# Trackme DD

## DESIGN DOCUMENT (DD)

## Release date: 10/12/18

Version 0.1

Table of content

[Trackme DD 1](#_Toc532145501)

[DESIGN DOCUMENT (DD) 1](#_Toc532145502)

[Release date: 10/12/18 1](#_Toc532145503)

[1. Introduction 2](#_Toc532145504)

[1.1. Purpose 2](#_Toc532145505)

[1.2. Scope 2](#_Toc532145506)

[1.3. Definitions, Acronyms, Abbrevations 2](#_Toc532145507)

[1.3.1. Acronyms 2](#_Toc532145508)

[1.3.2. Definitions: 2](#_Toc532145509)

[1.4. Revision History 3](#_Toc532145510)

[1.5. Reference Documents 3](#_Toc532145511)

[1.6. Document Structure 3](#_Toc532145512)

[1.6.1. Introduction 3](#_Toc532145513)

[1.6.2. Architectural Design 3](#_Toc532145514)

[1.6.3. User Interface Design 3](#_Toc532145515)

[1.6.4. Requirements Traceability 3](#_Toc532145516)

[1.6.5. Implementation, Integration and Test Plan 3](#_Toc532145517)

[1.6.6. Effort Spent 3](#_Toc532145518)

[2. Architectural Design 3](#_Toc532145519)

[2.1. Overview 3](#_Toc532145520)

[2.2. Component view 4](#_Toc532145521)

[2.3. Deployment view 5](#_Toc532145522)

[2.4. Runtime view 5](#_Toc532145523)

[2.5. Component interfaces 5](#_Toc532145524)

[2.6. Selected architectural styles and patterns 5](#_Toc532145525)

[2.7. Other design decisions 6](#_Toc532145526)

[3. Requirements Traceability 6](#_Toc532145527)

[4. Implementation, Integration and Test Plan 6](#_Toc532145528)

[4.1. General Presentation 6](#_Toc532145529)

[4.1.1. First step: Data Storing 7](#_Toc532145530)

[4.1.2. Second step: Requests Handling 7](#_Toc532145531)

[4.1.3. Third step: *Data4Help* load test 7](#_Toc532145532)

[4.1.4. Fourth step: EmergencyHandling 7](#_Toc532145533)

[5. Effort Spent 8](#_Toc532145534)

# Introduction

## Purpose

This is the Design Document of the TrackMe project. Its purpose is to present more technical details about the system to be implemented. It’s valid both for *Data4Help* and *AutomatedSOS:* when in some specific parts and diagram some differentiations must be done, it’s always explicitly indicated. Programmers and developers are supposed to read carefully this document and respect its principles.

## Scope

TrackMe is a project divided into two business services: the first one, *Data4Help*, aims at collecting data from users about their health status and provide them to subscribed third parties. This goal must be obtained respecting the privacy of all the users, which is always a crucial issue. On the other hand, the main scope of *AutomatedSOS* is allow third parties to provide medical assistance to the users in case of emergency. The services are not built independently each other, but the second one is built on top of the first one, and there is a unique system.

Therefore, the target of the application are both third parties and users, which can be called clients (see *Definitions*)

## Definitions, Acronyms, Abbrevations

### Acronyms

* SOA: Service Oriented Architecture
* GUI: Graphical User Interface
* API: Application Program Interface
* MVC: Model- View-Controller
* RASD: Requirements Analysis and Specification Domain

### Definitions:

* ***User”:*** person who interfaces to the application via a wearable device that provides his/her data to the system;
* ***“Third Party”:*** entity that interfaces to the web application in order to request data or offer assistance;
* ***“Client”:*** everyone using the service, both the person who wears the user and the third party;
* ***“Individual Request”:*** request of data of a single user from a third party; his permission is required;
* ***“Group Request”:*** request of a pool of anonymous data from a third party; if the number of users involved is less than 1000, the request is refused;

## Revision History

This is the first release of this document.

## Reference Documents

## Document Structure

### Introduction

In the first chapter there are some useful guidelines for the reader to understand properly the overall project and the document itself (i.e. its purpose, the notations used hereunder and so on).

### Architectural Design

Chapter 2 contains all the needed models, views and diagrams about the architecture of the application, and the explanations of all choices that have been made

### User Interface Design

Chapter 3 presents some interesting mock-ups, integrating the interfaces already presented in the RASD.

### Requirements Traceability

Chapter 4 puts in correlation all the components defined in this document with the requirements established in the RASD.

### Implementation, Integration and Test Plan

Chapter 5 points out all the details about how to implement and test all the elements, and how to integrate them to build the entire system and make sure everything works correctly.

### Effort Spent

In chapter 6 there are some data about the effort spent by each member of the group in elaborating this document.

# Architectural Design

## Overview

The architectural design of our application contemplates three logical layers:

* The **presentation laye**r handles the communication with the clients of the application, so it’s basically the GUI.
* The **application layer** contains all the services that we have already mentioned and is accessible from the presentation layer via some API.
* The **data layer** is concerned in storing data and giving access to them under request.

All the application layer is based on the SOA: every functionality to provide is associated to a technological service, and a service is implemented in a module. All the modules interact each other within a defined communication system, using the interfaces that every service exposes to the others. Because of that, there is huge decoupling between all the services, so that the ones that regard *AutomatedSOS* can be easily added to the system.

[bisogna spiegare meglio le SOA? Fare un discorso sulle prestazioni??]

There is no strict correspondence between the logical layers and the physical tiers where the application’s elements are deployed. In details,

* The **client device tier** could be represented (for the user) by a smartphone connected to a wearable device or, in alternative, by a smartwatch (as stated in the RASD document). Since some services are executed on this tier, it contains a part of the application layer in addition to the presentation layer. Further explanation for this choice is provided in this chapter. For the third party the tier is represented by a pc wo loads the web pages, and it contains only the presentation layer.
* The **web server tier** contains all the API endpoints for the website, so it is a part of the application layer which is responsible only for the third parties, since the users can access the system only via app. It only supports the correct working of the website: its business is transforming all the http requests in internal requests, increasing modularity and independency of the system (it introduces some internal classes (which are part of the model, see *Selected architectural styles and patterns*) to communicate between services, so every change in API does not affect all the other parts). Moreover, it contains all the static pages which do not need the participation of the other elements.
* The **application tier** consists of the remaining services (which actually are the majority). Here the core of the project is implemented, so a mainframe is required to guarantee that all the non-functional requirements are respected. This tier is the only one which communicate with all the others, to provide services or to ask for data.
* The **data tier** consists of a DBMS which is responsible for storing data in a consistent way, and to release when the application tier asks for them. It deals with all the data persistency mechanisms and how to avoid exposing them to the application layer.

In conclusion, the solution proposed is a *thick client* one, with three tiers and an extra intermediate tier to facilitate communication between the server and the third party.

## Component view

The following diagram shows the components which are the actual realization of the presented services of the application layer. As already mentioned, interfaces are always used (when possible) to increase independency, freedom of implementation and maintenance ease. All the components are seen as “black boxes”, which means we do not inspect their implementation, but we are concerned only about the services they must expose.

Besides the services, since the main architectural pattern applied is the MVC, it’s presented here also a basic representation of the Model, whose focus is on the main classes of our application and their relations (avoiding irrelevant attributes which may change over time). All the offered services have access to the Model: this chapter includes an object diagram to show the relations between the Model and the services.

To be more explicit,

* DataReadingService is responsible for reading data send by the device. For every data it creates an object of class Data with the corresponding value. When data transfer is possible (e.g. there is WiFi connection), it sends all the data to the the application’s Model;
* UserStatusService allows the user to shut down temporarely *AutomatedSOS;*
* ThresholdsService allows to compare data with some preestablished thresholds, to detect emergencies, and contains an internal state to prevent from comparing data when UserStatus asks for it;
* MalfunctionService is responsible for checking wheter the sensor is working properly and, if not so, alerting an emergency number, as stated in the RASD; furthermore, it creates objects of class Malfunction;
* RequestManager, the only component on the web server tier, handles the requests creating an object of class Request and sending it to the application tier:
* LoginService allows to login the application (for all the clients);
* SigninService allows to sign in the application (for all clients);
* RequestService is the component that handles requests: this implies asking for permission in case of individual requests and anonymizing data in case of group requests;
* DataCollectionService “consumes” Data objects destroying. It creates tuples and inserts them in the database;
* EmergencyService is the main component for *AutomatedSOS:* it allows to alert all the third parties (creating objects of class Alert) in case of emergencies and mark the alerts as “handling”;
* ThresholdsCalculationService contains the algorithms to calculate the thresholds for each type of user, depending of their history;
* DataElaborationService is responsible for calling the DataBaseService, for elaborate and manipulate data;
* NewDataSubscribtionService allows third clients to ask for future data and get periodic updates when some changes in Database happen;
* Router dispatches and forwards messages

The main reason for splitting the application layer and running some services on the user’s device is to avoid constant and too frequent communication of data between client tier and application tier, since data are detected every 500 ms and the RASD states strict requirements about how fast the system should react to emergencies. The presented architecture tries to limit as much as possible immediate data transfers and gives to the system freedom to do the back up when there are the best conditions to perform it.

A synchronization and consistency issue could be raised, especially for those objects of Model (e.g. Data, Request) which are used by different tiers. To handle it, classes of the model which can change (i.e. there are modifiers for that classes) are located only on the application layer, and their objects can be accessed only by remote references. In no case some local copies are created. On the other hand, the Model of the web server tier and the client device tier includes objects which are can only be created and destroyed (when “consumed”), and for all classes there is a unique creator service and a unique consumer service. The Data Tier does not contain the Model.

## Deployment view

This chapter focuses on giving more details on the mentioned physical tiers with this diagram. The thick client and the fourth tiers to interact with the third party

## Runtime view

## Component interfaces

The following diagram presents all the interfaces of the application layer.

## Selected architectural styles and patterns

* [devo parlare delle SOA???]
* MVC: the most important architectural pattern applied. It consists in dividing the three software components of user interface (View); core functionality and data(Model) and the response to user inputs (Controller).
* A Producer- Consumer pattern between the two services of DataReading and DataCollector: the producer-consumer pattern consists in two components where the first one “produces” some relevant information ( in our case objects of class Data) and puts them in a list in the corresponding User object, while the second one “consumes” these data performing some actions ( in our case, destroying them and inserting the corresponding tuples in the database). This allows the system to postpone the communication with the database and the insertion of data, improving the distribution of workload over time.
* Since the thresholds are located on the user’s device, we apply a Singleton pattern to create them: of course, we need only one object of class Thresholds for each user.
* A State Pattern is applied to ThresholdsService: when CompareData is called, the code executed depends on the state of the service (basically, when the state is “inactive”, there is no execution of code at all). The operation of choosing what to execute is transparent to anyone uses the interface and to the service itself. Changes of State can be triggered by UserStatusService.
* An ObserverPattern is applied to NewDataSubscribtionService: when a user marks a request as “new\_data”, the request is sent to this service instead of RequestService: therefore, NDSS adds this request as an observer to all the users involved in the requests, so it can be notified when some changes happen (i.e, when the DataCollectionService inserts some data in the database) and respond to the RequestManager that updates are ready.
* A Façade Pattern is used to hide all the algorithms developed to calculate the threholds for each type of user. The Façade Pattern consists in providing a simple external interface which hides a huge number of internal interfaces and complex blocks of code and classes. In fact, the provided algorithms can present big complexity to use and can also change frequently when the medical team (which supports their development, see the RASD) gives more hints.
* Client- Server architecture: in this architecture there are two distinct processes that communicate with well-defined interfaces. Communication is started by the client, and proceeds through service invocation and message exchange. In details, the Client side is represented by the *ThirdPartyClient* component, while the Server side is represented by the *RequestManager* component. The only exception to this architecture is when the server pushes to the client un update informing that a new emergency is occurring and there is need of immediate help.
* Three tiers architecture

## Other design decisions

# Requirements Traceability

# Implementation, Integration and Test Plan

## General Presentation of the implementation strategy

The implementation of the *TrackMe* system will be mainly **business oriented.** This means that some business services, which correspond to some user visible features, have been identified and ordered by their relevance for the company, and every service is associated to a **thread**: a thread is a portion of several modules that together provide a business service. Consequently, the order of implementation follows these threads.

The main reason for this approach is that in this way the company could give away early releases of the app, and then enrich the system and provide successive releases. Some of these so-called business services are presented in the runtime view, where the real use of the system by the clients is simulated.

Moreover, in this way there is more probability to find out some functions that can be provided by the same blocks of code, so reuse of code is facilitated.

## Integration and testing strategy

As a general rule, Unit Testing of partial implemented components is always made after implementing the parts which are needed to satisfy the thread taken into account: because of that, it is never mentioned hereunder.

This implies the following procedure: before performing the integration test between all parts of all components involved in providing the business service, some stubs and drivers are developed to test first all the parts independently each other, and then the integration test is made. Consequently, the strategy chosen for integration test is the bottom-up one.

In these chapters we present the process to execute for implementing, integrating and testing the entire system, following the mentioned threads.

### First step: Data Storing

The main business service of *TrackMe* is to read data and store them, since health values are the main resource of the company. Because of that, the thread which includes *DataReadingService* and *DataCollectionService* and allows to detect data and insert them in the DB is the first to be implemented, together with the Database.

Since there are no parts of *DataCollectionService* which don’t have to be implemented for *DataStoring,* a Unit Test of the entire service is done here.

A first integration test is needed between the two services, and, most of all, the second one and the Database. Since the three parts into account are deployed on three different tiers, the integration test helps to verify immediately the correct communication between the tiers. In fact, detecting later some network issues could be hard and expensive to correct, since communication affects the entire system.

### Second step: Requests Handling

Request Handling is, at this point of implementation, necessary to reach a basic (thus non-complete) functioning of *Data4Help.* This thread includes *RequestManager*, *RequestService*, *DataElaborationService*, and again the Database.

Again, an integration test between components on three different tiers is executed: it allows to test communication and validate the architecture proposed, besides the simple integration of single components. After this step, the database, which represents the data layer should be completely tested and integrated with the system.

### Third step: *Data4Help* load test

It could be useful to make, at this point, a load test to verify that the system is able to handle quickly requests and data messages when they increase in number, since there could be performance bottlenecks.

Now the system is ready for a first release.

### Fourth step: EmergencyHandling

We need to add now the main thread for *AutomatedSOS*: *DataReadingService, ThresholdsService, EmergencyService and ThresholdCalculationService* are involved in this step. With the exception of *ThresholdsService*, where the state pattern is not implemented yet, they are implemented in every part, so Unit Testing of all them is made now.

Integration test is then made to validate the thread.

### Fifth step: Enrich functionalities

To enrich functionalities and fulfil the RASD requirements, we can add now MalfunctionService and the thread which allow the user to use statistical tools to elaborate Data, which relies on *RequestManager*, *RequestService*, *DataElaborationService*. Unit Testing of all components is possible now, as well as the usual integration test.

### Sixth step: Addition features

*UserStatusService* and *NewDataSubscribtionService* and left as last because they have the worst cost-benefit ratio: the first one interacts with *ThresholdsService*, requiring even some changes in this component, and the second one interacts with the Model, but none is crucial for the company, so waiting for these components to be implemented and tested could lead to a loss of time. Unit Testing for *ThresholdsService* is possible now.

### Seventh step: AutomatedSOS load test

A load test of all the components which support *AutomatedSOS* is done here to verify the correct behaviour in case of a huge number of emergencies to dealt at the same time.

### Eighth step: System test

A complete system test, when all the components are developed, and all the necessary integration and unit test are successfully completed, is the final step before the release to the clients of all the application.

## SignInService and LogInService

Since these services do not affect the others in any way and concern important issues such as security of data and privacy, their development, integration and testing shold be done independently from the others, i.e. in parallel to the presented steps: this choice implies allocation of exclusive resources and more time to perform the necessary unit tests and integration tests with the database and in general the whole system. Eventually they are included in the load tests and in the final system test. Because of

## The Model

Since the model is shared by all components

# Effort Spent