TrackMe: DD
Software Engineer 2 - 2018/2019

Riccardo Poiani, Mattia Tibaldi, Tang-Tang Zhou
Politecnico di Milano

Version 1.0

December 10, 2018

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide a description of the system architecture of the TrackMe project. Furthermore, this report, with the joint information of an implementation and test plan, will be helpful to guide a software team during the development of the products. The DD shows the main system components with their behaviors, interfaces and design patterns, in order to facilitate the programming phase. Practically, the description is required to coordinate a team under a single vision, and, thus, it outlines all parts of the software and how they will work.

1.2 Scope

TrackMe is composed of three main features: Data4Help, AutomatedSOS and Track4Run. Data4Help is thought as a service that allows to monitor the position and the health status of individuals. It provides different types of diagnostic procedures: heartbeat, blood pressure and blood oxygen saturation levels. Moreover, Data4Help permits the users to see their health statuses in order to be conscious of their condition. Third party customers can demand for specific data of a particular user, by sending him a request message, or they may ask aggregated data on a group of user, whose number of members is greater then 1000. User may accept or decline individual requests; in this last case they are also enabled to block a third party customer (i.e. he will be no more able to send request to that user). TrackMe, also, implements AutomatedSOS service: when the health parameters of a user go below the standards, the system contacts an emergency room and an ambulance is provided. Finally, with Track4Run the organizers can set up a race and all the interested people (i.e. runners) may sign up to the race. During the event, the application will automatically monitor the runners, and spectators will be able to follow the race through the system: at the end of the race, the organizers have to close the run and the spectators will be able to see a leaderboard.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- user: a person who has registered on the system (it is equivalent to subscribed user).
- third party: a person or company that wants to access data
- health parameter: a value regarding healthcare status of a specific individual that can be heartbeat, blood pressure or blood oxygen saturation levels
- threshold: a critical value regarding health parameters, which is unique for every user
- two-phase commit: a mechanism for implementing a transaction across different software components which guarantees ACID properties. It is composed by two phases: prepare-phase and commit/rollback-phase

- transaction: A transaction is an elementary, atomic unit of word performed by an application
- firewall: a network security system that monitors and controls incoming and outgoing packets based on specific rules.
- gateway: a network hardware used in telecommunications that allows data to flow from one network to another.
- service: a set of features or applications offered to the client

1.3.2 Acronyms

- RASD - Requirement Analysis and Specification Document
- GPS - Global Positioning System
- DD - Design Document
- REST - Representational State Transfer
- 2PC - Two-Phase Commit
- ACID - Atomicity Consistency Isolation Durability
- BASE - Basic Availability Soft-state Eventually consistent
- HTTPS - Hypertext Transfer Protocol Secure
- API - Application Program Interface
- NFC - Near Field Communication
- BT - Bluetooth
- DBMS - DataBase Management System
- DB - DataBase
- SIM - Subscriber Identification Module
- UTC - Universal Time Coordinated
- AES - Advanced Encryption Standard
- JSON - JavaScript Object Notation
- XML - Xperimental Markup Language
- SQL - Structured Query Language
- DMZ - Demilitarized zone
- LAN - Local Area Network
- ER - Entity Relationship

1.3.3 Abbreviations

Gi - Goal number i, where i is a number

Ri - Requirement number i, where i is a number

1.4 Revision history

- v1.0: The document regards analysis of design and architecture of the system before starting the implementation.

1.5 Reference documents

The DD refers to different documents:

- Slides about DD of Software Engineering 2
- old DD documents such as: "the DD to be analyzed", DD of 2015/16 and 2016/17.
- old Integration and test plan document of 2016/17

It, also, refers to many websites:

- `https://microservices.io`
- `http://eventuate.io`
- `https://www.rabbitmq.com`
- `https://www.docker.com/`
- `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`
- `https://www.baeldung.com/transactions-across-microservices`
- `https://logz.io/blog/logging-microservices/`
- `https://dzone.com/articles/microservice-design-patterns`
- `https://dzone.com/articles/communicating-between-microservices`
- Transaction Management in Microservices
- Microservices communications why you should switch to message queues
- Microservices when not to use them

The first two websites are very useful to learn the architecture of microservices but not enough. Moreover, there are also interesting videos that explain very well microservices architecture and their pattern:

- `https://www.youtube.com/watch?v=txlSrGVCK18`

1.6 Document structure

The document is structured as follow:

- The first section introduces the design document. It explains the scope of the project, text conventions and the structure of the document
- The second section illustrates the main components of the system and the relationships between them. This section will also focus on the main architectural styles and patterns adopted designing the system
- The third section presents the user interface design and further details about its implementations
- The fourth section associates the goals presented in the RASD with the choices taken, and it shows as the decision can allow to reach the goal
- The fifth section illustrates the implementation and integration test plan. This plan should be followed during the development of the product

2 Architecture Design

2.1 Overview

In this whole chapter the architecture design process is described, and it contains all the significant architecture decisions. Therefore, first of all, in this section a high-level and very general architecture for the TrackMe project is presented and discussed.

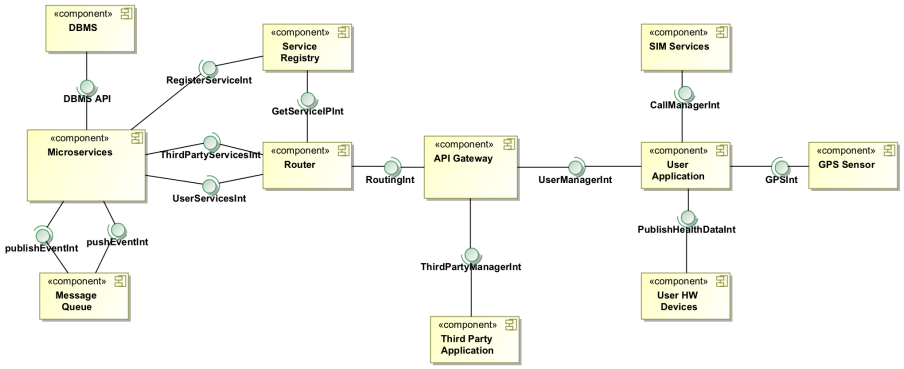

Figure 1: High-level component diagram

As a reference architecture, the microservices one has been considered: this decision will be later discussed.
From the picture, it is evident that the clients can access the services by means of an API gateway, that will provide authentication to the users, and forwards the requests to a router. The router will query the service registry, in order to know where instances of the demanded service are located: once that it retrieves this information, it will be able to send and load balance requests. Of course, a new service instance needs to register to the service registry.
Another relevant information is that the project can be ideally split, at a high level, into two applications: one for the users and one for the third parties customers: they both interact with the API gateway, that will guarantee to access all the microservices that compose the core functionality of the business. More specifically, the "microservices" block includes Data4Help and Track4Run functions, while the goals of AutomatedSOS service are reached locally and directly by means of the user application, that, when needed, will perform a call with the help of the SIM service. The user application also needs to communicate with a device that is able to collect data regarding his health: more specifically, this instrument will send data toward the user application autonomously, without being asked. Examples of this machines are smartrings and smartwatches. The data regarding the position of the user is, instead, retrieved via GPS's API.
Note, also, that the microservices need to keep some data persistent, and, thus access to a DBMS, by means of its API. Moreover, some services need also to communicate and share information: this is achieved via the message queue, that is charged of receiving and forwarding messages to the interested parts.

2.2 Component view

In this section, components are expanded and better analyzed.
The main analysis regards the microservices block:

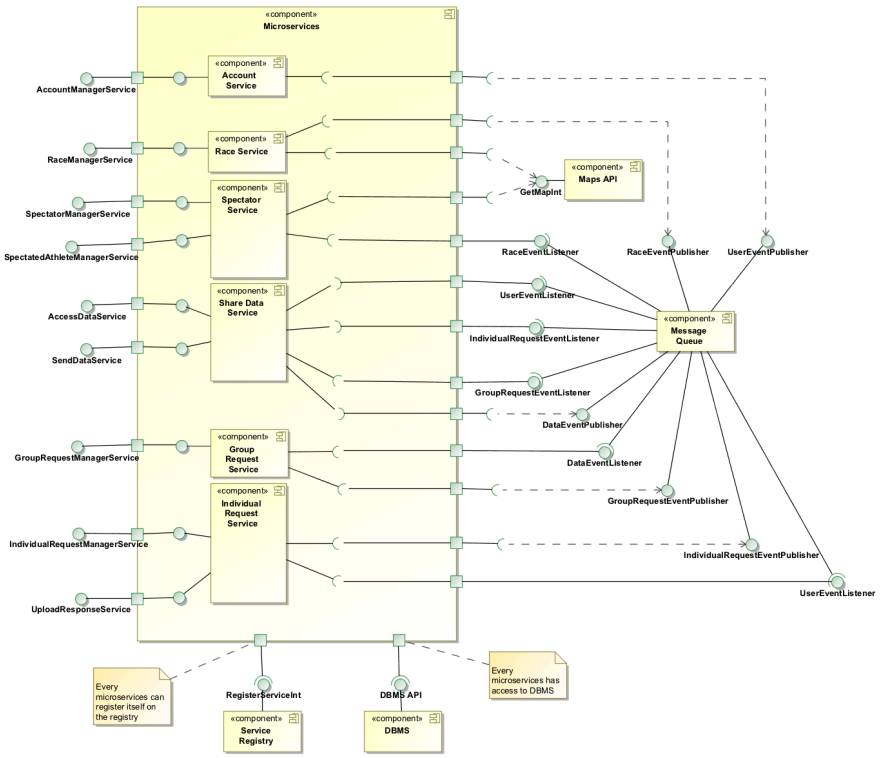

Figure 2: Component diagram

Various service components are present, and they all expose at least an interface that is accessed by the clients via the joint combination of API gateway and router. It follows a brief description of the various services, and a better specification of the interfaces they provide to users and third party customers:

- The account service provides all the functions related to user sessions, such as login and logout, and also the registration of clients
- The race service makes possible for an athlete to enroll in a run; for a third party customer to set up one, and also close it; and it also enables the possibility of retrieving the available races and their status (e.g. in course, terminated). Therefore, it manages the "more static" part of a race
- The spectator service regards the dynamic part of a race, such as the fact that a user can spectate athletes who are running (therefore they can retrieve their positions and the leaderboard). This service is also in charge of receiving position data of the participants

- Share data service is responsible for the monitoring the user health statuses and positions: here a user will upload his data. Furthermore, third party customer will access the data that they have previously requested, by means of "AccessDataService" interface

- The group request service provides to the third parties the possibility of uploading group requests and see their status

- The individual request service enables third party customer to upload individual request and monitor their status, and allows a user to check if there are some requests for him, and, eventually, upload responses

Microservices have their own data and, therefore, access DBMS's API, while Maps's API are necessary only for race and spectator service, in order to manage and double check position data and feasible paths.
The diagram also shows the main communications that take place between components. The following message exchange is needed to completely and correctly exploit the project functions.

- When a third party is accessing individual data through the data share service, it needs to know if a request associated with those data was sent and accepted. Indeed, the request acceptance and its data, are managed in the following way: when an individual request has been accepted, a message is sent through the message queue with the help of an interface, IndividualRequestEventPublisher. Then, this message reaches the share data service thanks to the interface provided by it (i.e. IndividualRequestEventListener).
  Moreover, to guarantee that individual requests must be performed on registered users, a match with the account service is necessary. When a new user registers, the account service will send the data about the user through the message queue to reach all the services which need it. One of these services is the Individual Request Service which needs the user to check if an individual request is coherent with the system (UserEventListener and UserEventPublisher are the interfaces that accomplish this task).

- A slightly different reasoning holds for the management of the group requests. In this case, the exchange is bidirectional. In particular, when a new group request is performed, the share data service will be notified by the message queue via the GroupRequestEventListener, and it will be sent to the message queue by GroupRequestEventPublisher.
  When the share data service receives this event, it will query its data in order to retrieve information regarding the number of users involved: of course, this information needs to be sent back to the group request service, that will accept or refuse the request (DataEventListener and DataEventPublisher are the interfaces that accomplish this task). Of course, the acceptance of the request may trigger again the new group request event. However, this will be later exposed and commented in the run time view.
  Moreover, to guarantee that the filters on the aggregated request are applied correctly (e.g. data on the users whose age is greater then 40) a match with the account service is needed. Indeed, when a user registers

to the service, the account service will sends user information to the message queue with UserEventPublisher. The information will be forwarded to the share data service with UserEventListener.

- Another important message exchange is between spectator and race services: in particular, data that the spectator service is receiving must be matched with an active run, and thus it needs to know the status of the run: the changes will be communicated by RaceEventListener and RaceEventPublisher.

2.3 Deployment view

Regarding the deployment, there are many quality to take into consideration. One of these is the scalability property: at the beginning TrackMe might only have few users, but once the usage is increasing more and more, the system must continue to work even if the requests from the users are huge. To achieve this quality of service, TrackMe needs the help of cloud database providers. These ones are specialized in the management of big database; and in this case the Share Data Service requires a huge database in order to store all the data collected from the users. Furthermore, since the services with the highest throughput are considered to be the spectator and the data share one, these services will be deployed in a cluster: Docker can be used in order to provide images of the containerized environment. This, also, enhances scalability.

On the other side, services such as Account service, Individual Request Service and Group Request Service do not require so much power. Therefore, these services, at least in a first moment, are not thought to need a cluster: indeed, they will be deployed on the same physical machine, but with docker images in different virtual machines. In such a way each microservices has its own application server, while the database server can be shared among these services. However, with this approach, it will be easy to scale, in case it is necessary.
Note that the Spectator Service requires to save a good amount of data and the access might be frequent due to the requirement regarding the possibility to see all the positions of the runners during a race. Consequently, the Spectator Service has its own database server machine. A further thing that is worth a comment is the fact that the router could be a cluster too, in order to avoid that it becomes the bottleneck of the architecture.

To access these microservices one needs to use the mobile application which has to be installed on an Android device.

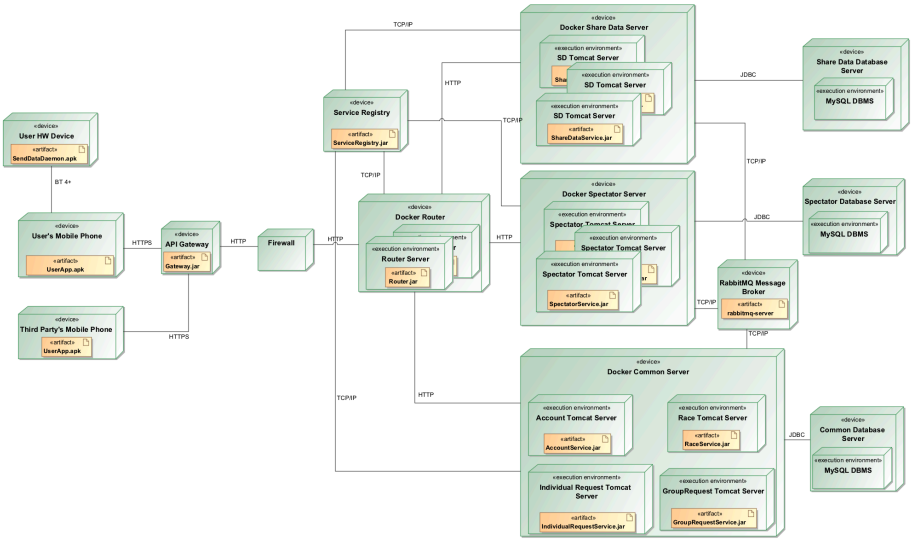

Figure 3: Deployment diagram

The deployment diagram represents the system to-be which is based on the microservices "reference architecture". Even if microservices offer the possibility to implement different services with different programming language or framework, what is simpler is to keep using the same environment. In this case, Tomcat application server and MySQL DBMS are strongly used. Furthermore the diagram shows the presence of a firewall that adds an additional layer of security to the LAN and also implement a DMZ in order to allow an external network node to access only what is exposed in the DMZ.

2.4 Runtime view

In this section, to clarify better how the system works dynamically at runtime, a series of sequence diagram is shown:

- Log-in diagram
- Request-acceptance diagram
- Publish group request diagram
- Automated SOS call diagram
- Close race diagram
- Send cluster of data diagram

2.4.1 Log-in sequence diagram

The log in is an important feature of large interest, because it is also central for the process of handling the users' session. The next diagram exposes the procedure for the user.

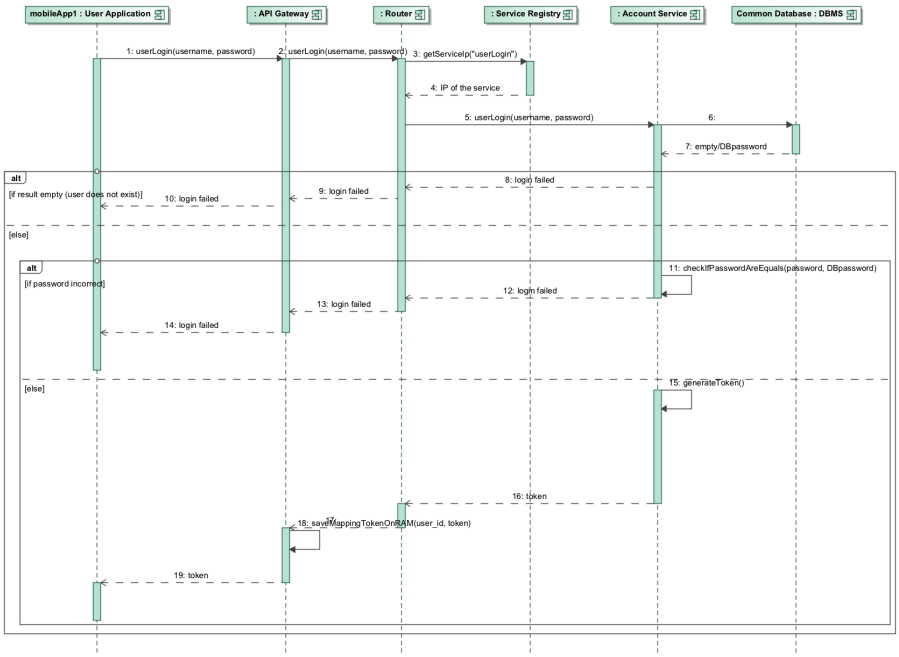

Figure 4: Sequence diagram: log in

Only in this diagram, it is shown how the router and service registry works when they are called from the API gateway due to achieve simplicity on other diagrams which are more complex. As the diagram show, if the log-in is done correctly, a token is generated from the account service and returned back to the API gateway and to the client. The information will be used by both the client and the API gateway to manage the user session: all the next requests will be authenticated using that token.

2.4.2 Request-acceptance sequence diagram

An important issue in the project is the process in which the user gives a reply to an individual request performed by a third party customer. This is exploited in the following diagram, that assumes that the user is already logged in and that the list of pending request is non-empty.

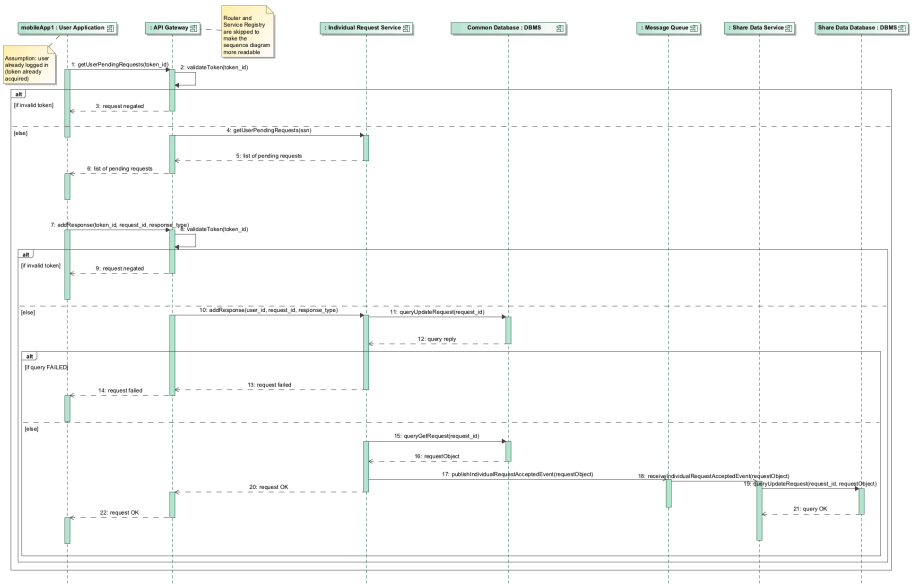

Figure 5: Sequence diagram: request acceptance

Some comments follows in order to clarify some essential points:

- The API gateway functions also as a proxy for the authentication service. In particular, it keeps track of the active sessions of the users (identified by a token_id). This information is in memory since it was returned from the account service in the moment in which that user has performed the log-in
- The communication between services is kept asynchronous, in order to reduce the response time toward the user that initially used the individual request service. This is acceptable because there are no strict requirement on the efficiency on the access of data by third party customer (i.e. the data should not be available in the exact moment in which an user sends a response: some latency is tolerated).
- The share data service needs information about requests that have been accepted, in order to correctly provide the data access in the right way (i.e. granting access if an accepted request on the data has been accepted). Therefore, when a request is assigned to a new status in the individual request service, this information is communicated
- the Message Queue component is responsible to forward the event of new request status to all the other services which has subscribed to this event. This helps to achieve a messaging system able to send asynchronous messages across services.

2.4.3 Publish group request sequence diagram

Another important feature is the fact that third party customers can perform requests on aggregated data.

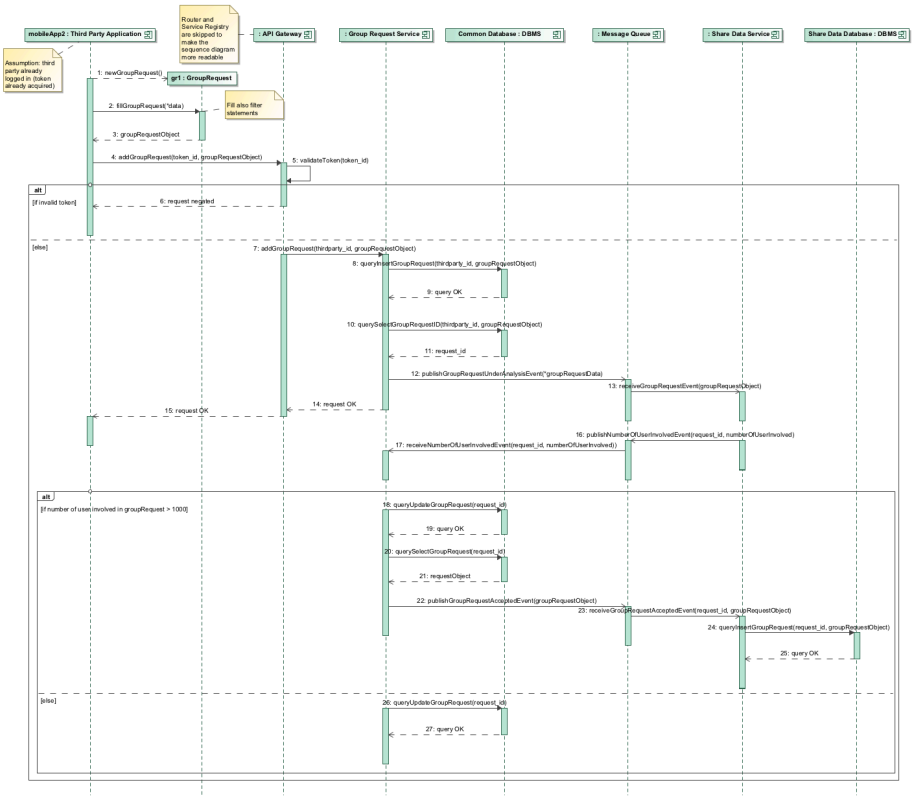

Figure 6: Sequence diagram: group request

The same comments of the previous diagram hold, also, in this situation. However, it is relevant to clarify the asynchronous exchange of messages that happens between the group request service and the share data service. In the first place, it is necessary to point out that when a third party customer sends a group request, it is reasonable to assume that some time to serve and verify the request can be taken, but a non-desired behavior is to block the client in waiting the response. The status of the request can be checked in a second moment, always via the group request service. Therefore, when a petitioner asks for some aggregated data, the group request service will delegate to the share data service the task of analyzing the request: it will then reply with the number of user involved. If this number is greater than 1000, the group request service changes the status of the request from "under analysis" to "accepted" and it will send this information to the share data service, that, thus, will be able to provide correct access to the requested data.

2.4.4 Automated SOS call sequence diagram

Another critical feature of the project is the one regarding AutomatedSOS:

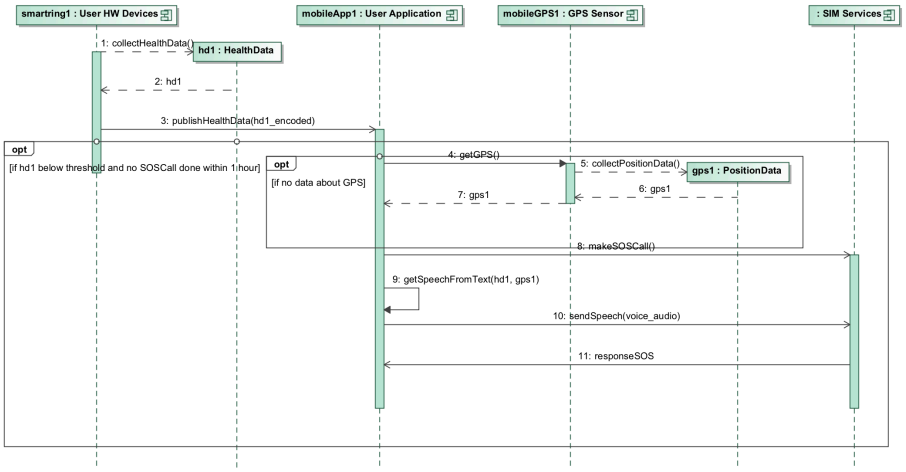

Figure 7: Sequence diagram: automated SOS call

This feature is implemented locally, in the mobile application, in order to guarantee the best efficiency and reduce to the minimum delays that may be introduced due to data transfer over the Internet. Indeed, the diagram shows that the user hardware device component will collect user health data, and send them to the user application: if this observes some parameters below the threshold, it will perform an SOS call. The call is handled in the following way: the mobile app uses a default template to communicate with an emergency room operator all the information needed to provide assistance (i.e. health data and position). The emergency room operator replies with a response that should be "Yes" or "No", in order that the application is aware of it.

2.4.5 Close race sequence diagram

The procedure of closing a race is here described:

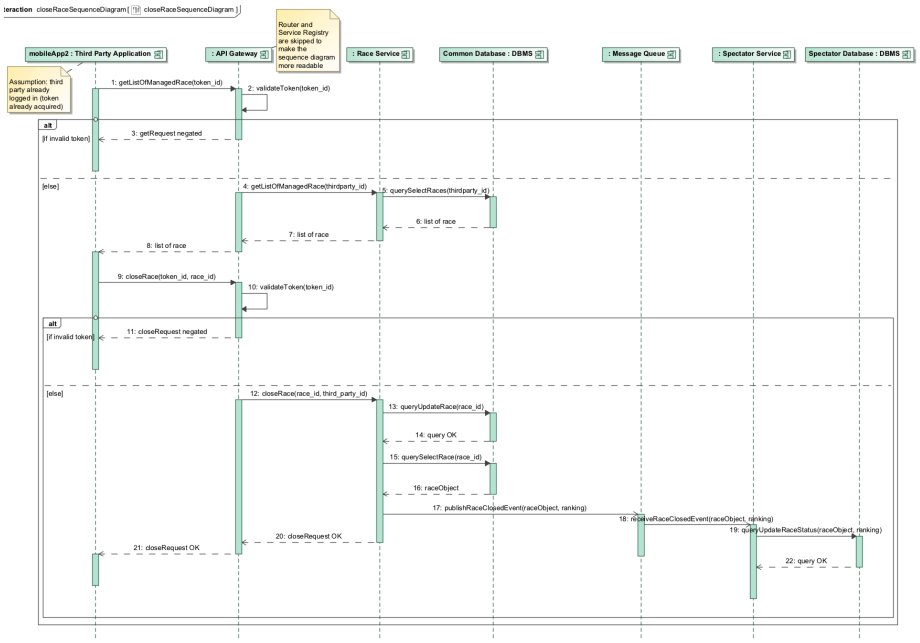

Figure 8: Sequence diagram: close race

It is worth to comment the asynchronous message from race service to spectator service. Indeed, in this case the race is successfully closed, therefore, data collection from athletes from that event should be stopped and not more available; and the same holds for the fact that users are no more able to see their positions, but only the leaderboard. In order to accomplish this, an asynchronous call is here performed, to inform the spectator service. Note that some delay could happen, due to asynchronicity, in preventing the above mentioned features; however this is tolerated, because it is not something considered that critical.

2.4.6 Send cluster of data sequence diagram

The interaction between components during the send of cluster of data done by a user is described as follows:

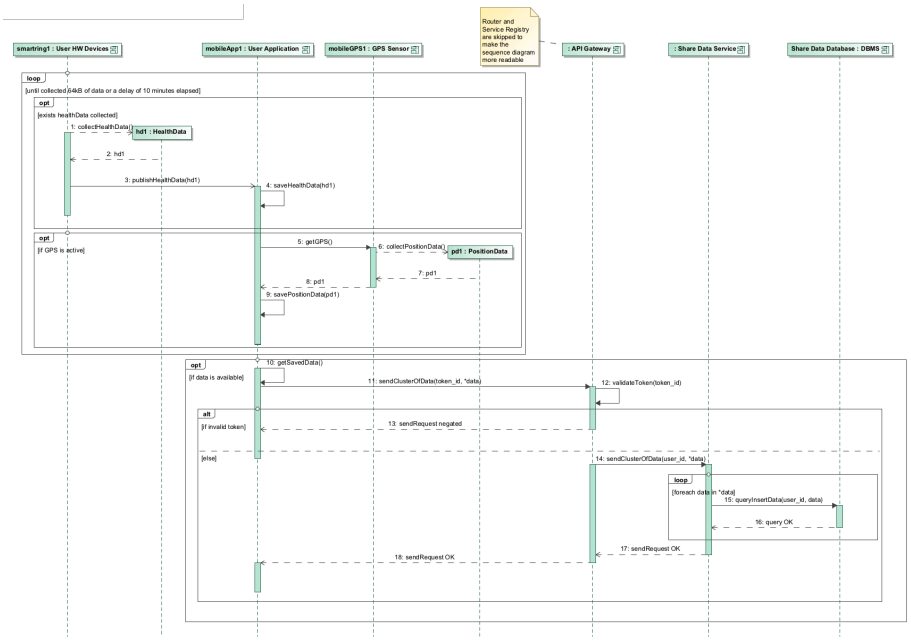

Figure 9: Sequence diagram: send cluster of data

The most of the request done by users is related to sending data to the TrackMe system. Therefore, this is one of the bottleneck of the system. By taking into consideration that more or less every application server accepts about 8kB of header HTTP request and the data to send at each time:

- token_id: it is a token of 16 bytes, which is necessary to identify the user;
- position data: it is, basically, two double representing latitude and longitude, but since the RESTful message cannot send double, it can be seen as two strings of 10 bytes;

Degree precision versus length

decimal places	decimal degrees	DMS	qualitative scale that can be identified	N/S or E/W at equator	E/W at 23N/S	E/W at 45N/S	E/W at 67N/S
0	1.0	1° 00′ 0″	country or large region	111.32 km	102.47 km	78.71 km	43.496 km
1	0.1	0° 06′ 0″	large city or district	11.132 km	10.247 km	7.871 km	4.3496 km
2	0.01	0° 00′ 36″	town or village	1.1132 km	1.0247 km	787.1 m	434.96 m
3	0.001	0° 00′ 3.6″	neighborhood, street	111.32 m	102.47 m	78.71 m	43.496 m
4	0.0001	0° 00′ 0.36″	individual street, land parcel	11.132 m	10.247 m	7.871 m	4.3496 m
5	0.00001	0° 00′ 0.036″	individual trees, door entrance	1.1132 m	1.0247 m	787.1 mm	434.96 mm
6	0.000001	0° 00′ 0.0036″	individual humans	111.32 mm	102.47 mm	78.71 mm	43.496 mm
7	0.0000001	0° 00′ 0.00036″	practical limit of commercial surveying	11.132 mm	10.247 mm	7.871 mm	4.3496 mm
8	0.00000001	0° 00′ 0.000036″	specialized surveying (e.g. tectonic plate mapping)	1.1132 mm	1.0247 mm	787.1 μm	434.96 μm

Figure 10: Degree precision versus length

- health data: this type of data can be a heartbeat value (i.e. integer), a blood pressure value (i.e. minimum pressure or maximum pressure, which

means two integers), or a bloody oxygen level (i.e. integer). Therefore, at most it is possible to have 2 integers when a health data is sent.

- timestamp: it usually weights 8 bytes when the timestamp has a timezone. To be coherent with everyone, the timestamp will always have a timezone of Greewich UTC.

Other than data, if the messages are, for instance, JSON they need a key which is a string. More or less a key can be estimated to be on average 10 bytes. Therefore, after this consideration, a possible recommendation about how periodically the user should send data is to do it once the data collected exceed 64 kB or when 10 minutes are elapsed. This choice is due to the fact that too much requests within a short time could overload the servers, but the system should also have updated data, so, if 10 minutes are elapsed from the last transfer and there are data available, then the user application should send data to TrackMe.

2.5 Component interfaces

Before showing the component interfaces, it is essential to show the class diagram of the model to be developed in order to explain better what is next.

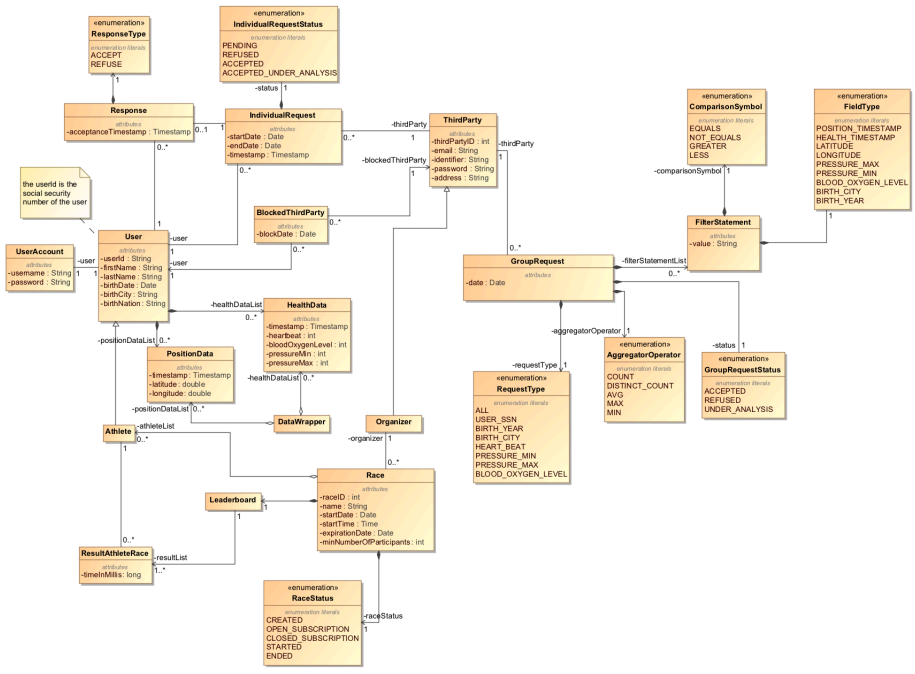

Figure 11: Class diagram of the model

Here follows the interfaces of the various components that are available. Note that interfaces of external components (e.g. DBMS's API, CallManagerInt, GPSInt, MapsAPI) are not present.

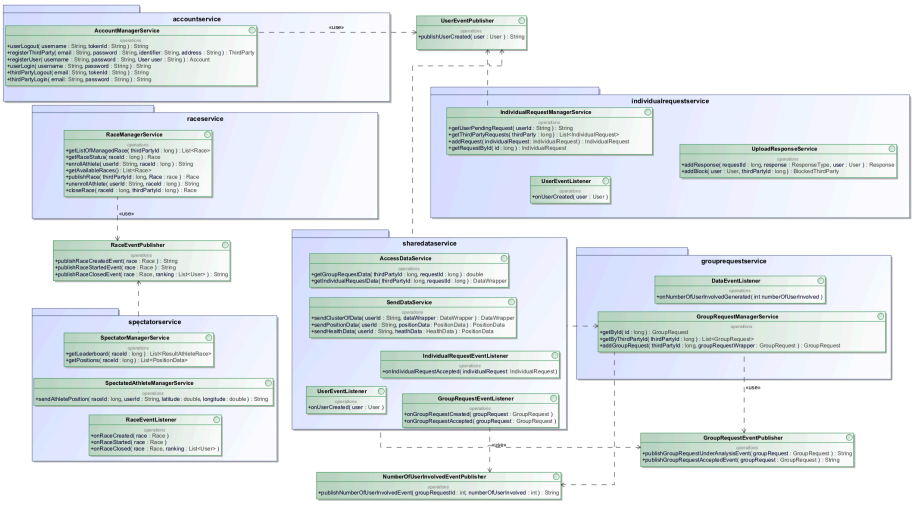

Figure 12: Component interfaces

In order to simplify the diagram, the API gateway has been excluded, but what it exposes is just the following: it provides a link for each interface that is external w.r.t. the microservices block in figure 2. However, there is a little modification in the parameters requested: indeed, the id of the client that is requesting the service (e.g. userID, thirdPartyID) is substituted with tokenID, in order to manage the authentication. This holds, clearly, with the exception of the registration and login feature, since a token is not available at this early stage of the user session.
A same reasoning is applied for what concerns the router: it just forwards the "translated" requests.

2.6 Selected architectural styles and patterns

In this section, all the adopted architectural styles and patterns are explained sufficiently to define why the specific style or pattern is chosen.

2.6.1 Microservices

The selected architectural style, as already mentioned, is the microservices architecture. This choice is supported by multiple reasons. In particular, a strong motivation in the decision process is the conjunction of the following two facts:

1. Data4Help and Track4Run are features that provides totally different and independent functions. Track4Run can, without any problem, even be deployed in a different moment

2. Scalability is one of the main QoS that the system requires. Indeed, the application is expected to be used by many people in the future, and therefore, designing the architecture making scaling fast is really important

It is important to note that it could happen that one of the two functions needs to scale, while the other does not. More specifically, this reasoning also applies to some pieces of functionality that are internal to both Data4Help and

Track4Run. For instance, it is reasonable to assume that the functions of data collection from users (positions and health statuses) will be much more used and will generate much more network traffic, compared to the one that regards the requests. In order to clarify this, all the active users will periodically send data at each moment of the day, while third party customers that are performing requests do not keep forwarding request to the user: it is something that happens more rarely. The same holds, for example, when considering the set up of run events from the organizer, and the fact that spectators will monitor in real time positions of athletes during a race.
Therefore, the combination of these things, suggests that the scaling should be independent w.r.t. the function considered.
Furthermore, the microservices architecture makes it easier to implement failure isolation: it is not desirable that a failure in Track4Run leads in a failure in also Data4Help. Moreover, failing in managing the request should not prevent the data collection: as one can see the reason also applies to the various business capabilities that are bounded in the single Data4Help, or Track4Run, feature. Due to this, the overall availability is improved.

2.6.2 Eventually consistency and compensation

Before going to analyze all the patterns adopted for the microservices architecture, one must define the most important thing in a microservices architecture: how to achieve consistency of data. For instance, solutions for this issue are:

- Avoiding transactions across microservices: usually if a microservices architecture needs to have distributed transactions it means that there are redundant data. But if this is avoided, the lack of redundant data means that everytime a microservice needs to ask data from another microservice, a request is sent and if the other microservice is under high load, it will not be able to process it fast.

- Two-Phase commit protocol (2PC): The distributed transaction of 2PC consists on two steps: Prepare phase (lock-phase) and Commit-rollback phase (unlock-phase). The problem with this protocol is that it is too slow compared to the time for an operation of a single microservice.

- Eventual consistency and compensation: it is a model different from ACID transactions, but it is a mechanism to achieve consistency eventually at some point in the future. This might not achieve the same thing as 2PC but it is faster and if the architecture does not need the strict properties of ACID, then it is better. Otherwise, it could be very difficult to implement it.

Out of the three solutions, 2PC is not an option since the messages are synchronous, which means too slow. Therefore, Eventual consistency and compensation is the one adopted by this project due to the fact that the system will scale a lot and there is no problem if consistency is achieved in the future.

2.6.3 Design patterns

Now, some patterns adopted related to the microservices architecture are exposed:

- API gateway: this is a component already introduced in the high level component diagram and it satisfies and deals with the following problem: how do clients of the application access to the individual services? The API gateway is a single entry point for all clients and it can expose different API for each client: this suits well for the TrackMe project. This should be implemented with an event-driven reactive approach in order to scale if it is necessary to manage big loads of data

- The access token pattern is very useful, since the application is composed of various services and an important issue is how to communicate the identity of the petitioner to the service that handles the request. The pattern suggests to implement the API gateway in such a way that it authenticates requests and passes an access token that securely identifies the client (a service may later include the access token in requests that it makes to other services)

- Database per service pattern: all the services need to persist data in some kind of databases and the solution is to keep each microservice's data private to that service and accessible only via its API. So a database can't be accessed directly by another service. In particular the pattern of schema-per service is always guaranteed, while a database-server-per-service is allocated to the services that are though to be the one with the highest throughput (i.e. spectator service and share data service)

- The messages and the communications between services, that were already introduced in the component diagram, are implement by means of asynchronous messages, since a chain of synchronous calls could lead to slowdown the entire architecture. Note, that this leads to the replication of some data, however, performance is considered more important than cost in this case. A way to achieve a better communications between services is to use something similar to the API gateway: a message queue architecture which requires also an additional service called message broker that is responsible for gathering, routing and distributing messages from services to other services (something similar to a mail service)

- As shown, services need to call one another, but a microservices-based application typically runs in a virtualized environment where the number of instances of a service and their locations changes dynamically. The solution is to use a router that runs at a well known location. The router will query a service registry, which may be built into the router, and forwards the requests to an available service instance. Of course, when a new instance is created, it has to perform registration to the service registry.
  The API gateway will send translated requests to the router, that will load balance them onto the active instances. This pattern gives great benefits: indeed, the clients do not deal with the discovery of the service, but it comes with the drawback of an additional component that need to be installed and configured, and, if needed, replicated, in order to scale

- Microservices expose REST API that will be provided only by HTTPS endpoints. This will ensure all the benefits of REST API, such the fact

that services are stateless, with the security of an high-level encryption protocol. Other benefits of the REST API which are relevant to the requirements of the project are: separation of concerns between client and server, reliability and scalability.

- Saga pattern: this design pattern is, practically, a way to achieve the solution of Eventual consistency and compensation described before. It solves the trouble of distributed transactions without using 2PC by using a sequence of local transactions where each transaction updates data within a single service. Even though 2PC guarantees ACID properties, it is better Saga pattern that achieves only different properties, called BASE: Basic Availability Soft-state Eventually consistent. But with Saga, there are two main different ways to implement a saga transaction:
  - Events/Choreography: each service subscribes and publishes to other service's events
  - Command/Orchestration: there is a central coordinator service which is responsible about decision making and sequencing business logic

  In the project it is implemented the version of event transaction.

2.7 Other design decisions

2.7.1 Frameworks

The chosen and described architectural style, leads to some choices in the framework that will be used for the development and the deployment. In particular, to implement a microservices architecture, a microservices chassis framework is needed, in order to have the best support in the developing phase. An option here is to adopt both Spring Boot (for what concerns the microservices) and Spring Cloud (that facilitates the set up of distributed system software). They are widely adopted and greatly supported by the community; also many guides and articles are available on the internet.

2.7.2 Asynchronous messages

RabbitMQ, that is a message broker server, is used to implement the exchange of messages among services. This choice is motivated by the following facts: it enhances delivery and order guarantee, redundancy, decoupling and scalability. Finally, it is well supported in Spring boot.

2.7.3 Data transfer encryption

Another important issue is regarding the communication between the user hardware device and the user application. This is encrypted with AES-128.
The communication happens by means of bluetooth: the version 4.0 + LE, that is available from 2010, provides some nice features for the project, because it grants enough bandwidth (1Mbps when the low energy mode is enabled) for transmitting the data, and it is also a low energy consumer. Furthermore, there is no need of internet connection (with the limits of the network coverage): the devices just need to be close, but this is a totally rational assumption, given the

context of the application. Note that this version ensures a good compatibility with a large set of devices.

2.7.4 Android application

The user application will be developed for the Android operative system, at least in a first moment. This is due to the criticality of the automatedSOS service. Assuming that the application will be deployed with Italy as the first market target, this choice covers the biggest part of the market.

2.7.5 JSON

REST API will be implemented using JSON as a protocol for exchanging data. Compared to XML, it is less verbose and JSON packets require less size. Furthermore, it is well supported by Spring Boot.

2.7.6 Group Request Filtering

Up until now, group requests were more or less abstract: there was no definition of what group requests could possibly get from data. Therefore, here it is defined what group requests return. The only outcomes they give are aggregated data, i.e. count of tuples, sum of values, average of values or other aggregate operator available (e.g. in SQL). Another thing to clarify is that the tuples of the result from a group request have to satisfy a certain constraint requested by the petitioner. Consequently, the following filters are the ones available:

- Filtering by birth city
- Filtering by birth nation
- Filtering by birth year
- Filtering by the GPS position data
- Filtering by the health statuses
- Filtering by date and time

Since this type of service is very difficult to implement for the user (user-friendly difficult), each filter is applied to decrease the number of users and every query is applied only to users that have at least one position data and one health data. Basically there is no group-by statement in the query.

3 User interface design

Since the application is thought as a set of microservices, the user interface may be designed according to this style. For this reason, composition UI generated by microservices, will be adopted. With a composition approach, frontend application is divided into modules, one for each service, and each module is developed independently. Then, the UI is assembled using some low level server-side, like an API Gateway. The mock-ups for this application were presented in the RASD Document in section 3.1.1.

4 Requirements traceability

4.1 Functional requirements

In this section, the requirements and goals specified into the *Requirements Analysis and Specification Document* are discussed. The following list provides the design components that allows to satisfy the requirements and goals:

[G1] Allow a user to access its own data. Requirements([R10])

1. Account service.
2. Share data service.

As already mentioned in the *Component view* chapter, the account service provides all the user session functions, like the log in into the application, fundamental for accessing user's data. Furthermore, the account service allows a user to perform all the other possible actions, since the user must be subscribed and logged into the application: for this reason, this is always necessary for reaching all goals, but for more legibility it is described only in this first point. Share data service, instead, is responsible for monitoring the user's information, and allows to download the user data from the server.

[G2] Allow a user to contribute to data sharing by providing information about his location and health status. Requirements([R11])

1. Account service.
2. Share data service.

Share data service allows a user to upload its data through SendDataService. Of course, this is possible only for a user who is logged in.

[G3 & G4] Once the health parameters of a user have been observed below the threshold for the first time after one hour, an ambulance is sent to the user location. The time experienced between the moment in which the health parameters of a subscribed user are observed below the threshold and the time in which the emergency point is contacted is equal or less than 5 seconds. Requirements([R12]-[R13]-[R14]-[R14]-[R15]-[R16]-[R17])

1. Account service.
2. GPS sensor.
3. User HW Device.
4. SIM services.

This is one of the most critical feature. The user hardware device provides the heath status of the owner and, when at least one parameter is observed below the threshold, a reading from the GPS sensor is performed, if needed. With this data (i.e. GPS coordinate and health status), the SIM service is able to call the emergency room and provide them all the necessary information. Note that for guaranteeing this functionality the connection between the mobile device and the sensors must be present, and the user must be in an area, where is possible to perform a call.

[G5] Allow a user to participate in a run managed by third parties, as an athlete, if all starting conditions are satisfied. Requirements([R18]-[R19]-[R20]-[R21]-[R22])

1. Account service.
2. Race service.

The service that allows to achieve this goal is the Race service. This service, as already specified, makes possible for an athlete to enroll in a run by showing a list of the available race. Furthermore, it manages the race status and all the possible actions that a user or the third party can perform on it. (i.e. close it, set date, subscribe, etc...)

[G6] Allow spectators (i.e. user) to see on real-time the "correct" positions of all athletes taking part in a run, with at most 15 meters of radius error. Requirements([R23]-[R24])

1. Account service.
2. Spectator service.

As it can be deduced from the name, Spectator service regards the dynamic part of a race and allows spectators to see on real-time the athletes in a run. The service receiving position data from the participants and shares it with the spectators by showing on the map.

[G7] The maximum time to accept an individual request from any third party is 30 days; after that, the request will expire. Requirements([R25])

1. Account service.
2. Individual request service.

All the constrictions on the individual requests are handled by the Individual request service, which receives the requests from the third party customers. This service is charged of calculating the time between the date when the request is posted, and the current date: after that, it decides if the request is expired or not, and, eventually, updates the status.

[G8 & G9] Allow a user to accept or refuse a request from third parties. Allow a user to block requests made by a specific third party and all the pending requests will be refused. Requirements([R26]-[R27]-[R28]-[R29])

1. Account service.
2. Individual request service.

Always the Individual request service manages the accepting or the refusing of the request or the blocking of one by showing a list of the pending requests.

[G10] Allow spectators and runners to see the leaderboard, when a run is completed. Requirements([R30]-[R31]-[R32])

1. Account service.
2. Spectator service.

The Spectator service, furthermore, allows spectator to see the leaderboard on the map during the race.

[G11] Allow organizers (i.e. third parties) to set up a run, by defining its name, its path, date, start time, expiration date, and the minimum number of participants. Requirements([R33]-[R34]-[R45])

1. Account service.
2. Race service.

Race service allows third party customer to set up a race and manage it. The service provides an API for posting a race with all the required information.

[G12] Allow a third party to access data specified in a request if the user accepts the request or if he accepted one or more requests from the same third party that provided access to the same data. Requirements([R35]-[R36]-[R37]-[R38])

1. Account service.
2. Individual request service.
3. Share data service.

In the *Component view* section the Individual request service has already been discussed and it enables third party customer to upload individual request and monitor their status. Furthermore, it enables a user to accept a request.
Share data service is necessary in order to access the data.

[G13] Allow a third party to access statistical and anonymized data if and only if the number of individual involved is greater than 1000. This is satisfied after the request is approved. Requirements([R39]-[R40]-[R41]-[R42])

1. Account service.
2. Group request service.
3. Share data service.

The Group request service collaborates with share data service, which controls the number of users involved. If it is greater than 1000 the request is accepted and the data is provided to the third party customer, with specific filters required in the request message. The access to the data is provided with Share data service.

[G14] Allow a third party to subscribe to non-existing data. They will have access to them, after the data is generated. Requirements([R43]-[R44])

1. Account service.
2. Individual request service.
3. Group request service.
4. Share data service.

Each request service allows to subscribe to non-existing data and then after the data is generated, the status of the request will be changed and the access to the data will be provided.

4.2 Non-functional requirements

Other requirements like performance, reliability, availability, security, maintainability and portability are now discussed in particular.

4.2.1 Performance

The system must be able to manage huge number of requests, as written also in the RASD, and for achieving this performance requirement TrackMe uses a cloud database. Furthermore, since the Spectator service and the Share data service requires a elevated throughput, these services are deployed in acluster. This, also, enhances scalability.

4.2.2 Reliability

The application has to be reliable as possible. To reach this requirement the application services are duplicated on several machines and all test are tested very carefully to reduce errors and bugs as much as possible.

4.2.3 Availability

Note that the data are replicated among the services, that increases reliability and performance, but it introduces consistency problems (this problems and their solution are well discussed before in the *Eventually consistency and compensation* section). For all these things, the reliability of the application is very high.

4.2.4 Security

Also the security and the privacy of data are requirements very important: each transactions between the user hardware device and the user application, and all temporary data store into the mobile device are encrypted with AES-128. Moreover, each exchange of data between the user application and the TrackMe server use HTTPS protocol for a high security communication: indeed, microservices expose REST API, which introduces also other benefits likes separation of concerns between client and server, reliability and scalability. Furthermore, the use of the API Gateway and the access token pattern easiness the management of the security in the system, since it is the only way to interact with the application and the token certifies the action.

4.2.5 Maintainability

The use of external services, framework and the microservices architecture allows to achieve an easy maintainability of the system. Also the use of specific patterns helps to achieve this scope.
The portability is achieved by developing the application in Android. Indeed, after an analysis of the Italy market, has resulted that Android is the most popular mobile system and the frameworks chosen are also well supported by the current market.

5 Implementation, integration and test plan

5.1 Implementation plan

First of all, since client and server can be developed at the same time: indeed, there is low coupling between them, and this is also enhanced by the REST API. To give some structure to the chapter, first the implementation of the server components are discussed, and than the ones regarding the client.

A great property of microservices structure is that microservices can be developed independently. Before starting the implementation of each service, it is required to have defined the protocol message of the network and the ER diagram. Afterword it is possible to set up all the DBMS system and the main logic of the service. Each service can be seen as a completely separate system with its memory stack, its logic and its user interface. At the end, the user interfaces are merged and reachable from a single mobile application. The order in which the services are implemented depends on a number of factors, like the complexity of the service and the dependence of other modules. In this sense, the components of the TrackMe server, could be developed in the following order:

1. Share data service. This service is one of the more important server side component. It takes the GPS data and the health data and it shares them through the two interfaces: access and send data service. This is the first module since it is the main core functionality of the whole project.

2. Individual request service and Group request service. The request services are the main modules for the interaction with the third party customer, and, also the one that are more dependent w.r.t. to the core business of Data4Help. For this reason, it follows the share data service.

3. Race service. The integration with maps API and the several features, that it is composed, makes the programming of this component rather long. Although this fact has already been considered in the design of the system, this still represents a point of possible problems.

4. Spectator service. Spectator service is another service rich of feature to implement and its user interface is not so easy. This component and the previous one need the configuration of an external service (i.e. Google Maps)

5. Account service. This service is likely the most easy to develop, because it is, somehow, a basic feature for most of the applications. However,the security of the access has not to be underestimated, and its importance is crucial in the system.

As one can see from the order, first of all the functions of Data4Help are implemented, since it is considered to be the most relevant feature of the project.
Parallel to the service, it is important to implement the message queue, since this will give the possibility of starting integration test much earlier. After this, for completing the server side, the main parts of communication system must be built, in particularity the components which are interested are:

1. API Gateway: this will be the first one among the remaining server components, since it gives the possibility to set up the connection with the clients

2. Router and API gateway: these are strongly connected, since they are part of the same architectural design pattern, so they will be developed simultaneously

As already mentioned, the client side can be developed at the same time as the server side. A good approach at the beginning of the client side builds the skeleton of the user interface for each service, that helps the programmer to have clear ideas on the aspect of the complete application. This allows understanding if all the requirements are satisfied and if the user experience with the application is satisfactory. Then the development of the user application proceeds with the set up of the communication with the hardware, like GPS sensor and Call service.
On the mobile side, there are several features to implement. It is a good idea start the implementation from the Data4Help, due to its dependency from external services, like GPS sensor and the user HW device that made the service hard to be implemented. A benefit of this choice is that after this implementation the other results easier and all the interfaces with the hardware are just built and ready to be integrated with AutomatedSOS service. Furthermore, this service is only on the client side and it doesn't require the communication with the TrackMe servers. Note that the third party application is always the same application with a different user interface, which allows sending the data request message to other users or to TrackMe system, manages their answers and download data from the servers and it disables the other functionality available in the user application. The discrimination between the two GUI is the kind of account, indeed during the subscribing process, the user chooses the type of account (i.e. simple user or third party) and obviously provides different information. Now two application's features are developed and at the end, after having configured the maps API, there is time to implement the third feature (i.e. Track4Run), that result to be separate from the other two and it rather easy to develop client side.

5.2 Test plan

Considering the implementation plan of the TrackMe system, the chosen strategy for the integration test is bottom-up. Starting from the leaves of the hierarchy, it never needs the use of stubs. This kind of strategy requires, therefore a unit test for each module. A unit is the smallest possible testable software component. Usually, it performs a single cohesive function. This decision is supported by the fact that a unit is small, so it is easier to design, execute, record, and analyze test results for than larger chunks of code are. Defects revealed by a unit test are easy to locate and relatively easy to repair. Furthermore, since a lot of services have several features, units tests result from the perfect choice. Indeed, one of the main benefits of unit testing is that it makes the coding process more agile and it improves the quality of the code. A unit test also helps to achieve a good line coverage, that for this project must be at least 90% for each service and components, which has presented in the *Component view* section. Note that for critical components, like message queue, router and API Gateway

the coverage will be 100% of lines. When the message queue component and two services are developed and their unit tests are performed, the integration test can start.

5.2.1 Entry Criteria

As just described, the integration test should start as soon as two components of the system and the message queue are released, but some preconditions should be satisfied:

- In order to test the system components, some low-level modules and external APIs should be available. In particular:
  1. For integration tests involving the AutomatedSOS feature, the user application with each one of its components (i.e. SIM service, GPS sensor, and User HW devices) should be fully implemented.
  2. For integration tests involving maps components, like Race service and Spectator service, the Maps API should be available and fully usable.
  3. The DBMS of the services, that should be tested, should be configured and operative in order to allow to test all the components which need access to the databases.
  4. For integration tests between server side and client side requires the API Gateway.
- In order to test the integration of two components, the main features of both of them should have been developed and, as already mentioned, the related unit tests should have been performed.

Note that the message queue component is fundamental for the integration test, without it, communication between two services is not available. For this reason, the message queue is the first component done and its line coverage is so high. A good test of this module discriminates the possible causes of error in the debugging process during the integration test.

5.2.2 Elements to be Integrated

Referring to the *design document*, the system is composed of several components, that can be divided into three categories:

- Front-end components: mobile application (i.e user application and third party application)
- Back-end components: All services deployed on the server and the communication components.
- External components: all the components which refer to functionalities provided by external service and the DBMS.

There are four types of partial integration: back-end with external components, back-end with itself, front-end with external components and front-end with the back-end. For the back-end components the integration can proceed in this way:

- Integration of the internal services with the external service. The main integration of back-end components with external components are:
  - Account service
    1. Account manager service, DBMS
  - Race service
    1. Race manager service, DBMS, Maps API
  - Spectator service
    1. Spectator manager service, DBMS, Maps API
    2. Spectate athlete manager service, DBMS, Maps API
  - Share data service
    1. Access data service, DBMS
    2. Send data service, DBMS
  - Group request service
    1. Group request manager service, DBMS
  - Individual Request Service
    1. Individual request manager service, DBMS
    2. Upload response service, DBMS
- Integration among the services inside the server side.
  - Account service, message queue, Individual request service
  - Account service, message queue, Shared Data service
  - Shared Data service, message queue, Individual request service
  - Shared Data service, message queue, Group request service
  - Race service, message queue, Spectator service

The integration between front-end components and external components is:

- User application, SIM service, GPS sensor

Finally the main integration of front-end components with back-end components are:

- User application, API Gateway
- Third party application, API Gateway

5.2.3 Integration testing strategy

Given that the entire development effort mainly follows a bottom-up strategy, it is better to opted to use a similar approach with respect to the testing phase. Due to the complexity of the system for a better visibility a critical modules testing strategy should be performed on critical services.
When all the component are made and the application is ready for testing if the system satisfied the goals and the non-functional requirement described into the RASD document, a system test will be performed. For completeness, below it is presented the three system tests strategies that will be used on the system:

1. Performance test: it checks if the TrackMe application respects the performance described. In this particular case, it checks if the system performs a SOSCall within 5 seconds when the constraints are satisfied.

2. Load Testing: the load test exposes bugs such as memory leaks, mismanagement of memory and buffer overflows, excellent for application, as this, that takes data from the sensors and tries to manage them in a correct way.

3. Stress Test: finally this test checks the availability, reliability, and scalability of the system.

As well shown in this section of the document, the test is taken into great consideration, because of the TrackMe application is responsible for the life of the people that use it. Indeed the security and the correctness of the software must be better as possible when a human's life is in danger.

6 Effort Spent

6.1 Riccardo Poiani

Date	Task	Hours
17/11	Architecture overview	2
18/11	Architecture overview, component diagram	7.5
19/11	Component diagram, architectural style	3.5
20/11	Sequence diagram and revision	3.5
21/11	Component interfaces, sequence diagrams	6
22/11	Patterns, other design decisions	5
23/11	Fix overview and component view	4
24/11	Fix component interfaces	1
25/11	Document revision	2.5
10/12	Document revision	4
	Overall	39

6.2 Mattia Tibaldi

Date	Task	Hours
21/11	Introduction	3
22/11	Introduction and User interface design	2.5
24/11	Requirements traceability	6
25/11	Document revision	2.5
26/11	Test Plan	5.5
27/11	Test Plan and revision	5.5
9/12	Document revision	1.5
10/12	Document revision	2.5
	Overall	29

6.3 Tang-Tang Zhou

Date	Task	Hours
17/11	Architecture overview	2
18/11	Architecture overview, component diagram	7.5
19/11	Deployment diagram	3.5
20/11	Sequence diagram and revision	3.5
21/11	Sequence diagram	5
22/11	Fix diagram	4.5
23/11	Fix Deployment diagram, design patterns, and definitions	5
24/11	Add/Fix reference documents, document structure, revision history and definitions	2
25/11	Document revision	2
09/12	Fix diagram	4.5
10/12	Document revision	4
	Overall	43.5