Politecnico di Milano

Master's Degree in
Computer Science and Engineering

Software Engineering 2

TrackMe
Design Document

Authors
Alberto Archetti
Fabio Carminati

Reference professor
Elisabetta Di Nitto

v.0 - December 7, 2018

Contents

1 Introduction

1.1 Purpose

TrackMe wants to develop a software-based service that allows individual users to collect health data, called `Data4Help`. This data can be retrived from the system and visualized according to different filters by a user interface.
The system allows third parties registration. Third parties can request access to users'collected data in two ways:

Single user data After a third party makes a request to the system for a single user data sharing, by providing user's fiscal code, the system asks the user for authorization; if positively provided, the third party is granted access to the user's data

Amonymous group data Third parties can be interested in big amounts of data, but not in who are the people providing it; the system, once the request is sent by the third party, checks if the data can be effectively anonymized (it must find at least 1000 people that can provide data matching the third party request's filters) and, if positively evaluated, grants access to the anonymized data to the third party

Third parties can subscribe to new data and receive it as soon as it is collected by the system.

Another service that TrackMe wants to develop is `AutomatedSOS`, built on `Data4Help`. This service analyzes users'data and calls a SOS whenever data exceedes the basic health parameters. For this particular purpose, system performance will be a critical aspect to be taken into account, because even the slightest delay matters in critical health situations.

We will list the project **goals**, described in the RASD document:

`G.U1` Users can collect, store and manage their health data

`G.U2` Users can choose to have their health monitored; if their health is critical, an ambulance will be dispatched

`G.T1` Third parties can ask single users for their health data sharing

`G.T2` Third parties can request access to anonymized data that comes from groups of people

`G.T3` Third parties can subscribe to new data and receive it as soon as it is produced

1.2 Scope

1.2.1 World

Our *world* is composed of two main types of actors: *users* and *third parties*. Users are interested in monitoring their health parameters and third parties are interested in developing services or researches that exploit data gathered from the users. `Data4Help` is the service that acts as a bridge between these actors'needs.
Phenomena that occur in the *world* and are related to our application domain are

- physical conditions of the users
- third parties'projects, researches and interests
- ambulances dispatched by the SOS system

These phenomena exist in the *world*, but cannot be observerd directly by our system.

1.2.2 Shared phenomena

In order to communicate with the *world*, our system needs to share some aspects with it. We will list the aspects controlled by the world, but observable by the machine:

`S.1` physical parameters of the users, gathered through sensors on wereable devices

`S.2` third parties requests to the system for the data they need

`S.3` users'location, acquired through GPS signals

On the other hand, the aspects that occur in the machine, but are observable by the world are

`S.4` interfaces that organize the gathered data that can be filtered according to time or type of data

`S.5` messages for the SOS system, that are sent in case of critical health of a user

`S.6` payment requests

1.3 Definitions

Data Quantitative variables concerning health parameters

- **Aggregate data** See *DataSet*
- **Anonymous data** *data entry* that doesn't contain information about the user from which it was produced; a *data set* is said to be anonymized if it contains only anonymous *data entries* and its cardinality is greater or equal than 1000
- **Data entry** Tuple that corresponds to the user's parameters in a particular moment
- **Data set** Set of *data entries*; depending on the context, it can identify a set of entries all belonging to a single user or or a set of anonymous entries belonging to more that 1000 users; a *data set*, among all *data* that the system is storing, can be identified and constructed according to the filters of a third party request

Request Third parties can ask the system for some data sharing through requests; requests are encoded through filling a form; the system, provided that the request is satisfiable, grants the third party access to the requested data

Third party Actor interested in collecting data from a single user or from an anonymous group of users

Threshold Numerical values related to a particular health parameter; they act as boundaries between the domain of critical health status and normal health status

User Actor interested in his/her health data collecting and managing; a user can also be interested in automating SOS calls whenever his health status becomes critical

Some of these definitions may already be present and further explained in the RASD document.

1.4 Acronyms and abbreviations

API Application Programming Interface

DBMS Database Management System

Data Whenever the context refers to generic groups of *data entries*, the terms *data* and *data set* are interchangeable

REST REpresentational State Transfer

System Software product that TrackMe wants to develop; can be interchanged with *S2B*

S2B Software To Be

1.5 Revision history

Date	Version	Log
–	`v.0`	DD first draft

1.6 Document structure

This document describes architecture and design of `Data4Help` and `AutomatedSOS` systems. The description will start with a top-down approach, in order to make the reader familiar with the overall structure; a bottom-up approach will then be adopted, in order to describe components in a isolated way. This document is devided in

- Section 1 is a brief introduction on the project to be developed in order to make this document self-contained
- Section 2 describes the high-level architecture (high-level components, their interation, runtime views and architextural decisions)
- Section 3 provides an overview on how the user interface will look like
- Section 4 contains mapping between software requirements, described in the RASD document, and design elements
- Section 5 identifies the order in which subcomponents will be implemented, integrated and tested document

2 Architectural Design

2.1 Overview

The architecture is a three-tier architecture (Figure 1): it allows to separate clearly *presentation* layer, *business* layer and *data* layer. These sets of components will communicate through defined interfaces and will be treated as black boxes during their interaction. This modular approach enhances modifiability and extensibility.

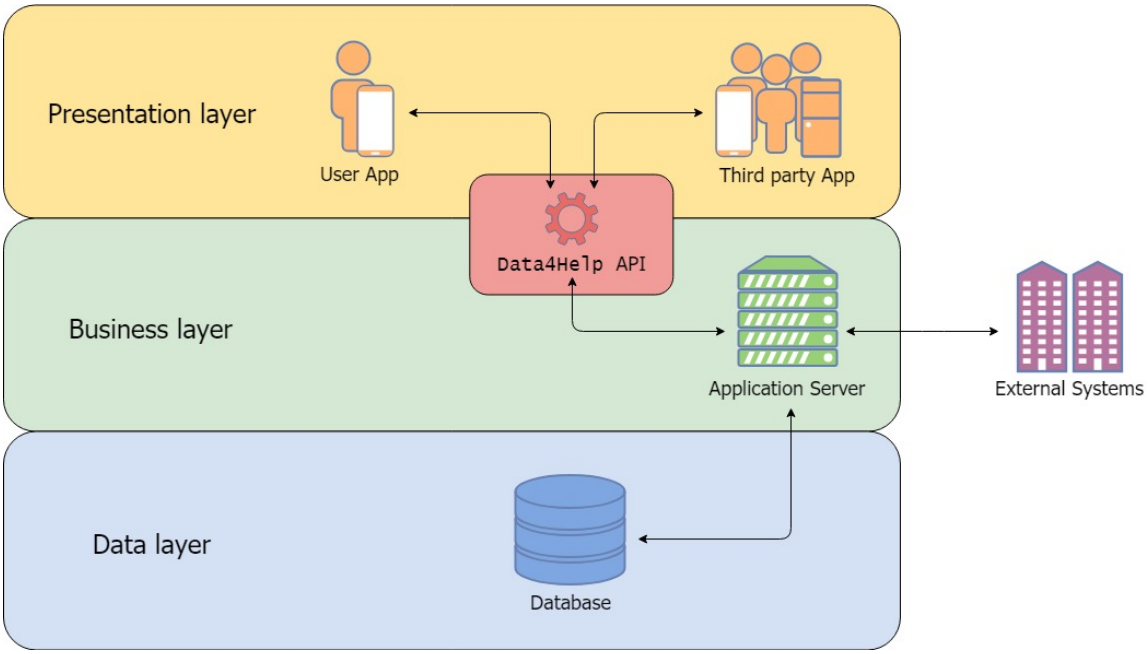

Figure 1: Overall architecture of the system

The main components of the system are

App Application installed on users'devices that communicates with the system; its purpose is to show data to the user and forward his/her requests to the Application Server; we will focus on the smartphone app for Andriod or iOS systems, as it is the main front-end application that our clients need

Application Server Back-end component on which the logic of the application takes place; it elaborates the requests it receives and interacts with external services and the data layer; we will focus mainly on this component, as it shall handle all the information dispatching from different layers

Database Component responsible for data storage; it shall grant ACID properties (Atomicity, Consistency, Isolation and Durability) and shall provide a management service that handles query parallelization and optimization, as data access policies from different accounts

External Systems Systems that interact with `Data4Help` or `AutomatedSOS`; they handle functionalities not internally developed in the system, such as payment handling and ambulance dispatching

2.2 Component view

In this section we will analyze every high-level component in terms of its subcomponents and provide the main interface interaction between different components (Figure 3). For details on component interfaces see Section 2.5.

2.2.1 App

The application component is the front-end of the system. Our clients will interact with the system through the front end. We will provide

- a smartphone application, capable of exploiting all of the system functionalities: it shall render data, provide forms for the clients (users and third parties) and communicate with the Application Server
- an API that allows more experienced users or other developers to automate communication with our system; the API is particularly useful when third parties need to analyze huge quantities of data that a smartphone graphical interface cannot render
- a command line tool that automates API calls for third parties that don't want to develop a different program to interact with our system; the command line tool will be written in Java, in order to favor the compatibility with most operating systems

It is important to note that the smartphone application exploits the API for communication with the Application Server. Every `Data4Help` or `AutomatedSOS` service can be required by API communication.

2.2.2 Application Server

The Application Server holds the application logic. It is the only component of the *business* layer, but it is the most crucial component of the system. Its role is to coordinate the information flow between the user layer and the data layer and to incorporate external systems'services.

In the architecture the Application Server is the only link to the database. External systems or clients cannot directly access persistent data of our system.

The Application Server is also the only link to the *presentation* layer, as the Application Server coordinates the user-external system interation.

Subcomponents of the Application Server are

`AccountManager` This module handles creation, authentication and management of users and third parties'accounts; before exploiting our system's functionalities users and third parties need to be authenticated by this module after providing their credentials

`DataCollector` This module communicates with users'application and periodically receives data entries, as soon as they're collected by users'wearables

`EmergencyDetector` This module is in charge of automatically analyze data entries inserted in the system if their owner subscribed to `AutomatedSOS`; it is separated from the `DataCollector` because emergency detection can be exploited in many ways, depending on the medical literature on the topic; this feature should be independant and isolated from the rest of the architecture

`EmergencyDispatcher` This module builds emergency messages and forwards them to the SOS system

`FilterManager` This module composes filter constraints on data entries that can be fetched from the database

`PaymentGateway` This module is in charge of communicating with the external payment system is order to process payments between third parties and TrackMe

`RequestManager` This module is in charge of composing, verifying and elaborating third parties'requests, both of single user type and anonymous group type

`SetBuilder` This module generates data set oriented queries for the database; queries can be accepted or declined by the database, depending on the account permissions concerning data entries access

2.2.3 Database

The database is the only component of the *data* layer. Queries are managed by a DBMS that optimizes and elaborates them in parallel. Data stored in the database is persistent and shall not be lost due to external factors. The database service will not be directly developed by us, but will be bought from the existing ones. In Figure 2 we reported the Entity-Relashionsip diagram for the data stored in the database.

The *data* layer is only accessible from the Application Server. It won't implement any application logic, except from DBMS functionalities: it will just respond to queries and passively store data.

An important factor for `Data4Help` is the data access policy: Data Entries should be available only to the users that produced them, when inserted in the database. If a Data Set is shared to a third party, that third party shall be allowed to retrive Data Entries that belong to that Data Set from the database. Therefore the access policy shall be dinamic and shall consider `Data4Help` different accounts.

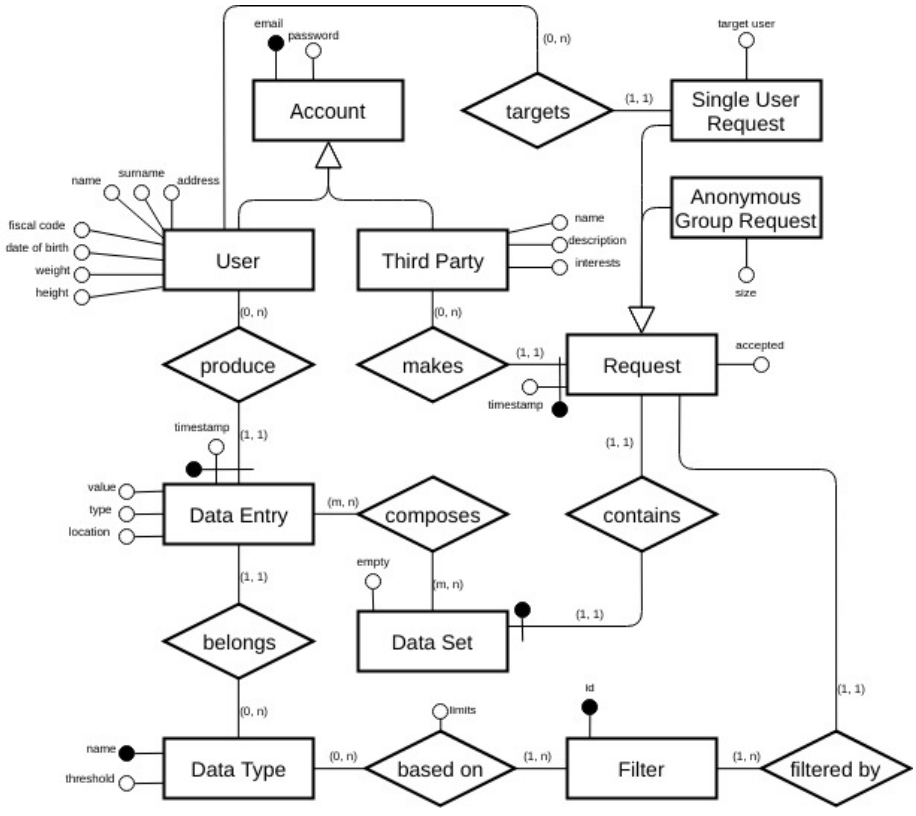

Figure 2: Entity-Relationship diagram for the database

2.2.4 External Systems

In this section we will present the main external systems that interact with the Application Server.

`Data4Help` relies on an external payment handler. The Application Server, once has composed a third party request, evaluates its price and asks third party for payment, by exploiting the external payment handler service. The service manages the effective payment from the third party to TrackMe and signals errors occurred during the procedure.

`AutomatedSOS` relies on an external SOS system. The SOS system dispatches ambulances and handles health emergencies by accepting automated calls. `AutomatedSOS`, on the Application Server, detects health dangers as soon as they're collected from the front-end components forwards an emergency message to the SOS system.

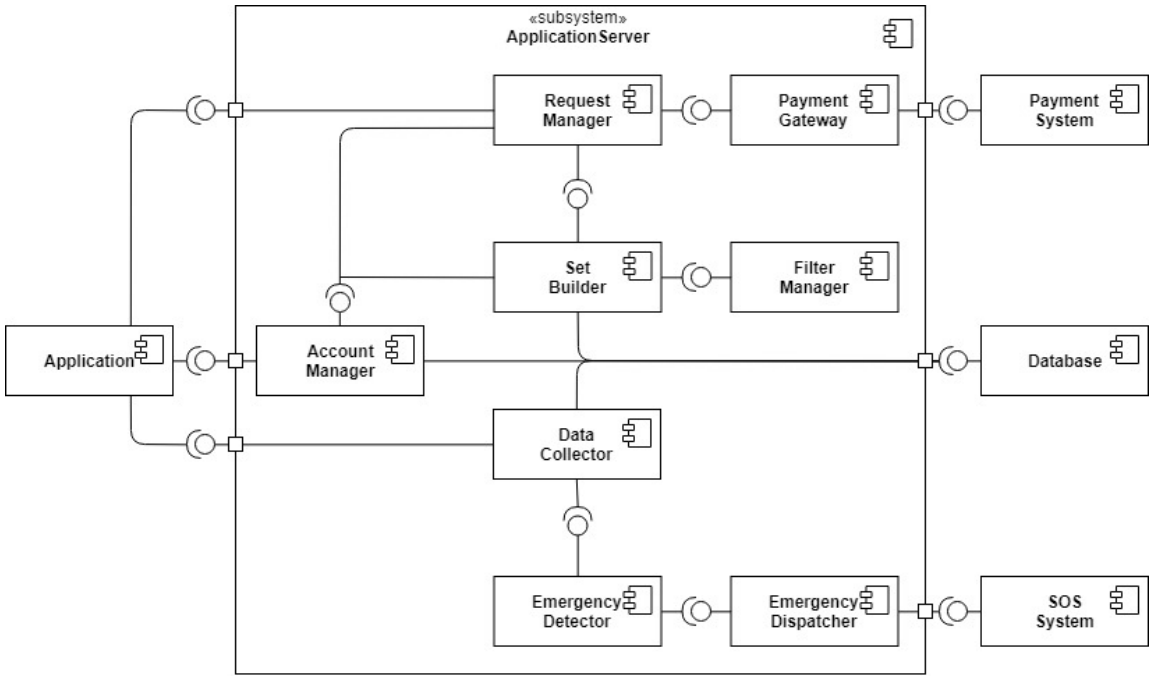

Figure 3: UML component diagram of `Data4Help` and `AutomatedSOS`: component interaction is described in Section 2.3 and interfaces are described in Section 2.5 and Section 2.6

2.3 Deployment view

We decided to deploy our system exploiting the services provided by Amazon AWS (Figure 4). Amazon AWS hides most of the back-end complexity in terms of scalability, reliability and fault tolerance of the system. It automatically handles server calls in a distributed way, by just providing the code we want to run [5]. The pros of this deployment choice may be summarized in

- automatic load balancing of client-server communication and automatic scaling of the system
- no need to purchase physical servers, as Amazon AWS provides software services that are entirely exploited on their machines
- Amazon AWS provides a payment policy proportional to the effetive resource usage required by our system: there is no need for physical resource planning, as the allocated resources and the payment amount adapt automatically to the usage we require
- coding, testing and integration phases are handled in a professional environment, the Amazon AWS platform, that provides easy-to-use and powerful instruments that allow developers and testers to work faster and more efficiently, by focusing only on system-related tasks and not worrying about service integration issues

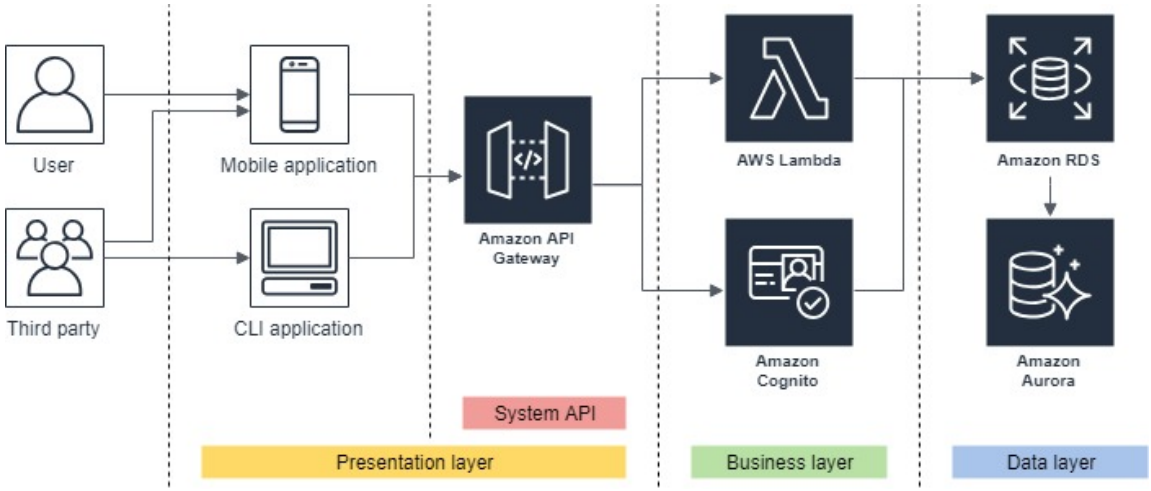

Figure 4: Mapping of the architectural layers on the Amazon AWS services

It is important to note that in the other parts of this document we didn't treat the Application Server as a physical node, but more as a logical node. In fact, the Application Server functionalities will be deployed by Lambda functions, that have no concept of node or server on which business logic is handled. Nevertheless the set of defined Lambdas has the same external behaviour of a stateless server that waits for API calls and performs tasks between its logical components. By hiding the deployment details of the Application Server, we intended to isolate this section from the rest of the document, in order to ease future changes or additions.

In the deployment view (Figure 5) we find:

`Mobile device` Runs the mobile Application of the presentation layer; the Application is compatible with Android and iOS operating systems, as they're the most common mobile operating systems between TrackMe target clients; there are no particular hardware constraints regarding the mobile device, other than having a touch screen

`PC` Runs the CLI Application of the presentation layer; the Application is written in Java, therefore it is compatible with most desktop operating systems

`AWS Lambda` Platform where it is possible to upload application code (written in one of the supported languages such as Node.js, Java, Go, C and Python) which will be run in an environment called Lambda function; every Lambda function is stateless (Section 2.7), meaning that it is completely independant from other functions and can be executed in a distributed and very efficient way; Lambdas autonomously manage memory, CPU usage, network and other resources, so we are responsible only for our code

`Amazon Cognito` Handles user registration, login, and access control quickly and easily; it is basically a suite that handles all the functionalities required to the `AccountManager`, therefore it can be deployed on this service

`Amazon RDS` Amazon Relational Database Service performs all the administration tasks such as database setup, patching and backups: we can focus only on how should the database behaviour be and not how can it be achieved

`Amazon Aurora` Amazon Aurora is a relational database engine used to deploy the data layer of the system; it delivers up to five times the throughput of standard MySQL; Amazon Aurora's storage is fault-tolerant and self-healing, and disk failures are repaired in the background without loss of database availability[1]

[1] `https://aws.amazon.com/rds/aurora/details/`

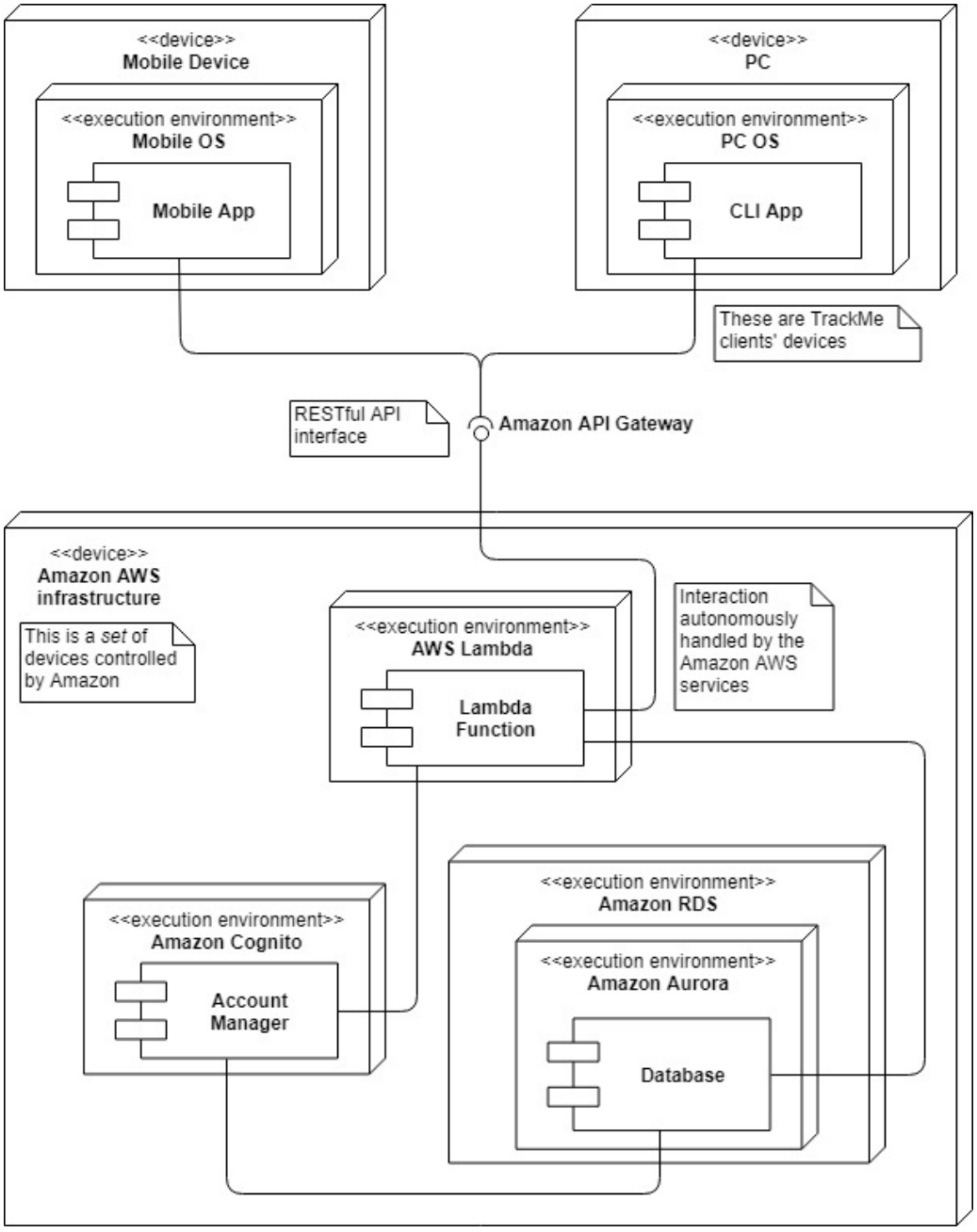

Figure 5: Deployment Diagram

2.4 Runtime view

In this section we will present some sequence diagrams that show the major interaction processes between the system components. All the methods performed between components are described in detail in Section 2.5 and Section 2.6.

2.4.1 Account Registration

The sequence diagram in Figure 6 shows the registration process to the `Data4Help` system from the point of view of an user. The same process applies for third parties, by substituting the `createUserAccount` method with `createThirdPartyAccount`.

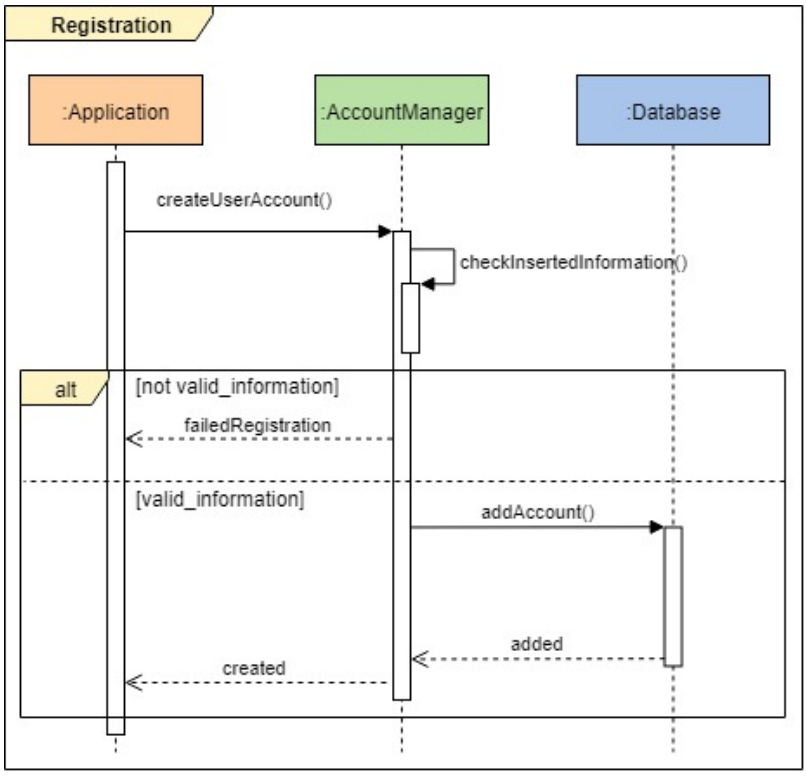

Figure 6: Account registration to `Data4Help` system

2.4.2 Data entry acquisition

The sequence diagram in Figure 7 shows the process of acquiring a new data entry for a user account, without exploiting `AutomatedSOS` service. It shows also the login procedure, that is valid for both users and third parties. This procedure will be omitted in the next sequence diagrams, but it shall be performed correctly by every Application involved in the shown processes, in order to be able to exploit all the system functionalities.

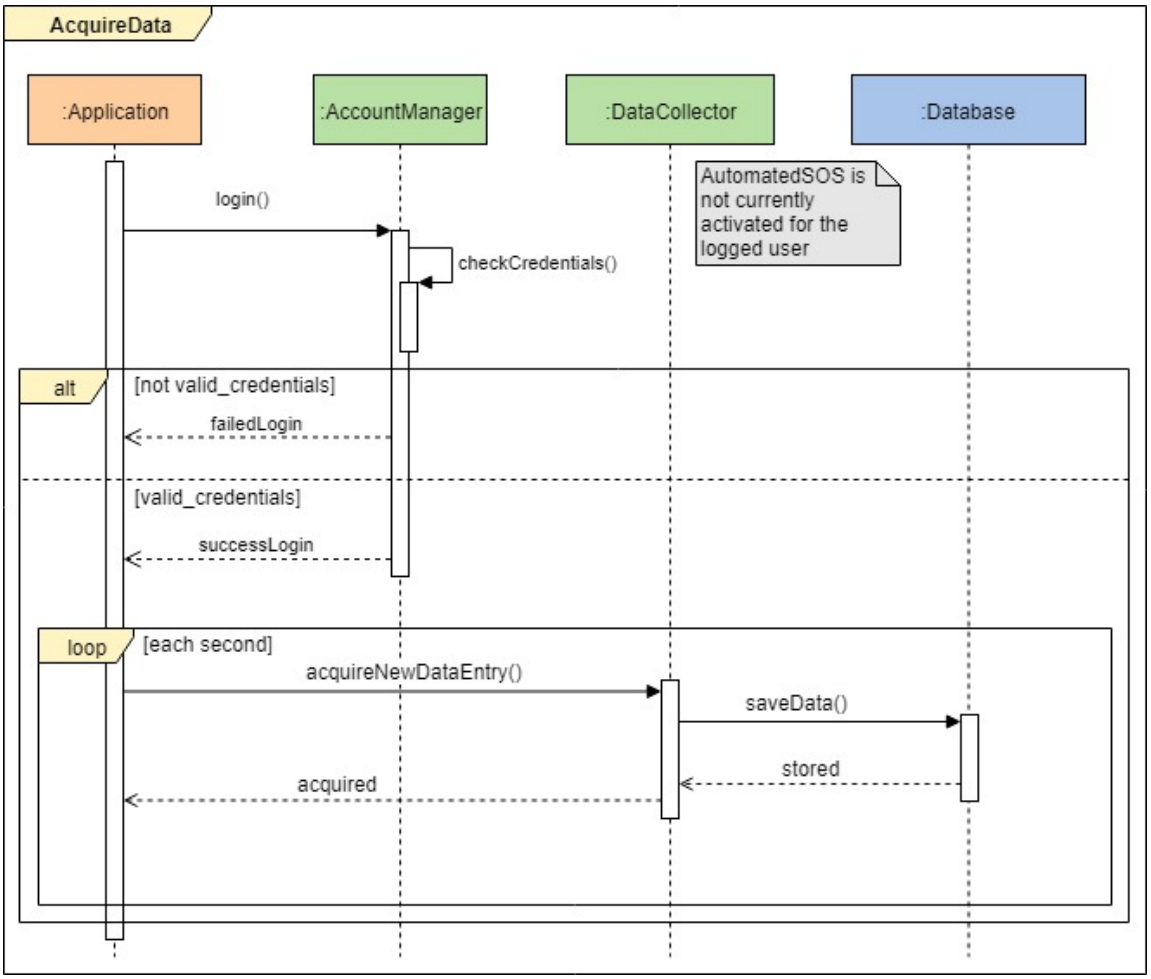

Figure 7: Data entry acquisition loop

2.4.3 Filtering

Figure 8 shows the update of graphical data rendering options (see Figure 3 in Section 3.1.1 of the RASD document).

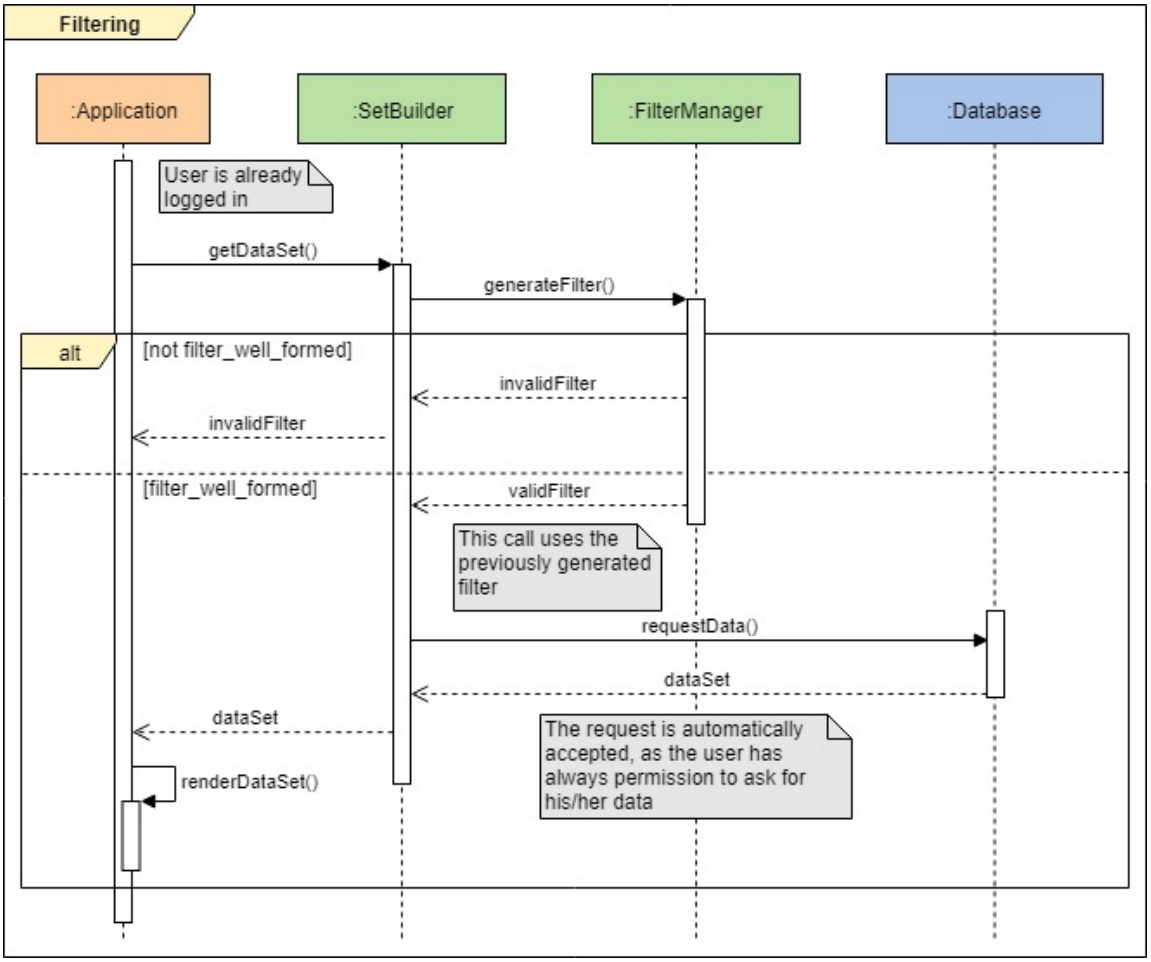

Figure 8: Graphical filters selection

2.4.4 Emergency call

In Figure 9 we see the `AutomatedSOS` activation for a user account that was not already subscribed to it and the SOS calling process.

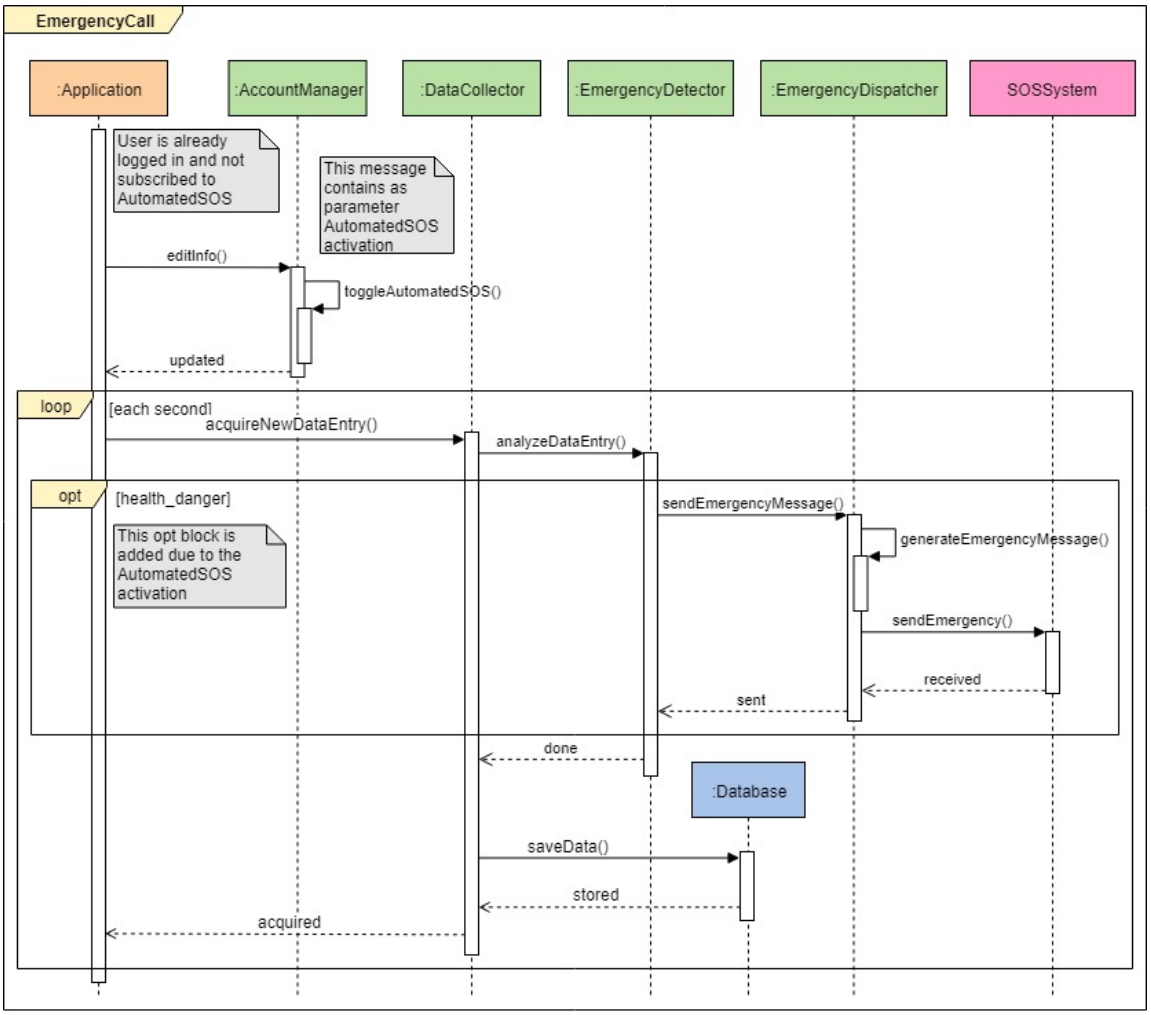

Figure 9: Trigger of `AutomatedSOS` emercency call

2.4.5 Single User request

The sequence diagram in Figure 10 shows a third party that performs a single user request. Although it will depend on future TrackMe policies, we added the payment operation to this process.

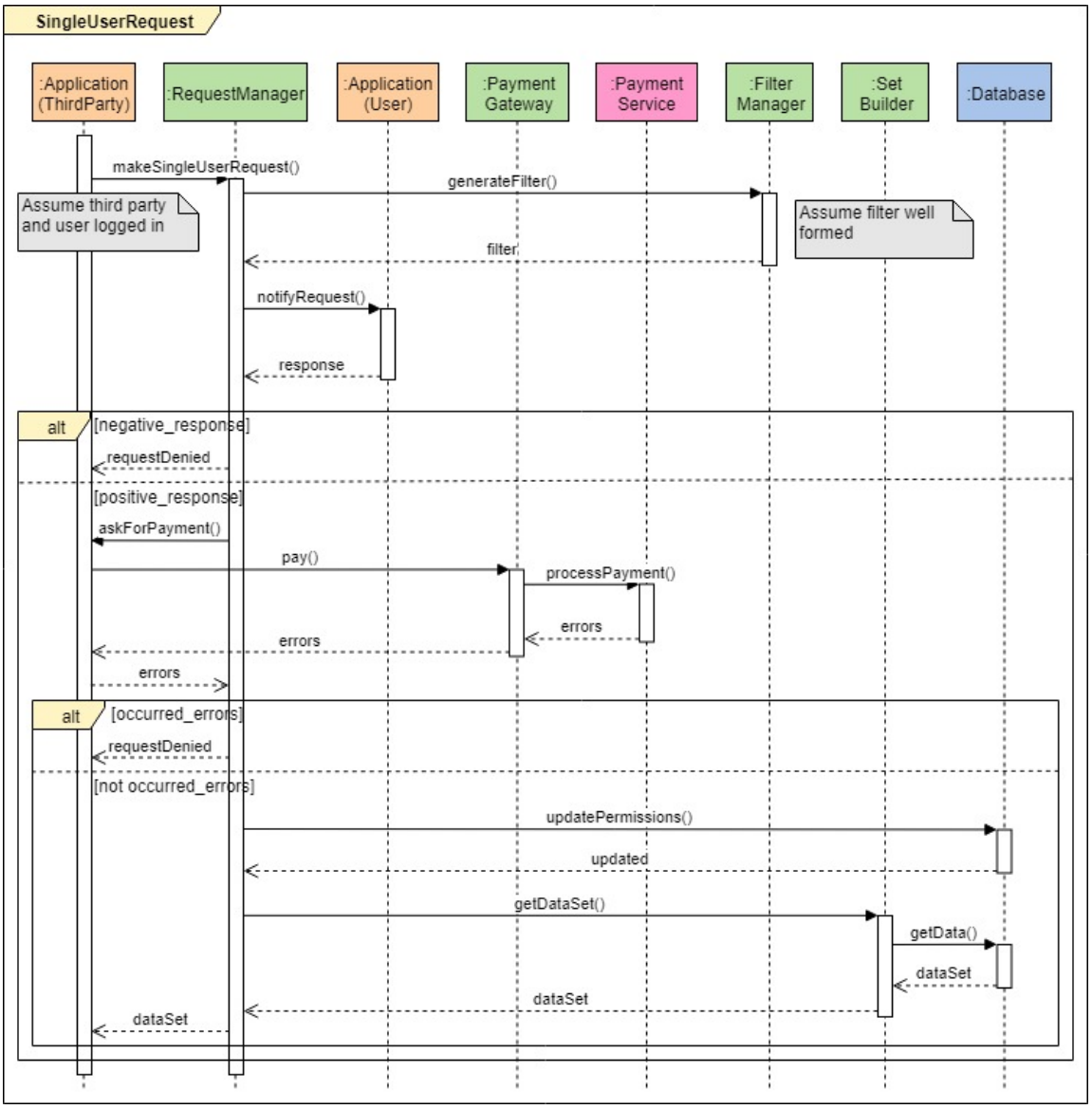

Figure 10: Single user request elaboration process

2.4.6 Anonymous Group request

The sequence diagram in Figure 11 shows a third party that performs an anonymous group request.

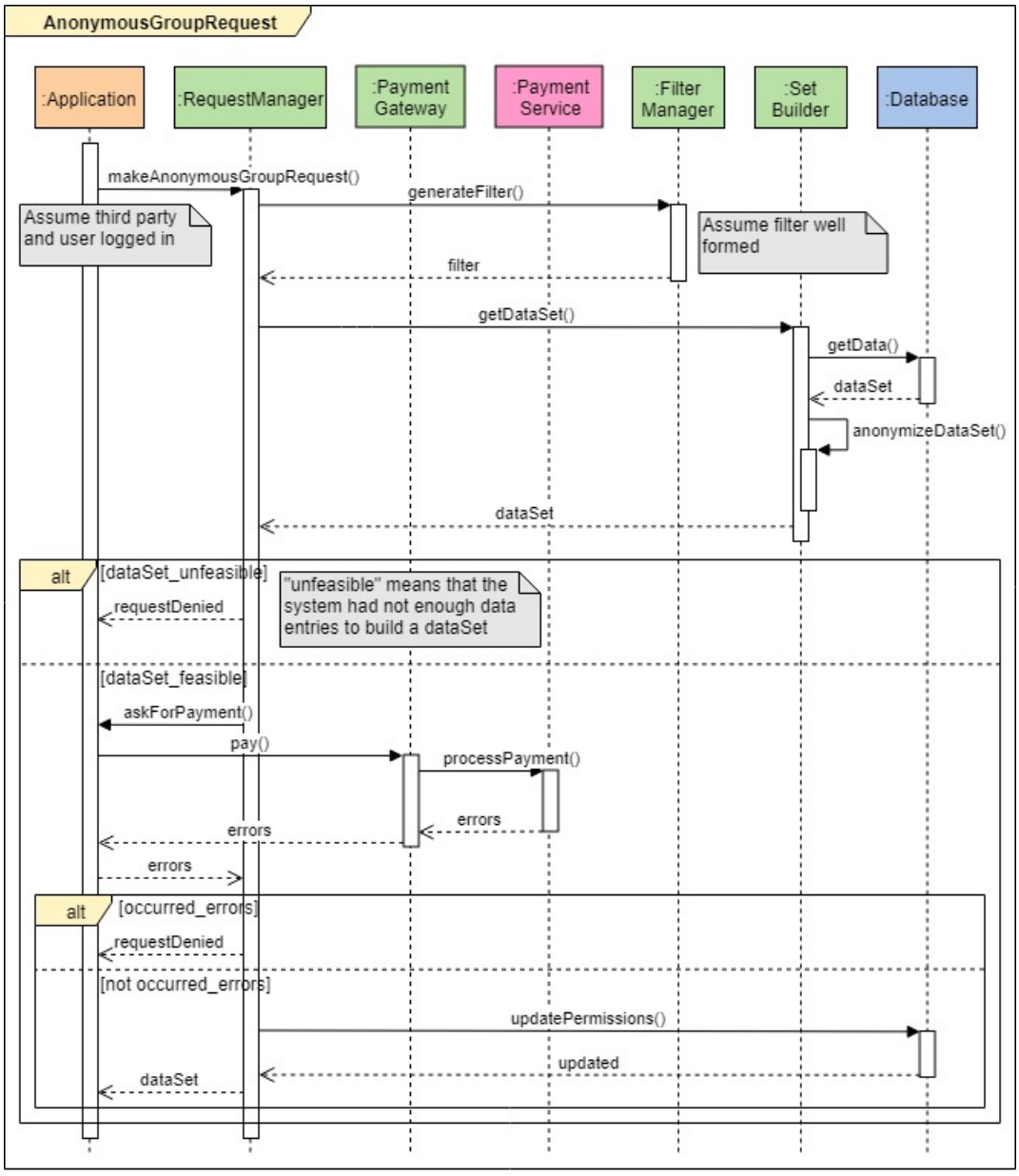

Figure 11: Anonymous group request elaboration process

2.4.7 Update single-user request

The sequence diagram in Figure 12 shows an update process of a single-user request: the request has already been accepted by the target user and has already been payed by the third party. By making the request, the third party subscribed to new data as soon as it will be produced. New data is fetched through the `updateRequest` method.

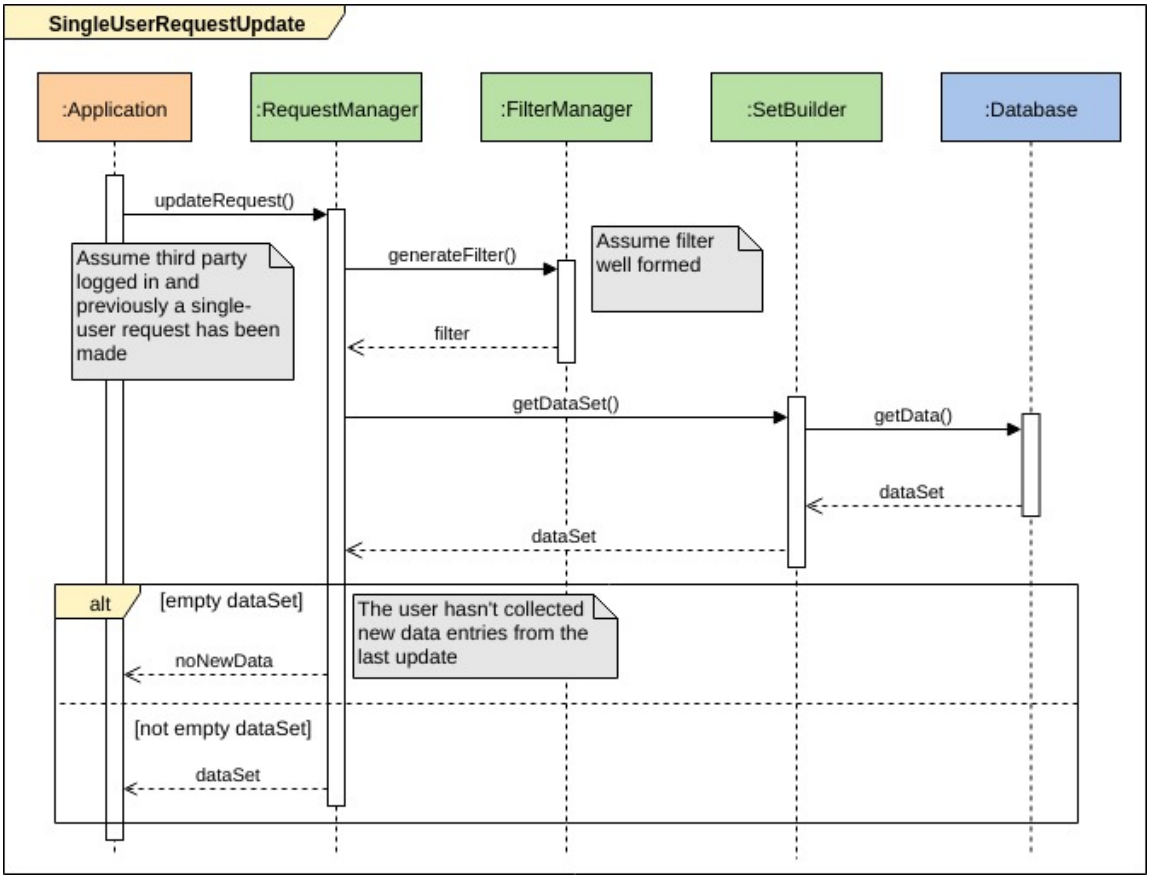

Figure 12: Ask for newly produced data of a single-user request

2.4.8 Update anonymous-group request

The sequence diagram in Figure 13 shows an update process of a anonymous-group request: the request has already been payed by the third party. By making the request, the third party subscribed to new data as soon as it will be produced. New data is fetched through the `updateRequest` method.

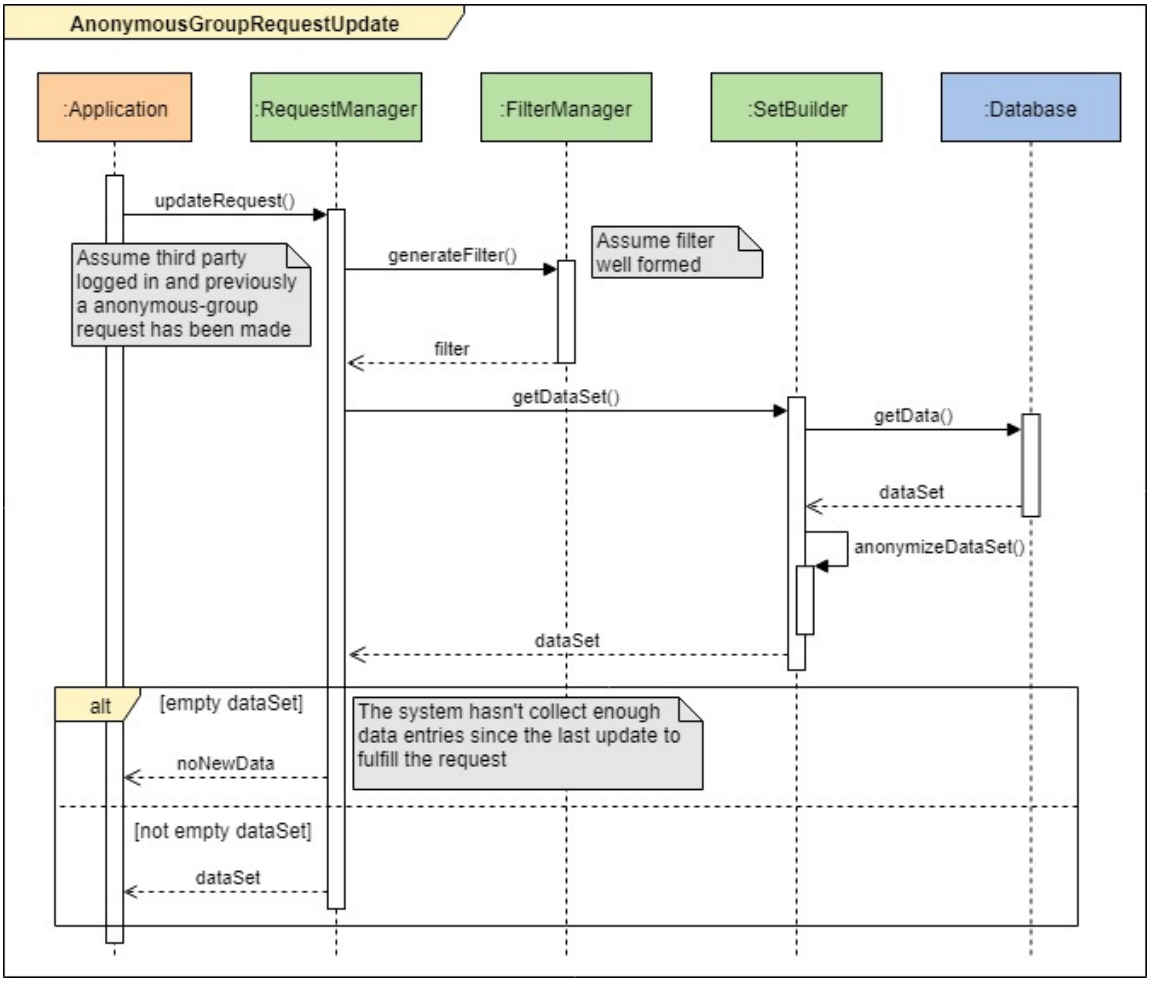

Figure 13: Ask for newly produced data of a anonymous-group request

2.5 Application Server interfaces

In this section we will present the details concerning the interfaces of the subcomponents of Application Server defined in Section 2.2.2. Every component will offer or require some functionalities through interface methods in a way that, once all components are assembled, the Application Server shall not have uncovered functionalities that will not be covered by other systems (Section 2.6). All of this methods has been called at least once in the sequence diagrams of Section 2.4.

2.5.1 `AccountManager`

`createUserAccount` Generates a new user account, provided email, password and other valid information, specified in Section 2.2.1 of RASD document; the return value specifies if the procedure ended correctly or if some incorrect information made it abort

`createThirdPartyAccount` Generates a new third party account, provided email, password and other valid information, specified in Section 2.2.1 of RASD document; the return value specifies if the procedure ended correctly or if some incorrect information made it abort

`login` By providing email and password, a client can login into his/her account and exploit system functionalities[2]; the return value is positive if information provided is correct and negative if there's no account that matches given credentials

`editInfo` Updates the client's profile information with the the new set of information passed as parameter; the return value confirms if the procedure ended correctly; it may even toggle `AutomatedSOS` real time health danger detenction checks on inserted data

2.5.2 `DataCollector`

`acquireNewDataEntry` This method acquires a new data entry collected on the logged user's application; if the logged user is subscribed to `AutomatedSOS`, it forwards the data entry to the `EmergencyDetector`

[2]the *login* procedure is exploited, according to the stateless communication paradigm, by providing the credentials to every request from the client side, in order to guarantee isolation between the requests

2.5.3 `EmergencyDetector`

`analyzeDataEntry` This method analyzes passed data entry and checks whether all parameters are above or below defined thresholds; if some parameters exceed thresholds, it will call the `EmergencyDispatcher` in order to forward the SOS call

2.5.4 `EmergencyDispatcher`

`sendEmergencyMessage` This method builds and forwards an emergency message to the SOS system

2.5.5 `FilterManager`

`generateFilter` This method returns a new filter instance with the passed parameters on type and boundaries of data, that can be used to narrow the data domain of interrogation during database queries; the return value specifies if the filter is well formed or if it cannot be created

2.5.6 `PaymentGateway`

`pay` This method triggers a payment call to the external payment system that returns a positive or negative exit status, depending on the correct execution of the procedure; the return value contains the errors of the payment procedures

2.5.7 `RequestManager`

`makeSingleUserRequest` This method, provided a target user and the proper filters over the data that the third party wants to collect, generates a new single user request by the third party for the target user's data; it returns the data set that contains the requested data if the procedure ended correctly or a notification error, if the user declined the request

`makeAnonymousGroupRequest` This method, provided the proper filters over the data that the third party wants to collect, generates a new anonymous group request; it returns the data set that contains the requested data if the procedure ended correctly or a notification error, if the system wasn't able to properly anonymize the data set

updateRequest This method, provided a single-user or anonymous-group request by a third party that hasn't already been expired[3], checks if there are new data entries that fulfill the request, then returns them or an empty set if there's no new data entry

2.5.8 SetBuilder

getDataSet This method accepts some filters as parameters and forwards a query based on such filters to the database (filters shall be previously elaborated by `FilterManager`); the return value is either the set of data entries fetched from the database subject to the filter's constraints or an error message if the query couldn't be performed (the asking user hasn't access permissions to the selected data entries or in the database there isn't enough data to satisfy the query requirements)

2.6 Other interfaces

In this section we will explore the interface methods of the components that communicate with the Application Server. These components rely on external services and their interfaces may be very complex and may change over time, due to the fact that in the most cases, we are not directly developing them. Therefore our description is at a high level of abstraction. Furthermore, we will focus only on the critical methods that are mandatory for the system in order to communicate correctly with the Application Server.

2.6.1 Application

The following methods are not called by the Application Server, but are useful in order to understand the expected behaviour of the Application when recieving data from the Application Server.

renderDataSet Visually renders a data set on the application screen

showNotification Shows a notification error or a data request notification (in case of user account targeted by a third party single user request) on the application screen

[3]Third parties can subscribe to new data and receive it once has been collected by the system; requests have an expiry date, after which updates cannot be performed

2.6.2 Database

`addAccount` This method adds a new user or third party account to the account set of the system; the return value states if the procedure ended correctly

`requstData` This method analyzes the passed query throug the DBMS service in parallel with other queries and returns the tuples corresponding to the required data, or returns an error if the asking account hasn't the permissions to read the data

`saveData` This method saves the passed data into the persistent memory

`updatePermissions` This method updates permission accesses to data entries of the system, in order to share them between user and third party accounts; the return value states the correct ending of the procedure

2.6.3 External Systems

Payment system and SOS system are the only external systems required by the Application Server. The payment system offers

`processPayment` This method accepts payment data and performs the effetive money movement from third party to TrackMe; it returns the exit status of the process, positive if it ended correctly or negative, alongside an error log, if the process was not succesfull

while the SOS system offers

`sendEmergency` This method accepts emergency messages and dispatches ambulances according to the data contained in the passed emergency message

2.7 Selected architectural sytles and patterns

Model-View-Controller We adopted the MVC pattern because it separates the three most important architectural aspects of the system: data storage, user interfaces and business logic. It allows to develop in parrallel every component, in isolation from the others, granting a less error prone developement and a more extensible architecture to future additions or punctual changes. The MVC pattern can be adopted at varoius architectural levels, as we can find it also in the mobile application.

Three-tier architecture It allows to develop in isolation the presentation layer, the business layer and the data layer. It allows also to deploy on different physical nodes these layers, enhancing the MVC paradigm and the complete isolation of the layer components. The physical separation of nodes enhances reliability and availability of the services, as the application logic and data may be distributed on more than one node, lowering the failure probability and enhancing overall performances.

Client-Server It is the standard model of communication of the World Wide Web.

Thin client Data and logic of the system are handled in the data and business layer respectively. The presentation layer has the only role of showing data to the user and performing requests that will be handled by the other layers (Figure 14). In this way the system can easily adapt to changes and updates in the presentation layer.

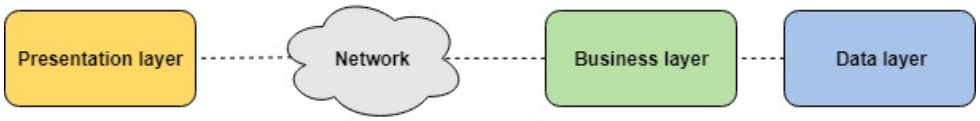

Figure 14: Thin client architecture: logic operations and data storage are not performed by the client nodes, on which the presentation layer reside

RESTful API and stateless communication In order to make the system scalable with the number of requests, it is a good practice to grant stateless communication between client and server. According to the REST paradigm, every HTTP request encapsulates all the data needed to be performed, therefore it is completely independent from other requests. This paradigm allows different requests to be handled from different server nodes, increasing scalability possibilities and load balancing options. The server doesn't hold any session information about the client. Figure 15 and Figure 16 describe a client-server REST interaction.

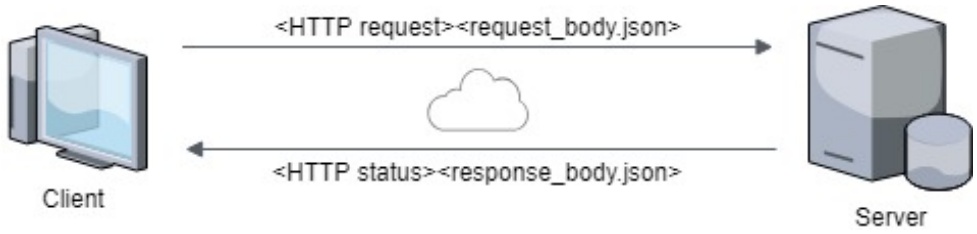

Figure 15: REST communication structure

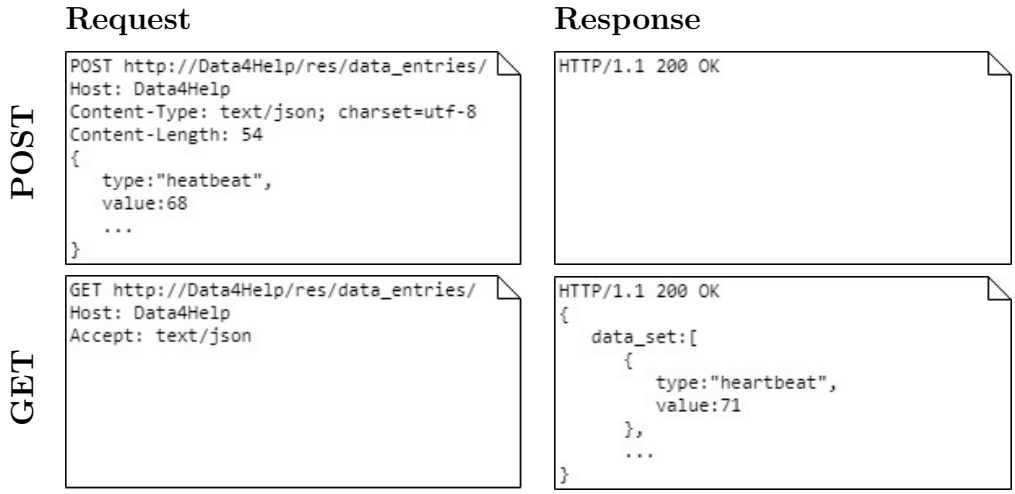

Figure 16: Example of REST communication messages

The purpose of these messages is to illustrate how the REST paradigm works: in the first row we see a client that uses the `POST` command to upload a new data entry in the server while in the second row we see a client or a third party that asks for some data entries with the `GET` command

3 User Interface Design

In this secrion we provide further details on the interface design of the Application. In the RASD document we provided a set of mockups that represent the overall feel of the Application screens. Here we will provide the navigation flow between the screens. The adopted syntax is **bold** for buttons, *italics* for screens (referenced by figure number in the RASD document) and square brackets for conditional branches.

3.1 Screen flow graph for users

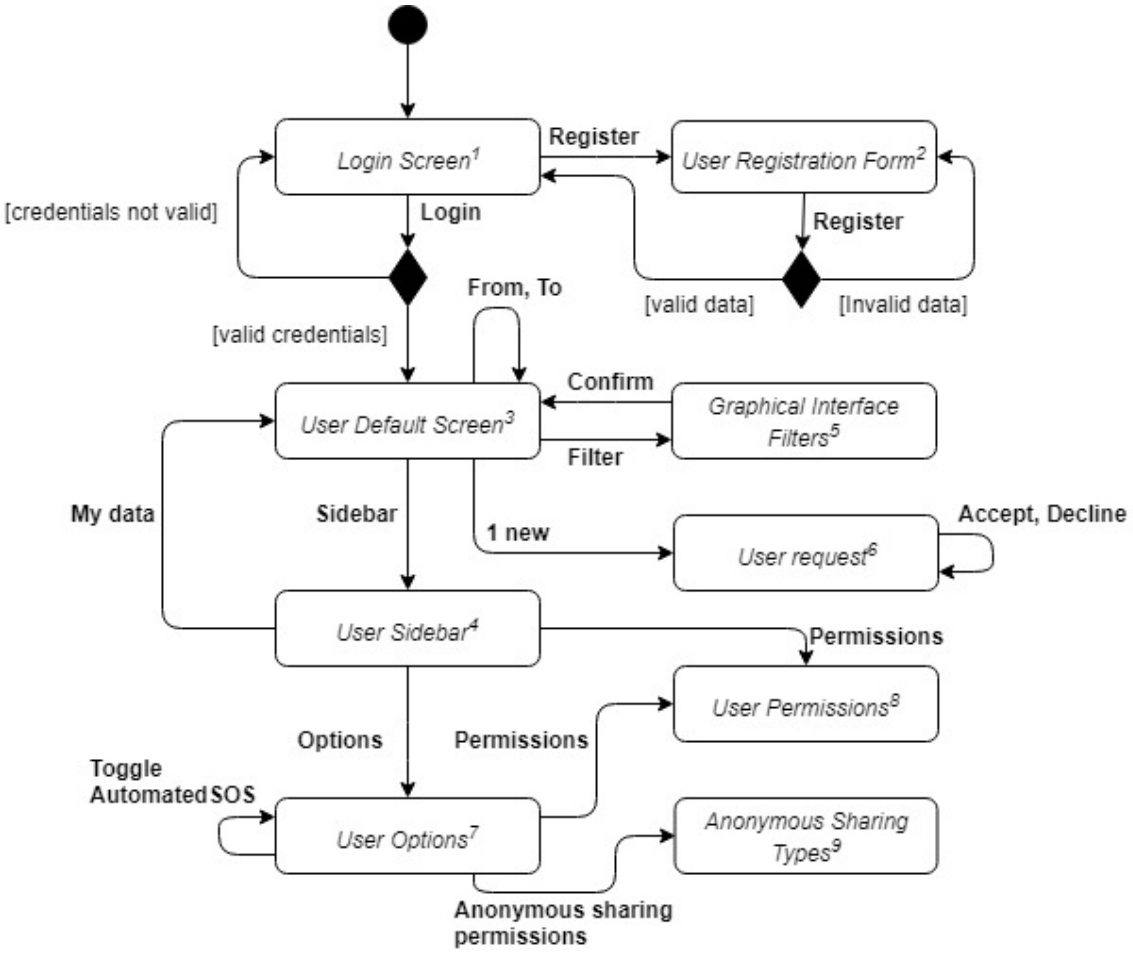

Figure 17: Flow of screens in the user application

3.2 Screen flow graph for third parties

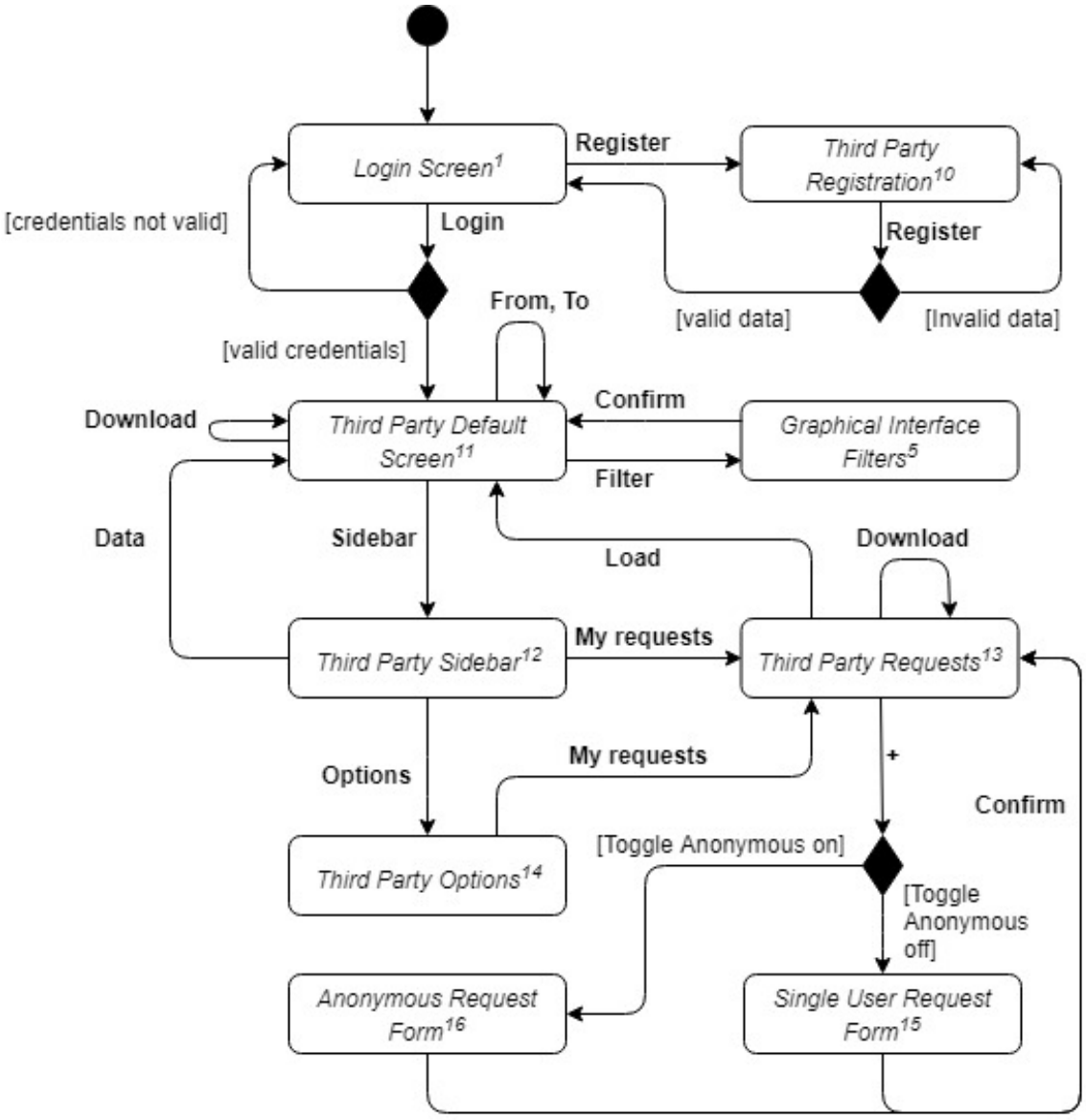

Figure 18: Flow of screens in the third party application

4 Requirements Traceability

In the following tables we present the mapping between the requirements defined in the RASD document and the system components of Section 2.2[4].

4.1 Account handling

Requirement	Description	Components
R.A1	Registration as user	AccountManager, Database
R.A2	Registration as third party	AccountManager, Database
R.A3	Distinguish user and third party accounts	AccountManager
R.A4	Account uniqueness	AccountManager, Database
R.A5	Account login	AccountManager, Database
R.A6	Only authenticated accounts can exploit functionalities	AccountManager, Database

4.2 Data encoding

Requirement	Description	Components
R.D1	Encode and store data	DataCollector, Database
R.D2	Retrive data	SetBuilder, FilterManager, Database
R.D3	Not erase stored data	Database
R.D4	Data property	AccountManager, DataCollector, Database
R.D5	Share data among multiple accounts	AccountManager, Database
R.D6	Compone data sets	SetBuilder, FilterManager, Database

[4]In this section, for readibility purposes, we give a brief description of the requirements. For the full definition see Secrion 3.2 of RASD document

4.3 Interfaces

Requirement	Description	Components
R.I1	Registration form	Application, AccountManager
R.I2	Render data graphically	Application
R.I3	Third parties' request form	Application, RequestManager

4.4 Data sharing requests

Requirement	Description	Components
R.R1	Single user request form	RequestManager
R.R2	Notify requests to users	Application, AccountManager, RequestManager
R.R3	Notify deny to third parties	Application, AccountManager, RequestManager
R.R4	Anonymous group request form	RequestManager
R.R5	Check anonymity	RequestManager, SetBuilder
R.R6	Grant access to anonymous data	RequestManager, Database
R.R7	Grant access to newly produced data	RequestManager, Database

4.5 SOS calls

Requirement	Description	Components
R.S1	Apply to AutomatedSOS	AccountManager
R.S2	Monitor parameters	EmergencyDetector
R.S3	Send emergency message	EmergencyDispatcher, SOSSystem
R.S4	Build emergency message	EmergencyDispatcher

4.6 Payment

Requirement	Description	Components
R.M1	Third parties shall pay	AccountManager, PaymentGateway, PaymentSystem

5 Implementation, Integration and Test Plan

5.1 Dependency graph

In Figure 19 we find the dependency graph between the system components. The dashed arrows state that the starting component needs to use some functionalities of the target component. The arrows have been derived directly from the sequence diagrams of Section 2.4 (every method call corresponds to an arrow between teo compoonents). Arrows can have one of three colors:

- **black**: generic dependency
- **red**: SOS related dependency
- **blue**: payment related dependency

At first, red and blue dependencies, as well as their target components may be excluded from the graph, due to the fact that these dependencies are not relevant to the core functionalities of the system part that shall be developed first. We will divide the integration and testing procedure in high level steps, in order to simlpify the deadline definition.

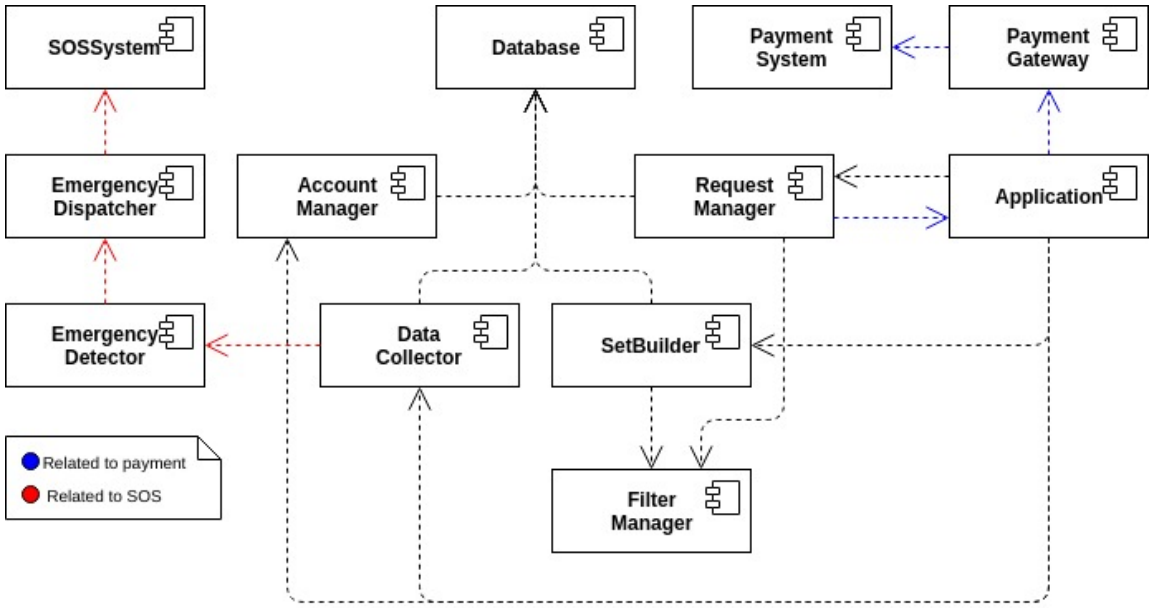

Figure 19: Dependency graph between the components of the system; in red we find SOS related dependencies and in blue payment related dependencies

5.2 Step 0

For the integration and testing procedure we will adopt a top-down approach, starting from the most critical component that has no *use* relation starting from it: the Database.

The Database shall be tested through the stubs and shall respect the following requirements, in order to move to the next integration and testing step:

- save account data provided by the `AccountManager` stub and data entries provided by the `DataCollector` stub
- retrive on request account data or data entries respecting data ownership
- be able to update data ownership on request
- be able to compose data sets according to the queries given by the `SetBuilder` stub

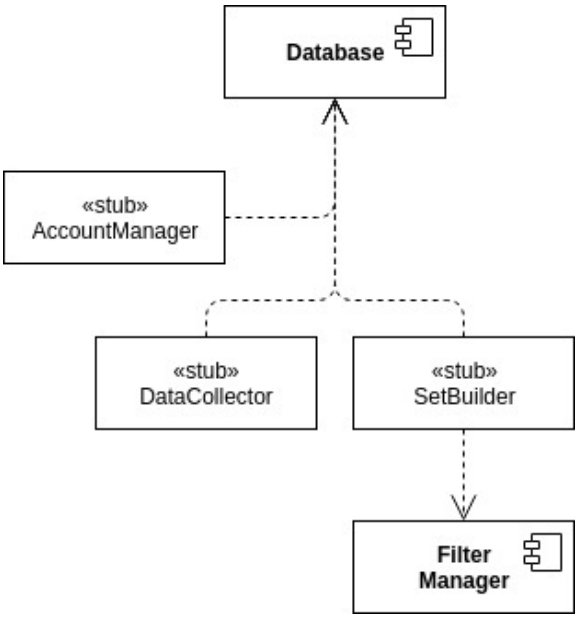

Figure 20: Integration and testing, step 0

In parallel to the Database developement, the `FilterManager` also can be implemented and tested, as it shares no dependency to still unimplemented components. It shall be able to convert parametrized filters to filter objects, checking the integrity and validity of the passed parameters.

5.3 Step 1

In this step some of the crucial components of the Application Server will be added to the system.
Before moving to the next step,

- the `AccountManager` shall be able to create accounts of users and third parties according to the Application stub requests
- the `DataCollector` shall be able to retrive data entries collected by the Application stub and interact with the Database in order to save them on persistent memory with the ownership set to the user that produced them
- the `SetBuilder` shall be able to build queries for the Database according to the `FilterManager` filters and shall be able to anonymize data sets returned by the Database

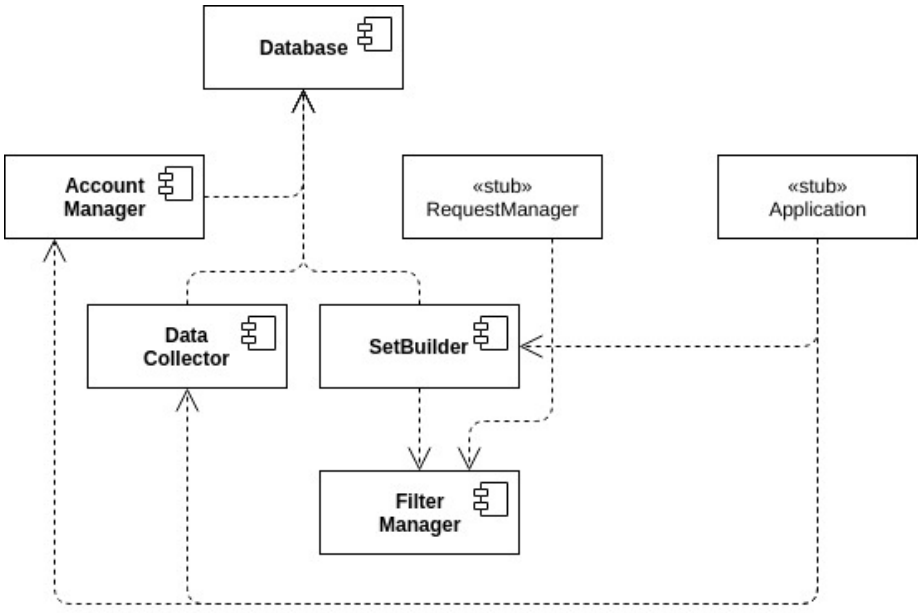

Figure 21: Integration and testing, step 1

At this point the data management features, except from the request related ones, shall be covered on the Application Server: it is able to store and retrive data concerning both accounts and data entries and is able to manipulate it by building filtered sets and anonymizing them.

5.4 Step 2

In this step the `RequestManager` will be added to the Application Server. It shall

- generate single-user or anonymous-group requests from the request form instances provided by the Application stub
- perform the operations to request the data set to the other components and return the data set or the error notification
- update periodically the non-expired requests and return new data sets as soon as new data entries are inserted in the Database from the `DataCollector`

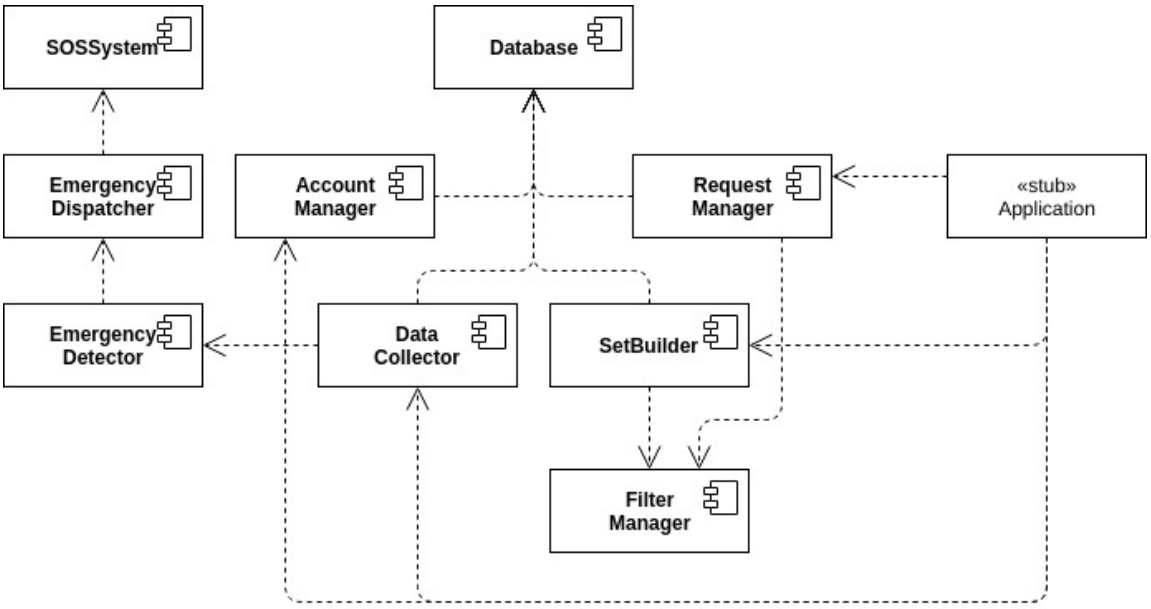

Figure 22: Integration and testing, step 2

In parallel to the `RequestManager` integration, we can focus on the red dependences, related to the SOS handling. We can now integrate in isolation from the other integration operations of this step the `EmergencyDetector` and `EmergencyDispatcher` components. The first shall be able to analyze the data entries as they're collected by the `DataCollector`, according to the current medical standards, while the second shall be able to build emergency messages according to the `EmergencyDetector` records and dispatch them to the SOS system. Clearly, the `EmergencyDetector` shall be implemented, tested and integrated before the `EmergencyDispatcher`, but the two components are not very complex with respect to the others and therefore we can safely allocate their integration in the same step.

5.5 Step 3

In the last step we can finally integrate the Application and the `PaymentGateway`. The `PaymentGateway` is the last component of the Application Server (blue *use* relations); it communicates with the Application, to mediate the interaction with the external payment system. The Application is the very important component to be integrated in this step, as it shall provide a graphical interface that acts as bridge between the users and the API calls to the Application Server. These calls shall interact with the `AccountManager`, `DataCollector` and `RequestManager` components, as shown in the component diagram (Figure 3).

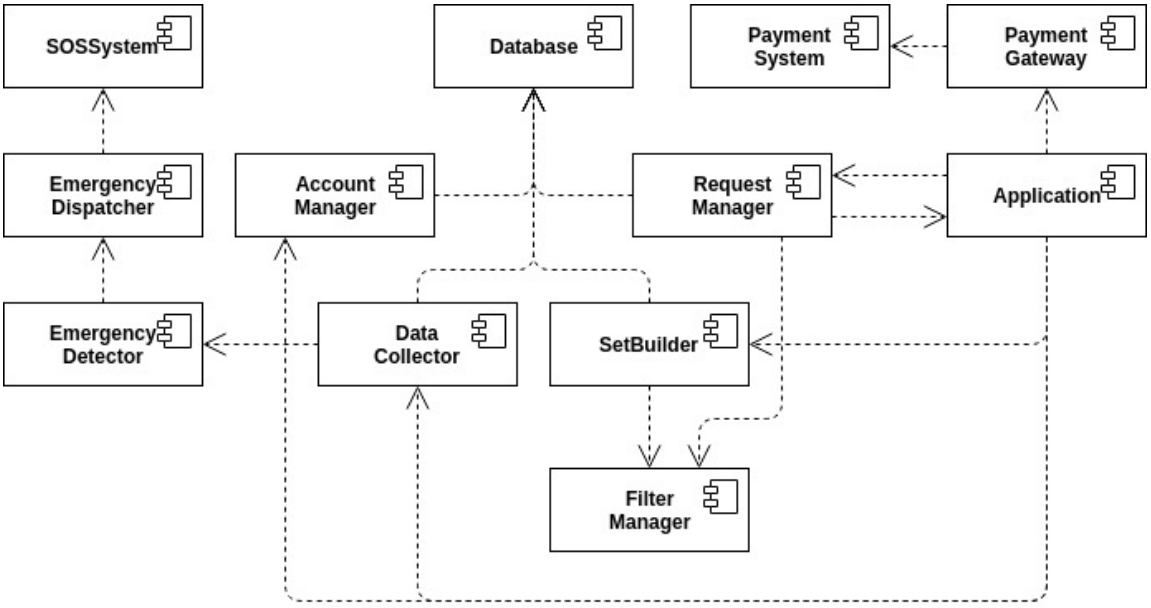

Figure 23: Integration and testing, step 3

At this point the top-down integration procedure is completed and system has been assembled in all of its parts. It can be tested as a whole entity, in all of its functionalities.

Effort spent

Date	Archetti Alberto	Carminati Fabio	Activity
12/11/2018	1	1	Introduction sketch
24/11/2018		6	User Interface Design
24/11/2018	3		High-level components
25/11/2018	2		Application Server sub-components
26/11/2018		5	Architectural Design
27/11/2018	2		Component interfaces
27/11/2018		3	Requirement Traceability
28/11/2018		1	Requirement Traceability
30/11/2018	1	1	High-level components
1/12/2018	6		Update sequence diagrams and component interfaces; added component diagram
2/12/2018	5	1	Architectural styles and patterns, UI, traceability
3/12/2018		3	Update sequence diagrams and interfaces
4/12/2018	4	3	Deployment, integration and testing
7/12/2018	3		Deployment, styles and patterns

Tools used

In this section we will list the tools used to produce this document:

- LaTeX for document writing and building
- `latextemplates.com` for the title page
- `draw.io` for drawing diagrams

References

[1] Mandatory Project Assignment AY 2018-2019

[2] IEEE Standard for Information Technology - Systems Design - Software Design Descriptions

[3] Collection and Processing of Data from Wrist Wearable Devices in Heterogeneous and Multiple-User Scenarios
`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5038811/`

[4] Wearable Devices in Medical Internet of Things: Scientific Research and Commercially Available Devices
`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5334130/`

[5] Amazon AWS
`https://aws.amazon.com/it/`

[6] Google Fit API
`https://developers.google.com/fit/overview`

[7] PayPal API
`https://developer.paypal.com/docs/`

[8] RapidSOS Emergency API
`https://info.rapidsos.com/blog/product-spotlight-rapidsos-emergency-api`

[9] Slides of the course by Prof. Di Nitto
`https://beep.metid.polimi.it/`

[10] Diagrams
`https://www.visual-paradigm.com/features/uml-tool/`