POLITECNICO
MILANO 1863

Data4Help

Lorenzo, Molteni, Negri

ITD - Implementation & Testing Document

January 13, 2019

Contents

1 Introduction

1.1 Purpose and Scope

This Document contains a detailed and exhaustive explanation of the implementation of the *Data4Help* project. The purpose of this document is to explicitly detail the rationale of all choices made and provide a complete overview of how the project source code is structured and designed. The document also tackles the software testing activities and shows what are the main tests that have been performed on the product.

1.2 Definitions, Acronyms, Abbreviations

1.2.1 Definitions

- **Framework** a software framework can be defined as a standard way to build and deploy applications. It is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions. Software frameworks include compilers, code libraries, tool sets, and application programming interfaces (APIs).
- **Library** a software library is a set of pre-written pieces of code: classes, algorithms, structures and procedures that implement standard functionalities saving developers the time to implement them from scratch. It is important to point out the difference between libraries and frameworks. While libraries' functions are explicitly called by the programmer when they are needed, a frameworks makes programming faster by setting some constraints on how the project is structured and organised.
- **Package Manager** a package manager is a software tool that automates the process of installing, upgrading, configuring, and removing libraries. A package manager in particular deals with software dependencies and versions and prevents software mismatches and missing prerequisites. In this project two different package managers have been used: CocoaPods to help the development of the XCode project for the front-end and NPM, Node Package Manager, to deal with the back-end dependencies.

1.2.2 Acronyms

DBMS	Data Base Management System
REST	REpresentational State Transfer
API	Application Programming Interface
OS	Operating System
UI	User Interface

HTTP	HyperText Transfer Protocol
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
DB	Database
AWS	Amazon Web Services

Table 1: Acronyms

1.2.3 Abbreviations

- **(Rn):** n-th Requirement

1.3 Reference Documents

- RASD Document
- Design Document
- Implementation and Testing project assignment
- The Swift Programming Language (Swift 4.2)
- The Criteria Committee of the New York Heart Association. Nomenclature and Criteria for Diagnosis of Diseases of the Heart and Great Vessels. 9th ed. Boston, Mass: Little, Brown & Co; 1994:253-256.

1.4 Overview

The remaining parts of the implementation document are organised as follows:

- **Requirements implemented:** shows the functions that are actually implemented in the software and how they are mapped on the requirements identified in the RASD document.
- **Design choices and adopted Frameworks:** lists all the adopted programming languages, middlewares and frameworks adopted and provides their advantages and disadvantages. It also shows the reasons behind the implementations decisions made to follow the solution identified in the Design Document.
- **Source code structure:** explains how the source code is structured and organised for both the front-end and the back-end.
- **Testing:** provides information on how testing has been performed. In particular it describes the procedures followed and the main test cases that has been considered as well as their outcome.

- **Future development:** lists all non implemented functionalities and provides a description of future releases
- **Installation:** contains all the information needed to install and run the software.

2 Requirements Implemented

2.1 User Authentication

- [**R1**] The S2B allows users to create either a *Single User* or a *Third Party* account.
- [**R5**] To access the service users must log in with their account credentials.

Database
To access *Data4Help*'s services a user will be asked to register to the service. Information concerning registration is stored in our database in the Registration table. Each row in the table uniquely identifies a user through its userID and stores a boolean value saying whether the account has been activated or not. After registration, each login session will be identified by a token that will be bound to the user's ID until logout.

Frontend
The client application performs an automatic login if some user credentials are already stored in the *System*. Otherwise the user will be brought to the login view, which present an undifferentiated form to fill (username, password) in order to access the *System* either as a *Single User* or as a *Third Party*. When the user clicks on the login button the application checks whether all fields have been filled and proceeds to send a HTTP POST request to the backend. In case of positive response the application uses the content of the response message to determine whether the logged user is a *Single User* or a *Third Party* and to consequently perform a segue to the appropriate view. For what concerns the registration of new users, the *System* provides differents views for *Single User* and *Third Parties*, each with the appropriate form fields to fill. If after clicking the registration button a positive response is received the applications goes back to the login view. The user, after activating its account from the email, will be able to log in.

Backend
Authentication is managed by the /req/auth endpoints, offering registration for a *Single User* and for a *Third Party*, account activation and login. When registering, the database is queried to find out whether the provided data is already associated to another account; if not, a unique userID is generated, and an activation token is sent by email to the address using to register. When the user activates its account, if the token is valid, the database is updated to complete the creation process. Then, whenever the user logins with correct credentials, a secure token is generated using JWT (HS256 encryption), storing the id of the user and the type of account (*Single User* or *Third Party*) as its content.

2.2 User Management

- [**R1**] The S2B allows users to create either a *Single User* or a *Third Party*

account.

- [**R2**] A *Single User* account can be created if and only if the user provides his Fiscal Code.
- [**R3**] A *Third Party* account can be created if and only if a valid P.IVA is provided.
- [**R4**] All users can create an account if and only if they provide a unique email and a password.
- [**R5**] To access the service users must log in with their account credentials.
- [**R35**] The S2B allows organisers to create a *Third Party* account using *Data4Help*.
- [**R43**] The S2B allows runners to create a *Single User* account using *Data4Help*.

Database
According to the type of account created, all information concerning a user will be stored in either the PrivateUser table or the ThirdPartyUser table. In both tables users are uniquely identified by a userID and an email. Both will be associated with a properly hashed password. The table concerning *Single User* data will also store the user's FC, if available, its name, birthday and sex. That concerning *Third Parties* instead will store the company's name, P.IVA and description. Finally, the UserSettings table uniquely concerns *Single User* accounts and stores information about which types of health data to import from their device. When imported, all health data concerning a user is stored in the UserData table. Each record is identified by the user's userID, its type, its timestamp and stores the measured value.

Frontend
When the user opens the app for the first time he will be asked what kind of health data he is willing to import into *Data4Help*'s *System*. User credentials and information are set during registration phase and following the thin client paradigm there is no persistent user model, but all data (shown in the settings tab) are retrieved with a HTTP GET request to the backend. Both *Third Parties* and *Single Users* can modify some of their information at any time from the Settings tab. Some critical information cannot be changed after registration. These include: email, FC, P.IVA.

Backend
For settings management, two endpoints are offered: /single/info and /tp/info, respectively for *Single Users* and *Third Parties*. When sending a GET request to those endpoints, the server queries the database to retrieve the settings of the user that can be modified, thus not including password and sex, for example. When the request received is a POST, the body will contain the fields that the user wants to update, so the database will be queried to save these new settings.

2.3 Creation of Single Requests

- [**R9**] Third Parties can submit a request to access data of a *Single User*.
- [**R10**] Single Requests must specify either the email or the FC of the desired user.
- [**R11**] Single Requests are forwarded only to the specified user.
- [**R12**] A user can accept or refuse requests forwarded to him.
- [**R13**] Third Parties can access user's data if and only if their request is accepted by said user.
- [**R17**] Single requests must specify the requested data types.
- [**R18**] A request to a *Single User* must specify whether or not the *Third Party* is subscribing to that request of data.
- [**R19**] Subscriptions to requests must specify a duration.
- [**R20**] Subscriptions to requests can be ended by both the *Third Party* and the *Single User* at any time.
- [**R21**] If none of the requested data types of the *Single User* is available, the *Third Party* receives an error message.
- [**R22**] Third Parties can access only the requested data types that are available.
- [**R23**] Third Parties can download all data obtained through requests on their devices or have it sent by email.
- [**R39**] The request is sent on behalf of the organiser using its email and P.IVA.
- [**R40**] The request sent is with a subscription that lasts until the end of the run.
- [**R41**] The request sent by the S2B has as requested data types the position and all available health parameters of the user.
- [**R47**] A user can't accept the request if he doesn't have at least his position available as requestable data type.

Database
When a *Third Party* creates a Single Request, all its related information is stored in the SingleRequest table. Each request is uniquely identified by a requestID and specifies its *Third Party* sender, *Single User* receiver, its status and specification on whether it is a subscription or not. The content of each request must be stored in the RequestContent table. The content includes the request's unique ID and the requested type of health data. Each Single Request

addresses a *Single User* that is able to accept it or refuse it. The information on the status of the request is stored in the SingleRequest table under the field status that is initially pending and is modified to accepted or refused according to the *Single User*'s action. The various statuses are described by the request-status enumeration.

Frontend
Third Parties can send single requests from the associated view in the Research tab. The view contains a form to be filled with either FC or email to identify the *Single User*, and a serie of toggles to specify what kind of data are requested. After the "Send request" button is pressed an HTTP POST request containing all the information is sent to the backend. *Single Users* can see all the request pertaining to them in the "MyFollowers" tab and can accept or refuse pending requests by tapping on some buttons. *Third Parties* can see a list of all the Single Requests they send in the "MyHistory" tab and they are able to either stop the eventual subscription prematurely or download a csv file containing all the data related to a certain Group Request.

Backend
For *Third Parties*, the server offers the /tp/sendSingle endpoint to create and send a request to the target user. To be able to send it, there shouldn't be another pending request from the *Third Party* to the *Single User*. If this condition is met, then the request is inserted in the DB. *Third Parties* can then download the results of a request by sending a POST to /tp/downloadSingle, but only if such request (identified by its unique id) was approved, in this case data is retrieved from the database up until the date of the request, or the end of the subscription (if present). *Single Users* can accept or refuse a request they received through the /single/choice endpoint, and also terminate an ongoing subscription through the /sub/endSingle endpoint. To retrieve the list of requests meant for them, *Single Users* can query the /single/list endpoint.

2.4 Creation of Group Requests

- [**R19**] Subscriptions to requests must specify a duration.
- [**R23**] Third Parties can download all data obtained through requests on their devices or have it sent by email.
- [**R26**] Third Parties can submit a group request to access data of groups of users.
- [**R25**] Group requests must include at least one search parameter.
- [**R26**] Group requests must specify the requested data types.
- [**R27**] Group request results are provided if and only if the number of users matching the search parameters is higher than 1000.

- [**R28**] Group request results include only the data retrieved by the *System* matching the search parameters.
- [**R30**] All group requests must specify whether the *Third Party* is subscribing to that request of data.
- [**R31**] Subscriptions to requests can be ended by the *Third Party* at any time.

Database
All information concerning group requests created by *Third Parties* is stored in the GroupRequest table. This keeps track of all requests identified by a unique requestID and specifying the *Third Party* sender, the request status and information concerning the subscription. Furthermore each group request may be associated to one or more tuples in the SearchParameters table. This stores all filters for group requests by associating a requestID to a type of health data, its requested lower and upper bounds. All group requests have initially set the field status to pending. However, once they are evaluated, the field is then set to either accepted or refused according to the evaluation of the filters on users.

Frontend
Third Parties can send group requests from the associated view in the Research tab. The view contains a serie of toggles to select the requested data and a number of sliders and text fields to specify the filtering parameters for the request. After the "Send request" button is pressed an HTTP POST request containing all the information is sent to the backend. *Third Parties* can see the list of all the Group Requests they sent in the "MyHistory" tab and they are able to either stop the eventual subscription prematurely or download a csv file containing all the data related to a certain Group Request.

Backend
The server offers the /tp/sendGroup endpoint to create and send a group request. To download the requested data, and thus have the request evaluated from the server, *Third Parties* can interrogate the /tp/downloadGroup endpoint, where the condition of having more than 1000 users matching the Search Parameters is checked, and if it's true then data is provided to the *Third Party* anonymously. Like for *Single Users*, *Third Parties* can access the /req/tp/list to retrieve a full list of all their single and group requests, and they can terminate ongoing subscriptions through both the /sub/endGroup and /sub/endSingle endpoints.

2.5 Data Management

- [**R29**] Sensitive data is excluded from group request results.
- [**R6**] The S2B automatically imports new data whenever the application is opened.

- [**R7**] When the application is open, the S2B continuously import data in background.
- [**R8**] The S2B binds collected data only to the user's account that imported it.

Database
User related data that is stored persistently belongs to two categories: sensitive data and health data. The first type includes user's FC, email, position and password. These are not filtered by database queries and not available to *Third Parties* performing group requests. Furthermore, passwords are hashed in the database to ensure privacy. The second category concerns user's health data that, as mentioned above, is stored persistently in order to provide health monitoring services.

Frontend
The client application saves a copy of the credentials of a logged user to perform an automatic login using Swift UserDefaults. A cached version of latest health data is instead saved using another persistent storage option offered by Swift, Core Data. All the data that pertains to the results of the requests comes from the backend and can be downloaded by *Third Parties* and shared or saved in a number of ways.

Backend
The instances of the application server are completely stateless, so all data is stored on the database. To ensure protection of sensible data, passwords are hashed before inserting them in the database, requests from the clients must contain an encrypted token to identify the sender, and the results of a group request do not contain data that can pinpoint a user (including its id, which is replaced by another one).

2.6 Health Status Monitoring

- [**R14**] The S2B allows *Single User*s to visualise their historical data using Time Series.
- [**R15**] The S2B allows *Single User*s to visualize their historical data using aggregated statistical operators.
- [**R16**] The S2B allows *Single User*s to visualize their historical data over multiple timespans.

Database
Data4Help's health monitoring services is provided by storing health data of all its users relating it to its type and time of acquisition. As mentioned above, a *Single User*'s health data is stored in the UserData table, where each record is identified by the userID, the type of health data, its timestamp and the measured value. The types of health data that can be stored are defined in the

datatype enumeration and used all around the schema.

Frontend
The client application takes charge of importing all the health data that the user allowed to be acquired by *Data4Help* monitoring service. All the information are retrieved from HealthKit which is a framework provided by iOS designed to store all health related data coming from smartwatches, smartbands, healthcare and fitness apps or directly from the user. These information are periodically sent to the backend system when the application is active or running in background.

Backend
For what concerns data management, the /upload endpoint is used by *Single Users* to upload their data, while the /stats/avg endpoint can be used to retrieve useful statistics regarding both the requesting user and other users (anonymised) as a comparison.

2.7 AutomatedSOS

- **[R32]** Only private users can choose whether or not to enable *AutomatedSOS*.
- **[R33]** *AutomatedSOS* can be enabled only if the user grants permission to make emergency phone calls.
- **[R34]** If *AutomatedSOS* is enabled and the *System* detects that a user's heart rate is below or above the critical threshold for his age, an ambulance is called.

Database
AutomatedSOS's functionalities are exclusively implemented in the client application and do not need to store persistent data onto the database.

Frontend
The client application implements *AutomatedSOS* feature in its entirety. *Single User*s can toggle a switch inside MyHealth tab and activate *AutomatedSOS* feature. When *AutomatedSOS* is on and the *System* register abnormal heart rate or systolic/diastolic pressure activity it prompts the user to receive medical attention by automatically suggesting a call to the emergency service. The user is free to decide whether to tap the call option or to dismiss the alert.

Backend
AutomatedSOS's functionalities are exclusively implemented in the client application and do not need the application server to work.

2.8 Run Management

- [**R36**] Only Third Parties can create a run.
- [**R37**] When creating a run, the organiser must specify its the path, duration and maximum participants.
- [**R38**] On the day of a run the S2B sends a Single Request to every user who has registered to that run.
- [**R39**] The request is sent on behalf of the organiser using its email and P.IVA.
- [**R42**] The S2B provides a list of all existing runs visible to everyone using *Track4Run*.
- [**R44**] Only users with a *Single User* account can join an existing run.
- [**R45**] A user can only join a run in the list provided by the S2B.
- [**R46**] A user can't check-in for a run if he doesn't accept the request received by the organiser of such run.
- [**R47**] A user can't accept the request if he doesn't have at least his position available as requestable data type.
- [**R48**] If a user fails to check-in in the run, the S2B removes him from it.
- [**R49**] An existing run can be spectated without logging in the *System*.
- [**R50**] An existing run can be spectated by selecting it in the list of existing runs.
- [**R51**] Spectators can see the live position of all runners participating in the run they are spectating.

Following the requirements expressed in the Implementation and Testing Project Assignment, we decided to implement *AutomatedSOS* as the additional service, thus all requirements regarding *Track4Run* are not satisfied.

3 Design Choices

3.1 Database choices

PostgreSQL
In order to securely store the large amount of data managed by the *Data4Help* system, several DBMS have been considered. After opting for a relational model, as stated in the Design Document, we looked for a DBMS that could meet our needs. We considered:

- MySQL
- Microsoft SQL
- PostgreSQL

We chose PostgreSQL because, along with MySQL, is the only real open-source solution. It has been preferred over MySQL due to the fact that MySQL licence forces either to make the developed software available as open-source or to buy a licence from Oracle, that is the owner of MySQL. PostgreSQL instead is released under the PostgreSQL license, a liberal Open Source license, similar to the BSD or MIT licenses.

RDS
The PostgreSQL database instance is deployed on AWS, using a free tier subscription for the RDS service. It consists in a db.t2.micro PostgreSQL instance, with 20 GB of storage and automated backups with a retention period of one day. The creation and management of tables, as well as the insertion of mock data, has been supported by MySQLWorkbench, a visual database designing and modeling tool for MySQL server relational database that facilitates the creation of new physical data models and modification of existing database tables.

3.2 Back-end choices

Node.js
In order to develop the backend system for an application like *Data4Help*, which can be very data intensive as the number of users grow, we had to find a platform whose main requirement is offering a non-blocking environment. We considered:

- Node.js (JavaScript)
- JavaEE (Java)
- Django (Python)

Amongst the three, we chose Node.js as it natively offers an event-driven asynchronous environment with non-blocking I/O operations. With the releases of Nodes after version 7.6, it's possible to use Async-Await statements instead of Callbacks, making it easier to write simple and manageable asynchronous code.

Having a single thread is not problematic for our application, since no heavy computation is required on the application server, but it's mainly I/O operations with regards to the database.

Moreover, Node.js comes with a complete package manager in NPM, offering easy management of package requirements through the package.json file, useful especially since the server should be replicated. Despite being relatively new, Node.js paired with JavaScript offers a wide range of external packages, like Express for network management, which are stable and well supported by the community.

External libraries
The most important packages, available through NPM, that were used to develop the server, are:

- **Express:** the standard framework for web-applications in Node.js, allowing to define fully REST-compliant APIs. Paired with helmet as middleware to guarantee an additional layer of security.
- **Pg:** a non-blocking PostgreSQL client used to create and manage connections with the database.
- **Joi:** a schema description language that allows to specify a structure for objects, and validate them. Its main use was to describe the proper structure of HTTP requests.
- **Config:** a manager for configuration files, allowing to specify environmental variables and other settings based on the different execution environments (production, development, testing).
- **Morgan and Winston:** two libraries used respectively for logging requests and errors.
- **Nodemailer:** utility library to send activation emails to complete the registration of a user.
- **Bcryptjs:** utility library to safely hash passwords using salts.
- **Jsonwebtoken:** Node.js implementation of JSON Web Tokens, used to create encrypted authentication

EC2
For what concerns the deployment of the server, we found the most viable options to be AWS and Heroku. We chose the former over the latter for various reasons, including:

- Homogeneous environment with regards to the database, allowing to share user policies and accounts. This also includes Amazon tokens for computational powers and storage space.

- Easier scalability through Auto Scaling.
- 99.99% guaranteed Availability, matching the requirements expressed in the RASD.

The EC2 instance is a t2-micro running Ubuntu Bionic 18.04 LTS, and is reachable with the following static ip we exported: 52.57.95.222. For configuration and direct management SSH was used to access the instance.
The instance is running with an nginx proxy on top of it, forwarding all incoming requests to 127.0.0.1:3000, where our Node.js application is listening.
We setup PM2, a Node.js process manager, to automatically launch our server with the right configuration after every reboot, so there is no need to launch it manually, even after a crash of the instance.

3.3 Front-end choices

3.3.1 iOS

Data4Help is developed as a native iOS App exclusively designed for iPhones. All users owning an iPhone that runs iOS 8 or above will easily download the application from Apple's App Store. The following reasons led us to choose the iOS framework to develop our user interface:

- **Operating System** iOS is a widely spread operating system, approximately owning a 19% share in the worldwide market. Furthermore, 76% of iPhone users run the latest version of the operating system compared to only 6% for Android users. Running the most updates software version will enable us developers to release new versions of the app and allow it to run smoothly and fully optimized at all times.
- **Documentation** Apple provides full documentation for iOS developers, including guides about app marketing, design and business. The Apple Developer Documentation is by far the most comprehensive and extensive resource about iOS development. It includes documentation for Foundation, the fundamental framework that underlies iOS apps, and the Swift Standard Library.
- **Interface peculiarity** iOS applications have a native look and feel. Furthermore, Apple has released design guidelines to encourage a unique brand look among the millions of apps in their stores
- **Device compatibility** Developing an iOS application enables to adapt the user interface to all existing iPhone models. This feature limits device fragmentation that would be inevitable when developing in the Android Framework due to the uncountable number of shapes and sizes of its compatible devices.
- **Frameworks** iOS provides useful and fully documented frameworks to work with during development. Among these we have used Healthkit to which we will dedicate a brief paragraph below.

- **Programming language and environment** Apple supports developing with its latest version of Swift, a simple and easy to use programming language to which we will dedicate a brief paragraph below.

While having all the above described positive features, developing an iOS app limits the service's portability, as the entire front-end code would have to be rewritten for a different framework, and has pretty high maintenance costs.

3.3.2 Swift

Apple encourages developers to adopt the latest version of Swift programming language for their applications. Swift is a fully documented language with many positive features such as the following:

- **Simplicity:** Swift is a programmer-friendly language. It is simple and easy to use thanks to its lightweight syntax and to its possibility to experiment with code and see immediate results in a playground.
- **Speed:** Swift is compiled and optimized to get the most out of modern hardware.
- **Built-in patterns:** Swift limits common syntactic and stylistic errors thanks to built-in programming patterns such as the following:
  - Variables are always initialized before use.
  - Array indices are checked for out-of-bounds errors.
  - Integers are checked for overflow.
  - Optionals ensure that nil values are handled explicitly.
  - Memory is managed automatically.
  - Error handling allows controlled recovery from unexpected failures.

3.3.3 External Frameworks

In order to implement many of *Data4Help*'s core features we have adopted external libraries installed through Cocoapods. Cocoapods is an application level dependency manager for Swift that provides a standard format to manage external libraries. CocoaPods focuses on source-based distribution of *Third Party* code and automatic integration into Xcode projects. Our application is developed using the following frameworks:

- **Alamofire:** Alamofire is an HTTP networking library written in Swift. It aims at creating a networking interface that feels native to Swift by harnessing NSURLSession and the Foundation URL Loading *System*. It is built to handle asynchronous calls and performs efficiently thanks to many best practices.

- **SwiftyJSON:** This library enables developers to seamlessly deal with JSON types in Swift and avoid explicit typing that may result in verbose code. It is essential to handle networking requests that are encoded in JSON.

- **Charts:** This library allows to create beautiful and elegant charts for iOS applications including bar charts, pies, bubbles and radars. It is a great solution to provide users with a direct, simple and enjoyable view of statistics concerning their health data.

3.3.4 Apple Frameworks

Data4Help is developed by using the following iOS frameworks:

- **Foundation:** This framework lies at the core of all iOS applications. It provides a base layer that includes data storage and persistence, text processing, date and time calculations, sorting and filtering, and networking.

- **Healthkit:** *Data4Help*'s core feature is the retrieval of users' health data and the monitoring of their health status. Thanks to HealthKit we can easily retrieve all health data collected by the user onto his smartphone. This becomes a valuable data source to provide meaningful health analysis to all users. This framework also allows to seamlessly integrate other medical wearable devices. Their measured data is in fact collected by iOS in the Health app and becomes immediately accessible to developers.

- **UIKit:** This framework is essential to build the required infrastructure for an iOS app. It provides developers with a window and view architecture for the user interface, the event handling infrastructure for delivering Multi-Touch and other types of input to the app, and the main run loop needed to manage interactions among the user, the *System*, and the app. Among its more advanced features, we have also adopted 3DTouch to implement quick actions for a more user friendly experience.

- **CoreData:** This framework allows us to handle a model layer on our client application. It provides a relational model to store and manage simple data directly onto the user's device. We have adopted this framework mainly to temporarily store health data and statistics coming both from the Health app and from the application server (if concerning compared data).

4 Source Code Structure

4.1 Backend project structure

The backend was developed using JetBrains WebStorm as IDE, but any IDE supporting JavaScript can be used. The root of the project is organised with the following custom folders:

1. **bin**: contains only the actual executable, the www.js file.
2. **config**: contains all the .json files for the settings of the project, values for environmental variables, and PM2 script.
3. **middlewares**: contains all the middlewares used by Express.
4. **routes**: contains all Express endpoints.
5. **schemas**: contains all Joi schemas for validating the requests.
6. **startup**: contains scripts to be executed at startup, which associate middlewares to the Express app.
7. **tests**: contains all unit and integration tests.
8. **utils**: contains utility classes like the database setup, mail sender and security functions.

An in-depth description of each folder will follow.

4.1.1 bin

Containing only www.js, this is the folder for executables. The purpose of www.js is to create a server, listening on the port provided as environmental variable (if none is set, port 3000 is used). After creating the server, the Express app is required, running all startup scripts, and then associated to the port the server is listening on.

4.1.2 config

This folder contains the ecosystem.config.js file, used by PM2 to start the Node application, which specifies the path to the executable to run and the settings for each environment. Other than this file, there are 3 JSON files, used by the config module: default.json, production.json and test.json. The first one, default.json, contains the default settings that are used if no execution environment is specified. These settings includes the properties of the database, of the mail service, the hostname of the server and the private key used to encrypt and decrypt Json Web Tokens. The second one, production.json, overrides only the hostname property, and sets it to the static ip of our deployed EC2 instance. The third one, test.json, overrides the properties of the database, allowing to use a local test database instead of the RDS one.

4.1.3 middlewares

This folder contains the custom middlewares used in the project: authenticator.js, error.js, validator.js. The first one, authenticator.js, deals with the authentication process involving users after their registration and login. As previously said, all HTTP requests must contain an header, "x-authToken", which is a Json Web Token that, once decrypted using the jwtPrivateKey in the configuration files, contains two fields: userid and usertype. If decryption fails, the authentication process is stopped and an error is returned to the client, otherwise the request is filled with the two decrypted fields and passed to the next middleware. The second one, error.js, defines an error logger using Winston, which logs everything on console and on file, and also defines a middleware whose purpose is to log the error with the logger, send a HTTP 400 response to the client, and rollback the database. This error middleware is called when there's an error involving a query to the database, for example a violation of an integrity constraint. The final middleware, validator.js, takes every request and validates it against the Joi schema defined for that request in the *schemas* folder. If validation succeeds, the request is passed to the next middleware, otherwise an error response is returned to the client.

4.1.4 routes

This folder contains all the Express routes offered to clients: auth.js, data.js, req.js and settings.js. App.js is also included here, whose purpose is to require all middlewares and bind them to the app object before exporting it to www.js. The Network Manager can be seen as implemented by both App.js and bin.js. The first route, auth.js, is the implementation of the Authentication Manager, and offers all the endpoints to register an account, activate it and login. The second route, settings.js, is the implementation of both the *Single User* Manager and *Third Party* Manager, offering all the endpoints to retrieve the settings of an account (password, full name, company name and such), and to modify them. The third route, data.js, is the implementation of the Data Manager, offering all endpoints to upload data of a *Single User* and to retrieve useful statistics from it. The fourth and final route, req.js, is the implementation of both the Single Request Manager and Group Request Manager, offering all endpoints to send requests, retrieve their lists, download their results, accept them and terminate their subscriptions.

4.1.5 schemas

This folder contains only schemas.js, which embeds all the schema objects used by validator.js to validate each request against its schema. Each schema is exported by binding it to the path of the endpoint whose request should validate.

4.1.6 startup

This folder contains all the scripts to be executed at startup to create the middlewares: logging.js, prod.js, routes.js and validator.js. The first one, logging.js, takes the Express app as input and binds to it only Morgan, then defines two functions to handle all unhandled exceptions. The second one, prod.js, binds to the app only middlewares that should be used in production, namely Helmet and Compression. The third one, routes.js, binds all endpoints in the routes folder to the Express app, and finally binds the error middleware to it. The fourth and final one binds the Joi validator and defines the format of requests (JSON).

4.1.7 utils

This folder contains different files. The first one, dbconnection.js, is the implementation of the Data Storage Manager, handling the connection to the production database (RDS) or the test database (local), depending on the execution environment. It also offers a method to query the db, using a connection pool. The second one, mailer.js, only handles the dispatching of activation mails, and connects to the used provider. The third one, utils.js, contains utility functions, among which also the Privacy Manager to hash the passwords.

4.2 Database description

The implemented structure of the database is depicted in the following E-R diagram.

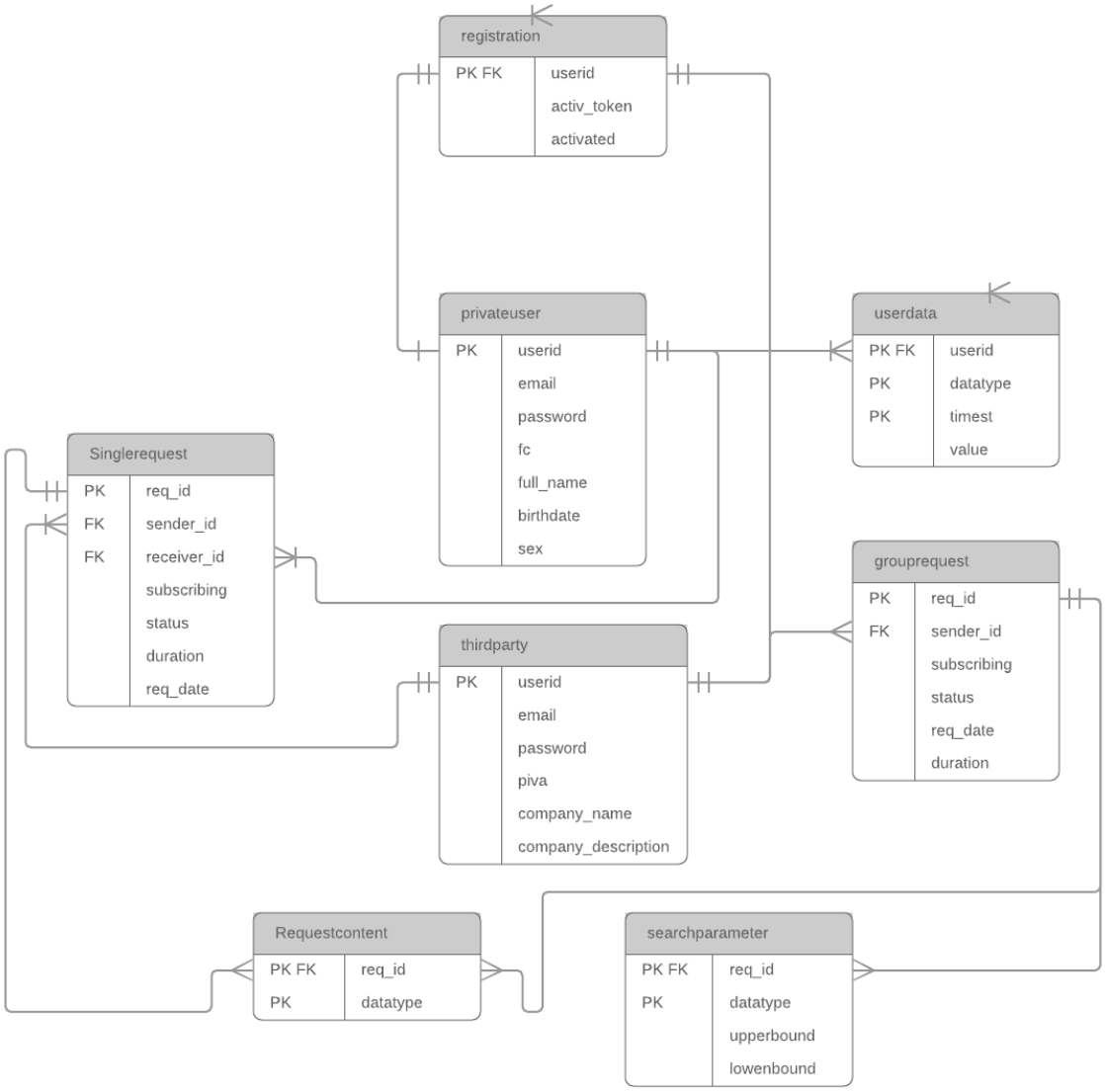


Figure 1: Database ER diagram

4.2.1 Enums

Some custom data types have been defined to support the storage of health data and other various information such as the gender of the *Private User* and the status of a *Single Request*. The table down below shows the detail of each of them.

ENUM	VALUES
datatype	standinghours, heartrate, distancewalkingrunning, sleepinghours, weight, height, age, activeenergyburned, stepcount, systolic_pressure, diastolic_pressure
gender	F, M
requeststatus	pending, accepted, refused

Table 2: Enums

4.2.2 Tables description

Here follows an accurate description of how all the tables designed in the previous document (See Design Document) have been implemented. The description includes for each table the column name, the data type, if it is a primary key and if it is nullable.

Group Request

Column name	Data type	PK	Nullable
duration	integer	NO	YES
req_date	date	NO	NO
req_id	integer	YES	NO
sender_id	varchar	NO	NO
status	requeststatus	NO	NO
subscribing	boolean	NO	NO

Singlerequest table

Column name	Data type	PK	Nullable
req_id	integer	YES	NO
sender_id	varchar(10)	NO	NO
receiver_id	varchar(10)	NO	NO
subscribing	boolean	NO	NO
status	requeststatus	NO	NO

duration	integer	NO	YES
req_date	date	NO	NO

Privateuser table

Column name	Data type	PK	Nullable
userid	varchar(10)	YES	NO
email	varchar(40)	NO	NO
password	varchar(100)	NO	NO
fc	char(16)	NO	NO
full_name	varchar(30)	NO	YES
birthdate	date	NO	YES
sex	gender	NO	NO

Thirdparty table table

Column name	Data type	PK	Nullable
userid	varchar(10)	YES	NO
email	varchar(40)	NO	NO
password	varchar(100)	NO	NO
piva	char(11)	NO	NO
company_name	varchar(20)	NO	YES
company_description	varchar(100)	NO	YES

Registration table

Column name	Data type	PK	Nullable
activ_token	varchar(11)	YES	NO
activated	boolean	NO	NO
userid	varchar(10)	YES	NO

Requestcontent table

Column name	Data type	PK	Nullable
req_id	integer	YES	NO
datatype	datatype	YES	NO

Searchparameter table

Column name	Data type	PK	Nullable
req_id	integer	YES	NO
datatype	datatype	YES	NO
lowerbound	integer	NO	YES
upperbound	integer	NO	YES

Userdata table

Column name	Data type	PK	Nullable
userid	varchar(10)	YES	NO
datatype	datatype	YES	NO
timest	timestamp	YES	NO
value	float8	NO	NO

4.3 XCode project structure

In this section we will describe the structure of our client application. It is implemented as an iOS Application using Swift programming language. The structure of the Xcode Project follows Apple's best practice guidelines on the MVCS design pattern. As a result, all classes are organized in four sections: Views, Controllers, Model and Utilities. Controllers in Swift include also View-Controllers, classes merging features of both views and controllers such as being responsible for the handling and updating the content of views, responding to their interaction with the user and for coordinating with other objects such as other controllers in the app. In the following paragraphs we will details each section with its main classes and functionalities.

4.3.1 Views

In this section we will describe how all views are implemented and organized within the project. Every view has been designed starting from a Storyboard, a unique XCode file that allows to prototype and design multiple views and connect them with each other. The project contains a total of three storyboards: "SingleUser.storyboard", "ThirdParty.storyboard" and "Main.storyboard", each containing a sequence of views belonging to a same section. Every view in the storyboards is connected to a specific subclass of UIViewController responsible for its content and its interaction with all other objects.

The **"Main.storyboard"** represents the sequence of views that are displayed when a user has not logged into his account. The user may either create a new *Single User* or *Third Party* account or log in with his credentials. When a user successfully enters his credentials a *segue* is wired programmatically to connect the current view controller to the following one to be visualized. For this purpose, the Main.storyboard keeps a reference to both "SingleUser.storyboard" and "ThirdParty.storyboard".

The **"SingleUser.storyboard"** contains all views that can be accessed by a user when he has logged into a *Single User* account. There are three main views connected to each other by a tab bar controller. Each view is associated to a tab: MyHealth, MyFollowers and MySettings. The first view is responsible for the health monitoring feature. It displays simple bar and bubble charts to get immediate information regarding the user's health status and the comparison with the average for his age. MyFollowers is organized with a table view with custom cells to display all requests directed to that user. All the contained requests are either past ones or pending ones to be accepted or refused. A navigation bar is also used to retrieve a specific one. Finally, the MySettings view contains all general information associated to the user's account, such as email, password and birth date, and enables their editing.

The **"ThirdParty.storyboard"** contains all views that can be accessed by a *Third Party* user when he logs into his account. As for the previous storyboard, all views are organized by a tab bar controller. Each view is associated to one of the three tabs: Research, History and Settings. With the Research view a *Third Party* user can create either a single or group request by selecting its filters onto the display. The History tab allows to retrieve all past and current requests and check whether they are still pending or have been accepted or refused. Finally the Settings sections displays all the general information associated to the *Third Party* account and enables their editing.

Cells
Under this folder are stored all the classes used to represent and manage a custom cell in a UITableViewController.

- **SURequestCell.swift**: is the cell used in *Single User*'s MyFollower tab table view. Its content shows information about a *Single Request* sent to the logged *Single User*. Namely it shows the *Third Party* name, the requested datatypes and the application of a subscription. In case of a pending request two buttons are also added in order to either accept or refuse it.

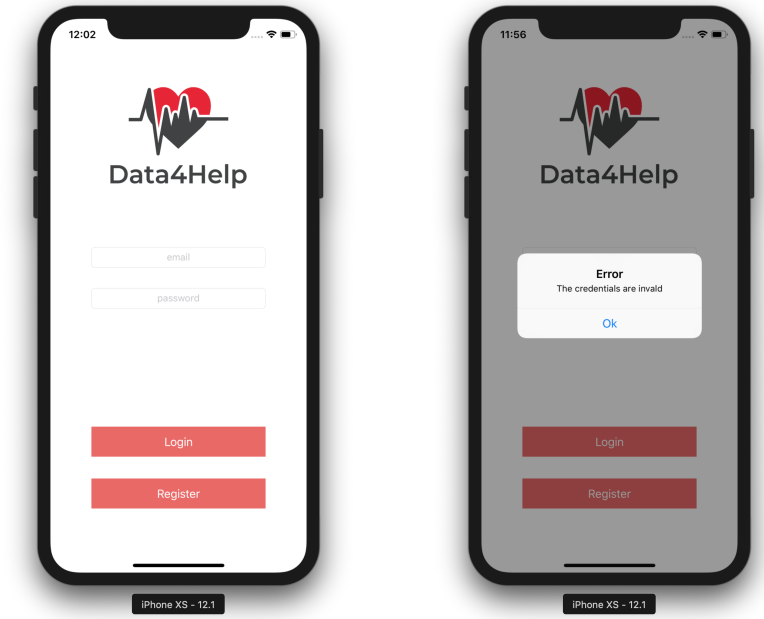
- **TPRequestCell.swift**: this cell represents the content of a *Single Request* in the MyHistory tab of a *Third Party*. It shows the name of the *Single User* the request was sent to, the requested datatypes, the date and whether it is subscribing or not. The cell also present a toggle that allow a *Third Party* to end a subscribing request before its expiring date. Finally there is a download button that allow the user to download the csv containing the data of an accepted request.

- **TPGroupRequestCell.swift**: this cell is used in the same context of the previous one but to represent the content of *Group Requests*. The main difference lies in the cell layout.

4.3.2 Controllers

Main Controllers

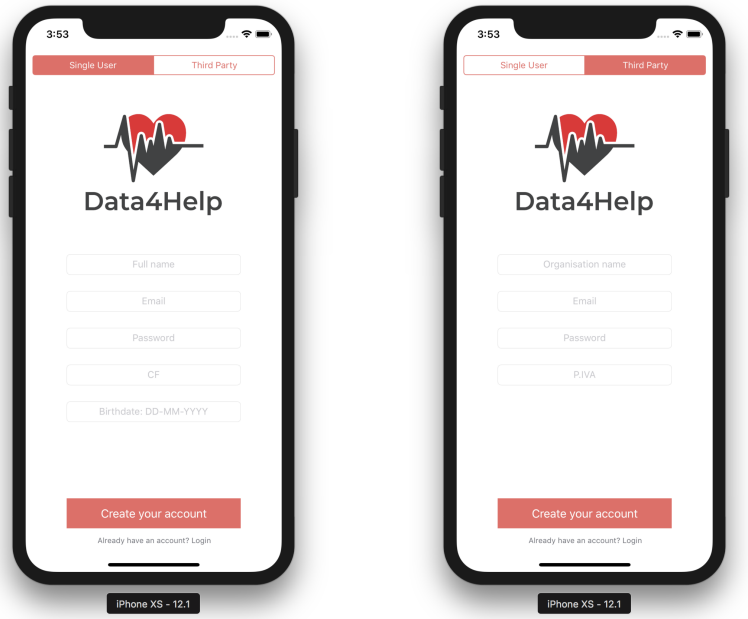
- **LoginViewController.swift**: this is the initial view controller and it handles login operations. Depending on the user interaction it can perform a segue to the Registration Controllers or, after sending a login request, perform a segue to *Single User* or *Third Party* storyboard. If the input credentials are invalid, an error prompt informs the user.



(a) Login view (b) Login error

Figure 2: Login screenshots

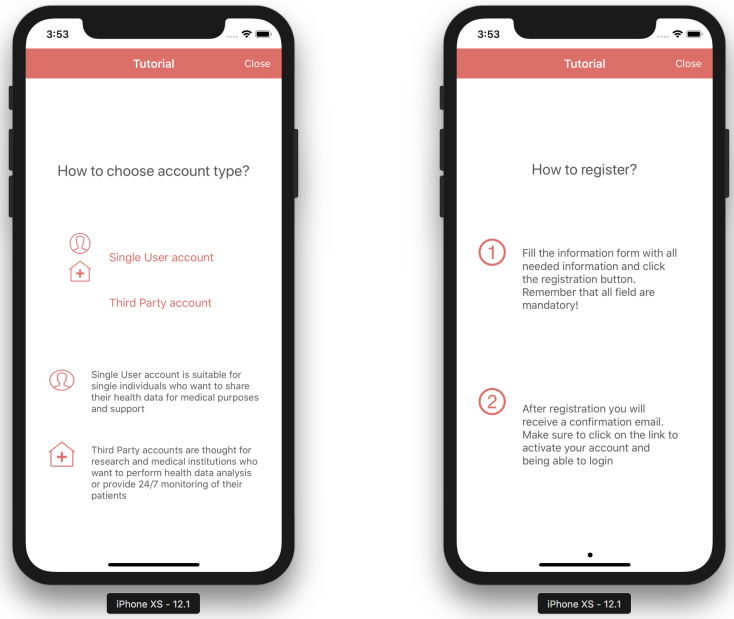
- **SURegisterViewController.swift**: it shows the registration form for *Single Users* and sends registration requests to the backend. In case of successful response it performs a segue to login view.
- **TPRegisterViewController.swift**: it shows the registration form for *Third Parties* and sends registration requests to the backend. In case of successful response it performs a segue to login view.



(a) *Single User* register view (b) *Third Party* register view

Figure 3: Register views screenshots

- **TutorialViewController.swift**: it presents a list of simple instruction useful to decide what is the most suitable kind of profile for the new user. It also tells how to complete the account activation phase after signing up from the application.



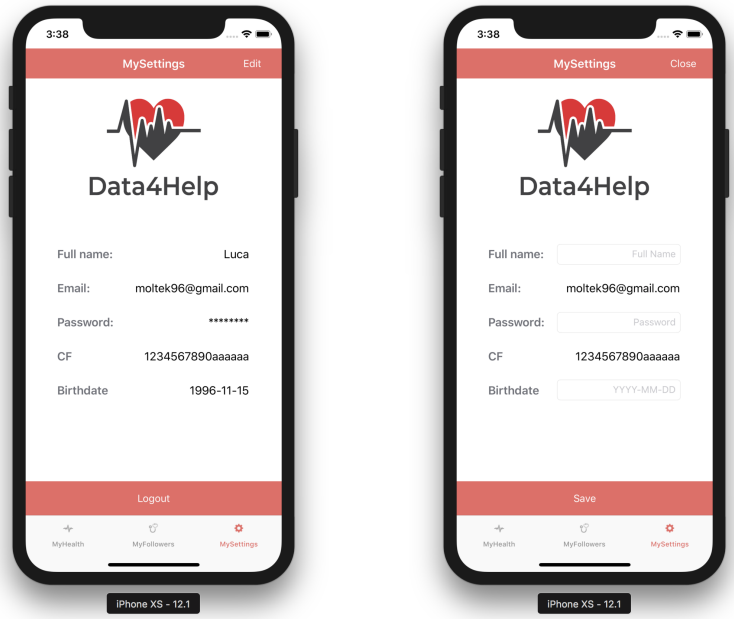
(a) Tutorial page 1 (b) Tutorial page 2

Figure 4: Tutorial view screenshots

SU Controllers

Settings controllers

- **SUSettings.swift**: acts as a container for the views managed by SUViewSettings and SUEditSettings. These two views are alternatively shown to the user depending on whether he pressed an edit button in the top right corner of the screen. The view also present in the bottom part a button allowing to logout or to save the edited informations.
- **SUViewSettings.swift**: manages the view containing the info and settings related to the logged *Single User*. They namely are full name, email, password (obscured), FC and birthdate.
- **SUEditSettings.swift**: presents a layout almost identical to SUViewSettings but some field are presented as texfields in order to allow the editing of some information.



(a) Settings view

(b) Edit settings

Figure 5: *Single User* Settings tab screenshots

MyFollowers controllers

- **RequestController.swift**: it manages the UITableView containing all the received requests. The request are divided in three sections: accepted, pending and refused. It also provides a search bar to easily filter requests based on the sender name.
- **MyCellDelegate.swift**: a protocol allowing SURequestCell class to call some methods from RequestController.

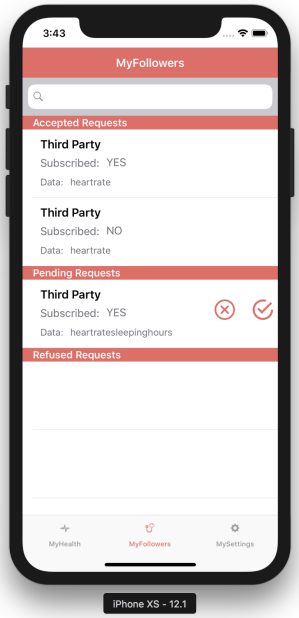
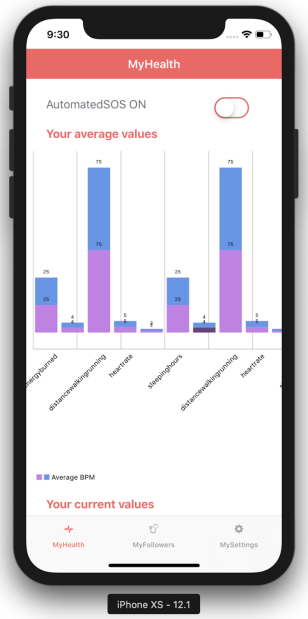


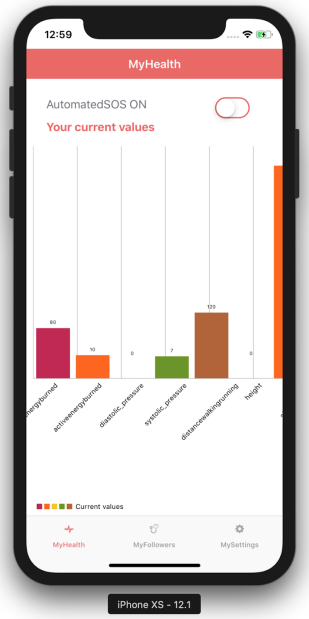
Figure 6: MyFollower tab

MyHealth controllers

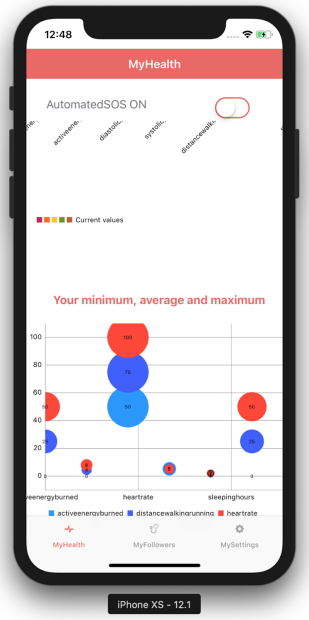
- **MyHealth.swift**: a subclass of UIViewController responsible for the behaviour of the MyHealth tab in the *Single User* interface. This class handles the user interaction with the *AutomatedSOS* button and with the bar and bubble charts displaying health data statistics inside the UIViewContiners. All charts are special UIView objects provided my the "Charts" framework. Their dataset can be refreshed by simply tapping onto them.
- **AverageBarChartViewController**: a subclass of UIViewController responsible for setting up and reloading the data sets for a specific bar chart. This is the chart with two input data sets to compare: the average data values of the user and of all others of his age. All values are requested to the application server with a D4HStatisticsRequest.
- **CurrentBarChartViewController**: a subclass of UIViewController responsible for setting up and reloading the data set of the bar chart containing the user's current data. No request to the application server is required as all current values can be quickly retained by the local storage.
- **BubbleChartViewController**: a subclass of UIViewController responsible for setting up and reloading the data set for the bubble chart. This charts shows for all types of data the minimum, average and maximum values of the current month.



(a) Average bar chart



(b) Current data bar chart



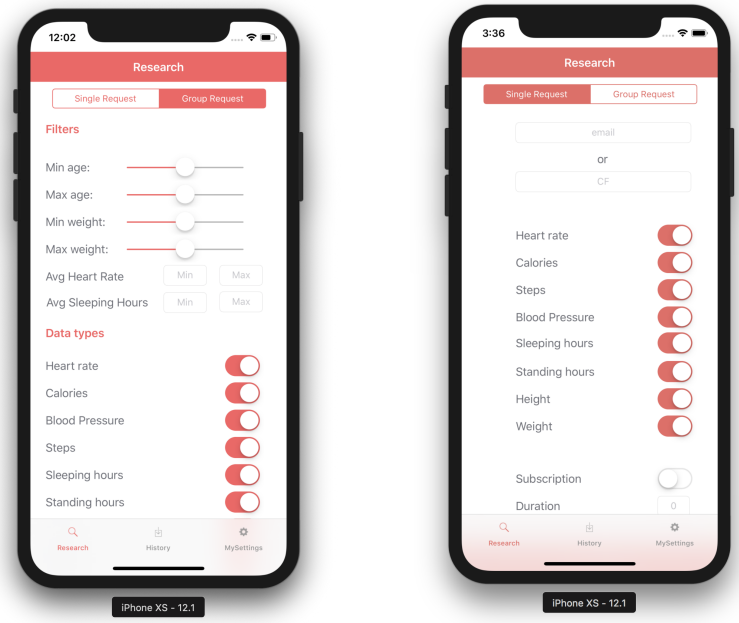
(c) Bubble chart

Figure 7: MyHealth tab screenshots

TP Controllers

Research controllers

- **SRequest.swift**: it manages the form used to define and perform a request to a *Single User*. Elements of this form are texfields to specify *Single User*'s FC or email and a sequence of toggles to specify the desired datatypes. At the bottom of the view a button allows to send the request to the backend.
- **GRequest.swift**: it manages the form used to define and perform *Group Requests*. This form is made of two sections. The first presents a number of sliders and textfields used to define filtering parameters for the request. The second section contain a sequence of toggles used to specify the requested datatypes. At the bottom of the view a button allows to send the request to the backend.
- **ResearchViewController.swift**: acts as a container for the views managed by SRequest and GRequest. These two views are alternatively shown to the user depending on whether he tapped one of the two sections of the segmented controller in the top part of the screen.



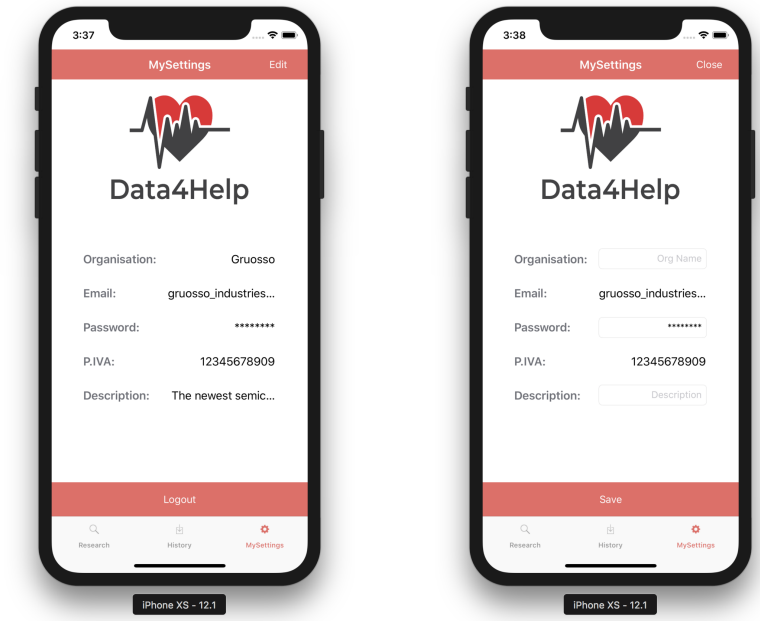
(a) Group Request view

(b) Single Request view

Figure 8: Research tab screenshots

Settings controllers

- **TPSettings.swift**: acts as a container for the views managed by TPViewSettings and TPEditSettings. These two views are alternatively shown to the user depending on whether he pressed an edit button in the top right corner of the screen. The view also present in the bottom part a button allowing to logout or to save the edited informations.
- **TPViewSettings.swift**: manage the view containing the info and settings related to the logged *Third Party*. They namely are full organisation name, email, password (obscured), PIVA and description.
- **TPEditSettings.swift**: present a layout almost identical to TPViewSettings but some field are presented as texfields in order to allow the editing of information.



(a) Settings view

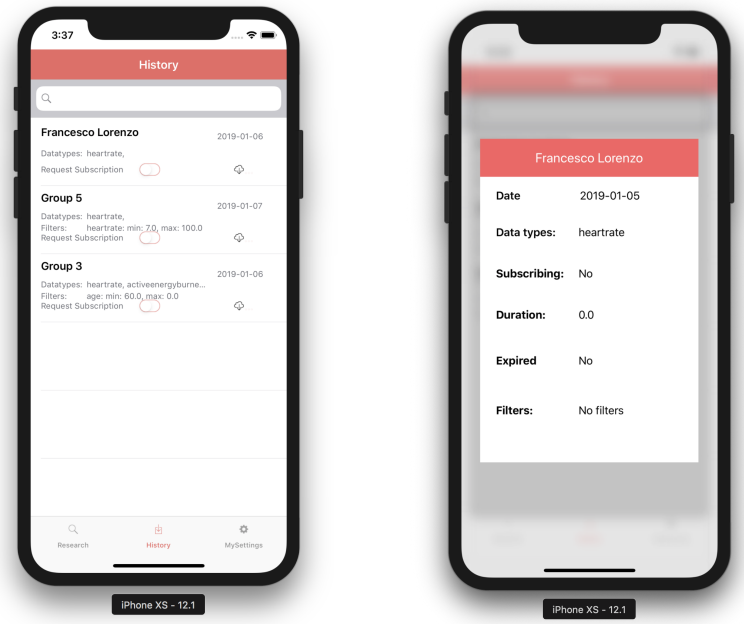
(b) Edit settings

Figure 9: *Third Party* Settings tab screenshots

MyHistory controllers

- **TPRequestsController.swift**: it manages the UITableView containing all the sent requests. It also provides a search bar to easily filter requests based on the receiver name in case of *Single Request* or group id in case of *Group Request*.

- **RequestCellDelegate.swift**: a protocol allowing TPRequestCell and TPGroupRequest classes to call some methods from TPRequestsController.
- **PopUpRequestViewController.swift**: a subclass of UIViewController responsible for displaying a popup containing all information related to a specific request. The popup is presented after applying *3D Touch* to a request, both single and group, in the History tab of *Third Parties*.



(a) MyHistory view (b) PopUp view

Figure 10: MyHistory tab screenshots

Health Controllers

This group gathers all controllers responsible for interacting with the Healthkit framework and for enabling *AutomatedSOS*.

- The **DataManager** class is a Singleton controller class responsible for interacting with HealthKit. This class is at the core of *Data4Help* as it is essential to enable features such as Health Monitoring and *AutomatedSOS*. A unique Healthstore object is kept in order to retrieve all health data from the user's device. Once a *Single User* has logged into his account, after having requested his authorization, the DataManager enables background delivery for data from the Health app and imports it as soon as it is

produced. All newly imported data is stored into the local CoreData Storage in order to be immediately retrieved when required and to be later sent to the server. As soon as a new health sample is imported it is handled by an appropriate function according to its type: Quantity, Category and Correlation samples. The table below defines all imported types of data and their measurement unit.

Health data	**Type**	**Unit**
Heart Rate	HKQuantitySample	count/min
Blood pressure	HKCorrelationSample	mmHg
Steps	HKQuantitySample	count
Standing Hours	HKCorrelationSample	hours
Distance Walking Running	HKQuantitySample	miles
Active Energy Burned	HKQuantitySample	kiloCalories
Weight	HKQuantitySample	pounds
Height	HKQuantitySample	feet

The controller also to retrieves from the healthstore the user's biological sex, stored as an HKCharacteristicType, and sends it to the application server together with the information manually input by the user. By setting an appropriate timer, the DataManager periodically sends to the server application a message containing all newly imported data. The period is set to one hour in order not to overload the server with requests for every new import. If a user decides to enable *AutomatedSOS*, this new information is stored by the DataManager. As soon as new data is produced, it is immediately passed to the *AutomatedSOS* controller class.

- *AutomatedSOS* is a controller class responsible for analyzing all imported health data and deciding whether a user is in a critical health condition. If this is the case, then it prompts the user to call an ambulance. User confirmation is required to establish the call with the emergency services. This class becomes active only when the user decides to activate the *AutomatedSOS* functionality. Enabling this feature can be done also through a quick action by using *3D touch* on the app's icon.

Figure 11: History tab

The parameters evaluated by the class to establish whether a user is in a critical health condition follow the New York Heart Association Guidelines. Specifically, it analyses values of heart rate and blood pressure that define either a Stage 2 Hypertension (with systolic and diastolic pressures respectively above 140 and 90) or a hypotension (with systolic and diastolic pressures respectively below 90 and 60). This class can be extended to support other analysis to be performed in order to define a critical health condition. As mentioned in the Design Document, all the logic of *AutomatedSOS* is located within the Client application in order to evaluate critical conditions immediately and without any delay due to the network.

4.3.3 Models

Data4Help Model

Data4Help Model is a folder containing all the classes used at runtime to represent the state of the application. It presents data structures to keep track of *Single* and *Group Requests* parameters and content, but also classes for observations and statistics used to draw graphs. Finally Health Parameters and bounds used to perform requests. Most of this objects are thought to be instantiated using the content of network calls responses and thereby present initialisers that accept JSON data.

Requests and Responses
Under the Model folder are stored all the Request and Response objects used by the NetworkManager. Both Requests and Responses inherit respectively from D4HRequest and D4HResponse allowing the NetworkManager to handle the addition of new requests and responses without changing the code that perform the network calls. Requests and Responses are also organised in a serie of subfolders divided by argument: Request, Data, Settings and Authentication. Regarding Requests they all inherit the method "getParams", called by the Network Manager to retrieve the JSON encoded parameters to be added into the HTTP request body. Responses on the other hand have an initialiser that allows to build them from the JSON data contained in the HTTP response received by the network manager. The Request and Responses folder presents an additional file called D4HEndopints that contains a data structure that enumerates all the different endpoints reached by the network calls.

CoreData Model
CoreData is a persistence framework that allows to create a relational entity-attribute storage serialized in an SQLite store. In this project it is used as a local storage to store health data imported from Healthkit and the status of *AutomatedSOS*. There are three entities to store all the necessary information: "Data", "BiologicalSex" and "*AutomatedSOS*". The first entity is used to store all data imported from Healthkit. It is gathered in chunks as soon as it is produced and periodically sent to the server in order not to overload it with requests. Data entities have three attributes: "type", "value" and "timestamp". All stored values correspond to Healthkit objects of time HKSample. The biological sex of a user is store in healthkit as an HKCharacteristicType object being an immutable feature of the user. It is stored in a separate entity and sent to the application server when requested during registration. Finally also the state of *AutomatedSOS* is stored permanently in the local storage. All of *AutomatedSOS*'s logic is developed in the client application so it is essential to store its state and reload it after the app's termination.

Properties.swift
This file contains a struct used to store and manage user's *accessToken*, password and email. It allows to easily include the *accessToken* in an http header or eventually revoke all credentials in case of logout. It also contains a function called during the logout routine that allows to perform a segue back to the login view. User's password and email are persistently stored using UserDefaults in order to perform an automatic login in case the user closed the app without explicitely logging out.

4.3.4 Utilities

Network Manager.swift
Network manager is a support class that defines a series of handy and general purpose methods that use AlamoFire to perform POST and GET request. Net-

work Manager implement a singleton pattern ao that a shared instance of the class is always available everywhere in the application. This choice also fits well with the stateless nature of this component. As already mentioned the Network manager allows to perform network calls from everywhere in the client application and it offer a very generic and expandible structure that easily support the introduction of new requests and endpoints.

Storage Manager
The Storage Manager is the controller classes devoted to interacting with *CoreData*, a persistence framework that allows to create a relational entity-attribute storage serialized in an SQLite store. A local storage in the client application is needed to collect newly imported data from Healthkit used to be displayed to the user or gathered and sent in chunks to the server. The information stored in *CoreData* is essentially all health data related to the user and the activation state of *AutomatedSOS*. While the imported health data is cached only for a certain period of time, the information about *AutomatedSOS* is stored permanently in order to be retrieved after the app's termination. This controller is a singleton class and is referenced throughout the whole project whenever new data needs to be stored permanently or retrieved.

App Delegate
The Application Delegate is the class responsible for handling all events related to the application's lifecycle such as when it is launched, when it works in background and foreground and when it terminates. When the application is launched for the first time it performs setup operations such as setting up *CoreData*, requesting the user's authorization to access Health, initializing *AutomatedSOS* and setting up background fetching of health data. It is also responsible for handling quick actions that are set up dynamically during the app's execution. When the app is launched through a quick action, the App delegate is responsible for calling the appropriate handler. The Application Delegate follows the Delegation pattern. The core purpose of this pattern is to allow an object to communicate back to its owner in a decoupled way.

5 Testing

Testing has been done following the specifications present in the Design Document.

For the Backend, two frameworks were used: Jest and Supertest. Jest is a widespread JavaScript testing library with mockup functionalities, high-level abstractions, various types of assertions and offers clean code structure. Supertest is a library used mainly for Integration testing, and provides an easy way to test Express endpoints and make HTTP assertions. Testing code is located in the tests folder, recreating the source code structure inside it.

5.1 Unit Testing

5.1.1 Backend

Unit Testing in the backend is limited, as there are few modules that have specific functions not relying on external components, and implementing meaningful algorithms. The most suitable and critical module that needed unit testing was schemas.js, defining the schemas of requests, and was thoroughly tested. To properly perform Unit Testing, the module was directly tested, not relying on its function as a middleware. Same properties (like password and email) were tested for one schema, as nothing changes for the others. The test cases performed are the following:

Schema	**Test Description**	**Desired Output**
Single User registration	– Object respecting the schema	Error is null
Single User registration	– Missing password – Password length isn't 8 characters – Password not being a string	Validation error
Single User registration	– Missing email – Email too long – Wrong regex – Email not being a string	Validation error

Single User registration	– Missing FC – FC length isn't 16 characters – FC not being a string	Validation error
Single User registration	– Missing full name – Full name too long – Full name not being a string	Validation error
Single User registration	– Missing birthdate – Birthdate not a date – Birthdate not respecting ISO specifications Validation error	Validation error
Single User registration	– Missing sex – Sex not “M” or “F”	Validation error
Third Party registration	– Object respecting the schema	Error is null
Third Party registration	– Missing P.IVA – P.IVA length isn't 11 characters – P.IVA is not a string	Validation error
Third Party registration	– Missing company name – Company name longer than 20 characters – Company name not being a string	Validation error

Third Party registration	– Company description not being a string	Validation error
Login	– Object respecting the schema	Error is null
Single User settings	– Object respecting the schema	Error is null
Third Party settings	– Object respecting the schema	Error is null
Import data	– Object respecting the schema	Error is null
Import data	– Data type array missing – Invalid data type in the array – Length not matching the value array	Validation Error
Import data	– Value array missing – Invalid value in the array – Length not matching the data type array	Validation Error
Import data	– Timestamp array missing – Invalid value in the array – Length not matching the value array	Validation Error

Statistics	– Object respecting the schema	Error is null
Send a single request	– Object respecting the schema	Error is null
Send a single request	– Subscription is not a boolean	Validation error
Send a single request	– Duration is not an integer – Duration is present if subscribing is false	Validation error
Send a group request	– Object respecting the schema	Error is null
Send a group request	– Missing parameters array – Invalid value in the array – Length not matching the bounds array	Validation error
Send a group request	– Missing parameters array – Invalid value in the array for lowerbound – Length not matching the bounds array	Validation error
Accept a request	– Object respecting the schema	Error is null

Accept a request	– Missing ReqID – ReqID not being an integer	Validation error
Accept a request	– Missing choice – Choice not being a boolean	Validation error
Download a request	– Object respecting the schema	Error is null

Table 12: Unit testing test cases

5.2 Integration Testing

5.2.1 Backend

As previously explained, integration testing has been done using Supertest as framework. The core of the backend, the Express app, has been tested endpoint by endpoint from when a request is received up to when the response is sent (and received by Supertest), together with all its dependencies. In particular, in order to maintain production and development separate, a local PostgreSQL database was created, and all its data is completely erased after every test suite, so tests don't interfere with each other. All integration tests do not check for syntactic mistakes, as they were already tested through Unit Testing, but rather for semantic errors. The test cases performed are the following:

Endpoint	**Test Description**	**Desired output**
/auth/reg/single	Valid request is provided	HTTP 200: Database contains the new user, activation mail is sent
/auth/reg/single	Register a user who's already registered	HTTP 403: Account is not created
/auth/reg/tp	Valid request is provided	HTTP 200: Database contains the new user, activation mail is sent
/auth/reg/tp	Register a user who's already registered	HTTP 403: Account is not created

/auth/activ	Activate a registered account providing a correct token	HTTP 200: Account is activated in the database
/auth/activ	Activate an already activated account	HTTP 403: Account is not activated
/auth/activ	Activate a non existing account	HTTP 401: Account is not activated, can't access the resource
/auth/login	Login using correct credentials for an existing account	HTTP 200: Login successful, authToken in the response
/auth/login	Login with an invalid email	HTTP 401: Login failed, can't access the resource
/auth/login	Login in a non activated account	HTTP 401: Login failed, can't access the resource
/auth/login	Login with wrong password	HTTP 401: Login failed, can't access the resource
/data/upload	Upload data of a logged user	HTTP 200: Data is now present in the database
/data/upload	Upload data of a non logged user	HTTP 401: Can't access the resource
/data/upload	Upload data which in part was already present	HTTP 200: Only new data is imported
/data/stats/avg	Retrieve statistics of a logged user with data of other users already present in the db	HTTP 200: Statistics retrieved in the response, taking into account also other users
/req/tp/sendSingle	Send a single request by a logged *Third Party*	HTTP 200: Request is now present in the database
/req/tp/sendSingle	Send a request to a non existing user	HTTP 403: Can't find the user, request is not inserted in the db
/req/tp/sendSingle	Send a request to an user who has a pending request already	HTTP 403: Can't send the request, db is not updated

/req/tp/sendGroup	Send a group request by a logged *Third Party*	HTTP 200: Request is now present in the database
/req/single/choice	Accept and refuse an existing pending request	HTTP 200: Request is updated in the database
/req/single/choice	Accept a non pending request	HTTP 403: Can't accept a non pending request
/req/single/choice	Accept a non existing request	HTTP 403: Can't accept a non existing request
/req/single/list	Retrieve list of requests of a *Single User*	HTTP 200: Response contains all the requests to the user
/req/tp/list	Retrieve list of requests of a *Third Party*	HTTP 200: Response contains all the requests (single and group) of the user
/sub/endSingle	Terminate an ongoing subscription (by both *Single User*s and *Third Parties*)	HTTP 200: Subscribing is now false in the database, subscription is ended
/sub/endSingle	Terminate an expired subscription	HTTP 403: Subscription was already expired
/sub/endSingle	Terminate a subscription of a non accepted request	HTTP 403: Can't terminate the subscription
/sub/endGroup	Terminate an ongoing subscription to a group request by the sender *Third Party*	HTTP 200: Subscribing is now false in the database, subscription is ended
/sub/endGroup	Terminate an expired subscription	HTTP 403: Subscription was already expired
/sub/endGroup	Terminate a subscription of a non accepted request	HTTP 403: Can't terminate the subscription
/tp/downloadSingle	Download data of an accepted single request	HTTP 200: Response contains data of the target user
/tp/downloadSingle	Download data of a non accepted request	HTTP 403: Can't download the data
/tp/downloadSingle	Download data of a non existing request	HTTP 403: Request does not exist

/tp/downloadGroup	Download data of a group request	HTTP 200: Response contains anonymised data requested
/tp/downloadGroup	Download data of a non existing request	HTTP 403: Request does not exist
/tp/downloadGroup	Download data of a request that doesn't match the 1000 users constraint	HTTP 403: Not enough users match the Search Parameters
/settings/single/info	Update settings of a logged users	HTTP 200: Settings are updated in the database
/settings/single/info	Retrieve settings of a logged users	HTTP 200: Response contains the settings
/settings/single/info	Update settings of a non logged users	HTTP 401: Can't access the resource
/settings/single/info	Update settings of another user	HTTP 401: Can't access the resource
/settings/tp/info	Update settings of a logged users	HTTP 200: Settings are updated in the database
/settings/tp/info	Retrieve settings of a logged users	HTTP 200: Response contains the settings
/settings/tp/info	Update settings of a non logged users	HTTP 401: Can't access the resource
/settings/tp/info	Update settings of another user	HTTP 401: Can't access the resource

Table 13: Integration Testing Test cases

5.3 System Testing

System testing is essential to test the system's compliance with the specified requirements. The test cases described in the following table allow us to test the interaction between Data4Help's frontend and backend, verify the correctness of all services and receive feedback through the user interface.

Schema	**Test Description**	**Desired output**

Register a single user	– Insert valid information	– Go back to the Login View – Receive an activation link via email
Register a single user	– Send an empty request	– Receive an error alert
Register a single user	– Insert invalid information (birthdate, password or CF malformed)	– Receive an error alert
Register a third party	– Insert valid information	– Go back to the Login View – Receive an activation link via email
Register a third party	– Send an empty request	– Receive an error alert
Register a third party	– Insert invalid information (P.IVA, or password malformatted)	– Receive an error alert
Login	– Insert valid credentials	– Enter logged-in view
Login	– Send an empty request	– Receive an error alert

Login	– Insert invalid credentials	– Receive an error alert
Login	– Insert credentials of an account that has not been activated	– Receive an error alert
Create a Single Request	– Insert valid email or CF and select some datatypes	– Receive an alert confirming the request has been sent
Create a Single Request	– Insert invalid email or CF	– Receive an error alert
Create a Single Request	– Insert valid email or CF but empty list of datatypes	– Receive an error alert
Create a Group Request	– Select filters and datatypes	– Receive an alert confirming the request has been sent
Create a Group Request	– Insert empty list of datatypes	– Receive an error alert
Refresh History tab	– Open History tab to check current requests – Create a new single or group request – Open History tab – Pull to refresh	– Receive an updated list of requests with the newly created one

Edit Third Party Settings	– Edit Settings with valid information and press save	– Edits are saved correctly and editing mode is closed
Edit Third Party Settings	– Edit settings with invalid information	– Receive an error alert
Update MyHealth View	– Tap charts for update after adding new data to Health	– See updated Charts
Refresh MyFollowers tab	– Open MyFollowers tab to check current requests – Create a new single request from a third party account – Pull to refresh	– Receive an updated list of requests with the newly created one with status pending
Accept single request	– Send a single request to the user from a third party – Select accept button on a pending single request	– The request will appear in the Accepted Requests section of the MyFollowers tab – The request will appear in the Accepted Requests section of the History tab of the third party sending it

Refuse single request	– Send a single request to the user from a third party account – Select refuse button on a pending single request	– The request will appear in the Refused Requests section of the MyFollowers tab – The request will appear in the Refused Requests section of the History tab of the third party sending it
Activate AutomatedSOS	– Enable AutomatedSOS switch in MyHealth – Add low heart rate sample in Health for testing	– Receive an alert to call an ambulance as soon as the sample is added
Activate AutomatedSOS	– Disable AutomatedSOS switch – Add low heart rate sample in Health for testing	– No alert to call an ambulance is received when the sample is added
Enable AutomatedSOS	– Close app – Press app with 3DTouch and select "Enable AutomatedSOS"	– App opens and AutomatedSOS switch is on
Edit Single User Settings	– Edit Settings with valid information and press save	– Edits are saved correctly and editing mode is closed

Edit Single User Settings	– Edit settings with invalid information	– Receive an error alert

Table 14: System Testing Test Cases

6 Installation

6.1 Preliminary Operations

The following Installation procedure is valid for MacOS. Please contact us for troubleshooting. The Backend can be setup for other OS, but some steps might be different, please refer to online documentation if needed. The Frontend can only be setup for MacOS, as XCode is available only for said environment.

First of all you need to clone our GitHub repository.

1. Make sure you have git client installed.
2. Open a shell and change directory to where you want to install our repository.
3. Type “git clone https://github.com/iPhra/LorenzoMolteniNegri.git”

6.2 Backend

To access the Backend source code, and modify it, you need Node.js and an IDE for JavaScript.

1. Download the latest LTS release of Node.js from https://nodejs.org/en/download/.
2. Download a JavaScript IDE: we recommend WebStorm as it's the one used for development, but other options like Visual Studio Code are fine.
3. Open a shell and head to /LorenzoMolteniNegri/Backend/NodeJS.
4. Type “npm install” to install all dependencies.
5. You can now run the Backend server locally by either:
    - Opening a shell in /LorenzoMolteniNegri/Backend/NodeJS and typing “npm start” (recommended).
    - Opening your IDE and creating a new Project selecting /LorenzoMolteniNegri/Backend/NodeJS as root, then creating a script to run it. Please refer to the documentation provided for your IDE, make sure to select bin/www.js as the executable to run.

To run all tests you need to install a local PostgreSQL database.
Follow the steps mentioned in this guide https://www.codementor.io/engineerapart/getting-started-with-postgresql-on-mac-osx-are8jcopb for what concerns installing Postgres through Homebrew, creating a user named ”iphra” with empty password and a new database called ”data4help”.
Then, you have to recreate the DB schema provided in this documentation using a SQL client of your choice. For SQLWorkbenchJ, to access and modify the database, follow these steps:

1. In the Select Connection window, choose the Driver for PostgreSQL among those available.
2. Use this url: “jdbc:postgresql://localhost/data4help”.
3. Username should be ”iphra”.
4. Leave all other options default, and connect to the database.
5. You can now recreate our schema by using Statement windows to type SQL commands.

When the database is ready, open Terminal and head to /LorenzoMolteniNegri/Backend/NodeJS, you can now run all tests by typing “npm test”.

6.3 Frontend

To access the frontend's source code you must have XCode installed. We recommend having the latest version of the software installed (10.1) and the latest update of your OS. We cannot guarantee correct functioning with previous versions. You must now follow these steps:

1. Open a shell and reach the directory of the XCode project located in /LorenzoMolteniNegri/iOS/Data4Help/
2. Delete the files Data4Help.xcworkspace, Podfile.lock and the directory Pods
3. Type *pod install* to reinstall pods and generate a new workspace
4. In XCode open the newly generated workspace, located in /LorenzoMolteniNegri/iOS/Data4Help/
5. In the project's General tab insert valid Developer Account credentials in the “Team” field (E.g. the ones of your AppleID) and a bundleId different from the actual one.
6. To build the project you must select a destination under $Product \rightarrow Destination$. You can either:
    a) Select an iOS Simulator. We recommend using an iPhone XS simulator to take advantage of the wide screen and latest capabilities. If you use a simulator you must run the server locally. In the $Utilities \rightarrow NetworkManager.swift$ file, be sure to have GCP.D4HAPIbaseURL as prefix of your url in the function getD4HUrlWithKey of GCP.
    b) Select your personal device after having connected it to the USB port. Please be sure to be running iOS 8.0 or a later version. We recommend running the latest version of iOS. If you use your personal device you must connect to the deployed server. In the $Utilities \rightarrow NetworkManager.swift$ file, be sure to have GCP.D4HAPIbaseURLdeployed as prefix of your url in the function getD4HUrlWithKey of GCP.

7. Run the application. If the destination is your personal device an error will appear after having installed the app for the first time. The error appears because the Developer App certificate for your account must be trusted on your device. To remove it proceed as follows:

    a) In your device go to Settings.

    b) In Settings navigate to $General \rightarrow Profiles\&Management$ and select your Developer App certificate to trust it.

8. Open the app.

7 Future developments

Some future developments have already been planned, but are not included in this first release due to either lack of time or financial limitations. In particular, the motivations are:

- **Nginx proxy and server replication:** due to the limitations of the AWS free plan, it was not possible to replicate the server on multiple instances. As a consequence, the nginx proxy wasn't setup to handle authentication and load-balancing, because it's on the same instance running the only server. Nonetheless, the setup would be quite easy, as in a real world scenario the instance running the proxy would be the only one to have a Node.js application with the /auth/login endpoint and Joi validation middleware (see Section 4.1), so after performing authentication it would forward the request to available instances on the same network.
- **HTTPS protocol:** HTTPS was not used to secure communications between nodes due to difficulties in obtaining a free certificate, so standard HTTP is instead used. In future releases, the certificate would be bought and consequently used to setup a safe channel.
- **Missing datatypes:** some datatypes, like position, are not available to be imported because they would require the integration of different services other than HealthKit. In particular, since *Track4Run* has not been implemented, the position of a user is not as important as health features like heartrate and stepcounts, so we decided to focus on those instead.
- **TTS service:** implementing a TTS service able to be used during phone calls requires subscription services, so we instead opted for a simple call alert, which is free.
- **Push notification** *System***:** implementing push notifications for subscriptions was not possible as it would require Apple developer membership, which is not free. Nonetheless, users are able to always download the latest data available until the subscription ends, but this means that the interaction is asynchronous, and is different from the Runtime View depicted in the DD, as we had to introduce two new endpoints.
- **UI overhaul:** the current UI is experimental and will be subject to tweaks and improvements in future releases.

8 Efforts

We did not precisely track the hourly contribution of each member, but we estimate the workload to be up to 80 hours per person.
The implementation began on the 25th November 2018 and conluded on the 13th Jaunary 2019, including the ITD document. Below is a screenshot taken from GitHub that reflects our contribution per person, though keep in mind that the lines of code displayed are skewed because they also include the Documentation.

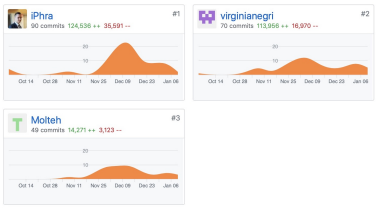


Figure 12: GitHub contribution

The Implementation and Testing was divided as follows:

- **Francesco Lorenzo:** contributed to the deployment of the RDS database, and its management afterwards. Implemented the Backend side, dealing with the WebStorm project, as well as performing Unit Testing and Integration Testing for it. After development he dealt with Server deployment.
- **Luca Molteni:** contributed mainly on the development of the Frontend. Responsible for iOS application network layer, requests list tabs and settings tabs.
- **Virginia Negri:** contributed to the development of the Frontend. Responsible for implementing the main views, the health monitoring features including AutomatedSOS and for managing the client CoreData storage.