POLITECNICO
MILANO 1863

Department of Computer Science and Engineering

# Design Document (DD)

TrackMe

- v1.1 -

*Authors:*

**Avila**, Diego 903988

**Schiatti**, Laura 904738

**Virdi**, Sukhpreet 904204

January 13th , 2019

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

## 1.1 Context

**TrackMe** develops health-monitoring devices devoted to measure and record different parameters related to the health status of a person (i.e. body temperature, blood pressure, heart pulse rate and percentage of $O_2$ in the blood) and also their location. TrackMe health smartwatches are synchronized with an app that gives users access to their data and stats. Also, TrackMe is offering new services to their customers, so as to exploit the data collected from those devices.

## 1.2 Purpose

The requirements elicitation and analysis activities concerning the whole TrackMe system are presented with detail in the RASD (see **References** section). Then, the purpose of this document is to discuss more technical aspects regarding architectural and design choices that must be made, so as to follow well-oriented implementation and testing processes.

More precisely, the document presents:

- Overview of the high level architecture
- The main components and their interfaces provided one for another
- The runtime behavior
- The design patterns
- Implementation plan
- Integration plan
- Testing Plan

.

## 1.3 Scope

The TrackMe environment is composed by three systems whose scope can be summarized as follows:

**D4H** is a system capable to provide third parties the health data collected from individuals that wear TrackMe devices. This is, third parties can request data of a single individual (who can accept or reject it), and also of groups of users. The data is available only under certain conditions and is being retrieved anonymized.

Additionally, **D4H** sends invitations to individuals older than 60 years that live in certain cities, offering them a personalized 24/7 monitoring service called **ASOS**. To establish which cities are available, D4H requests them to ASOS those cities in which there is at least one health care service that has a contract with TrackMe. If a user accepts to activate this service, D4H sends his basic information and last measured health status to ASOS. After receiving the data, ASOS stores it and assigns to the user an emergency contact according to his address.

The system monitors individuals by comparing their health status with previously defined thresholds. If any of the parameters of a user is out of its normal range, a notification will be send to his associated health care service by means of a predefined communication interface within the next five seconds. The health care service reaction time cannot be controlled by ASOS since it is out of the scope

Finally, with **T4R** run organizers are able to setup runs, define their running circuit and send invitations to D4H users of their interest (i.e. according to some criteria), inviting them to participate in upcoming runs. The spectators are able to follow the participants' location using the T4R website. The location of every participant will only be available during the race.

## 1.4 Definitions, Acronyms, Abbreviations

### 1.4.1 Definitions

- **Health status**: Collection of the last measured overall physical health parameters of a user or a group of users.
- **Running circuit**: Path defined by the organizer for the run, using the set of nodes.

- **Anonymize**: The action of anonymize means that an individual's identity cannot be inferred using the available data.
- **Parameter out of its normal range**: Meaning that the parameter is under or above a defined threshold.
- **Filter**

### 1.4.2 Acronyms

- DD: Design Document
- RASD: Requirement Analysis and Specification Document
- D4H: Data4Help
- ASOS: AutomatedSOS
- T4R: Track4Run
- GUI: Graphical User Interface
- MVC: Model View Controller

### 1.4.3 Abbreviations

- $[Rn]$: n-requirement.

## 1.5 Revision history

It is important to keep track of the revisions made to this document:

| Version | Last modified date | Comments |
|---|---|---|
| 1.0 | $11^{\text{th}}$ November, 2018 | |
| 1.1 | $13^{\text{th}}$ January, 2019 | *(a)* |

Table 1.1: Revision history timeline

*(a)* Update regarding diagrams looking forward to be precise with the actual implementation of the system.

## 1.6 Document structure

This document is divided in seven parts, each one devoted to approach each one of the steps required to apply requirements engineering techniques.

- Chapter 1 gives an introduction of the design document. It contains the purpose and the scope of the document, as well as some abbreviation in order to provide a better understanding of the document to the reader.
- Chapter 2 deals with the architectural design of the application. It gives an overview of the architecture and it also contains the most relevant architecture views: component view, class view, deployment view, runtime view and it shows the interaction of the component interfaces. Some of the used architectural designs and designs patterns are also presented here, with an explanation of each one of them and the purpose of their usage.
- Chapter 3 refers to the mock-ups already presented in the RASD document.
- Chapter 4 explains how the requirements that have been defined in the RASD map to the design elements that are defined in this document.
- Chapter 5 presents the implementation, integration and test plan. It includes the how the different components of the application are integrated with each other, how they react, the testing strategy taken into account and analyse the risks in the application.
- Chapter 6 shows the effort spent by each group member while working on this project.
- Chapter 7 includes the reference documents.

Chapter 2

# Architectural design

## 2.1 Overview

Application architecture design is a process which has to be executed in a defined flow. High-level components and their interaction is displayed in Figure 2.1.

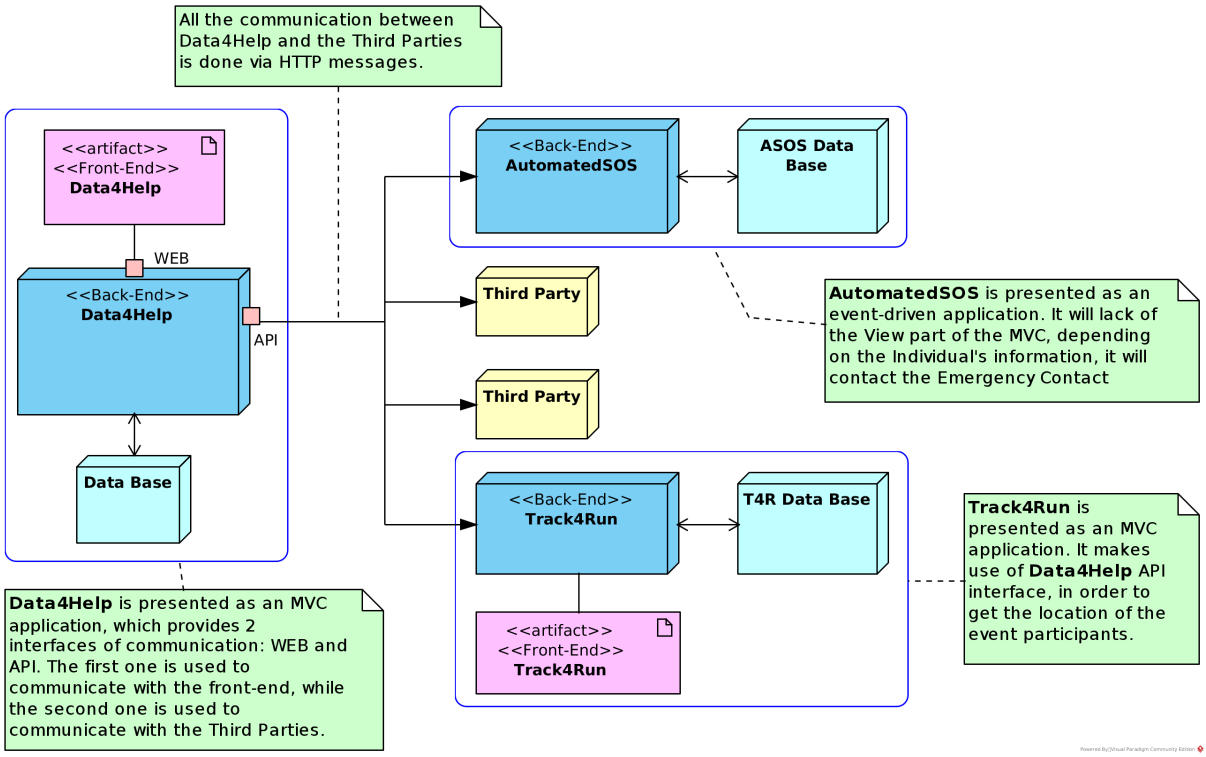


Figure 2.1: High-level architecture Diagram

As can be seen in the Figure 2.1, there are three main systems, which are Data4Help, AutomatedSOS and Track4Run. Two of the systems have an MVC architecture in which they provide an interface to the web users (Individuals, Third Parties, Participants, Organizers, and Spectators), and the last one (AutomatedSOS) does not provide any interface to them. All the communications between Data4Help and the Third Parties are done via HTTP messages in a RESTful way.

Finally, AutomatedSOS can be presented as an event-driven application, which receives the Individuals' information, and based on the defined thresholds it decides to contact or

not the assigned emergency contact.

## 2.2 Component view

### 2.2.1 Data4Help component diagram

#### Overview

In the Figure 2.2 it can be seen the component diagram of D4H, with all its external interfaces. It can be noticed two main components: **DataBase** and **D4HBackend**, where the former one refers to the Data4Help database, and which provides an interface used by the **DBManager** component.

On the other hand, **D4HBackend** component, contains all the components related to D4H, which are needed to provide the interfaces used by the third parties and the web site. The following interfaces are used by the web site: **SignupWeb**, **LoginWeb**, **SearchWeb**, **RequestWeb**, **UserWeb** and **SubscriptionWeb**, while the **RequestAPI** interface is used by the third parties. In this case, D4H provides the interface in order to let the third parties send requests for accessing the health status and location of the individuals to them. Is it important to notice that the *RequestPort* is used to show the difference between the interfaces that are consumed by Third Parties, and the interfaces used internally by the web site. The same can be noticed in the third party components, like Track4Run and AutomatedSOS, with their *DataPort* and *NotificationPort*.

Furthermore, **AuthenticatorManager** component has the responsibility of validate the different credentials, and to provide the secret codes to the third parties, to do so, it interacts with the **TokenDB** component using the **TokenConnector**.

Moreover, the **DataService** component has the responsibility of get the data from the TrackMe smart devices, and store them in the data base. The **Devices** component, make reference to all the TrackMe smart devices, which connect to the **DataService** component using an *Internal* interface.

Finally, it can be seen the different third parties related to D4H. It worth mentioning **Track4Run** and **AutomatedSOS** components which, even though they are part of the TrackMe environment, they are treated as third parties in the sense they are decoupled of D4H. Moreover, all third parties must provide an interface to D4H in order to let it communicate the incoming changes of the subscriptions.

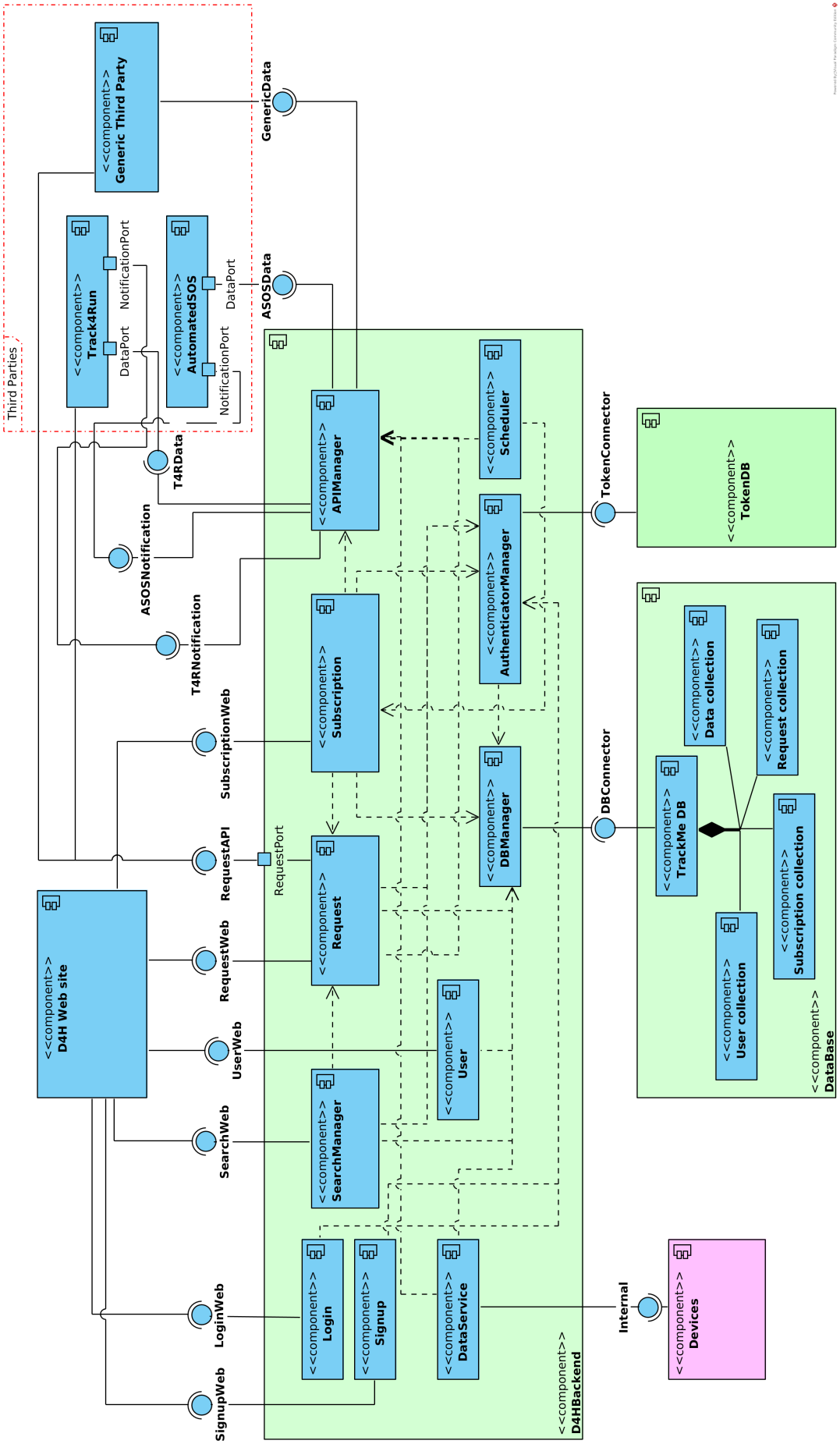


Figure 2.2: Data4Help Component Diagram

### Description

In the Table 2.1 a description of the components involved in D4H is shown.

| Component descriptions | |
|---|---|
| **Component** | **Description** |
| Login | Component is responsible of the login and logout actions into the site. It provides the session access tokens to the users and removes them when the user performs a logout. |
| Signup | Component responsible of register web-users, either Individuals or Third Party companies. It provides two different interfaces to the web-site, one for the Individual user and another one for the Third Party user, and provides a session access token and a secret key and application id to the Third Party users. |
| SearchManager | Component responsible of handling the searches of Individuals or group of Individuals' information. It should be able to anonymize the information of the group of users, and to respond with an error message when the Third Party user tries to access an Individual's information who have not accepted the request. |
| Request | Component responsible of adding the requests to a specific Individual and to accept or reject them. It should provide an internal interface, in order to send, accept and reject requests from within the web-site, and an external interface in order to let the Third Parties to send requests to specific users from their back-end. Finally, it should be capable of connecting to the notification API of the Third Parties in order to let them know that an Individual has accepted a Request. |
| Subscription | Component responsible of sending the information of the users to the subscribed Third Parties. It should be able to connect to the Third Parties endpoints in order to send the information regarding the saved queries. Also, it should expose an internal interface in order to let the Third Party users to save a particular query, delete it or get all the saved queries. |

| Component | Description |
|---|---|
| User | Component used by the front-end in order to get information about the current user (Individual or Third Party). Furthermore, it is responsible to get and store the changes in the profile of the current user, such as the Third Party configuration, which involve the URLs of the DataPort and NotificationPort |
| Scheduler | This component contains two schedulers: the DataScheduler and the ASOSRequestScheduler. The first one, is responsible to send the data related to the subscriptions of bulk data, it runs at a fixed time rate (an hour). The DataScheduler looks for all the subscriptions that should be executed, makes a search using the saved query, anonymize the data, and sends it to the Third Party. The second scheduler, runs once a day, and is responsible to create the requests of ASOS to all the elderly individuals |
| DBManager | Component responsible of connecting the database. It is related to all the other components since they depend on it. |
| AuthenticatorManager | Component responsible of validate and generate the session access tokens, and to provide the secret key and application id to the Signup component. |
| D4H Web site | Component that represent the frontier between the final user and the system. From this component, the Individual user is capable to accept or reject Requests, and the Third Party users are capable to search information of a particular Individual or a group of them, send Requests and/or creating subscriptions. |
| DataBase | Component that contains the main database (TrackMe DB), with all its collections and the token database (TokenDB) which contains all the valid access tokens and secret keys. |
| APIManager | Component responsible of connecting to the configured services of the Third Parties. It is responsible of sending the data of the subscriptions to the Third Parties, and of sending the notification of requests sent by the Third Party. |

Table 2.1: Component descriptions of D4H

### 2.2.2 AutomatedSOS component diagram

#### Overview

In the Figure 2.3 it can be seen the ASOS components. It can be noticed that the system has a similar structure of T4R: the **ASOSBackend** component communicates with **DataBase** component through the **DBConnector** interface, and provides a **DataAPI** interface to receive the updated information of the individuals.

On the other hand, **ASOSBackend** component sends notifications to the different **Health Care Service** components using the provided **Alarm Interface**.

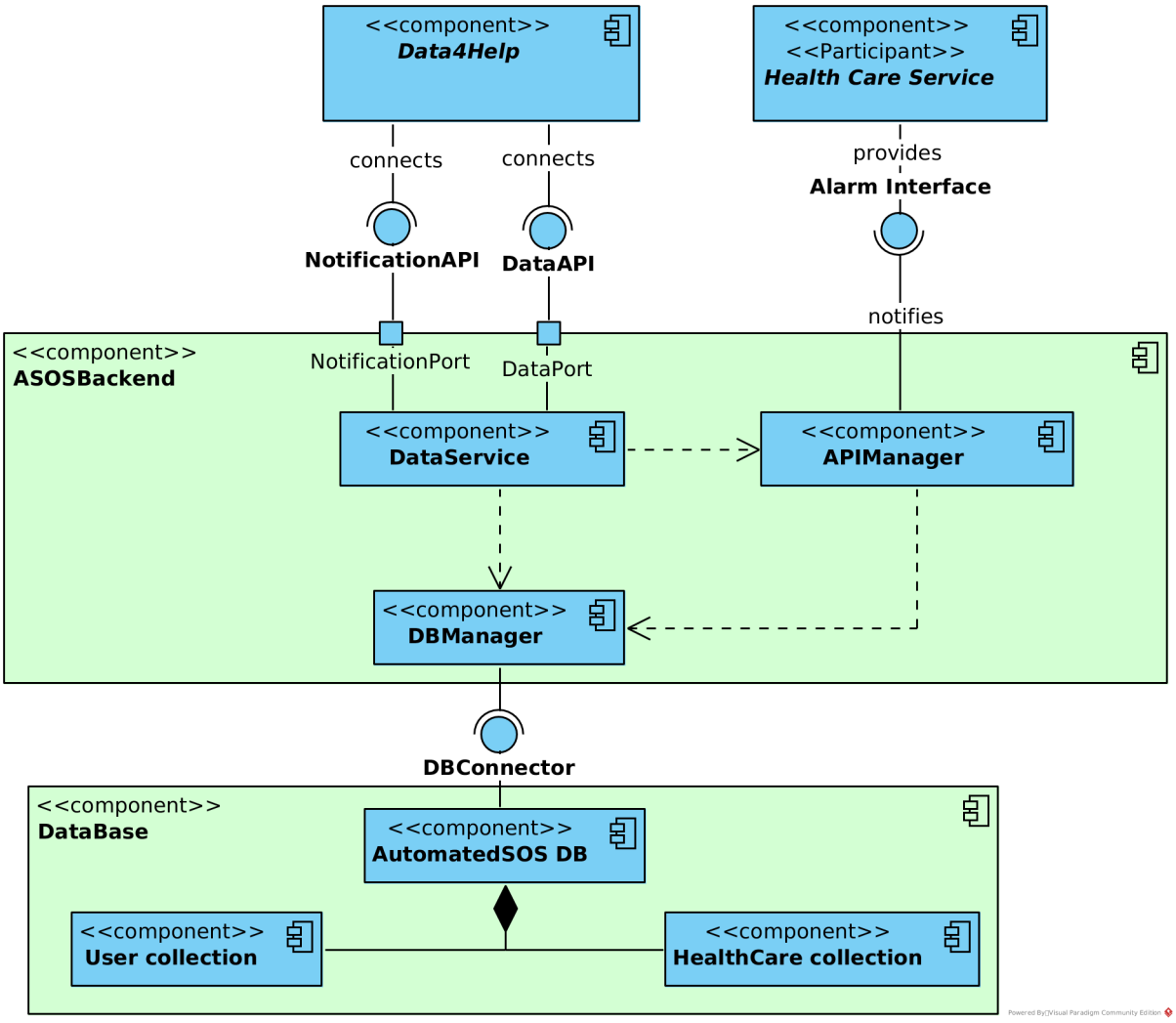


Figure 2.3: AutomatedSOS Component Diagram

#### Description

In the Table 2.2 a description of the components involved in ASOS is shown.

| Component descriptions | |
|---|---|
| **Component** | **Description** |
| DBManager | Component responsible of connecting the database. It is related to all the other components since they depend on it. |
| DataBase | Component that contains the main database (AutomatedSOS DB), with all its collections. |
| DataService | Component responsible to get the location and health data of the Individuals every time it has changes. It should be able to obtain the information sent by D4H, and to get the information related to the Individuals who accepted the request. Moreover, it should be capable to get the Notifications every time an Individual approves or rejects a request. |
| APIManager | Component responsible of connecting to the health-care service assigned to an Individual, when its parameters are out of normal range. |

Table 2.2: Component descriptions of ASOS

### 2.2.3 Track4Run component diagram

#### Overview

In the Figure 2.4 it can be seen the T4R components. As in the D4H component diagram, the *DataBase* component provides an interface in order to let the **DBManager** component communicate to it.

The main structure of the system is similar to the structure seen in D4H component diagram. The web site communicates with the back-end using the following interfaces: **LoginWeb**, **SignupWeb**, **UserWeb** and **NotificationWeb**. On the other hand, T4R provides the **DataAPI** interface, which is responsibly of receiving all the data of the participants, and uses the **RequestAPI** interface, in order to send the requests for accessing the individuals' location and health status.

Finally, as in D4H, the **AuthenticatorManager** component is responsible of validate the users' credentials.

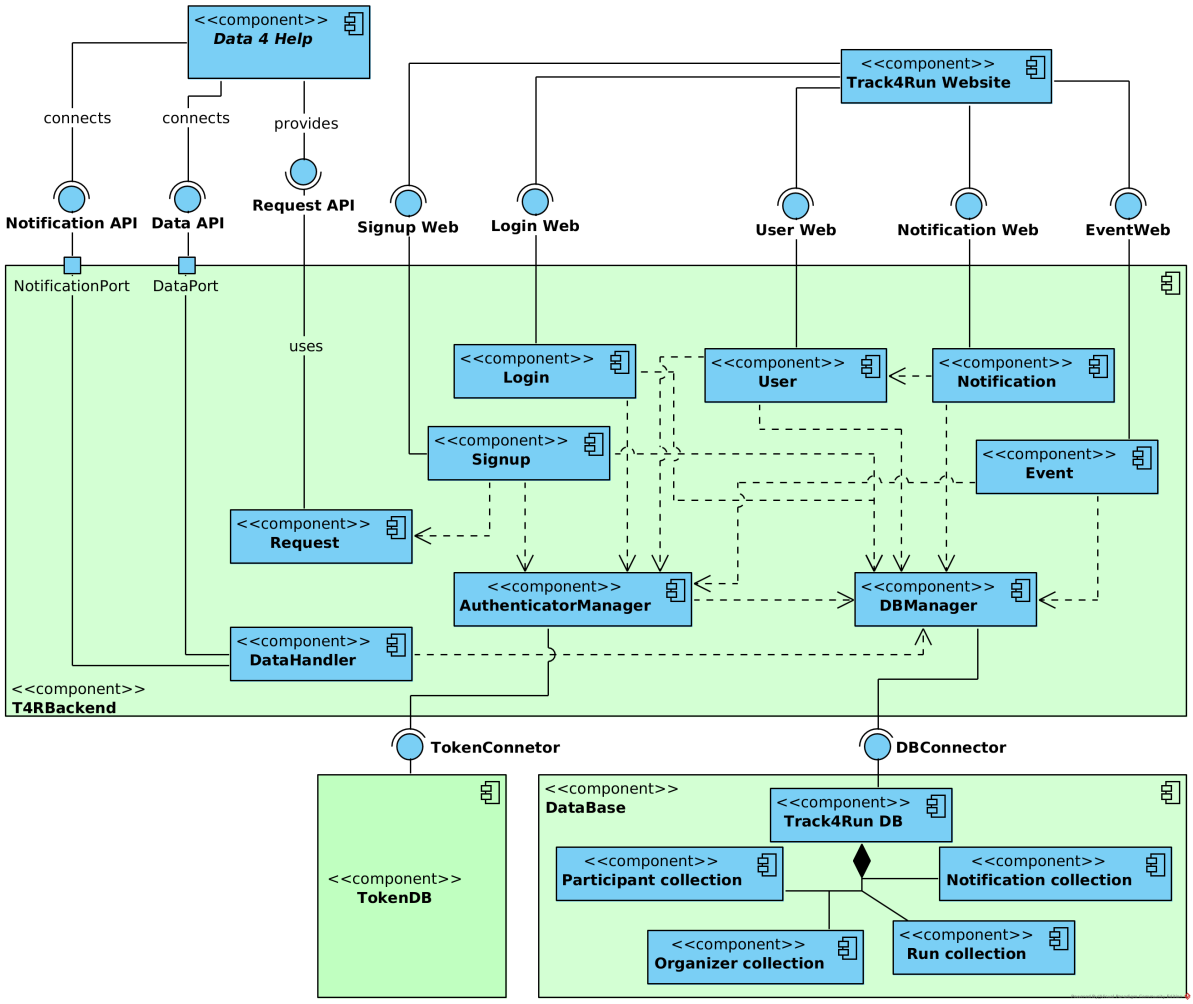


Figure 2.4: Track4Run Component Diagram

## Description

In the Table 2.3 a description of the components involved in T4R is shown.

| Component descriptions | |
|---|---|
| **Component** | **Description** |
| Login | Component is responsible of the login and logout to the site. It provides the session access tokens to the users and removes them when the user performs a logout. |
| Signup | Component responsible of register Participant and Organizer users. |
| AuthenticatorManager | Component responsible of validate and generate the session access tokens. |
| DBManager | Component responsible of connecting the database. It is related to all the other components since they depend on it. |

| Component | Description |
|---|---|
| T4R Web site | Component that represent the frontier between the final user and the system. From this component, the Participant user is capable to enrol in a Run, and the Organizer users are capable to create Run events, define the running circuit and send invites to Participant users. |
| DataBase | Component that contains the main database (Track4Run DB), with all its collections and the token database (TokenDB) which contains all the valid session access tokens. |
| Request | Component responsible of sending Request for accessing location of a specific Individual in D4H. It should be able to connect to D4H through the provided RequestAPI. |
| User | Component responsible of manage the Participant operations. It should be able to show all the enrolled and non-enrolled participants of a run. |
| Event | Component responsible of manage the Event operations. It should be able to create running events, to define the running circuit, and to get all the available running events. |
| Notification | Component responsible of manage the Notification operations. It should be able to send invitations to participants, and let them accept or reject it. |
| DataHandler | Component responsible to get the location and health data of the Individuals during a Race event. It should be able to obtain the information sent by D4H during a race event, and to get the information related to the Individuals who accepted the sent request. |

Table 2.3: Component descriptions of T4R

## 2.3 Component interfaces

In the following section all the interfaces of the systems (D4H, T4R, ASOS) are presented. Furthermore, classes belonging to the components, in particular *Service* classes count on methods that are related to the external interfaces. All the exposed methods expected input parameters and the expected outputs wil be listed in this section.

The endpoints are categorized as *setupWebEndpoints* or *setupApiEndpoints* according to

whether they are exposed for the web (front-end) or for third parties directly. In this section interfaces exposed for the web are explained in detail.

It is important to mention that `Spark.Request` and `Spark.Response` classes belong to the *Java Spark Framework*, and does nothing to do with the *D4HRequest* class of TrackMe system.

### 2.3.1 Data4Help interfaces

To have access to the exposed methods, it is required for requests to have the following HTTP headers to each request. The headers are required for all the methods except for *login*, *signupIndividual* and *signupThirdParty*.
**USER_ID**= userId:String
**ACCESS-TOKEN**= accessToken:String

- **Login**
  - **LoginService**
    - `LoginResponse login(Spark.Request req, Spark.Response response)`
      `POST /web/login`

      | **Input** *[LoginRequest]* | **Output** *[UserWebAuth]* |
      |---|---|
      | `email: String` | `userId: String` |
      | `password: String` | `accessToken: String` |
      | | `role: D4HUserRole` |

    - `String logout(Spark.Request req, Spark.Response response)`
      `POST /web/logout`
    - `String isValidToken(Spark.Request req, Spark.Response response)`
      `HEAD /web/`
  - **UserResource**
    - `D4HUser getByEmailAndPass(String email, String password)`
    - `ThirdParty getThirdPartyBySecretKey(String secretKey)`
    - `Individual getBySSN(String ssn)`
    - `List<Individual> getByQuery(D4HQuery query)`
- **Signup**

- **SignupService**
  - `SignupResponse signupIndividual(Spark.Request req, Spark.Response res)`
    `POST /web/individual/signup`

| **Input** *[IndividualSignupRequest]* | **Output** *[UserWebAuth]* |
|---|---|
| `email: String` | `userId: String` |
| `password: String` | `accessToken: String` |
| `name: String` | `role: D4HUserRole` |
| `ssn: String` | |
| `birthDate: Date` | |
| `gender: Gender` | |
| `bloodType: BloodType` | |
| `weight: Float` | |
| `height: Float` | |
| `address: Address` | |
| `address.country: String` | |
| `address.province: String` | |
| `address.city: String` | |

  - `SignupResponse signupThirdParty(Spark.Request req, Spark.Response res)`
    `POST /web/thirdparty/signup`

| **Input** *[ThirdPartySignupRequest]* | **Output***[UserWebAuth]* |
|---|---|
| `email: String` | `userId: String` |
| `password: String` | `accessToken: String` |
| `certificate: String` | `role: D4HUserRole` |
| `name: String` | |
| `phone: String` | |
| `taxCode: String` | |

- **Search**
  - **SearchService**
    - `Collection<Data> search(Spark.Request req,`
      `Spark.Response res)`
      `POST /web/search`

| **Input** *[D4HQuery]* | **Output** |
|---|---|
| `ssn: String` | `ssn: String` |
| | `name: String` |
| | `data: Data` |
| | `data.location: Location` |
| | `data.location.longitude: Long` |
| | `data.location.latitude: Long` |
| | `data.healthStatus: HealthStatus` |
| | `data.healthStatus.heartRate: Integer` |
| | `data.healthStatus.healthStatus.systolic: Double` |
| | `data.healthStatus.healthStatus.diastolic: Double` |
| | `data.healthStatus.bodyTemperature: Double` |
| | `data.healthStatus.bloodOxygen: Integer` |
| `country?: String` | `data: List<Data>` |
| `city?: String` | `data[*].healthStatus: HealthStatus` |
| `gender?: Gender` | `data[*].healthStatus.heartRate: Integer` |
| `minAge?: Integer` | `data[*].healthStatus.systolic: Double` |
| `maxAge?: Integer` | `data[*].healthStatus.diastolic: Double` |
| `bloodType?: BloodType` | `data[*].healthStatus.bodyTemperature: Double` |
| `province?: String` | `data[*].healthStatus.bloodOxygen: Integer` |

  - **DataResource**
    - `Data getByIndividualId(Individual i)`
    - `Data getByIndividual(Individual individual)`
    - `Collection<Data> getByIndividualList(Collection<Individual> individuals)`
    - `Collection<Data> getAnonymizeByIndividualList(Collection<Individual> indi`

- **D4HRequest**
  - **D4HRequestService**
    - `D4HReqResponse createRequest(Spark.Request req, Spark.Response res)`
      `POST /api/requests/`

| **Input** *[D4HReqRequest]* | **Output** |
|---|---|
| `ssn: String` | `requestId: String` |
| | `thirdParty: ThirdParty` |
| | `thirdParty.name: String` |
| | `individual: Individual` |
| | `individual.name: String` |
| | `status: D4HRequestStatus` |

* `D4HReqResponse updateRequestStatus(Spark.Request req, Spark.Response res)`
  `PATCH /web/requests/:requestId`

| **Input** *[D4HReqRequest]* | **Output** |
|---|---|
| `ssn: String` | `id: String` |
| `status: D4HRequestStatus` | `thirdParty: ThirdParty` |
| | `thirdParty.name: String` |
| | `individual: Individual` |
| | `individual.name: String` |
| | `status: D4HRequestStatus` |

* `Collection<D4HReqResponse> getAllRequests(Spark.Request req,`
  `Spark.Response res)`
  `GET /web/requests/:requestStatus`

| **Input** | **Output** |
|---|---|
| | `requests: List<Request>` |
| | `requests[*].id: String` |
| | `requests[*].thirdParty: ThirdParty` |
| | `requests[*].thirdParty.name: String` |
| | `requests[*].individual: Individual` |
| | `requests[*].individual.name: String` |
| | `requests[*].status: D4HRequestStatus` |

– **D4HRequestResource**

* `Collection<D4HRequest> getById(String userId)`
* `Collection<D4HRequest> getByUserId(String userId, D4HRequestStatus status`
* `Collection<D4HRequest> getByThirdPartyId(String thirdPartyId, D4HRequestS`
* `D4HRequest add(Request D4HRequest)`
* `D4HRequest accept(D4HRequest r)`

    * `D4HRequest reject(D4HRequest r)`

- **Subscription**
  - **SubscriptionService**
    * `SubscriptionResponse createSubscription(Spark.Request req, Spark.Response res)`
      `POST /web/subscriptions/`

| **Input***[SubscriptionRequest]* | **Output** |
|---|---|
| `filter: D4HQuery` | `subscriptionId: String` |
| `filter.ssn?: String` | `filter: D4HQuery` |
| `filter.city?: String` | `filter.ssn: String` |
| `filter.province?: String` | `filter.city: String` |
| `filter.country?: String` | `filter.province: String` |
| `filter.gender?: Gender` | `filter.country: String` |
| `filter.minAge?: Integer` | `filter.gender: Gender` |
| `filter.maxAge?: Integer` | `filter.minAge: Integer` |
| `filter.bloodType?: BloodType` | `filter.maxAge: Integer` |
| `timeSpan: Integer` | `filter.bloodType: BloodType` |

    * `Collection<Subscription> getAllSubscriptions(Spark.Request req, Spark.Response res)`
      `GET /web/subscriptions/`

| Input | Output |
|---|---|
| | subscriptions: List<Subscription> |
| | subscriptions[*].id: String |
| | filter: D4HQuery |
| | subscriptions[*].filter.ssn: String |
| | subscriptions[*].filter.city: String |
| | subscriptions[*].filter.province: String |
| | subscriptions[*].filter.country: String |
| | subscriptions[*].filter.gender: Gender |
| | subscriptions[*].filter.minAge: Integer |
| | subscriptions[*].filter.maxAge: Integer |
| | subscriptions[*].filter.bloodType: BloodType |
| | subscriptions[*].thirdParty: ThirdParty |
| | subscriptions[*].thirdParty.name: String |
| | subscriptions[*].thirdParty.certificate: String |
| | subscriptions[*].thirdParty.phone: String |
| | subscriptions[*].thirdParty.taxCode: String |
| | subscriptions[*].timeSpan: Integer |

        * `void removeSubscription(Spark.Request req, Spark.Response res)`
          `DELETE /web/subscriptions/:subscriptionId`

    - **SubscriptionResource**

        * `Collection<Subscription> getAllByIndividual(ObjectId individualId)`
        * `Collection<Subscription> getAllByOwner(ObjectId thirPartyId)`
        * `Subscription getByOwnerAndId(String thirdPartyId, String sid)`
        * `void removeByRequest(D4HRequest req)`

- **User**

    - **UserService**

        * `D4HUserResponse getProfileInfo(Spark.Request req,`
          `Spark.Response res)`
          `GET /web/me/`

| Input | Output |
|---|---|
| | `name: String` |
| | `ssn: String` |
| | `birthDate: Date` |
| | `gender: Gender` |
| | `bloodType: BloodType` |
| | `weight: Float` |
| | `height: Float` |
| | `address: Address` |
| | `address.country: String` |
| | `address.province: String` |
| | `address.city: String` |
| | `pendingRequests: Long` |
| | `approvedRequests: Long` |
| | `rejectedRequests: Long` |
| | `name: String` |
| | `phone: String` |
| | `taxCode: String` |
| | `secretKey: String` |
| | `appId: String` |
| | `config: TPConfiguration` |
| | `config.individualPushUrl: String` |
| | `config.bulkPushUrl: String` |
| | `config.notificationUrl: String` |
| | `pendingRequests: Long` |
| | `approvedRequests: Long` |
| | `rejectedRequests: Long` |

* `TPConfiguration updateThirdPartyConfig(Spark.Request req, Spark.Response res)`
  `PATCH /web/me/config`

| **Input** *[TPConfiguration]* | **Output** |
|---|---|
| `config: TPConfiguration` | `config: TPConfiguration` |
| `config.individualPushUrl: String` | `config.individualPushUrl: String` |
| `config.bulkPushUrl: String` | `config.bulkPushUrl: String` |
| `config.notificationUrl: String` | `config.notificationUrl: String` |

### 2.3.2 AutomatedSOS interfaces

ASOS has a different architecture as the ones seen in T4R and D4H, because it does not relay on a web-site nor a user. The system receives the data of the subscribed Individuals through the DataAPI, which is the interface provided to D4H, to which it connects. The NotificationAPI, let D4H connect to ASOS in order to send the status of the Requests sent to an Individual. In the following list, the exposed interfaces are shown, linked with the classes and methods, and detailing the inputs and outputs:

- **DataService**
  - DataService
    - `void getIndividualData(Spark.Request req, Spark.Response res)`
      `POST /api/data`

| **Input** *[IndividualDataRequest]* | **Output** |
|---|---|
| `ssn: String` | |
| `data: Data` | |
| `data.location: Location` | |
| `data.location.longitude: Long` | |
| `data.location.latitude: Long` | |
| `data.healthStatus: HealtStatus` | |
| `data.healthStatus.heartRate: Integer` | |
| `data.healthStatus.systolic: Double` | |
| `data.healthStatus.diastolic: Double` | |
| `data.healthStatus.bodyTemperature: Integer` | |
| `data.healthStatus.bloodOxygen: Integer` | |

    - `void getRequestNotification(Spark.Request req, Spark.Response res)`
      `POST /api/notification`

| **Input** *[NotificationRequest]* | **Output** |
|---|---|
| `individual: ASOSUser` | |
| `individual.ssn: String` | |
| `individual.birthDate: LocalDate` | |
| `individual.gender: Gender` | |
| `individual.bloodType: BloodType` | |
| `status: RequestStatus` | |

  - ThresholdResource

    - `HashMap<HealthParameter, Threshold> get(Integer age)`
    - `Boolean isOutOfRange(Double value, Threshold threshold)`
  - UserResource
    - `ASOSUser getBySSN(String ssn)`
    - `void remove(String userId)`

- **APIManager**
  - APIManager
    - `void sendNotification(EmergencyContact contact, ASOSUser user)`

- **DBManager**
  - DBManager
    - `Morphia.Datastore getDatastore()`

### 2.3.3 Track4Run interfaces

The following is a list of all the components of T4R, and the classes that belong to them, with the exposed methods.

As in D4H, the Service classes have methods that are related to the endpoints used by the web-site. A special case is the DataHandler component, which is responsible of getting the notifications from D4H and the data of the individuals. The NotificationAPI and DataAPI refer to the interfaces exposed to D4H and to which it connects. On the other hand, the Request component connects to D4H by using the RequestAPI, in order to send Requests for accessing individual location during an event. In the following list, the exposed interfaces - used by the web-site and by D4H - are shown, linked with the classes and methods, and detailing the inputs and outputs:

- **Login**
  - **LoginService**
    - `LoginResponse login(Spark.Request req, Spark.Response response)`
      `POST /web/login`

| **Input** | **Output** |
|---|---|
| `email: String` | `userId: String` |
| `password: String` | `accessToken: String` |
| | `type: UserType` |

* `void logout(Spark.Request req, Spark.Response response)`
  `POST /web/logout`

| Input | Output |
|---|---|
| `userId: String` | |
| `accessToken: String` | |

– **UserResource**

* `T4RUser getByEmailAndPass(String email, String password)`

- **Signup**

– **SignupService**

* `SignupResponse signupParticipant(Spark.Request req, Spark.Response res)`
  `POST /web/individual/signup`

| Input | Output |
|---|---|
| `email: String` | `userId: String` |
| `password: String` | `accessToken: String` |
| `name: String` | |
| `ssn: String` | |

* `SignupResponse signupOrganizer(Spark.Request req, Spark.Response res)`
  `POST /web/thirdparty/signup`

| Input | Output |
|---|---|
| `email: String` | `userId: String` |
| `password: String` | `accessToken: String` |
| `name: String` | |
| `phone: String` | |
| `website: String` | |

– **UserResource**

* `void add(T4RUser u)`

- **User**

– **UserService**

* `Collection<Participant> getEnrolledParticipants(Spark.Request req, Spark.Response res)`

GET /web/:eventId/participants

| Input | Output |
|---|---|
| | participants: Collection<Participant> |
| | participants[*].userId: String |
| | participants[*].name: String |

* void enrollToEvent(Spark.Request req, Spark.Response res)
  POST /web/participant/:userId/:eventId
* void cancelEnrollment(Spark.Request req, Spark.Response res)
  DELETE /web/participant/:userId/:eventId

- **Event**
  - **EventService**
    * Collection<Event> getAvailableEvents(Spark.Request req, Spark.Response re
      GET /web/event

| Input | Output |
|---|---|
| | events: Collection<Event> |
| | events[*].eventId: String |
| | events[*].name: String |
| | events[*].startDate: Date |
| | events[*].organizer: Organizer |
| | events[*].organizer.name: String |

    * void createEvent(Spark.Request req, Spark.Response res)
      POST /web/event

| Input | Output |
|---|---|
| name: String | |
| organizerId: String | |
| startDate: Date | |
| endtDate: Date | |
| path: Collection<Coordinate> | |
| path[*].longitude: Long | |
| path[*].latitude: Long | |

    * void updateEvent(Spark.Request req, Spark.Response res)
      PUT /web/event/:eventId

| Input | Output |
|---|---|
| name: String | |
| startDate: Date | |
| endtDate: Date | |
| path: Collection<Coordinate> | |
| path[*].longitude: Long | |
| path[*].latitude: Long | |

  - **EventResource**
    - Event getById(String eventId)
    - Collection<Event> getAll()
    - void add(Event e)
    - void update(Event e)

- **Notification**
  - **NotificationService**
    - Collection<Notification> getPendingNotifications(Spark.Request req, Spark.Response res)
      GET /web/:userId/notification

| Input | Output |
|---|---|
| | notifications: Collection<Notification> |
| | notifications[*].event: Event |
| | notifications[*].event.eventId: String |
| | notifications[*].event.name: String |

    - void accept(Spark.Request req, Spark.Response res)
      PUT /web/:userId/notification/:notificationId
    - void delete(Spark.Request req, Spark.Response res)
      DELETE /web/:userId/notification/:notificationId
    - void createNotification(Spark.Request req, Spark.Response res)
      POST /web/notification

| Input | Output |
|---|---|
| userId: String | |
| eventId: String | |

  - **NotificationResource**
    - `Notification getByUserId(String userId)`
    - `void update(Notification u)`
  - **EventResource**
    - `Event getById(String eventId)`
    - `void addParticipant(String eventId, String participantId)`

- **DataHandler**
  - **DataHandler**
    - `void getParticipantLocation(Spark.Request req, Spark.Response res)`
      `POST /api/participant/data`

| Input | Output |
|---|---|
| `ssn: String` | |
| `data: Data` | |
| `data.location: Location` | |
| `data.location.longitude: Long` | |
| `data.location.latitude: Long` | |
| `data.healthStatus: HealtStatus` | |
| `data.healthStatus.heartRate: Integer` | |
| `data.healthStatus.bloodPreasure: Integer` | |
| `data.healthStatus.bodyTemperature: Integer` | |
| `data.healthStatus.bloodOxygen: Integer` | |

    - `void getRequestNotification(Spark.Request req, Spark.Response res)`
      `POST /api/participant/notification`

| Input | Output |
|---|---|
| `name: String` | |
| `ssn: String` | |
| `birthDate: Date` | |
| `gender: Gender` | |
| `status: RequestStatus` | |

- **Request**
  - **RequestService**
    - `void sendRequestToParticipant(String ssn)`

## 2.4 Database view

In the previous section, an architectural landscape of D4H, ASOS and T4R was provided by means of high-level component diagrams. Those diagrams showed how components communicate with the rest of the system (i.e. through interfaces). Now, it is also meaningful to describe data models involved using class diagrams.

### 2.4.1 Data4Help data model

As explained in the RASD, the whole data model of TrackMe up to now will be treated as a "black box", this means **D4H** will have a local copy of the *basic information* and *collected data* only of users who activated this service. The classes considered in D4H, their attributes, and relationships are shown in Figure **??**.

- **D4HUser**: credentials of all users interacting with the system.
- **Individual**: main attributes of users wearing the devices.
- **Data**: health status, location and timestamp collected from devices.
- **ThirdParty**: companies requesting data of individuals or bulk data. For registering, all third parties must provide a certificate to verify that it is a legally constituted company (it was assumed that there is a mechanism capable of doing this checking).
- **TPConfiguration**: an important issue to tackle is the way in which the system will communicate to third parties to send them notifications or data they request. Then, they are asked to provide three endpoints, *individualpushUrl*, *bulkPushUrl* and *notificationUrl*.
- **D4HRequest**: third parties can make requests to access data of a given individual. Each request has an associated *status*, initially is *PENDING* and can become either *APPROVED* or *REJECTED* according to whether the individual accepts it or not.
  - After the status of a request is updated to *APPROVED*, a subscription to receive new data of the intended individual is automatically created.
  - *Requests for bulk data* are treated differently. The only constraints are managed by the system (if they do not hold the third party is notified), then there is no need to store them.
- **Subscription**: when searching for bulk data, third parties can decide if they want to subscribe to that specific *filter*. This means, they will receive new data after a

certain amount of time defined by a *timeSpan* (hours). New data will be send to individuals according to the nextExecution attribute of the subscription.

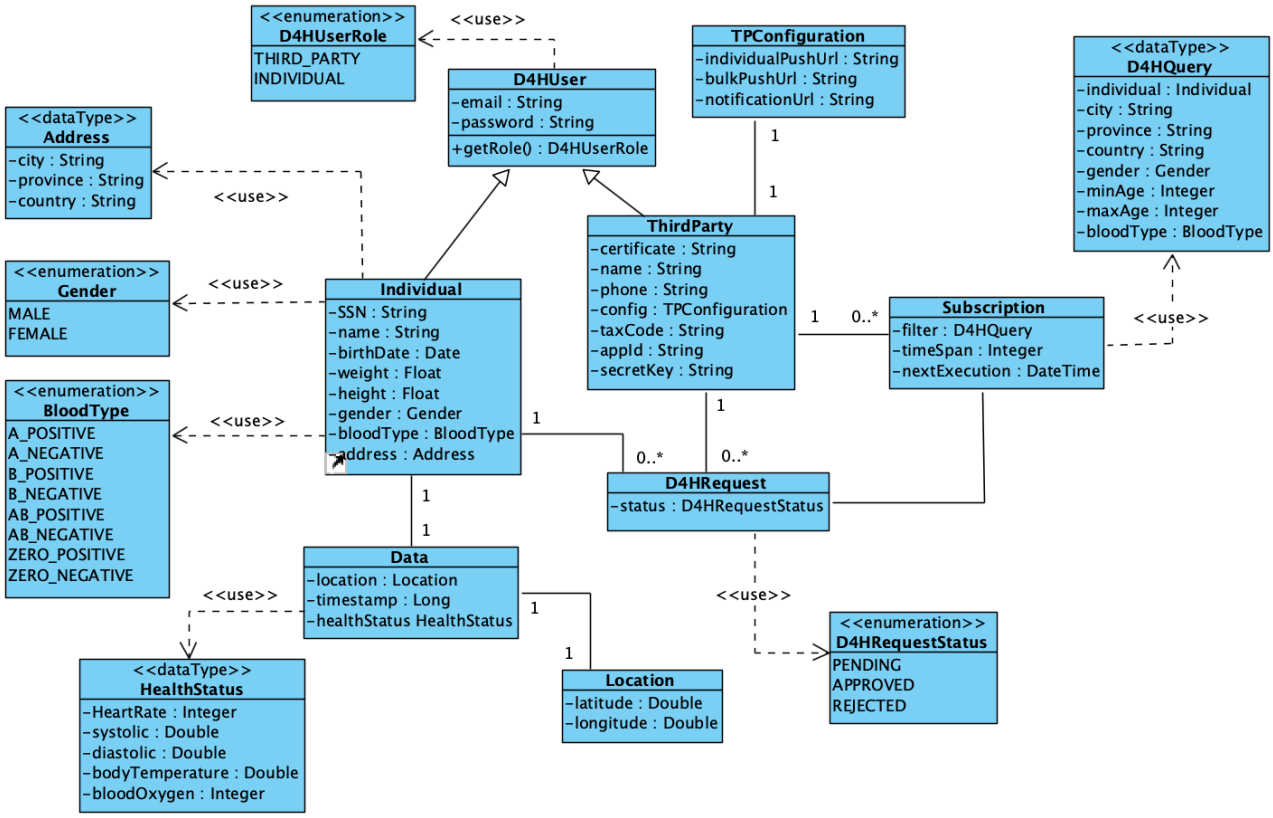


Figure 2.5: Data4Help Class Diagram

### 2.4.2 AutomatedSOS data model

The classes considered in D4H, their attributes, and relationships are shown in Figure 2.5.

- **ASOSUser**: individuals subscribed to D4H that decided to activate ASOS service. Each individual is associated atutomatically to the **EmergencyContact** corresponding to his address. They also have a *status* so as to avoid duplicate notifications to the health care service.
- **EmergencyContact**: available health care services by addresses. Each health care service will receive notifications by means of a URL and only of users living in the same city.
- **Data**: health parameters measured by means of the wearable devices and received from D4H.

- **Threshold**: normal ranges for each health parameter according to the age of the individual (i.e. minimum and maximum values).

Data coming from d4h is compared against predefined thresholds, all of them are described as follows (Figure 2.4):

| **HealthParameter** | **minAge** | **maxAge** | **minValue** | **maxValue** |
|---|---|---|---|---|
| HEART_RATE | 60 | 65 | 80 | 120 |
| HEART_RATE | 65 | 70 | 78 | 116 |
| HEART_RATE | 70 | 75 | 75 | 113 |
| HEART_RATE | 75 | 130 | 73 | 109 |
| SYSTOLIC | 0 | 130 | 90 | 250 |
| DIASTOLIC | 0 | 130 | 60 | 140 |
| TEMPERATURE | 60 | 65 | 35.2 | 36.9 |
| TEMPERATURE | 65 | 130 | 35.6 | 36.3 |
| BLOOD_OXYGEN | 0 | 130 | 97 | 100 |

Table 2.4: Thresholds for every health parameter

It is necessary to clarify that the parameters (i.e. users' data) are not stored, they are used exclusively to make comparisons with their corresponding *thresholds*.

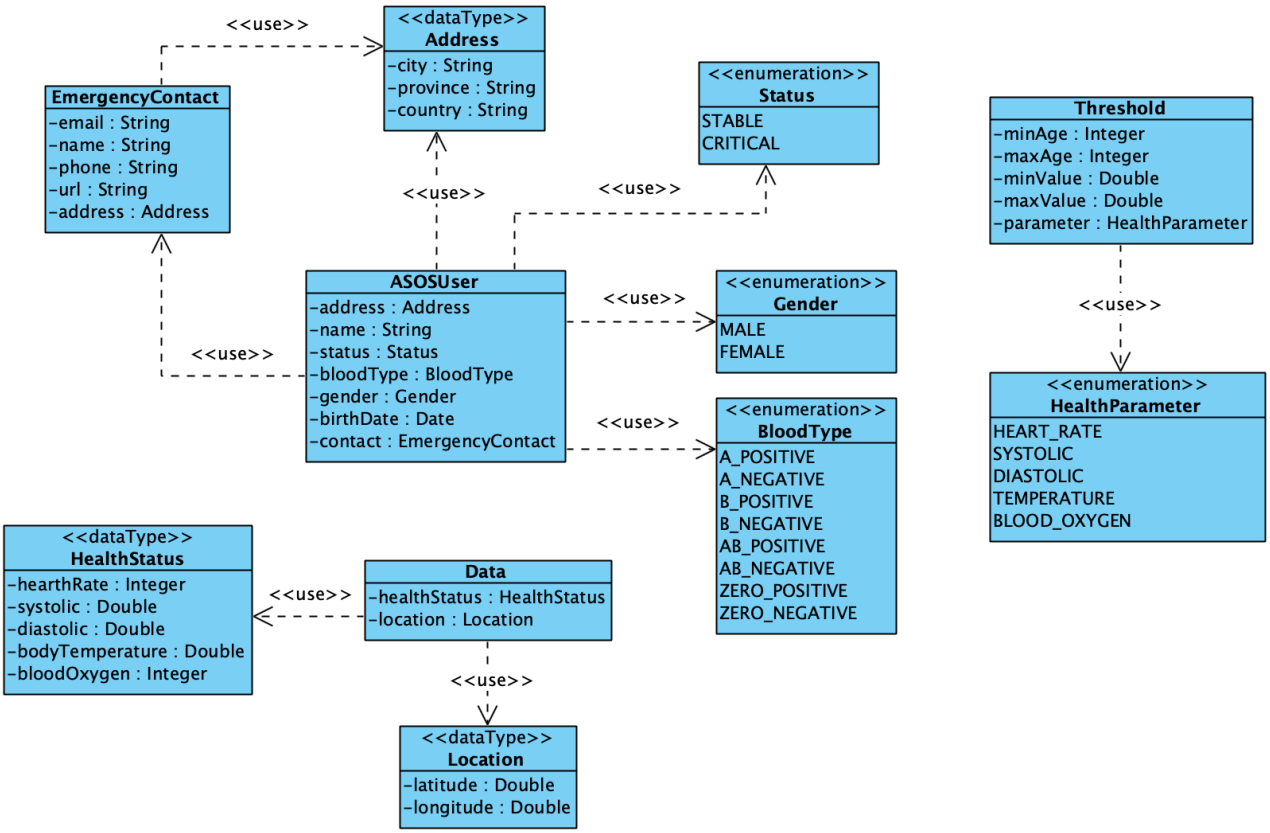


Figure 2.6: AutomatedSOS Class Diagram

### 2.4.3 Track4Run data model

- **Run**: characterized by a name, start and end times, and a path.
- **Organizer**: responsible for setting it up runs and send invitations to individuals.
- **Participant**: individuals enrolled in any run.
- **Invitation**: invitations to participate on runs sent by organizers.

The system does not need to keep record of the spectators.

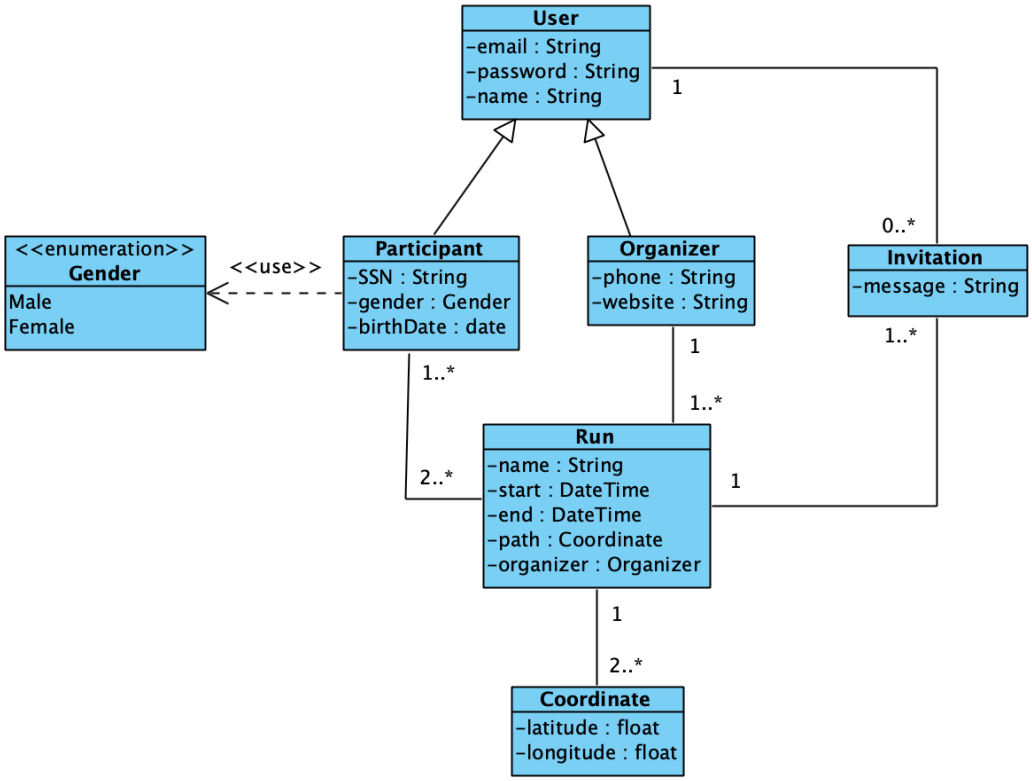


Figure 2.7: Track4Run Class Diagram

## 2.5 Deployment view

In the current chapter the deployment diagrams of D4H, T4R and ASOS are shown. All diagrams follow the same structure and design: yellow nodes represent external nodes that interact with the system, green nodes represent hardware devices (i.e. web server, database server, etc.), pale orange nodes represent execution environments such as JVM, and blue boxes represent high level components. Finally, the protocols used to communicate between different nodes displayed as red links.

### 2.5.1 Data4Help deployment diagram

In the Figure 2.8 can be seen the deployment diagram of D4H. As can be seen, the **Web Server** node, is a device which contains the execution environment that holds the **D4H Backend** component. The **Web Server** node, is related to two different devices: a **MongoDB Server** which is the environment of the **TrackMe DB**, and a **Redis Server** which is the environment of the **TokenDB**.

Moreover, the **Web Server** is related to three third parties: two of them are **Track4Run**

execution environment, and **AutomatedSOS** execution environment. The third third party is a **Generic Third Party Backend** node that is external to the TrackMe environment.

It worth notice, that the **Smart devices** are the ones that provide the data of the individuals to D4H. These devices connect to the D4H back-end using the internal interface, as shown in Figure 2.2.

Finally, the deployment diagram does not show the interaction between the web-users (individuals or third parties) with the system. This interaction is inherent to a web site, and it happens between the client and the **D4H Web page** component.

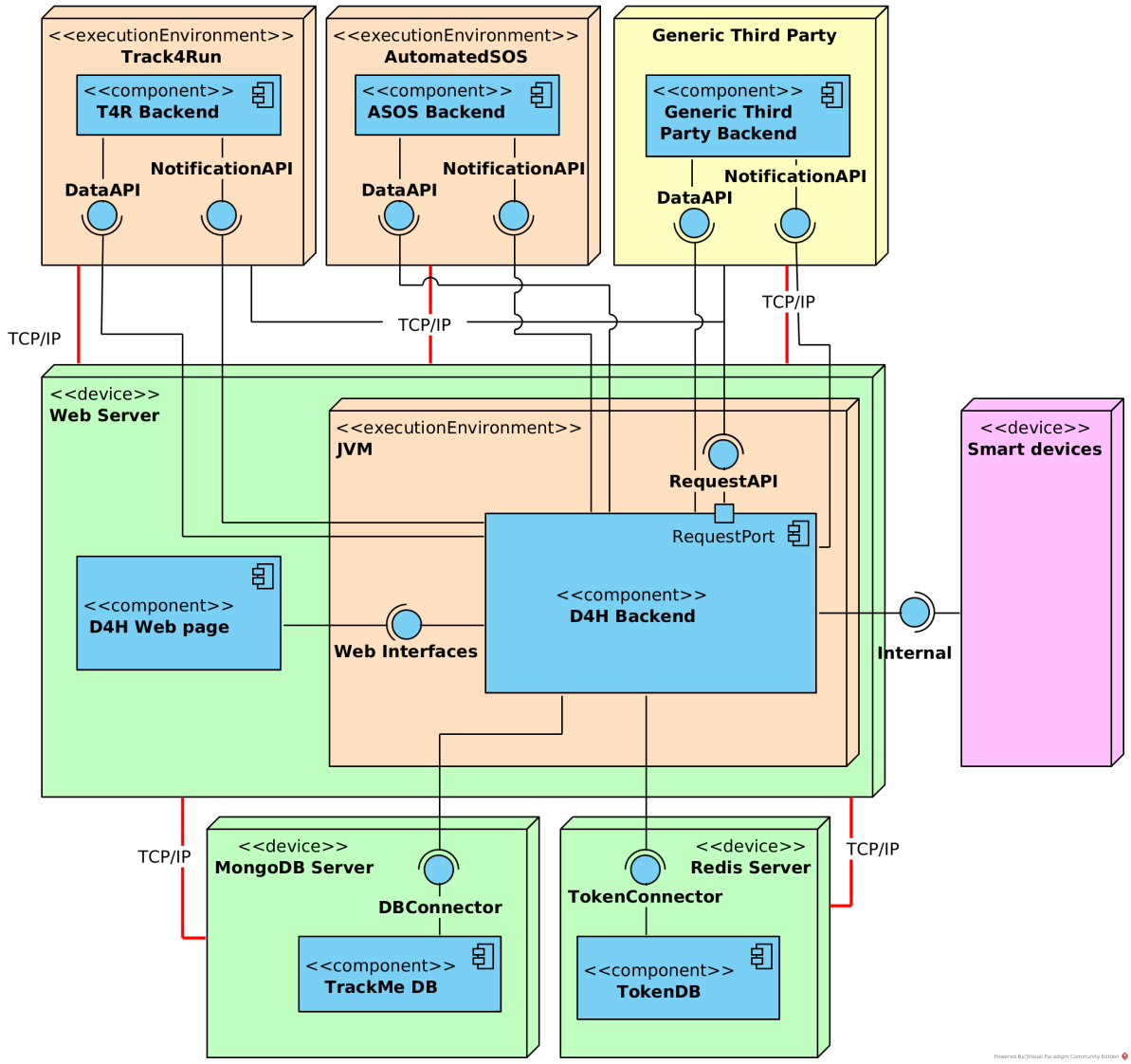


Figure 2.8: Data4Help Deployment Diagram

### 2.5.2 AutomatedSOS deployment diagram

In the Figure 2.9 the ASOS deployment diagram is shown. Since ASOS does not have a web site, the main node is an **Application Server** that hosts the execution environment for the **ASOS Backend** component. The **Application Server** interacts with a **MongoDB Server** node, using the TCP/IP protocol. Moreover, it interacts with an external node which represent the different **Health Care Service**s, and with the execution environment of **Data4Help**.

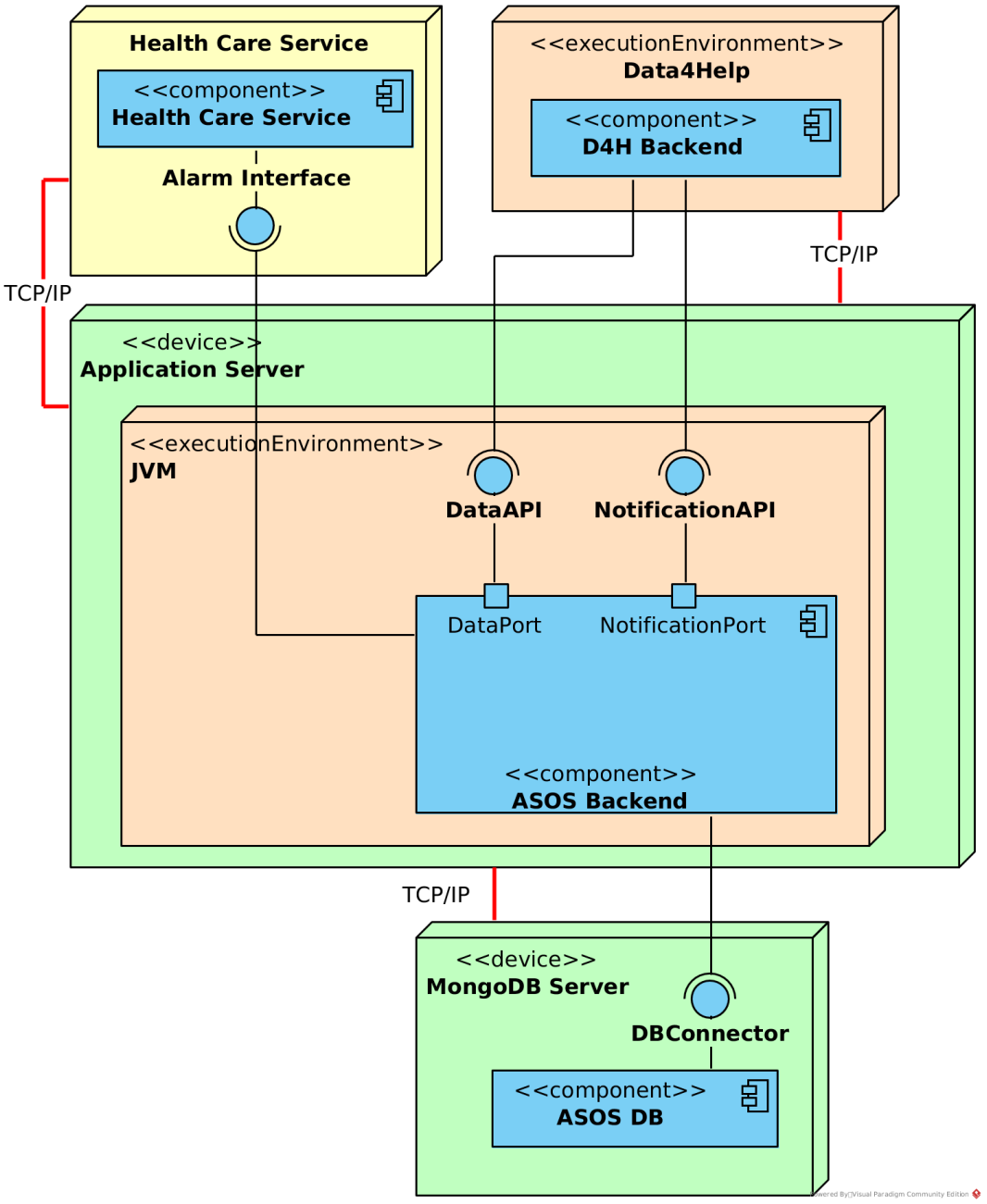


Figure 2.9: AutomatedSOS Deployment Diagram

### 2.5.3 Track4Run deployment diagram

In the Figure 2.10 the T4R deployment diagram is shown. It can be seen the interaction between **Data4Help** execution environment and the **Web Server** node. This interaction happens thanks to the different interfaces provided by D4H and T4R.
Furthermore, the **Web Server** node interacts with the **MongoDB Server** node using the TCP/IP protocol, and it works thanks to the **DBConnector** interface provided by the **T4R Database** component.

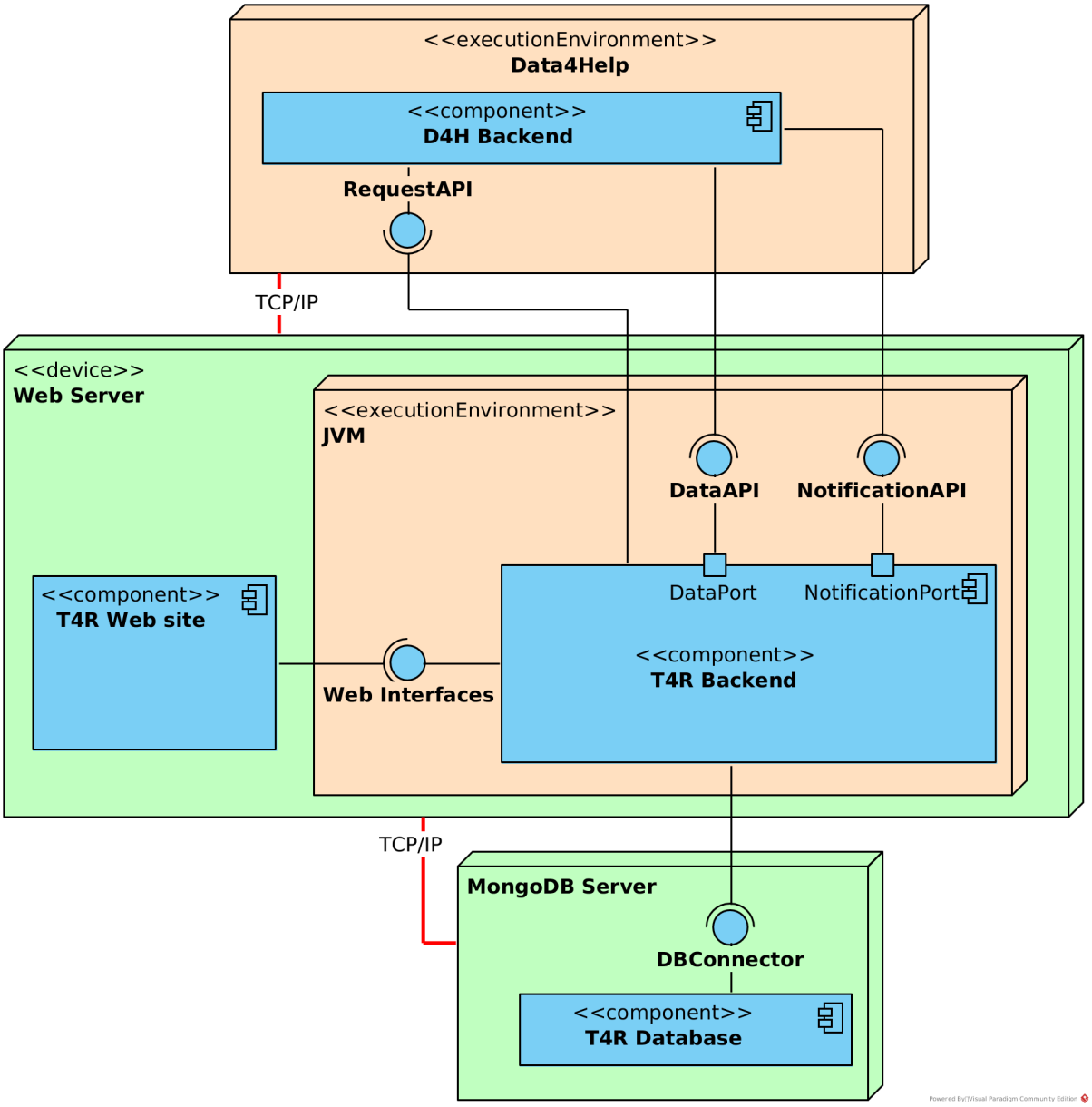


Figure 2.10: Track4Run Deployment Diagram

## 2.6 Runtime view

In the following section sequence diagrams of D4H, T4R and ASOS are shown.

### 2.6.1 Data4Help sequence diagrams

In D4H the **:SignupService** component receives a signup request, and communicates with the **:UserWebAuth** in charge of creating an accessToken and userId for the session of the user, they are send back to the **GUI**. This process applies for third parties as well, by means of **signupThirdParty** exposed method.

- **Registration**

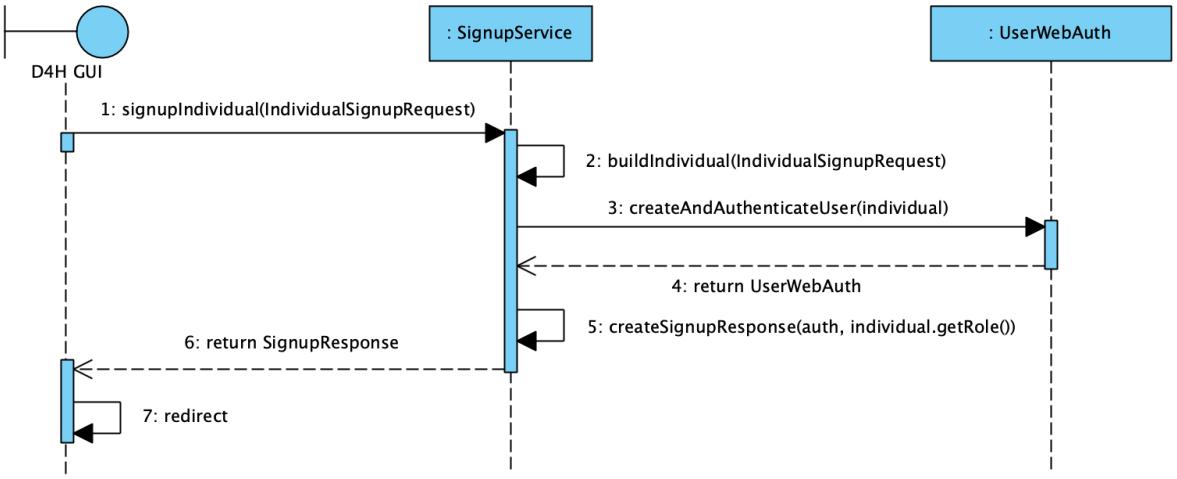


Figure 2.11: Signup Sequence Diagram

- **Login**

  The system also provides login and logout mechanisms to both types of users: individuals and third parties. The **:LoginService** component is in charge of contacting the **:AuthenticationManager** to set the accessToken for the user. This token is retrieved to the **GUI** to be used for upcoming requests. The GUI decides whether to redirect the user to its corresponding dashboard or display an error according to credentials checking.

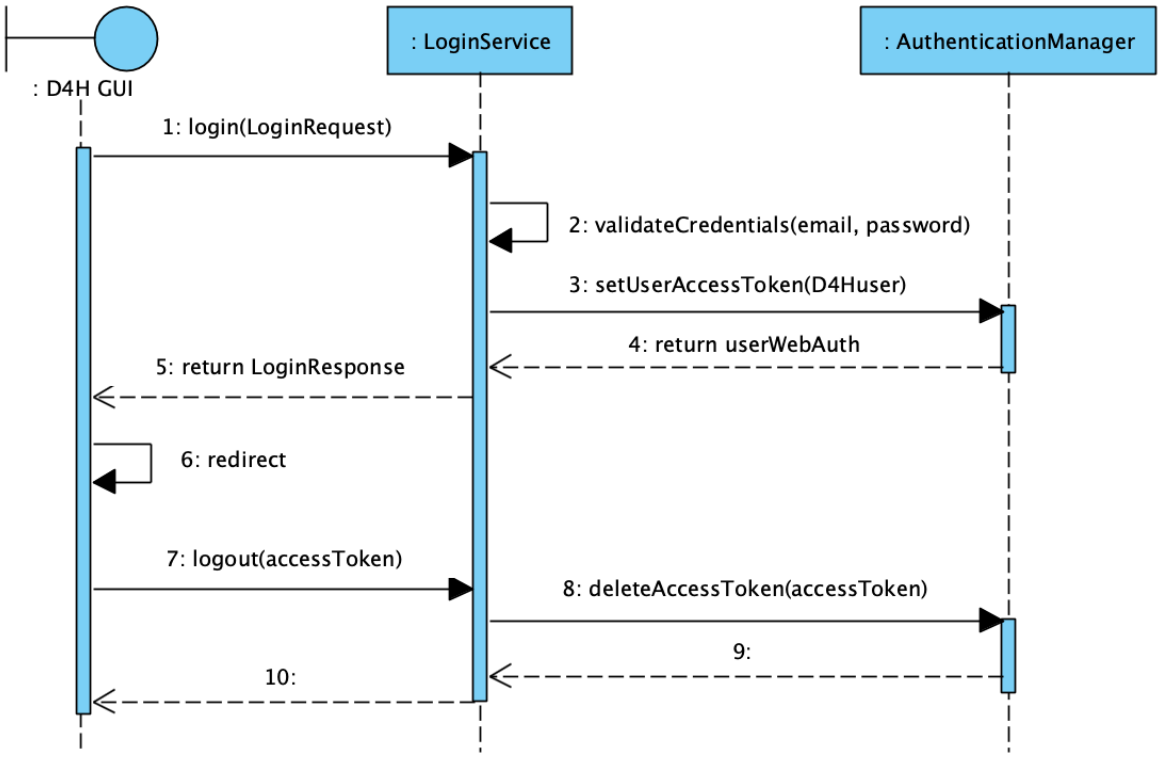


Figure 2.12: Login and Logout Sequence Diagram

- **Get All Requests**

  After login into the system, any individual can check all requests he received from third parties. The **:D4HRequestService** component is the responsible for this process. It gets the requests by means of the **:RequestResource** and returns them to the **GUI** (Figure 2.13).

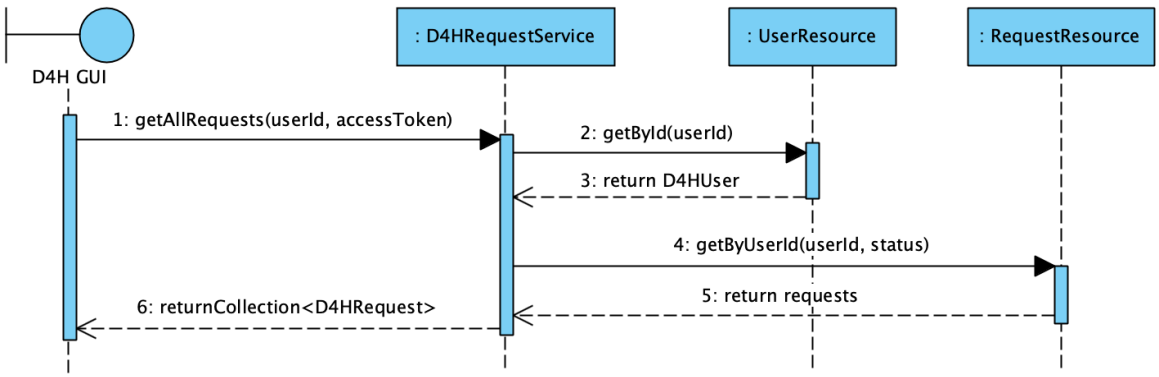


Figure 2.13: Show Requests Sequence Diagram

- **Manage Requests**

  The **:RequestService** component provides to each individual a way to manage his

own requests (i.e. accept or reject them). In both cases the status of the request is updated to *APPROVED* or *REJECTED* respectively. If the request was accepted, the component creates a subscription allowing the third party associated to the request to receive new data. Otherwise, the request is deleted. In both cases, the third party who created the request will be notified (Figure 2.15).

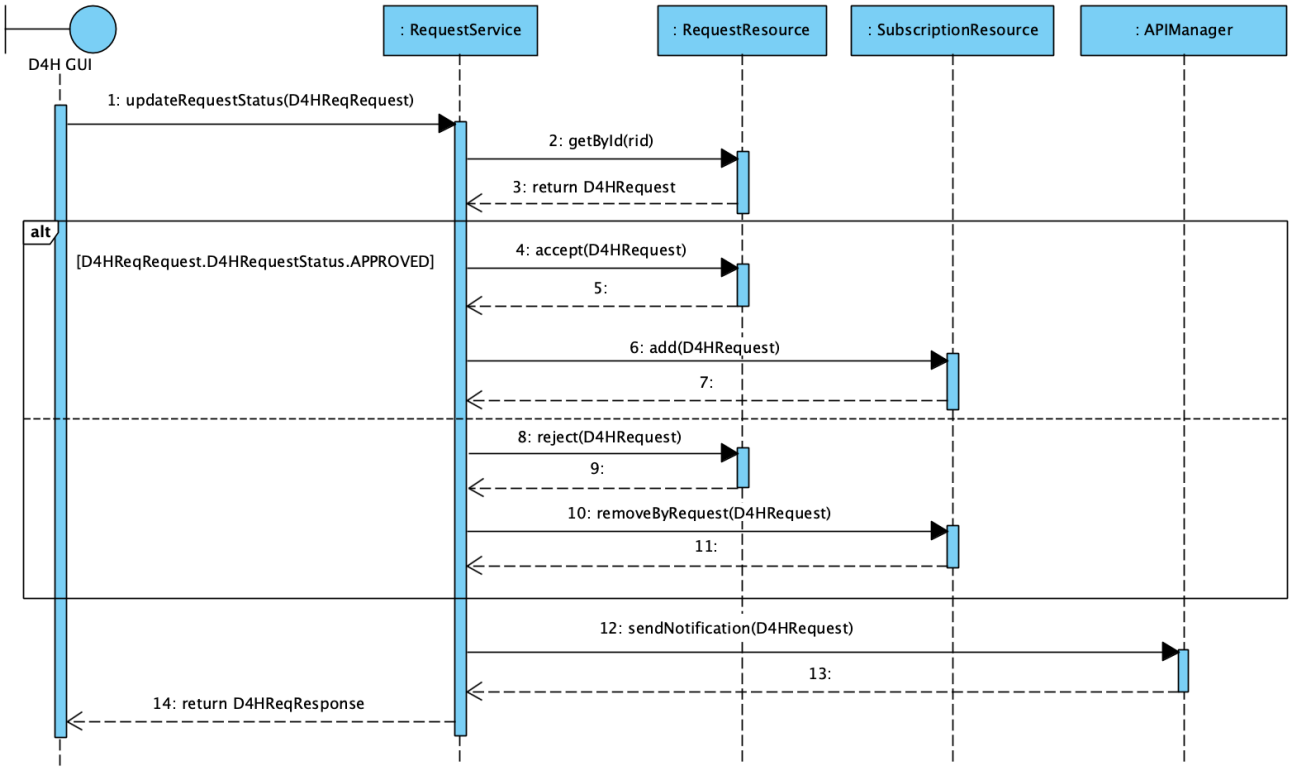


Figure 2.14: Manage Requests Sequence Diagram

- **Notify Third Party**

  Notifications to third parties are handled by the **:APIManager** component. In detail, **:D4HRequestService** asks :APIManager to send a notification to a given third party, to do so this component asks to the :**DBManager** the *APIConfiguration* (of the third party) that contains the URL to which the message will be posted.

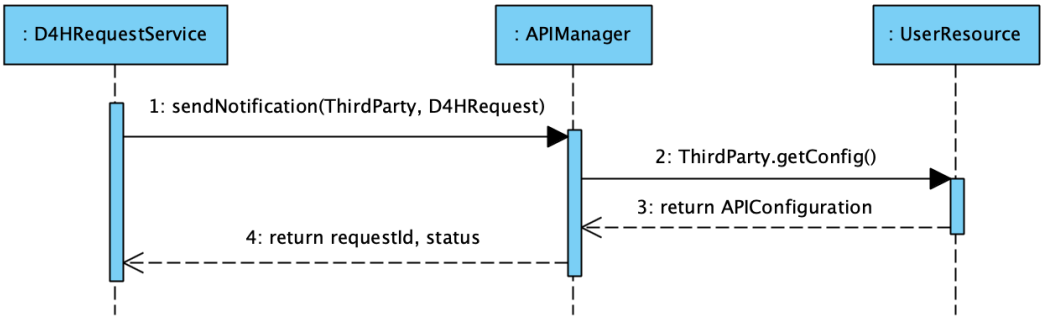


Figure 2.15: Notify Third Party Sequence Diagram

- **Send Request**

  As concerns to third parties, they can send requests (Figure 2.16) to individuals in order to have future access to their data. All this process is handled by **:D4HRequestService** component.

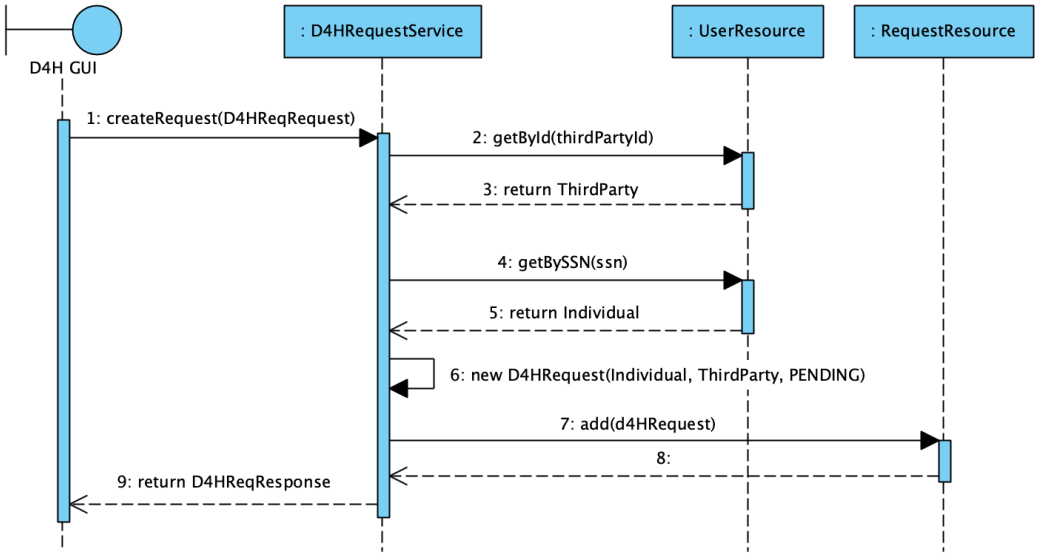


Figure 2.16: Data4Help Send Request Sequence Diagram

- **Search for individual or bulk data**

  Third parties can search for data of a specific individual or bulk data. If a third parties has approved requests to data of any individual, he can search for that data using the ssn of the individual. This operation is carried out by the **:SearchService**, which provides to the **GUI** information concerning the request.

  If third parties want to search for bulk data, they have several available filtering criteria. All those filters are received by the **:SearchService**, which returns to the

**GUI** the data anonymized (Figure 2.17) .

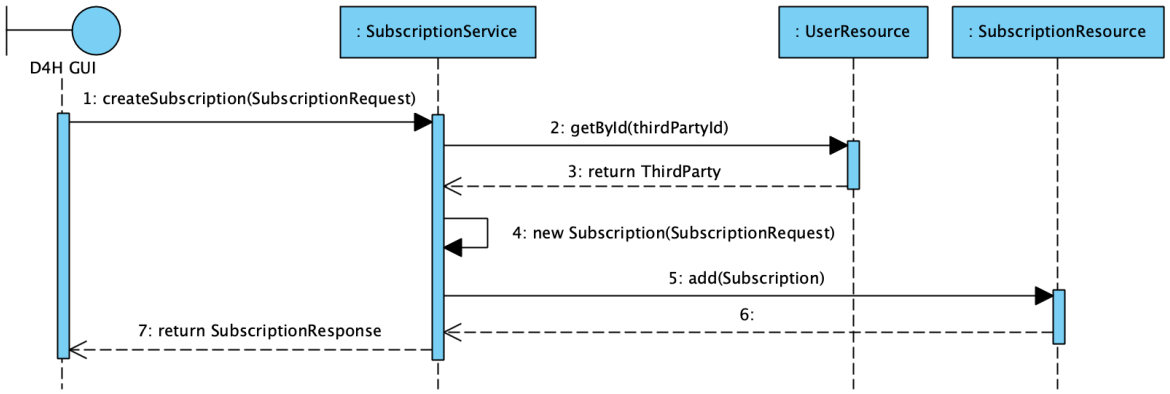


Figure 2.17: Data4Help Access Data Sequence Diagram

- **Subscribe to data**

  Third parties can subscribe to a specific search, this way he will get new data according to the the *time span* specified by the third party (Figure 2.18).

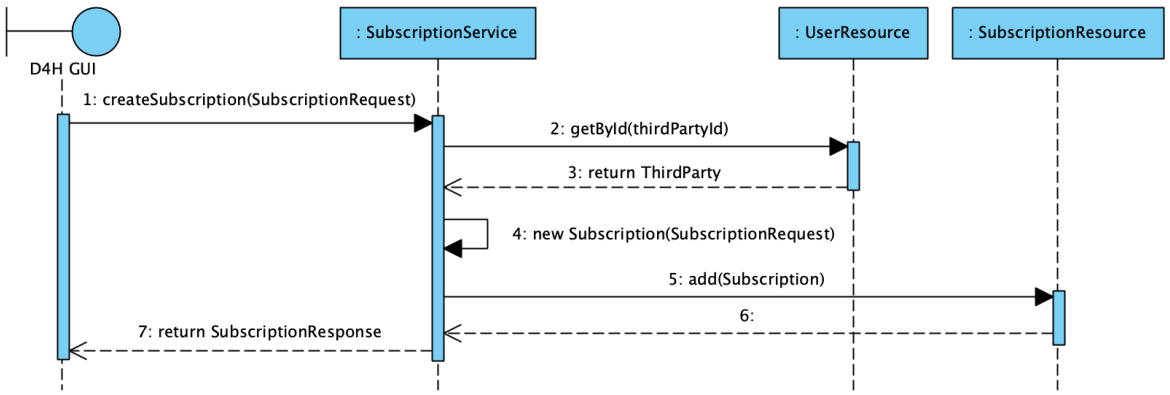


Figure 2.18: Data4Help Subscription Sequence Diagram

### 2.6.2 AutomatedSOS sequence diagrams

Since ASOS is an event-based service, it does not provide a GUI. That is why the main "actor" is Data4Help, which is the boundary of a different service. In the Figure 2.19 the *Get Request Notification* use case is shown. As mentioned before, the component that initiates the process is Data4Help, which sends a message every time an Individual accepts or rejects the ASOS request. If the individual has accepted the request, the **:DataService** component will update the information of the Individual in the database, and will assign an emergency contact based on its

Address.

On the other hand, if the individual has rejected the request, :DataService component will remove it from the database.

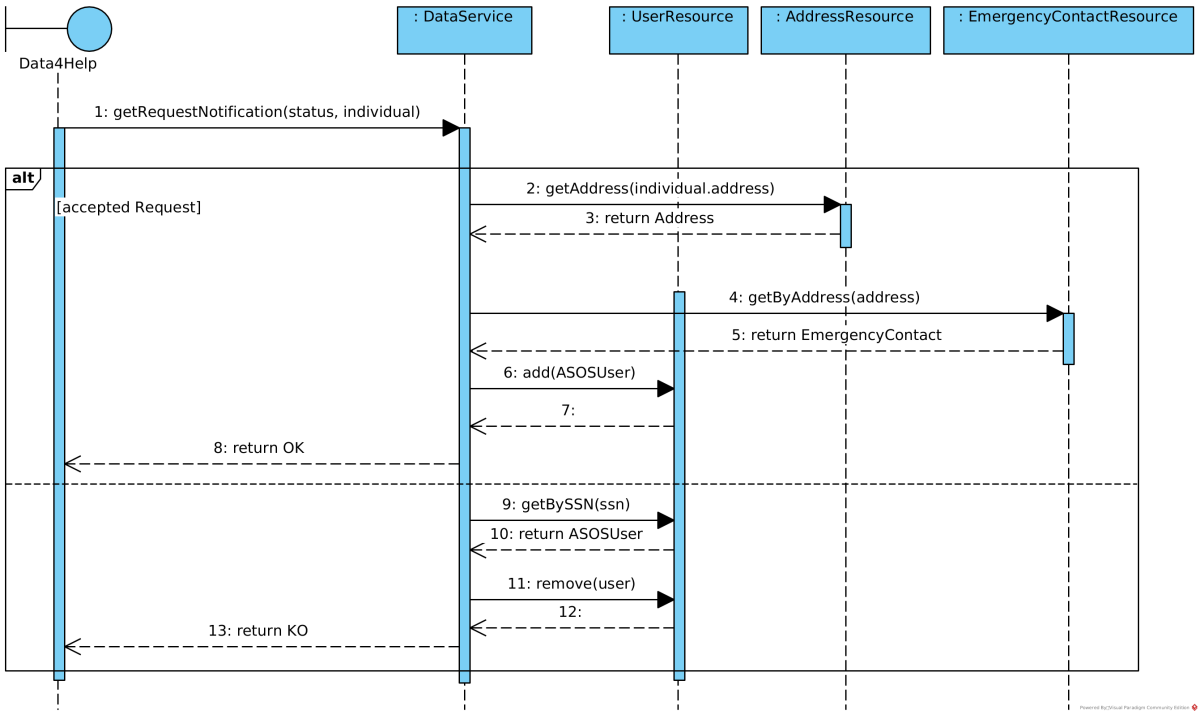


Figure 2.19: AutomatedSOS get notification sequence diagram

Finally, the core functionality of ASOS is shown in the Figure 2.20. It can be seen that, as before, the "actor" that initiates the process is Data4Help service, which sends the Individual health data. This message is handled by **:DataService** component, which will compare the information with the previously defined thresholds. If the Data is out of range, the :DataService will send a message to the **:APIManager** component in order notify the emergency contact assigned for the Individual. The :APIManager component is responsible of getting the assigned emergency contact information, and notifies the **EmergencyService**.

It worth mentioning that the notification to the health-care service is done asynchronously. On the other hand, if the data is ***not*** out of range, the system will do nothing.

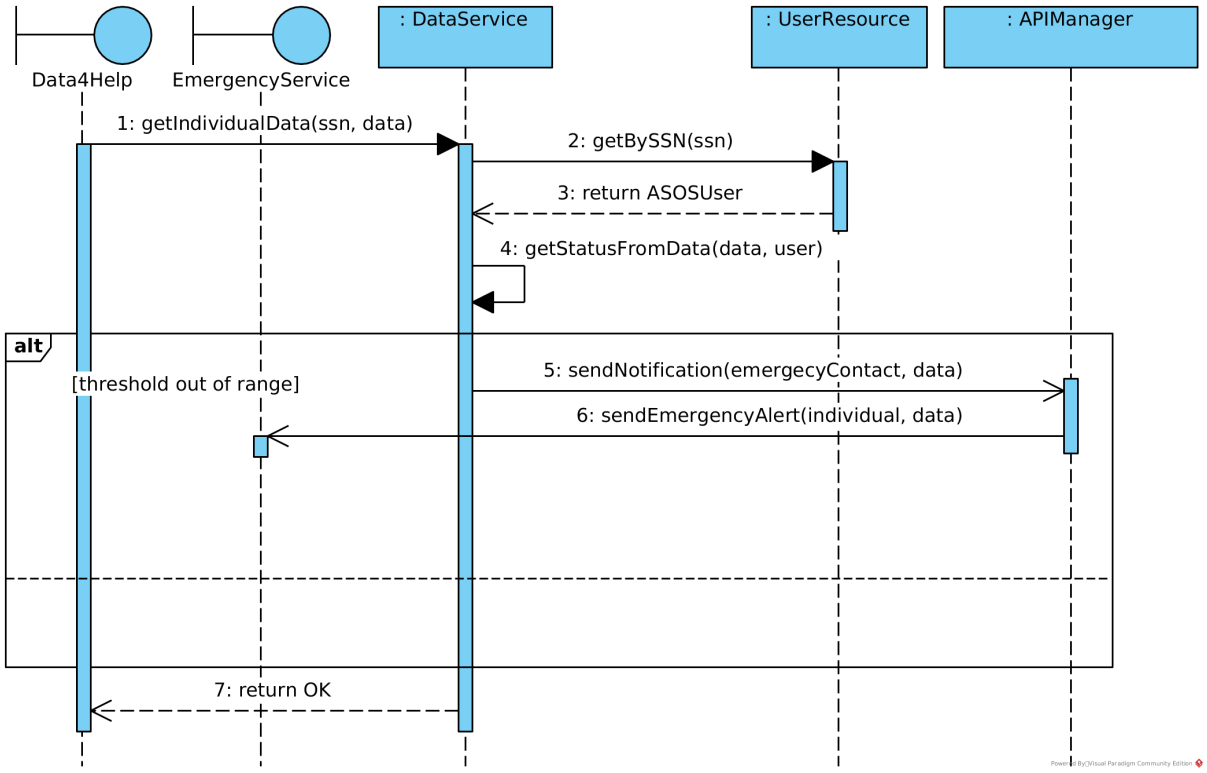


Figure 2.20: AutomatedSOS receive data and notify emergency service sequence diagram

### 2.6.3 Track4Run sequence diagrams

In the Figure 2.21 the signup process is shown. It can be seen the **:Signup** component, which is responsible of getting the participant's information, and with the help of the**:DBManager** component, save it into the database.
The **:AuthenticationManager** component is responsible of create the access token for the session. Finally, the :Signup component sends a message to the **:Request** component which, asynchronously, sends a request for accessing the individual's information to D4H. The process ends when the **GUI** redirects the new user to its dashboard.

It worth mentioning, that the organizer signup process is similar to the signup process of the Participant. The only difference is that, in the case of the Organizer, the **:Signup** component does not send a message to the **:Request** component.

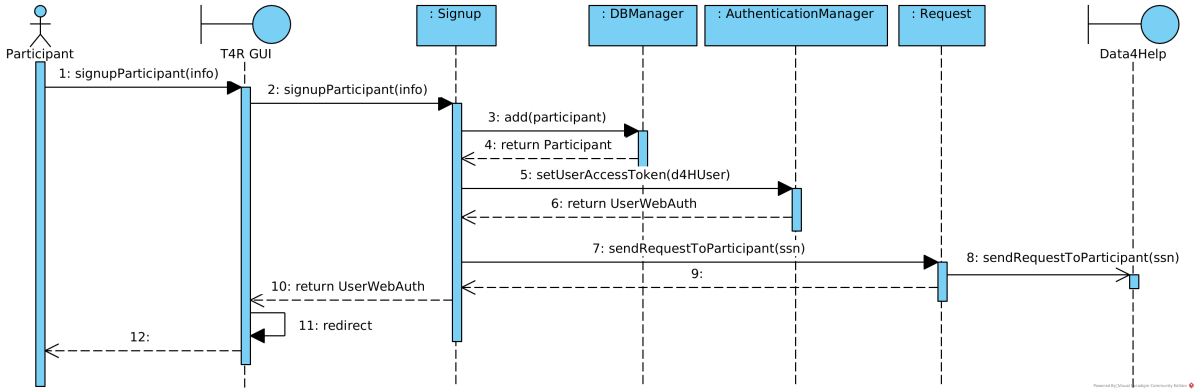


Figure 2.21: Track4Run Signup sequence diagram

In the Figure 2.22 the login and logout processes are shown. It can be noticed that the actor is either a participant or an organizer, since the process is the same for both of them. If there exist a user that has an email and password as the one sent to the **:Login** component, then it should ask the **:AuthenticationManager** component to create an access token for that user. If the credentials are right, the **GUI** will redirect the user to its dashboard. Otherwise, it will show an error.

In the case of logout, the GUI should send the access token and the userId to the **: Login** component, which will send a message to the **:AuthenticationManager** component in order to validate it. If it is valid, it will remove the access token in order to remove the session, and the **GUI** will redirect the user to the home page.

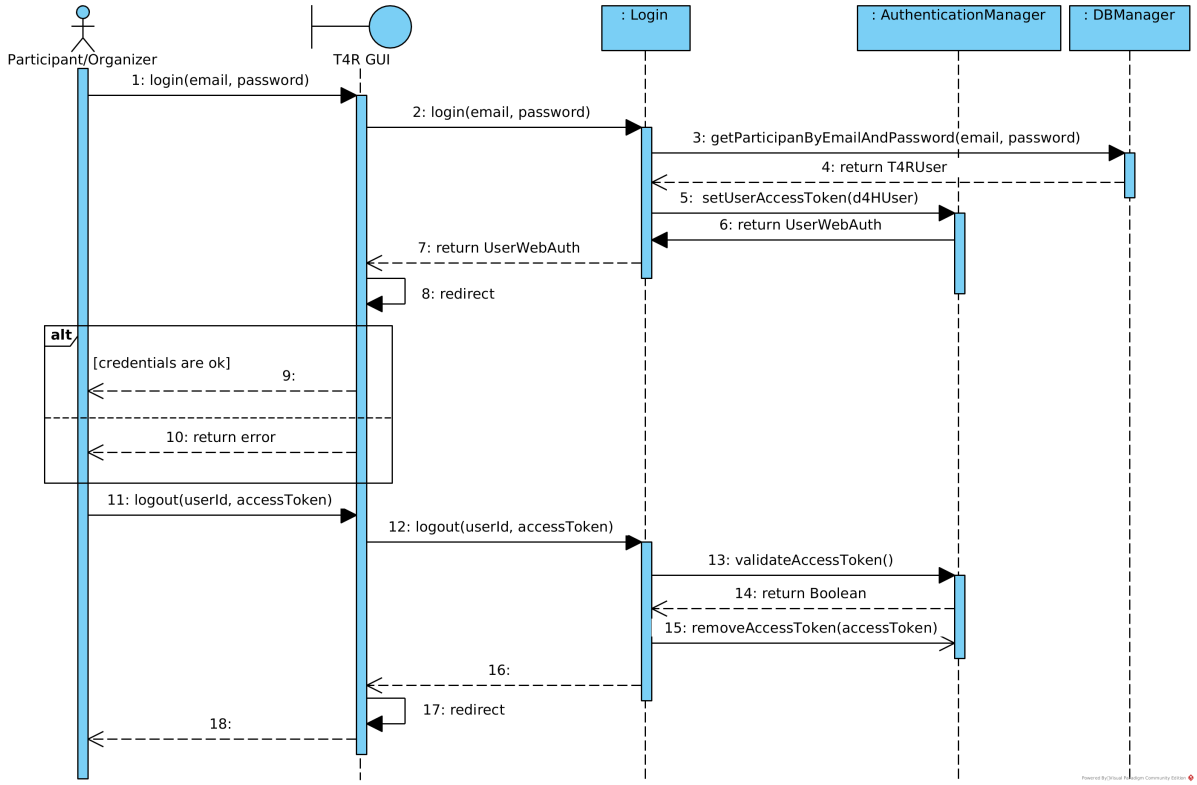


Figure 2.22: Track4Run Login/logout sequence diagram

In the Figure 2.23 the create a run process is shown. This process involves the organizer, who should send the startDate, endDate and name for the event. These parameters will be handled by the **:Event** component, which will save the new event in the database with the help of **:DBManager** component. Moreover, the organizer can add the coordinates for the running circuit by updating the previously create event. After this step, the **GUI** will show the possible participants to invite, and the organizer can skip this, or choose and send the invitations to selected participants. In that case, and after choosing which participants will receive an invite, the GUI will send a message to the **:Notification** component, which will add the notifications into the database. This final process is asynchronous.

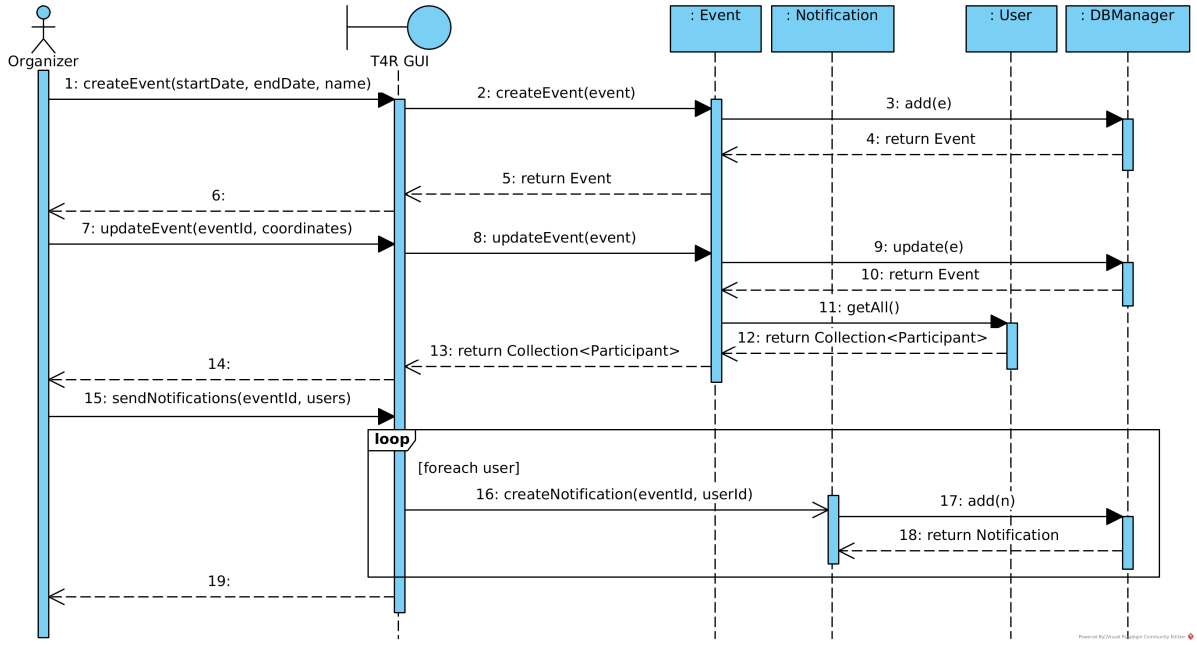


Figure 2.23: Track4Run create run sequence diagram

Finally, in the Figure 2.24, the accept and reject invitation processes are shown. First, the participant will ask the **GUI** for all the pending notifications, and it will ask to the **:Notification** component for the notifications of the user. If a user decides to accept a notification, the GUI will send a message to the **:Event** component, which will add the participant to the :Event.

On the other hand, if the participant decides to reject the invitation, the GUI will send a message to the :Notification component in order to delete it.

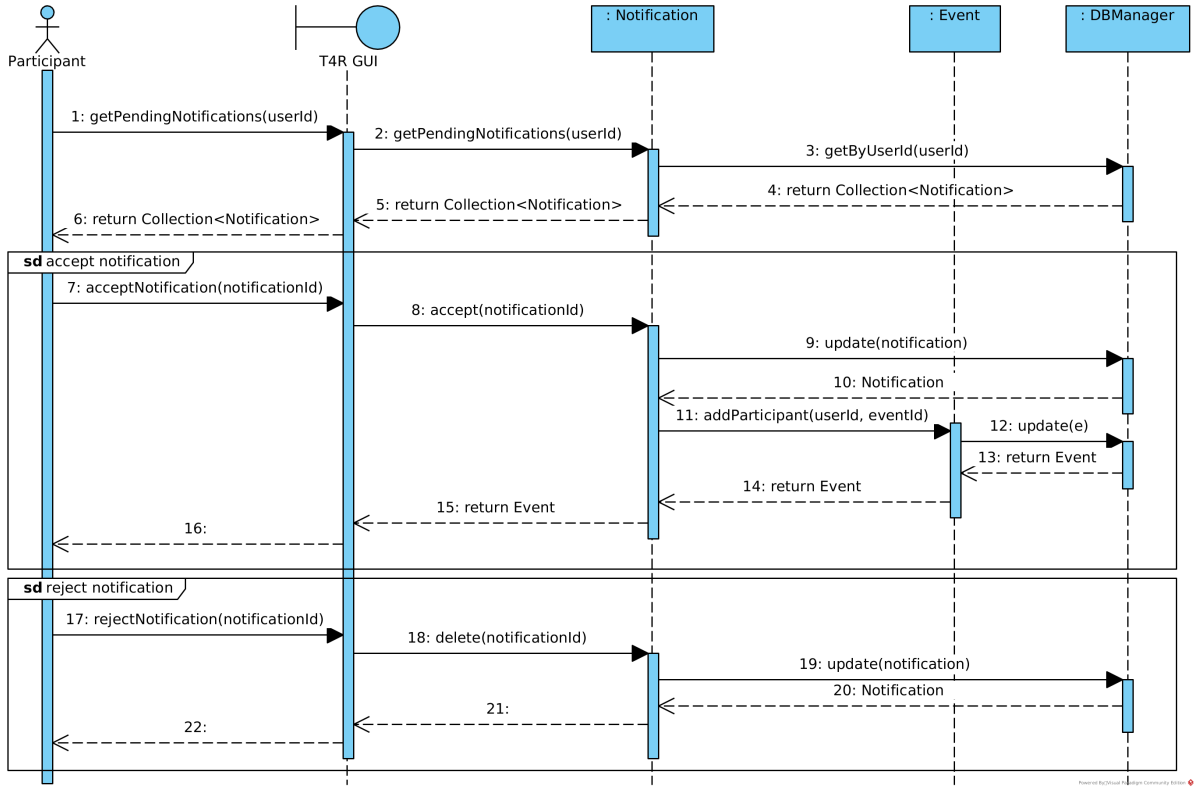


Figure 2.24: Track4Run accept/reject invitation sequence diagram

## 2.7 Selected architectural styles and patterns

Resuming the high level architecture of the system briefly explained on *section 2.1 Overview*, it is important to describe MVC and event-driven design patterns, how they work and why are the proper ones to build up each subsystem.

### 2.7.1 Model View Controller architecture (MVC)

The Model View Controller (commonly known as MVC) design pattern helps to build applications that are easier to test and maintain, is the way of structuring server side code It comprises of three major components:

- **Model**: this is the layer that represents the application's data and define the logic for manipulating that data.
- **View**: this represents the presentation or the user interface layer, something visible in the user interface.
- **Controller**: this layer typically contains the business logic of your application, and acts as a mediator between model and view components.

In other words, the *model* is the part of the application that is responsible for the logic needed for the treatment of data. Normally model objects retrieve data (and store data) from a database. The *view* is the parts of the application that is responsible for the visualization of the data. Usually views are created from the model data. The *controller* is the part of the application that is in charge of the interaction with the user. Normally, drivers read data from a view.

MVC design pattern is perfect for D4H and T4R systems because it is based on *separation of concerns*, this makes the application's code easier to test and maintain. In a typical MVC design, the request first arrives at the controller which binds the model with the corresponding view. This separation helps manage complex applications, because the developer can focus on one aspect at a time and also simplifies the development of group applications, different developers can work in parallel.

### RESTful web services

The communication between D4H/T4R with their users is done via HTTP requests following REST principles. REST (Representation State Transfer) is an architectural style for communication based on strict use of HTTP request types (Figure2.25) .

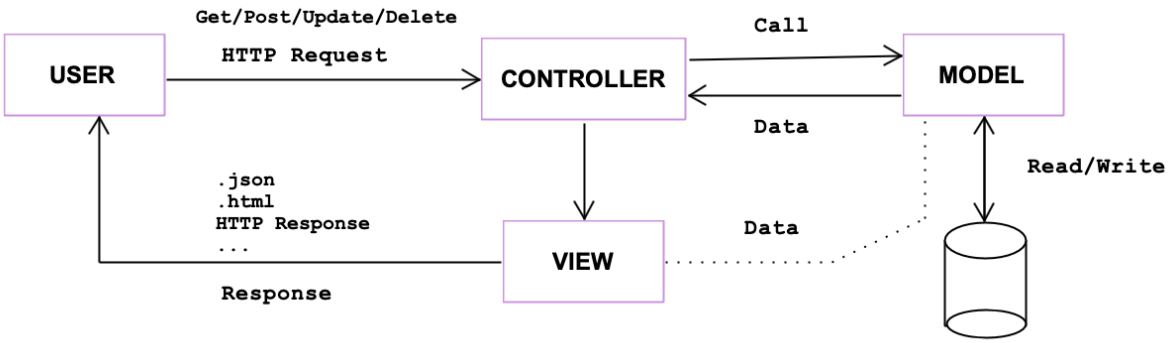


Figure 2.25: RESTful web services

One of the most important REST principles is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. The client wouldn't notice if the server were to be restarted at any point between the requests.

On the server side, the application state and functionality are divided into resources. A resource is an item of interest, a conceptual identity that is exposed to the clients.

Example resources include application objects, database records, algorithms, and so on. Every resource is uniquely addressable using a URI (Universal Resource Identifier). All resources share a uniform interface for the transfer of state between client and server. Standard HTTP methods such as GET, POST, PUT, and DELETE are used.

The RESTful HTTP requests are categorized according to method types as the following:

- **GET**: used to retrieve resource representation/information only – and not to modify it in any way.
- **POST**: used to create a new resource into the collection of resources.
- **PUT**: used primarily to update existing resource (if the resource does not exist then API may decide to create a new resource or not).
- **DELETE**: used to delete resources (identified by the Request-URI).

### 2.7.2 Event-driven architecture (EDA)

The event-driven is a popular distributed asynchronous architecture, made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events. Basically, a system sends event messages to notify other systems of a change in its domain and it does not really care much about the response. EDA are useful to simplify data management, so as to tackle real-time processing with minimum time lag (high volume and high velocity of data).

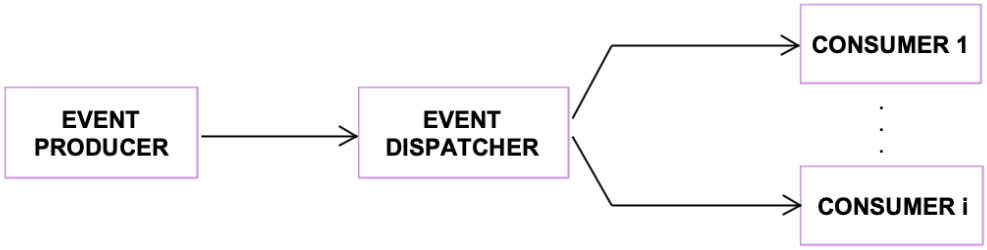


Figure 2.26: Event driven architecture

For ASOS, a simple event processing will be implemented, this means that an event immediately triggers an action in the consumer. This is, after receiving Individuals' data, and based on the defined thresholds it decides whether to notify or not their associated health care services.

## 2.8 Other design decisions

In the following section a list of suggested development frameworks and technologies are listed. The 3 systems, are designed to be highly decoupled so it is not necessary to develop them with the same technologies, even though it is recommended. Further details will be given in future documents.

To begin with, it is suggested to build Java based applications, since developers have expertise working with this language. Moreover, it is proposed to use **Java Spark Framework**, which is a micro-framework for MVC applications, because pure Java web development has traditionally been very cumbersome. The configuration for this framework is minimal and with a short learning curve.

Furthermore, for database management a suggestion is to use **Morphia** which is a highly active project to connect with MongoDB. Also, it is simple, straightforward to use, it supports type conversion and uses MongoDB aggregation framework. While for the Redis database, it is recommended to use **Lettuce**, which is a fully non-blocking Redis client, that provides reactive, asynchronous and synchronous data access. It has also, a highly active community and a very well written documentation.

Besides, for the front-end development, it is suggested to use the last release of **Angular** (version 7.x), which is a very well known and widely used framework developed by Google. It is suggested to use Google Maps platform for displaying the running circuit and the participants in the T4R web-site.

Finally, the only external APIs to whom TrackMe environment depends on, are the health-care services to which ASOS should contact in some cases. Both D4H and T4R do not depend on any external services.

Chapter 3

# User interface design

The user interfaces were introduced in the RASD, however in this section they are examined in depth.

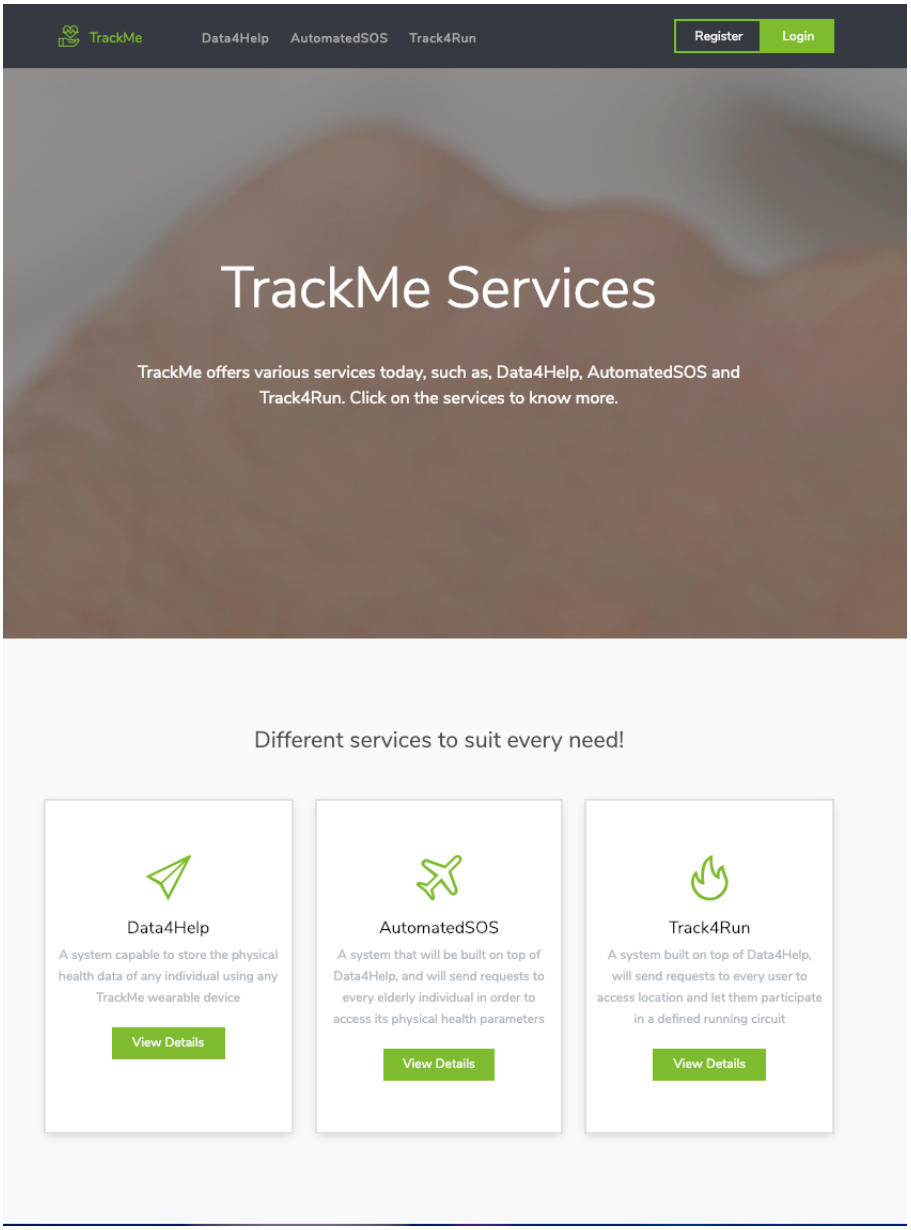


Figure 3.1: TrackMe's Home Page

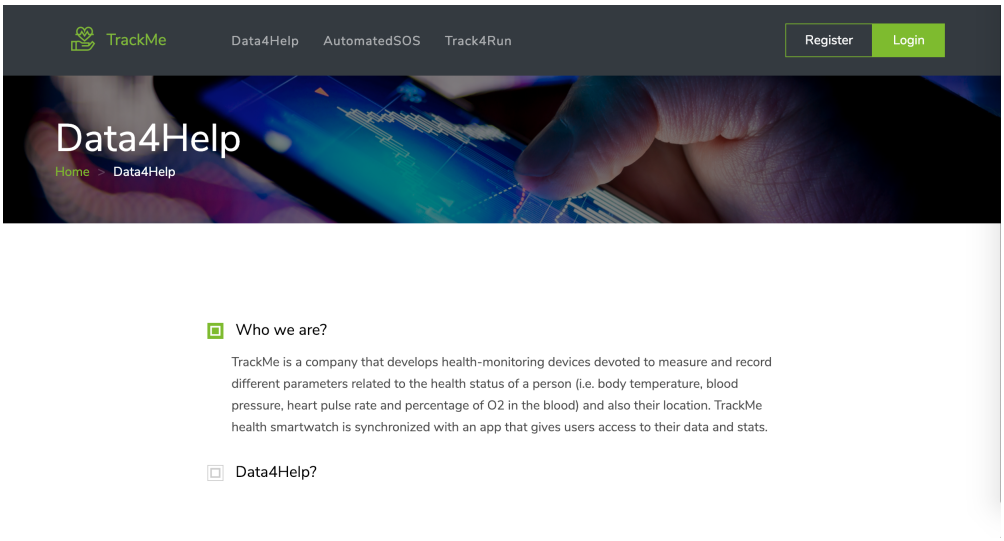


Figure 3.2: Data4Help Information Page

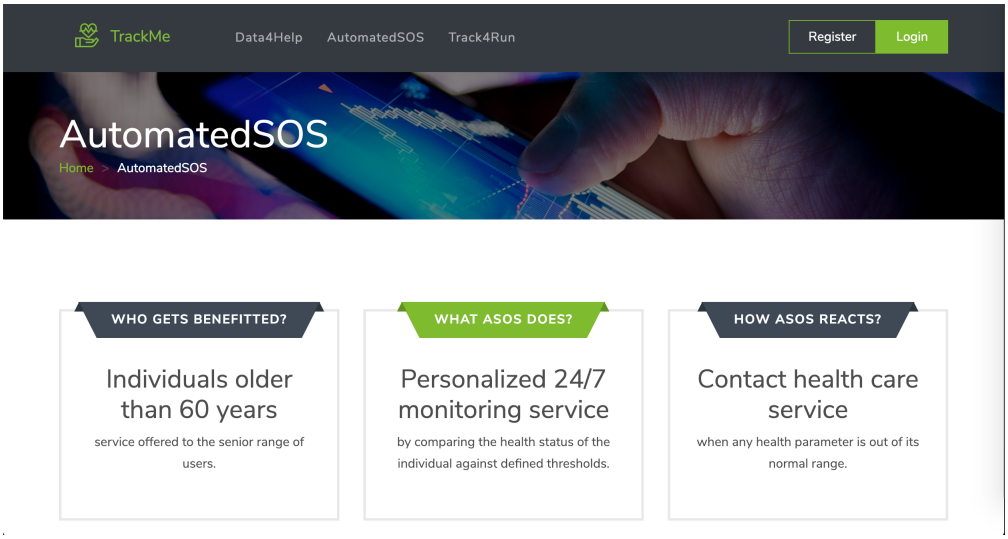


Figure 3.3: ASOS Information Page

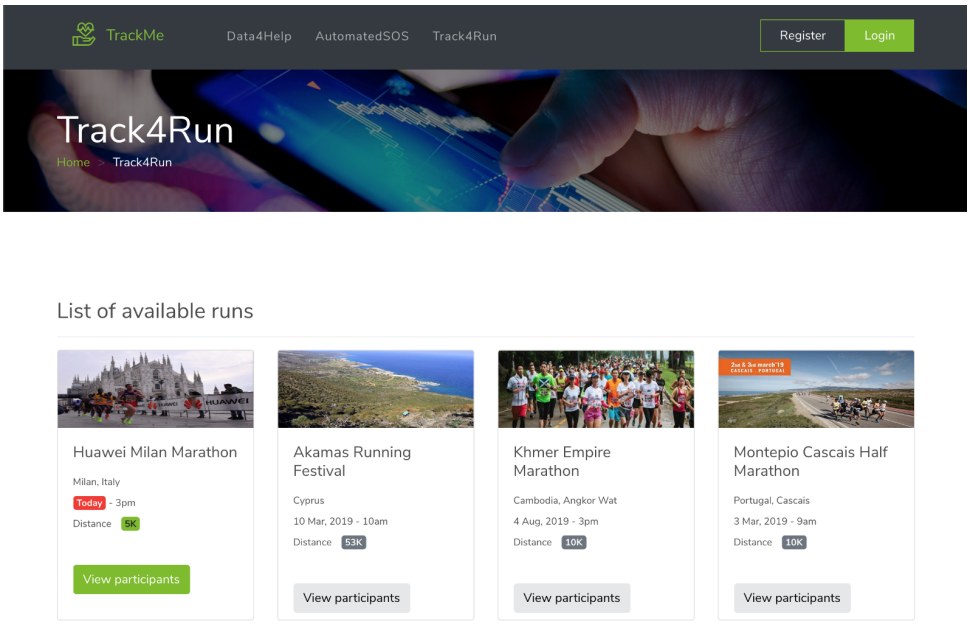


Figure 3.4: Track4Run Web Page

In the Figure 3.4, Spectators are able to visualize all the available runs in which are currently going on; and by clicking on 'view participants' button, they are redirected to the map-view of the selected run, where they can view the live location of the participant users.
In the Figure 3.5, can be seen the web page through which users can Login into the system (if already registered).

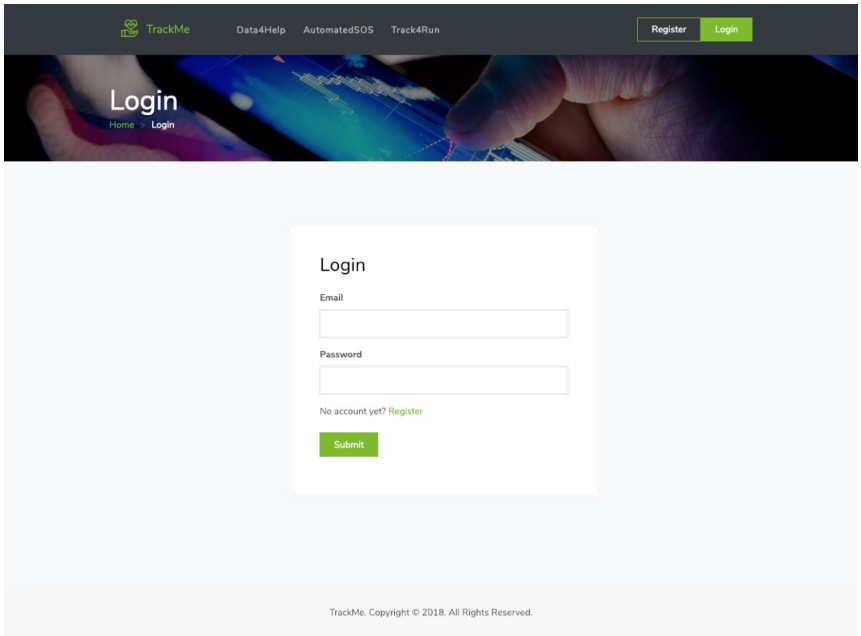


Figure 3.5: Login Page

And if not already registered into the system, they can register themselves through the Register web-page (Figure 3.6). There are separate register forms for the Individual users and for Third Party users.

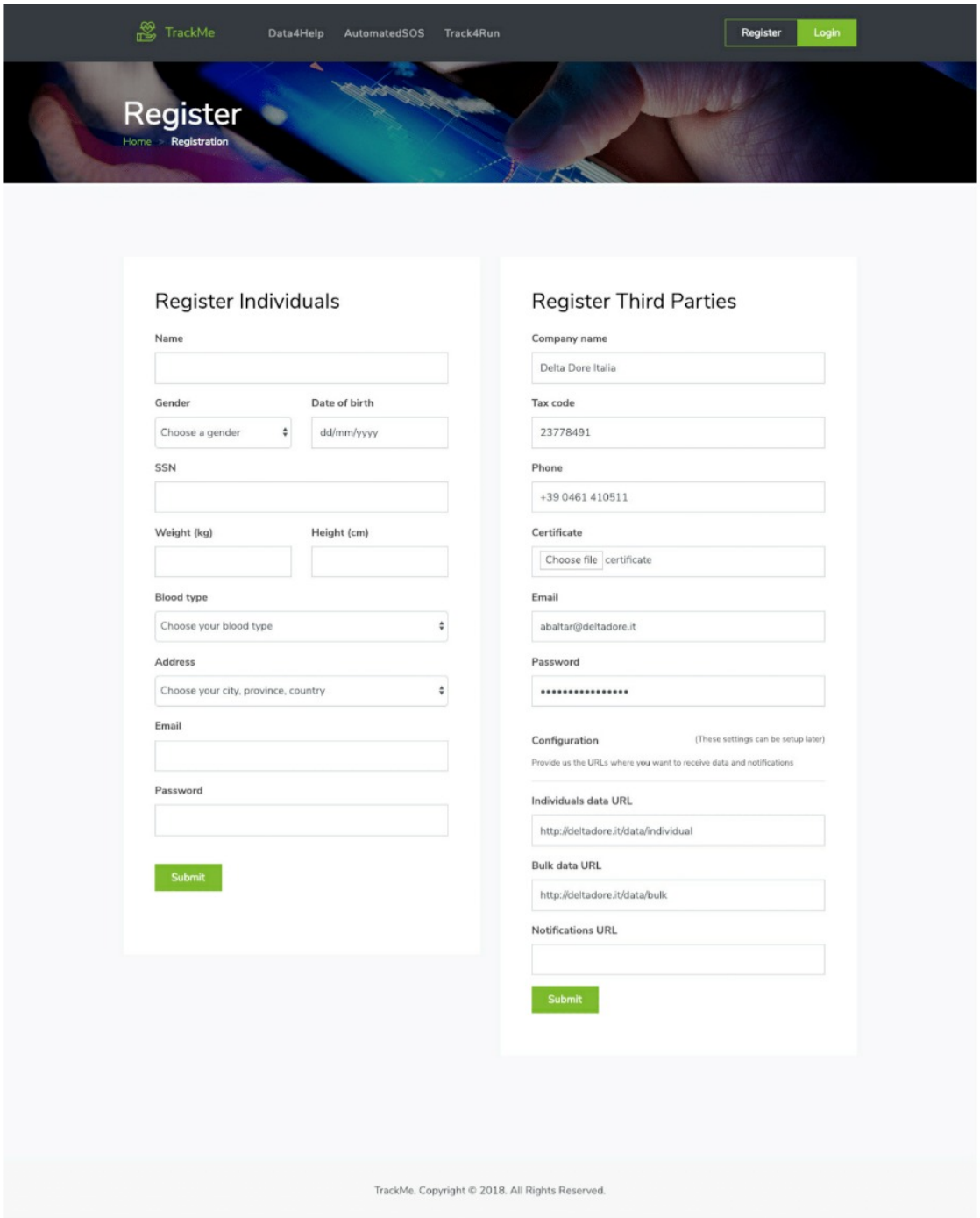


Figure 3.6: Registration Page

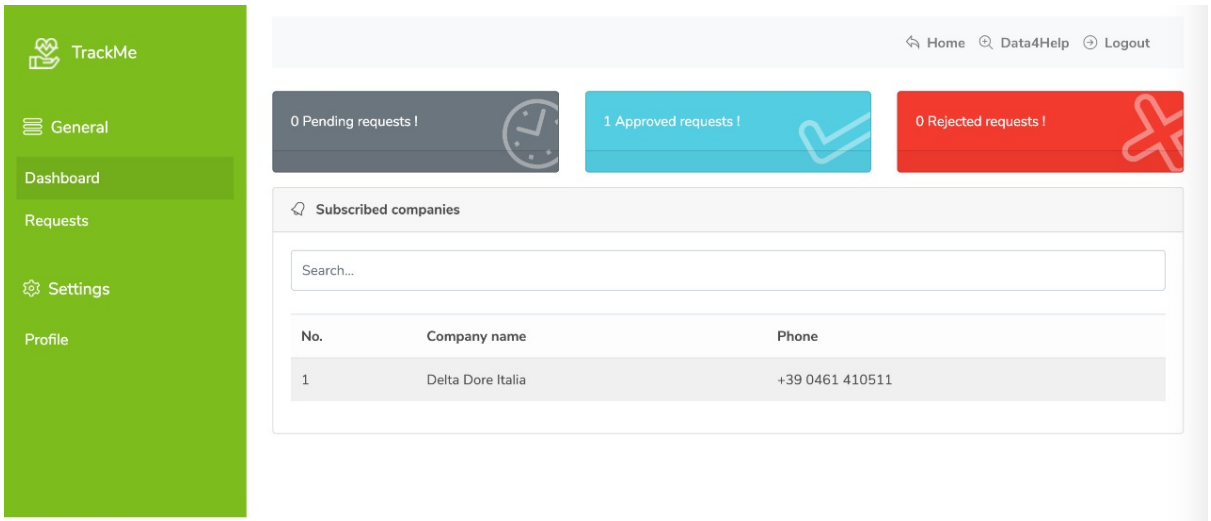


Figure 3.7: Individual Dashboard Page

The above Figure 3.7 displays the dashboard for individual users, this page is responsive and a real-time view of the application for the individual user's point of view. Here, the user can seen his/her pending requests, accepted requests, and a list of all the requests he/she approved or rejected.

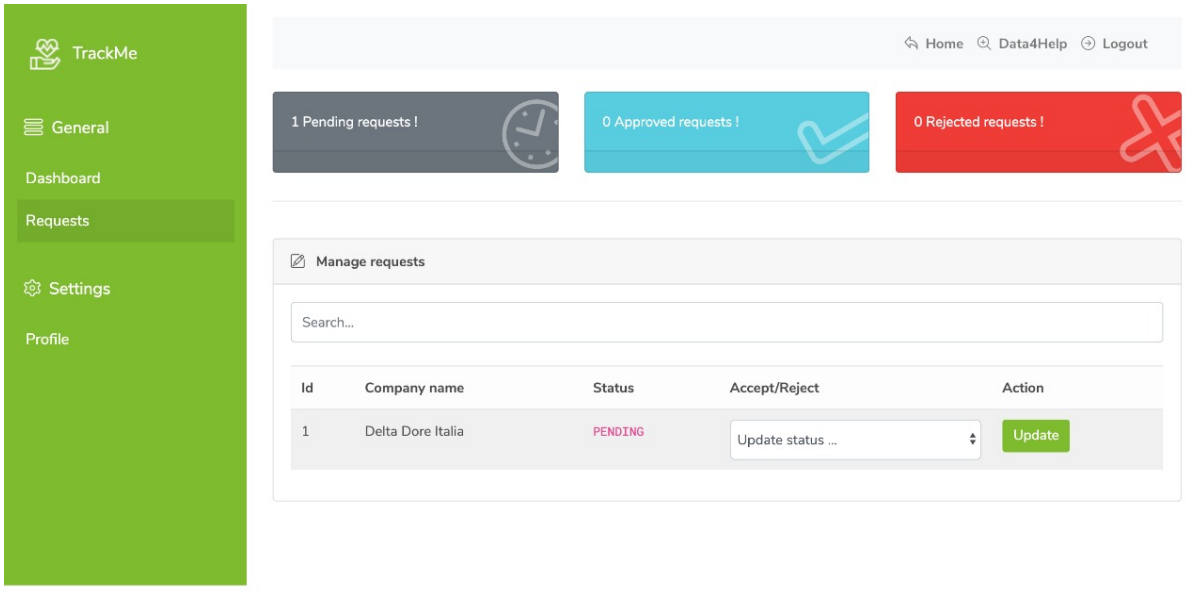


Figure 3.8: Manage Requests Page

In the Figure 3.8 the individual user have options to accept or reject any request for data acquisition. As soon as an action is taken upon the request, it gets deleted from the page.

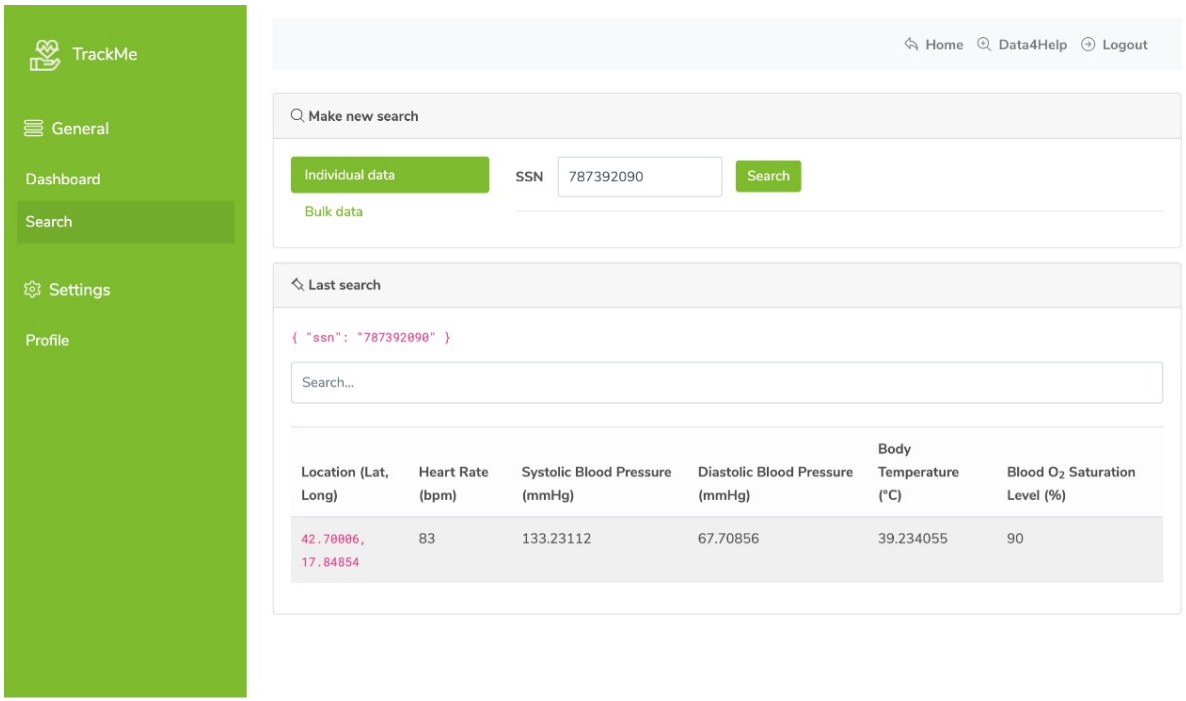


Figure 3.9: Third party Dashboard Page 1

The above Figure 3.9, displays the dashboard for third party users, this page is responsive and a real-time view of the application for the third party user's point of view. Third parties can search for data using the filtering criteria provided by the system such as country, province, city, age, blood type, or SSN.

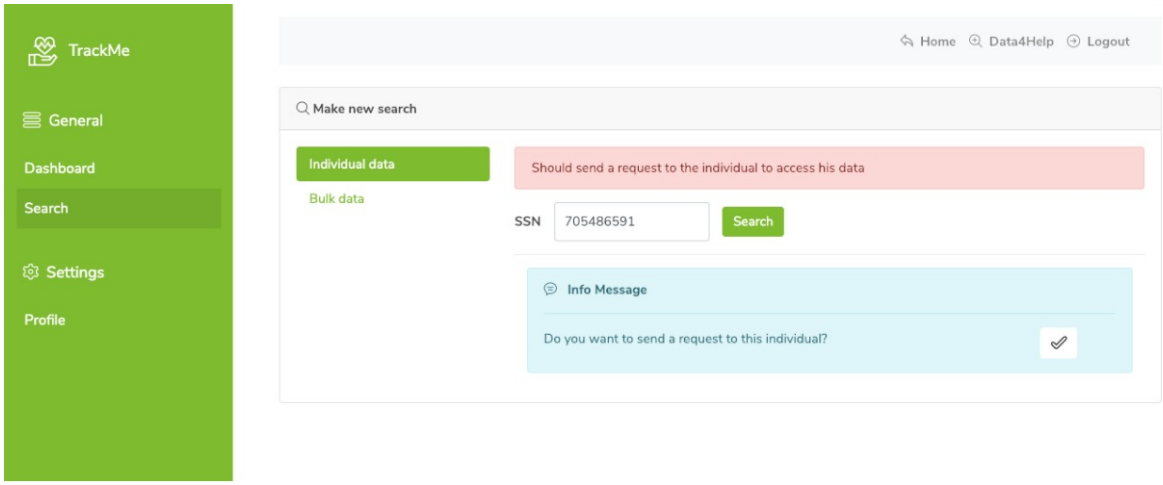


Figure 3.10: Third party Dashboard Page 2

The above figure 3.10, states that, if they click on the 'Check' sign then the request is sent to the individual which has the same SSN entered by the third party.

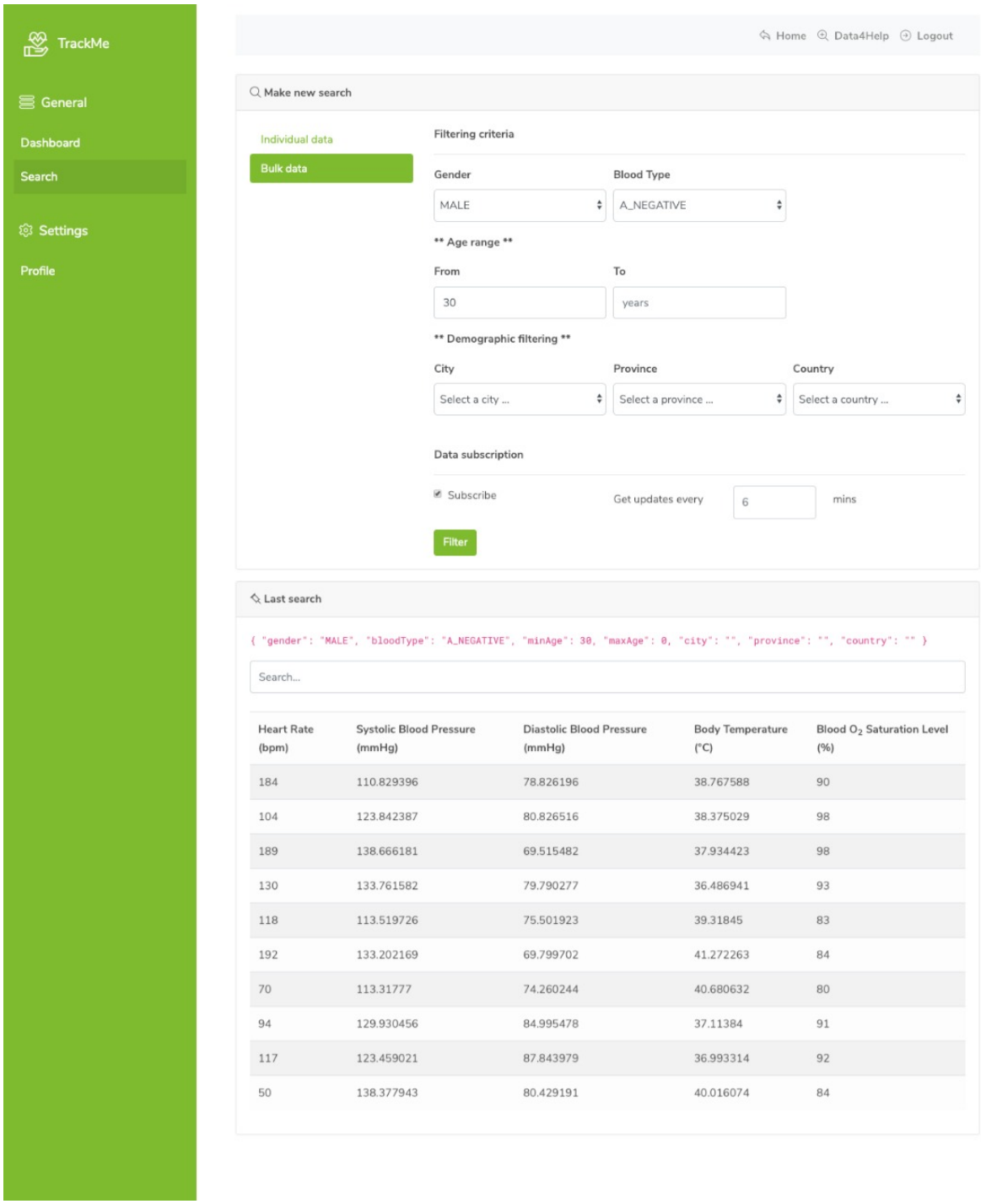


Figure 3.11: Third party Dashboard Bulk Request Page

The above figure 3.11, user s able to select the filter criterias according the which user wishes to get the data. Filter criterias are such as gender, age, demographic findings or the date of subscription. User is able to get filtered data, if already there (from previous requests).

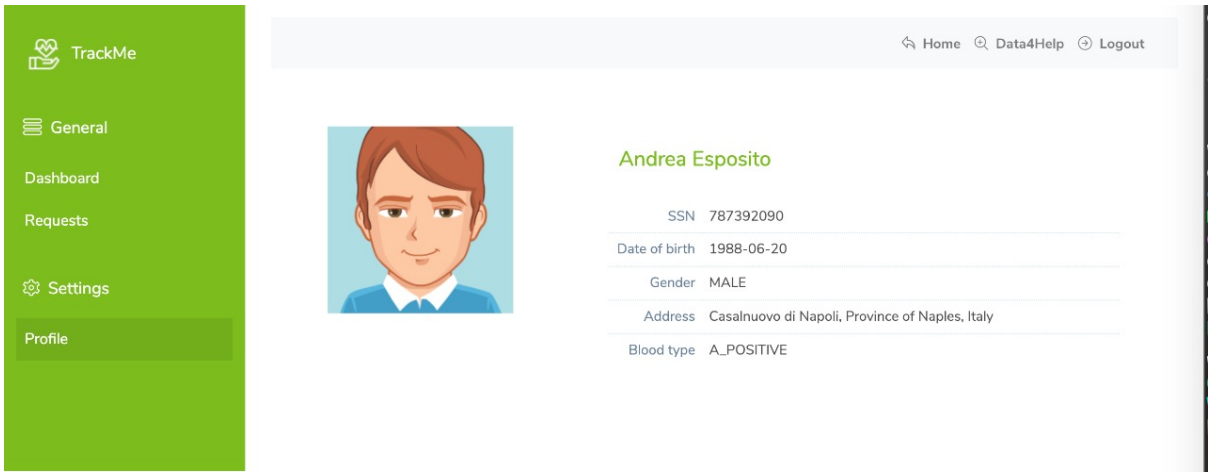


Figure 3.12: Individual Profile page

The above figure 3.12, user is able to view the details of his profile. This profile data is personal to every user. They can only access their own data; details include: ssn, name, address, blood type etc.

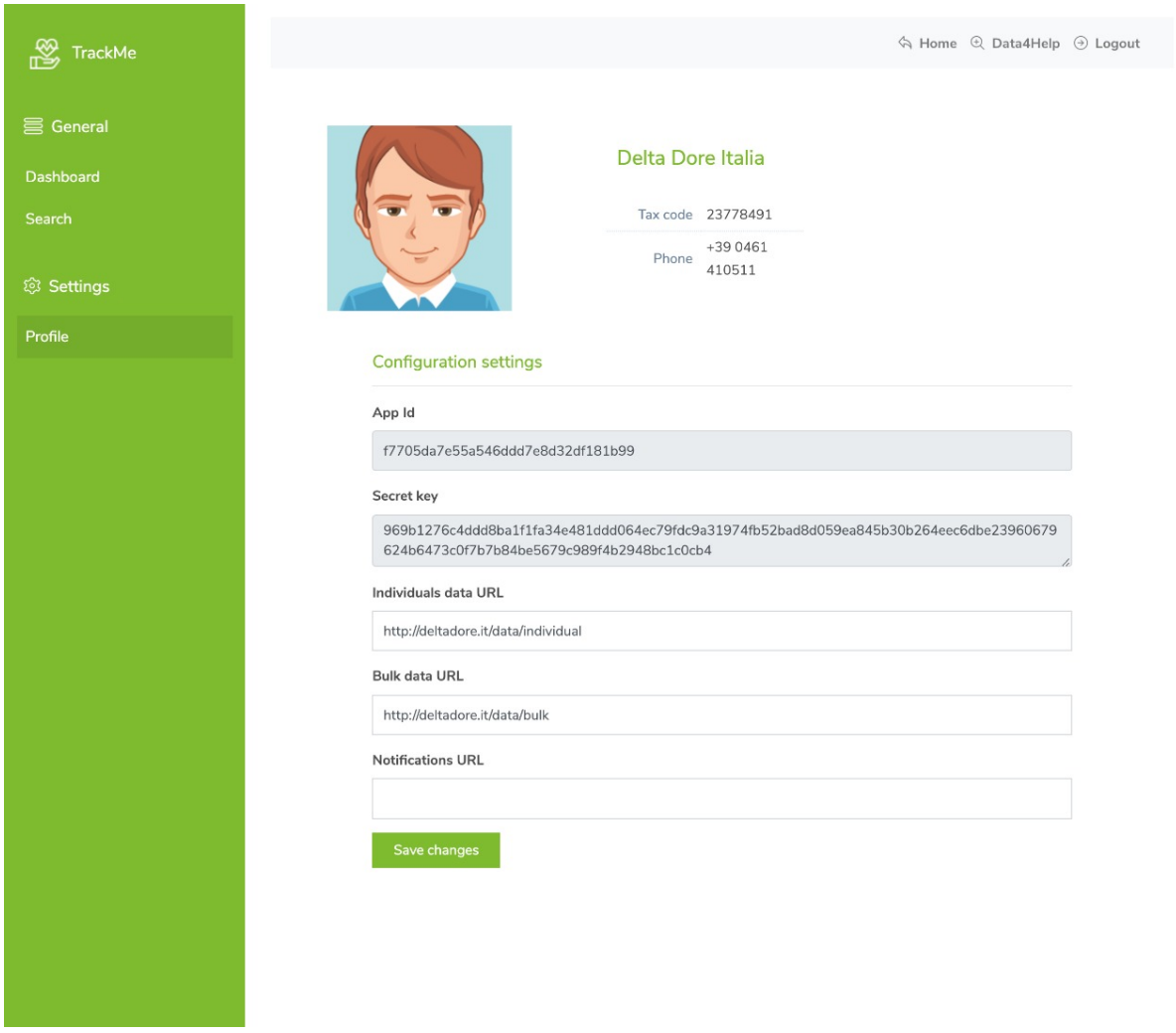


Figure 3.13: Third party Profile page

The above figure 3.13, user is able to view the details of his profile. This profile data is personal to every user. They can only access their own data; details include: tac code, contact and the configuration details (such as App id, secret key, bulk data and individual data URLs etc.

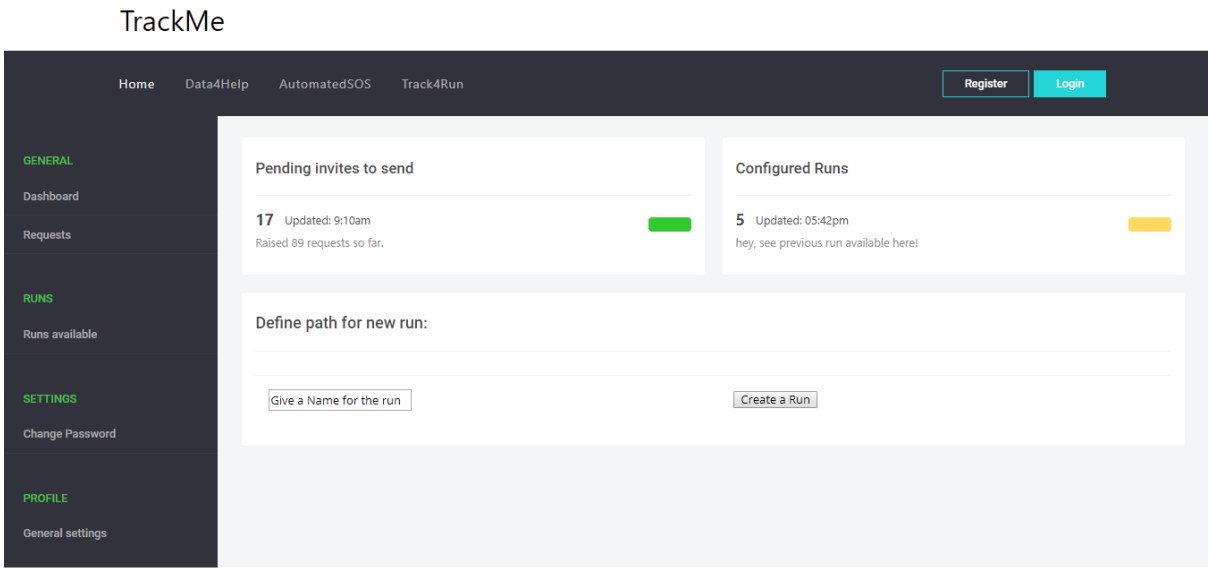


Figure 3.14: Organizers' Dashboard Page

In the Figure 3.14, the organizer can see the number of pending invites for the already configured runs which are still pending to send to the targeted participants; also, organizer's are able to view the number of already configured runs from the past; In addition, they can start configuring a new run by clicking on 'Create new run' button.

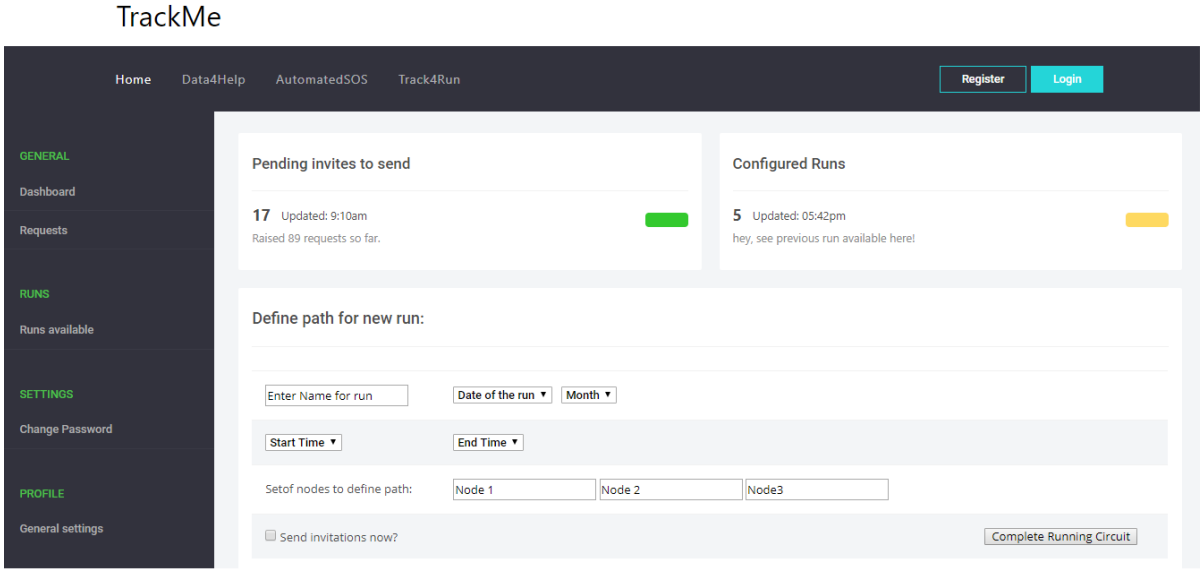


Figure 3.15: Organizers' Dashboard Page

In the Figure 3.15, Organizer's are able to define the parameters to define a complete

running circuit such as: start and end time, name of the run, date of the run, nodes of the running circuit and here they can select whether to send invitations now or not by clicking the check-box; Thus, by clicking on 'Complete running circuit' button, the configuration gets completed and based on the details above, run is configured.

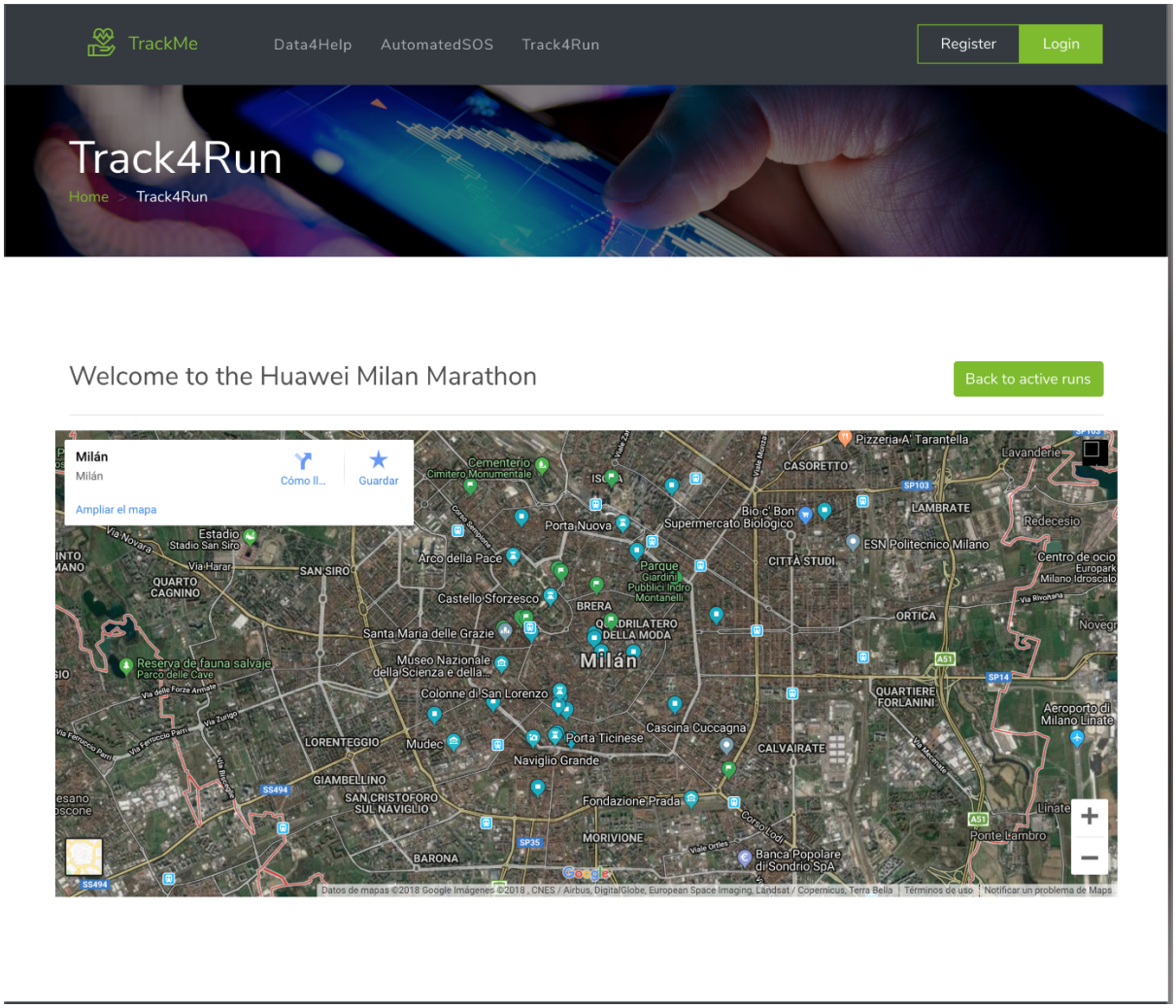


Figure 3.16: Spectators' Map-view Page

In the Figure 3.16, Spectators are able to view the participants' live position only during the run and time left for the race to end, because after the end of this time, spectators won't be able to view the map.

Chapter 4

# Requirements traceability

In this section is shown how the goals and requirements specified in the RASD are mapped to the design components defined in this document.

- **Data4Help**

  [R1] The system must allow an individual to register a new account
  **D4H::SignupService**

  [R2] The system must allow an individual to access to their account
  **D4H::LoginService**

  [R3] The system must allow an individual to accept or reject their requests of accessing personal data
  **D4H::D4HRequestService**

  [R4] The system must be able to communicate with TrackMe database in order to obtain the health status and location of an individual
  **D4H::SearchService**

  [R5] The system must allow a third party company to register a new account
  **D4H::SignupService**

  [R6] The system must allow a third party company to access to its account
  **D4H::LoginService**

  [R7] The system must be able to notify the individual that a third party company wants to access its data
  **D4H::D4HRequestService**

  [R8] The system must allow a third party company to search for an individual health status and location using his/her SSN
  **D4H::SearchService**

  [R9] The system must allow a third party company to filter data of an anonymized group of individuals by country, age, gender and blood type parameters
  **D4H::SearchService**

  [R10] The system must be able to anonymize the data of a group of individuals
  **D4H::APIManager**

[R11] The system must allow a third party company to subscribe to an individual health status and location
**D4H::SubscriptionService**

[R12] The system must allow a third party company to subscribe to data of an anonymized group of individuals
**D4H::SubscriptionService**

- **AutomatedSOS**

[R13] The system must be able to send a request for monitoring an individual's data when he/she is older than 60 years old
**ASOS::DataService**

[R14] The system must be able to monitor, and compare against defined thresholds, the health status of an individual
**ASOS::DataService**

[R15] The system must be able to contact the health-care service associated to an individual
**ASOS::APIManager**

- **Track4Run**

[R16] The system must allow a participant to register a new account
**T4R::Signup**

[R17] The system must allow a participant to access to their account
**T4R::Login**

[R18] The system must allow an organizer to register a new account
**T4R::Signup**

[R19] The system must allow an organizer to access to their account
**T4R::Login**

[R20] The system must allow an organizer to create a race event
**T4R::Event**

[R21] The system must allow an organizer to define the running circuit of a race event
**T4R::Event**

[R22] The system must allow an organizer to send invitations to participants to enroll in a race event
**T4R::Notification**

[R23] The system must allow a participant to accept or reject an invitation to a race
**T4R::Notification, T4R::Request**

[R24] The system must allow any spectator of a run to view in a map the participants' location
**T4R::DataHandler**

[R25] The system must allow a spectator to click on a participant location in order to view his/her health status
**T4R::DataHandler**

Chapter 5

# Implementation, integration and test plan

## 5.1 Requirements of implementation and testing

In order to start implementation process of the project, RASD and DD must be finished and available for all the teams of the project, so they will be informed about requirements and adopted design choices. A good analysis of each document is also required to be sure that implemented system will be secure and reliable.

**Development Tools** The Java programming language is suggested to develop the system because it suits very well for communication among different platforms and devices. In addition, Java Standard Edition will be used to deploy our business functionalities and to build a reliable and powerful web tier to let users communicate with main server using a standard web browser.

**Testing Tools** In order to achieve all the testing purposes we recommend to use different testing tools:

- For unit testing of each component can be used *JUnit* framework. It's a good framework since it is well tested and supported. It can test each functionality of component into an isolated environment, and can be used also to test some little interaction between components.
- *Mockito* tool is suggested as it is a No expect-run-verify technique. It is another well known and highly used framework, in which the produced tests are very readable.
- *Grinder* software may be used to make load tests on Web Server.
- *HammerDB* may be used to test functionalities and performance of Database server.

## 5.2 Implementation Strategy

Implementation process will follow the bottom-up strategy: we plan to implement first single component isolated. Once at least 60% of a component will be developed, its unit testing phase can start.

The implementation of the *TrackMe* system will be done module by module and component by component. The order in which it is carried out depends on a number of factors like the complexity of the modules and services, the dependence of other modules on the component being implemented and to the system as a whole, and it should also take into account the possibility of discovering flaws with the proposed design. The later should be dealt in a way that, if such an unfortunate event does happen, the flaws should be found and corrected as soon as possible, to limit the cost of the change of design. Identify here the order in which system plan to implement the subcomponents of your system and the order in which system plan to integrate such subcomponents and test the integration.

### 5.2.1 Implementation order

Due to modularity of system components and to the adopted strategy there are no constraints on implementation order of components. But due to some critical aspects of a few components we want to prioritize implementation of those components.

The most critical components of the system that could take more than others to be implemented may be:

1. DBManager, AuthenticationManager, LoginService and SignupService
2. RequestService, SubscriptionService, NotificationService
3. SearchService, DataService
4. APIManager, UserService
5. AutomatedSOSDB, TokenDB, Track4RunDB, Data4HelpDB

(note that by specifying the names of interfaces of components, we are also considering the concrete implementations, in whichever number they exist)

## 5.3 Integration and Testing

### 5.3.1 Entry criteria

The integration of components and its testing should start as soon as possible, but before they can commence, some conditions must be met. First of all, the external services and their APIs that are going to be used in the application should be available and ready. This applies to the already mentioned services, to the DBMS and the server on which it will be running on.

Next, the modules which are being integrated should have at least the operations concerning one another created, if not completed completely. The operations that have been developed should pass the unit tests in order to be sure that the components are working fine on their own and that if an integration test fails, the problem lies in the in the integration itself.

### 5.3.2 Integration test strategy

The main goal of integration process is to avoid as much errors as possible at each step of the process, so the system will incrementally integrate components as soon as they are completely developed and released.

*Bottom-up* design will be adopted for most of the integration process: at the beginning only components that have less bindings to other components or which can work without other component will be integrated. In this way we can obtain feedbacks about system functionalities as soon as components are released and in addition we can parallelize integration of different subsections of the system.

For the most critical components or for more complex system parts we will use instead *Critical Modules strategy*: components that fit very well for *Critical Modules* strategy are those in Data4Help subsystem, because the AutomatedSOS and Track4Run will use the Data4Help system and are the most frequent interaction performed into our system. For this reason *Bottom-Up* strategy will be applied only once Data4Help subsystem will be fully integrated using *Critical Modules*.

### 5.3.3 Integration order

In this section, the list and order of every integration that is performed is shown. As already stated, the integration will be performed from the bottom-up.

It should be noted that there will be no explicit integration of the Data4Help Backend, Track4Run Backend with any of the other components. This is because the nature of the component, the extent of the usage and dependency of other components on it and the implementation plan, that clearly states that the Data4Help Backend will be the first part that is implemented, mean that the integration itself is already being done during the implementation phase of the depending components and its correctness will inherently be tested by the unit tests of each component.

1. As written in the introduction, at the beginning we will integrate the *Data4Help* subsystem, composed by these components:

   - Data4Help Web Site
   - DBManager, AuthenticationManager, LoginService and SignupService
   - RequestService, SubscriptionService
   - SearchManager, UserService, DataService
   - Schedulers

   Integration tests will check cases like addition of new user to database, accessing the system with correct credentials, the consistency of making request, the reachability of all notifications to accept/reject request in the dashboard, the modification of subscription chosen by the user (if he wants any).

2. Then we could start to implement part of the *AutomatedSOS* subsystem:

   - DataService, DBManager
   - APIManager

   This integration step will check the communication of the Data4Help system with external sources HealthCareService when the vital signs are out of the normal range.

3. Then we could start to implement part of the *Track4Run* subsystem:

   - Track4RunWebService
   - Track4RunDB
   - RunCollection

- TokenService, OrganizerCollection and ParticipantCollection

This integration step will check the communication of the Data4Help system with Track4RunWebService and when RunCollection is modified, check the integration with the existing requestService and notificationService of Data4HelpWebService.

4. After that, other components can be joined together into small subsystems to test their interactions:

   - *AuthenticationManager* and *LoginService* : their interactions must not brake the consistency of Account information and they have to check only authorized users can access information
   - *SearchManager, RequestService* and *SubscriptionService* : system will check consistency between search into a request and subscription of the same user
   - *DataService, DBManager* and *APIManager* : system will continuously check latest data and verify with the thresholds, if varies, notifies HealthCareService through APIManager of the same user
   - *RunCollection* and *ParticipantCollection* : system will check consistency between run created and allows users participated in the same run

5. After that we can start to integrate user interactions:

   - *UserInterface* and *RequestService* : tests sending request for specific or anonymized search through different user interfaces
   - *UserInterface* and *NotificationService* : tests will check asynchronous communication between user and server for the dispatch of event messages
   - *UserInterface* and *RunCollection* : tests will check asynchronous communication between user and runs available for the live view on the map of participants
   - *UserInterface* and *HealthCareService* External Resource : will tests the communication with external services of User Interface and Health-Care Service and the synchronization of Data with threshold values when user's data is out of range, notifies using external alarm interfaces

Chapter 6

# Effort spent

| Team Work | |
|---|---|
| **Task** | **Hours** |
| Planning Architecture | 8 |
| Architectural design overview | 4 |
| Choosing Patterns | 3 |
| Checking document | 2 |
| **Total** | 17 |

Table 6.1: Time spent by all team members

| Individual Work | | | | | |
|---|---|---|---|---|---|
| **Diego Avila** | | **Laura Schiatti** | | **Sukhpreet Kaur** | |
| **Task** | **Hours** | **Task** | **Hours** | **Task** | **Hours** |
| Component view | 11 | Scope | 3 | Purpose | 2 |
| Component interfaces | 14 | Database view | 6 | UI design | 15 |
| Deployment view | 6 | Runtime view | 8 | I and T plan | 6 |
| Runtime view | 4 | R traceability | 3 | Design decisions | 2 |
| Corrections | 4 | Final user interfaces | 7 | | |
| | | Architectural styles | 4 | | |
| **Total** | 39 | **Total** | 31 | **Total** | 25 |

Table 6.2: Time spent by each team member

Chapter 7

# References


- Requirement Analysis and Specification Document: AA 2017-2018.pdf". Version 1.0 - 26.10.2017
- Henriksen, A., Haugen Mikalsen, M., Woldaregay, A. Z., Muzny, M., Hartvigsen, G., Hopstock, L. A., Grimsgaard, S. (2018)
  Using Fitness Trackers and Smartwatches to Measure Physical Activity in Research: Analysis of Consumer Wrist-Worn Wearables. Journal of medical Internet research, 20(3), e110. doi:10.2196/jmir.9157.
  Retrieved from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5887043/
- IEEE. (1993). IEEE Recommended Practice for Software Requirements Specifications (IEEE 830-1993).
  Retrieved from https://standards.ieee.org/standard/830-1993.html
- Sloane, A. M. (2009). Software Abstractions: Logic, Language, and Analysis by Jackson Daniel, The MIT Press, 2006, 366pp, ISBN 978-0262101141.
- Spark. A Micro Framework For Creating Web Applications - http://sparkjava.com/
- Morphia - http://morphiaorg.github.io/morphia/
- Lettuce - https://lettuce.io/
- Angular - https://angular.io/
- Google Maps Platform - https://cloud.google.com/maps-platform/