POLITECNICO
MILANO 1863

Software Engineering 2

AA 2018/19

# Design Document

*Authors:*
**Federico Ferri** 10522586
**Ahmad El Bayoumi** 10425225

December 10, 2018

# Contents

.

# 1 Introduction

## 1.1 Purpose

In this document we present the architectural styles and design choices that are made for TrackMe. The main purpose has been already touched upon the argument in the REQUIREMENT ANALYSIS AND SPECIFICATION DOCUMENT (see Reference documents), however,we will now go into more specific details, explaining through diagrams and comments an ideal architectural structure of the system as well as showing the algorithms that are specifically designed for the application.

## 1.2 Scope

In relation with the scope of the document, some requirements were simplified in order to present in a correct way the design decisions. Here is shown a general design of the software, however there are some details that can only be decided during the implementation and therefore are not present in this document. TrackMe is an application that is composed of two different services, one is Data4Help, that aims to make it easier for third parties to track the movements and the fitness state of a group of individuals, the other is AutomatedSOS that is used to help elders to have emergency assistance in case of need. Therefore, it is targeted for every user who needs to achieve such purpose. In particular, the system:

- provides third parties with the possibility to request for informations about a specific group of people or for an individual
- checks if it follows privacy guidelines
- sends the requested informations to said party
- constantly tracks health status of elders
- sends an alert if the individual vital signs become dangerous

The main purpose of TrackMe is to improve the relationship of the common citizen with the government and the companies that trough the informations gathered by the applications can give better services to the customer and ditch services that are not necessary for the community, all of this is obtained through the information report that is given to the ”third party” after a request is approved. An information report must contain all of the requested informations in the best way possible so that it can guarantee an optimal service for the third party and a great respect towards the user’s privacy.

## 1.3 Definitions, Acronyms, Abbreviations

DEFINITIONS

D-1 **Information**

noun - facts about a situation, person, event, etc.

D-2 **Request**

noun - the act of politely or officially asking for something

D-3 **Vital Signs**

noun - signs that show the condition of someone’s health, such as body temperature, rate of breathing, and heartbeat

D-4 **Fitness Level**

noun - (Fitness + Level): the condition of being physically strong and healthy + the amount or number of something

D-5 **Position**

noun - the place where something or someone is, often in relation to other things

D-6 **Emergency Call**

noun - an occasion when you use the phone in order to speak with emergency services

D-7 **System**

noun - a set of computer equipment and programs used together for a particular purpose

D-8 **User**

noun - someone who uses a product, machine, or service

D-9 **Feasible request**

adjective + noun - a request that is able to be made, done, or achieved

ACRONYMS

ACR-1 **GPS**: Global Positioning System

ACR-2 **EC**: Emergency Call

ACR-3 **API**: Application Programming Interface

ACR-4 **DR**: Data Request

ACR-5 **UD**: User Data

ACR-6 **TP**: Third Party

ABBREVIATIONS

ABR-1 **D-n**: Definition number n

ABR-2 **ACR-n**: Acronym number n

ABR-3 **ABR-n**: Abbreviation number n

ABR-4 **RE-n**: Functional Requirement number n

ABR-5 **G-n**: Goal number n

### 1.4 Revision History

Version 1.0 Released on December 10th,2018 - 9 pm

### 1.5 Reference Documents

Mandatory Project Assignment AY 2018-2019.pdf

# 2 Architectural Design

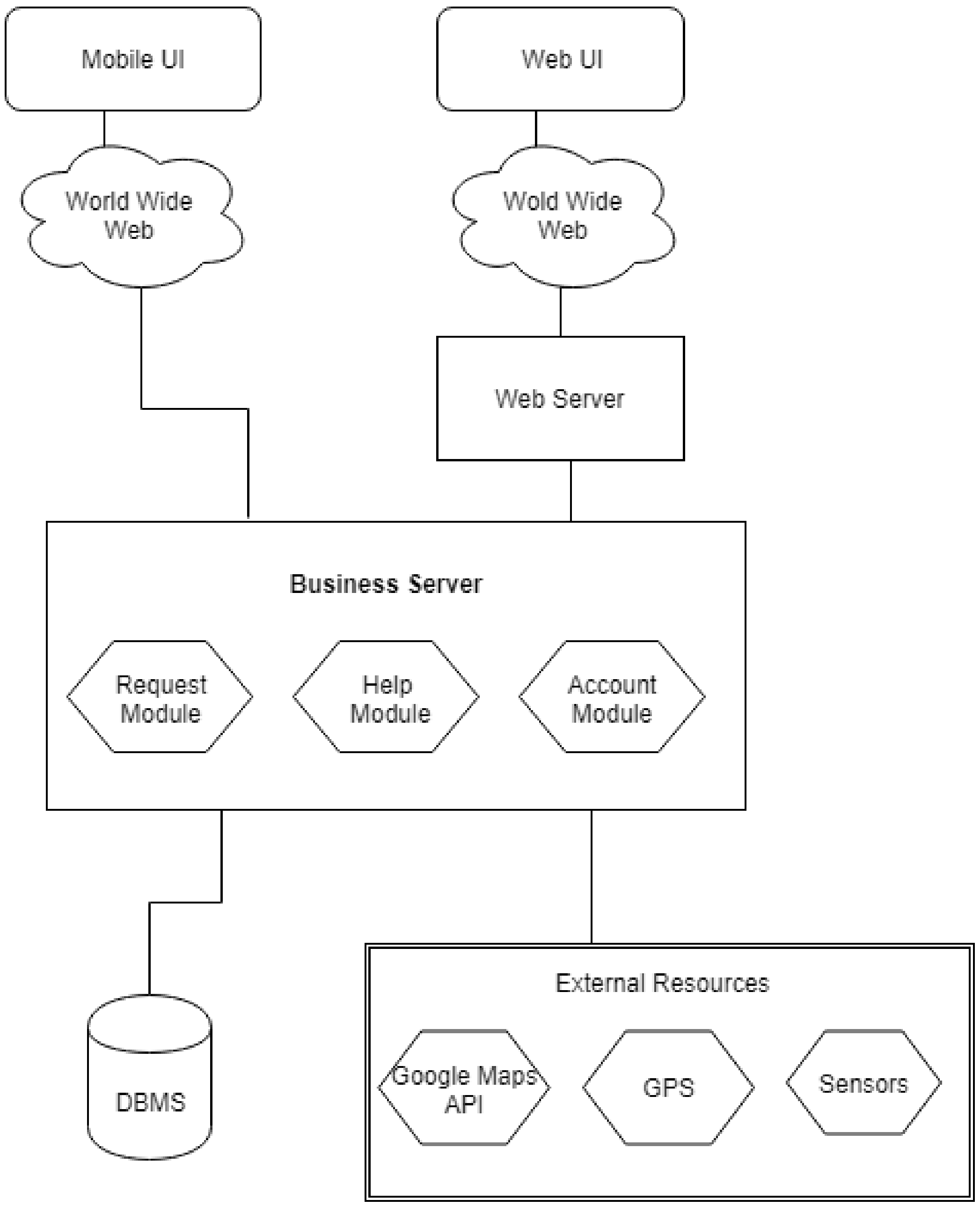
## 2.1 Overview

The system architecture is defined using a top down design method. First of all, the components related with the main functionalities of the system are introduced at high level perspective. The organization of the components allows the use of an architecture based on the client-server paradigm, where the main logic of the system is implemented into the business server and the different interfaces act as clients. In particular, we adopted the "Remote Presentation" type in order to distribute application logic among different tiers, that means there is no logic in the layers on the user side. We distinguished four tiers:

- User Interfaces
- Web Server
- Business Server
- Data tier: contains the DBMS and external resources.

The user will be divided in two categories, the consumer, that will use the mobile app in order to provide informations, and the "third party", who will use the web app in order to make request and analyze previous requests. Note that for the mobile app there are only 3 tiers, since the web server is not accessed.

The first tier contains two different interfaces: depending on the type of the user it is possible to interact with TrackMe using a browser interface or a mobile interface. The system is designed such that it uses the same business server that will respond in a different way depending on which interface is being used. In the case of mobile interface the communication is "direct" therefore all of the messages are directly "translated" in the app.Instead in the case of the web interface the data comes from a intermediate tier called Web Server (second tier). TrackMe's web server has the task of receiving the messages sent from the server and converting them into web pages that are readable by the user's computer. In both cases the communication is carried out by Internet protocols.



The third tier is the Business Server where the application logic runs. On there we have some general modules:

- **RequestModule**

  This module is responsible for handling all of the requests made by third parties according to the criterias that they selected while making said request, moreover this module has to check if certain requests should be allowed and needs to be able to control all of the steps needed to perform individual requests

- **SOSModule**

  SOSModule is the module made to interact with AutomatedSOS, it will need to handle independently all emergency requests that will be triggered by the device carried by the user

- **AccountModule**

  It contains everything related with the management of the user information such as personal information and preferences.

- **MonitoringModule**

  MonitoringModule is the key module when it comes to tracking the user this module is needed to track the gps position and to calculate his fitness level

The fourth and final layer contains the database system that is responsible for managing all of the application data. All of the modules that are found inside of the server are connected with this particular layer. Inside the database all of the informations about the users, his location informations and it's fitness level. The other part of the fourth layer is made of the external resources, in this particular part we find all of the resources that are used by at least some of the components found in the system, these components have to provide all of the technical information of the user such has GPS and Fitness infos (Sensors). Google Maps'API will be used to make it easier to select geographical areas in which the third party will be interested about knowing informations, this particular API will be used to track movements of the users too and make an overall history.

## 2.2 Architectural Attributes

### 2.2.1 Performance

It is important to remember that most of the functionalities of the systems are carried out in a main server,meaning that all its resources are used by the algorithms. This decision was taken since it is easier to control the status of only one server in order to take advantage of its resources and by running most of the "logic" on the server the UI have to only charge data from making the whole user experience much faster. However we need to remember that the final result also depends on the user itself, for example internet connection of the user when the software it's used, for example the system necessary to make an emergency call relies on it. Additionally, the data base performance is clearly depending on the technology to use. In this case a well structured and indexed relational database will give a good performance, such that the queries are returned in the best possible time. Another important point is the performance of the external resources like APIs, which is not responsibility of TrackMe, however there shall exist a control over their performance, in order not to impact too much the time to respond to the application

### 2.2.2 Security

The most important characteristic of the system is to take into account the two attributes for data: confidentiality and integrity. Security is guaranteed thanks to the security of the system that keeps all of the confidential informations into the database which lays beyond many level of "defense" such as a firewall and by being totally encrypted. Integrity is guaranteed tanks do DBMS which keeps all the informations into parallel arrays of servers and therefore it will protect them.

### 2.2.3 Availability

It should be noted that the modules are built and work in parallel, this creates a good availability for all of the functionalities in the software. Also different modules will have instances for each different user and therefore they will allow to have a pool of resources where each user, depending on it's "type", has its own group of functionalities available.

### 2.2.4 Modifiability

There are many reasons for which we can say that this architecture is a good choice. The first reason to say so is that components are separated and therefore each component carries out a different objective. Another one should be that the system preserves the two principles design: reduce coupling and keeping a high level of abstraction, in the sense that it is possible to add new modules or delete modules while the system is still working without creating damages. The latter would be really useful in case for example we would like to add a new way of checking a person's medical health by implementing the use of a new sensor

## 2.3 Component View

Inside of the component view analysis we used all of the components of the high level architecture and we devided them in the same way (4 tiers).

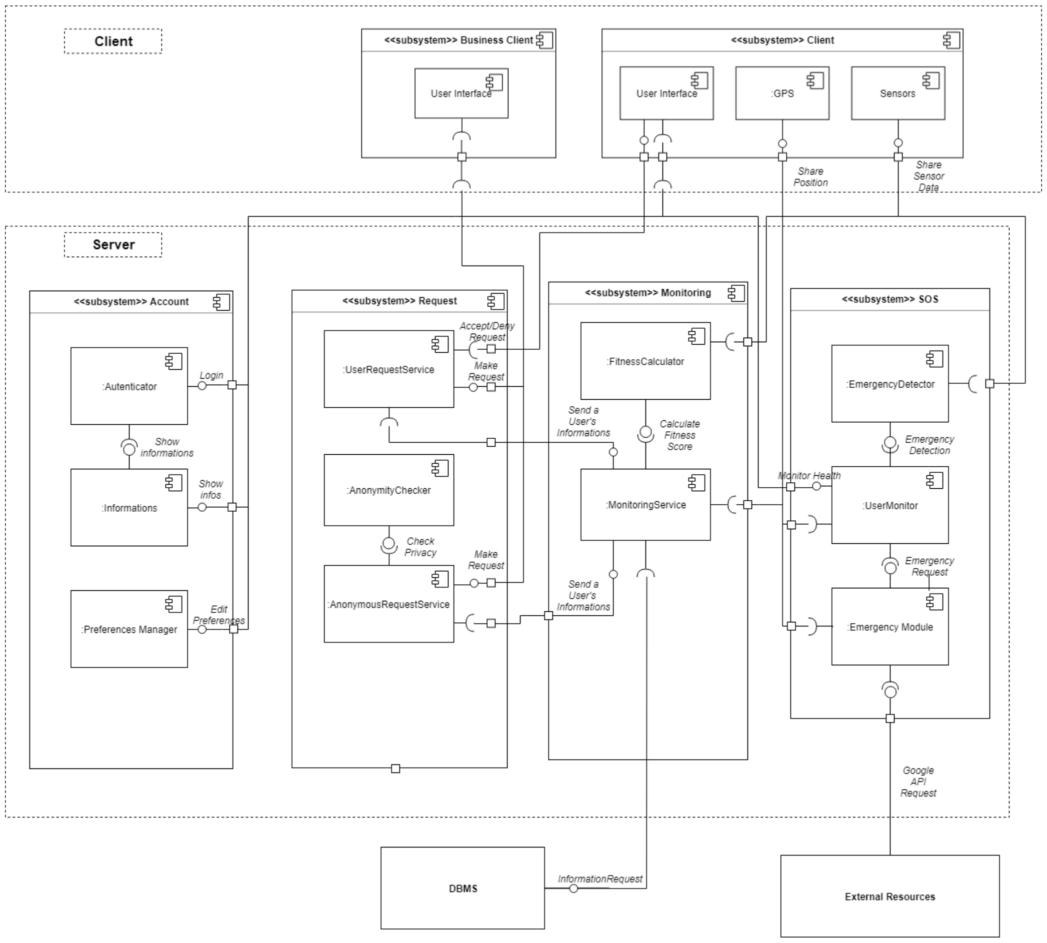
TrackMe is based on a client-server architecture that uses a thin client (in fact it is the server that what mostly gets the job done) therefore we can define what the client and the server do for the application:

- **Client** : The client side is very slim as it plays the role of the "middle-man" between the user and the back-end part of the system which will be the job of the "User Interface". Inside of the client we find the GPS and the sensors module too, these are adapters that signal to the server the position of the user and his current vital signs(through which we can easily find the vital signs).

- **Server** Inside of the server we can find three "big" components:

  - **Request**: The Request subsystem has to manage all of the requests made by the third parties and in case it has to handle communication with the single user and the DBMS.
  - **Account**: The Account subsystem has to manage all of the account informations and it need to let the user register and log-in.

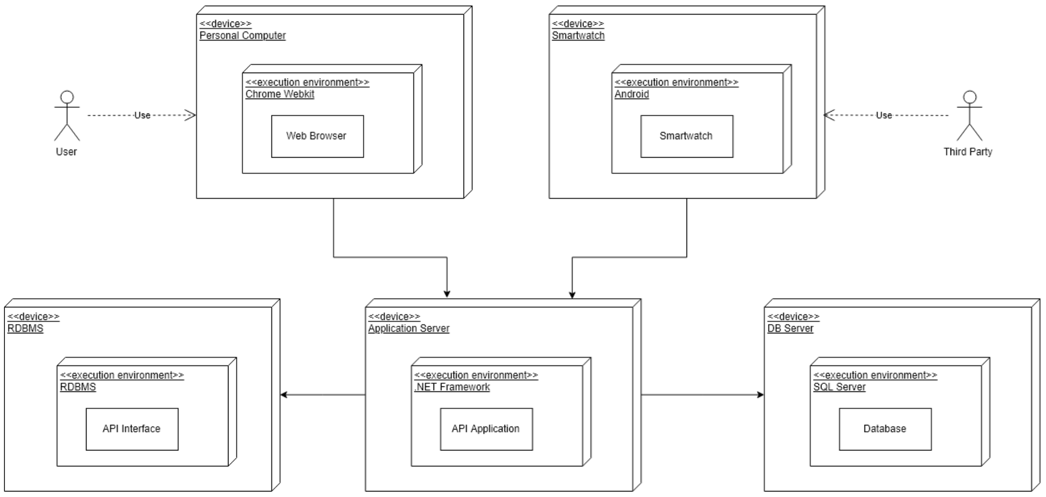
- **SOS**: The Help subsystem has to manage all of the required modules needed for AutomatedSOS, these will need to handle an emergency request and check the user's vital signs.
- **Monitoring**: The Monitoring subsystem is needed to track all of the resources of the user, such as GPS and vital signs, it will have to calculate a "fitness score" too



In addition to the two components specified above we have the DBMS, whose purpose is to handle of the data of the application, and the "external sources" which is composed of all the external interfaces (such as Google Maps' API).

## 2.4 Deployment View

Below we can find the deployment view for this project:



1. Smartwatch: this is the wearable application that will be used by the user that can be monitored. The application will retrieve all the measures form the sensors of the smartwatch and will periodically send them to the application server.

2. PC: this client will be used by the third-parties to do new monitoring requests and to view information about users yet monitored.

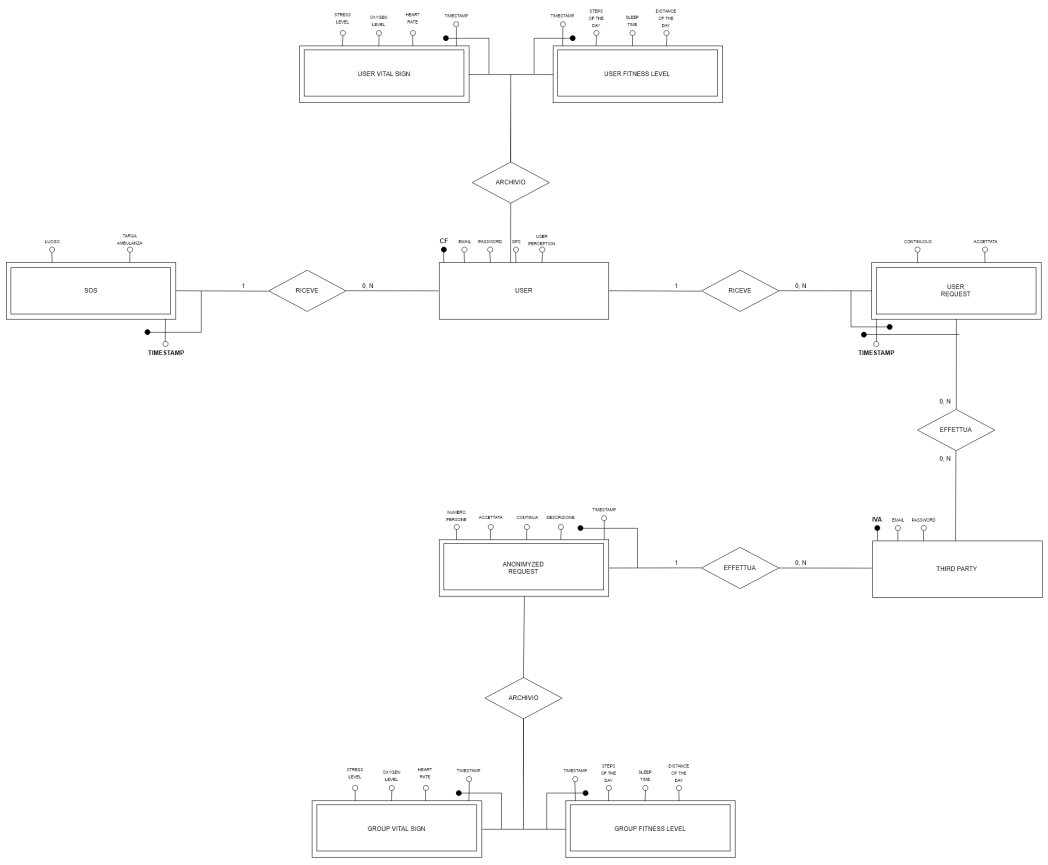
3. Application server: it contains the core logic of the application. The server will gain information and measures from the user client and notify the third-party client with fresh data. It has the responsibilities first of handling the requests made by third parties and for second of archiving periodically the health status of users and groups of anonymous users requested by third parties.

4. DB Server: the DB will store the account information (credentials, email and so on..) and keep track of all requests made, accepted or not, and all the users data monitored by the third-parties.

5. External Service: these services will provide API demanded by the application server (e.g. Google Maps)

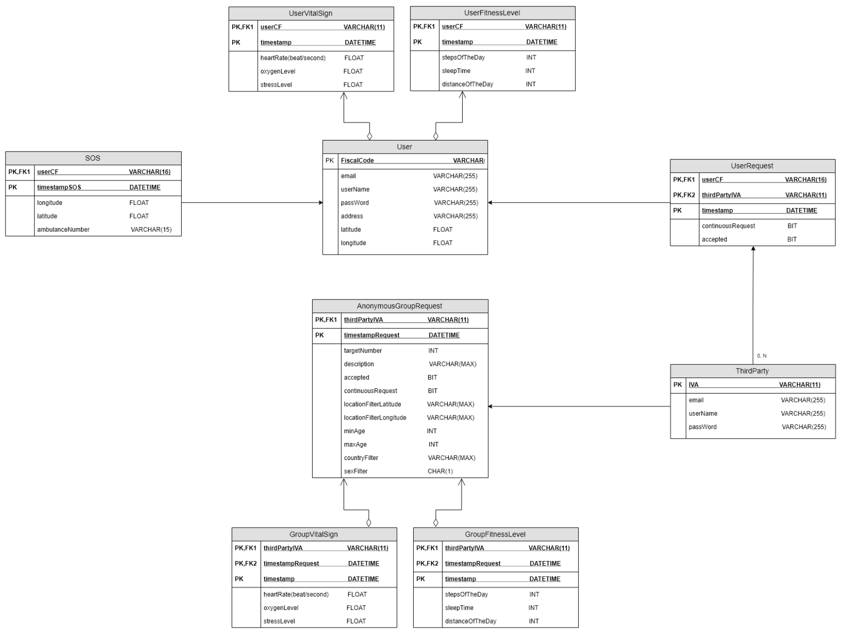
### 2.4.1 Database View

To have a better view on how the database will be structured we created an entity relationship diagram that is shown below:



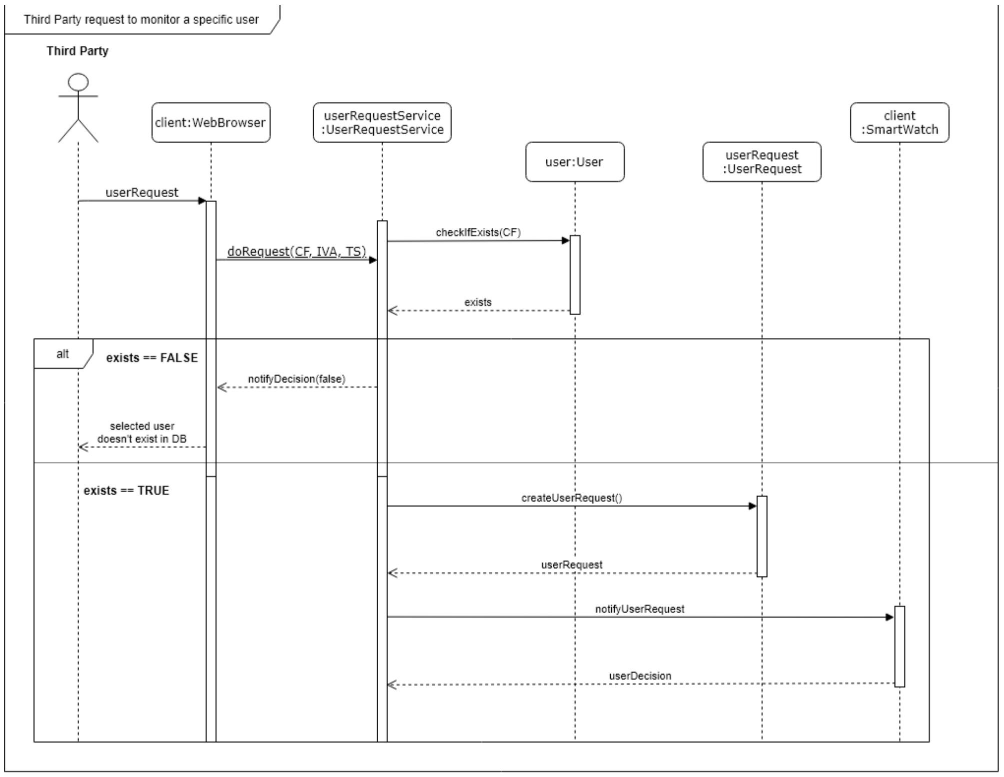
### 2.4.2 Database Implementation

After building the high level diagram of Entity Relation schema, we show how the schema will look like into the database system. With this schema we also show which types are used to define each attribute of the tuples.



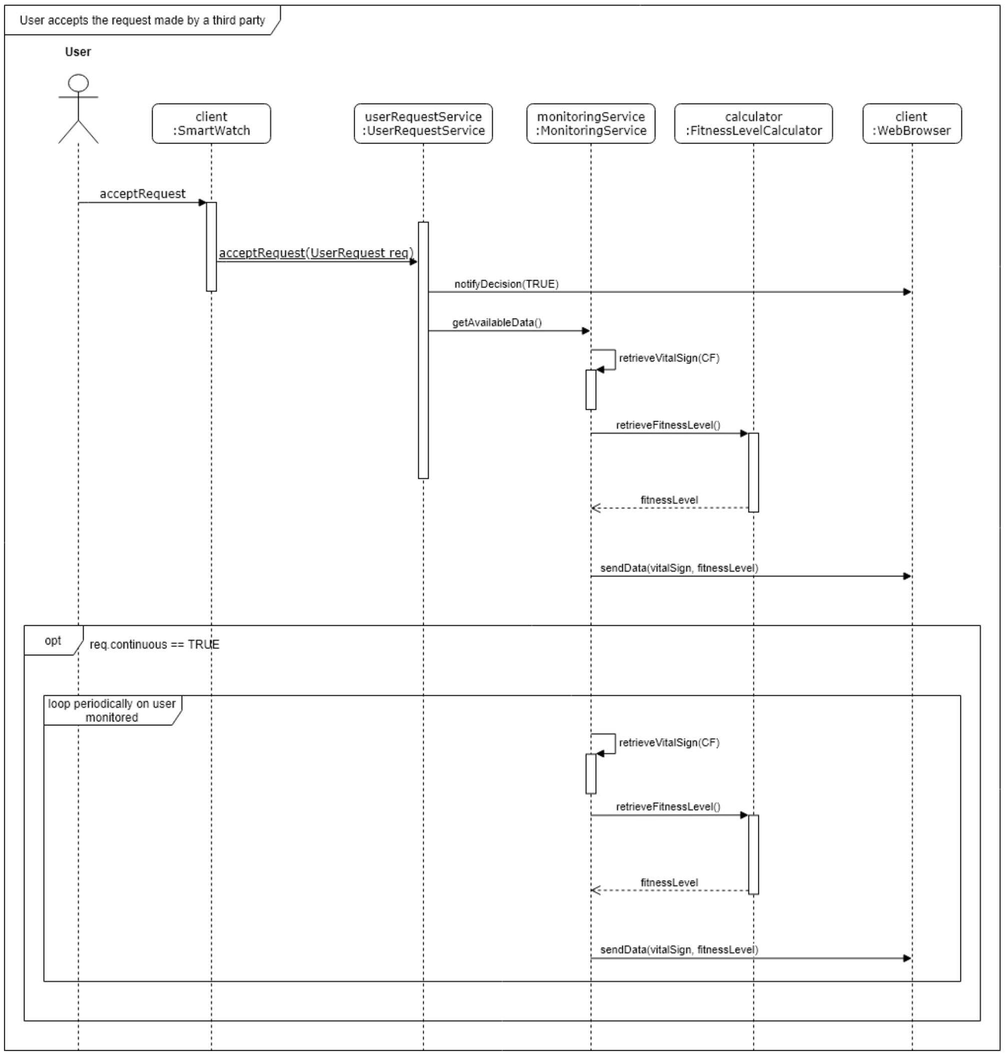
## 2.5 Runtime View

### 2.5.1 Third Party request to monitor a specific user



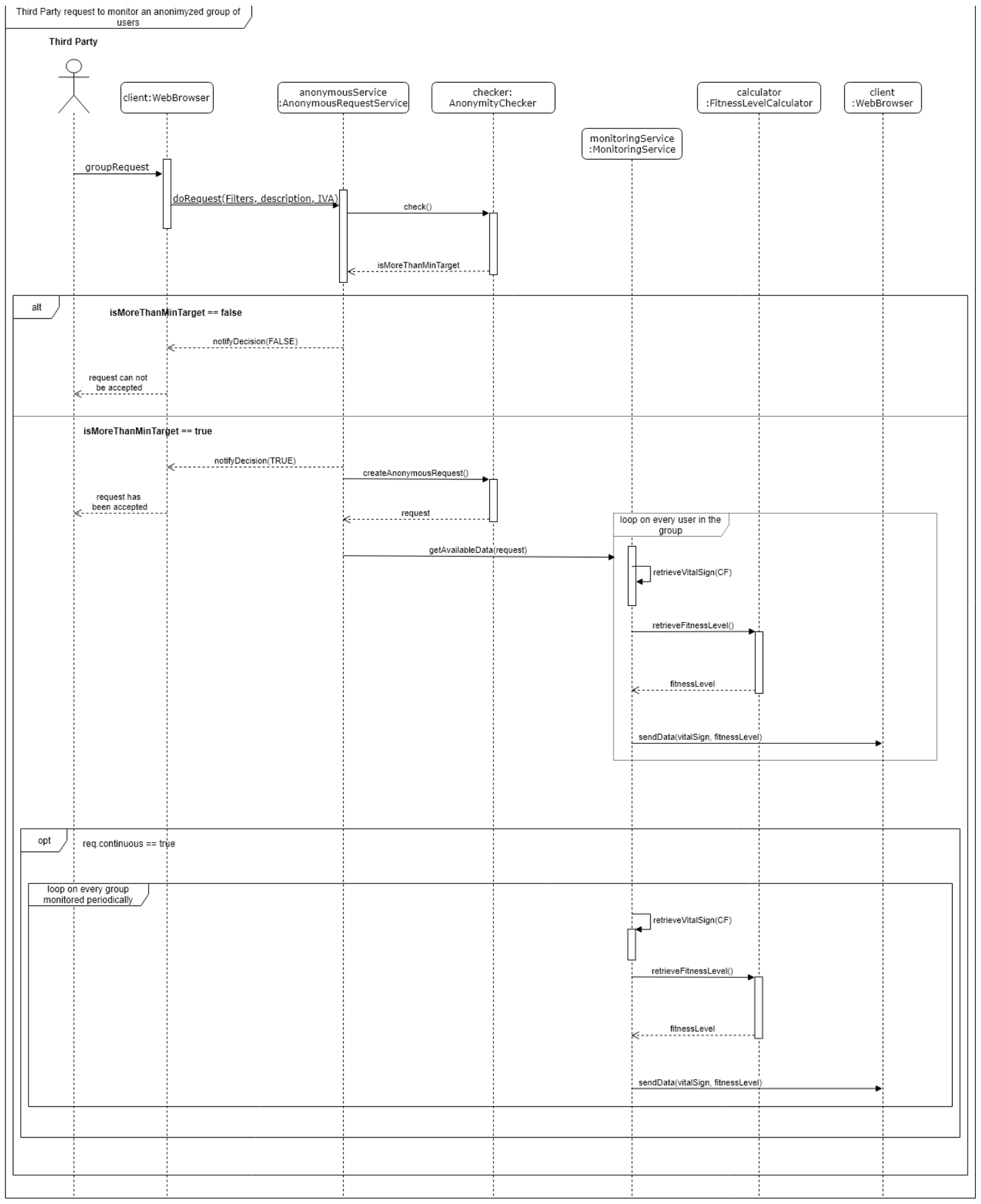
This sequence diagram show the process of a third party requesting to monitor a user by specifying his fiscal code. The UserRequestService first of all checks if the user exists, and notifies the third party in both cases. If the user exists, the same service create the UserRequest and forward the request to the user and waits its response.

### 2.5.2 User accepts the request made by a third party


User accepts the request made by a third party
User
client :SmartWatch
userRequestService :UserRequestService
monitoringService :MonitoringService
calculator :FitnessLevelCalculator
client :WebBrowser
acceptRequest
acceptRequest(UserRequest req)
notifyDecision(TRUE)
getAvailableData()
retrieveVitalSign(CF)
retrieveFitnessLevel()
fitnessLevel
sendData(vitalSign, fitnessLevel)
opt
req.continuous == TRUE
loop periodically on user monitored
retrieveVitalSign(CF)
retrieveFitnessLevel()
fitnessLevel
sendData(vitalSign, fitnessLevel)

This diagram shows the continuation of the process explained above. The user has been previously notified of the third party's request and he decides to accept it. The UserRequestService handles the response first by notifying the third party of the fact, and then it passes the ball to the MonitoringService, which will first retrieve the VitalSign information of the user (heart rate, oxygen level, stress level and user perception) and then will delegate the FitnessCalculator to use its algorithm to define a FitnessLevel of the user, based on his sensor's measures. Finally it forwards the results to the interested third party. Now, if the request is a single request (not continuous monitoring), then the process stops, otherwise the MonitoringService periodically retrieves VitalSign and FitnessLevel informations as shown above.

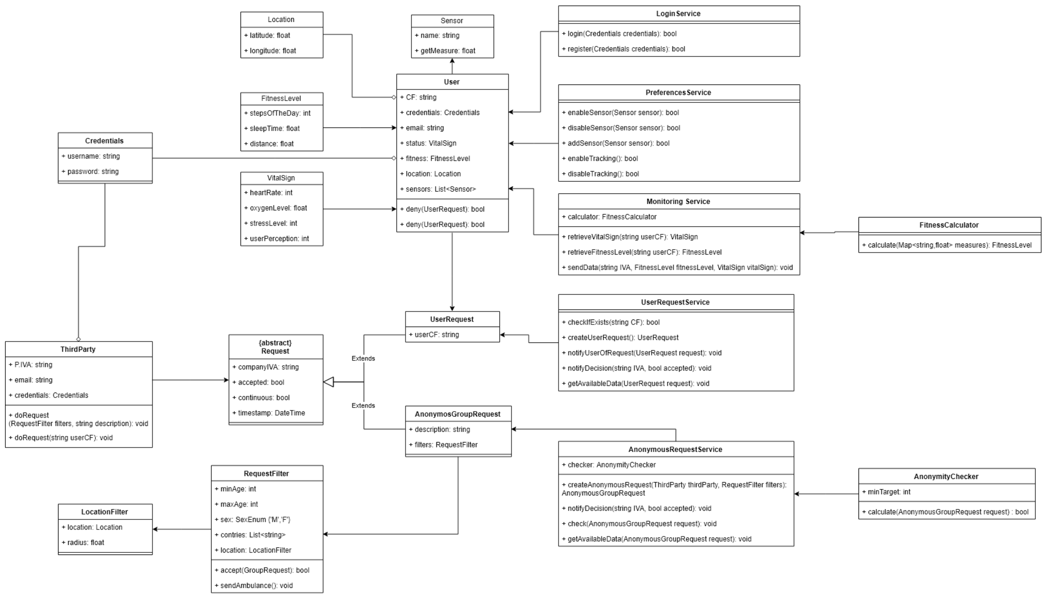
### 2.5.3 Third Party request to monitor an anonymized group of users


Third Party request to monitor an anonimyzed group of users
Third Party
client:WebBrowser
anonymousService :AnonymousRequestService
checker: AnonymityChecker
monitoringService :MonitoringService
calculator :FitnessLevelCalculator
client :WebBrowser
groupRequest
doRequest(Filters, description, IVA)
check()
isMoreThanMinTarget
alt
isMoreThanMinTarget == false
notifyDecision(FALSE)
request can not be accepted
isMoreThanMinTarget == true
notifyDecision(TRUE)
createAnonymousRequest()
request has been accepted
request
loop on every user in the group
getAvailableData(request)
retrieveVitalSign(CF)
retrieveFitnessLevel()
fitnessLevel
sendData(vitalSign, fitnessLevel)
opt
req.continuous == true
loop on every group monitored periodically
retrieveVitalSign(CF)
retrieveFitnessLevel()
fitnessLevel
sendData(vitalSign, fitnessLevel)

In this process, a third party request information about a generic group of users by defining some filters (e.g. age, country, city..). The request is passed to the AnonymousRequestService that handles it by checking the feasibility of the request. It achieves this result by delegating the operation to its AnonymityChecker component, that checks if the cardinality of the required group of users is minor to a prefixed number sufficient to anonymize the identity of the users. If the result is positive than as usual the third party will be notified, then the AnonymousRequestService will ask the MonitoringService the information needed of every user included in the group and finally send it to the third party. As for the UserRequest, we can have a continuous monitoring request, in this case again the MonitoringService will periodically send to the third party the information of all users of the group (obviously keeping anonymized the data by sending the average of the information).

## 2.6 Component interfaces

Below we present the component interfaces, this present the dependencies between the applications and the server.


Location
+ latitude: float
+ longitude: float
Sensor
+ name: string
+ getMeasure: float
LoginService
+ login(Credentials credentials): bool
+ register(Credentials credentials): bool
User
+ CF: string
+ credentials: Credentials
+ email: string
+ status: VitalSign
+ fitness: FitnessLevel
+ location: Location
+ sensors: List<Sensor>
+ deny(UserRequest): bool
+ deny(UserRequest): bool
FitnessLevel
+ stepsOfTheDay: int
+ sleepTime: float
+ distance: float
PreferencesService
+ enableSensor(Sensor sensor): bool
+ disableSensor(Sensor sensor): bool
+ addSensor(Sensor sensor): bool
+ enableTracking(): bool
+ disableTracking(): bool
Credentials
+ username: string
+ password: string
VitalSign
+ heartRate: int
+ oxygenLevel: float
+ stressLevel: int
+ userPerception: int
Monitoring Service
+ calculator: FitnessCalculator
+ retrieveVitalSign(string userCF): VitalSign
+ retrieveFitnessLevel(string userCF): FitnessLevel
+ sendData(string IVA, FitnessLevel fitnessLevel, VitalSign vitalSign): void
FitnessCalculator
+ calculate(Map<string,float> measures): FitnessLevel
UserRequestService
+ checkIfExists(string CF): bool
+ createUserRequest(): UserRequest
+ notifyUserOfRequest(UserRequest request): void
+ notifyDecision(string IVA, bool accepted): void
+ getAvailableData(UserRequest request): void
UserRequest
+ userCF: string
ThirdParty
+ P.IVA: string
+ email: string
+ credentials: Credentials
+ doRequest
(RequestFilter filters, string description): void
+ doRequest(string userCF): void
{abstract}
Request
+ companyIVA: string
+ accepted: bool
+ continuous: bool
+ timestamp: DateTime
Extends
Extends
AnonymosGroupRequest
+ description: string
+ filters: RequestFilter
AnonymousRequestService
+ checker: AnonymityChecker
+ createAnonymousRequest(ThirdParty thirdParty, RequestFilter filters): AnonymousGroupRequest
+ notifyDecision(string IVA, bool accepted): void
+ check(AnonymousGroupRequest request): void
+ getAvailableData(AnonymousGroupRequest request): void
AnonymityChecker
+ minTarget: int
+ calculate(AnonymousGroupRequest request) : bool
RequestFilter
+ minAge: int
+ maxAge: int
+ sex: SexEnum {'M','F'}
+ contries: List<string>
+ location: LocationFilter
+ accept(GroupRequest): bool
+ sendAmbulance(): void
LocationFilter
+ location: Location
+ radius: float

## 2.7 Selected architectural styles and patterns

### 2.7.1 Multi-layer architecture

The most fitting architecture for the system is a layered architecture. We have at top level a Presentation Layer which is both the two GUI of the application, the user client one and the third party one. The Presentation Layer communicates with the below layer, the Application one. This layer provides the main logic of the system application. The last one is the Data Layer which is the only one that handles the database and offers to the Application Layer the API's needed to retrieve and edit the saved data.

### 2.7.2 Client-Server

As evidenced by the presented architecture, the entire solution is based on the client-server paradigm. In particular we have two different clients, a web platform client used by third parties to make monitoring requests, and an Android application for wearable devices used by users. This last one benefits of the built-in API's of Android WearOS for retrieving the informations from the sensors of the device (typically a smartwatch). The server acts as a listener for the third party client, as it waits for requests made by third parties and handles them by its own if we are considering the anonymous group requests, or by forwarding the request to the user if the request concerns a specific user.

### 2.7.3 Model-View-Controller pattern

The MVC solution is concretized in the nature of the ASP.NET MVC project. This pattern is the best way to benefit of our multi-layer architecture. The main view of this system is for sure the third party Web application. This is the view of our architecture. The model is the data contained in our database, optionally concretized in our code with the use of an ORM like Entity Framework. The controller interacts with both the two views in two ways:

- **HTTP POST Endpoints:**

  These endpoints are used by third parties to submit their requests to the server and, in case of user requests, forwarding to the users.

- **Publisher / subscriber paradigm:**

  Every time our server application retrieves new fresh data from the users client, it notifies the subscribed third parties of the new data available.

## 3 User Interface Design

The mock-ups needed for this project were already presented into the RASD and, since there hasn't been an update on those yet we are not going to repeat what we already specified inside of the RASD.

## 4 Requirements traceability

In this section are shown the requirements and the goals as they are shown in the RASD. As a general rule, :UserInterface component is used to show to the user all the functions and calculations performed on the back-end components, therefore we omitted to write it.Below are listed all of the design components to which TrackMe goals and requirements are mapped:

[**G 1**] - The third parties should be able to make requests for informations

RE-1 : The system must allow the user/third party to login using the correct credentials
**:Authenticator**

RE-2 : The system must allow the third to insert a general profile for his targeted users
**:AnonymousRequestService**

[**G 2**] - The user should be able to accept/refuse individual requests

RE-3 : The system must allow the user to register a new account

**:Authenticator**

RE-4 : The system must allow the user to accept/deny a request

**:UserRequestService**

[**G 3**] - The device should be able to monitor the user's vital signs

RE-5 : The system should ask the device for the permission to access certain sensors

**:FitnessLevelCalculator**

RE-6 : The system should ensure that it will constantly run in the background

**:UserMonitor**

[**G 4**] - The user should receive useful informations on how to improve it's general health

RE-7 : The system should periodically analyze the data given by the user

**:FitnessLevelCalculator**

RE-8 : The system should periodically output a report of the analyzed data

**:UserMonitor :Informations**

[**G 5**] - The third parties should be able to subscribe to informations updates about a specifc group that they previously targeted

RE - 9 : The system should allow third parties to subscribe to data from a specific group of people

**:AnonymousRequestService :MonitoringService**

RE - 10: The system should constantly monitor groups to which a third party is subscribed

**:MonitoringService**

RE - 11: The system should automatically send informations, once updated, to third parties

**:AnonymousRequestService**

[**G 6**] - The devices should be able to call for an ambulance in case the vital signs drop

RE - 12 : The system should be constantly connected to the emergency services

**:EmergencyModule**

RE - 13 : The system should ask for permission to make calls

**:EmergencyModule**

# 5 Implementation, Integration and Test planning

## 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 afterwards 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. As of the user's client, Java (Android studio), was the chosen programming language since it is perfect for the mobile implementation, in particular, the client, is going to be implemented inside of a smartwatch device that runs Android Wear. For the third party's client, instead, we decided to have a more web-based implementation since the app is going to be accessed through the Web, we therefore chose to use AngularJS framework that is based on the Typescript programming language. .NET Core is our choice for the development of the back-end part since the whole system is going to be implemented using Microsoft's technology, therefore because of this we decided to implement the database by using SQL Server. For the testing part we decided to focus on the following testing tools:

- NUnit: for the testing of the server side platform we decided to use NUnit since it is the recommended testing tool for .Net Core.
- JUnit: as of the android application we decided to stick with JUnit being the most popular testing tool for the Java programming language and considering that it comes built in inside of the Android Studio IDE.

## 5.2 Implementation Strategy

As we previously said the implementation process will follow the bottom-up strategy: we therefore plan to implement first all of the components isolated from each other and the unit testing of the single component will begin only once at least 50% of the component will be developed.

### 5.2.1 Implementation Order

There is no mandatory order for the implementation of the components since the system is modular and the strategy that was chosen doesn't require a specific order. However since the system has some key features that need more time and effort to be fully implemented it is correct to start the implementation from those components:

- UserMonitor
- MonitoringService
- EmergencyModule

## 5.3 IntegrationStrategy

The main goal of the whole integration process is to avoid as much errors as it is possible at each step of the process in a way that ensures that 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: in the first stages the integration will begin only of components that have as few as possible requirements from other components or if they do not need other components to be integrated. Doing so will allow as to get an instant feedback about the system functionalities as soon as the components are released, moreover the integration of different non-related components can be run in parallel.

### 5.3.1 Integration Plan

Following what we already said into the introduction of the Integration Strategy (section 5.3) we will begin the implementation from this components:

- **:AnonymityChecker**
- **:FitnessCalculator**
- **:EmergencyDetector**
- **:EmergencyModule**
- **External Resources**

These modules do not need at all other modules since they are either completely independent or they only need the informations of specific sensors, in this way the integration testing is going to be very smooth since there is no need for assumptions. After the "roots" of the system are ready we are going to move a bit upwards and pass to modules that need support for the ones written earlier:

- **:UserMonitor**
- **:MonitoringService**
- **:PreferenceManager**
- **:Informations**

The second cycle of integration testing shouldn't be too difficult since all of the modules only use one or a few modules that were used in the beginning and therefore there shouldn't be lot's of problems with this specific stage. For the third cycle of developement we are going to develop some modules that are "advanced" but still not top tier and therefore are still needed for the implementation of other modules:

- **:AnonymousRequestService**
- **:Autenticator**
- **:UserRequestService**

The third cycle of testing might be a bit slower since there is a higher chance that the testing of the previous parts did not completely cover all of the possible cases. Once the third cycle of testing it's over we arrive to the final stage of the integration, which is the integration of the user interface

- **User Interface (User Client)**
- **User Interface (Business Client)**

## 6 Effort Spent

### 6.1 Federico Ferri

Section 1(Purpose,Scope,Definitions,Acronyms,Abbreviations): 7 hours
Section 2(Overview,Architectural Attributes,[Component,Runtime] View,component interfaces, Styles and patterns): 13 hours
Section 4 (Requirements traceability): 2 hours
Section 5 ( Implementation, Integration and Test planning): 11 hours

### 6.2 Ahmad El Bayoumi

Section 1(Purpose,Scope,Definitions,Acronyms,Abbreviations): 5 hours
Section 2(Overview,Architectural Attributes,[Component,Deployment] View,component interfaces, Styles and patterns): 15 hours
Section 4 (Requirements traceability): 1 hours
Section 5 ( Implementation, Integration and Test planning): 10 hours

## 7 References

Tools that were used to create this document:

- overleaf (online LaTeXeditor)

- draw.io (used to design all of the diagrams)

Other references:

- https://cloud.google.com/maps-platform/ (Google Maps API)
- .NET Core Platform
- https://developer.android.com/studio/ (Android Studio)
- AngularJS