation of their API. Then they source the .js file in "routes" folder that contain the implementation of their endpoints funcionality.

The doc.yml is the source code of the documentation, read by swagger when starting the component through the swagger-jsdoc node module.

4.1.3 WebServer

The web server has a main file, WebServer.js, that set up the environments and configures an object useful to send emails named "trasporter" and the relative logic realized thanks to a Node.js module called "nodemailer". In the folder "public" there are the html pages that the third party comapany will use, in particular "register.html", to register to the service with a form. The contents of the fields are checked with the javascript file "javascripts/validationForm.js" before sending the email. The style of web pages is defined by a css file "stylesheets/style.css".

4.2 MobileApp & WearOS app

MobileApp is composed of two submodule, Flutter code and the android platform specific code.

4.2.1 Flutter code

as shown in Fig. 1, the structure of the code is pretty simple

controllers Contains the class channelController, responsible for platform specific methods invocation. In brevity, this class link Flutter code with Java code.

models A modelization of the database classes, made easier to manage data received trough HTTP requests.

profileManager As seen in the DD, this component will handle all the functionalities regarding

the user, such as registration, login, registering a new Wearable device etc. This is where the main logic of the app is located

screens This is were Flutter shines, those are the screens of the application, a collection of statefull and stateless widgets.

styles files with useful font and theme that give a distinctive style to the application

utils a collection on useful methods and class. CountryCodePicker is a custom version of the dart plugin while validators are the function used to verify the user form inputs.

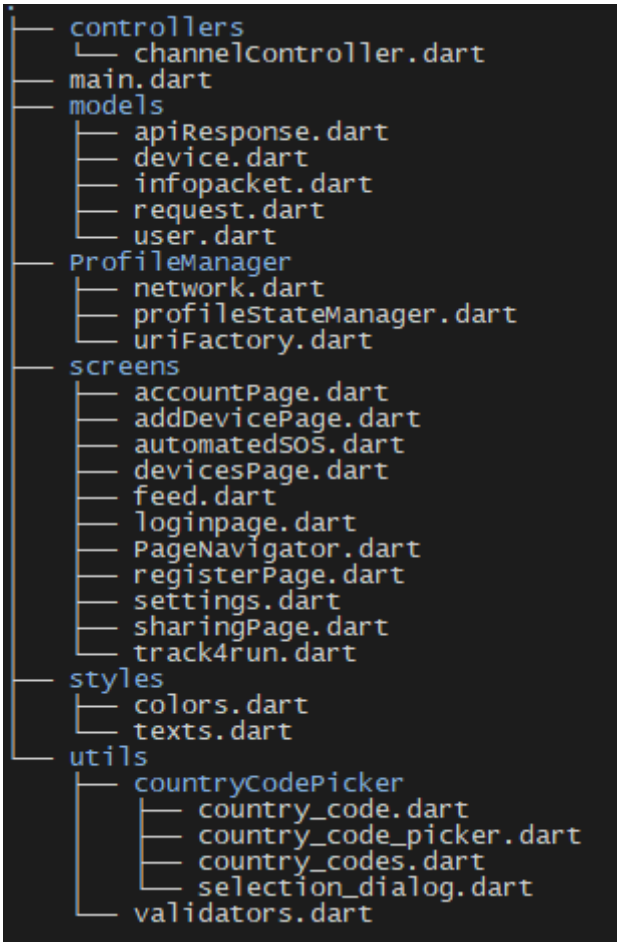


Figure 1: tree view for the lib folder

4.2.2 Platform specific code (Java)

Is composed of two macro classes located in
"../MobileApp/TrackmeMobile/trackmemobile/android/app/src/main/java/com/trackme/trackmemobile"

MainActivity.java Launch the application and contains a CommunicationPlugin class that handle the platform call made by Flutter.

InfoPacketHandler.java Handle the connection with the WearOS device, receive the infoPackets on a nonBlockingQueue of fixed size and dispatch the received infoPacket to the TrackMe mobile application server (Fig2), the system is capable of handling a wearsOS device disconnection or a Mobile app disconnection.

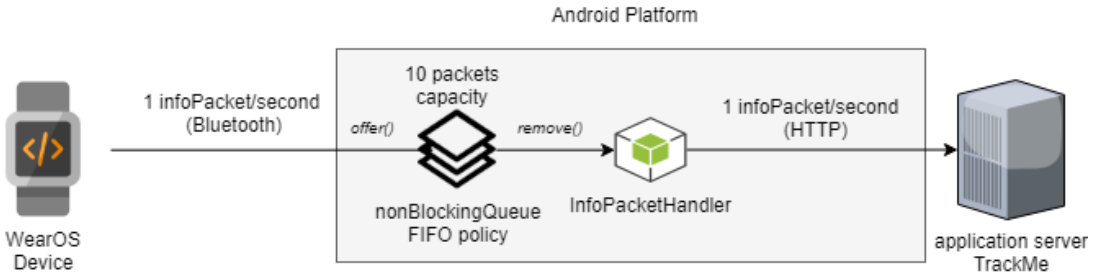


Figure 2: infoPacket "cycle of life"

4.2.3 TrackmeWear

The WearOS app was not planned in advance. During the development the team encountered an unexpected problem: for some reason it is impossible to simulate live monitoring sensors on a WearOS device emulator (see reference). After some discussion about the problem, it was decided to develop a small companion application to simulate live monitoring sensors, in particular hearbeat, systolic and diastolic blood pressure. Thus, The application establish a connection with the smartphone and start an infinite stream of infoPackets with a THR of 1 packet/second, data is sent through a simulated bluetooth protocol. The user can modify data values using the sliders on the screen of the WearOS device. This simple app is written totally in Java.

MainActivity.java Start the application, handle the connection with the smartphone and initialize and update the screen with the sliders.

MakeInfoPacket.jave A collection of static methods to craft the infoPacket.

MobileWearableService connect to the Smartphone.

4.3 Notable algorithms

The source code is commented and should not be difficult to follow and understand; however, there are some more elaborate portions for which could be useful a little explanation:

- Filters for group requests, passed in the POST body, are parsed in groups: each group of filters has a number associated to it that goes after the filter's name, for example:

  ageStart1=18&ageEnd2=25&street2=risorgimento

are two different groups, respectively group 1 and 2. Multiple filters can be specified per group, and if no number is specified it is assumed to be number 1. Each filter inside the group is in a logical AND relation with all the other filters in the same group, while the different groups are in logical OR relation with all the other groups. So, the example above would search for users that (are at least 18 years old) OR (are at most 25 years old AND live in risorgimento street).

- Subscriptions for authorized requests require that the company set a server capable of receiving data from TrackMe's server, sent as POST requests containing user's data in JSON format, and must respond to every request with status code 200 (an example of such server is provided in TrackMeServer/support/subscriptionHandler.py). It must provide a reachable link to such service when subscribing. If such server is not reachable or won't respond at the moment of subscribing, the operation will fail. If the subscription is succesfully activated, and the company's server then stops satisfying the above requirements, the subscription will be automatically deactivated.

4.3.1 Blood pressure

Although the specific of the project states that "an ambulance must be dispatched in t¡5 seconds from the start of an emergency", we thought it was mandatory to implement a system capable of discerning between an emergency and a casual fluctuation in the readings. A simple yet effective solution was to assign a "weight" to systolic and diastolic values and increment a variable counter if the data received is offscale. We can then confront the counter with a threshold variable and send an infopacket if and when the "THR ¡ counter". To make the system resistent to casual fluctuation, when a perfectly normal readings is received we decrement the counter of an appropriate amount

5 Testing

All components were tested as they were being developed. The tools used for testing vary based on the component's platform.

5.1 Backend server

The components of the Backend server were tested with Node js through the use of Mocha and Chai modules, and with Postman software tool. Every REST endpoint and funcionality was tested after its development, among the previously developed ones.

Depending on the functionality, a test case could include one or more endpoints/funcions, or even more than one component, thus ensuring the correct operation of all the agents involved.

Mocha tests are defined in the test/test.js file, while a simple Postman collection is provided in test/postman.json. See chapter 6 for instructions on how to use them.

Stress testing can be simply performed stopping one or more components in the middle of some operation. The system is very resistant to failures, by design: while many funcionalities need some other component to operate, particulary the Database Server and the DBMS, they can be running isolated by the others and simply answer with an error message when an operation can not be completed successfully; as soon as the required components come back up, all the funcionalities return fully functional. This is allowed by keeping track of every successfull operation in the database, performing atomic data transactions between components where every endpoint has only one "success" point where it modifies the state of the system, and by gracefully handling errors.

The support/populateDatabase.py script can also be used for load/stress testing, being that it runs in multi-threaded mode. Even in such a running environment, the system performs well, keeping the response time of most funcionalities around 60ms on average, while the requests sending large payloads take a little longer (around 4500ms for 300MB of data). These measures were done launching the script on 4 cores (on a quad-core CPU) with all components hosted on localhost.

5.2 Mobile App

Mobile app model was tested and integration testing with the database was performed using postman.

6 Installation Instruction

The system was fully tested only on linux environment, for which the following instructions will be valid, though it should run on all major platforms with proper adjustments. To evaluate all components of this prototype, at least these software packages must be installed and working on the system in use:

- Node JS (11.6.0)
- npm (6.5.0)
- Flutter (1.0.0 - rev. 5391447fae)
- Dart (2.1.0)
- JDK/OpenJDK 8
- Android Studio or Intellij Idea - for the wearable application
- adb tool

In parenheses is indicated the version used in development. Following are the steps to launch the various components:

6.1 Backend server

- Eventually import the project ITD/TrackMeServer in Android Studio / Intellij Idea (not required).
- Navigate to ITD/TrackMeServer and run "npm install" to download the required dependencies (defined in package.json).
- Edit the common/config.json file with address and port on which each component should listen for incoming connections.
- Edit the common/config.json file setting to "true" the postgres.useHeroku variable to use an already set up DBMS. It is though advised to use a local DBMS connection (PostgresQL) if possible, since it allows to browse the data in the database. In this case, set the useHeroku variable to "false" and properly set the other three variables.
- Run the components by executing
  "node DatabaseServer/DatabaseServer.js &
  node ApplicationServerMobileClient/ApplicationServerMobileClient.js &
  node ApplicationServerData4Help/ApplicationServerData4Help.js &
  node WebServer/WebServer.js &"
  or by launching them from the IDE.

Note that the working directory for the processes must be ITD/TrackMeServer, otherwise some necessary files won't be found. Also, each module can be launched on a different directory, or a different host, as long as the "common" and "node_modules" directories are present along the component's one.

Also, every variable defined in config.json can be overridden with environment variables. The structure of such variables is equal to the ones in the configuration file, with the following differences: all character are uppercase, and to address a sub-variable, use the underscore. For example, to temporarily use the heroku DBMS, one could launch the DatabaseServer like this: POSTGRES_USEHEROKU=true node DatabaseServer/DatabaseServer.js

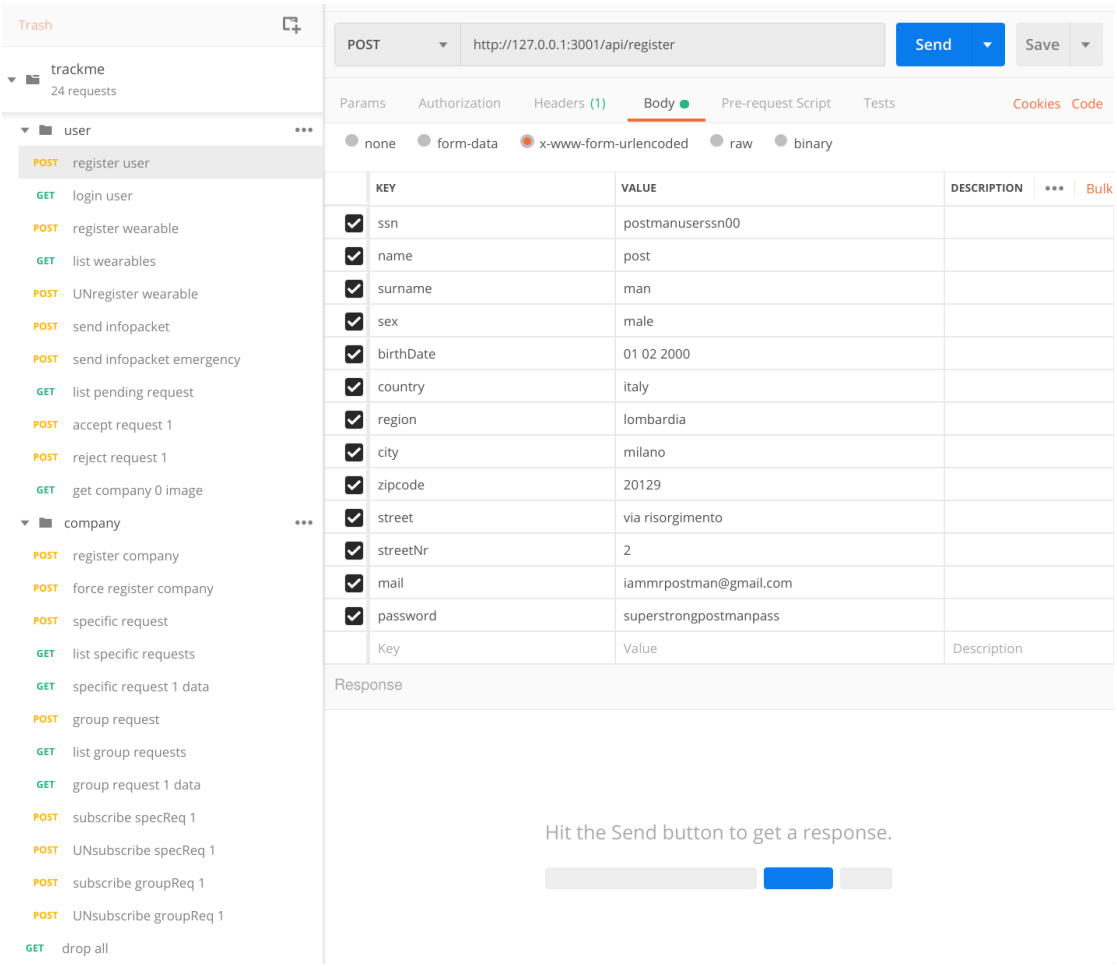
6.1.1 Automated tests

After the Mocha and Chai have been installed, the tests defined in test/test.js can be run by navigating to ITD/TrackMeServer and running "./node_modules/mocha/bin/mocha test/test.js", or by launching a proper Mocha run configuration from the IDE. Note that these tests require the three major components of the backend server to be running; also, after the tests have been run once, the database needs to be reset, otherwise some of them will fail.

6.1.2 Support files for evaluation

Inside the "support" directory there are some files useful for evaluating the project, as stated in chapter 4. To launch the python scripts, just run "python3 *path-to-py-file*". Note that they also need the common/config.json file to be in the working directory.

A Postman configuration, that was also used for testing, is also supplied. To import it, just launch Postman, then click import and open postman.json file. The following functions are defined:



6.1.3 Email credentials

The mail box where registration requests are sent can be checked using the following credentials:

email: wtrackme@gmail.com

password: Data4.Help

6.2 MobileApp TrackmeMobile

- Eventually import the project ITD/MobileApp/TrackmeMobile/trackmemobile into Android Studio / Intellij Idea (not required).
- Edit the file trackmemobile/lib/ProfileManager/network.dart and set the _url variable with the ip address on which the Application Server Mobile Client component will be reachable

(this won't be 127.0.0.1!). At the same way edit the variable basicUrl in trackmemobile/android/app/src/main/java/com/trackme/trackmemobile/InfoPacketHandler.java.

- Assuming Flutter SDK, Android SDK and Dart are installed, and Flutter's bin directory is in PATH environment variable, navigate to ITD/MobileApp/TrackmeMobile/trackmemobile.
- Run "flutter build apk –debug"; this will generate the apk in build/app/outputs/apk/debug/app-debug.apk
- Install the apk on the device (eg. with "adb install *path-to-apk*") and run it.

Alternatively one can build and install the apk from the IDE.

6.3 MobileApp TrackmeWear

- Import the project ITD/MobileApp/TrackmeWear into the IDE (required!).
- Create an Android App run configuration setting the Module field to "app", if does not exist.
- Launch an Android Wear virtual device.
- Run the app on the virtual device.
- Run in a terminal "adb -d forward tcp:5601 tcp:5601", to allow the wearable to communicate with the Android phone, and connect the smartphone to adb. Make sure it is listed as "device" when running "adb devices", not as "unauthorized".
- Download, install and run the Wear OS application from the play store on your Android phone, and pair it with the emulator.
- Run the wearable application on the emulator.

Note that the wearable device emulator won't connect with a smartphone emulator, so a real smartphone and the wearable emulator must be used.