# Trackme DD

## DESIGN DOCUMENT (DD)

## Release date: 10/12/18

Version 0.1

Table of content

[Trackme DD 1](#_Toc531013661)

[DESIGN DOCUMENT (DD) 1](#_Toc531013662)

[1. Introduction 2](#_Toc531013663)

[1.1. Purpose 2](#_Toc531013664)

[1.2. Scope 2](#_Toc531013665)

[1.3. Definitions, Acronyms, Abbrevations 2](#_Toc531013666)

[1.4. Revision History 2](#_Toc531013667)

[1.5. Reference Documents 2](#_Toc531013668)

[1.6. Document Structure 2](#_Toc531013669)

[2. Architectural Design 2](#_Toc531013670)

[2.1. Overview 2](#_Toc531013671)

[2.2. Component view 2](#_Toc531013672)

[2.3. Deployment view 2](#_Toc531013673)

[2.4. Runtime view 2](#_Toc531013674)

[2.5. Component interfaces 2](#_Toc531013675)

[2.6. Selected architectural styles and patterns 2](#_Toc531013676)

[2.7. Other design decisions 2](#_Toc531013677)

[3. User Interface Design 2](#_Toc531013678)

[4. Requirements Traceability 2](#_Toc531013679)

[5. Implementation, Integration and Test Plan 2](#_Toc531013680)

[6. Effort Spent 2](#_Toc531013681)

# 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;

## 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

All the system is based on the SOA: every functionality to provide is associated to a 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.

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.

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 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. As already mentioned, interfaces are always used (when possible) to increase independency, freedom of implementation and maintenance ease.

Besides the services, since the main architectural pattern applied is the MVC, it’s presented here 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 document includes an object diagram to show the relations between the Model and the services.

* 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);
* 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.
* 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

## Runtime view

## Component interfaces

## Selected architectural styles and patterns

* 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, while the second one “consumes” these data performing some actions ( in our case, destroying them and inserting the corresponding tuples in the database).
* 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

## Other design decisions

# Requirements Traceability

# Implementation, Integration and Test Plan

# Effort Spent