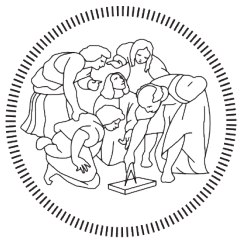


POLITECNICO
MILANO 1863

ITD

Software Engineering 2 Project - TrackMe

v1.0 - 13/01/2019

Authors

- Daniele Montesi - *912980*
- Nicola Fossati - *915244*
- Francesco Sgherzi - *915377*

Contents

1 Introduction

1.1 Purpose

This Document represents the Implementation and Testing Document for the Data4Help system. The purpose of this document is to provide a comprehensive overview of the implementation and testing activity for the development of the application. The main recipients of this document are the project manager, developers and testers.

1.2 Scope

As described in the RASD document and in the DD document, the Data4Help service is made of four distinct components: three clients (Data4Help Smartwatch App, Data4Help Mobile App, Website) and a server application (Data4Help Core).
The users, who are divided into Companies, Individuals and Run organizers, can subscribe to the service and take advantage of the functionalities offered.

Data4Help App aims at providing a intuitive user interface that allows users to see their health data, coming from smart devices such as Smart-watches, and participate to a Run organized by Data4Help. Data4Help website aims giving to Third parties(i.e. the companies) a portal to access to the system functionalities such as query on user's data.

During the whole document, we will make several cross-references to the RASD and DD documents , in order to maintain a certain degree of coherence and consistency.

1.3 Definitions, Acronyms, Abbreviations

i.e.	*Id est*, that is
w.r.t	with respect to
Company	Third party company
UI	User Interface

1.4 Revision history

1.5 Reference documents

- **|REFD1| MVP**
- **|REFD2| IEEE Std 1016-2009 Standard for Information Technology, Systems Design, Software Design Descriptions**
- **|REFD3| Representational State Transfers**

1.6 Document structure

This document is divided in the following chapters:

Implemented requirements Explains which of the functional requirements described in the RASD are accomplished.

Design choices provides reasons about the implementation decisions taken in order to develop the application.

Source code structure explains and motivates how the source code is structured both in the front end and in the back end.

Testing provides the main testing cases applied to the the application

REST API describes the API implemented for the application, with some example requests and responses.

2 Implemented requirements

In this section there are described the functionalities implemented w.r.t the requirements listed in the RASD document. Below there are listed all the functionality of the system, explained in terms of Database, Frontend, Back-end handling.

2.1 Actor Registration

- **Implemented** requirements
  - **[$\mathbf{RM}_M$]**: Users can register, after providing a username, a password and a Fiscal Code/Social Security Number and have connected a compatible Smartwatch
  - **[$\mathbf{R2}_M$]**: Users can only have one account
  - **[$\mathbf{R11}_W$]**: Run organizers can register
  - **[$\mathbf{R1}_W$]**: Companies can register, after providing a username, a password, an email and a company name
  - **[$\mathbf{R14}_C$]**: Can validates information provided by the user during registration
- **Non-implemented** requirements
  -

Database
The database stores the actor data in the table "account". Passwords are hashed and salted in order to avoid a security flaw. The specific data for each type of actor is stored in specific tables, each for every actor type: individual_account, company_account, run_organizer. If the registering actor has, for instance, the same email of an actor already present on the database, the transaction is rolled back. The same can be said for other `unique` parameters, such as the `SSN` for the `individual_account`

Back-end
The server first checks the required parameters for registration, depending on the actor, hashes the password and performs the first insertion into the table reserved for the actor (namely `individual_account` `run_organizer_account` `company_account`). Then generates a token which is stored and then sent to the actor mail in order to verify the account.

Front-end

- **Application**: A user can register a new account after having decided whether to access the Data4Help service or Track4Run service.
- **Website**: Company can register through the homepage clicking on a button that opens a modal. After inserting the data, company clicks 'Send' and becomes automatically registered requiring an email verification.

2.2 Actor Authentication

- **Implemented** requirements
  - **[R1$_M$]**: Users can log-in
  - **[R12$_M$]**: Run organizers can login
  - **[R2$_W$]**: Companies can log-in
- **Non-implemented** requirements
  -

Database
The given actor is requested from the database, if the comparison of the stored password and the given one is successful a token is inserted into the `user_token` table.

Back-end
The server checks the required parameters for the request, then proceedes to compare the hashed and non-hashed provided password. If the compare is successful it creates a token and stores it into the database, otherwise it negates the access to the actor.

Front-end

- **Application**: The application contains, under each subsection, a Login form. The application validates the information, performs the query to the server and show the result of the login. Both Users (individuals) and Run Organizers can log in using the application.
- **Website**: Company can login through the homepage clicking on a button "Sign in" that opens a modal. After inserting the data, company clicks 'Send' and access to the dashboard of his account.

2.3 Individuals Management

- **Implemented** requirements
  - **[R6$_M$]**: Logged-in users can accept/decline a company individual monitoring request
  - **[R2$_C$]**: Can send online notifications via email to all users
- **Non-implemented** requirements
  - **[R1$_C$]**: Can send online notifications via SMS to all users
  - **[R3$_M$]**: Logged-in users can edit their account info
  - **[R5$_W$]**: Logged-in companies can update their account information
  - **[R3$_C$]**: Can send online notifications via the Mobile app to its users
  - **[R3$_W$]**: Logged-in companies can see their history and account information

Database
The authorizations for each individual monitoring request is stored in the table individual_query, in the column "auth", which is boolean (true if accepted, false if rejected, null if not already evaluated). Back-end accesses column "email" in table account in order to send emails to the users.

Back-end
The `/queries/query/individual/pending` endpoint allows the user to accept or decline the pending monitoring request. The server sends online notification to every actor when he registers, in order to verify the email, and to the company that has performed a query when new data are available.

Front-end

- **Application:** Individual users can accept or decline a company monitoring request by the section reachable through the navigation drawer in the Dashboard.

- **Website:** Company can see all account information in the panel 'Account' accessible via the navbar through all the pages, after having logged in.

Comment on not implemented requirements
We decided to not implement the editing of registered account because we asked users for details that are less subject to variation (for instance name, surname, company name, SSN...).
Instead of sending SMS/push notification, we decided to send an email to the users since the integration with an external service providing email was easier in this stage of the project.

2.4 Company plan subscription

- **Implemented** requirements
  -
- **Non-implemented** requirements
  - **[R8$_C$]**: Can provide a payment method for data requested by companies.
  - **[R15$_C$]**: Can charge companies on their payment method respecting Track4Me pricing policy

Database
The server answers with the mockup.

Front-end
Web site: The page is only a mockup, user can see all subscription plan available by clicking in the panel 'Account' accessible via the nav bar through all the pages, after having logged in. Company can start the purchasing process after clicking "Purchase" under the respective subscription plan.

Comment on not implemented requirements
We decided not to implement the plan subscription since it required an external payment service that, at this stage of the project, required too much effort. For the moment, the component is active but gives just a mockup response

2.5 Data management

- **Implemented** requirements
  - **[R4$_C$]**: Can save and store permanently user data
  - **[R5$_C$]**: Can receive and store health parameters and geographical position of registered users
  - **[R2$_S$]**: App can send data registered locally to Data4Help Mobile App

- **Non-implemented** requirements
  - **[R1$_S$]**: App can read data from sensor and store it locally.
  - **[R5$_M$]**: Logged-in can specify the nature of the daily activities (i.e. running, biking, swimming, hiking)

Database
User data is stored in the tables "accelerometer", "gps_coordinates", "hear_rate", "user_data".

Back-end
The individual user can send his data via the indiv/data endpoint. The server inserts the user data into the respective table, if well formatted. Then proceedes to notify the company for new data if the user is in one of the previously posted query.

Front-end

- **Application:** The data is not actually read from the sensors but the app can generate dummy test data and send it to the server, by the Test option in navigator drawer in the Dashboard.

- **Web site:** Company can see data relative to their queries or to their monitored individuals right after having performed a query, clicking in the link provided by the server that redirects the user to a XML file containing the data requested.

Comment on not implemented requirements
For this stage of the project we decided that implement a smart-watch app able to read sensor data was too costly, so we made a stub component that generates data.

2.6 Query management

- **Implemented** requirements
  - **[R6$_W$]**: Logged-in companies can query on some group of individuals data
  - **[R7$_W$]**: Registered companies can request access to data of individuals
  - **[R8$_W$]**: Logged-in companies can access to an individual data, if the user has given approval

- **[R9$_W$]**: Logged-in companies can export data previously queried using Data4Help
- **[R6$_C$]**: Can execute queries of companies on individuals if the user has accepted the monitoring request from the company
- **[R6bis$_W$]**: Logged-in companies can subscribe to a query data
- **[R7$_C$]**: Can execute queries of companies checking if the searches involve more than 1000 anonimyzed users

- **Non-implemented** requirements
  -

Database
General information of queries are saved in table "query", while in tables "query_user", "radius_query" and "general_query" are saved the specific information of queries. The list of users subjected to the query is stored in the `query_user` table.

Back-end
First, the given query is performed to assess feasibility (i.e on a high enough number of user) and then it is inserted in the database. The list of users subjected to the query is inserted into the database as well.

Front-end - Website
Company can perform a query for group of individuals in the panel 'Query' accessible via the navbar through all the pages, after having logged in.
In the Query page it will appear a button to start a query. When clicked, it opens a modal with all fields to filter query parameters. If left empty, the filter is just ignored.
The user can click "send" in order to perform the query.
Same works for single individuals: a company can perform a query clicking in the panel 'Monitoring' accessible via the navbar.

2.7 Race Management

- **Implemented** requirements
  - **[R8$_M$]**: Logged-in users can register to a run before the start time
  - **[R9$_M$]**: Logged-in users can see a run information
  - **[R13$_M$]**: Logged-in run organizers can organize a run
  - **[R14$_M$]**: Organizers of a run can define the path and additional information for that run

- **Non-implemented** requirements

- **[R15$_M$]**: Organizers of a run can start the run

Database
Information about a run are stored in the table "run". The checkpoints of the run are saved in the table "run_check_point". The subscriptions of each individual to runs are stored in the table "run_subscription";

Back-end
The `/runs` endpoint gives the client a way to exploit the requirements specified. More specifically:

- **[R8$_W$]**: `/runs/join` allows the user to join a run, if exists
- **[R9$_W$]**: `/runs/positions` allows the user to gather informations on the position of the runners in a given run, if exists
- **[R13$_W$] [R14$_W$]**: `/runs/` allows a run organizer to create a run, specifying its path, if it is well formed

Front-end - Application
The user can register to a nearby run by using the Track4Run functionality reachable from the Dashboard. Here the user can also see the run information. Run Organizers can create a new run from their control panel, clicking the plus icon and defining checkpoints. The run start automatically at the defined time.

Comment on not implemented requirements
We simply assumed that the run automatically starts at the specified time when it was creates, at this stage of the project.

2.8 Users Spectating Race

- **Implemented** requirements
  - **[R10$_M$]**: Logged-in users can see participant position on a map, if the run has started
  - **[R13$_C$]**: Can send the position of athletes during a run to spectator's devices.
- **Non-implemented** requirements
  - **[R12$_C$]**: Can compute each athlete's rank in a run and send it to each user device.

Database
The positions of the users are retrieved exploiting Data4Help functionality, so the data is in the "gps_coordinates" table.

Back-end
The server requests the runners position from the database for the given run, if exists.

Front-end - Application
The users can see the position of participants on a map by clicking on View on the Track4Run section in the dashboard.

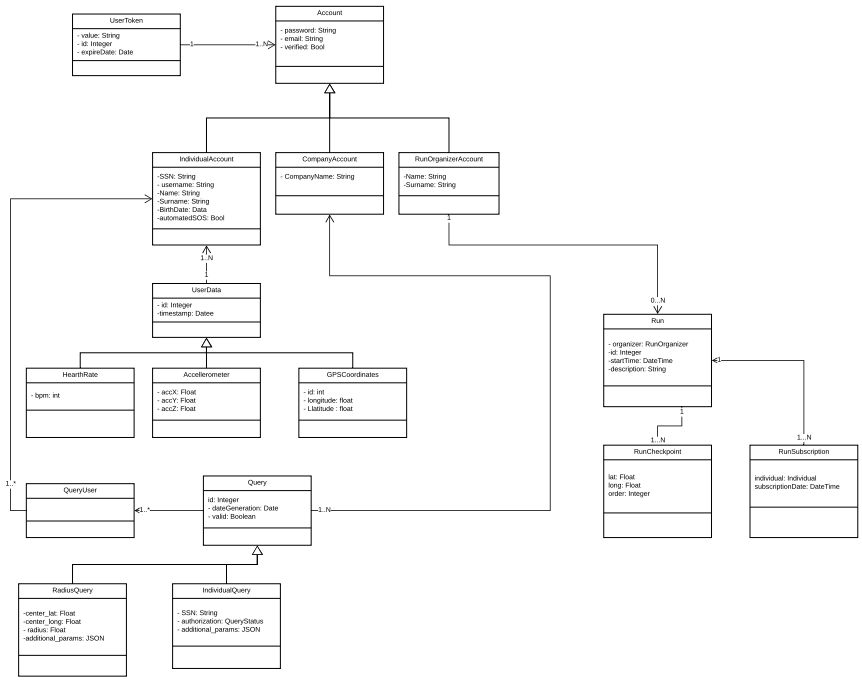
Comment on not implemented requirements
At this stage of the project, implementing the requirement was too costly.

3 Design Choices

3.1 Data Layer

In order to manage data for our system, according to DD document, we needed a relational database. We could choose any SQL database. We opted for `PostgreSQL` because of its:

- stability;
- high scalability;
- being opensource;
- easy to set up and use;
- availability of libraries for many programming languages;
- possibility to obtain free instance on Heroku;



3.2 Business Logic Layer

3.2.1 External services

An external email service is used to send a verification email to each actor in order to verify the account. Our system is email-provider independent,

therefore it just needs the email, the password to be set via environmental variables. See the insallation section for the details.

3.2.2 Platform & Used libraries

Nodejs
Nodejs is a Javascript runtime. This platform has been chosen because it is OS indipended, it offers many stable and well known libraries (ie `express` for the backend), it is well supported by the majority of the *PAAS* and *Heroku*, in our case.
Additionally its asynchronous handling of *IO* has been proven to be fundamental for applications expecting high throughput.

Libraries
Express
Express is a *Free and Open Source* web application framework that provides a robust set of features for building *API*. Its functional approach in the use of middlewares and in routing (i.e. adding a new middleware to an *endpoint* is done via adding one or multiple functions in the declaration of the endpoint) was proven to be critical to enhance the scalability of the code. Moreover the use of functions as middlewares provides great modularity to the application (i.e. adding and enpoint is done via adding a function in response to that route).

Node Postgres
Node Postgres is a collection of modules for interacting with the `PostgreSQL` database. Due to the limitation imposed by *Heroku PostgreSQL* (i.e. maximum of 20 active connections) it has been opted for this library as it provides an excellent pooling system, allowing the limiting of the maximum connection per pool.
Additionally it provides database *Parametrized queries*, where the query string is passed directly to the database and parameters are substituted there instead of perform a string concatenation on the server, that could lead to an *SQL Injection*.

Security and Authentication
BCrypt
BCrypt is the *Javascript* implementation of the bcrypt hashing function based. I provides a simple and criptographically secure way to hash password and compare them during authentication.

JSONWebToken
jsonwebtoken is the *javascript* implementation of the JSON Web Token Web standard. It provides a secure (*URL Safe*) way to store user informations that need to be transferred between two parts.

Testing
Jest
Jest is an all in one 'zero configuration' testing platform. It provides simple methods to seamlessly test synchronous and asynchronous functions (which was mandatory for us) and either:

- Make an assertion on their output values
- Make an assertion on the resolving or rejection of an asynchronous function
- Make an assertion on the throwing of exception

Additionally, testing those three aspects is done similarly between synchronous and asynchronous functions, thus reducing the upfront setup code for the test.

3.3 Presentation Layer/Client

3.3.1 Website

HTML
For what concerning the company management, we decided to develop a website.
It was developed using html as hypertext markup language.
Advantages of using Html are:

- easy to create web pages
- every browser supports HTML language
- it is a tag based language
- is compressible text so is easy to download

Features of Html5 HTML5 is the latest version of HTML. Below are some HTML5 features used for the system:

- **Sections**, used to organize webpage content into thematic groups

```
<div class="section">
          <div class="row center">
            <h5 class="header col s12 ">Data4Help let companies query on their d

            <h5 class="header col s12 light">Add new query by clicking the follo
            ...
```

- **Nav**, tag used for the navbar of the page

```
<div class="navbar-fixed">
    <nav>
      <div class="nav-wrapper">
        <a href="#!" class="brand-logo">Data4Help Website</a>
        <ul class="right hide-on-med-and-down">
          <li><a href="dashboard.html">Dashboard</a></li>
        ...
      </div>
    </nav>
  </div>
```

- **Footer**, tag used for footer and brief description of the page

```
<footer class="page-footer orange">
    <div class="container">
      <div class="row">
        <div class="col l6 s12">
          <h5 class="white-text">TrackMe</h5>
          <p class="grey-text text-lighten-4">TrackMe is a company that wants to
          ...
```

- **Header**, tag used for important text in the page

```
<div class="row center">
            <h5 class="header col s12 ">Data4Help let companies query on their d

            <h5 class="header col s12 light">Add new query by clicking the follo
```

Javascript For the website, we decided to use Javascript as scripting programming language. It has the following advantages.

- Web-oriented interface
- compatible with a wide number of devices because does not rely on a particular OS
- is the only programming language to do web applications
- compatible with any browser

Some of the components of javascript used are below:

- **Modals**, used to signup/signin forms and for queries forms.

```
<!-- Modal Trigger for Successful Query add-->
<a class="modal-trigger" href="#modal1"></a>

<!-- Modal Structure -->
<div id="modal1" class="modal">
  <div class="modal-content " id="modal_success">
    <h4>Success</h4>
    <p>Query has been created successfully. Json file will be downloadable from
      list.</p>
 ...
```

- **Parallax**, is an effect where the background content or image in this case, is moved at a different speed than the foreground content while scrolling. Used in the index page.

```
\begin{verbatim}
<div class="parallax"><img src="images/parallax1.jpg"></div>
  </div>

  <div class="section no-pad-bot" id="index-banner">
    <div class="container">
  ...

    </div>
  </div>
```

3.3.2 External libraries

The following external libraries have been used in order to add some functionality to the app.

- JQuery:
- MaterializeJs

Materialize As a framework for the html page, we decided to develop the website using Materialize, since it provides for very good graphical effects without particular effort. Materialize is a modern responsive CSS framework based on Material Design by Google.
Material Design is a design language that combines the classic principles of successful design along with innovation and technology.

Some of the components used

JQuery We decided to use jQuery to start requests on the back-end using the endpoints provided by the server. jQuery is a JavaScript library designed to simplify HTML DOM. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library.

3.3.3 Mobile App

iOS / Android For the presentation layer, for what concerning the individual management and the run organizer part, we decided to build a native mobile app for the two main operating systems on the market, iOS and Android, using Flutter. In this way we can cover about the 100% of the market. [https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/] Advantages of using Flutter are:

- ability to share the same codebase between Android and iOS;
- ability to perform a fast development;
- the output is an app with native performance both on Android and iOS.

On the other hand, the Flutter project exited recently from the beta stage, and is in active development, but it's enough stable to be used in a production environment.

Dart Application built on Flutter needs to be written in Dart, which is a programming language designed by Google to let developers create high-quality, mission-critical apps for iOS, Android, and the web. It is an Object oriented programming language, which ereditates a lot from others programming language like Java, C# and C++ in order to be very easy to learn. It also support many new programming style, like the reactive programming, and support is strongly typed.

It compiles directly to native code (in case of Flutter) so that applications can run natively on devices, allowing a better management of resources and optimization of the subproducts of the compilation. There are also a ton of different libraries that allow interoperability between Dart application and pre-existent software.

It is also free and open source.

External libraries The following external libraries have been used in order to add some functionality to the app. All the libraries are open source and publicly available from the Dart repository.

- **google_maps_flutter**: allows to show a Google Map component in the app, which offers also the ability to place markers on it;

- **charts_flutter**: allow to show chart of any type while being consistent with Material Design.

External libraries The following internal libraries have been used:

- **http**: to make http/https queries to the backend,
- **material_flutter**: offers the user interfaces main blocks, the widgets, in Material Design. It also manages all the life cycle of the app and the navigation in it.

The MVP pattern The Flutter application follows the MVP pattern that does not only define the actors of the app but also describe the way of how they communicate together. It consists of three components:

- **Model**: it contains the logic that retrives the data that needs to be show to the user. It basically makes requests to the backend and decode the responses.
- **View**: it mains purpose is to show the data of the model to the user, formatted in an elegant way. It receives the user input and passes it to the Presenter.
- **Presenter**: it contains the business logic and manages the communications between the Model and the View, while keeping them completely separated.

Note that this is a *thin client*, i.e. all the validation of the data inserted by the user is done mainly on the server. On the client side is done only the needed preprocessing for interoperability.

4 Source code structure

4.1 Flutter project structure

The following section's aim is to explain the structure of the Flutter project, that is the project for the Android/iOS frontend, managing the Individuals and Run Organizers operations. This section only explains the content of every file but does not deep into details; for more information, please look at the source code.

This section will cover only the content `lib` folder, as the `android` and `ios` folder contain platform specific files, automatically generated by the Flutter project creator.

As stated before, the project is based on the MVP pattern and so the source code is organized in three folder, `model`, `view` and `presenter`. In this way it's more easy to maintain the code keeping the communication with the backend completely separated from the logic of the View.

Model The main aim of the model is to "hide" the retrieving of the new data from the server. It contains all the logic to interact with the server. In particular:

- `UserModel.dart`: this is the actual model of the User. It contains all the necessary code to make HTTP request to the server to retrieve data and keeps track of the authentication code.
- `RunOrganizerModel.dart`: is the model of the Run Organizer. It contains the necessary code to retrieve the Run Organizer's details from the server and manage the auth code.
- `UserPersonalData.dart`: model for the user personal information. Contains the personal data of a user, like the name and the surname.
- `UserData.dart`: model for the user generated data. It contains the gps coordinates, the accelerometer information and data about heartRate. It also take care of decoding from and ancoding to the JSON format, in order to communicate with the server.
- `RunPoint.dart`: model for a single checkPoint in a Run.
- `Run.dart`: model for a Run. Contains information about the date and time of start and end, and the organizer.
- `PendingQueryRequest.dart`: model for individual requests of information. It contains the query id and the name of the Company.

Presenter The aim of the presenter is to take care of the interaction of the user with the View and react to them. It forwards all the request to the model, after doing some checks.

- `UserPresenter.dart`: the presenter for the User model; it verifies the user input and if all is ok forwards the request to the model.
- `RunOrganizerPresenter.dart`: the presenter for the Run Organizer mdoel; it verifies the user input and if all is OK forwards the request to the model.

View These are all the files under the View folder. Each files contains a specific screen of the application (or part of it). There are two main subfolder, one for each main part of te app.

For what concerning *data4help*:

- `CheckSmartwatchConnection.dart`: simulates the verification of the connection of a correct Smartwatch. If found sends the user to the login screen.
- `Data4HelpLogin.dart`: allows the user to insert its email and password and login, or to go to registration page if it does not have an account.
- `Data4HelpRegister.dart`: allows an unregistered user to register a new account.
- `Dashboard.dart`: multipage activity that allows the user to view and manage its infromation.
- `dashboard/DashboardMainPage.dart.dart`: shows to the user a recap on its daily activities.
- `dashboard/DashboardDetailPage.dart`: shows to the user a detailed page containing its daily activity. Allow the user to load a specific day.
- `dashboard/DashboardPendingQueriesRequests.dart`: shows to the user the pending requests for individual data by a Company. The user has the ability to accept or deny them.
- `dashboard/DashboardRunRegistrationPage.dart`: shows to the user the nearby runs and allow the Individual to subscribe to them if they are not started or to watch them of they are ongoing.
- `dashboard/DashboardTestPage.dart`: allows the user to send test data to the backend, simulating a connected smartwatch.
- `dashboard/WatchRun.dart`: shows the positions of the runners of a specific run on a map.

For what concerning *track4run*:

- `Track4RunLogin.dart`: allows the run organizer to insert its email and password and login, or to register if it does not have an account.
- `Track4RunRegister.dart`: allows an unregistered run organizer to register a new account.
- `DashboardRunOrganizer.dart`: shows to the run organizer all the run organized by him.
- `DashboardRunOrganizer.dart`: allows the run organizer tod efine a new run by specifying the details and the checkpoints.
- `DashboardRunOrganizer.dart`: allow he run organizer to add new checkpoint to the run.

Others files

- `Config.dart`: contains the global static configuration of the application, like the hostname of the backend.
- `Main.dart`: the main entry point of the app. It loads the first screen shown to the user.

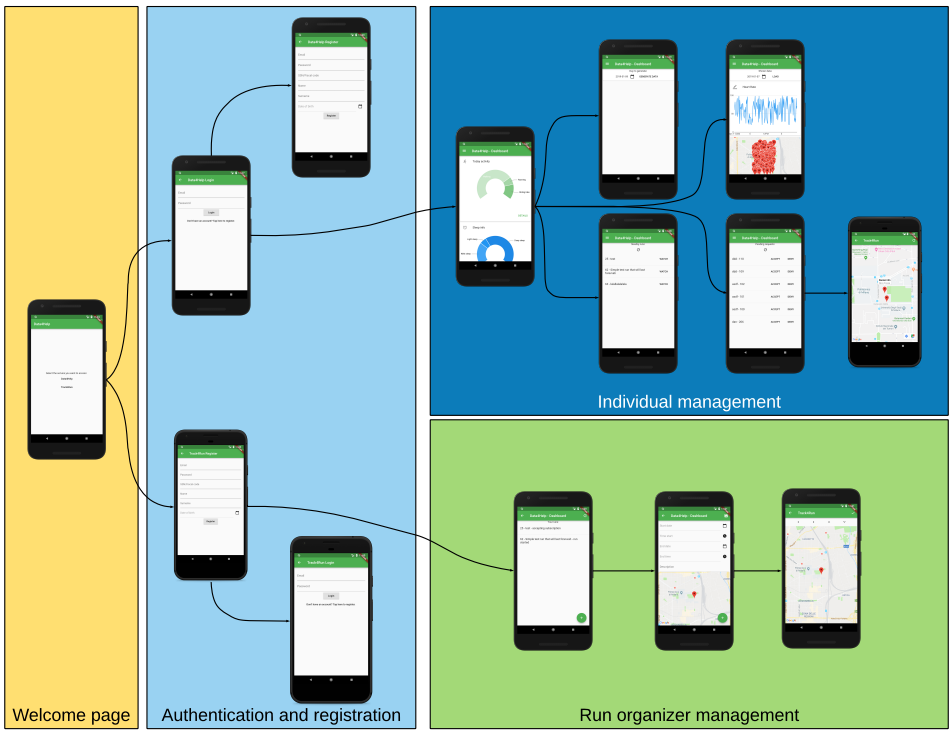


Figure 1: Storyboard structure of the Mobile Application

App structure The figure above shows the UI flow of the app. It is dived in four parts:

- **Welcome page**: allows the user to access either the management part of the Individual or of the Runs;
- **Authentication and registration**: allows the user to login and register (the upper part to the individual part, the lower part to run organizer part);
- **Individual management**: allow the user to access its information, send and retrieve its data, join a run and view position of the runners in an ongoing run.
- **Run organizer management**: allows the run organizer to access information about its organized run and create new runs.

4.2 Website structure

This section aims explaining the structure of the web page into the website directory.

The website directory contains 3 sub-folders

- CSS: contains the css configuration standardized by Materialized
- js: it contains only the initial file of the Materialized scripts. All other scripts are inside the html files.
- images: The images used for the graphic design. Contains the picture for parallax and the icons for data collection of queries performed.

Moreover, the website directory contains the html files describing the pages of the website. The following lists only explain the content of the html files and what do they provide to the users.
In order to see detailed information about the code, please look into the files themselves.

4.2.1 Html files

- *index.html* this is the Homepage of the website where user is re-directed if not logged-in. It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible the buttons for signin and signup.
- *services.html* this is the page that describes the services of Data4Help accessible through the website. It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible the buttons for signin and signup.
- verify-email.html this is the page where a user is redirected after having successfully signed-up. It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible the buttons for signin and signup.
- registration_success.html this is the page where a user is re-directed after having verified his email. It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible the buttons for signin and signup.
- *dashboard.html* This is the dashboard of the website. A user who still has an session (i.e. having a valid auth_token in the cookies) is re-directed in this page. The page shows the last 5 queries performed on a group of individuals by the company (if any). It contains a navbar where are shown the pages accessible for all logged users. Through the navbar are accessible a button for signout.

- *monitoring.html* this is the core page of the individual queries. It lists all the queries of individuals performed by the company (if any) and let it download the content in any time by clicking to the button "Download query". It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible a button for signout.

- *query.html* this is the core page of the queries for group of individuals. It lists all the queries for group of individuals performed by the company (if any) and let it download the content in any time by clicking to the button "Download query". It contains a navbar where are shown the pages accessible for all not-logged users and a brief description of the data4Help services. Through the navbar are accessible a button for signout.

- *subscription.html* This is the page where a company can subscribe to a payment plan. However, this implementation it was not implemented and only displays the 3 possible subscription plan that respects the "Pricing Policies" described in the RASD document. It contains a navbar where are shown the pages accessible for all logged users. Through the navbar are accessible a button for signout.

4.2.2 Javascript files

- *materialize.js* contains default initialization provided by Materialize

- *cookie-management.js* contains scripts that add/eliminate/update cookies of the user.

- *query-list-retrieval.js* contains the scripts that retrieves the list of queries performed by the logged user.

- *query-post.js* contains the scripts that permit users to perform a query for individuals and a query for radius search of group of individuals.

- *signin-signout-modal.js* contains the scripts that manages the modals for signin and signup.

- *signout-modal.js* contains the scripts that manages the modals for signout.

- *switch-management.js* contains the scripts managing the switch component for keeping company subscribed to a query (even if unsubscription was not implemented).

- *xml-download-management.js* contains the scripts that manage the download of a query content returned by the server.

Figure 2: Storyboard structure of the Website

Website structure The figure above shows the UI flow of the website. It is dived in four parts:

- **Home page**: allows the user to either login or sign up. If correctly signed up, shows a successful registration page;
- **Login and Registration form**: the forms that user compiles in order to register.
- **Dashboard**: the page where the user is redirected after correct login;
- **Query**: allows the company to perform a new query on a group of individual and to download the xml of the already performed queries.
- **Monitoring**: allows the company to perform a new query on individuals and to download the XML content of the already performed queries.
- **Query form**: the forms that user compiles in order to perform queries.

4.3 Backend project structure

This section aims at explaining the structure of the backend inside the `backend` directory. Note that this section only briefly describes the directories and the files in it, for more detailed explainations it is advisable to look at the comments in the source code.

The root directory contains the following files:

- `app.js` The entry point of the app. It defines how to handle different endpoints, how to catch errors and starts the server.
- `dbdump` Current dump of the database.

- `start.sh` Start script.
- `jest.config.js` The configuration file for `jest`, the testing framework.
- `package.json` Lists all the dependencies needed for the program to run and various other informations on the application.
- `package-lock.json` Contains the state of the dependency tree of the application.
- `Procfile` Contains the command to be run by Heroku to start the application.

And the following directories:

- `managers`: Contains the managers of the application.
- `routes`: Contains the routes definition for the application.
- `__tests__`: Contains the tests for the application.
- `utils`: Contains some utilites function used during testing and debugging
- `stub_endpoint`: Contains the stub of endpoint responses used during the initial phase of building the application. All the responses contain a success response and various error responses.

4.3.1 managers

The managers folder contains a single file:

- `config.js`: Allows to configure some parameters of the app, such as the minimum number of users for which allow a radius query, the database to connect to and the maximum amount of client allowed.

And the following folders with their respective files

authentication

- `AuthenticationManager.js`: Handles login and registration of actors.
- `requiredParams.json`: Required parameters for login and registration of each actor.

individual

- `IndividualsManager.js`: Handles post and retreival of individuals' data.

- `requiredParams.json`: Required parameters for the data posted.

query

- `QueriesManager.js`: Handles post, retreival and performing of the companies' queries.
- `requiredParams.json`: Required parameters for the queries.
- `templateQueries.json`: Template queries for insertion in the database

runs

- `RunManager.js`: Handles creation, retrieval of a run, joining a run and monitoring participants in a run.
- `runStatus`: Exports the constans for the status of a run.

subs

- `SubscriptionManager.js`: Handles the subscription. Not implemented.

token

- `TokenManager.js`: Handles operations concerning the token, such as the retreival of the actor or the check of the presence of the user in a database.

4.3.2 routes

The routes folder contains a single file:

- `router.js`: Joins all the defined endpoints end exports the as a single endpoint.

And the folder `endpoints`, which contains the following files:

- `auth.js`: Exports the endpoints for authentication. Handles login and registration of actors via the use of the functions declared in the `AuthenticationManager.js` file.
- `indiv.js`: Exports the endpoints for the individual. Handles post and retrival of individuals' data via the functions declared in the `IndividualsManagers.js` file.
- `queries.js`: Exports the endpoints for the queries. Handles post and retrival of company queries via the functions declared in the `QueriesManager.js` file.

- `runs.js`: Exports the endpoints for the runs management. Handles post and retrival of run organizers run, subscription of individuals to a run, monitoring of runners in a run via the functions declared in the `RunsManager.js` file.
- `subs.js`: Exports the endpoints for subscription to query plans. Not implemented.

4.3.3 `__tests__`

The routes folder contains a single file:

- `config.js`: Internal configuration for the tests. Exports the token for company, run organizer, individual their mail and their password. Additionally it exports the URL on which perform the tests.

And the following directories with their respective files:
auth

- `featureTest.jest.js`: Feature tests for the /auth endpoint
- `unitTest.jest.js`: Unit tests for the helper functions of `AuthenticationManager.js`

indiv

- `featureTest.jest.js`: Feature tests for the /indiv endpoint
- `unitTest.jest.js`: Unit tests for the helper functions of `IndividualsManager.js`

query

- `featureTest.jest.js`: Feature tests for the /queries endpoint
- `unitTest.jest.js`: Unit tests for the helper functions of `QueriesManager.js`

runs

- `featureTest.jest.js`: Feature tests for the /runs endpoint
- `unitTest.jest.js`: Unit tests for the helper functions of `RunsManager.js`

4.3.4 stub_endpoint

Contains the following folders:
auth

- `login.json`: Mock response on enpoint `/auth/login`.
- `register_company.json`: Mock response on enpoint `/auth/register_company`.
- `register_run_organizer.json`: Mock response on enpoint `/auth/register_run_organizer`.

- `register_user.json`: Mock response on enpoint `/auth/register_user`.
- `verify.json`: Mock response on enpoint `/auth/verify`.

indiv

- `data_POST.json`: Mock response for the POST request on endpoint `/indiv/data`.
- `data_GET.json`: Mock response for the GET request on endpoint `/indiv/data`.

query

- `query_POST.json`: Mock response for the POST request on endpoint `/queries/query`.
- `query_GET.json`: Mock response for the GET request on endpoint `/queries/query`.

runs

- `join.json`: Mock response for the POST request on endpoint `/runs/join`.
- `positions.json`: Mock response for the GET request on endpoint `/runs/positions`.
- `root.json`: Mock response for the POST request on endpoint `/runs/`.
- `run.json`: Mock response for the GET request on endpoint `/runs/run`.

subs

- `plan_GET.json`: Mock response for the POST request on endpoint `/subs/plan`. Still active.
- `plan_POST.json`: Mock response for the POST request on endpoint `/runs/positions`.

4.3.5 utils

Contains a single file:

- `testUtils.js` Utilities function used during testing.

4.4 Rest API

In the following paragraph are represented the Rest API. The API path has been modified putting as prefix `/v1`. In this way, we can provide a kind of versioning of the API interface so that future releases won't interfere with previous ones.

The following API are also present in the DD, this paragraph aims to describe in much details the request data and of the response.

Authentication Manager

- User Registration

Endpoint	/v1/auth/register_user
URL Params	
Method	**POST**
Request Data	mail: String password: String SSN: String name: String surname: String birthday: Date smartwatch: String

Success Response	code: `200 OK`
	content:
	`{`
	`"success": true,`
	`"auth_code": ...`
	`}`

Error Response	code: `422 UNPROCESSABLE ENTITY`
	content:
	`{`
	`success: false,`
	`error: 'InfoNotValid',`
	`message: ...`
	`}`
	
	`message` can be one of the following:
	• `Unsupported smartwatch`
	• `Mail already used`
	• `SSN not valid`

Uses	Allows the client to register a new User

Request example	POST /v1/auth/register_user
	content:
	`mail: example@gmail.com`
	`password: password`
	`SSN: 0123456789012345`
	`name: Nico`
	`surname: Fossa`
	`birthday: 1996-01-01`
	`smartwatch: SonySmartwatch3`

Response example	`{`
	`"success":true,`
	`"auth_token":"eyJhbGciOiJIUz..."`
	`}`

- Company Registration

Endpoint	/v1/auth/register_company
URL Params	
Method	**POST**
Request Data	email: String password: String company_name: String

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`auth_code: ...`
	`}`

Error Response	code: `422 UNPROCESSABLE ENTITY` content:

	{
	success: false,
	error: 'InfoNotValid',
	message: ...
	}
	
	message can be one of the following:
	• Email already in use

Uses	Allows the client to register a new Company

Request example	POST /v1/auth/register_company
	content:
	{
	email: "Htower@Htower.com",
	"password": "2Cid34242j",
	"company_name": "H-Tower",
	"type": 'company'
	}

Response example	{
	"success":true,
	"auth_code":"eyJhbGciOiJIUz..."
	}

- Run Organizer Registration

Endpoint	/v1/auth/register_run_organizer
Method	**POST**
URL Params	
Request Data	email: String password: String name: String surname: String

Success Response	code: 200 OK
	content:
	{
	success: true,
	auth_code: ...
	}

Error Response	code: 422 UNPROCESSABLE ENTITY content:

	{
	success: false,
	error: 'InfoNotValid',
	message: ...
	}
	
	message can be one of the following:
	• SSN not valid

Uses	Allows the client to register a new run organizer
Request example	POST /v1/auth/register_run_organizer content: mail: example.runorganizer@gmail.com password: password name: Nico surname: Fossa birthday: 1996-01-01

Response example	{
	"success":true,
	"auth_token":"eyJhbGciOiJIUz..."
	}

- User Login

Endpoint	/v1/auth/login
Method	**POST**
URL Params	
Request Data	email: String password: String type: String

Success Response	code: 200 OK
	content:
	{
	success: true,
	auth_token: ...
	}

Error Response	code: 404 NOT FOUND content:

	{
	success: false,
	error: 'UserNotFound' ,
	message: 'User does not exists'
	}
	
	--- 	
	
	code: 403 FORBIDDEN
	content:
	{
	success: false,
	error: 'InvalidCredentials' ,
	message: 'Invalid Credentials'
	}

Uses	Allows the client to login
Request example	POST /v1/auth/login content: email: example@gmail.com password: password type: individual

Response example	{
	"success":true,
	"auth_token":"eyJhbGciOiJIUz..."
	}

- Verify mail

Endpoint	/v1/auth/verify
Method	**GET**
URL Params	mail: String code: String type: String
Request Data	

Success Response	code: 200 OK
	content:
	{
	success: true,
	message: email verified
	}

Error Response	code: `401 UNAUTHORIZED`
	content:
	`{`
	`success: false,`
	`error: 'InvalidCode',`
	`message: 'Code is invalid'`
	`}`

Uses	Allows verification of the account

Request example	POST /v1/auth/verify
	content:
	`?mail=example@gmail.com&code=`
	`eyJhbGciOiJ...&type=individual`

Response example	`{`
	`success: true,`
	`message: email verified`
	`}`

Individuals Manager

- Data store

Endpoint	/v1/indiv/data
Method	**POST**
URL Params	
Request Data	auth_token: String data: JSON < Array < JSON > >

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`message: 'Sync successful'`
	`}`

Error Response	code: `422 UNPROCESSABLE ENTITY` content:

	{
	success: false,
	error: 'InvalidData',
	message: 'Data are invalid'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}

Uses	Allows the client to store sensor data in the database
Request example	POST /v1/indiv/data content:

	<pre>{
	"gps_coordinates":[
	{
	"lat":45.473641,
	"long":9.228704,
	"timestamp":"2018-01-01
	T00:00:00Z"
	}
	],
	"accelerometer":[
	{
	"acc_x":0.147145,
	"acc_y":0.016155,
	"acc_z":0.917696,
	"timestamp":"2018-01-01
	T00:00:00Z"
	}
	],
	"heart_rate":[
	{
	"bpm":82,
	"timestamp":"2018-01-01
	T00:00:00Z"
	}
	]
	}</pre>

Response example	<pre>{
	success: true,
	message: 'Sync successful'
	}</pre>

- Data retrival

Endpoint	/v1/indiv/data
Method	**GET**
URL Params	auth_token: String begin_date: Date end_date: Date
Request Data	
Success Response	code: `200 OK` content:

	{
	success: true,
	data: SensorsData
	}

Error Response	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}

Uses	Allows the client to retrive sensor data from the database
Request example	GET /v1/indiv/data content: ?auth_token=eyJhbGciOiJ...&begin_date= 2018-01-01&end_date=2018-01-01

Response example	<pre>{
	"success":true,
	"data":{
	"gps_coordinates":[
	{
	"lat":45.473641,
	"long":9.228704,
	"timestamp":"2018-01-01T00:00:00Z"
	}
	],
	"accelerometer":[
	{
	"acc_x":0.147145,
	"acc_y":0.016155,
	"acc_z":0.917696,
	"timestamp":"2018-01-01T00:00:00Z"
	}
	],
	"heart_rate":[
	{
	"bpm":82,
	"timestamp":"2018-01-01T00:00:00Z"
	}
	]
	}
	}</pre>

- User information retrival

Endpoint	/v1/indiv/user
Method	**GET**
URL Params	auth_token: String
Request Data	

Success Response	code: `200 OK`
	content:
	<pre>{
	success: true,
	user: ...userInfo
	}</pre>

Error Response	code: `400 BAD REQUEST` content:

	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}

	code: 404 Not Found
	content:
	{
	success: false,
	error: 'Not found',
	message: 'The user wasn't found'
	}

Uses	Allows the client to retrive user informations
Request example	GET /v1/indiv/user content: auth_token: eyJhbGciOiJ...

Response example	{
	"success":true,
	"user":{
	"email":"example@gmail.com",
	"verified":true,
	"ssn":"1234567890123456",
	"birth_date":"2018-12-27
	T00:00:00.000Z",
	"automated_sos":false,
	"smartwatch":"TestSmartwatch1",
	"name":"Nico",
	"surname":"Fossa"
	}
	}

Query Manager

- Query creation

Endpoint	/v1/queries/query
URL Params	
Request Data	auth_token: String query: json

Success Response	code: 200 OK
	content:
	{
	success: true,
	message: 'Query successfully posted'
	}

Error Response	code: 422 UNPROCESSABLE ENTITY
	content:
	{
	success: false,
	error: 'QueryTooRestrictive',
	message: 'Query on too few users'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'BadQuery',
	message: 'Query is invalid'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}
	
	--- 	
	
	code: 402 PAYMENT REQUIRED
	content:

	{
	success: false,
	error: 'PaymentRequired',
	message: 'Payment required'
	}

Uses	Allows the client to create a query

Request example	POST /v1/queries/query
	content:
	{
	"auth_token": "eyJhbGciOiJIUz...",
	"query": {
	"type": 'radius',
	"center_lat": 45.4398241,
	"center_long": 39.2489133,
	"radius": 10,
	"additional_params": {
	"accelerome"r: {
	"acc_x": [-10, 15]
	},
	"heart_rate": {
	"bpm": [120, 160]
	}
	}
	}
	}

Response example	{
	"success": true,
	"message": "Query successfully posted"
	}

- Query Retrival

Endpoint	/v1/queries/query
Method	**GET**
URL Params	auth_token: String
Request Data	
Success Response	code: 200 OK content:

	{
	success: true,
	queries: ...totalQueries
	}

Error Response	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}

Uses	Allows the client to retrive the queries created by a company
Request example	GET /v1/queries/query content: ?auth_token=sjHeGbeuUwj...\&query_id=13

Response example	{
	"success":true,
	"queries": {
	"individual":[{
	"id":249,
	"ssn":
	"1234567890123456",
	"auth":true,
	"additional_params":{
	"accelerometer":{
	"acc_x":[-19,2000]
	},
	"heart_rate":{
	"bpm":[10,200]
	}
	}
	"subscribed":false
	}]
	}

- Perform Query

Endpoint	/v1/queries/query/data
method	**GET**

URL Params	auth_token: String query_id: Integer
Request Data	

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`data: ...`
	`}`

Error Response	code: `400 BAD REQUEST`
	content:
	`{`
	`success: false,`
	`error: 'InvalidToken',`
	`message: 'Token is invalid'`
	`}`

Uses	Allows the client to perform a query
Request example	GET /v1/queries/query/data content: `?auth_token=eKdgNeIegO...&query_id=23`

Response example	`{`
	`success: true,`
	`data: {`
	`success: true,`
	`data: "accelerometer":[{"timestamp":"2019-01-07T00:0`
	`}`
	`}`

- (still) Unauthorized queries retrieval

Endpoint	/v1/queries/query/individual/pending
Method	**GET**
URL Params	auth_token: String
Request Data	
Success Response	code: `200 OK` content:

	{
	success: true,
	queries: ...queries
	}

Error Response	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}

Uses	Get unapproved individual querie
Request example	GET /v1/queries/individual/pending content: auth_token: eyJhbGciOiJ...

Response example	{
	"success":true,
	"queries":[
	{
	"id":248,
	"company_name":"Company1"
	},
	{
	"id":253,
	"company_name":"Company2"
	},
	{
	"id":250,
	"company_name":"Company3"
	}
	]
	}

- Allow/Negate individual query

Endpoint	/v1/queries/query/individual/pending
Method	**POST**
URL Params	
Request Data	auth_token: String query_id: Integer

	decision: Boolean

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`message: 'Response Saved'`
	`}`

Error Response	code: `400 BAD REQUEST`
	content:
	`{`
	`success: false,`
	`error: 'InvalidToken',`
	`message: 'Token is invalid'`
	`}`

Uses	Specify decision for individual query

Request example	POST /v1/queries/individual/pending
	content:
	`auth_token: eyJhbGciOiJ...`
	`query_id: 21`
	`decision: true`

Response example	`{`
	`"success":true,`
	`"message":"Response saved"`
	`}`

Subscription Manager

- Buy subscription plan

Endpoint	/v1/subs/plan
Method	**POST**
URL Params	
Request Data	auth_token: String mail: String plan: String
Success Response	code: `200 OK` content:

	{
	success: true,
	message: {$PLAN} bought
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}
	
	--- 	
	
	code: 402 PAYMENT REQUIRED
	content:
	{
	success: false,
	error: 'PaymentRequired',
	message: 'Payment required'
	}
	
	--- 	
	
	code: 404 NOT FOUND
	content:
	{
	success: false,
	error: 'PlanNotFound',
	message: 'Plan not found'
	}

Uses	Allows the client buy a subscription to a new plan

- Get plan informations

Endpoint	/v1/subs/plan
Method	**GET**
URL Params	plan: String
Request Data	

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`plan: ...planDetails`
	`}`

Error Response	code: `400 BAD REQUEST`
	content:
	`{`
	`success: false,`
	`error: 'PlanUnavailable',`
	`message: 'Plan is not available'`
	`}`
	
	code: `404 NOT FOUND`
	content:
	`{`
	`success: false,`
	`error: 'PlanNotFound',`
	`message: 'Plan not found'`
	`}`

Uses	Allows the client retrive informations about a subscription plan

Run Manager

- Create a Run

Endpoint	/v1/runs/run
Method	**POST**
URL Params	
Request Data	auth_token: String time_begin: Date time_end: Date description: String coordinates: Array < JSON >
Success Response	code: `200 OK` content:

	{
	success: true,
	run_id: $ID
	}

Error Response	code: 422 UNPROCESSABLE ENTITY
	content:
	{
	success: false,
	error: 'InfoNotValid',
	message: ...
	}
	
	message can be one of the following:
	• Time not valid
	• Coordinates not valid
	
	--- 	
	
	code: 403 FORBIDDEN
	content:
	{
	success: false,
	error: 'InvalidCredentials',
	message: 'Invalid Credentials'
	}

Uses	Allows the client to create a new run
Request example	POST /v1/runs/run content:

	<pre>{
	"auth_token":"eyJh...",
	"time_begin":"2019-01-19
	T09:00:00.000",
	"time_end":"2019-01-19
	T09:00:00.000",
	"description":"A simple run",
	"coordinates":[
	{
	"lat":45.476987,
	"long":9.234592,
	"description":"0 - 0"
	},
	{
	"lat":45.476987,
	"long":9.234192,
	"description":"1 - 1"
	}
	]
	}</pre>

Response example	<pre>{
	"success":true,
	"run_id":77
	}</pre>

- List all available runs

Endpoint	/v1/runs
Method	**GET**
URL Params	auth_token: String organizer_id? : String
Request Data	

Success Response	code: `200 OK`
	content:
	<pre>{
	success: true,
	runs: ...runList
	}</pre>

Error Response	code: `401 FORBIDDEN` content:

	{
	success: false,
	error: 'InvalidCredentials',
	message: 'Invalid Credentials'
	}

Uses	Allows the client to list all runs satisfying the parameters above
Request example	GET /v1/runs content: auth_token: eyau...

Response example	{
	"success":true,
	"runs":[
	{
	"organizer_id":70,
	"id":62,
	"start_time":"2019-01-05
	T09:25:00.000Z",
	"description":"Simple test run
	that will last
	forevvah",
	"end_time":"2025-01-05
	T10:25:00.000Z",
	"status":"RUN_STARTED"
	},
	{
	"organizer_id":70,
	"id":70,
	"start_time":"2019-01-11T00:03:00.000Z",
	"description":"A simple run for my love Dani &
	"end_time":"2019-01-13T00:03:00.000Z",
	"status":"RUN_STARTED"
	}
	]
	}

- Join a run

Endpoint	/v1/runs/join
URL Params	

Method	POST
Request Data	auth_token: String run_id: String

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`message: 'Joined run $RUN_ID'`
	`}`

Error Response	code: `403 FORBIDDEN`
	content:
	`{`
	`success: false,`
	`error: 'InvalidCredentials',`
	`message: 'Invalid credentials'`
	`}`
	
	code: `404 NOT FOUND`
	content:
	`{`
	`success: false,`
	`error: 'RunNotFound',`
	`message: 'Run not found'`
	`}`
	
	code: `422 UNPROCESSABLE ENTITY`
	content:
	`{`
	`success: false,`
	`error: 'RunError',`
	`message: 'Run doesn't accept participants'`
	`}`

Uses	Allows the client to join a run
Request example	POST /v1/runs/join content:

	auth_token: eyau... run_id: 154

Response example	{
	"success": true,
	"message": "Joined run 154"
	}

- Get the positions of runners in a run

Endpoint	/v1/runs/positions
Method	**GET**
URL Params	run_id: String auth_token: String
Request Data	

Success Response	code: 200 OK
	content:
	{
	success: true,
	position: position
	}

Error Response	code: 404 NOT FOUND
	content:
	{
	success: false,
	error: 'RunNotFound',
	message: 'Run not found'
	}

Uses	Allows the client get the position of a runner in a run
Request example	GET /v1/runs/positions content: run_id: 154 auth_token: eyau...

Response example	{
	"success":true,
	"positions":[
	{
	"user_id":59,
	"id":"Nico Fossa",
	"lastPosition":{
	"lat":45.47334,
	"long":9.228975
	}
	},
	{
	"user_id":62,
	"id":"fras Sgherzi",
	"lastPosition":{
	"lat":45.472069,
	"long":9.231138
	}
	}
	]
	}

5 Testing

5.1 Backend testing

5.1.1 Unit testing

`AuthenticationManager.checkRequiredParams`
Login

Functionality	Input	Assertions
Check required parameters - Individual	A username and a password	Does not throw an exception
Check required parameters - Run Organizer	A username and a password	Does not throw an exception
Check required parameters - Company	A username and a password	Does not throw an exception
Check required parameters - Individual	Missing username or password	Throws an exception
Check required parameters - Run Organizer	Missing username or passord	Throws an exception
Check required parameters - Company	Missing username or password	Throws an exception

Registration

Functionality	Input	Assertions
Check required parameters - Individual	All needed parameters ("email", "password", "SSN", "name", "surname", "birthday", "smartwatch")	Does not throw an exception
Check required parameters - Run Organizer	All needed parameters ("email", "password", "name", "surname")	Does not throw an exception
Check required parameters - Company	All required parameters ("email", "password", "company_name")	Does not throw an exception

Check required parameters - Individual	Some parameters missing	Throws an exception
Check required parameters - Run Organizer	Some parameters missingSome parameters missing	Throws an exception
Check required parameters - Company	Some parameters missing	Throws an exception

`IndividualsManager.toQueryArray`

Functionality	**Request**	**Assertions**
Test behaviour with valid request	Unpack the JSON into an Array suitable for insertion in the database	All the elements of the JSON are present in the array and they are equal to the corresponding value in the input structure.
Bad arguments	Bad arguments as input	Throws an exception.

`IndividualsManager.checkQueryParams`

Functionality	**Request**	**Assertions**
Check query parameters	Valid Individual Query (all parameters are present)	Not throw an exception
Check query parameters	Valid Radius Query (all parameters are present)	Not throw an exception
Check query parameters	Invalid Individual Query (missing parameters)	Throws an exception
Check query parameters	Invalid Radius Query (missing parameters)	Throws an exception

`RunsManager.getRunParamsFromRequest`

Functionality	**Request**	**Assertions**

Extrapolate run parameters from request	Valid request	Not throw an exception and correctly unpack only the needed parameters of the request without modifing their values.
Extrapolate run parameters from request	With bad request body	Not throw an exception and correctly unpack only the needed parameters of the request without modifing their values.

5.1.2 Feature testing

`auth`

Functionality	**Request**	**Assertions**
User registration	Valid registration	Expect the success value of the response to be true and the auth token to be present
Company login	Wrong email or password	Expect the status of the response to be 403 and the message to be Invalid Credentials
Run organizer login	Wrong email or password	Expect the status of the response to be 403 and the message to be Invalid Credentials
Individual login	Wrong email or password	Expect the status of the response to be 403 and the message to be Invalid Credentials
Company login	Correct email and password	Expect the success value of the response to be true and the auth token to be present
Run organizer login	Correct email and password	Expect the success value of the response to be true and the auth token to be present
Individual login	Correct email and password	Expect the success value of the response to be true and the auth token to be present

`indiv`

Functionality	**Request**	**Assertions**
User sends data	User sends data correctly formatted	Expect the success value of the response to be true and the message to be 'Sync successfull'

User sends data	User sends data badly formatted	Expect the request status to be 422 and the message to be 'Invalid data'
User retreives data	User retreives data correcly	Expect the success value of the response to be true and the data to be not `undefined`
User retreives data	User tries to retrive data sending a badly formatted request	Expect the success value of the response to be true and the data to be not `undefined`
User retrieves its personal information	Requests personal information	Expect the success value of the response to be true and data to be not `undefined`

`query`

Retreive company queries	Correctly form request	Expect the success value of the response to be true and queries to be not `undefined`
Post query	Post queries both individual and radius	Expect the success value of the response to be true and message to be 'Query successfully posted' and `query_id` not to be undefined.
Post query	Post queries both individual and radius - Malformed query, missing parameters	Expect the status of the request to be 400 and the message to contain the missing parameter.
Perform a query - individual	Correct parameters	Expect the success value of the response to be true and data and user not to be `undefined`.
Perform a query - individual	Correct parameters but unauthorized by user	Expect the status of the request to be 403 and the message to be 'Unauthorized'.
Perform a query - radius	Correct parameters	Expect the success value of the response to be true and data and user not to be `undefined` and the users' id to be `undefined`.
Pending individual query retrieval	Correct parameters	Expect the success value of the response to be true and queries not to be `undefined`
User allows or denies a pending query	Correct parameters	Expect the success value of the response to be true and message to be 'Response saved'

`runs`

Get the list of runs	Correctly formatted with the individuals' token or the run organizer token	Expect the success property to be true and the runs property not to be `undefined`
Get the position of the runners in a run	Correct parameters	Expect the success property to be true and the positions property not to be `undefined`
Get the position of the runners in a run	non existent run	Expect the status of the response to be 404 and its message to contain 'No run'
Join a run	Correctly formatted request	Expect the success property to be true and the message property to contain 'Joined run'
Join a run	Join an already joined run	Expect the response status to be 500 and the message to be 'You cannot join a run you've already joined'
Join a run	Non existant run	Expect the response status to be 404 and the message to be 'There isnt́ any run available with that id atm'
Create a run	Correctly formatted request	Expect the response success property to be true and the run_id not to be `undefined`
Create a run	Badly formatted request	Expect the response status to be 400 and the message to be 'Missing parameters'

5.2 Flutter project testing

5.2.1 Unit testing

PendingQueryRequest

Test	**Assertions**
Json decoding	Values decoded are exactly the ones present in the json provided

Run

Test	**Assertions**
Json decoding	Values decoded are exactly the ones present in the json provided

UserPersonalData

Test	**Assertions**

Json decoding	Values decoded are exactly the ones present in the json provided

5.2.2 Integration testing

Run Organizer

Functionality	**Request**	**Assertions**
Login	Correct username and password	The call returns a success state (true) and does not threw any exception
Login	Wrong username or password	The call threws an exception
List of runs	Get the list of run	The call does not return null
Run creation	Invalid run creation	The call threws an exception

Individual

Functionality	**Request**	**Assertions**
Login	Correct username and password	The call returns a success state (true) and does not threw any exception
Login	Wrong username or password	The call threws an exception
Register user	Registering an existent user	The call threws an exception
User information	Retriving user informations	The call does return the correct Name and Surname
List of nearby runs	Get the list of run	The call does not return null
Load user data	Get the user data of a valid date	The call does not return null
Push user data	Pushing new user data to the server	The lenght of the data retrieved is not 0
Runners positions	Get the positions of runners in a valid run	The call does not return null
Runners positions	Get the positions of runners in an invalid run	The call threws an exception
Pending queries	Get the list of pending queries	The call does not return null
Respond to query	Respond to an invalid query	The call threws an exception

6 Installation instruction

6.1 Database setup

6.1.1 Requirements

- Internet connection.
- At least 500MB of storage for the database binaries and the database itself to be free on disk.

This section explains how to install and setup `PostgreSQL` and the database image. This section assumes that the OS is *Solus Linux*. For installation instruction for other OSes, please visit the download page on PostgreSQL's website.

Installation
```
sudo eopkg install postgresql
```
Create a new user for postgres
```
sudo -u postgres createuser #username
```
Create a new database for postgres
```
sudo -u postgres createdb #dbname
```
Log in into the postgres console for that database
```
sudo -u postgres psql #dbname
```
Give a secure password to the user
```
(#dbname=#) ALTER USER #username WITH ENCRYPTED PASSWORD #password
```

Give the user all privileges to the database
```
(#dbname=#) GRANT ALL PRIVILEGES ON DATABASE #dbname TO #username
```

After exiting the console (Ctrl+D), restore the provided database image
```
psql -U #username #dbname < dbdump
```

6.2 Backend setup

This section explains how to install and setup `NodeJS and npm` and start the backend. This section assumes that the OS is *Solus Linux*. Since the only thing that differs between OSes is the *NodeJS* installation, it is advisable to look on the *NodeJS* website for additional installation instruction.

- Via package manager
- Via a graphical installer

When installing via package manager note that `npm` could be present as a separate package in your distribution's repository, if that's the case install it first via the recommended method for your distribution and then proceed in the guide.

Note: an instance of the backend is available at:
https://data4halp.herokuapp.com/

6.2.1 Requirements

- Internet connection.
- At least 50MB for the NodeJS binary and 100MB for the dependencies to be free on disk.

6.2.2 Setup

NodeJS and npm installation
```
sudo eopkg install nodejs
```

Go into the backend folder
```
cd IMPLEMENTATION/backend
```

Install the dependencies
```
npm install
```

6.2.3 Run

The application expects the following environmental variables to be set in the `start.sh` script.

- TEST_API="enabled"
  Enables logging and the stub enpoint
- DATABASE_URL
  Url of the database in the form of
  #dbuser:#password@#host:#dbport/#dbname
- JWT_SECRET="E9ql4cmZzDNG9qL8xh6F"
  The secret by which encrypt the jwt

- MAIL_PROVIDER="gmail"
  The email provider (List of supported mail providers)
- MAIL_ADDR
  The mail addres from which send the notification email
- MAIL_PASSWD
  The mail password
- LOCAL="enabled"
  Local testing
- PORT=12345
  Port on which run the application
- HOST="localhost:$PORT"
  The hostname of the application
- MIN_USER_NUMBER=2
  The minimum number of user from which accept the query

Run the application (on Linux and MacOS)
`bash start.sh`

6.2.4 Run the tests

After installing all the dependencies:
`./node_modules/.bin/jest`
Or if preferred, install `jest` globally:
`sudo npm install -g jest`
And then run the following command in the backend directory:
`jest`

6.2.5 Known issues

- The only tested email provider is GMail.

6.3 External services setup

6.3.1 Google Maps API

In order to be able to use the Google Maps View in the Flutter project you need to obtain a Maps API key from Google. To do so, please refer to the following webpage: Get API Key | Maps SDK for Android.

You should put the obtained key in the `AndroidManifest.xml` file. Please refer to the following document to get detailed instruction on how to do so: google_maps_flutter | Flutter Packages.

6.4 Flutter Application setup

In this section will be present all the necessary information to build the Flutter project.

6.4.1 Requirements

- At least 5GB of free space on the disk
- Internet connection

6.4.2 Flutter

In order to install the Flutter SDK with all its dependency, please refer to the following webpage: Install - Flutter.

Remember to also install all the needed dependencies to build Android Apps.

6.4.3 Build the app for Android

- Download the project into a local folder;
- open a terminal and go into the project folder;
- launch the command `flutter doctor` to check that the project is ok;
- launch the command `flutter packages get`;
- eventually change the backend's address with your address in `Config.dart`
- launch the command `flutter build apk` to actually build the apk.

6.4.4 Run the tests

In order to launch the test suite:

- open a terminal and go into the project folder;
- launch the command `flutter test` to run all the tests;

You can also run the test through Android Studio, clicking with the right mouse button on the test folder and selecting `Run tests in test...`

6.4.5 Known issues

- the app was developed under Android and has not been tested under iOS, so there can be some problem on this platform.
- You should include your personal Google Maps API as stated before, in order to make the Google Maps View work.

6.5 Website setup

In this section will be presented all the necessary information to run the website locally on your device.
Note: an instance of the website is available at:
http://nicofossa.altervista.org/data4help

The idea is to run the website locally on port 8000 using python with the terminal command
`python -m http.server`

6.5.1 Requirements

- Internet connection;
- Python.

6.5.2 Python

All the instructions to install Python on your device are explained in detail in the following link:
Beginner guide: How to download Python

Here the simple steps are resumed:

- Click on the python executable based on your OS. (windows, Max OS,...);
- Follow the instructions of installer.

6.5.3 Starting website on localhost

After having downloaded Python on your device you need only to open your terminal or cmd, access to the directory `frontend/website` present in the source zip, and type:
`python -m http.server`

Then you only need to open a browser and type `localhost:8000`.
You will be redirected into the index.html page.

7 Effort spent

During the initial part of the work we spent the great majority of time planning the initial scheduling of our tasks, in order to work as independently as possible during the later part of the project.

The first week of work was spent mainly on deciding which requirements to implement, dividing the tasks needed for the completion of the project, the technologies and frameworks to use; additionally we defined a mockup for the REST API.

After that we started each branch of the project separately, using the mockup defined together as guideline, although meeting periodically to keep each other update about the progress.

Finally we met to integrate the services and test every requirement with their action flow as defined in RASD and DD documents.

Specificaly, the team members focused on:

- **Nicola Fossati**: Flutter mobile app development and testing, design of the app user interface and experience, integration with Google Maps, integration testing with backend and initial database construction;
- **Daniele Montesi**: web app and initial database construction, integration and web user experience study;
- **Francesco Sgherzi**: mocking platform, backend design, development and testing, integration with email service, featuring testing and database construction and management.

Each member of the team spend around 110/120 hours on the project for both the IDT and the development.