



# POLITECNICO MILANO 1863

*Data4Help*

Lorenzo, Molteni, Negri

*DD* - Design Document

December 10, 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Purpose . . . . .	2
1.2	Scope . . . . .	2
1.3	Definitions, Acronyms, Abbreviations . . . . .	2
1.3.1	Definitions . . . . .	2
1.3.2	Acronyms . . . . .	3
1.3.3	Abbreviations . . . . .	3
1.4	Revision history . . . . .	3
1.5	Reference Documents . . . . .	3
1.6	Document Structure . . . . .	4
<b>2</b>	<b>Architectural Design</b>	<b>5</b>
2.1	Overview . . . . .	5
2.2	Component view . . . . .	6
2.3	Deployment view . . . . .	11
2.4	Runtime view . . . . .	13
2.5	Component interfaces . . . . .	27
2.6	Selected architectural styles and patterns . . . . .	43
2.7	Other design decisions . . . . .	45
<b>3</b>	<b>User Interface Design</b>	<b>47</b>
<b>4</b>	<b>Requirements Traceability</b>	<b>49</b>
<b>5</b>	<b>Implementation, Integration and Test Plan</b>	<b>54</b>
5.1	Overview . . . . .	54
5.2	Implementation . . . . .	54
5.3	Unit testing . . . . .	55
5.4	Integration testing . . . . .	56
<b>6</b>	<b>Effort Spent</b>	<b>59</b>

# 1 Introduction

## 1.1 Purpose

This document deals with the architectural description of *Data4Help* and its services and is the result of successive refinements made together with Stakeholders to satisfy their needs. On one hand, developers will find an in-depth description of the components of the *System*, the interactions between them, their deployment and how they will be implemented. On the other hand, the QA team will find plans for both unit testing and integration testing. Finally, requirements expressed in the RASD document will be dealt with here, describing how the components presented can satisfy them.

## 1.2 Scope

As explained in the RASD document, *Data4Help* is a service whose main purpose is to allow users to keep track of their health status, both on their own and by allowing experts to access their data. It will be available as a mobile application in the Italian app stores only. Private users can import data from their tracking devices and access it whenever they need, making it possible to analyze the evolution of specific health parameters in time. Companies can access data of private users by sending them requests or retrieve anonymized data from groups of users who meet certain conditions. All data requested by these Third Parties will be used only for medical purposes.

*AutomatedSOS* is the first service offered on top of *Data4Help*, and ensures that an ambulance is called whenever the health parameters of a user fall below a critical threshold, based on his age. The ambulance is provided with the last known position of the user.

*Track4Run* is the second service that uses *Data4Help* to allow users to create runs, take part in them and spectate their participants.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

- **Client:** Software *System* on the user's device that requests services to the Server.
- **Server:** Software *System* that handles requests from different clients.
- **n-tier:** Distributed architecture composed of n hardware components, each containing one or many layers.
- **n-layer:** Distributed architecture composed of n software levels, each distributed on one or many tiers.

- **Design Pattern:** Reusable software solution to a commonly occurring problem within a given context of software design.
- **Driver:** a piece of software that simulates a component that depends on the module undergoing testing.
- **Stub:** a piece of software that simulates a components on which the module undergoing testing depends.

### 1.3.2 Acronyms

<b>MVCS</b>	Model-View-Controller-Store
<b>DBMS</b>	Data Base Management <i>System</i>
<b>REST</b>	REpresentational State Transfer
<b>API</b>	Application Programming Interface
<b>OS</b>	Operating <i>System</i>
<b>UI</b>	User Interface
<b>QA</b>	Quality Assurance
<b>HTTP</b>	HyperText Transfer Protocol
<b>TTS</b>	Text-to-Speech

Table 1: Acronyms

### 1.3.3 Abbreviations

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

## 1.4 Revision history

Version	Date	Changes
1.1	10/12/2018	First deadline draft
2.1	13/1/2018	Final deadline draft

Table 2: Revision History

## 1.5 Reference Documents

- IEEE: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5167255&isnumber=5167254>
- Nginx documentation: <https://nginx.org/en/docs/>
- Project Assignment

## 1.6 Document Structure

The remaining sections of the DD are organized as follow:

- **Architectural Design:** details the *System*'s architecture by defining the main components and the relationships between them as well as specifying the hardware needed for the *System* deployment. The last part of this section will also focus on design choices and architectural styles, patterns and paradigms.
- **User Interface Design:** following what has already been included in the RASD this section defines the UX by means of view flow modeling.
- **Requirements Traceability:** shows the relations between the requirements from the RASD and the design choices of the DD and how they are satisfied by the latter.
- **Implementation, Integration and Test Plan:** provides a roadmapping of the implementation and integration process of all components and explains how the integration will be tested.
- **Effort spent:** describes how the work has been split between all members of the team and how long did the DD take to be completed.

## 2 Architectural Design

### 2.1 Overview

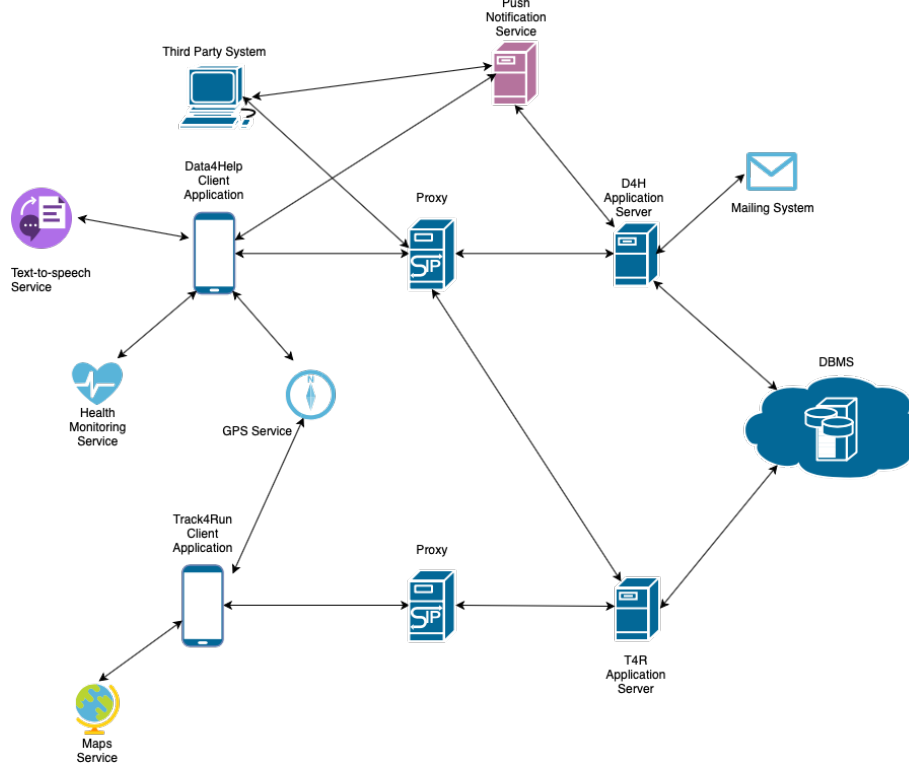


Figure 1: Overview of the *System*

The above image provides a general overview of the *System*'s architecture. We can immediately identify two horizontal layers and four vertical ones. The horizontal layers correspond to *Data4Help* and *Track4Run*'s architectures that while having their own Client Applications, Proxies and Application Servers, they also communicate internally to the *System*'s server network. Precisely, *Track4Run* accesses *Data4Help*'s functionalities and integrates them in its services by performing requests to its proxy. The vertical layers separate the Client side, including Client Applications and *Third Party Systems* accessing the applications' APIs, from the Server side, including the Application Servers, the Proxies and from the shared external Database, accessed through its cloud interface. Finally also all external services are included in the diagram, both those accessed by the Clients, such as Maps, GPS and Health Monitoring services, and those accessed by the Servers, such as a Mailing *System* and a Push Notification service.

## 2.2 Component view

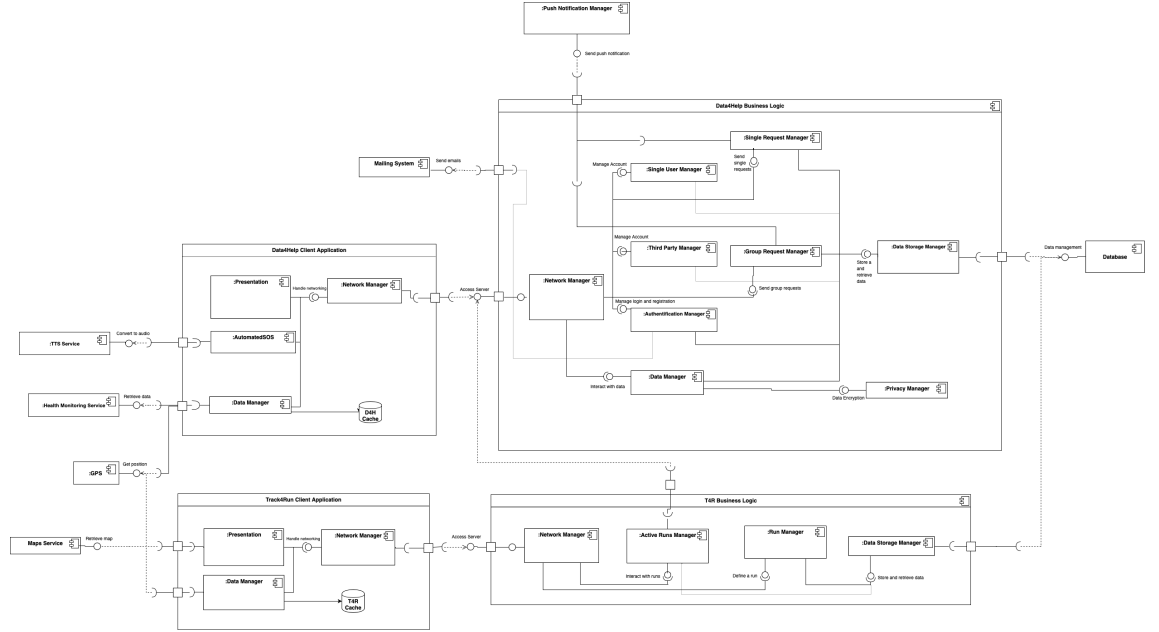


Figure 2: Component diagram

The purpose of this UML diagram is to show the internal architecture of the *System's* software. It is globally divided into five components: the *Data4Help* and *Track4Run* Client Applications, their Business Logic components and the shared external Database. Both business logic components communicate directly with a unique database. However *Track4Run's* business layer also accesses *Data4Help's* services through its APIs. These main layers are internally organized in modular components, each providing specific services. Components communicate with one another other by providing interfaces where they produce information that is used by the required interface of other components. In our diagram, this is represented internally to a component by assembly connectors ball-and-socket and externally by delegation connectors, required interfaces and provided interfaces. We will now focus on one component at a time and describe its functionalities and relationships with others.

### Data4Help Client Application

This component is located on the user's device. Its modules are the Network Manager, the Presentation Component, the Data Manager and *AutomatedSOS*. The first component is committed to dispatching all incoming and outgoing communications with the application server. The Presentation Component corresponds to the View in the MVCS Pattern. It requires data provided by the

business logic component to project it on the User Interface and does not add further functionalities. The Data Manager is the module that calls Health Monitoring Services APIs to import the user's health data. It gathers data of the required type and forwards it to the requesting application server. Finally, the last component contains the only portion of logic on the Client Application. It functions in background only if enabled by the user, otherwise remains inactive. Each time the client application imports new health data from the user's device, before forwarding it to the business component, *AutomatedSOS* performs a set of controls in order to decide whether an ambulance must be called.

### **Data4Help Business Logic**

This component is devoted to the implementation of *Data4Help*'s core logic. It is a stateless module that lies between the Client application and the central Database. It gathers all components that interact with each other to provide the *System*'s functionalities. We will now detail how each component is structured and what is its purpose.

- **Network Manager**

This component handles all incoming messages exchanged between Clients and Server. With respect to incoming the incoming flow, It dispatches all messages received from client applications to the specific handler components in the Business Logic. On the other hand, it gathers all outgoing messages and sends them to the correct Client Application through the unique port.

- **Authentication Manager**

The Authentication Manager exposes all methods related to the access to the platform. Specifically, it deals with user registration and login and to do so it continuously interacts with the database through the Data Storage Interface. It handles credentials verification and constraints and guarantees the creation of consistent accounts. Furthermore, in order to confirm to users they have successfully registered, the Authentication Manager sends them an email by accessing the Mailing *System*'s Interface.

- **Single User Manager**

The *Single User* manager handles all functionalities related to a *Single User* Account. Its services are directly called from users and performs them by interacting continuously with the Database through the Data Storage Interface. The provided services include settings update and change of password.

- **Third Party Manager**

The *Third Party* Manager deals with all services that concern *Third Party* Accounts. Specifically it exposes methods to update the user's settings and change its password. In order to provide these functionalities, this component continuously with the Database through the Data Storage Interface.



- **Data Manager**

This component is the main gateway between the Client Applications and the central Database. It handles the import of new data associated to a user's account and its retrieval, as well as organizing it and evaluating statistics before passing it back to the client who requested it.

- **Single Request Manager**

The Single Request Manager handles the creation and management of single requests performed by Third Parties and directed to *Single Users*. It provides both services to Third Parties, such as the creation of a new request and the retrieval of its response, and to *Single Users*, such as the possibility to accept or decline a request or to end its subscription. Both *Single Users* and *Third Parties* can also retrieve a list of their requests.

- **Group Request Manager**

This component is devoted to the management of group requests, created by *Third Party* users and addressed to the *System*. It offers services such as the creation of group requests, the enabling of subscriptions and the retrieval of past group requests. Retrieving the result is possible by querying the central Database through the Data Storage Interface.

- **Privacy Manager**

The Privacy Manager exposes methods to encrypt sensitive data. This operation is needed in order to store information about users in a secure way in order to reduce the effects of data leaks and to prevent Third Parties from accessing sensitive data with group requests.

- **Data Storage Manager**

The Data Storage Manager provides all methods to interact with the central Database such as data retrieval, storage and update.

### **Track4Run Client Application**

This component consists of the client side of *Track4Run*'s application. Its architecture is that of a thin client as it doesn't contain any logic of the *System*. It is composed of three modules: the Network Manager, that dispatches all incoming and outgoing messages, the Presentation component and the Data Manager, committed to import GPS data required to access the application's functionalities.

### **Track4Run Business Logic**

This component is devoted to the implementation of *Track4Run*'s core logic. While relying on *Data4Helps* main services such as authentication and account management and its request protocol, *Track4Run* adds further functionalities to its users thanks to the interaction of Run Manager and Active Runs Manager. Interaction with Client Applications and the central Database are handled respectively by the Network Manager and the Data Storage Manager. We will now detail each component's inner implementation:

- **Network Manager**

This component is devoted to handling all messages exchanged between business logic and client applications. Messages are received through a unique port and handled by the different business components.

- **Run Manager**

Run Manager exposes methods to Third Parties to create and manage new runs. Runs must be defined by adding an associated path, a duration and brief description. This component interacts with the central database as it needs to store all newly created runs.

- **Active Runs Manager**

This component grants access to users to all active runs and enables interaction with them such as the possibility to spectate or join them. Active Runs Manager is also linked to the central database in order to retrieve information about all active runs.

- **Data Storage Manager**

This component is the unique point of access to the Database in this tier. It handles all data management operations such as data storage, retrieval and update.

## External Interfaces

Some of the described components in our *System* are also dedicated to communicating with external services through specific interfaces. These communications are bilateral and essential to guarantee the application's functionalities. These components are both on the Client side and on the server side. We will now describe how each external service used by the *System* is accessed:

- **Database**

The component devoted to interacting with the central database on cloud is the Data Storage Manager. It is interposed between all other business logic modules and the external database. This component is shared by both *Track4Run* and *Data4Help*.

- **GPS**

On the Client side of the application, both *Data4Help* and *Track4Run* access data from the user's device's GPS through the Data Managers' interface. GPS information is essential for both health analysis and data requests in the first application and geolocalization while running in the second.

- **Health Monitoring System**

The Data Manager module is also responsible for the retrieval of health data from the user's device through its interface. This service too is essential for the core functioning of the applications.

- **Mailing System**

In order to inform users of having successfully created an account on *Data4Help*, the Authentication Manager requires access to the Mailing *System* to send a confirmation email.

- **Maps Service**

Access to a mapping service is essential for *Track4Run* to provide its core functionalities. Access is performed by the Client Application to enable the definition of a path for a run.

- **TTS Service**

This service is used when the *AutomatedSOS* component spots a critical situation for a Private User, and has to call an ambulance. The component prepares a text message containing the coordinates of the user and a brief help statement, and then passes this message to the TTS service to get the audio version, usable in the ambulance call.

## 2.3 Deployment view

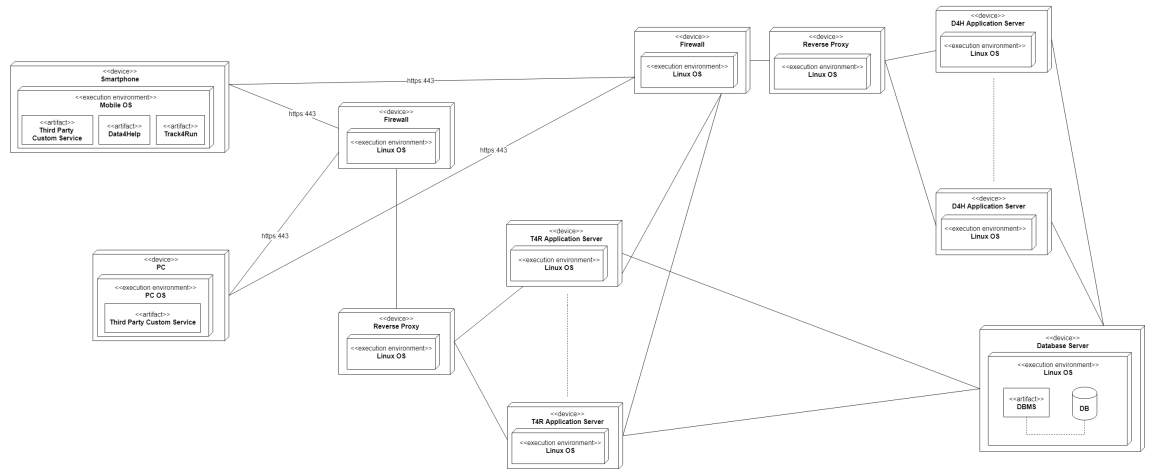


Figure 3: Deployment View

The *System* presents a multitier architecture in which the role of each node will be specified in the next paragraphs.

### Smartphone

This node acts as a client machine and could hosts *Data4Help* and *Track4Run* native applications.

### PC

This node works as well as a client and grants third parties access to *Data4Help* functionalities through the use of a web browser.

### Firewall

This component filters the access to the Reverse Proxy and is used to protect a trusted network from an untrusted network. A firewall provides protection from unauthorized requests or from various types of malicious attacks like DDoS.

### Application Servers

This level of the architecture encloses all the business logic of the *Systems*.

*Data4Help* and *Track4Run* Application Servers are fully replicated to balance the workload. *Track4Run* Application Servers communicate with *Data4Help* ones in order to make use of their APIs.

### **Reverse Proxy**

This node helps to achieve increased parallelism and scalability. It is responsible for the load balancing as it distributes requests between all Application Servers. The Reverse Proxy also increases security and anonymity by protecting the identity of our backend servers and acting as an additional shield against security attacks.

### **Database Server**

This machine is equipped with a relational DBMS and it is used to store and retrieve all data needed by the Application Servers.

## 2.4 Runtime view

### Proxy Runtime View

The following sequence diagram describes how the Reverse Proxy, implemented using Nginx, interacts with the Client and the Application Server. Whenever a Client forwards a request, the Reverse Proxy receives it on behalf of the Server, checks whether such request conforms to its defined schema and, in case the user has to be logged in, checks if the provided authToken is associated to an actual logged in user. This last check, in particular, is efficient to be done on the Proxy instead of each Application Server, because if a user logs in with a particular Server which then crashes, the user doesn't have to log in again, since the information is stored on the Proxy. One alternative could be to save the information on the DB, but that would require a query for every request sent by the Client, since the interaction is stateless. In all the diagrams following this one, the proxy will not be included for readability purposes, but keep in mind that it will analyze every request before forwarding it to the Server (but not checking the validity of the authToken when a user registers or spectates a run).

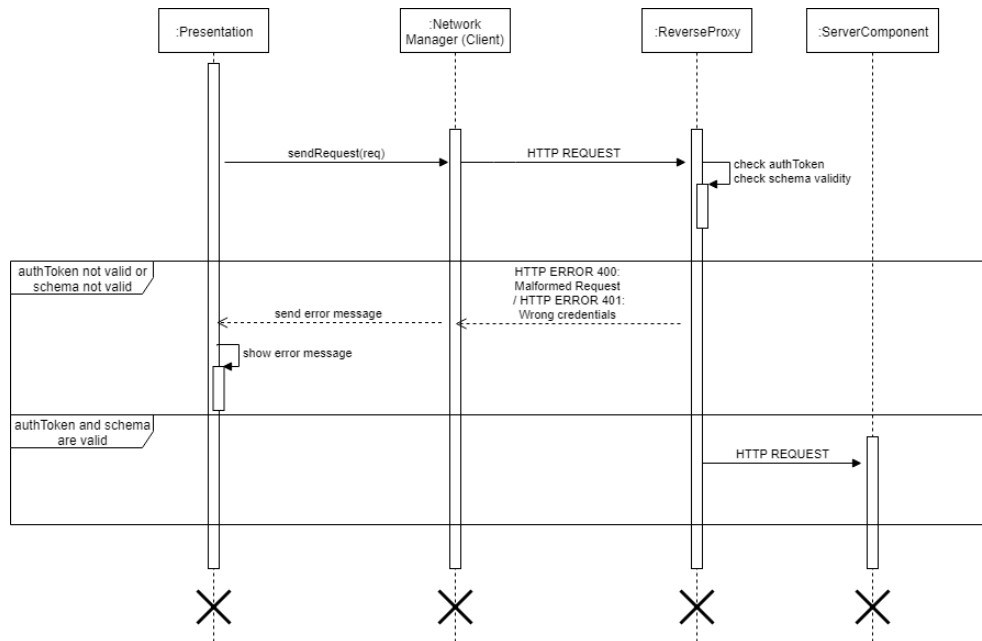


Figure 4: Proxy Runtime View

### Registration Runtime View

The following sequence diagram describes the order of events that occur whenever a user registers to *Data4Help*. The procedure is equivalent for both registering as a *Single User* and as a *Third Party*. The actors involved in this

scenario are the Presentation and Network Manager components on the Client side and the Network Manager, Authentication Manager, Data Storage Manager on the Server Side. External services that are invoked are the DBMS and the Mailing *System*. The sequence of events is guided by the user's HTTP registration request, filled with the user's email, password, FC, full name, birthdate and sex, all forming the registration form. The request is either malformed, such as having empty parameters, either structurally correct and defining the information to store in a new account in the database. However a request may still be invalid as it provides an already used email or FC, thus receiving an HTTP ERROR 403 response (Already Registered) or contains information violating the databases integrity constraints, thus receiving an HTTP ERROR 400 response (Malformed Request). Finally, in order to activate a newly created account, a user will click on the URL provided in the confirmation email and this will automatically send an HTTP request of account activation. If the account has already been activated the user will receive an HTTP ERROR 403 (Already Activated). In the case of wrong tokens or malformed requests the user will receive either an HTTP ERROR 401 (Wrong Token) or HTTP ERROR 400 response (Malformed Request).

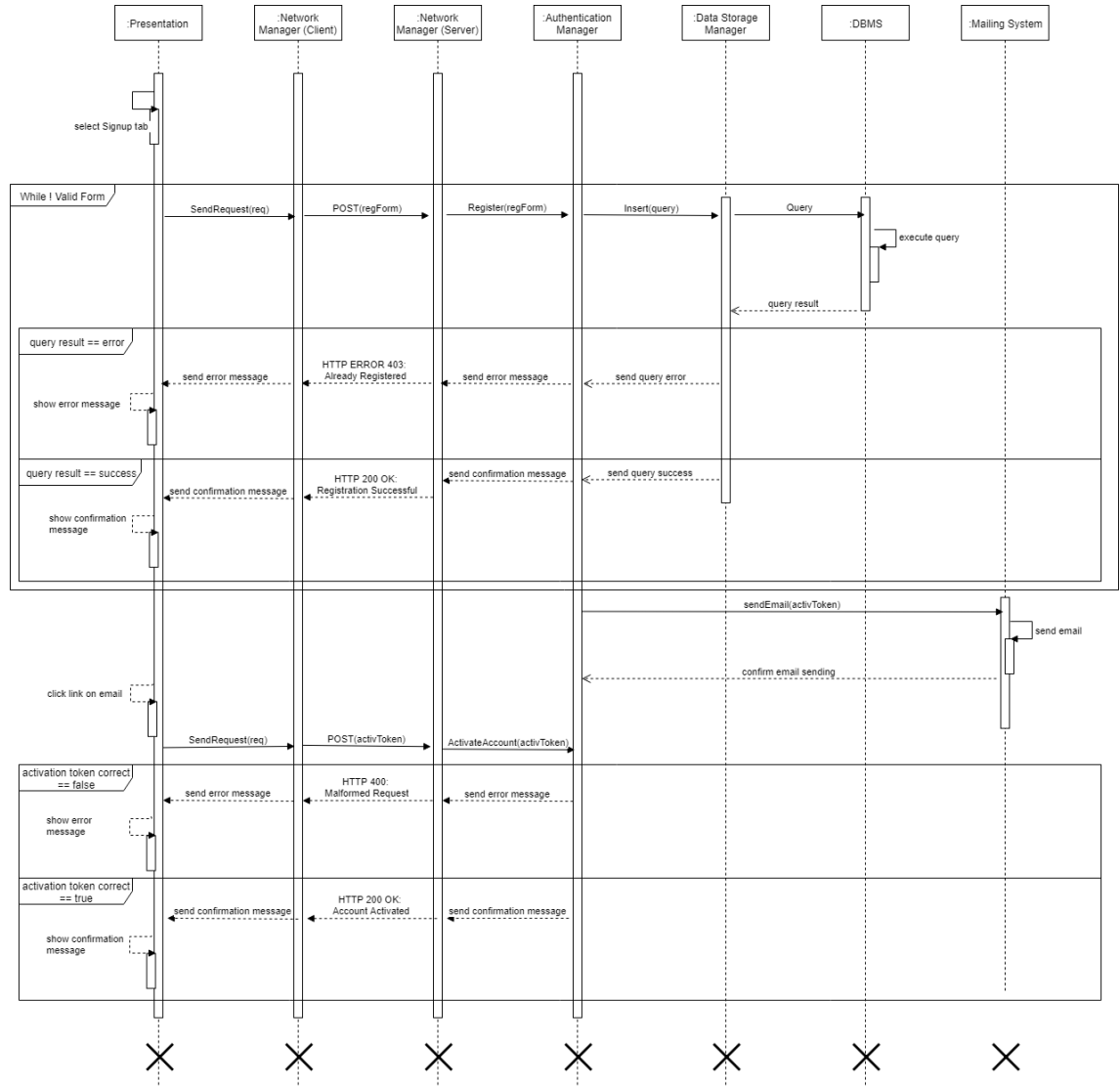


Figure 5: Registration Runtime View

### Login Runtime View

The following diagram represents the sequence of events that occur when a user tries to log into his account, being either *Single User* or *Third Party* accounts. The actors involved in the scenario are the Presentation and Network Manager on the Client side and the Network Manager, *Single User* Manager and Data Storage Manager on the Server side.

When a user logs into his account, an HTTP request is sent to the application server containing the user's credentials. If the request's format is invalid, such



as having empty parameters, the user immediately receives an HTTP ERROR 400 (Malformed Request) response. If instead the request is structurally correct, the *Single User* Manager checks the validity of the credentials by querying the Database. If the credentials are wrong or malformed an HTTP ERROR 401 (Wrong Credentials) or HTTP ERROR 400 (Malformed Request) response are sent.

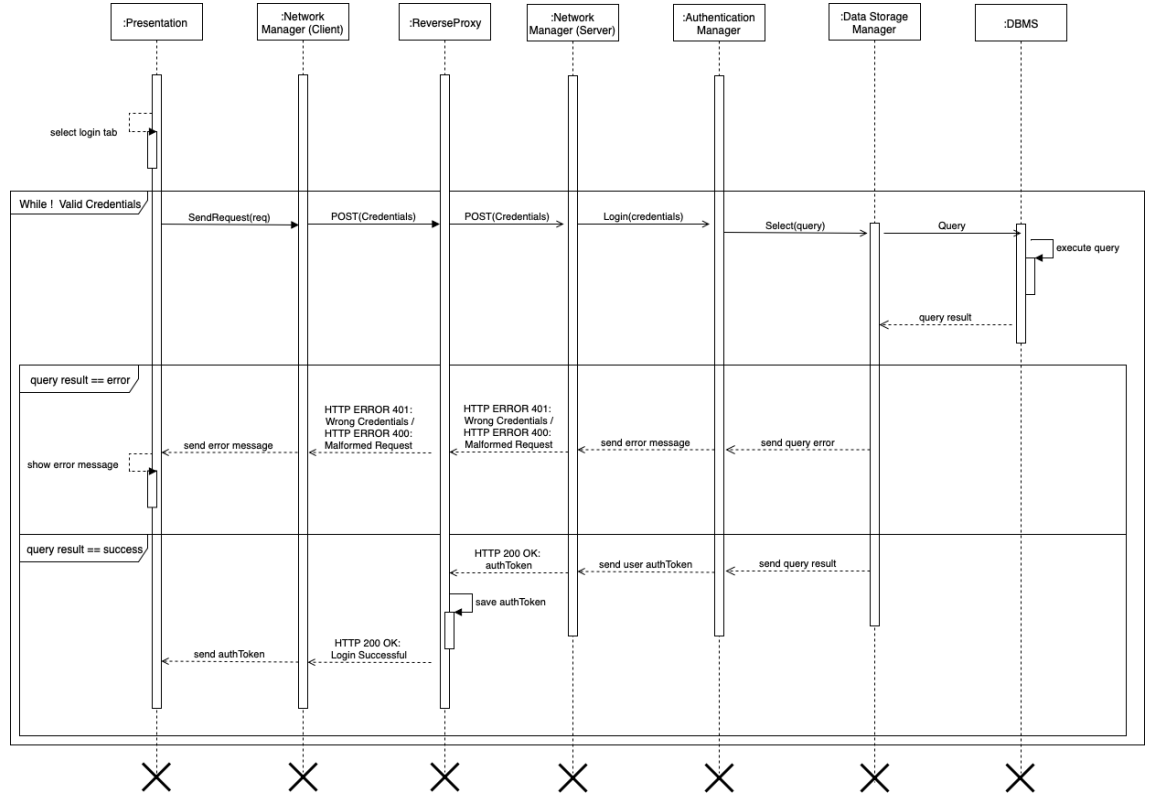


Figure 6: Login Runtime View

### Single Request Runtime View

The following diagram describes the events that occur after a Single Request is sent by a *Third Party* to a *Single User*. There are many components involved in this interaction. On the *Third Party* and *Single User* client side the Presentation and the Network Manager. Server side there are the Network Manager, Single Request Manager and Data Storage Manager. Finally the DBMS is invoked as an external service. When a *Third Party* select the research tab and enables all the parameters he is interested in, an HTTP request is sent to the Application server. A new Single Request is created by the Single Request Manager and it is stored into the database. After that a notification containing the data requested

by the *Third Party* is sent to the appropriate *Single User* who can either decide to accept or decline it. If the request is declined a notification is sent to the *Third Party* and the status of the request in the DB is updated. If instead the request gets accepted, the Application Server with the help of the *Single User* Manager retrieves the requested data and send them to the *Third Party*.

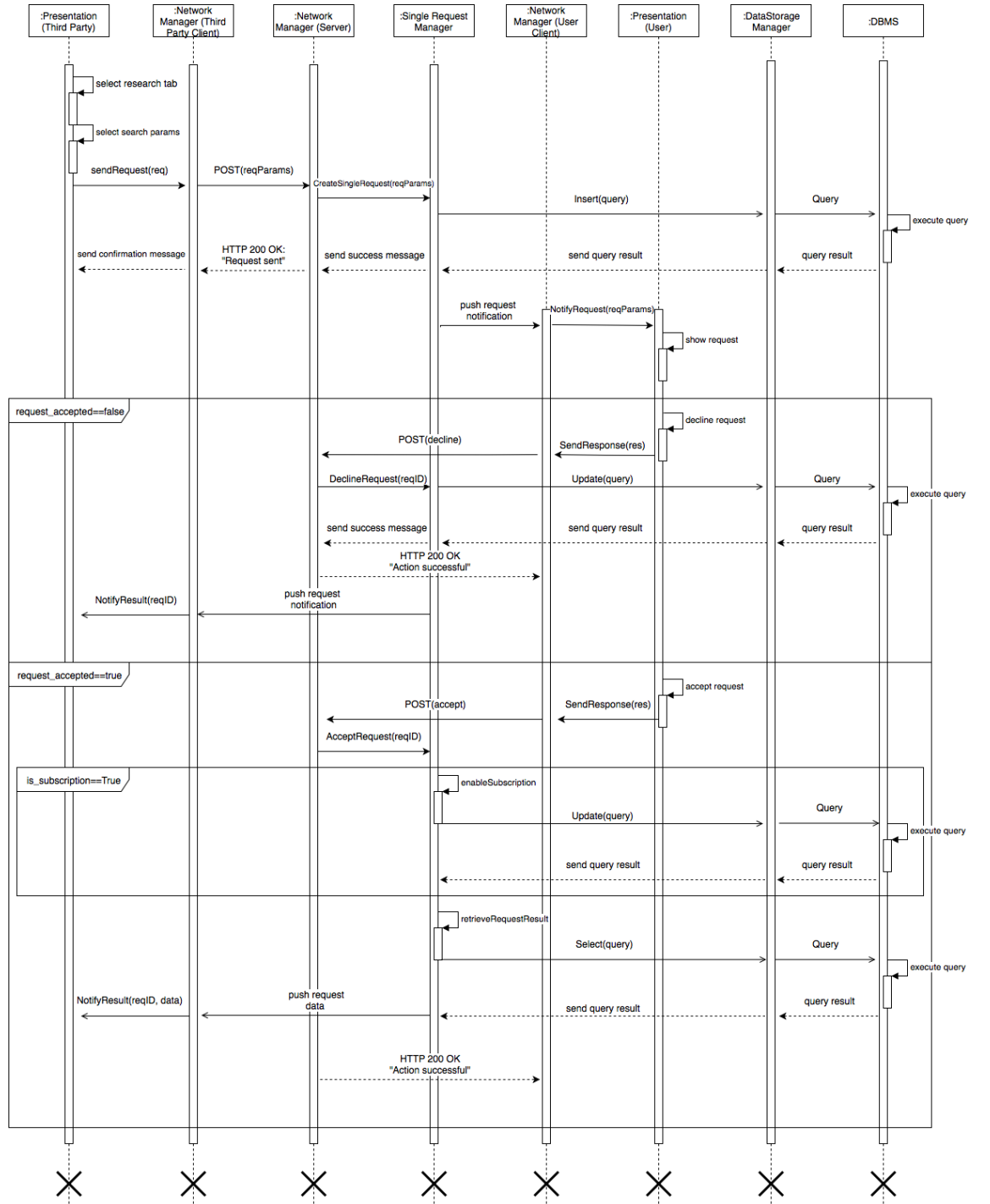


Figure 7: Single Request Runtime View

### Group Request Runtime View

The following diagram describes the events that occur after a Group Request is sent by a *Third Party* to *Data4Help* Application Server. The components involved in this interaction are the *Third Party* Presentation and Network Manager for client side and on server side the Network Manager, Group Request Manager and Data Storage Manager. Finally the DBMS is invoked as an external service. When a *Third Party* select the group research tab and selects all the parameters and filters he is interested in, an HTTP request is sent to the Application server. A new Group Request is created by the Group Request Manager and it is stored into the database. Then a query containing the parameters of the group request is sent to the DBMS. It retrieves the results and checks whether they are related to more than 1000 unique users. If not it sends back an empty response and the Group Request Manager with the help of the Network Manager notifies the *Third Party* that the request was not accepted. If the result matches more than 1000 unique users the Group Request Manager eventually enables the subscription option and sends the retrieved data to the *Third Party*.

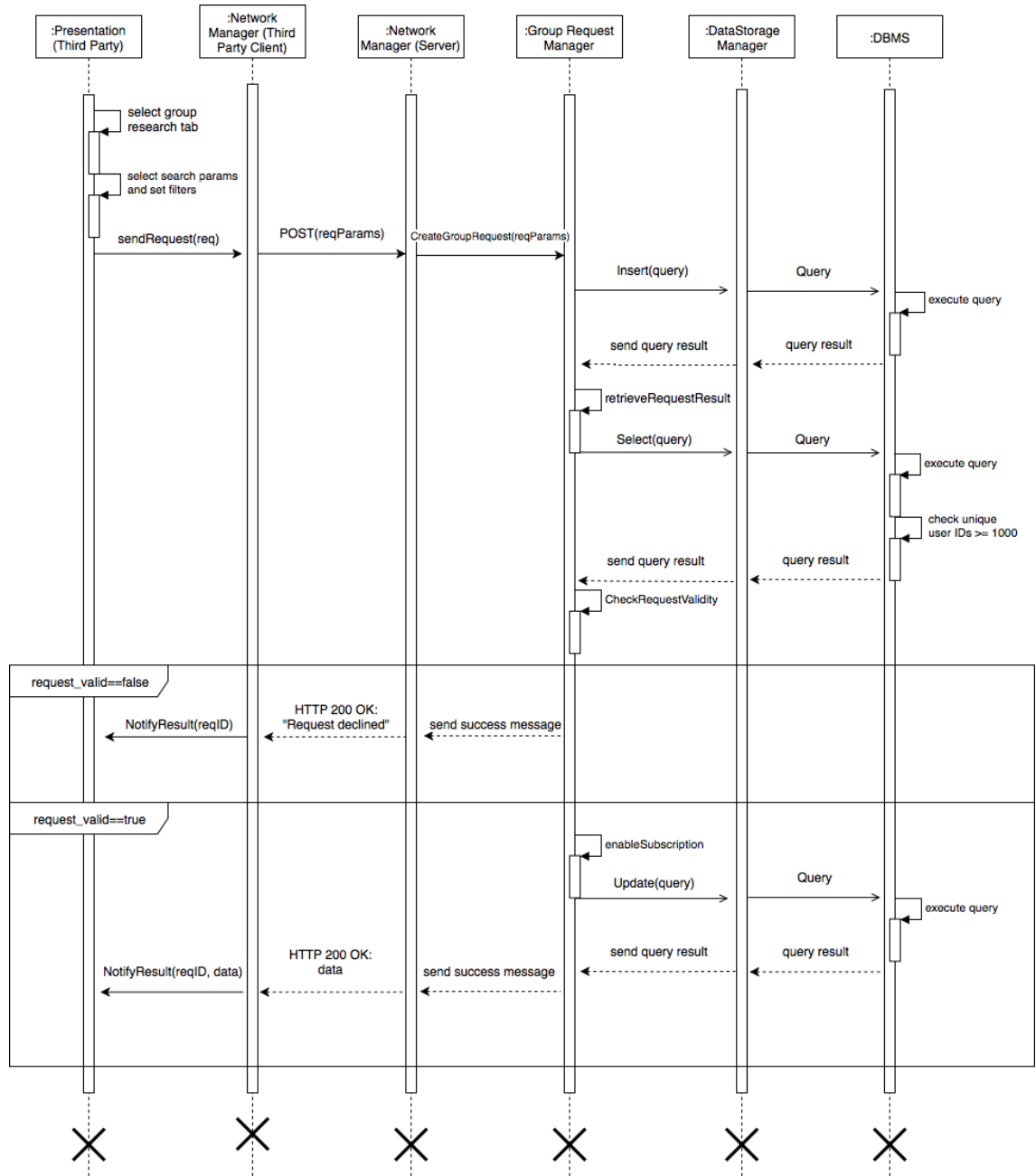


Figure 8: Group Request Runtime View

### AutomatedSOS enabling Runtime View

The following diagram represent the sequence of events that occurs when a *Sin-*

*gle User* decides to enable *AutomatedSOS* and how said feature works. The components involved are all part of the client side: Presentation, *AutomatedSOS*, GPS and TTS. A *Single User* enable *AutomatedSOS* by pressing a toggle in MyHealth tab. *AutomatedSOS* component is activated and start to periodically check the user health status. If parameters denoting a critical condition are observed, the current location of the subject is retrieved from the GPS module and *AutomatedSOS* converts a text message to speech, so it then performs an automated call to an ambulance.

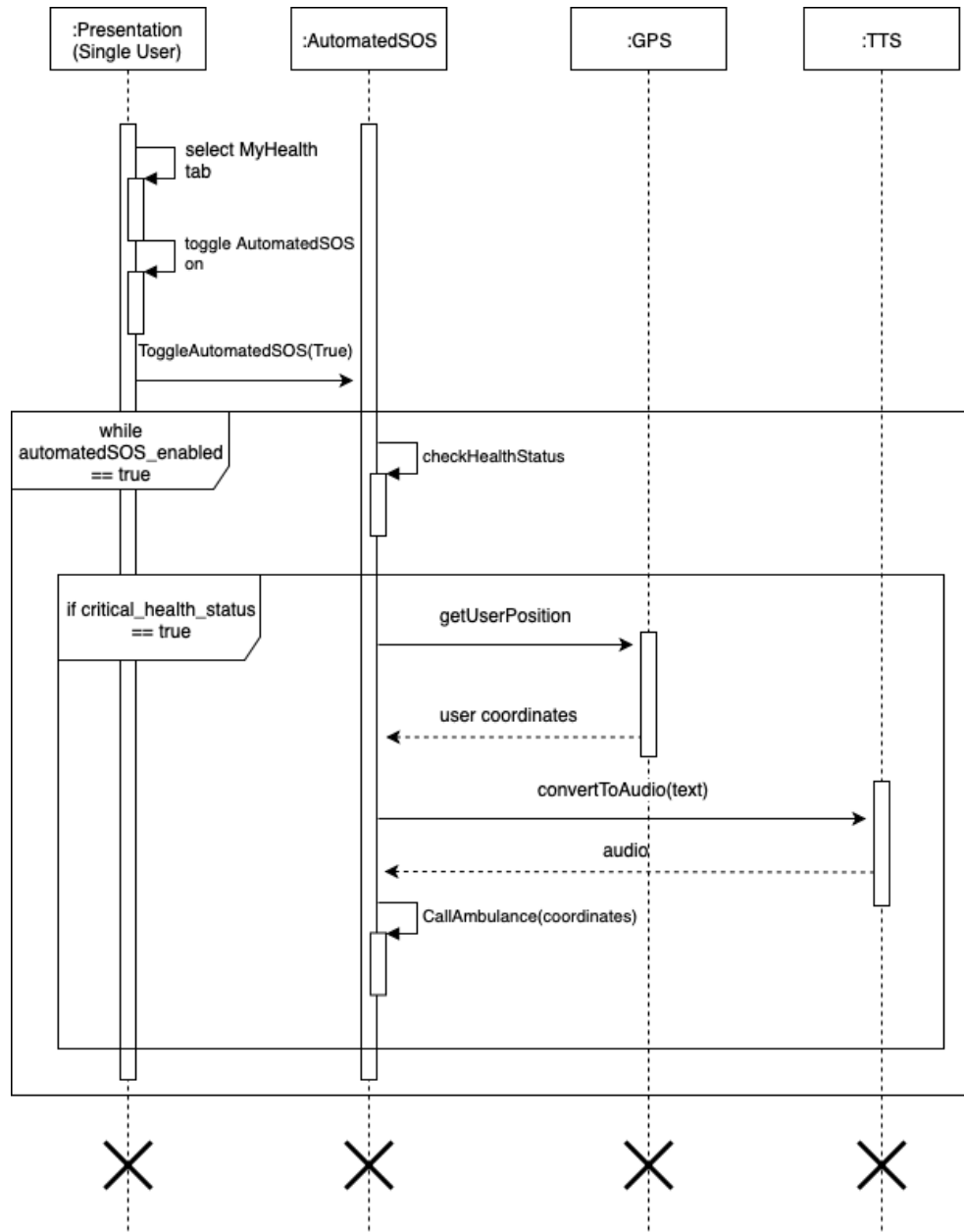


Figure 9: *AutomatedSOS* enabling Runtime View

### Organize a Run Runtime View

The following diagram describes the sequence of events that occur when a logged-in *Third Party* wants to create a run using *Track4Run*. The compo-

nents involved are the *Third Party* presentation and the Network Manager for the client, and the Network Manager, Run Manager and Data Storage Manager for the Application Server, who in turn interacts with the DB. The sequence of events begins with the user opening the New Run tab on his mobile device, and providing as information the info of the run. The application server receives this data, updates the database and generates the RunID, which is then sent to the user. A path is defined by the user, but before checking its validity the server ensures that a correct RunID is provided. If so, the server proceeds to control that the path is feasible: if not it responds with a HTTP 401 Error message, otherwise it updates the DB and responds with a HTTP 200 OK message to the user, terminating the interaction.

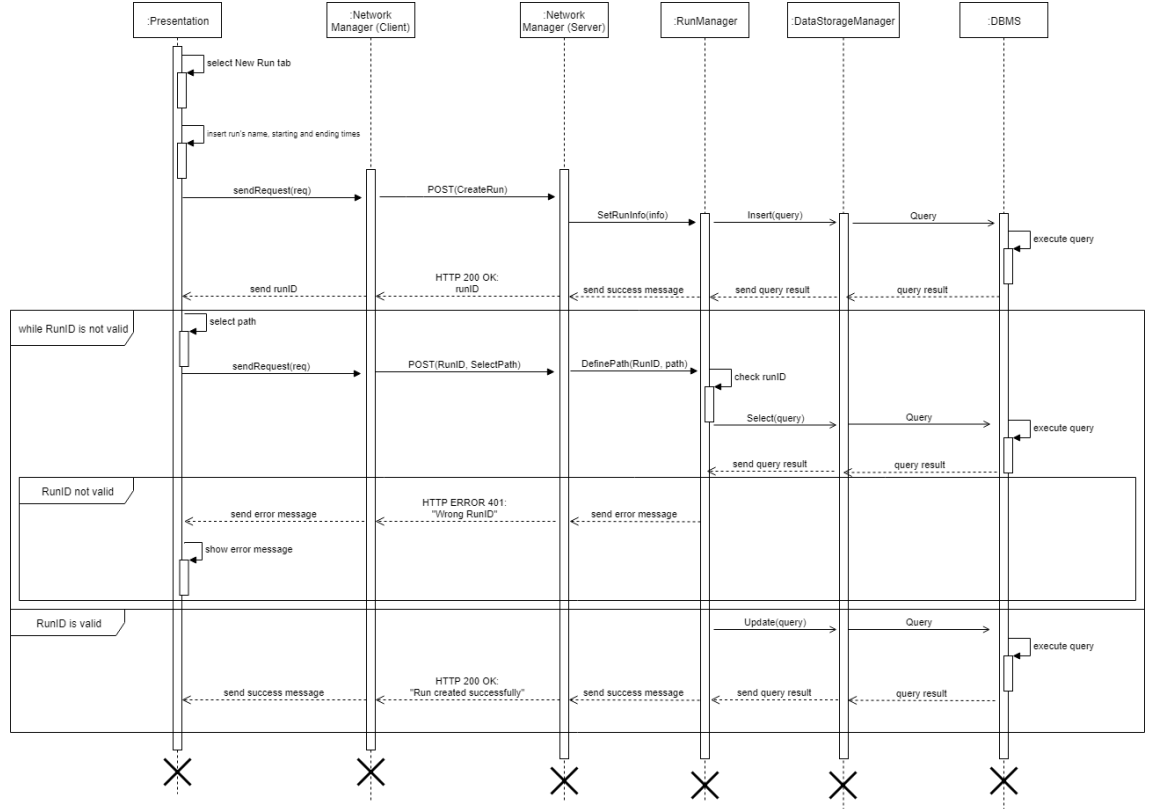


Figure 10: Organize a Run Runtime View

### Join a Run Runtime View

The following diagram describes the sequence of events that occur when a logged-in Private User wants to join a run using *Track4Run*. The components involved are the Private User presentation and the Network Manager for the client, and the Network Manager, Active Runs Manager and Data Storage Manager



for the Application Server, who in turn interacts with the DB. The sequence of events begins with the user opening the Active Runs tab on his mobile device, sending a request to the server to obtain the list of active runs. The server responds with such list, and the user chooses an active run to join. If the run chosen is not valid, the server will respond with a HTTP 401 Error message, otherwise it will update the DB and list the user as a participant. When the day of the run comes, the server pushes a request to *Data4Help* directly (avoiding the proxy as the user is already logged in *Track4Run*) which forwards it to every user, who has a timeout to accept it if he wants to take part in the run. If he doesn't, he is removed from it, otherwise he's listed as a definitive participant and the process terminates with a HTTP 200 OK message.

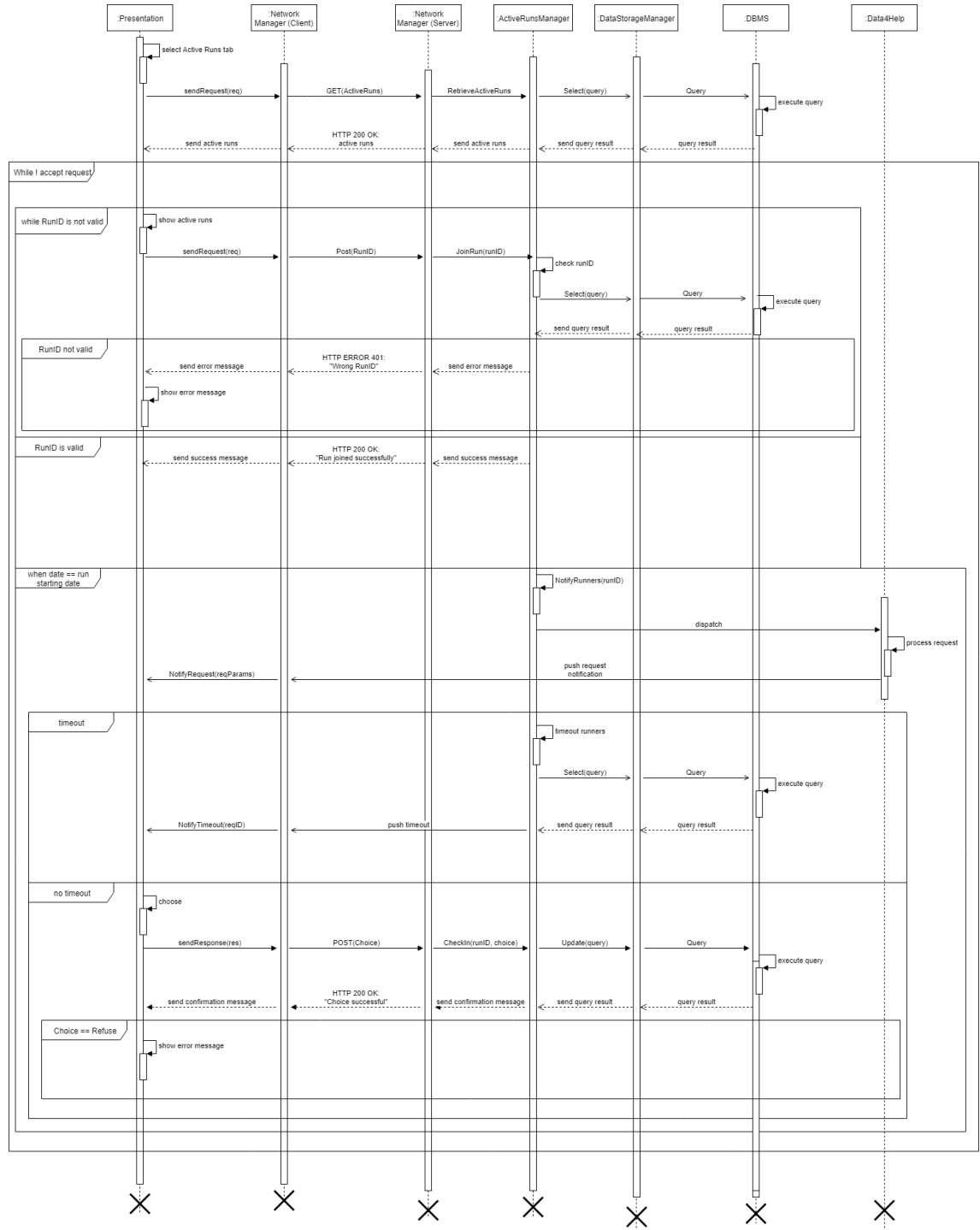


Figure 11: Join a Run Runtime View

### Spectate a Run Runtime View

The following diagram describes the sequence of events that occur whenever a user wants to spectate a run using *Track4Run*, without having to be logged in. The components involved are the User presentation and the Network Manager for the client, and the Network Manager, Active Runs Manager and Data Storage Manager for the Application Server, who in turn interacts with the DB. The sequence of events begins with the user opening the Active Runs tab on his mobile device, sending a request to the server to obtain the list of active runs. The server responds with such list and registers him as a spectator, then the user chooses an active run to spectate. If the run chosen is not valid, the server will respond with a HTTP 401 Error message, otherwise it will update the DB and list the user as a spectator for that run. The process terminates with a HTTP 200 OK message, allowing the user to watch the run live on his device.

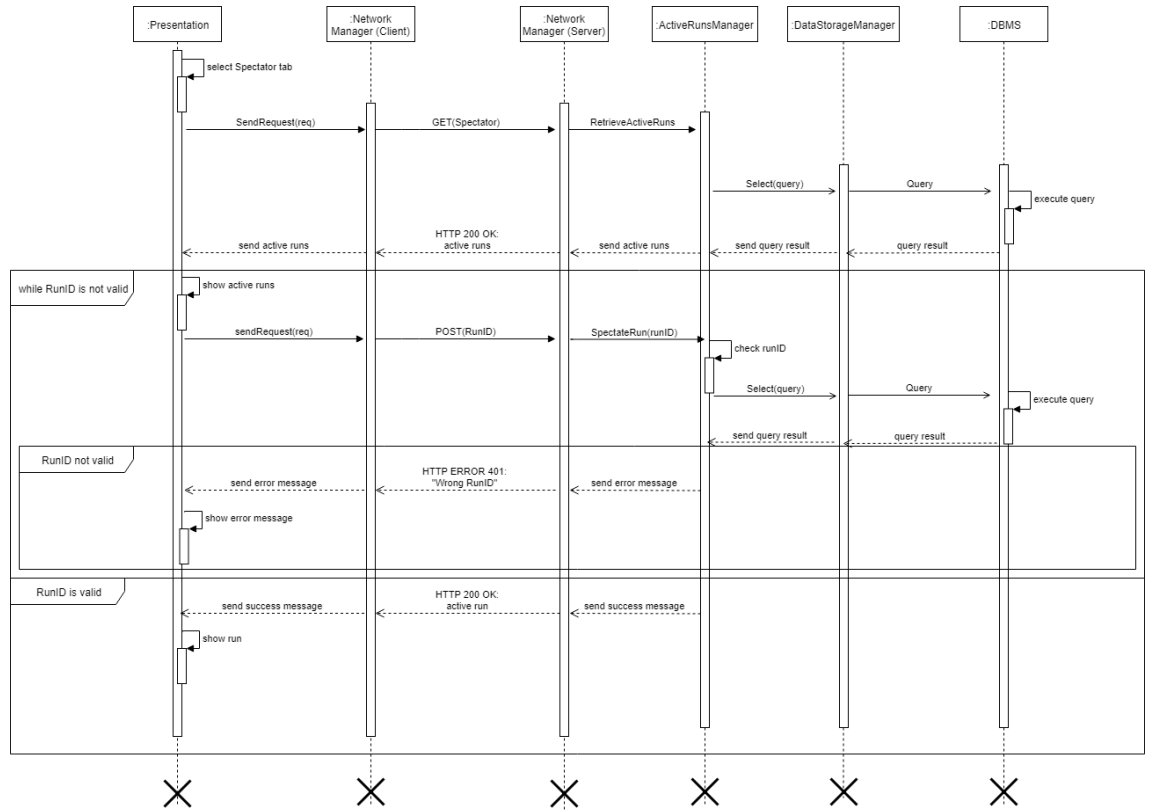


Figure 12: Spectate a Run Runtime View

## 2.5 Component interfaces

### Interface Diagram

Please note that, for readability purposes, most methods requiring the authentication token identifying a specific user do not take it as input parameter. Also, all methods shown in the Runtime Views that are self-calls are obviously not included here, as they are not exposed to the outside through an interface

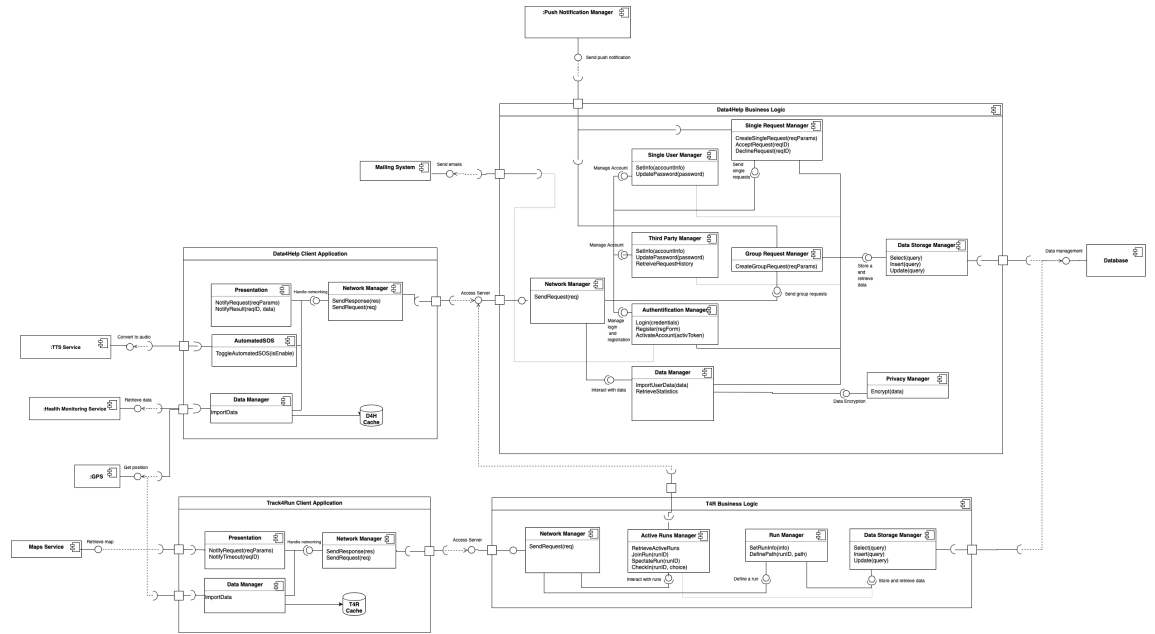


Figure 13: Component interface diagram

## REST API

info	Lets <i>Single Users</i> register to the service
endpoint	/*auth/reg/single
type	POST
req body	email : [string] password : [string] fc : [string] full_name : [string] birthdate : [date] sex : [string]
success code	code: 200 OK body : { "message" : "Registration successful, check your email to complete the creation of your account" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 403 body : { "error" : "Already registered" }
info	Lets third parties register to the service
endpoint	/*auth/reg/tp
type	POST
req body	email : [string] password : [string] piva : [string] company_name : [string] company_description : [string]
success code	code: 200 OK body : { "message" : "Registration successful, check your email to complete the creation of your account" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 403 body : { "error" : "Already registered" }

info	Lets a user log in the service
endpoint	/*auth/login
type	POST
req body	email : [string] password : [string]
success code	code: 200 OK body : { "userType" : [string] "authToken" : [string] }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "Account not activated" }  code: 401 body : { "error" : "Wrong password" }  code: 401 body : { "error" : "Email provided does not exist" }
info	Lets a user activate its account
endpoint	*/auth/activ?activToken=[string]
type	GET
req body	
success code	code: 200 OK  body : { "message" : "Account activated" }
error code	code: 401 body : { "error" : "Invalid token" }  code: 403 body : { "error" : "Account already activated" }

info	Lets a private user modify his account info
endpoint	*/settings/single/info
type	POST
req body	password : [string] full_name : [string] birthdate : [date] sex : [string]
success code	code: 200 OK
	body : { "message" : "Settings updated" }
error code	code: 400 body : { "error" : "Malformed request" }
	code: 401 body : { "error" : "You need to login with a Single User account" }
info	Lets a private user retrieve his account info
endpoint	*/settings/single/info
type	GET
req body	
success code	code: 200 OK
	body : { "settings" : { password : [string] full_name : [string] birthdate : [date] sex : [string] } }
error code	code: 400 body : { "error" : "Malformed request" }
	code: 401 body : { "error" : "You need to login with a Single User account" }

info	Lets a <i>Third Party</i> user modify his account info
endpoint	*/settings/tp/info
type	POST
req body	"company_name": [string] "company_description": [string] "password": [string]
success code	code: 200 OK body : { "message" : "Settings updated" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }
info	Lets a <i>Third Party</i> user retrieve his account info
endpoint	*/settings/tp/info
type	GET
req body	
success code	code: 200 OK body : { "settings" : "company_name": [string] "company_description": [string] "password": [string] } }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }



info	Lets a private user retrieve a list of all requests meant for him
endpoint	*/req/single/list?authToken=[string]
type	GET
req body	
success code	code: 200 body: { "requests": [ { "reqid": [string] "email": [string] "piva": [string] "company name": [string] "types": [[string],...[string]] "status": [string] "subscribing": [bool] "duration": [int] "req_date": [date] "expired": [bool] } ] }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Single User account" }

info	Lets a <i>Third Party</i> retrieve the list of its requests
endpoint	*/req/tp/list?authToken=[string]
type	GET
req schema	
success code	code:200 body: { "requests": [ "single": [ { "reqid": [string] "email": [string] "fc": [string] "full_name" : [string] "types": [[string],...[string]] "status": [string] "subscribing": [bool] "duration": [int] "req_date": [date] "expired": [bool] }], "group": [ { "reqid": [string] "types": [[string],...[string]] "parameters": [ { "datatype": [string] "upperbound": [double] "lowerbound": [double] } ] }] "status": [string] "subscribing": [bool] "duration": [int] "req_date": [date] "expired": [bool] }] }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }

info	Lets a <i>Third Party</i> send a single request
endpoint	*/req/tp/sendSingle
type	POST
req body	"email": [string] "fc": [string] "types": [[string],...[string]] "subscribing": [bool] "duration": [int]
success code	code: 200 OK body : { "message" : "Request sent" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 403 body : { "error" : "Target user does not exist" }  code: 403 body : { "error" : "There already is a pending request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }
info	Lets a <i>Third Party</i> send a group request
endpoint	*/req/tp/sendGroup
type	POST
req body	"types": [[string],...[string]] "parameters": [[string],...[string]] "bounds": [[float, float],...[float, float]] "subscribing": [bool] "duration": [int]
success code	code: 200 OK body : { "message" : "Request sent" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }

info	Lets a <i>Third Party</i> download the result of a single request
endpoint	*/req/tp/downloadSingle
type	POST
req body	"reqID": [string]
success code	code: 200 OK body : { "data" : { [{ "type" : [string] "observations" : [ { "value" : [string] "timest" : [string] } ] }] } }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }  code: 401 body : { "error" : "You can't access this request" }  code: 403 body : { "error" : "Request does not exist or wasn't approved" }

info	Lets a <i>Third Party</i> download the result of a group request
endpoint	*/req/tp/downloadGroup
type	POST
req body	"reqID": [string]
success code	code: 200 OK body : { "data" : { [{ "userid" : [string] "data" [{ "type" : [string] "values" : [{ "value" : [string] "timest" : [string] }] }] }] } }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Third Party account" }  code: 401 body : { "error" : "You can't access this request" }  code: 403 body : { "error" : "Less than 1000 users match the search parameters" }  code: 403 body : { "error" : "Request does not exist" }

info	Lets a user end the subscription
endpoint	*/req/sub/endSingle
type	POST
req body	"reqID": [string]
success code	code: 200 OK body : { "message" : "Subscription ended" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You can't access this request" }  code: 403 body : { "error" : "Request does not exist or the subscription is already off" }  code: 403 body : { "error" : "Request has already expired" }

---

info	Lets a user end the subscription
endpoint	*/req/sub/endGroup
type	POST
req body	"reqID": [string]
success code	code: 200 OK body : { "message" : "Subscription ended" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You can't access this request" }  code: 403 body : { "error" : "Request does not exist or the subscription is already off" }  code: 403 body : { "error" : "Request has already expired" }

info	Lets a private user accept or decline a single request
endpoint	*/req/single/choice
type	POST
req body	"reqID": [string] "choice": [string]
success code	code: 200 OK body : { "message" : "Action successful" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Single User account" }  code: 403 body : { "error" : "Request does not exist or is not pending" }
info	Lets a private user load data
endpoint	*/data/upload
type	POST
req schema	"types": [[string],...[string]] "values": [[string],...[string]] "timestamps": [[string],...[string]]
success code	code: 200 OK  body : { "message" : "Data imported" }
error code	code: 400 body : { "error" : "Malformed request" }  code: 401 body : { "error" : "You need to login with a Single User account" }

info	Lets a private user retrieve statistics
endpoint	*/data/stats/avg
type	POST
req body	"types": [[string],...[string]]
success code	code: 200 OK "body" : { "data" : { [ { "type": [string] "observations": [ "avg" : [double] "min" : [double] "max" : [double] "month" : [string] "year" : [string] ] "others": [ "avg" : [double] "min" : [double] "max" : [double] "month" : [string] "year" : [string] ] } ] } } }
error code	code: 400 "body" : { "error" : "Malformed request" }  code: 401 "body" : { "error" : "You need to login with a Single User account" }



info	Lets a <i>Third Party</i> create a run
endpoint	*/t4r/org/create
type	POST
req body	"runName": [string] "startTime": [string] "endTime": [string] "participants": [int]
success code	code: 200 OK "body" : { "message" : "Info set successfully" "runID" : [int] }
error code	code: 400 "body" : { "error" : "Malformed request" }  code: 401 "body" : { "error" : "Wrong credentials" }
info	Lets a <i>Third Party</i> set the path for a run
endpoint	*/t4r/org/path
type	POST
req body	"runID": [int] "path" : [ { "coordinate" : [string] } ]
success code	code: 200 OK "body" : { "message" : "Run created successfully" }
error code	code: 400 "body" : { "error" : "Malformed request" }  code: 401 "body" : { "error" : "Invalid path" }  code: 401 "body" : { "error" : "Wrong credentials" }  code: 401 "body" : { "error" : "Wrong runID" }

info	Lets a user (not necessarily logged in) fetch a list of active runs
endpoint	*/t4r/active/list
type	GET
req body	
success code	code: 200 OK "body" : { "run" : [ { "runID": [string] "runName": [string] "startTime": [string] "endPoint": [string] "participants": [int] } ] }
error code	
info	Lets a private user join an active run
endpoint	*/t4r/active/join
type	POST
req body	"runID": [int]
success code	code: 200 OK "body" : { "message" : "Run joined successfully" }
error code	code: 400 "body" : { "error" : "Malformed request" }  code: 401 "body" : { "error" : "Wrong runID" }  code: 401 "body" : { "error" : "Wrong credentials" }

info	Lets a spectator choose a active run
endpoint	*/t4r/active/spec
type	POST
req body	runID : [int]
success code	code: 200 OK "body" : { "path" : [ { "coordinate": [string] } ] }
error code	code: 401 "body" : { "error" : "Wrong runID" }
info	Lets a spectator get the coordinates of runners
endpoint	*/t4r/active/specUpdate
type	body
req schema	runID : [int]
success code	code: 200 OK "body" : { [ runnerName : [string] coordinate : [string] ] }
error code	code: 401 "body" : { "error" : "Wrong runID" }

## 2.6 Selected architectural styles and patterns

### Multi-tier Architecture

As stated in the Overview, the chosen architecture for the *System* is a multi-tier architecture, with three partially distributed layers. This division guarantees that each tier only deals with a specific task, so that computation is correctly spread amongst the *System* and there isn't a central node in charge of everything. Also, modifications to a specific layer won't affect the others, so the overall maintenance of the *System* is much easier.

In the first tier there is the Client's physical device, which can be a mobile device or a desktop PC, in the case of Third Parties directly accessing the Application Server. The Presentation layer is completely managed by this tier, with the aim of displaying to the user all informations received by the Application Server. This is done through a dynamic GUI on the mobile device, which also displays services offered by third parties such as the Map visualization service, used by *Track4Run*. Desktop PCs, on the other hand, do not implement the Presentation layer, as their only use of *Data4Help* and *Track4Run* is to query their APIs inside specific environments defined by the users. Mobile devices also implement part of the business logic to guarantee the fulfillment of *AutomatedSOS*'s nonfunctional requirements, and part of the data storage, as they hold a cache for data collected in the last week. Since these contributions to the Business and Data Storage layer are minimal, we will consider the Client tier to only implement the Presentation layer, thus being a Thin Client.

In the second tier there is have the Nginx Web Server, functioning both as a reverse proxy and as a load balancer. The purposes of these machines, used by *Data4Help* and *Track4Run*, is: To retrieve data from the Application Servers and provide it to the Client as if the proxy were the server itself. To redirect requests to the idle servers, avoiding congestions on each one. Nginx was chosen over Apache for its flexibility in handling requests and its efficient use of main memory, thanks to the event-driven approach used. This tier contributes to the implementation of the Business Logic layer, even though it doesn't do it directly, but it helps the Application Servers in doing it.

In the third tier there are the Application Servers of *Data4Help* and *Track4Run*. The deployment will be done by replicating each server as many times as necessary to guarantee the nonfunctional requirements expressed in the RASD. In case of a sudden increase in the number of requests, we could seamlessly add more machines to handle the traffic, as each one is stateless. In fact, this tier only implements the Business and Domain Logic layer, but not the Data Storage layer.

In the fourth and final tier we have the Database, shared by *Data4Help* and *Track4Run*. This tier is not handled by TrackMe directly, as a cloud solution was preferred to one on premises, sparing the company from having to ensure

the reliability of the DB and its management. This tier fully implements the Data Storage layer.

## REST

The Representational State Transfer style has been used as a guideline to implement the services offered by the Application Servers, and their interaction with the Clients. All services are accessible through stateless operations, so that together with the servers being stateless themselves allow for a scalable *System*. HTTP is the communication protocol used underneath, so CRUD operations are implemented through GET, POST, DELETE and PUT primitives. Using HTTP and complying with REST guidelines allows to provide clear and complete APIs to the Clients, usable by both mobile devices and desktop PCs, thus ensuring Portability.

## MVCS

The software *System*'s architecture will follow the MVCS paradigm. The acronym stands for Model View Controller Store.

- The Model corresponds to the component containing all the application's data and integrity constraints. In our *System* the Model is completely described in the external database.
- The Controller is responsible for the logic of the application and in our case is also stateless in order to be replicated and seamlessly support high traffic of requests. It corresponds to the business logic layer and in our Application is composed of the third tier and the *AutomatedSOS* component in the Client application.
- The View corresponds to the presentation layer, the client's only interface with the *System*.
- Finally the Store component handles the fetching and saving of data and is responsible for the interaction between logic and storage. In our case it is represented by the Data Storage Interface, the unique point of communication with the external cloud Database server in both *Data4Help* and *Track4Run*.

This best practice design pattern is commonly used in client server applications due to its power of decoupling all main components thus increasing maintainability: modifications to one component will be transparent to others.

## 2.7 Other design decisions

### Firewall

The *Systems* architecture includes two network firewalls that keep a separation between the reverse proxies and the external network. The external network is regarded as untrusted and may represent a menace to the internal, trusted, network of the *System*. Network firewalls operate by filtering the packets trying to pass through them. The reverse proxies may have been used to perform firewall functions as well but it has been preferred to decouple packet filtering from packet dispatchment.

### Database

It was decided to use a relational database model for both *Data4Help* and *Track4Run*, as opposed to a non-relational model, because it allows to represent the intrinsic relationships present in the data needed to be stored. Also, many complex operations will be made on the database, including Joins and Updates, which are not well handled by a non-relational model. A viable alternative would have been to use a relational database for *Data4Help*, and a non-relational one for *Track4Run*, since most of the data is just append-only made of couples {key,value}, containing the runnerID and its coordinates. Nonetheless, this hybrid option was discarded in order to be cohesive with *Data4Help*. The following is the E-R diagram of both *Data4Help* and *Track4Run*.

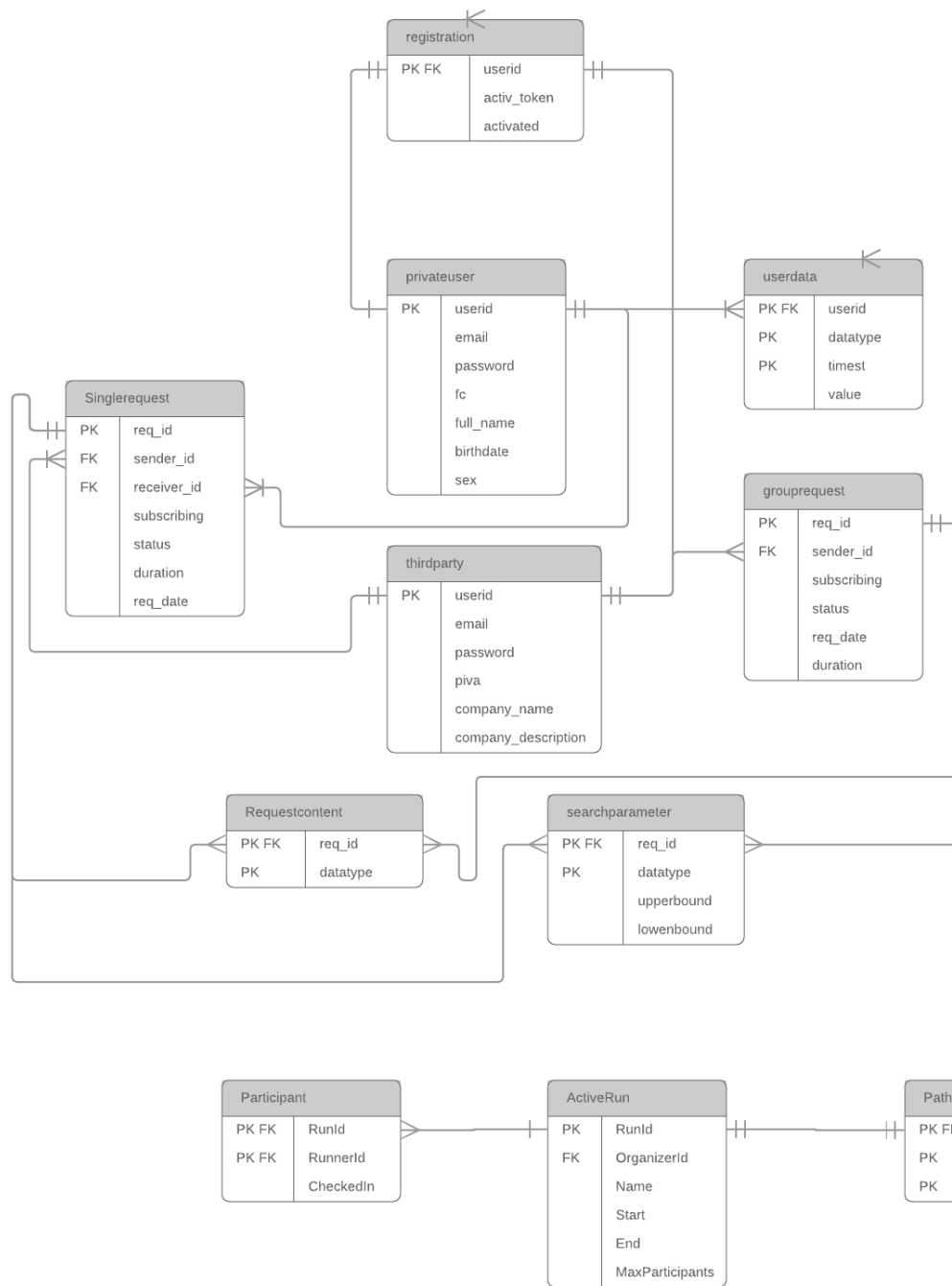


Figure 14: Entity-Relationship diagram

### 3 User Interface Design

We already assessed the UI design in section 3.1.1 of the RASD document. In this section, we provide a better overview about the User Experience flow by means of a User Experience diagram. The following two diagrams show the relations between the Application's sections.

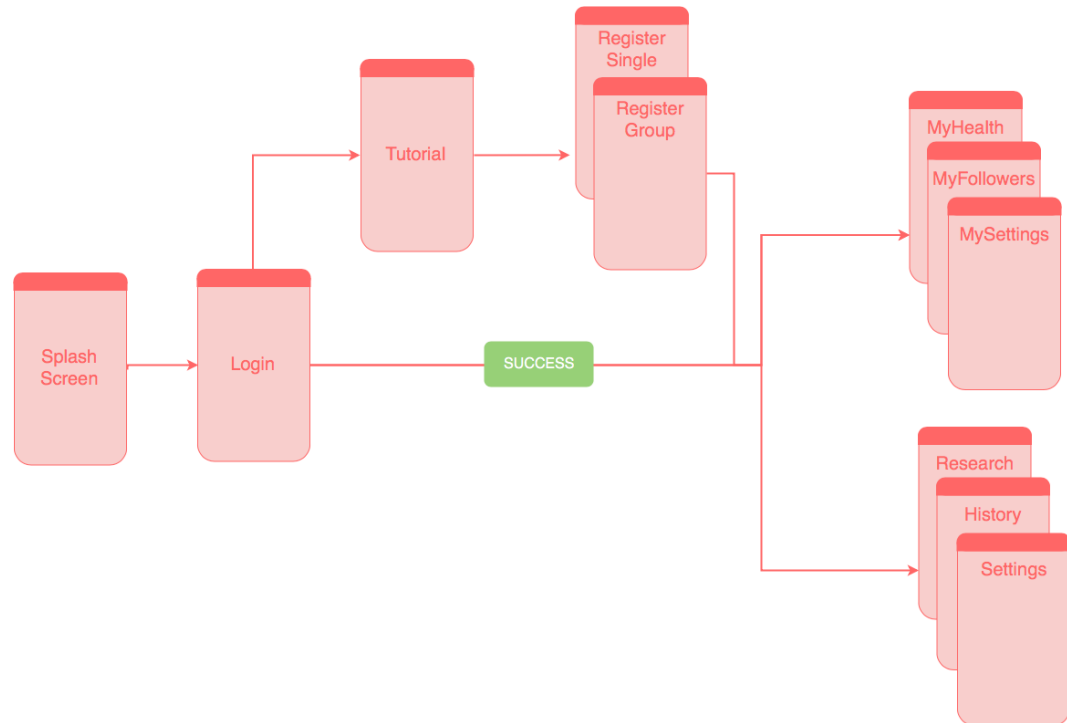


Figure 15: *Data4Help* UX Flow



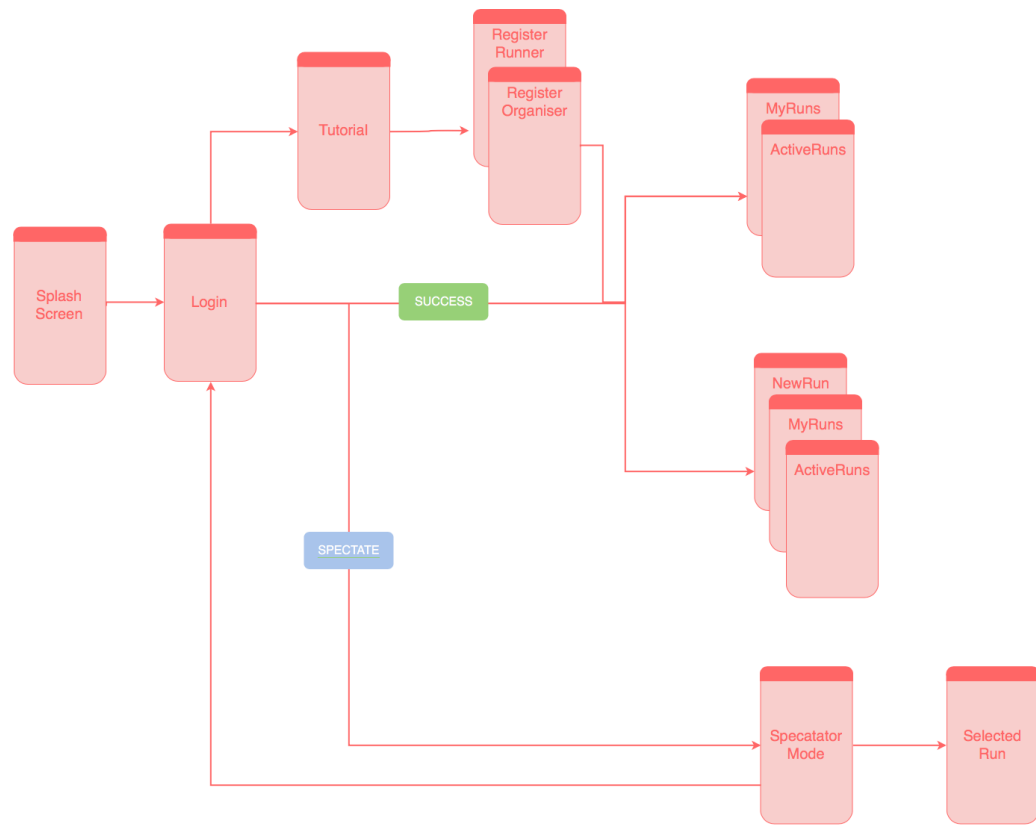


Figure 16: *Track4Run* UX Flow

## 4 Requirements Traceability

Component (DD)	Requirements (RASD)
Authentication Manager	<ul style="list-style-type: none"> <li>• [R1] The S2B allows users to create either a <i>Single User</i> or a <i>Third Party</i> account.</li> <li>• [R2] A <i>Single User</i> account can be created if and only if the user provides his Fiscal Code.</li> <li>• [R3] A <i>Third Party</i> account can be created if and only if a valid P.IVA is provided.</li> <li>• [R4] All users can create an account if and only if they provide a unique email and a password.</li> <li>• [R5] To access the service users must log in with their account credentials.</li> <li>• [R35] The S2B allows organisers to create a <i>Third Party</i> account using <i>Data4Help</i>.</li> <li>• [R43] The S2B allows runners to create a <i>Single User</i> account using <i>Data4Help</i>.</li> </ul>
<i>Single User</i> Manager	<ul style="list-style-type: none"> <li>• [R32] Only private users can choose whether or not to enable <i>AutomatedSOS</i>.</li> </ul>

## Single Request Manager

- [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.

Group Request Manager	<ul style="list-style-type: none"> <li>• [R19] Subscriptions to requests must specify a duration.</li> <li>• [R23] Third Parties can download all data obtained through requests on their devices or have it sent by email.</li> <li>• [R26] Third Parties can submit a group request to access data of groups of users.</li> <li>• [R25] Group requests must include at least one search parameter.</li> <li>• [R26] Group requests must specify the requested data types.</li> <li>• [R27] Group request results are provided if and only if the number of users matching the search parameters is higher than 1000.</li> <li>• [R28] Group request results include only the data retrieved by the <i>System</i> matching the search parameters.</li> <li>• [R30] All group requests must specify whether the <i>Third Party</i> is subscribing to that request of data.</li> <li>• [R31] Subscriptions to requests can be ended by the <i>Third Party</i> at any time.</li> </ul>
Data Storage Manager	<ul style="list-style-type: none"> <li>• [R29] Sensitive data is excluded from group request results.</li> </ul>
Data Manager (Client)	<ul style="list-style-type: none"> <li>• [R6] The S2B automatically imports new data whenever the application is opened.</li> <li>• [R7] When the application is open, the S2B continuously import data in background.</li> </ul>

Data Manager (Server)	<ul style="list-style-type: none"> <li>• [R6] The S2B automatically imports new data whenever the application is opened.</li> <li>• [R7] When the application is open, the S2B continuously import data in background.</li> <li>• [R8] The S2B binds collected data only to the user's account that imported it.</li> </ul>
Health Status Manager	<ul style="list-style-type: none"> <li>• [R14] The S2B allows <i>Single Users</i> to visualise their historical data using Time Series.</li> <li>• [R15] The S2B allows <i>Single Users</i> to visualize their historical data using aggregated statistical operators.</li> <li>• [R16] The S2B allows <i>Single Users</i> to visualize their historical data over multiple timespans.</li> </ul>
<i>AutomatedSOS</i> Manager	<ul style="list-style-type: none"> <li>• [R32] Only private users can choose whether or not to enable <i>AutomatedSOS</i>.</li> <li>• [R33] <i>AutomatedSOS</i> can be enabled only if the user grants permission to make emergency phone calls.</li> <li>• [R34] If <i>AutomatedSOS</i> is enabled and the <i>System</i> detects that a user's heart rate is below or above the critical threshold for his age, an ambulance is called.</li> </ul>
Run Manager	<ul style="list-style-type: none"> <li>• [R36] Only Third Parties can create a run.</li> <li>• [R37] When creating a run, the organiser must specify its the path, duration and maximum participants.</li> </ul>

Active Runs Manager	<ul style="list-style-type: none"> <li>• [R38] On the day of a run the S2B sends a Single Request to every user who has registered to that run.</li> <li>• [R39] The request is sent on behalf of the organiser using its email and P.IVA.</li> <li>• [R42] The S2B provides a list of all existing runs visible to everyone using <i>Track4Run</i>.</li> <li>• [R44] Only users with a <i>Single User</i> account can join an existing run.</li> <li>• [R45] A user can only join a run in the list provided by the S2B.</li> <li>• [R46] A user can't check-in for a run if he doesn't accept the request received by the organiser of such run.</li> <li>• [R47] A user can't accept the request if he doesn't have at least his position available as requestable data type.</li> <li>• [R48] If a user fails to check-in in the run, the S2B removes him from it.</li> <li>• [R49] An existing run can be spectated without logging in the <i>System</i>.</li> <li>• [R50] An existing run can be spectated by selecting it in the list of existing runs.</li> <li>• [R51] Spectators can see the live position of all runners participating in the run they are spectating.</li> </ul>
---------------------	--

Table 3: Requirements Traceability

## 5 Implementation, Integration and Test Plan

### 5.1 Overview

The approach followed in the design of the S2B was Top-Down, first defining the subsystems and their macro functionalities at a high abstraction level, and then refining each one up to methods and parameters needed. The subsystems we identified are:

- The client subsystem, containing the presentation and the components related to *AutomatedSOS* and data retrieval
- The server subsystem, containing most of the components related to Business Logic and interaction with the database
- External services, such as the DB and the Maps service

However our implementation and testing approach will be incremental and will follow both top-down and bottom-up strategies as they are the most reasonable for relatively small components and subsystems. In the following paragraphs we will detail our strategies and discuss their advantages.

### 5.2 Implementation

The order of implementation takes into account the necessity of parallelising the workload between developers, and can be identified as:

1. Setup of the Database and creation of the schema and its constraints
2. Implementation of the server and client subsystems at once
3. Integration with external services

The setup of the database will consist in the deployment of a running instance of a new database on the selected external provider. It will then follow the creation of all the tables identified in section 2.7 and of the related constraints.

The implementation of the Server and Client will proceed in parallel to optimize the efficiency between developers.

On the server side the implementation will follow this order:

- Data Storage Manager, Network Manager
- Auth Manager, Settings Manager
- *Single User* Manager, *Third Party* Manager, DataManager
- Single Request Manager, Group Request Manager
- Privacy Manager

In particular Data Storage Manager and Network Manager will be implemented first because they represent the access point to the Clients and the DB and eventual changes in the structure of the interaction between the various subsystems would have repercussions on all other components.

On the client side the implementation will follow this order:

- UI Components, Views
- NetworkManager
- *AutomatedSOS*, ViewControllers, DataManager

The DataManager component represents one of the most critical part of the *Systems* thus it will be immediately tested to check the interaction with the Health Monitoring Service.

### 5.3 Unit testing

During the whole duration of the implementation phase unit tests will be performed in order to spot as soon as possible bugs and flaws inside *System* components. Performing unit tests during the implementation of single software components is essential to be able to fix bugs with the lowest cost of repair in terms of effort and time and it also helps guiding the design of the *System*. Unit tests will be added incrementally as new code is written. A support tool will be used to perform their execution automatically once a new version of the software is available. An automated test tool will be used for two reasons. Firstly, it saves time for developers that do not have to manually perform all tests every time they add or change something. Secondly, it allows to make sure that a test that was valid for a previous version of the code still holds after some changes/additions in order to preserve consistency and compatibility. Unit tests on components will mainly focus on exercising interfaces, internal behaviour (e.g initialization) and modules' interaction. Regarding this last aspect, the only way to make components work even in isolation is to simulate the ones that are missing. Drivers will be used to simulate calls to a certain component. Stubs will be used to simulate the calls made by a component under testing.

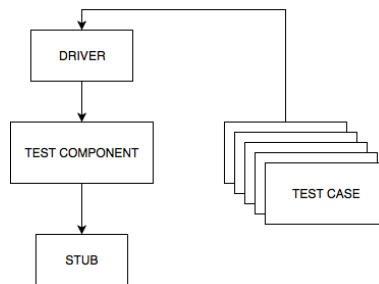


Figure 17: Test scaffolding



## 5.4 Integration testing

Once we have defined how we will approach the implementation of each component of our *System* and how to test it independently, we shall focus on how to test their interaction and validate their joint behavior. Integration between components means testing groups of components that depend on one another in order to assess their interaction and possibly expose any defects. We will perform integration testing at two different levels: at subsystem level, meaning between components of the same macro *System* such as modules of the Controller or of the View, and at a *System* level, linking all systems together.

As previously mentioned, subsystem integration will be performed within a macro *System* and will follow an incremental approach as done during implementation and testing. This approach will follow a top down procedure according to the use hierarchy of modules. Previously created stubs in fact will be now substituted by real components. The only stubs and drivers we will use in this type of integration will be those of the Client (driver), the DBMS(stub). We will adopt this approach both for *Data4Help* and *Track4Run*'s architectures. In order, we will first integrate the first two levels of the use hierarchy and proceed until the very last that is composed by the Data Storage Manager, the component with functions that are called by all other components and that interacts with external services.

The following diagram represents the use hierarchy of modules within the *Data4Help*'s server. As already mentioned, the order of integration we have opted for follows modules from the root to the leaves.

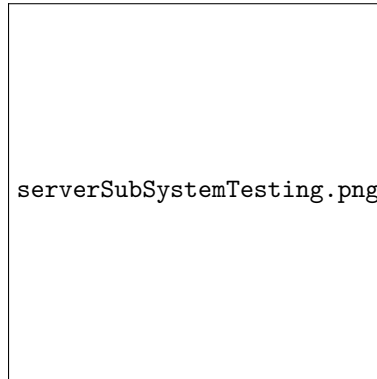


Figure 18: Server SubSystemtesting

The following diagram instead contains the use hierarchy within the *Data4Help* client and shows our integration flow (from the root to the leaves).

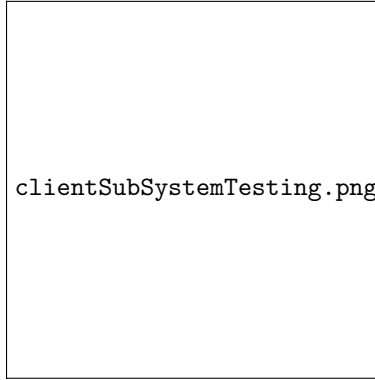
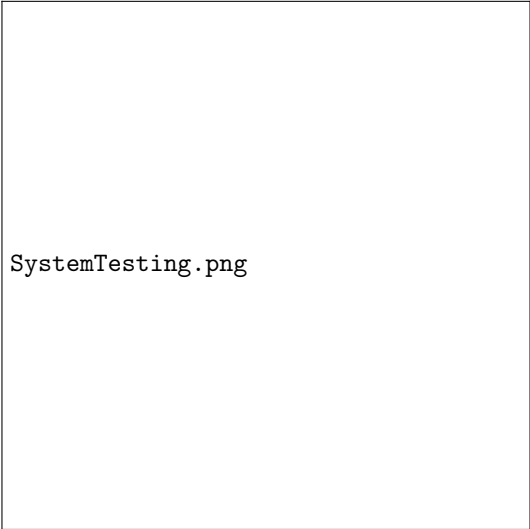


Figure 19: Client SubSystemtesting

As mentioned above, the same approach will be adopted with *Track4Run*'s sub-Systems.

After having integrated each component within a *System*, we will focus on integrating all systems together. Our strategy in this case will be a bottom up approach based on the use hierarchy. At the very bottom level there is the Model of our *System*, the first element that we have created and that is interrogated by all other modules. At first we shall integrate Controller and Model in order to test calls from the former to the latter. When integrating these two components we will also use Postman to simulate client requests. This service is a simple JavaScript code that enables us to send API requests instead of having a complete Client to interact with. Afterwards we will add the Client to our previously integrated systems and verify the complete functioning of our *System*. The Client corresponds to the root of our hierarchy as it interacts only with the controller that handles its requests and communicated with the database. Finally, when performing *System* integration, we will also include external service, such as Map Service, Health Monitoring Service or Push notification Service. Those used by the Server will be integrated with it after integration with the Model. Those instead used by the Client will be included at the very end.

The following diagram represents how our *System* test will be performed. It is true both for *Data4Help* and *Track4Run*. Arrows represent the use hierarchy, meaning that the arrows start from modules that use those corresponding to the end points. As mentioned before, our integration procedure will begin from the leaves to the root.



SystemTesting.png

Figure 20: System testing

The very last step of our testing will include the integration of *Data4Help* and *Track4Run*. As previously mentioned in this document, *Track4Run* relies on multiple functionalities provided by *Data4Help*'s modules, such as those exposed by components handling requests and managing user accounts. It is then essential to test integration between the two system in order to guarantee a seamless communication in front of the users.

## 6 Effort Spent

In the following tables we will summarize the effort spent by each member of the team on the *DD* Document.

Date	Task	Hours
19/11/18	Pre-analysis	2
20/11/18	Section 1 and Latex setup	1
21/11/18	Component View	3
23/11/18	Review meeting and task planning	2
24/11/18	RuntimeView and Overview	4
25/11/18	Review Component View and RuntimeView	2
26/11/18	Review	1
27/11/18	Requirements Traceability	2
28/11/18	Architectural Patterns	1
04/12/18	Group Revision	2
05/12/18	Integration testing	3
07/12/18	Group revision	2
08/12/18	<i>Track4Run</i> testing	2
08/12/18	Final Revision	2
08/12/18	Copy on Latex	2
		<b>Total</b>
		31

Table 4: Virginia Negri's effort

Date	Task	Hours
18/11/18	Document Structure	1
19/11/18	Pre-analysis	2
20/11/18	DB design	2
21/11/18	APIs	2
23/11/18	Review meeting and task planning	2
24/11/18	Deployment view	2
25/11/18	RuntimeView	4
02/12/18	ER Diagram	2
04/12/18	Group Revision	2
05/12/18	User Interface design	2
05/12/18	Implementation	1
06/12/18	Unit testing	1
07/12/18	Group revision	2
08/12/18	Final Revision	1
08/12/18	Copy on Latex	2

		<b>Total</b>
		28

Table 5: Luca Molteni's effort

<b>Date</b>	<b>Task</b>	<b>Hours</b>
17/11/18	Purpose and Scope	1
19/11/18	Pre-analysis	2
20/11/18	DB design	2
21/11/18	APIs	2
23/11/18	Review meeting and task planning	2
23/11/18	APIs	1
24/11/18	Architectural Styles	2
25/11/18	RuntimeView	3
29/11/18	APIs review	2
04/12/18	Group Revision	2
05/12/18	Implementation	2
07/12/18	Group revision	2
07/12/18	Revision	2
08/12/18	Final Revision	2
08/12/18	Copy on Latex	4
		<b>Total</b>
		31

Table 6: Francesco Lorenzo's effort