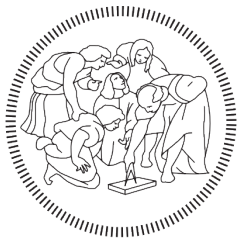


POLITECNICO
MILANO 1863

DD

Software Engineering 2 Project - TrackMe

v1.1 - 12/01/2019

Authors

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

Contents

1 Introduction

1.1 Purpose

The main purpose of this document is to exhaustively describe the Data4Help main architecture, its parts and how they interact. There is also a chapter that covers the user interface.

The main recipients of this document are the project manager, developers and testers, but it can also be useful for further development reference and maintenance.

1.2 Scope

As described in the RASD document, the Data4Help service is made of four distinct components: three clients (Data4Help Smartwatch App, Data4Help Mobile App, Website) and a server application (Data4Help Core).

In this document, we will discuss the several layer of the system and how the multiple parts interact with each other. The users, who are divided into Companies, Individuals and Run organizers, can subscribe to the service and take advantage of the functionalities offered. The system is in charge to verify that the data received will be stored in the database consistently. The system is build as Multi-tier, divided in Client, Business Logic, and Data tier. External interfaces must be implemented in the business logic, in order to exploit external services APIs.

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

1.3 Definitions, Acronyms, Abbreviations

i.e.	*Id est*, that is
w.r.t	with respect to
Company	Third party company
BLE	Bluetooth Low Energy

1.4 Revision history

v1.0 - 05/12/2018 - Release

v1.1 - 12/01/2019

- Modified endpoints and their descriptions.
- Corrected error in definition of a goal.

1.5 Reference documents

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

1.6 Document structure

This document is divided in the following chapters:

Introduction This chapter gives an overview of the document, its main recipient and how it's organized.

Architectural Design This chapter provide some details about the various component of the system, and how they interacts.

User interface design This chapter provide some mockups of the user interface. This will be useful during the development of the frontend.

Requirements traceability This chapter unfold how the functional requirements listed in the RASD document will be obtined using the architectural pattern proposed.

Implementation, integration and test plan This chapter will explain in which order the various subcomponents of the system will be realized, in which order they will be integrated and how the inegrations will be tested.

2 Architectural Design

2.1 Overview

This section is intended to explain in detail all the components used by the system in order to offer all the functionalities.
The figure below shows the high-level architecture of the system-to-be. All the components are better explained in the following paragraph.

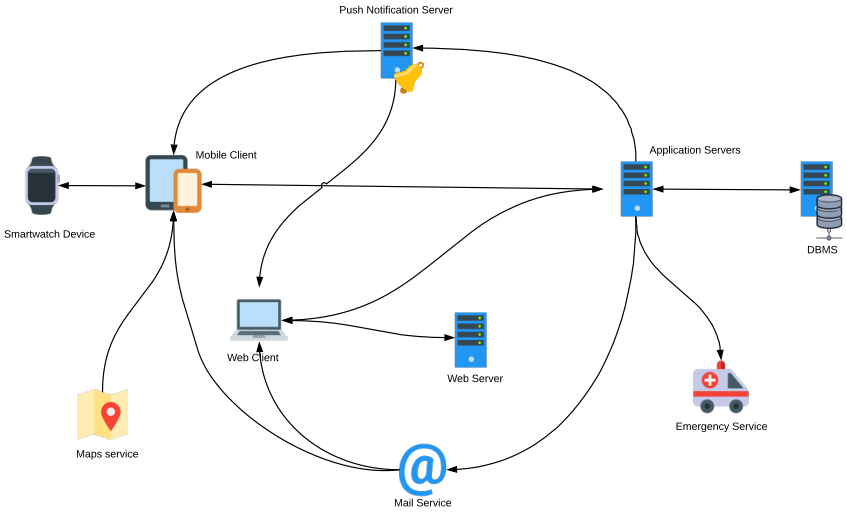


Figure 1: System overview graph

2.2 Component view

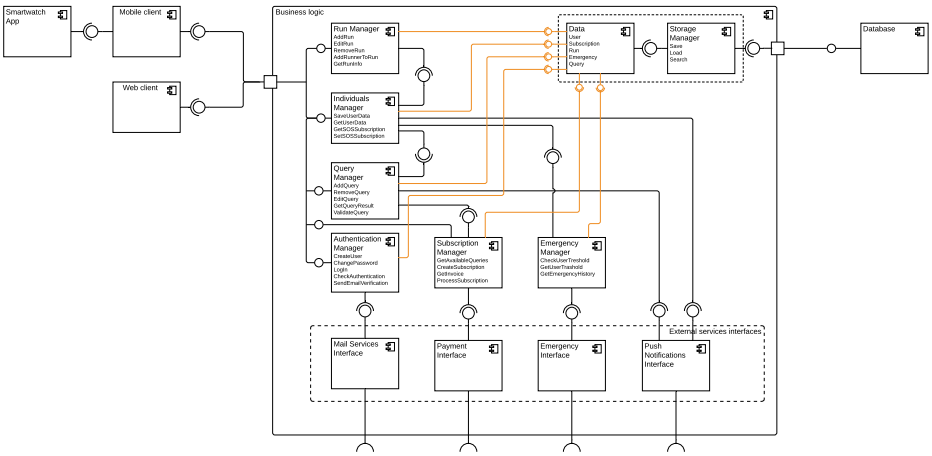


Figure 2: Component Diagram

The picture shows the components of the application divided in Client, Business Logic and Data tier.
Below are described more in detail all the components and interfaces that the system uses to offer its functionalities.

Smartwatch device
The Smartwatch device is directly connected to the mobile app of the user and does not interact with the business logic. Data coming from the sensors of the smartwatch are sent to the mobile app via bluetooth.

Mobile client and Web client
The clients of the system.

- The Mobile client is used by individuals that want to exploit the services of Data4Help.
- The Webclient is used by Companies that want to exploit the query services of Data4Help.

Individuals Manager
The Individuals manager has to

- notify the system when there are new data available and store them into it;
- manage user requests for historical data, accessing Data4help database;

- ask the user if he agrees to be monitored by a company that has requested an individual query, sending him a push notification;
- communicate data to Emergency manager (only if user is subscribed to AutomatedSOS);

Authentication Manager
The component that manages users log-in and registration and is in charge of:

- allowing user login;
- changing user authentication information;
- adding new user to database;
- communicating with Email interface in order to send verifications request to the user

Query Manager
The core component of the Business unit. It is in charge of:

- **Get query results**;
- **Validating queries**: verify if there is a sufficient number of users involved in the scope of the query (in case of a query on group of individuals), and if a company is allowed to do a query according to its payment subscription;
- **Adding queries** to a company account;
- **Deleting queries** from a company account that wish to unsubscribe
- **Updating queries**, when a company has subscribed to a query and wants to edit its query; interacts with the Push Notification interface to send the "New data" notification.

Subscription Manager
The component is in charge of managing the subscription of companies to a payment plan offered by Data4Help. It has to:

- **Subscribe**: let companies subscribe to new payment plans;
- **Store** in the database subscriptions of every company;
- **Let companies see** their subscription detail;
- **Interact** with the payment service to let company pay for subscriptions.

Emergency Manager
The component is in charge of

- **Evaluating** user health parameters;
- **Store** in the database threshold parameters for users subscribed to AutomatedSOS;
- **Checking** when a parameter of a user goes below the threshold. If so, the component is in charge to communicate with ambulance API throughout the Emergency interface.

Run Manager
The component is in charge of:

- **Creating races**: allows run organizers to create new races;
- **Listing run**: provides a list of runs be available to users;
- **Adding** new runners to a race;
- **Managing**: let run organizers manage a race's information.

External services interface
The system has 3 interfaces components that are in charge of communicating with the external services API. The interfaces are:

- Push Notification Interface: communicates with a push notification server API in order to send notifications to mobile client;
- Emergency Services Interface: communicates with Ambulance API provided by Hospitals in order to send an ambulance if the user health parameters go below thresholds;
- Mail Services Interface: communicates with email server to send email.
- Payment Interface: communicates an external payment service in order to create payment requests and validate them.

The diagram below describes in details the components and the entities of the application server.

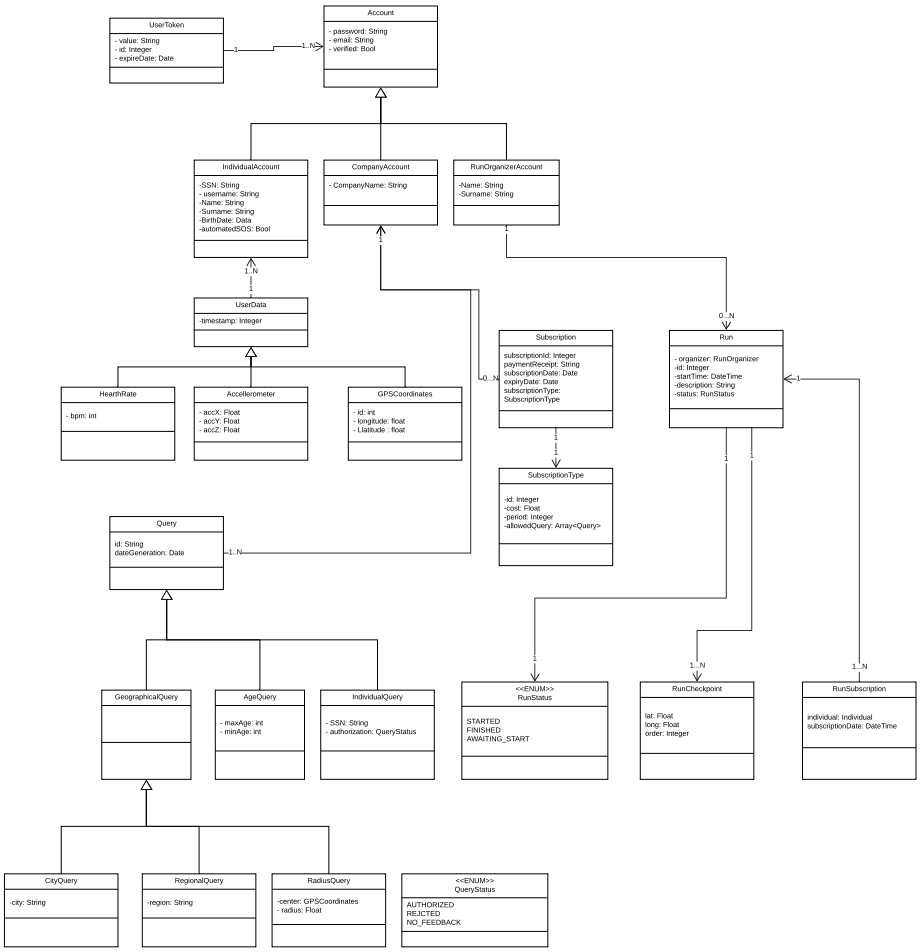


Figure 3: Entities diagram

2.3 Deployment view

The diagram shows the deployment of the system and its structure.

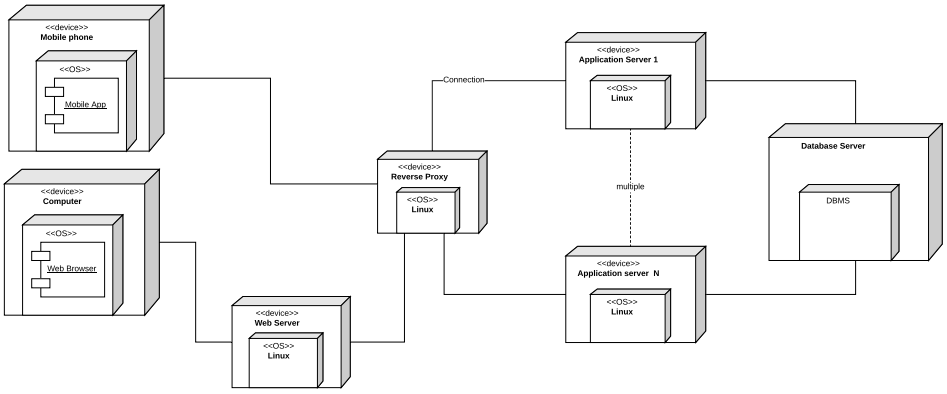


Figure 4: Deployment diagram

- **Client**: represents the first layer. We distinguish between 2 typologies of clients: Web Browser and Mobile app.
- **Server**: is part of the second layer. We distinguish:
  - Application server: contains all the business logic of the system.
  - Web Server: contains the data concerning to the web pages
- **Reverse proxy**: is part of the second layer of the system. Is a type of proxy server that retrieves resources on behalf of a client from one or more servers.
- **Database**: is the third layer of the system that stores all the data. The system uses a relational DBMS.

2.4 Runtime view

2.4.1 User registration

Each individual should register an account in order to use the System. The user initially fills a form on its smartphone providing all the required data. The Mobile App also collects the model of the smartwatch.
This data is sent over the internet, using an HTTP POST request to the AuthenticationManager, with all the required data in JSON format. The AuthenticationManager firstly checks if the smartatch is compatible. If it is, it checks that the email and SSN/fiscal code are not used by any individual already registered. If all the check pass an Individual and an Account are created in the database.
The account initially is not confirmed.
Then, a mail is sent to the email provided, with a verification code. When the user inserts the verification code in the app, an HTTP POST request is

made to the System and the user, if the code is correct, is put in the active state.

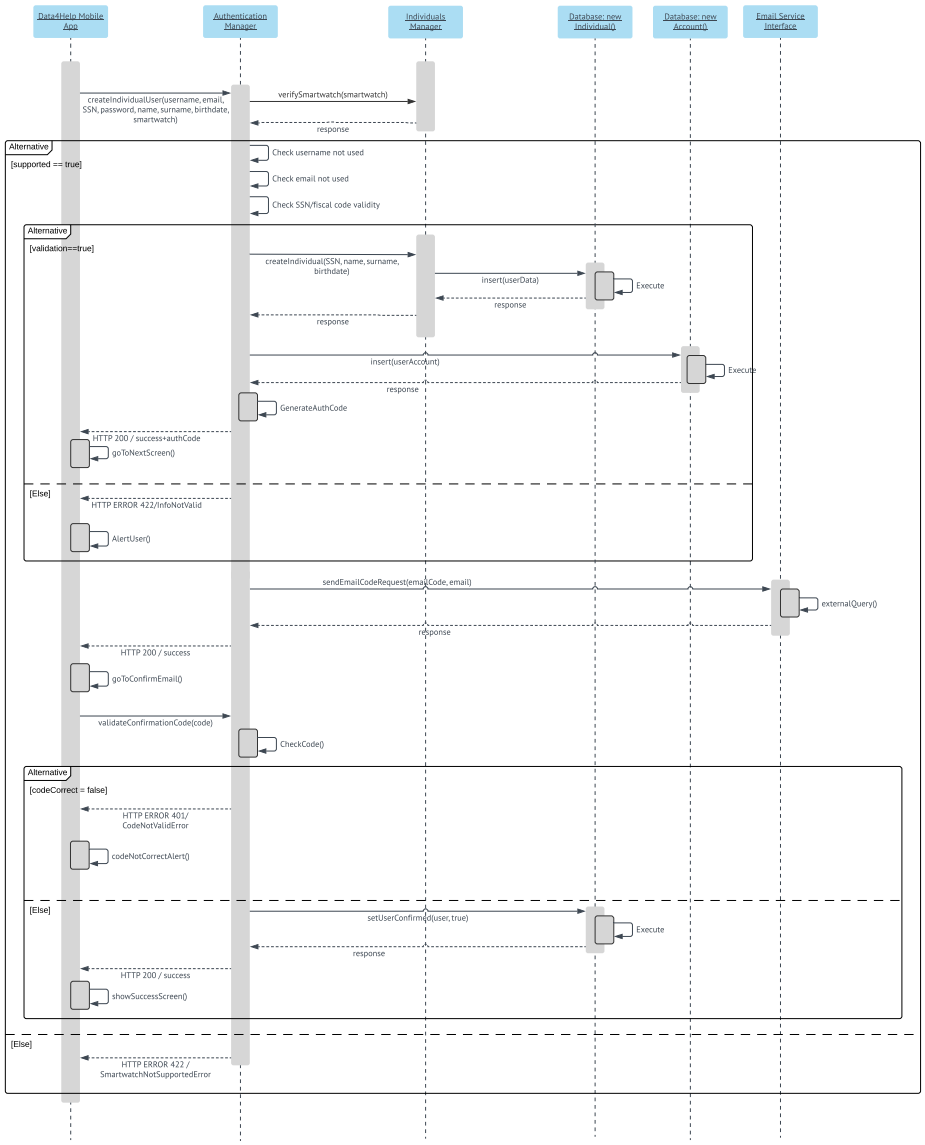


Figure 5: User registration Runtime View

2.4.2 General Login

When a generic user (one among Individual, Company or Run organizer) wants to login, an HTTP POST request is made to the authentication manager. It checks if the username and password combination are correct.
If they are, it generates an authToken, stores it in the database and then

returns it to the caller. The caller can use this authentication token for each subsequent call; the token will remain valid for 24 hours.

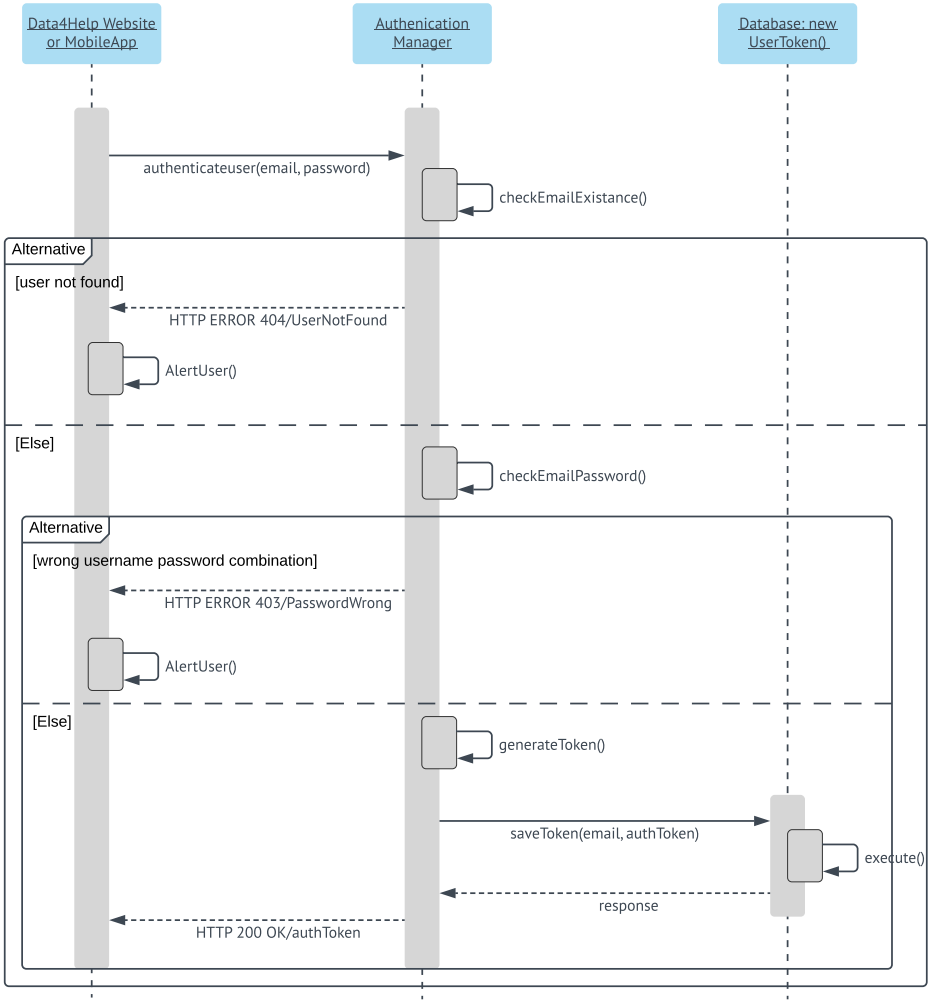


Figure 6: General Login Runtime View

2.4.3 Consulting Individual Activity History

When an Individual wants to look at its activity history, the app makes an HTTP GET request to the IndividualManager, including, beside the auth code, the begin date, the end date and the parameter type that he wants to get.
After checking the token, the IndividualManager gets the SSN and then looks for the data requested in the database. Then it returns the data found to the caller.

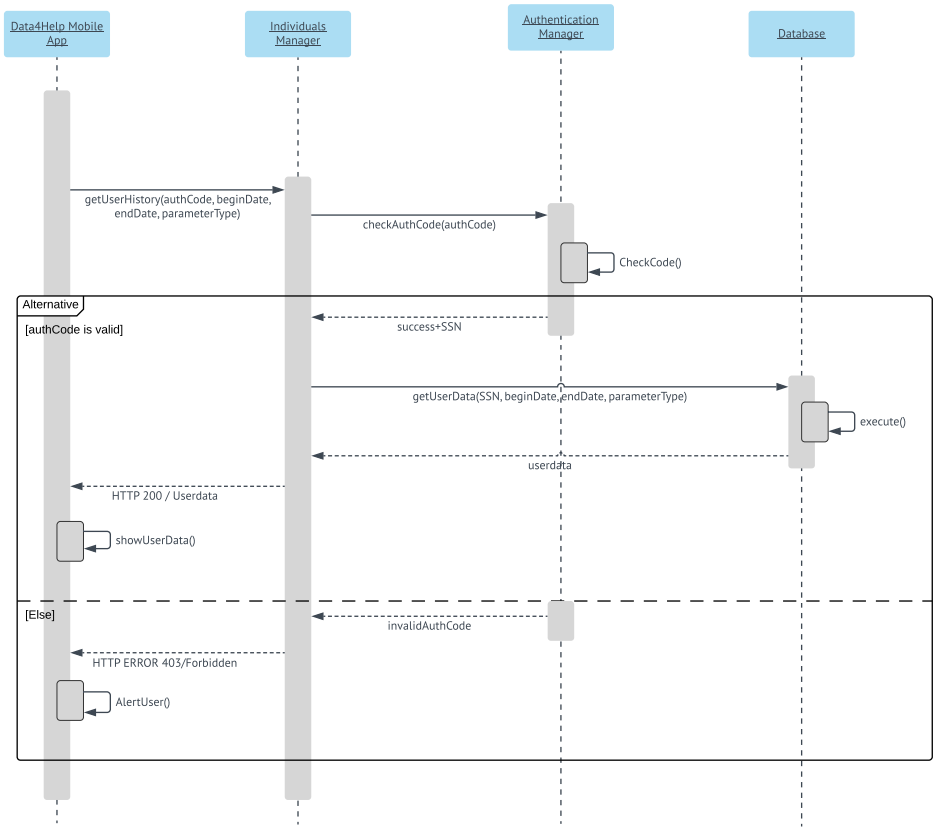


Figure 7: Consulting Individual Activity History Runtime View

2.4.4 Data Synchronization

When the bluetooth is activated, the Mobile App tries to enstabilish a connection with the Smartwatch App. When the connection is enstabilished, the Mobile App tries to get new data from the Smartwatch. If there is new data, it gets it and send it to IndividualManager through an HTTP POST request, alongside with the authToken perviously obtained. If the authToken is correct, the IndividualManager saves the data and then makes a call to checkUserTreshold in EmergencyManager and notifyCompanies in QueryManager.

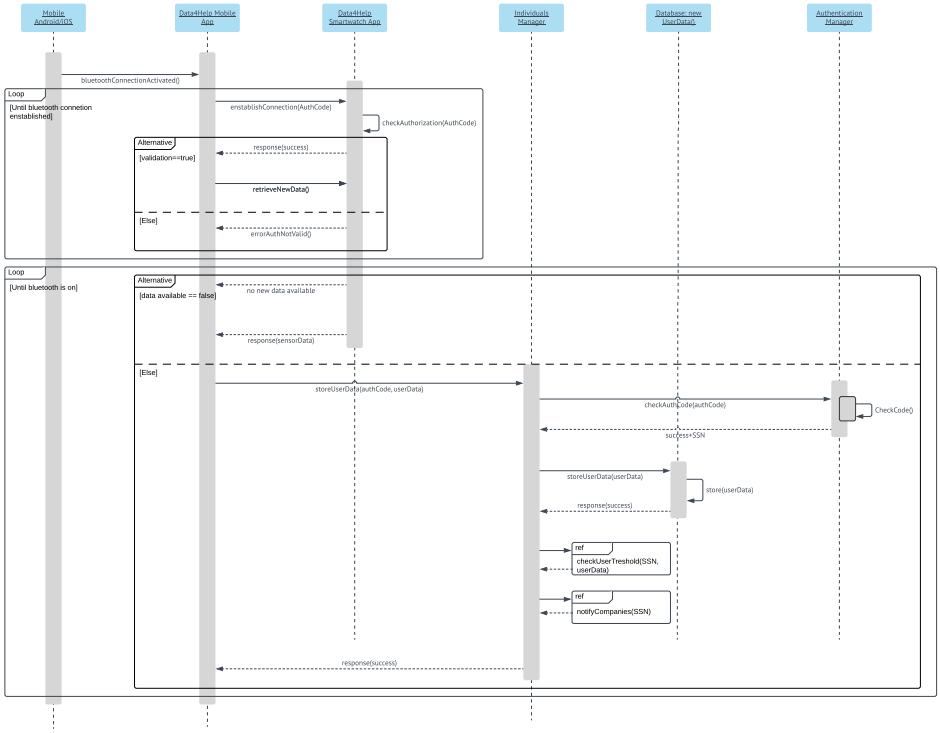


Figure 8: Data Synchronization Runtime View

2.4.5 Company Registration

The flow is the same as per the user regstration, excluding that the smartwatch is not checked and the user is only created in the AuthenticationManager database.

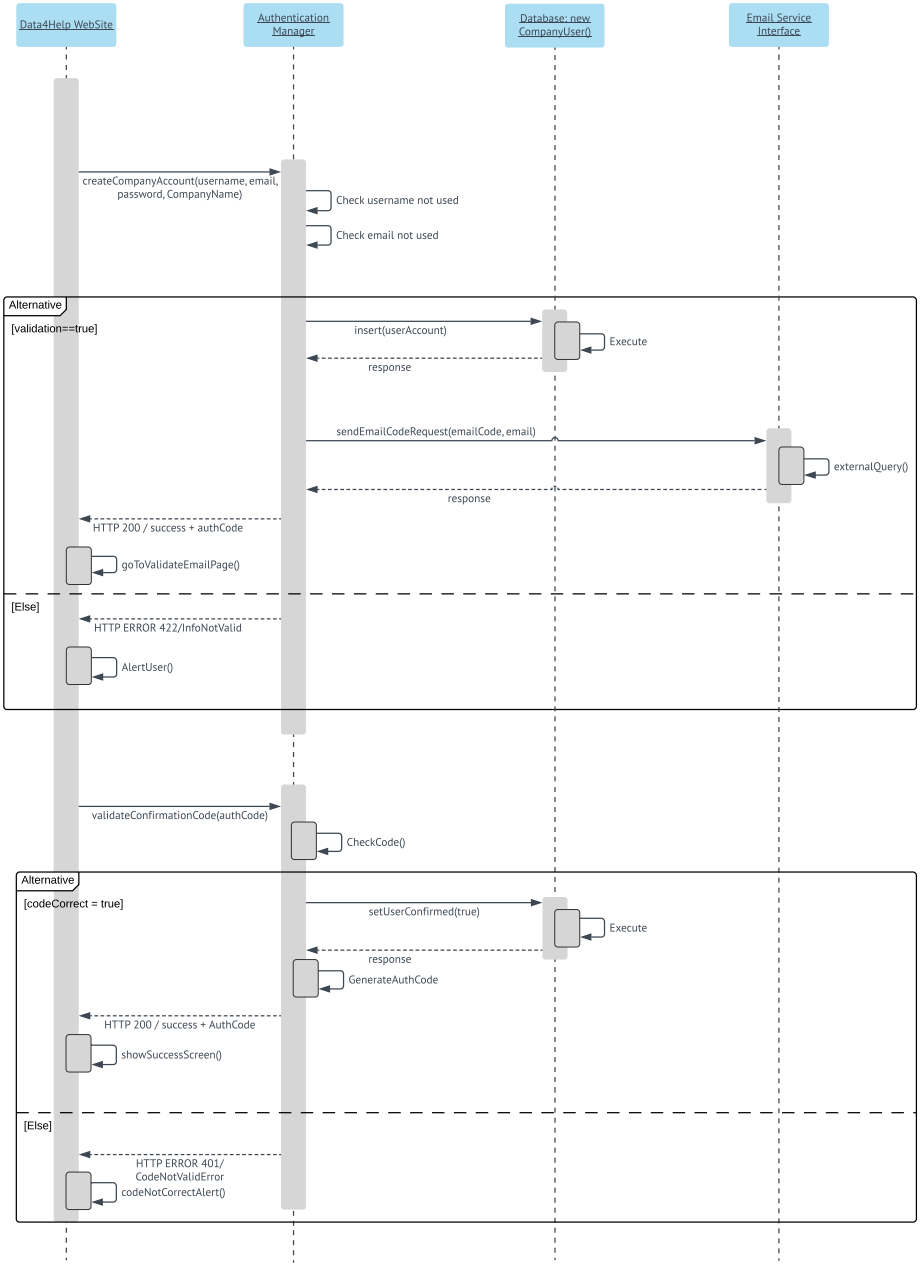


Figure 9: Company Registration Runtime View

2.4.6 Company Query Search Multiple Individuals

When a Company wants to make a query over an anonymized group of people, it accesses to the website and starts a query. The Website makes an HTTP GET request to the query manager that, after the usual security

check on the authToken, checks with the help of the SubscriptionManager if the company has a plan active that allows it to make this type of query, and return the result.
The Company fills the form and then the Website submits it via an HTTP POST request to the Query manager with the query parameters. The Query-Manager checks again the subscription and after that it compute the number of user involved in the query. If the number is below the treshold (1000) the query is blocked and an error is returned. Otherwise, the query is saved and the OK result is returned.
If the user then wants to download the query result, it can invoke the downloadData method on the Query manager that, after having checked the authorization, returns an XML file containing the result of the query.

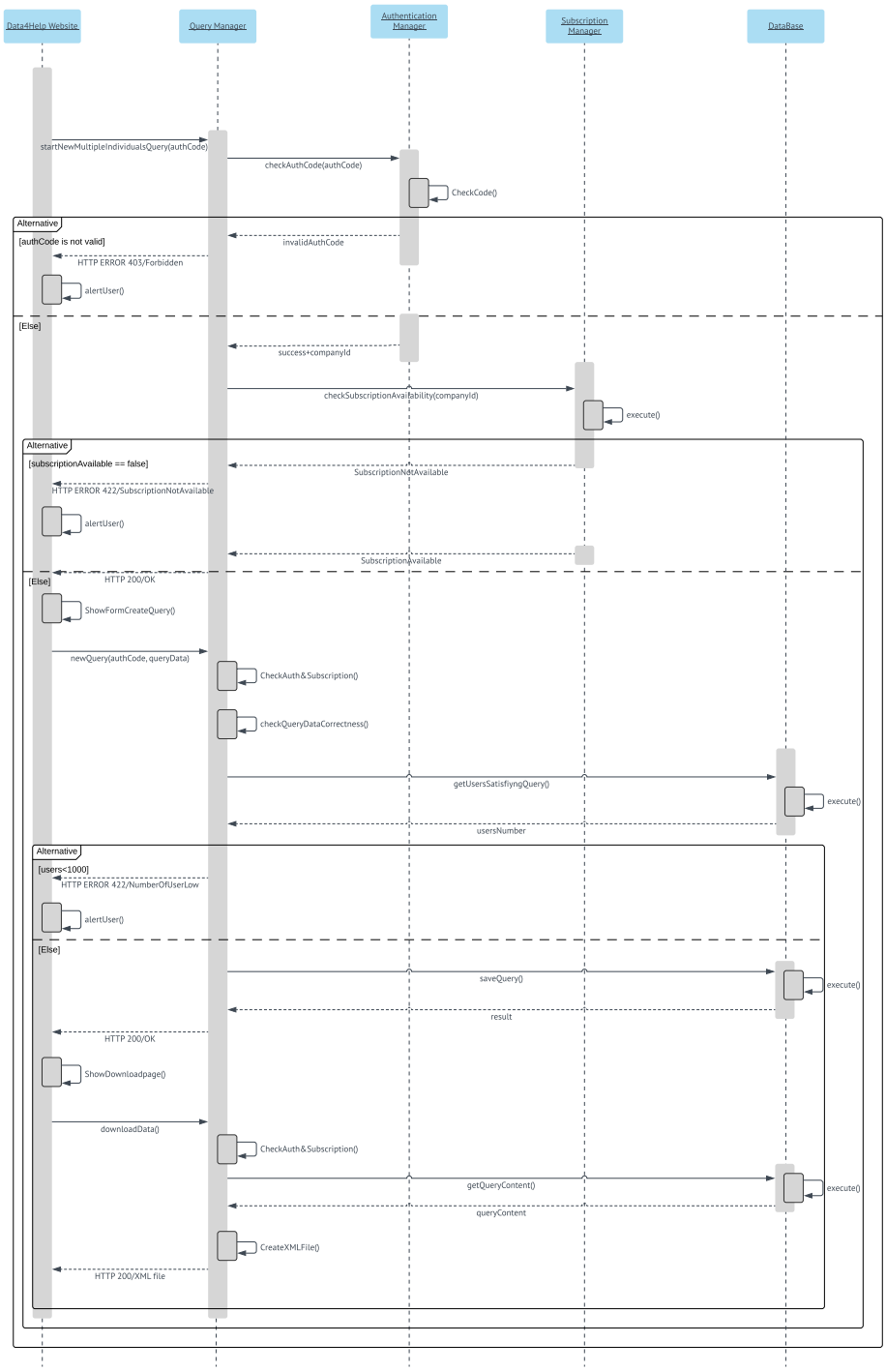


Figure 10: Company Query Search Multiple Individuals Runtime View

2.4.7 Company Request Individual Monitoring

When a Company wants to monitor a single user, it must ask trough the website the permission to the user.
The Website sends an HTTP POST request specifiyng the SSN/Fiscal code to monitor to the query manager, that, after checking the correctness of the authToken and obtained the CompanyId, checks with the help of the SubscriptionManager if the company has a plan active that allows it to make this type of query. (omitted from the graph for the sake of readability)
If the check passes, a new individual with the state noDecision is inserted in the Database.
Then, a notification is sent to the Mobile App instance of the user whose SSN is the object of the query.
When the user receives the notification it can either accept it or refuse it. After the click an HTTP POST request is made to the query manager and the query state is updated accordingly.

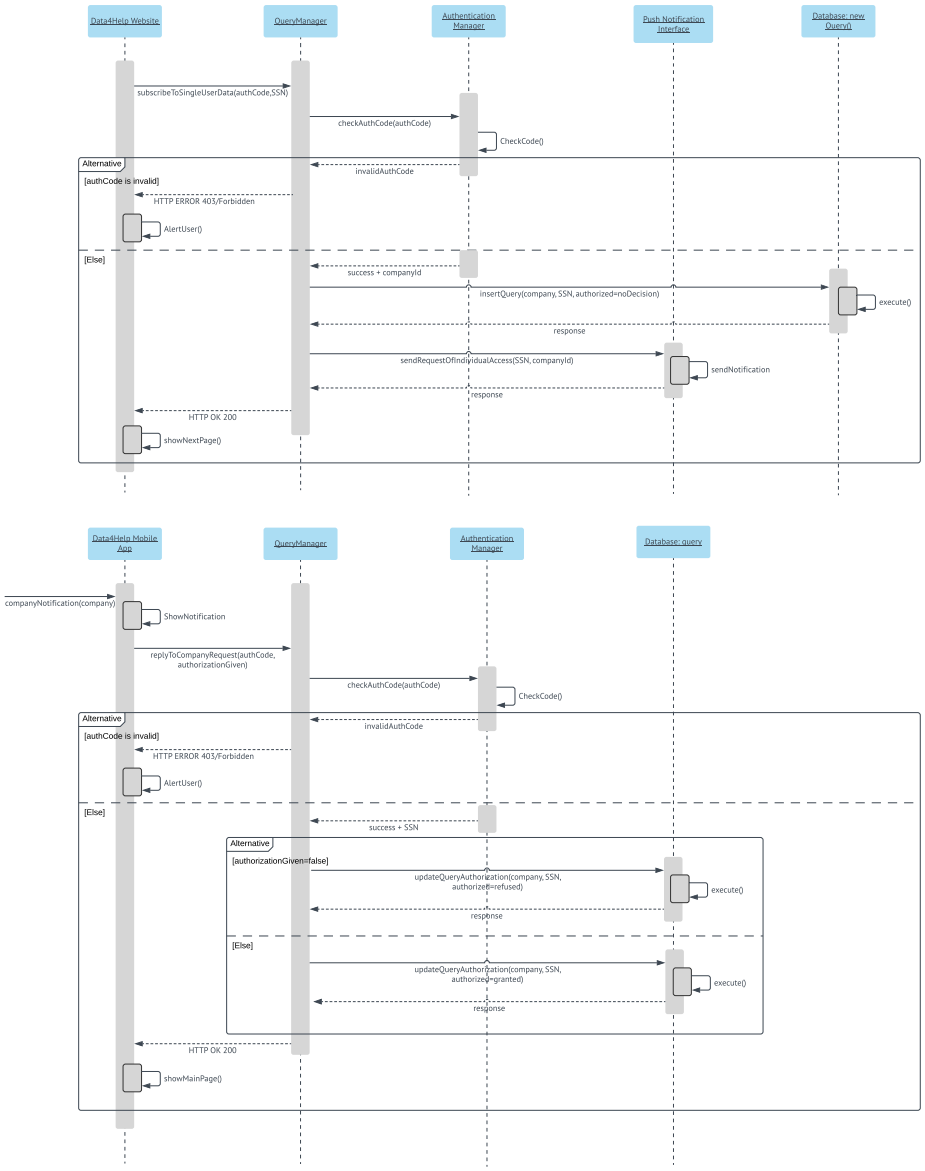


Figure 11: Company Request Individual Monitoring Runtime View

2.4.8 Company Consulting Individual

When a company wants to see an Individual's historical data, through the website, it makes an HTTP GET request to the QueryManager, specifying the SSN, the parameter type and the starting and ending date. The query manager checks the authToken, retrives the companyId and checks if the Company still has a valid plan to make this type of request. If that's indeed the case, the QueryManager checks if the Individual has given Company the

authorization to see its data. If this is the case, the user data is retrieved and returned.

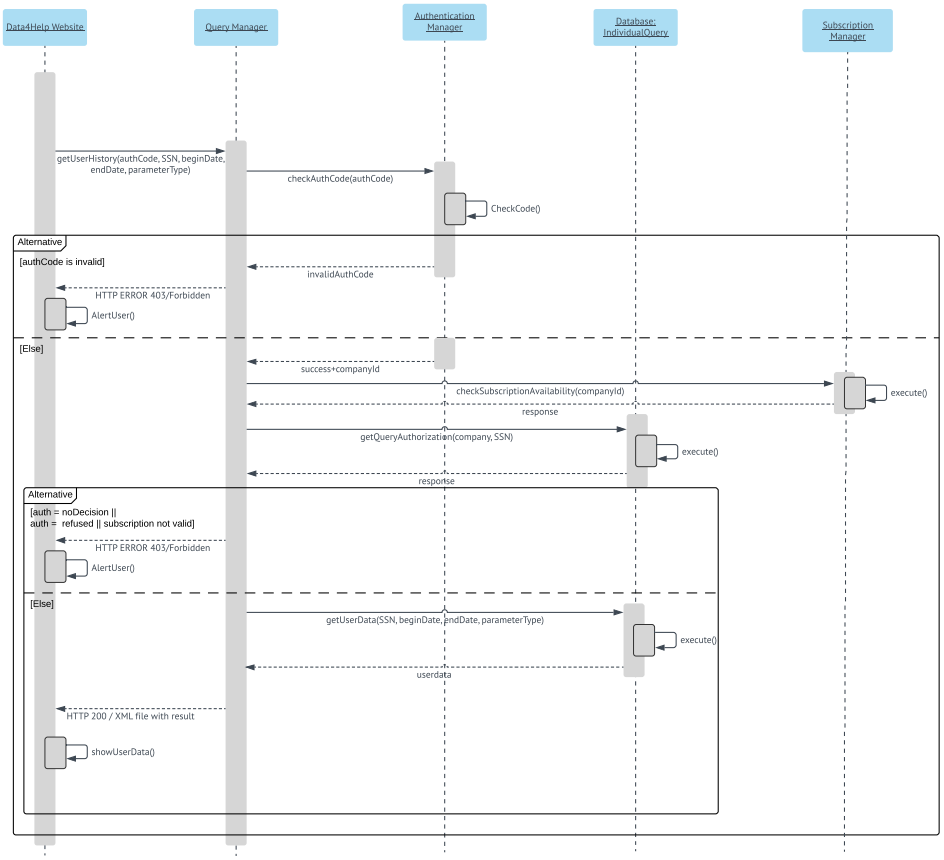


Figure 12: Company Consulting Individual Data Runtime View

2.4.9 Company Payment Processing

When a Company wants to buy a subscription plan, it activates the corresponding functionality in the Website, selects the subscriptionType and insert the credit card number and the CCV/Security code. This data is sent to the server through a HTTP POST request to the SubscriptionManager. After checking the authToken and retrieved the companyId, a call is made to the PaymentInterface to process the payment. If the payment is successfull, the subscription is inserted in the Database and linked to the company.

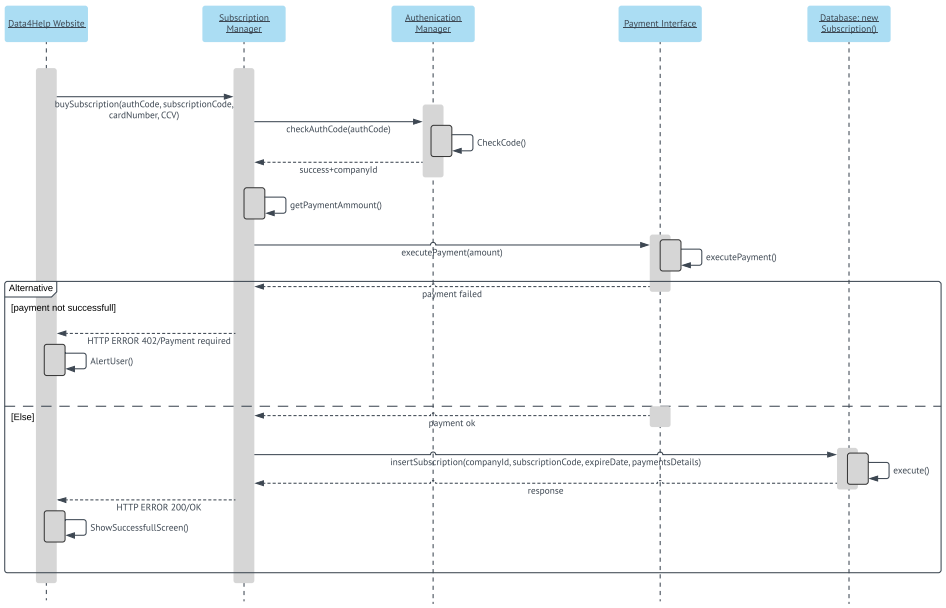


Figure 13: Company Payment Processing Runtime View

2.4.10 Emergency Situation

If a user is subscribed to the AutomatedSOS service, whenever the Individual Manager receives new data from the Mobile App, it also sends a checkUserThresholds request to the EmergencyManager, specifying the SSN of the individual and the health data received. Firstly the EmergencyManager checks if the user is subscribed to AutomatedSOS, if so, it checks the threshold for that specific user: in the case in which the data parameters are lower than the specified threshold, the EmergencyManager instantly sends a sendAmbulance request to the EmergencyInterface, which will contact the ambulance API and requests an ambulance for the person. To do so, in the request includes the user position.

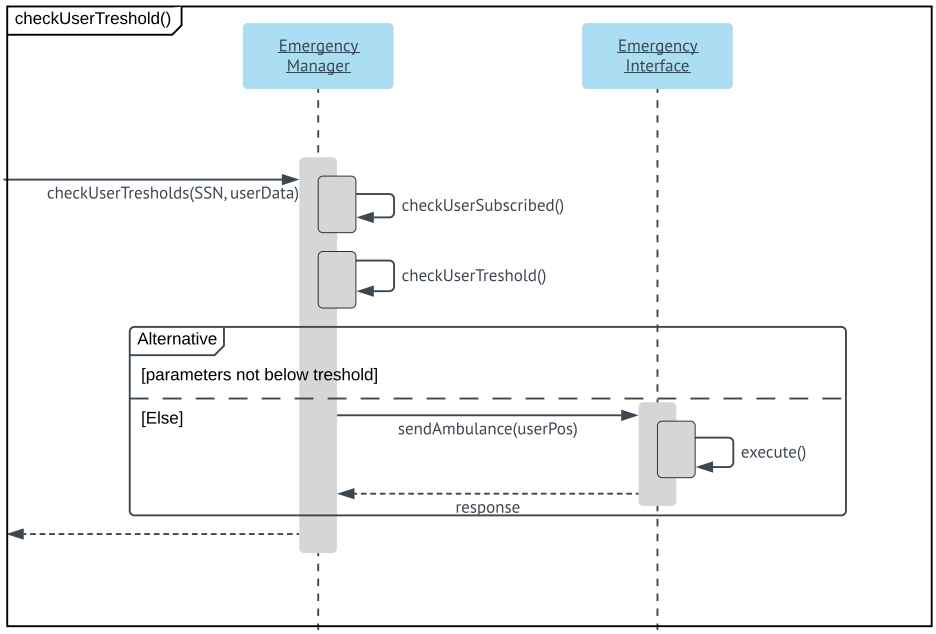


Figure 14: Emergency Situation Runtime View

2.4.11 Run Organizer Adds New Race

When a run organizer wants to add a new race using the Data4Help Mobile App, he goes to the corresponding section, fills information about the run and clicks on "Create new run".
The website sends a createNewRun request specifying the authToken and runData, the Authentication manager authenticates the user, and checks data consistency. If data is consistent, the Run Manager goes on inserting the new run into the Track4Me database and returns a successfull response to the client, otherwise it returns an error and alerts the user.

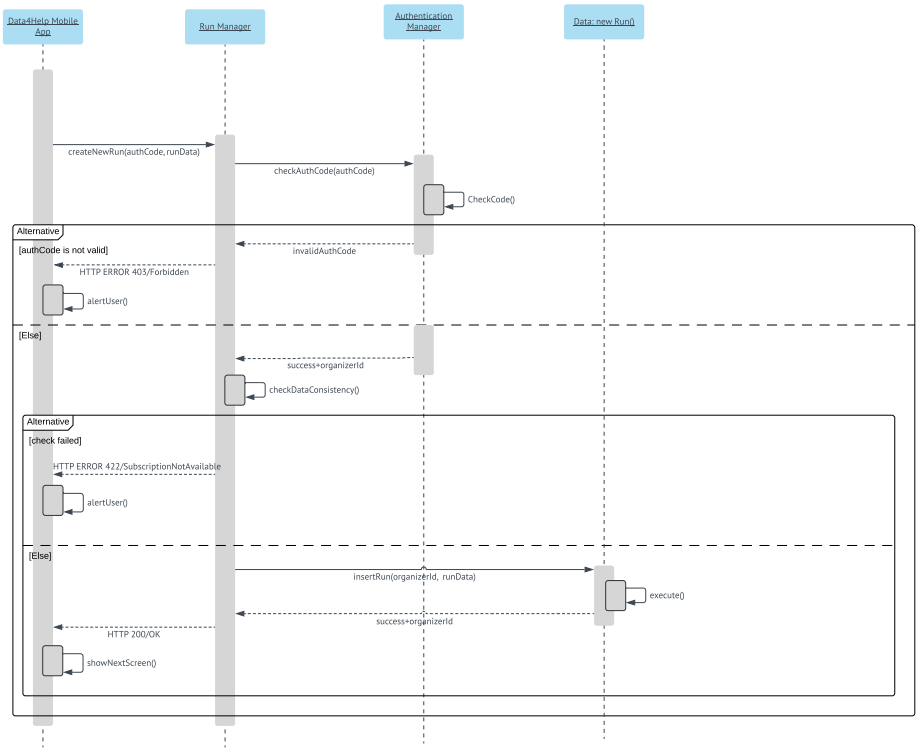


Figure 15: Run Organizer Adds New Race Runtime View

2.4.12 Runner Subscription To A Race

When a user of Data4Help wants to subscribe to a run, it must send a getRunList request containing its geographical position to the RunManager. Firstly, the RunManager computes which races have their starting point 30km close to the runner position, then returns the mentioned list to the user. Then, the user chooses the run he wants to subscribe to and sends a subscribeUser request to the RunManager containing the authToken and the runId.
The user is authenticated by the Authentication Manager, if the sent authToken is correct, the RunManager goes on checking if the chosen race has already started.
If the race has already started, the Run Manager sends an error to the client, otherwise, adds the user to the runner lists sending an addUserToRun to the Run Manager(specifying the user SSN and the runId).

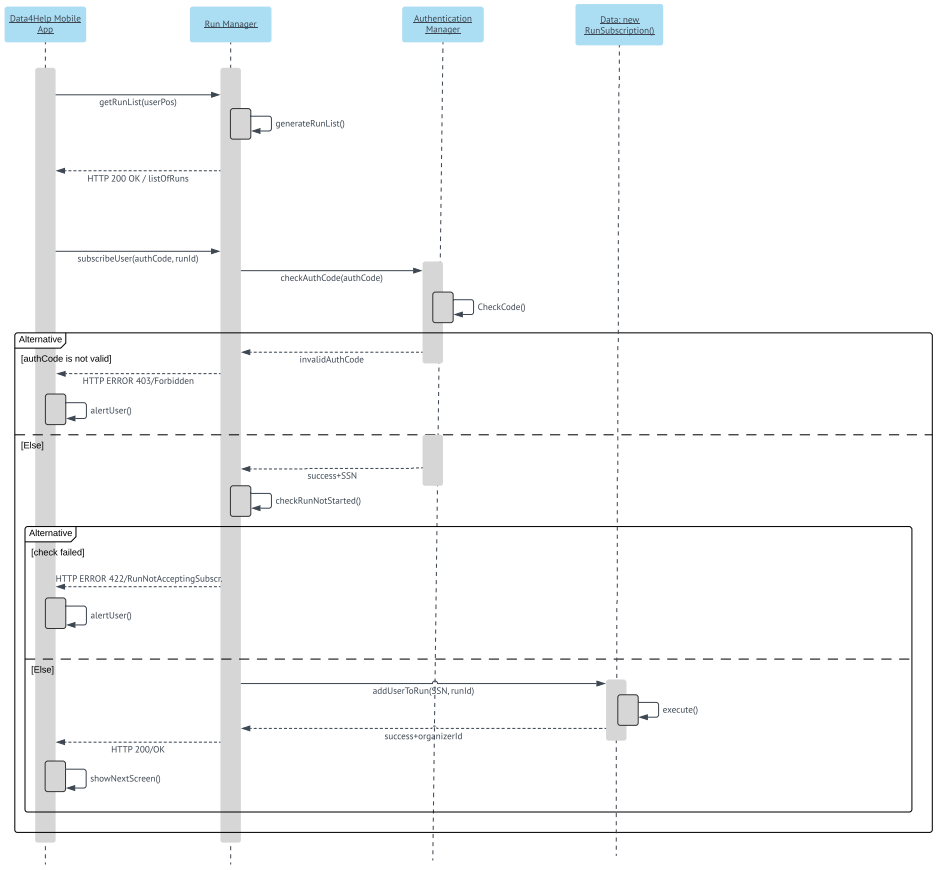


Figure 16: Runner Subscription To A Race Runtime View

2.4.13 Spectator Of Run Request Run Position

When a user wants to see the map of the runners of a run, he must request the runner positions to the RunManager.
The Mobile app sends an getRunnerPositions request specifiyng the code of the run followed to the RunManager that, after checking if the run is still active, get all the users position in run identifying them with the SSN.
Then, it sends a getUserPosFromDatetime request, specifying the runStart-Datetime, it computes if the user has passed checkpoints (computeCheck-Point), eventually add the information of the user in a list.
Finally, the userList with the data about checkpoint passed and the geographical position is returned to the Mobile App, that shows in a map all runner positions.

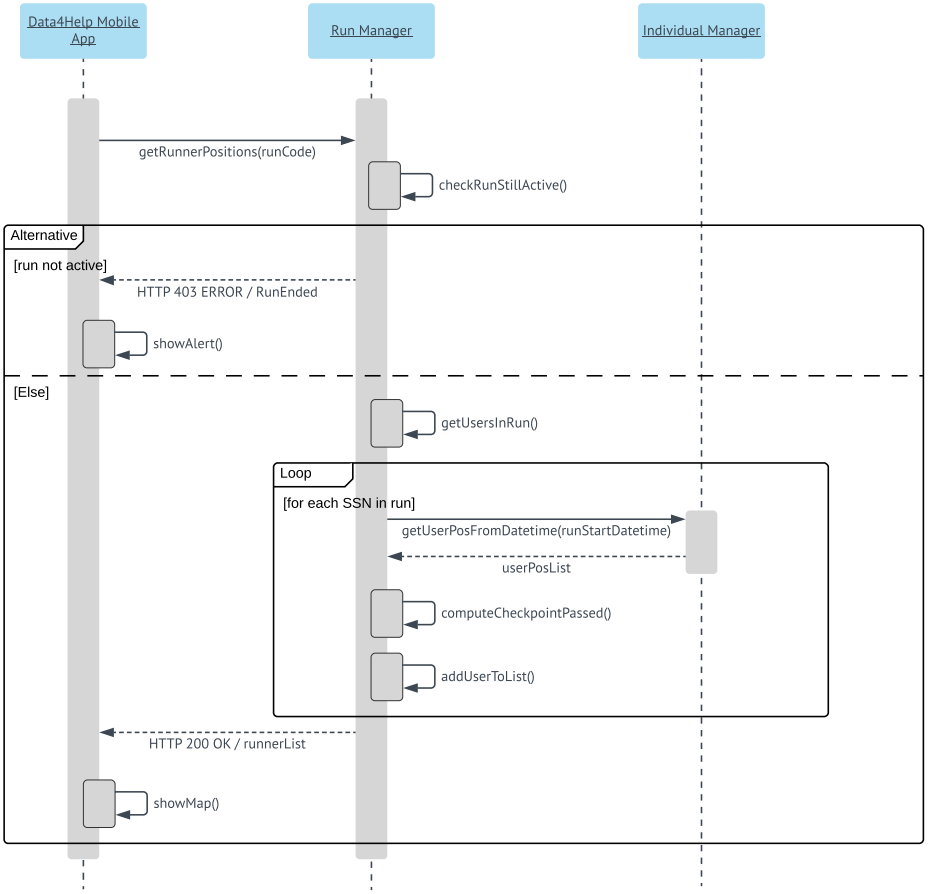


Figure 17: Spectator Of Run Request Run Position Runtime View

2.5 Component interfaces

2.5.1 REST API

Authentication Manager

- User Registration

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

	birthday: Date smartwatch: String

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

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

Uses	Allows the client to register a new User

- Company Registration

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

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

Error Response	code: `422 UNPROCESSABLE ENTITY` content:

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

Uses	Allows the client to register a new Company

- Run Organizer Registration

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

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

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

Uses	Allows the client to register a new run organizer

- User Login

Endpoint	/auth/login

Method	**POST**
URL Params	
Request Data	email: String password: String type: String

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

Error Response	code: `404 NOT FOUND`
	content:
	`{`
	`success: false,`
	`error: 'UserNotFound' ,`
	`message: 'User does not exists'`
	`}`
	
	code: `403 FORBIDDEN`
	content:
	`{`
	`success: false,`
	`error: 'InvalidCredentials' ,`
	`message: 'Invalid Credentials'`
	`}`

Uses	Allows the client to login

- Verify mail

Endpoint	/auth/verify
Method	**GET**
URL Params	mail: String code: String type: String
Request Data	
Success Response	code: `200 OK` content:

	{
	success: true,
	message: email verified
	}

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

Uses	Allows verification of the account

Individuals Manager

- Data store

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

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

Error Response	code: 422 UNPROCESSABLE ENTITY
	content:
	{
	success: false,
	error: 'InvalidData',
	message: 'Data are invalid'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:

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

Uses	Allows the client to store sensor data in the database

- Data retrival

Endpoint	/indiv/data
Method	**GET**
URL Params	auth_token: String begin_date: Date end_date: Date
Request Data	

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

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

Uses	Allows the client to retrive sensor data from the database

- User information retrival

Endpoint	/indiv/user
Method	**GET**
URL Params	auth_token: String
Request Data	
Success Response	code: 200 OK content:

	{
	success: true,
	user: ...userInfo
	}

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

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

Uses	Allows the client to retrive user informations

Query Manager

- Query creation

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

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

Error Response	code: 422 UNPROCESSABLE ENTITY content:

	{
	success: false,
	error: 'QueryTooRestrictive',
	message: 'Query on too few users'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'BadQuery',
	message: 'Query is invalid'
	}
	
	--- 	
	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}
	
	--- 	
	
	code: 402 PAYMENT REQUIRED
	content:
	{
	success: false,
	error: 'PaymentRequired',
	message: 'Payment required'
	}

Uses	Allows the client to create a query

- Query Retrival

Endpoint	/queries/query
method	**GET**
URL Params	auth_token: String

Request Data	

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

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

Uses	Allows the client to retrive the queries created by a company

- Perform Query

Endpoint	/queries/query/data
method	**GET**
URL Params	auth_token: String query_id: Integer
Request Data	

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

Error Response	code: `400 BAD REQUEST` content:

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

Uses	Allows the client to perform a query

- (still) Unauthorized queries retrieval

Endpoint	/queries/query/individual/pending
method	**GET**
URL Params	auth_token: String
Request Data	

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

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

Uses	Get unapproved individual querie

- Allow/Negate individual query

Endpoint	/queries/query/individual/pending
method	**POST**
URL Params	
Request Data	auth_token: String query_id: Integer decision: Boolean
Success Response	code: 200 OK

	content:
	{
	success: true,
	message: 'Response Saved'
	}

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

Uses	Specify decision for individual query

Subscription Manager

- Buy subscription plan

Endpoint	/subs/plan
Method	**POST**
URL Params	
Request Data	auth_token: String mail: String plan: String

Success Response	code: 200 OK
	content:
	{
	success: true,
	message: {$PLAN} bought
	}
	--- 	
	code: 400 BAD REQUEST
	content:
	{
	success: false,
	error: 'InvalidToken',
	message: 'Token is invalid'
	}
	--- 	

	code: `402 PAYMENT REQUIRED`
	content:
	`{`
	`success: false,`
	`error: 'PaymentRequired',`
	`message: 'Payment required'`
	`}`
	
	code: `404 NOT FOUND`
	content:
	`{`
	`success: false,`
	`error: 'PlanNotFound',`
	`message: 'Plan not found'`
	`}`

Uses	Allows the client buy a subscription to a new plan

- Get plan informations

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

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

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

	code: `404 NOT FOUND`
	content:
	`{`
	`success: false,`
	`error: 'PlanNotFound',`
	`message: 'Plan not found'`
	`}`

Uses	Allows the client retrive informations about a subscription plan

Run Manager

- Create a Run

Endpoint	/runs/run
Method	**POST**
URL Params	
Request Data	auth_token: String time_begin: Date time_end: Date description: String coordinates: Array < JSON >

Success Response	code: `200 OK`
	content:
	`{`
	`success: true,`
	`run_id: $ID`
	`}`

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

	code: `403 FORBIDDEN`
	content:
	`{`
	`success: false,`
	`error: 'InvalidCredentials',`
	`message: 'Invalid Credentials'`
	`}`

Uses	Allows the client to create a new run

- List all available runs

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

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

Error Response	code: `401 FORBIDDEN`
	content:
	`{`
	`success: false,`
	`error: 'InvalidCredentials',`
	`message: 'Invalid Credentials'`
	`}`

Uses	Allows the client to list all runs satisfying the parameters above

- Join a run

Endpoint	/runs/join
URL Params	
Method	**POST**
Request Data	auth_token: String run_id: String

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

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

Uses	Allows the client to join a run

- Get the positions of runners in a run

Endpoint	/runs/positions
Method	**GET**

URL Params	run_id: String auth_token: String
Request Data	

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

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

Uses	Allows the client get the position of a runner in a run

2.6 Selected architectural styles and patterns

In this section the main architectural decision are explained to better specify how a specific decision may affect the system to be.

2.6.1 Multitier architecture

The client-server architecture is implemented using a multitier architecture. In this way the Business Logic, the presentation and the Data Storage tier are maintained separated and isolated. Therefore, the application remains flexible and every tier can be implemented, tested and modified separately from the others.
In particular:

- *Presentation tier:* this is the only part of the system-to-be accessible directly from the user. Its purpose is to translate action of the user into action and requests to the System and translate back the response of the System into something human readable.
- *Web tier:* the purpose of this layer is to provide to the Company Web browser the web pages. The server does not have a connection with the Business Logic tier because it hasn't the need to make requests directly. Indeed, the web application will make the necessary requests directly to the Business Logic tier, modifying the webpage directly.

- *Business Logic tier:* this is the central tier as it contains the logic of the application. It is responsible of processing the user requests and saving data into the database. In the architecture proposed, this tier contains also the reverse proxy, a component that is needed in order to balance the user-generated network traffic to the multiple servers.
- *Data storage tier:* this component is actually a database; its purpose is to maintain the state of the data and ensure it persistence.

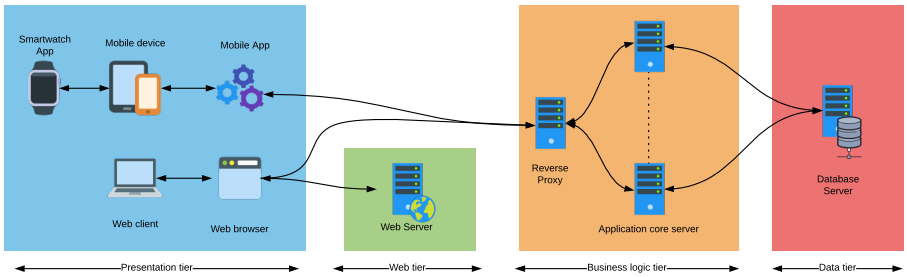


Figure 18: Multitier view of the system

2.6.2 Thin client

The system will use a Thin client paradigm that will handle only communications with the main server. Using a Fat client paradigm is not convenient in this case because our customers rely mostly on Mobile phones which can have troubles in handle computational effort. The server of Data4Help will have sufficient computational power to manage a lot of user requests at the same time.

2.6.3 RESTful

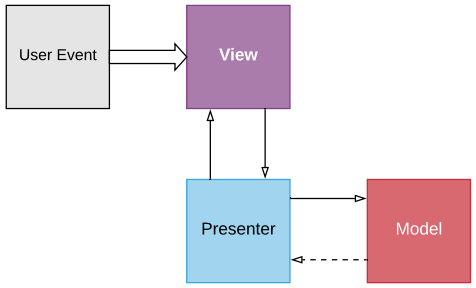
The communication between the Core and the smartwatch App / Web site is made using a RESTful (Representational State Transfer) service, that means that all the query are simple HTTP query, so a security layer can be easily added using SSL/HTTPS. Moreover, the RESTful was formalized with scalability in mind, so that a web service implemented using a RESTful architecture can be easily expanded to fulfill new needs. Being a simple HTTP request, it is compatible with all firewall/proxy, simplifying the configuration aspects of the system, and as the response body JSON can be used, a fully standardized and powerful but simple language to exchange data.

2.7 Other design decisions

2.7.1 Model View Presenter

The structure imposed by the chosen multi-tier architecture suggests the use of the **MVP** (Model-View-Presenter) architectural pattern.
Here are briefly presented the concept behind the **MVP** architecture, with respect to the component used in the service.

- **Model**: Handles the communication with the database.
- **View**: Handles the information displayed to the generic user, therefore being, in this situation, completely passive.
- **Presenter**: Serves as a middleware manipulating data exposed via queries by the **Model** in response to events dispatched by a generic user (e.g Login, request for subscription etc)



2.7.2 Stateless

In order to easily serve more or less users according to the demand of the service it has been opted for a stateless *Business Logic Tier*. This allows the service to be scaled when needed, therefore saving resources when requests are low.

3 User interface design

As shown in the UX Diagram below, the user must first select if he is a *Run Organizer* or a *Normal User*, then he can proceed in choosing wheather he wants to login or register.
The *Login* and *Registration* flow are similar, being associated with similar procedures. The only difference is that, due to the fact that a SmartWatch is required for the *Normal User* to use the app, the presence of it is checked during *Login* and *Registration*.
If the procedures mentioned before are successful the user lands on the *Main Screen*, on which a Navigation Drawer presents him the *Navigation Options*.

If the User is a *Run Organizer* the options are the following:

- View All Runs
- Organize a Run

If the User is a *Normal User* the options are the following:

- Health Graph
- All Runs Available
- AutomatedSOS

If the *Run Organizer* selects the *View All Runs* option he is presented with a screen in which all run organized by him are present. Tapping on a run results in a screen with all the information concerning it shown. If the *Run Organizer* selects the *Organize a run* option he is presented with the *Run Organization Form* screen which will be followed by, after submission, the *Run Info* screen.

The *Normal User* has the possibility to:

- View the *Health Graph* screen, by which, tapping on a graph, results in more information concerning the selected health parameters being shown.
- See *All Runs Available*, by which, tapping on a run, results in more informations on the run being show.
- Subscribe to *AutomatedSOS*.

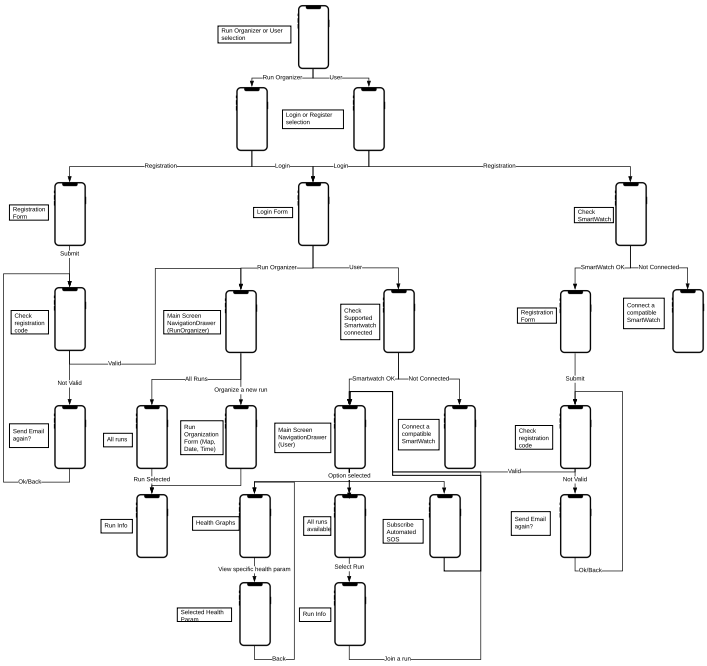


Figure 19: UX Diagram for Website

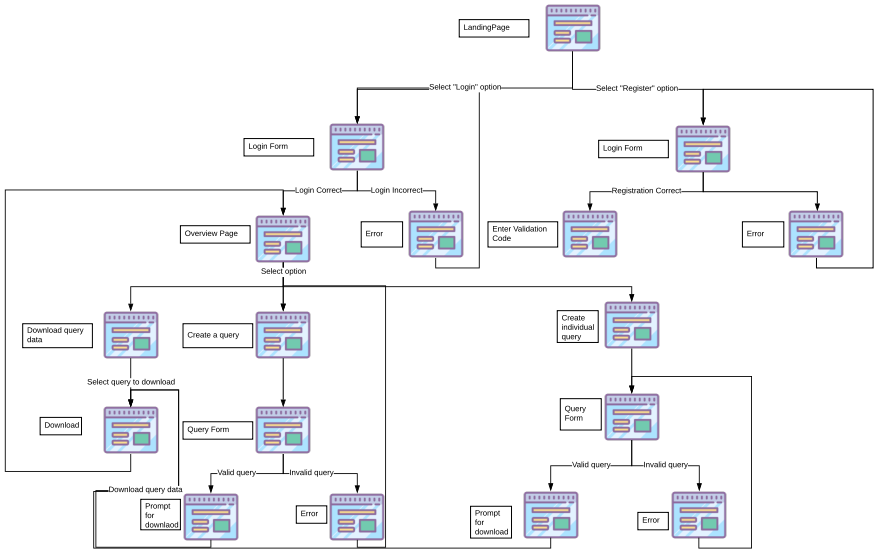


Figure 20: UX Diagram for Mobile App

4 Requirements traceability

In this paragraph is shown the mapping between every component and the requirement satisfied.

- **G1**: The system should be able to show acquired data via the Mobile App and the Website.
  (Requirements **[R1$_S$]**, **[R2$_S$]**)
  - Query Manager
  - Individuals Manager
- **G2**: The system should allow users to register.
  (Requirements **[RM$_M$]**, **[R14$_C$]**)
  - Authentication Manager
  - Individuals Manager
- **G3**: The system should allow companies to register.
  (Requirements **[]R1$_W$]]**, **[R14$_C$]**)
  - Authentication Manager
  - Email Service Interface
- **G4**: The system should allow registered companies to request data from an anonymized group of individuals, only if individuals in the group are more than 1000.
  (Requirements **[R2$_W$]**, **[R2$_S$]**, **[R5$_C$]**, **[R7$_C$]**)
  - Authentication Manager
  - Query Manager
  - Subscription Manager
- **G5**: The system should allow registered companies to request data from an individual person, only if individuals accept the request.
  (Requirements **[R2$_W$]**, **[R6$_C$]**, **[R1$_C$]**, **[R2$_C$]**, **[R3$_C$]**)
  - Authentication Manager
  - Query Manager
  - Push Notification Interface
  - Subscription Manager
- **G6**: The company should be able to pay through the system in order to buy data queries/subscribe to plans.
  (Requirements **[R2$_W$]**, **[R4$_W$]**, **[R8$_C$]**, **[R15$_C$]**)

  - Authentication Manager
  - Subscription Manager
  - Payment Interface

- **G7**: The system should allow a company to subscribe to new data and receive them as soon as they are produced.
  (Requirements **[R6bis$_W$]**, **[R2$_C$]**)
  - Authentication Manager
  - Query Manager
  - Push Notification Interface

- **G8**: The system should be able to monitor user's health parameter.
  (Requirements **[R1$_S$]**, **[R2$_S$]**, **[R4$_C$]**, **[R5$_S$]**)
  - Emergency Manager

- **G9**: The system should be able to react to the lowering of the health parameters below threshold in less than 5 seconds and send the position of the person to the ambulance system.
  (Requirements **[R1$_S$]**, **[R2$_S$]**,**[R5$_C$]**, **[R9$_C$]**,**[R10$_C$]**, **[R11$_C$]** **[R16$_M$]**)
  - Emergency Manager
  - Emergency Interface

- **G10** The system should allow run organizers to register.
  (Requirements **[R11$_M$]**, **[R14$_C$]**)
  - Authentication Manager
  - Email Service Interface

- **G11** If a run organizer is registered, it can define a run i.e. it can define the path that the participants should follow.
  (Requirements **[R12$_M$]**, **[R13$_M$]**, **[R14$_M$]**, **[R15$_M$]**)
  - Authentication Manager
  - Run Manager

- **G12** A user should be able to enroll to a run.
  (Requirements **[R1$_M$]**, **[R8$_M$]**, **[R9$_M$]**)
  - Authentication Manager
  - Run Manager

- **G13** Spectators of a run should be able to see each participant's position on a map.
  (Requirements **[R8$_M$]**, **[R12$_C$]**, **[R13$_C$]**)
  - Authentication Manager
  - Run Manager
  - Individuals Manager
- **G14**: Individuals should be able to log-in.
  (Requirements **[R1$_M$]**)
  - Authentication Manager
- **G15**: Companies should be able to log-in.
  (Requirements **[R2$_W$]**)
  - Authentication Manager
- **G16**: Run organizers should be able to log-in.
  (Requirements **[R12$_M$]**)
  - Authentication Manager

5 Implementation, integration and test plan

As previously mentioned in this document, the System can be divided into three main part: the Core, the Website and the Mobile App + Smartwatch App. More specifically:

- **Frontend components:** contains the Smartwatch App, the Mobile App and the Website.
- **Backend components:** contains the Core with all its components and the DBMS.
- **External components:** contains all the external components, such as the Payment Service, the communication with the Ambulance system and so on.

In order to implement, integrate and test the system, a *Bottom-up approach* will be used. In this way, the different subsystems can be implemented independently one from each other. As the dependencies of each subsystem are developed, the various subsystems can be progressively integrated and tested.
For what concerning the Mobile App and the Website, these can be implemented assuming that the Core is fully working and running, as they interact only using REST api (i.e. a stub server with mockup responses can be used as these are fully formalized in this document). Only when the Core is fully developed and tested an integration test can be made.

The components to be implemented, integrated and tested are, in order:

1. DBMS & Storage Manager;
2. Authentication Manager, along with Mail interface;
3. Individual Manager, along with Push Notification interface;
4. Subscription Manager, along with the Payment interface;
5. Query Manager, with integration with Push Notification;
6. Emergency Manager, along with Emergency interface;
7. Run Manager, with integration with Individual manager.

As the Core integration and testing is finished, the integration between the Website and the Core and the Smartphone App and the Core can be made in this order:

1. Website, Mobile App with Authentication manager;
2. Mobile App with Individual manager;

3. Website with Subscription manager;
4. Website with Query manager;
5. Mobile App with Run manager.

5.1 Sequence of component integration

The diagrams below describe the sequence of the component implementation, integration and testing of the system.

Integration of the backend Firstly all the components will be implemented and tested with unit tests. Database and Storage manager are the firsts to be implemented and tested because they offer methods to access data. Then the integration process will proceed in the following order:

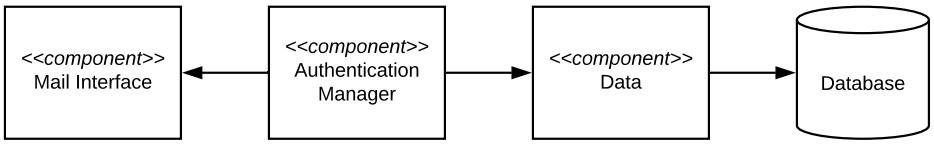


Figure 21: Auth Manager integration

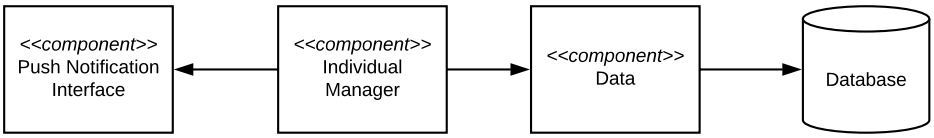


Figure 22: Individual Manager integration

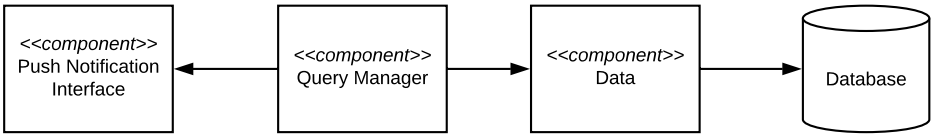


Figure 23: Query Manager integration

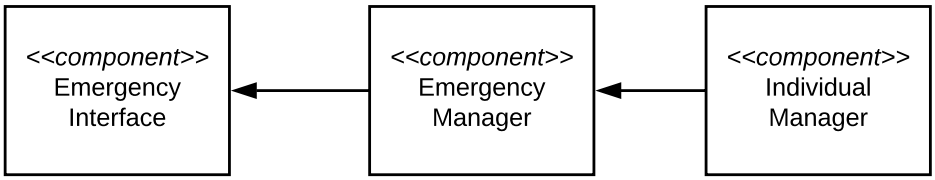


Figure 24: Emergencency Manager integration

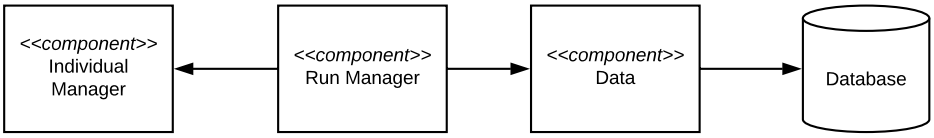


Figure 25: Run Manager Integration

In order to have the main functionality tested, we expect to integrate the external interfaces at the same time with the integration with other internal components.

Integration of the frontend with backend Once all the components of the backend are implemented and tested among each others, the frontend will be integrated and tested within the backend.
The subsystems to be integrated are 2:

- The Website Client, that interacts with the Authentication Manager, Query Manager, Subscription Manager;
- The Mobile app Client, that interacts with the Run Manager, Authentication Manager, and Individual Manager

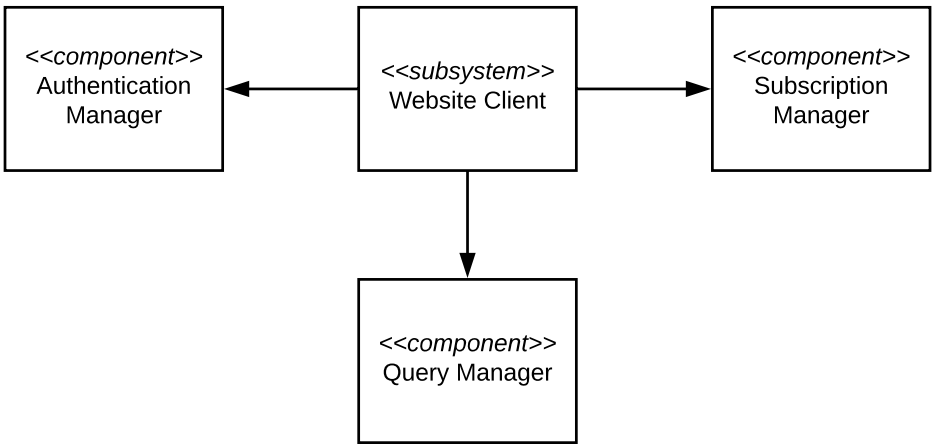


Figure 26: Website integration

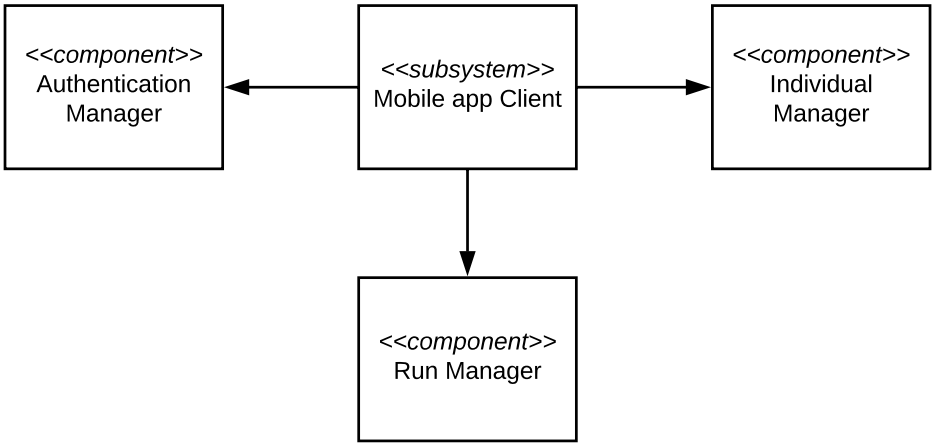


Figure 27: Mobile App integration

Integration of all subsystems Once the front-end has been integrated with the Back-end, the system will be fully tested, with the respect of all the components shown in the Component Diagram.

6 Effort spent

Date	Nicola Fossati	Daniele Montesi	Francesco Sgherzi
13/11/2018	2	0	0
19/11/2018	2	5	3
20/11/2018	3	3	0
21/11/2018	2	2	0
22/11/2018	3	3	0
23/11/2018	4	4	0
26/11/2018	3	3	6
27/11/2018	4	2,5	6
28/11/2018	2,5	2,5	0
29/11/2018	3	2	3
30/11/2018	3	1,5	2
01/12/2018	0	1	5
02/12/2018	2	3	3
03/12/2018	5	3	5
05/12/2018	3	3	3
Total	41,5	38,5	36