Francesco Pontiggia 921085 10524642
Arianna Ricci 921275 10536266

Trackme DD

DESIGN DOCUMENT (DD)

Release date: 10/12/18

Version 2.0

Table of content

1. Introduction

1.1. 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.

1.2. 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 to 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*)

1.3. Definitions, Acronyms, Abbrevations

1.3.1. Acronyms

- GUI: Graphical User Interface
- API: Application Program Interface
- MVC: Model- View-Controller
- RASD: Requirements Analysis and Specification Domain
- DD: Design Document

1.3.2. 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;

1.4. Revision History

This is the second release of this document. We have fixed the description of the component diagram.

1.5. Reference Documents

- TrackMe RASD v.1.1

1.5.1. Tools Used

- GitHub
- StarUML

- Draw.io
- MockFlow

1.6. Document Structure

1.6.1. 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).

1.6.2. 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

1.6.3. User Interface Design

Chapter 3 refers to chapter 3.1.1. of the RASD.

1.6.4. Requirements Traceability

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

1.6.5. 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.

1.6.6. Effort Spent

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

2. Architectural Design

2.1. Overview

The architectural design of our application contemplates three logical layers:

- The **presentation layer** 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.

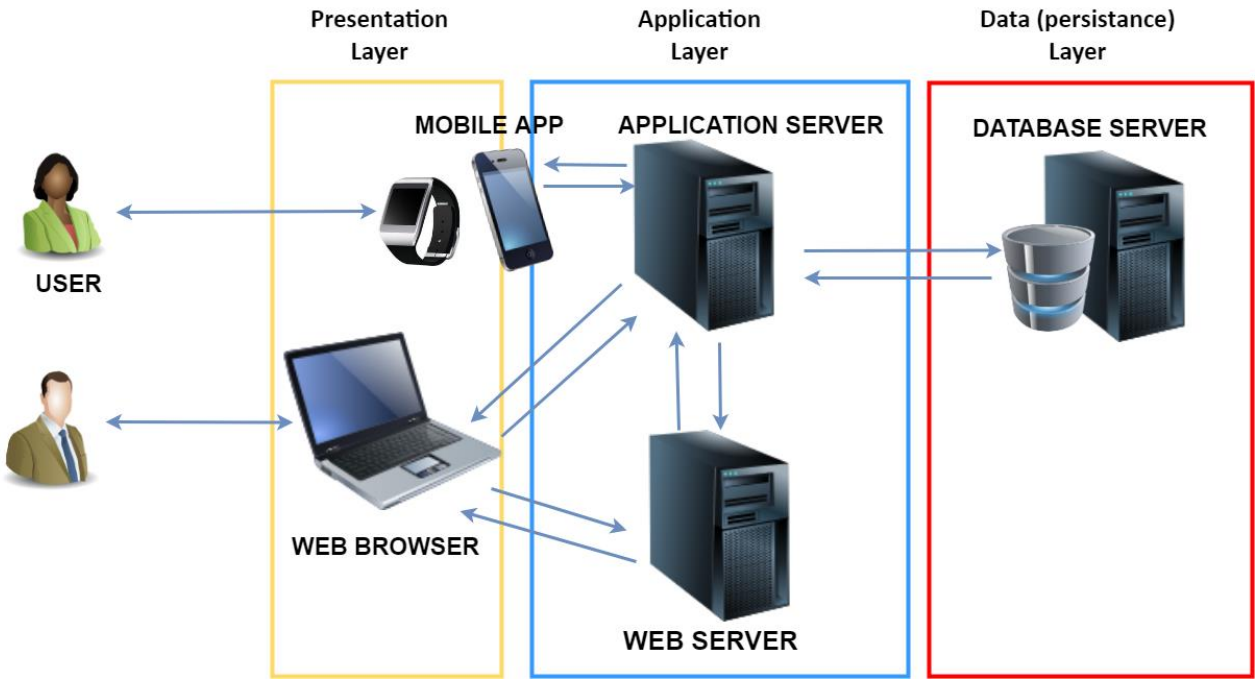

Figura 1: Architecture Diagram

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 which 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.
  While this tier is very important for Data4Help and it is used also by AutomatedSos in standard conditions, it may result useless or even harmful in the emergencies' management. For this reason, it is bypassed in notifying alerts. In this case the application tier communicates directly with the client tier. (see "Emergency Handling Sequence Diagram").

- 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 communicates 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.

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.

2.2. Component view

All the application layer is based on the following principle: 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.
Obviously, in the case in which the user is registered only to *Data4Help,* there is no need for downloading the components which regard only *AutomatedSOS*.

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. 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.

There is only one diagram, since it regards a single system which includes all the components of the two applications: indeed, MarketingComponent indicates the component which includes the graphical interface to the user. In the case of Data4Help, it means also the application we have supposed to build for "marketing purpose" (see RASD, *Assumptions*), while for AutomatedSOS it means simply the graphics of the mobile application.

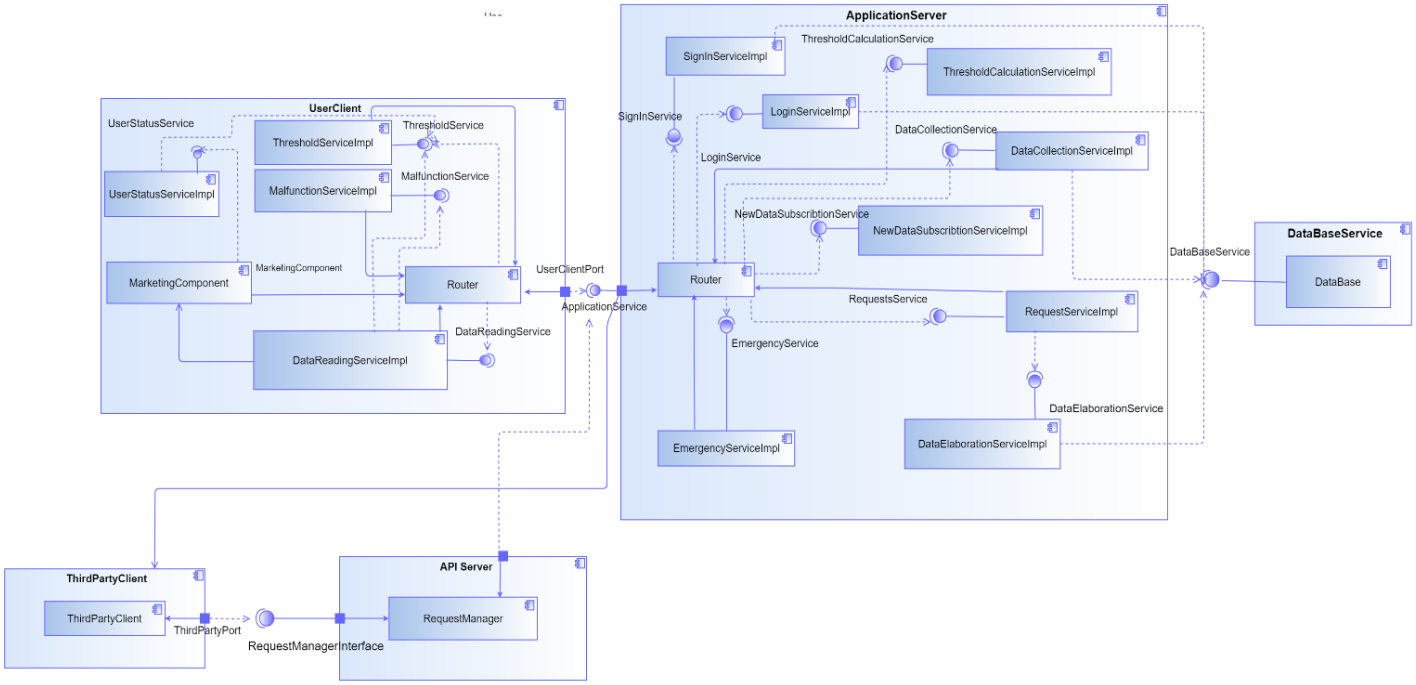

Figura 2: Component Diagram

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;

- **MarketingComponent** and **ThirdPartyClient** are part of the presentation layer and include manly the graphical interfaces;

- **DatabaseService** is part of the data layer and manages the carrying out of the queries;

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.

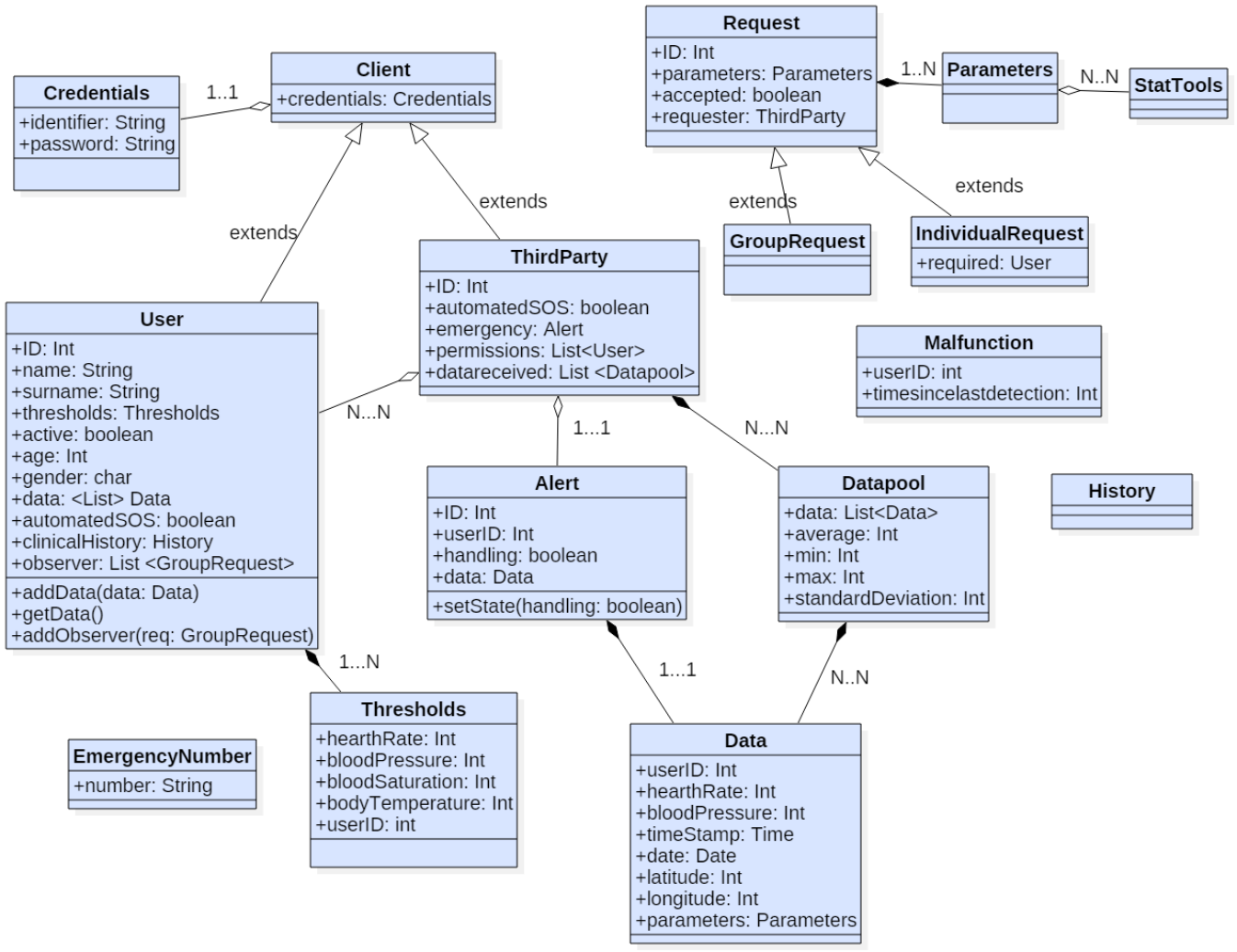

Figura 3: Model Class Diagram

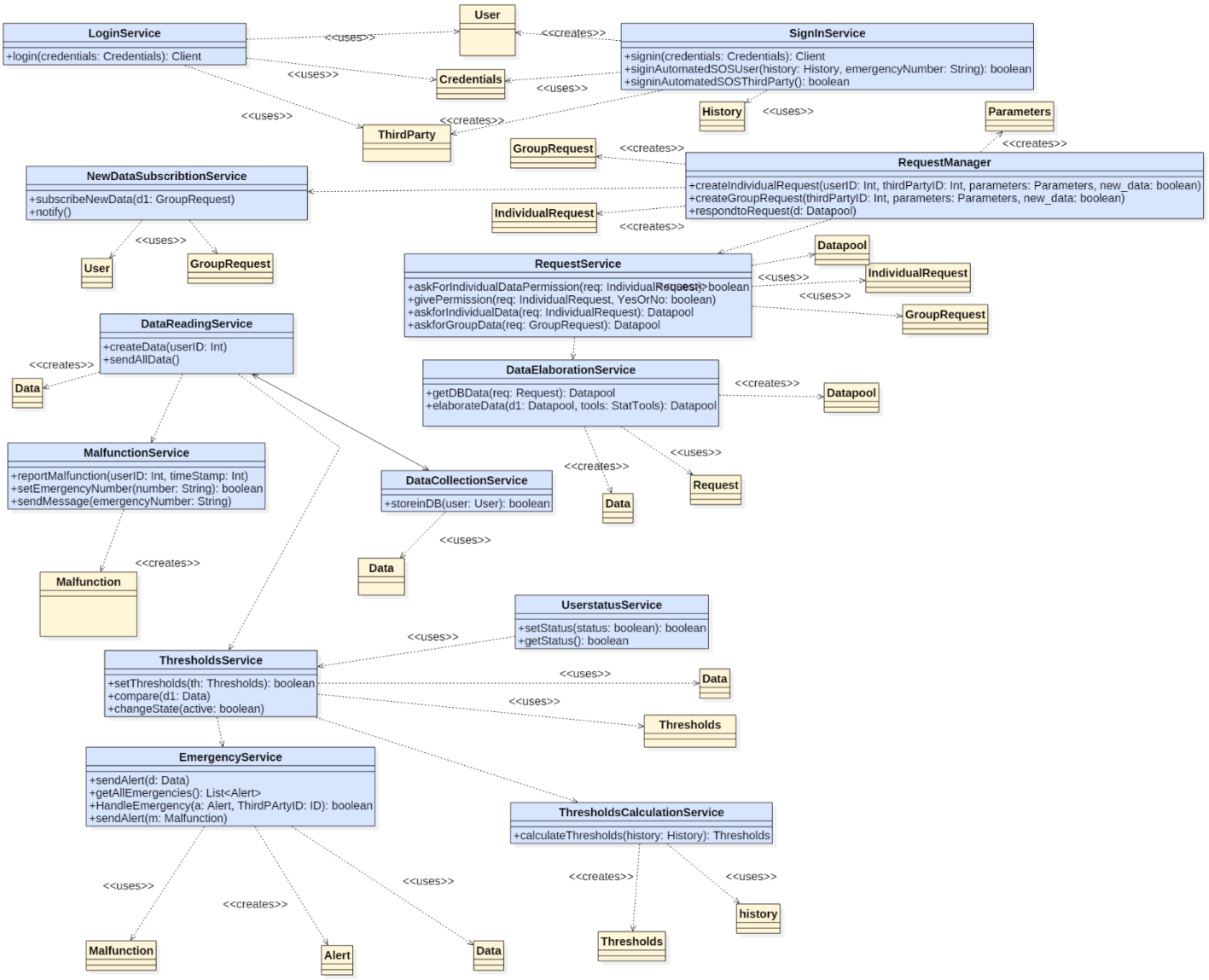

Figura 4: Object Diagram

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.

2.3. 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

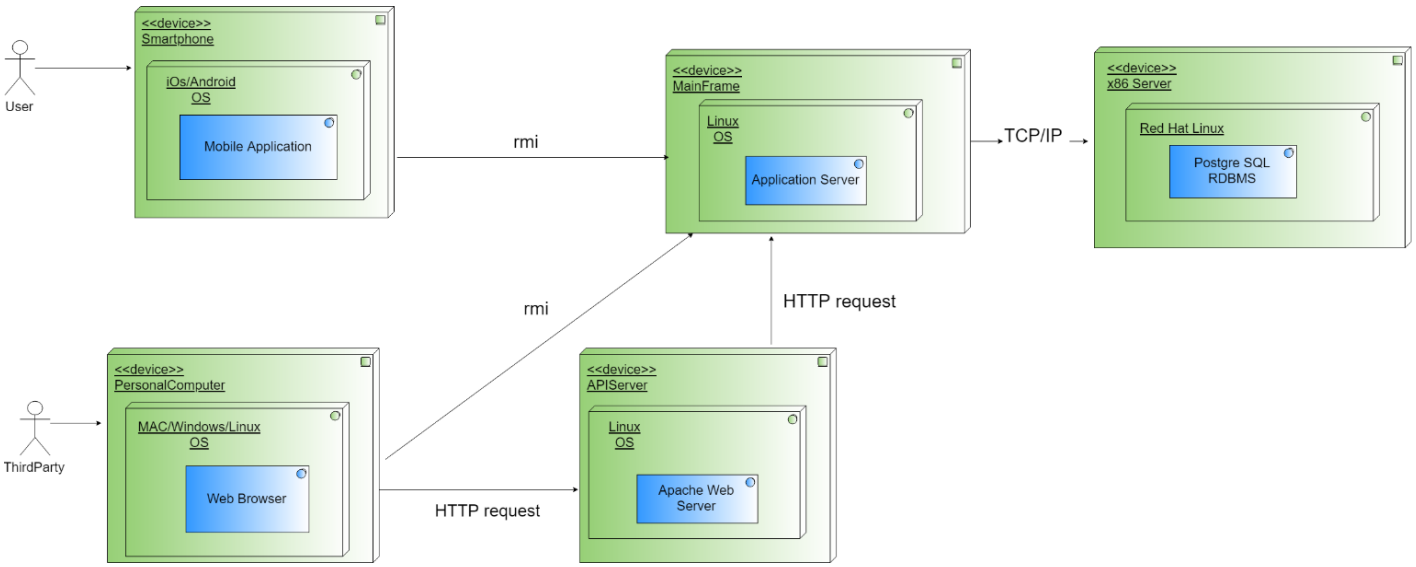

Figura 5: Deployment Diagram

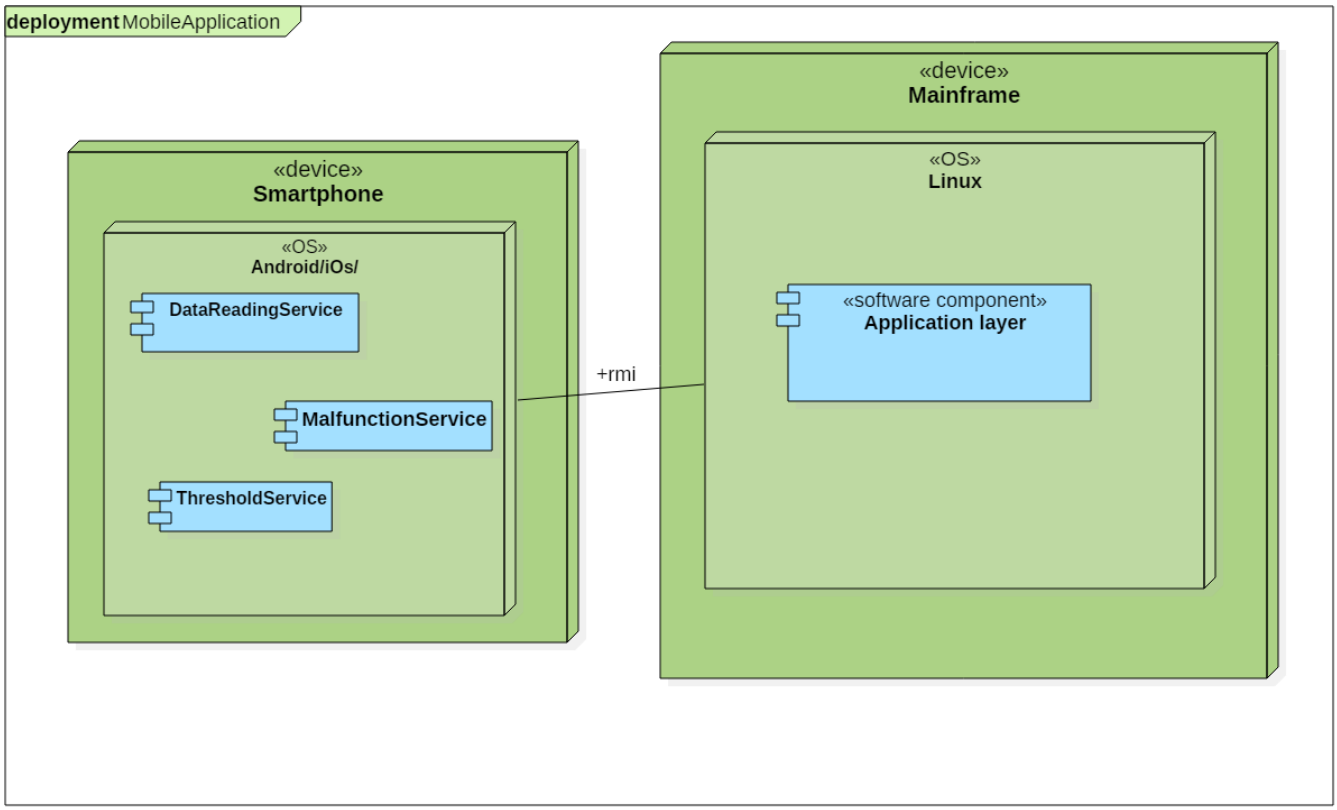

Figura 6: Deployment Diagram Mobile Application

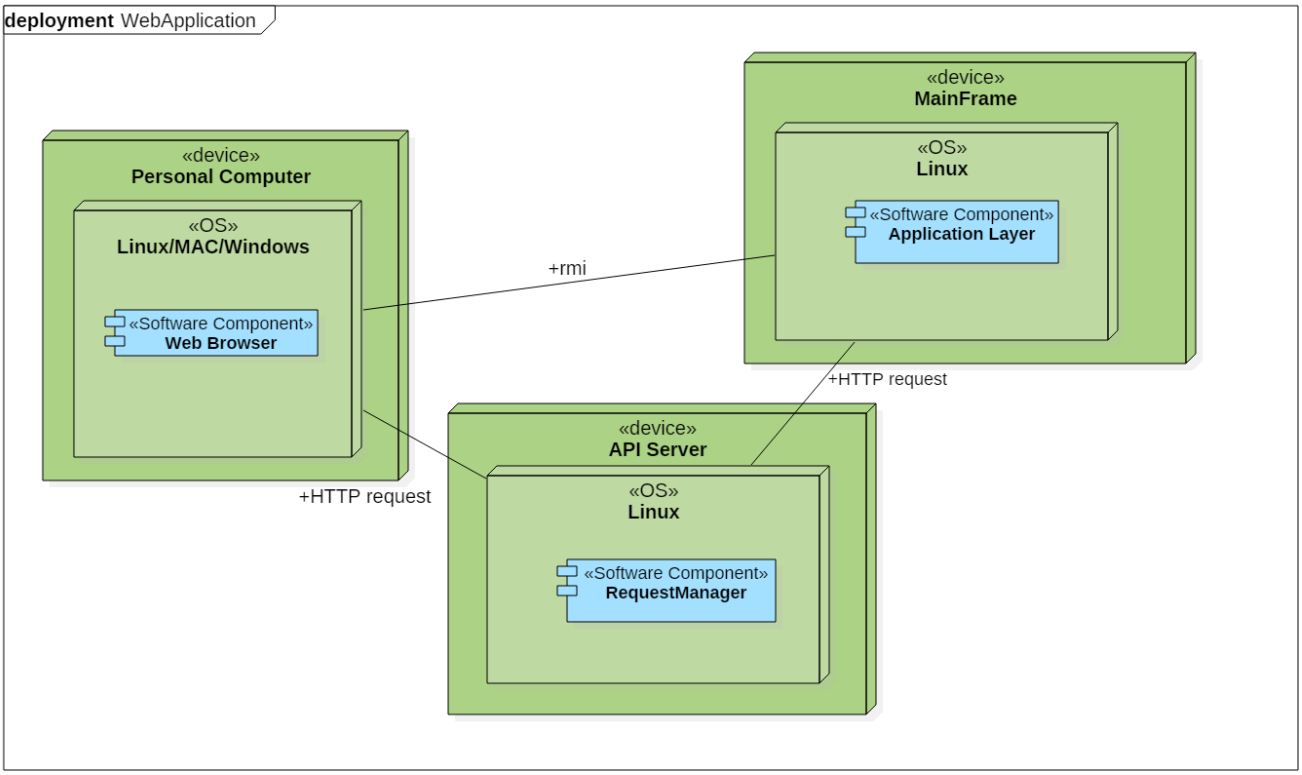

Figura 7: Deployment Diagram Web Application

2.4. Runtime view

2.4.1. Create Data

This diagram shows the creation and the storing in the database of a single data.
The components mainly involved in this process are DataReadingService and DataCollectionService. The first receives the information from the MarketingComponent and creates a data based on the given parameters.
Then it adds the data to the data's list of the user ("producing" data).
Then DataReadingService asks the DataCollectionService to store data in the database, by passing the user id as a parameter. So DataCollectionService gets data from the user and once received, sends it to DataBaseService, in order to store it ("consuming" the data). Then it requests new data, according to the "Producer/Consumer" pattern.
In this diagram is modelled the creation of a single data as an example of what the system continuously does in standard condition to create and store daily data.

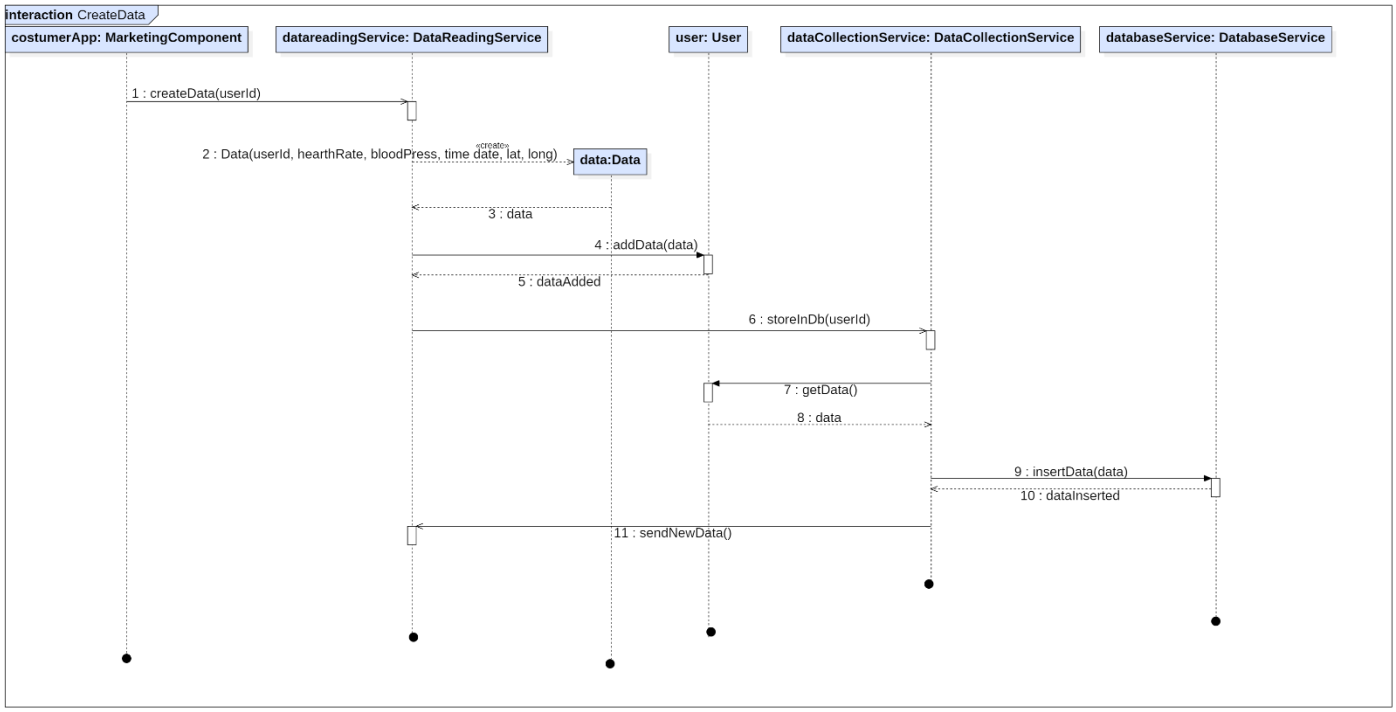

Figura 8: Create Data Sequence Diagram

2.4.2. Emergency Handling

This diagram shows how an emergency is handled by the application. the components most involved in this operation are EmergencyService and ThresholdService.
We supposed that ThresholsService uses a State Pattern or whatever it is characterized by two different states: on and off, set according to the user status .
So DataReadingService asks ThresholdService to compare the data with the thresholds every time it received data from the device, then the ThresholdService makes different operations depending on the State in which it is situated.
If its State is "OFF", it simply returned the call, without doing anything.

If its State is “ON”, it compares the data received from DataReadingService. If the value is okay, nothing happens, otherwise the collected data are sent to EmergencyService.
Then EmergencyService creates an alert containing all the data received from ThresholdService, which is forwarded to all the Third Parties.
When a third party takes charge of the emergency, EmergencyService edits the alarm status accordingly.

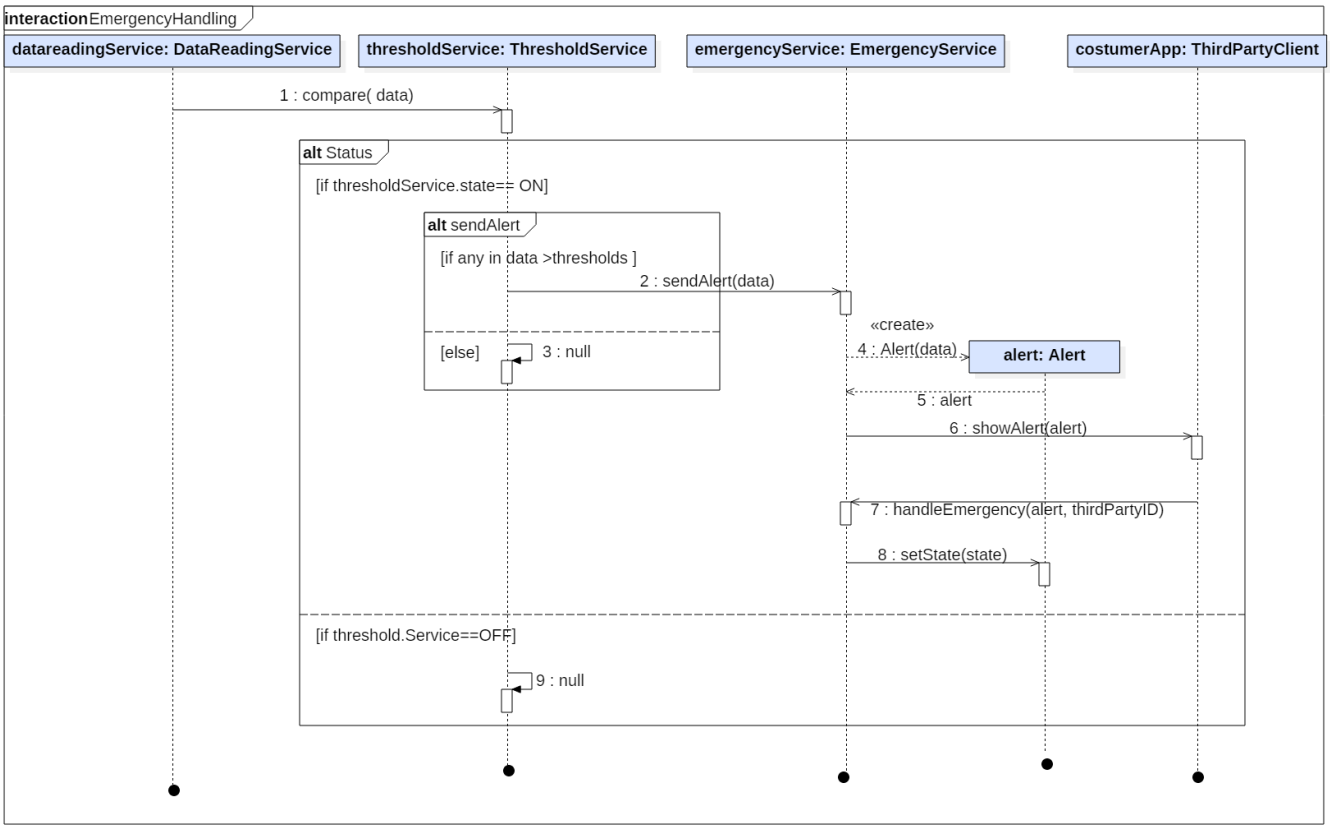

Figura 9: Emergency Handling Sequence Diagram

2.4.3. Make Group Request

This diagram shows the procedure to create a request for a group of data by a third party. It shows the interaction between the different requests and data management services that are provided by the application, but here only the case in which there isn’t request for new data is shown. (this alternative is shown in the “New Data Subcription Diagram”).
The first step passes from the RequestManager, which creates the grouprequest with the parameters received and forwards it to the RequestService. It sends the request to the DataElaborationService.
The DataElaborationService makes a query to the database with the information contained in the request and creates data objects from the tuples received as answer. Then it pushes them in a datapool object and sends it to the RequestService (in the diagram the creation of data objects isn’t modelled because it concerns only the DataElaborationService and not its interactions with other components).
If the data is properly anonymized (it involves more than 1000 users) and there are no statistical tools in the parameters of the request, the RequestService sends it to the RequestManager , that forwards the data to the costumer app of the Third Party. Otherwise the RequestService ask the DataElaborationService to elaborate the data and receives a new datapool, containing also the tools.
If the data cannot be anonymized, the RequestService refuses the request as the RequestManager.

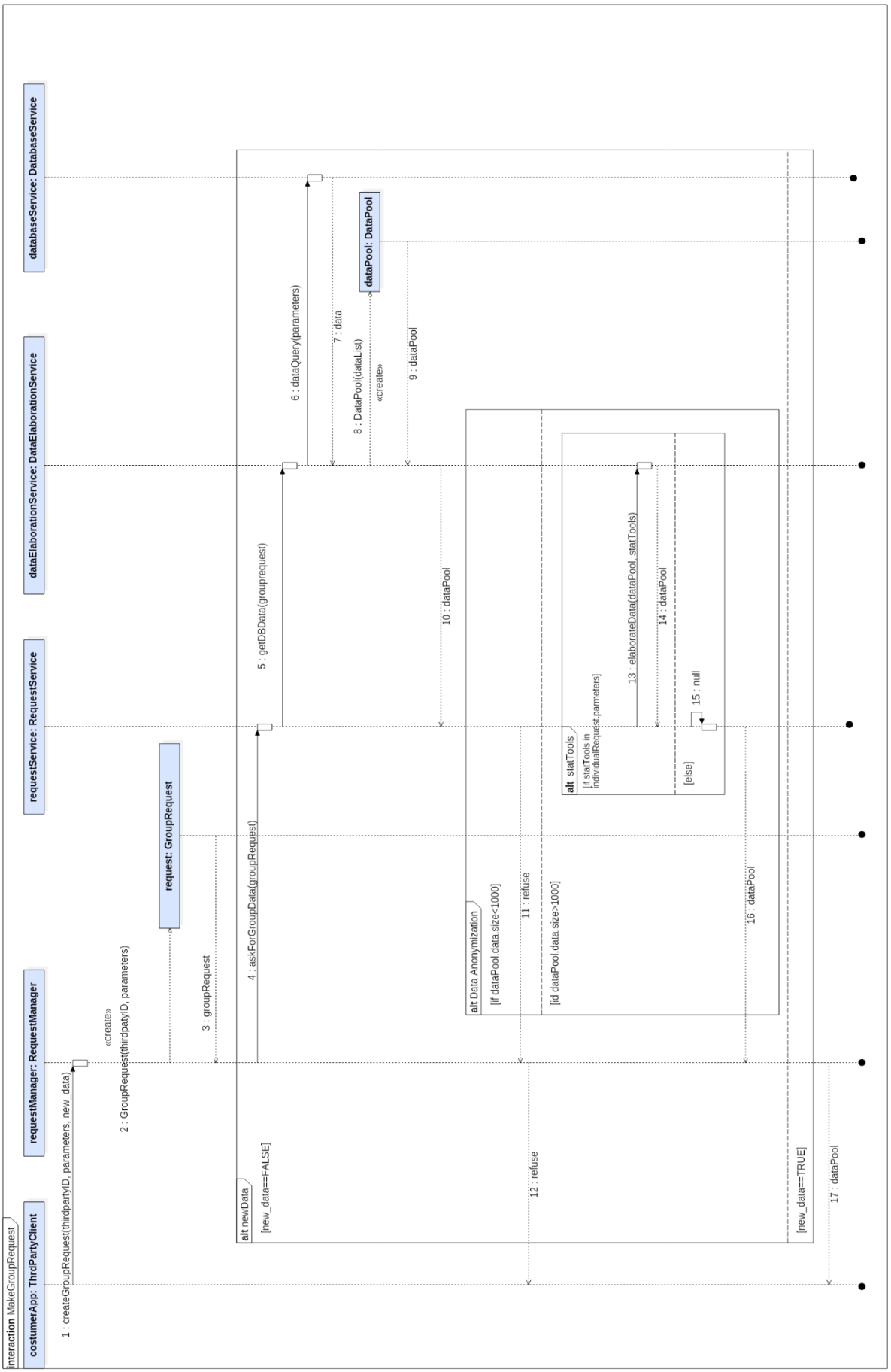

Figura 10:Group Request Sequence Diagram

2.4.4. Subcribe New Data

This diagram shows the process to follow to subscribe to new data. It then shows the case where in the request for a group of data is subscribed to new data. (It is the alternative [new_data==TRUE] of the “Make Group Request” diagram).
In this case then, after creating the request, the RequestManager notifies the NewDataSubscriptionService (and not the RequestService) that registers itself as observer of the users that match the request parameters.
Then a timeout starts and there are two different options: the timeout expires without anything happened or the NewDataSubscriptionService receives a notify.
The first option means that there are no new data for the subscription and so the request’s refuse is sent to the costumer app.
The second option means that there were updates about the data required. Indeed, the notify is sent to NewDataSubscriptionService only when the DataCollectorService asks to the user data to update the database. In this case the data are stored on the Databae, the RequestManager is notified that data is ready and the process continues as in a request for a group of data until the datapool does not arrive to the costumer app. (see “Make Group Request” diagram).

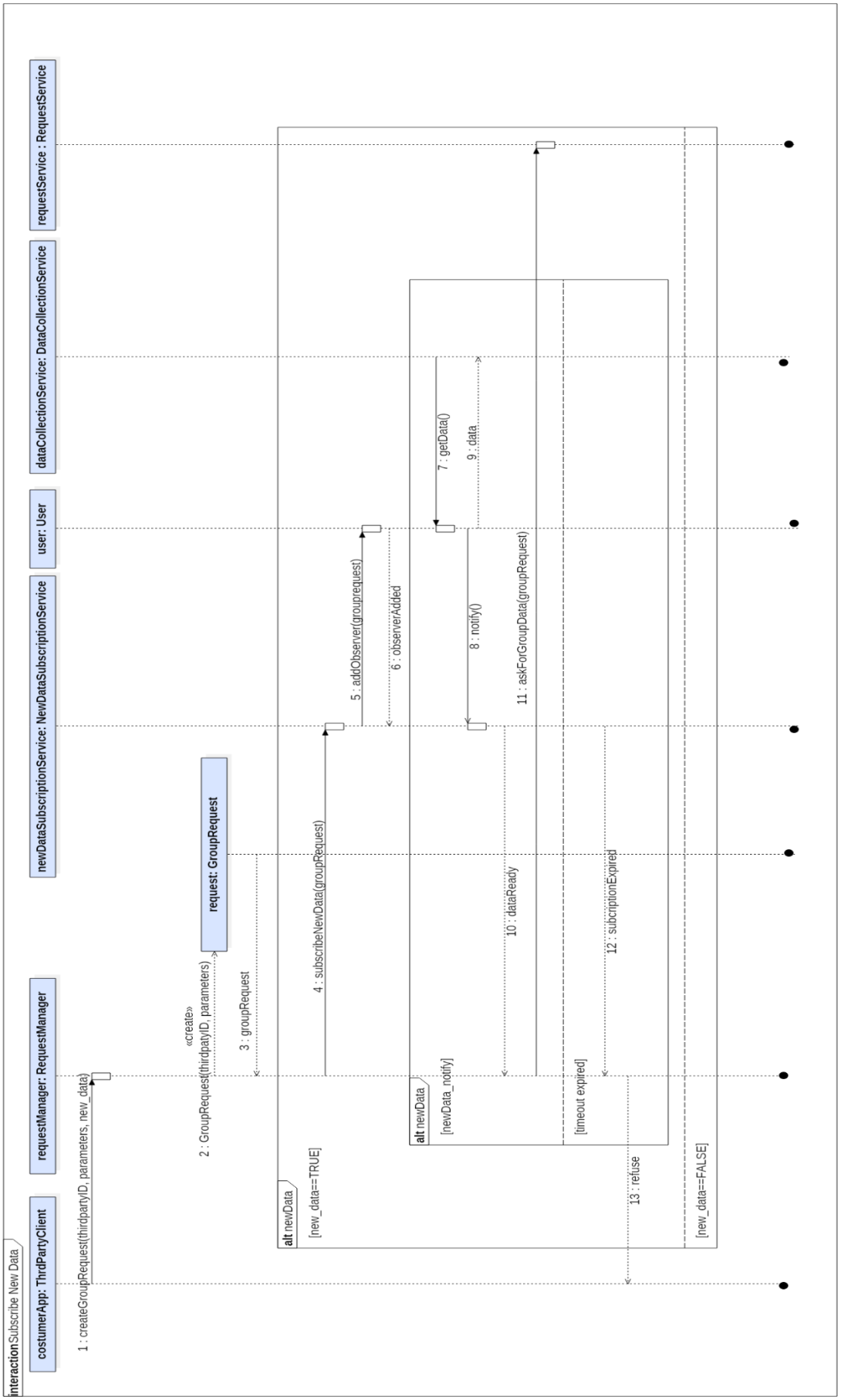

Figura 11:Subscribe New Data Sequence Diagram

2.4.5. Make Individual Request

This diagram shows the data of a specific user request's management. We decided to model only the case where the user gives access to the data because we found it more meaningful for the interaction between components (in the other cases there is just a rejection message in response to the request.)
As for the group request, RequestManager creates the individualrequest with the parameters sent from the third-party costumer app. Then the RequestManager forwards the request to the RequestService, which sends it to the user costumer app (MarketingComponent). When the user gives his/her permission to access data, the RequestManager is allowed to perform the actual request to the RequestService which forwards it to the DataElaborationService which sends the query to the DatabaseService.
Then DataElaborationService creates a datapool that is sent back up to the costumer app; as for requests for group there may be statistical tools.

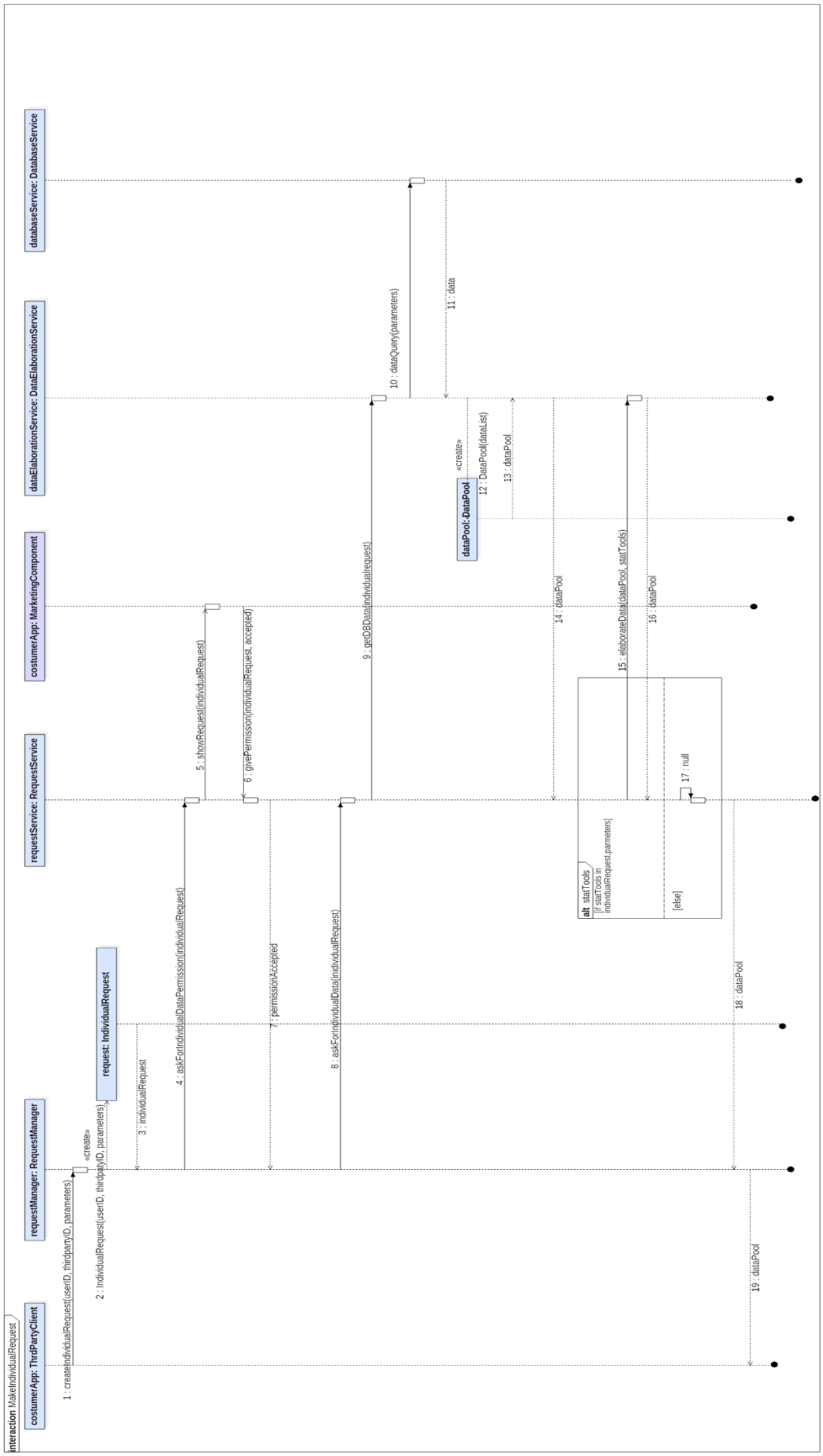

Figura 12: Individual Request Sequence Diagram

2.4.6. Malfunction Detection

This diagram shows the management of a malfunction: the component mainly involved is MalfunctionService.
If DataReadingService does not receive data for a pre-established period of time, it reports the malfunction to MalfunctionService, sending the userID and the timestamp of the last data received. Once received the notice, the MalfunctionService creates a malfunction notification with the information received and forwards it to the costumer app. After that there are three possible cases.
In the first case the client responds that everything is ok, so nothing more happens.
In the second case the client responds that an emergency is occurring, so the MalfunctionService sends an alert to EmergencyService, including a malfunction object.
in the third case does not get any response, so the MalfunctionService sends a message to the emergency number.

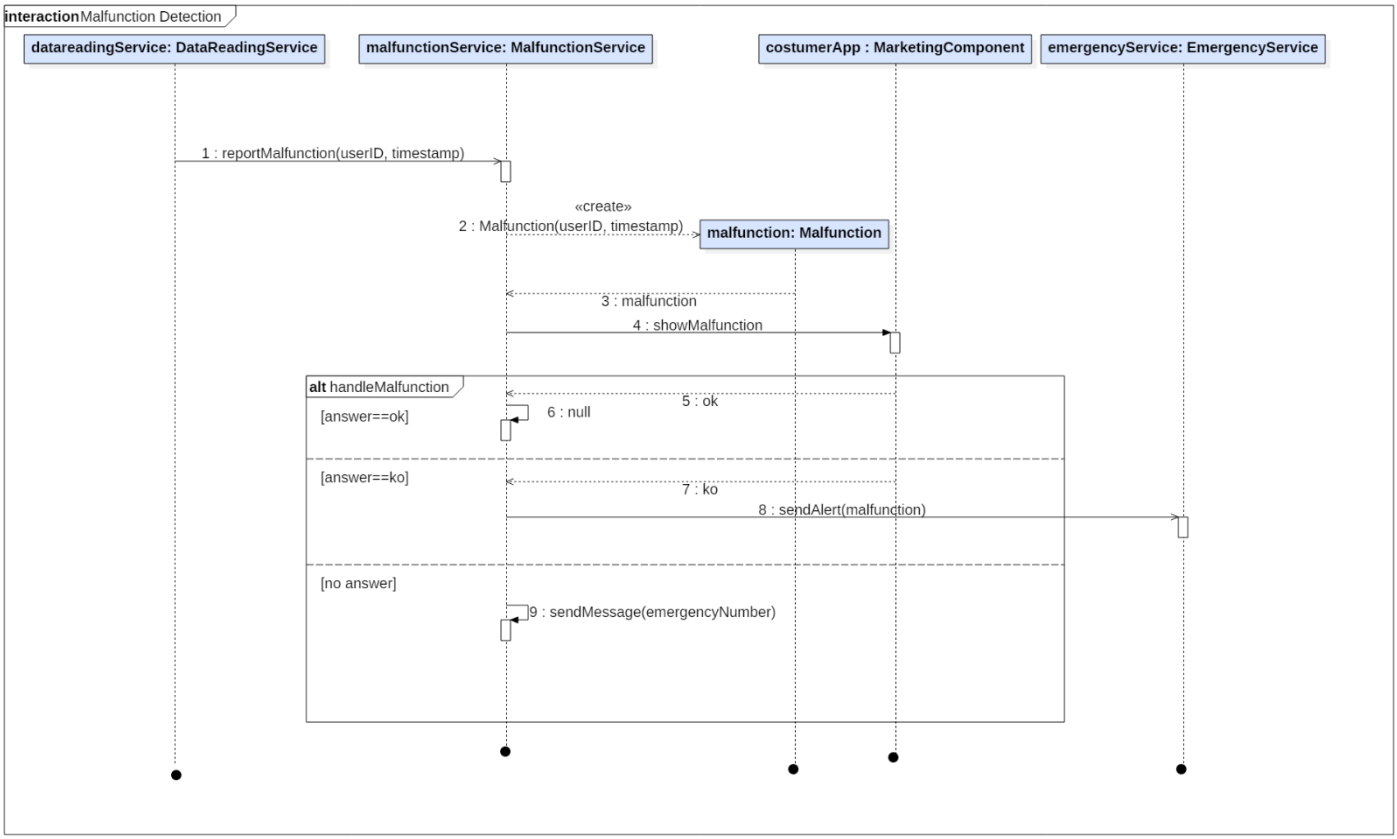

Figura 13: Malfunction Detection Sequence Diagram

2.5. Component interfaces

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

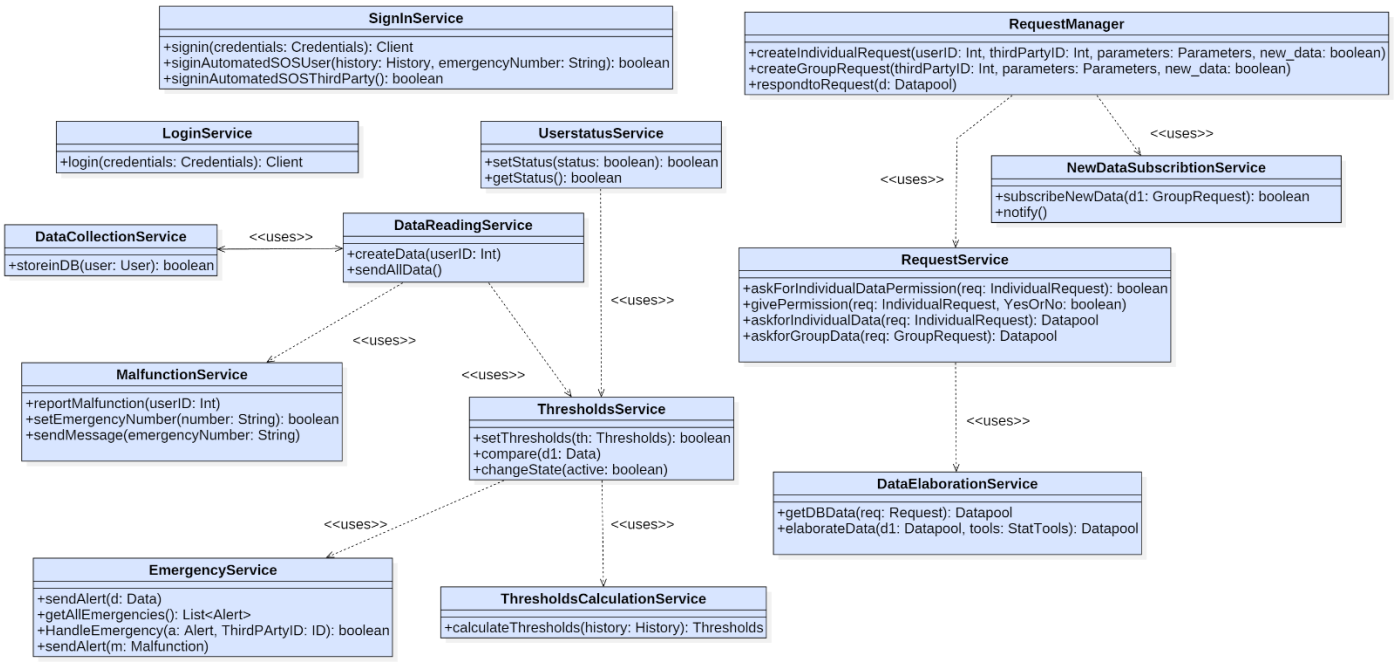

Figure 14: Interfaces Diagram

2.6. 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 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.

3. User Interface Design

Regarding the User Interfaces we chose to enrich the part already presented in the RASD, improving the mockups in an incremental version of the document.

To avoid redundancy this chapter includes only a reference to the chapter 3.1.1. of the RASD.

4. Requirements Traceability

- **[G1]** Provide a form of unique identification (registration/login) of all clients of the services.
  (Requirements **[R1], [R2], [R3]**)
  - LoginServiceImpl
  - SigninServiceImpl

- **[G2]** Allow the user to avoid being associated to his data without his permission.
  (Requirements **[R6], [R7], [R8], [R9]**)
  - RequestManager
  - RequestServiceImpl
  - DataElaborationServiceImpl
  - MarketingComponent

- **[G3]** Allow third parties to request access to data of some specific individuals.
  (Requirements **[R6], [R10]**)
  - DataReadingServiceImpl
  - DataCollectionServiceImpl
  - RequestManager

  - RequestServiceImpl
  - DatabaseService
  - DataElaborationServiceImpl

- **[G4]** Allow third parties to request access to anonymized data of groups of individuals.
  (Requirements **[R9], [R11], [R12]**)
  - DataReadingServiceImpl
  - DataCollectionServiceImpl
  - RequestManager
  - RequestServiceImpl
  - DatabaseService
  - DataElaborationServiceImpl

- **[G5]** Allow third parties to subscribe to new data.
  (Requirements **[R6], [R7], [R8], [R9]**)
  - DataReadingServiceImpl
  - DataCollectionServiceImpl
  - RequestManager
  - RequestServiceImpl
  - DatabaseService
  - DataElaborationServiceImpl
  - NewDataSubcriptionServiceImpl

- **[G6]** Allow third parties to obtain the most adapt data for their needs.
  (Requirements **[R15], [R16], [R17]**)
  - RequestManager
  - DataElaborationServiceImpl
  - RequestServiceImpl

- **[G7]** Third parties are alerted, whenever a user is in danger of life (personalized), the application is working properly and there is internet connection.
  (Requirements **[R18], [R19]**)
  - DataReadingServiceImpl
  - EmergencyServiceImpl
  - ThresholdsServiceImpl

- **[G8]** If the user's health status is not clear due to malfunctions, an Emergency number is alerted within an hour.
  (Requirements **[R4], [R20], [R21]**)
  - DataReadingServiceImpl
  - MalfunctionServiceImpl

- **[G9]** The user can temporarely suspend AutomatedSos in special cases (non-intrusive).
  (Requirements **[R22]**)
  - ThresholdsService
  - UserStatusServiceImpl

5. Implementation, Integration and Test Plan

5.1. 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.

Another good point of this implementation strategy is that a structural error in the general architecture could be detected very early in the implementation.

5.2. 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.

Testing follows step by step the implementation strategy: after implementing a thread, an integration test is performed to verify the correctness of the implementation with respect to the business service to provide.
Testing parts of the system which provide user visible features so early (when all the other parts may not exist) is a very good aspect of the plan.
These rules imply 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.

Under the description of every step, there is a component diagram which shows which components have been already implemented (in blue), which are being implementing at the current step (in green) and which haven't been implemented yet (in white).

5.3. The Model

Since the Model is shared by all components and is supposed to be stable over time, its implementation should be put first, considering also that eventual errors in Model could lead to expensive re writing of lots of blocks of code or to problems of adaptation due to incorrect modelling. All the classes are tested to verify the consistency of the Model before implementing any service. No explicit Integration Test with the services is planned, because the intensive usage and interaction makes it difficult to perform and not so much useful:

they are actually integrated and tested “on the go” at every step of the presented integration process.

5.4. The Implementation, Integration and Test Steps

5.4.1. 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.

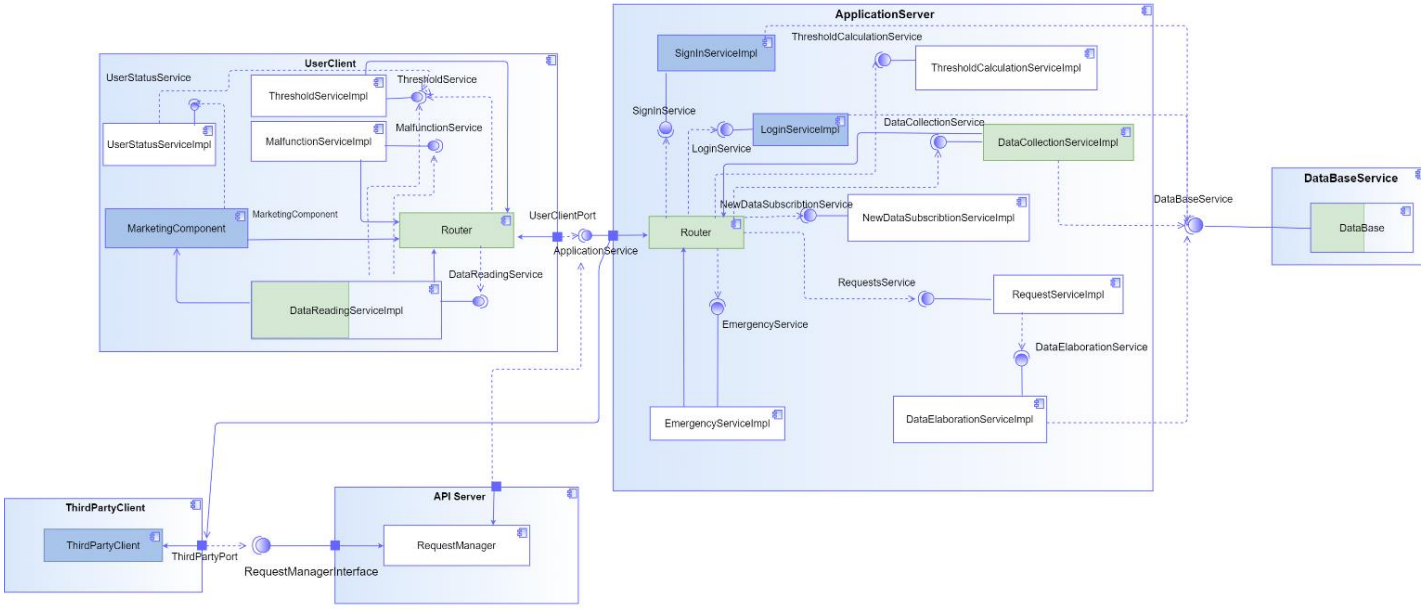

Figura 14: Implementation Step 1

5.4.2. 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.

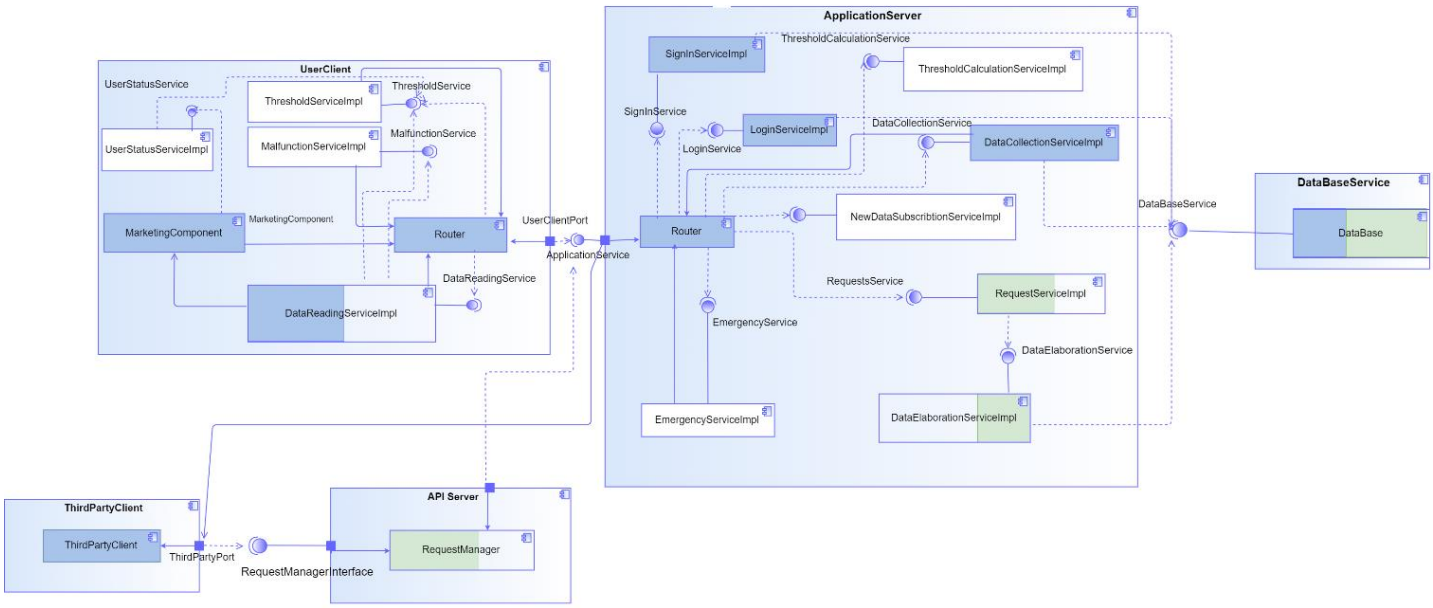

Figura 15: Implementation Step 2

5.4.3. 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.

5.4.4. 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.

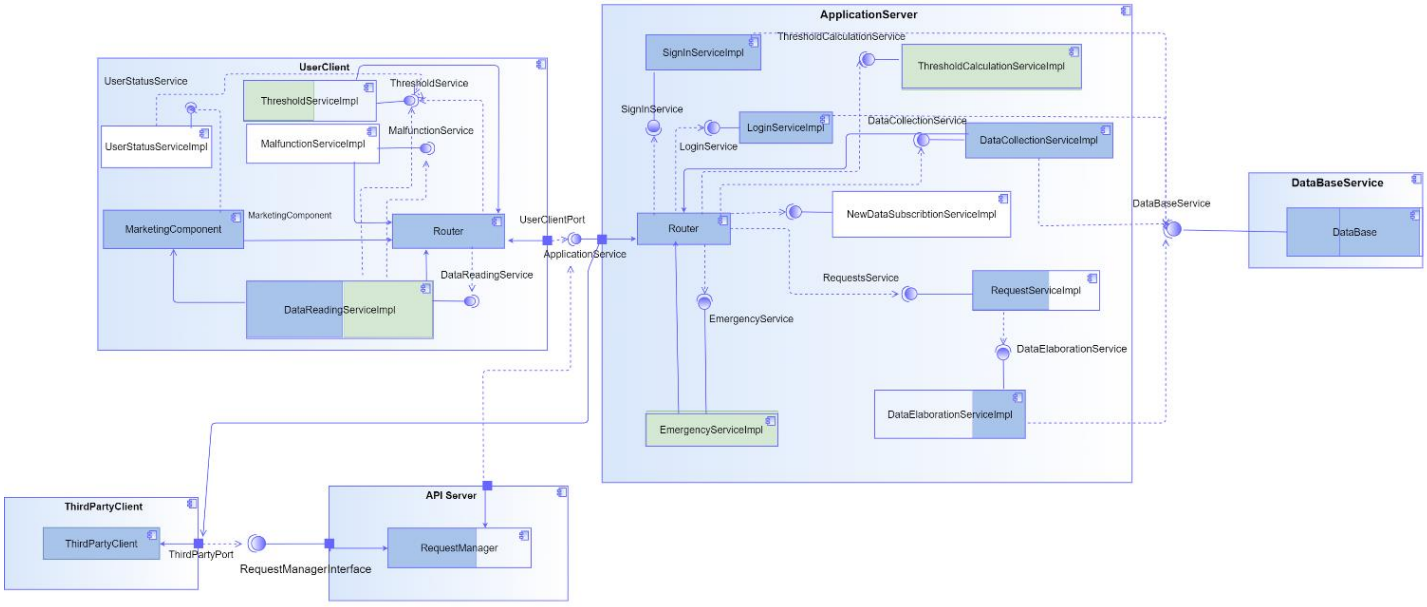

Figura 16:Implementation Step 4

5.4.5. 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.

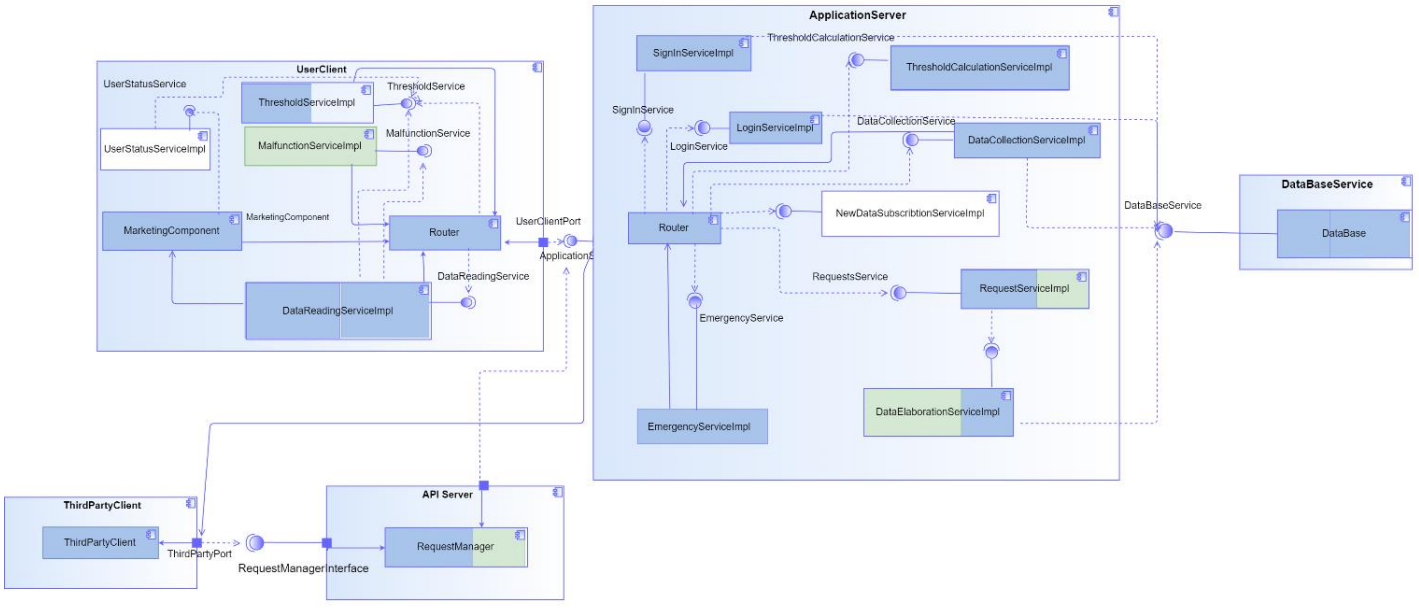

Figura 17:Implementation Step 5

5.4.6. 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.

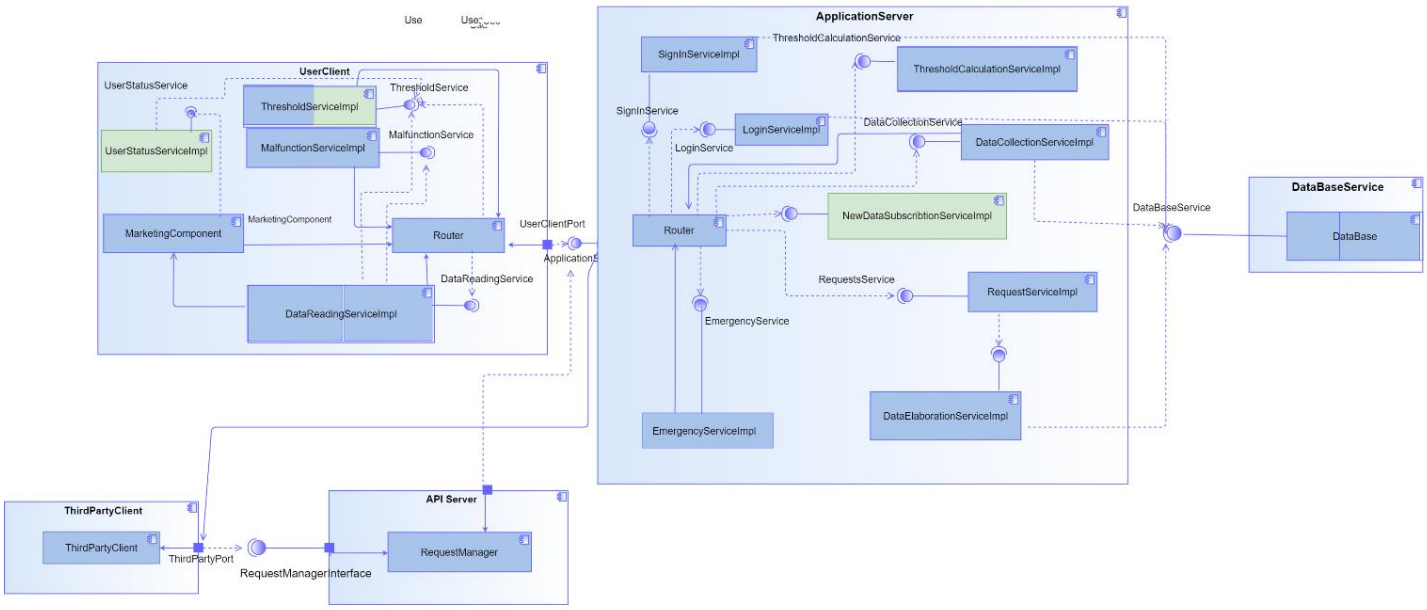

Figura 18:Implementation Step 6

5.4.7. 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.

5.4.8. 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.

5.5. 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 should 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. Of course, every release of the system (even partial) can't exist without a completely implemented and sufficiently tested version of these services. Because of that, in the previous diagrams they are always depicted as "already implemented".

5.6. The presentation layer

The implementation of the components of this layer could be postponed as much as desired, provided that some stubs and drivers are developed to correctly test the remaining part of the system. The reasons for this choice are that these are not crucial modules for the company and, moreover, there can be frequently significant changes in the code (e.g. in the

marketing component, where marketing issues could affect at every time the implementation).

6. Effort Spent

The following table aims at summarizing the effort spent by each component of the group, in discussing, examining solutions and writing them. After working together in the first phase to define the core of the project, everyone focused on some specific topic. The data are expressed in hour.

	Francesco Pontiggia	Arianna Ricci
Purpose and Scope	9	7
Architectural Design	20	21
Requirements Traceability	1	3
User Interfaces Design	1	4
Implementation, Integration and Testing Plan	16	4
Revision	6	8