TrackMe

DD

Design Document

Version: 1.0

Antonio Urbano
Enrico Voltan

Release date: 10/12/2018

Table of Contents

1. Introduction

1.1.Purpose

The purpose of this document is to give more technical details than the RASD about TrackMe system in order to provide an overall guidance to the architecture of the software product.
While the RASD presented a general view of the system and what functions the system is supposed to execute, this document aims to present the implementation of the system including main components and their interfaces, run-time behaviours, the high-level architectures and the corresponding deployment design. It also presents in more details the implementation and integration plan, as well as the testing plan.

1.2.Scope

The project TrackMe, which is a service-based on mobile application and web application, has two different targets of Customers:

- Third-Parties
- Users

First of all, the system must provide the registration and login services.
To log into the system, both the Users and the Third-Parties will use their own credentials, such as username and the related password.
More precisely, the sign up and the sign in processes are carried out via:

- web app by Third Parties
- mobile application by the Users

Furthermore, the system allows Third Parties to require accessing to Users' data via web app. More precisely, after choosing the type of request (single User data request or data w.r.t a specific group of anonymous Users) and filling out the corresponding fields, Third-Parties can make a data request.
On the other side the mobile app allows Users to see their own pending requests, to accept or reject a new single User data request or to withdraw a previously accepted request.
Group requests are, however, handled directly by the system in order to verify if it is able to properly anonymize the requested data.
The answer to a specific data request must be communicated on the corresponding Third-Party's web app and for the accepted requests the system will provide a set of APIs in order to allow Third Parties to access data.

Through the mobile app, Users can also keep track of their health status by comparing them with the threshold values and update some fields of personal data (weight, height, etc.).
Finally, Users via mobile app can also subscribe to the AutomatedSOS service. For these Users the system must monitor constantly their health status in order to notify an external Ambulance Service in case of emergency.

1.3.Definitions, Acronyms, Abbreviations

1.3.1. Acronyms

- **RASD:** Requirement Analysis and Specification Document
- **DD**: Design Document
- **MVC**: Model View Controller
- **REST**: REpresantional State Transfer
- **API:** Application Programming Interface
- **GPS:** Global Positioning System
- **Bpm:** Beats Per Minute
- **SSN:** Social Security Number
- **OS:** Operating System

1.3.2. Abbreviations

- **[Gn]:** n-th goal
- **[Rn]:** n-th functional requirement
- **Tp-a:** Third-Party administrator

1.4.Document Structure

This document is divided into six sections:

- Section 1 gives an introduction of the design document. It contains the purpose and the scope of the document, as well as some abbreviation in order to provide a better understanding of the document to the reader.
- The second section deals with the architectural design of the application.
  It gives an overview of the architecture and it also contains the most relevant architecture views: component view, class view, deployment view, runtime view and it shows the interaction of the component interfaces. Some of the used architectural

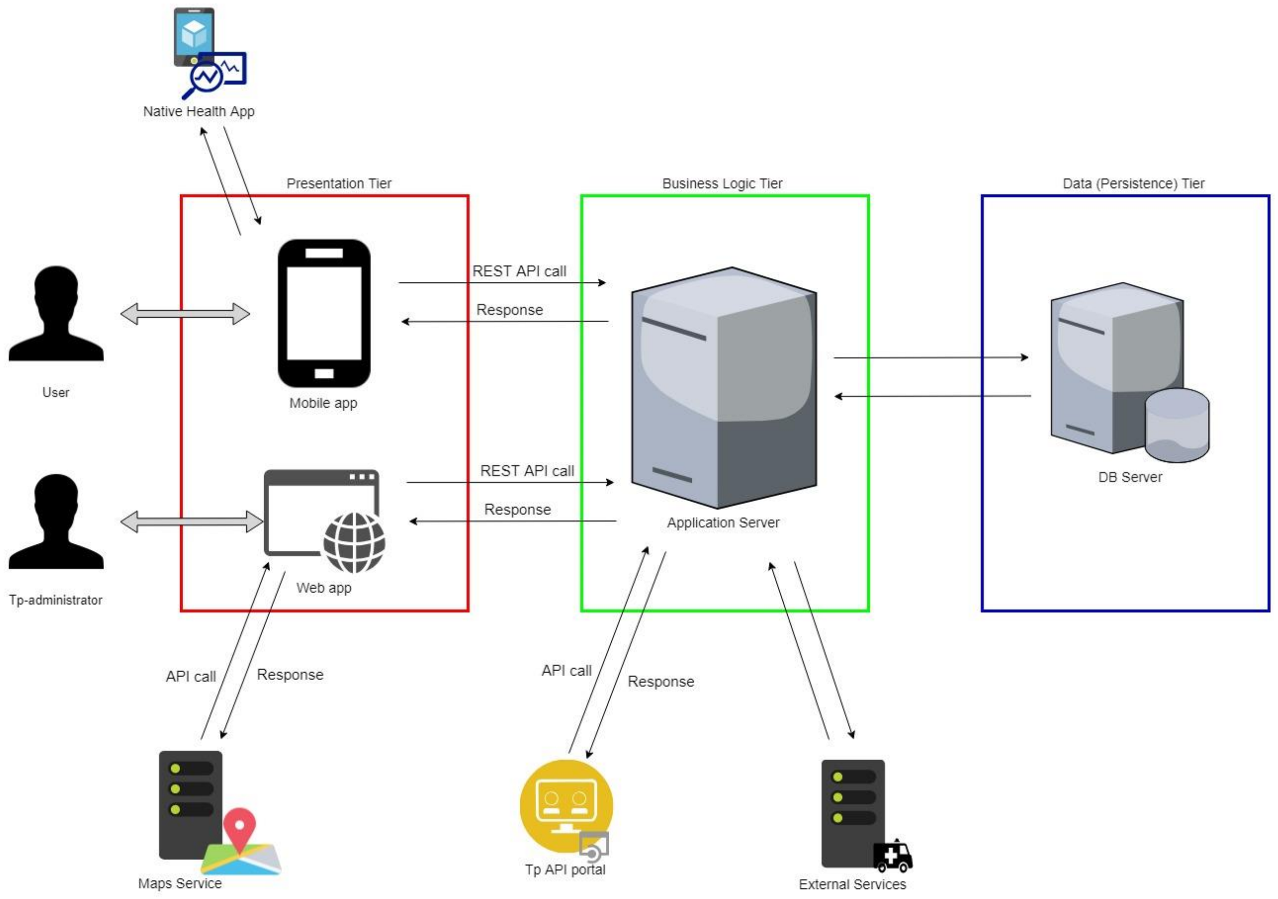
designs and designs patterns are also presented here, with an explanation of each one of them and the purpose of their usage.

- Section 3 refers to the User Design Interface previously presented in the RASD document through some mock-ups w.r.t the Users and the Third Parties interfaces.
- The fourth section explains the mapping between the requirements previously defined in the RASD and the design elements that are defined in this document
- The fifth section provides the description of the implementation and testing strategy adopted in the whole project and of the order in which it is planned to integrate such subcomponents.
- Section 6 shows the effort spent by each group member while working on this project.
- Section 7 includes the reference documents

2. Architectural design

2.1. Overview and High-Level Architecture

The TrackMe system is based on a widely used 3-tier Architecture:



Where:

- **Presentation tier**: It is a graphical user interface accessible by the end Customers. It consists of a mobile application used by the individual Users a web app used by Third-Parties.

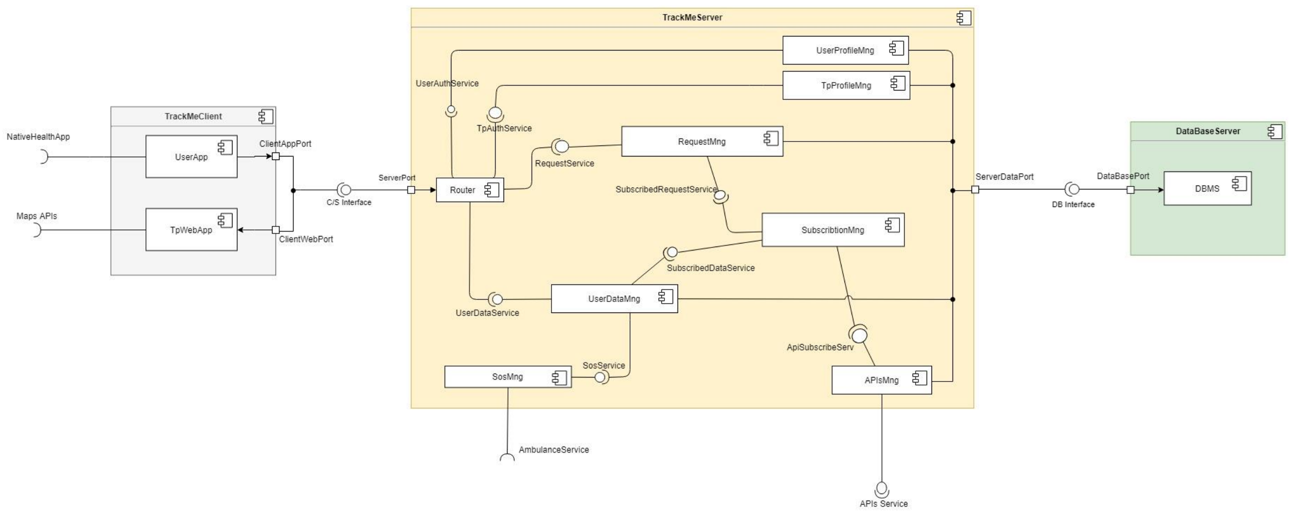
  The GUI communicates with the application server, by using a set of REST APIs calls, but as well as with the external services. More precisely the mobile app communicates with a native health app in order to collect constantly health status values, while the web app communicates with a Maps Platform by using its APIs.
- **Business Logic tier**: This layer contains the logic core of the services provided by TrackMe.

  It defines a common set of application features that will be available to end Customers (both Users and Third-Parties) and communicates with tier 3 and with external services (the Ambulance Service and the Third-Parties' APIs portals).

- **Data (Persistence) tier**: It is the layer where all the main data are stored (e.g. all the credentials for the Login processes, the health status values collected form the Users' mobile apps, the data relative to the Third-Parties' subscriptions and requests, data for the AutomatedSOS service, etc.).
  It co-works with the Business Logic Tier by providing the data needed in the most operations.

2.2. Component view

The previously introduced components are more precisely described and examined by using a component diagram as follows.
While describing the parts, the notation will be the interface they are providing in order to be more general.
The main focus of the diagram is the application server and, except for the database server and its DBMS, the other external services are not depicted here. Indeed, the diagram deals only with the services that they provide, while their structure will not be analysed here, since they are observed as black-boxes.



As we can see the application Server consist of the following components:

- **UserProfileMng**: the component responsible for the authentication request by the user (both registration and log in requests).
  For each authentication request by a user, after receiving credentials and checking with the Database, it interacts (through the Router) with the mobile app to provide a

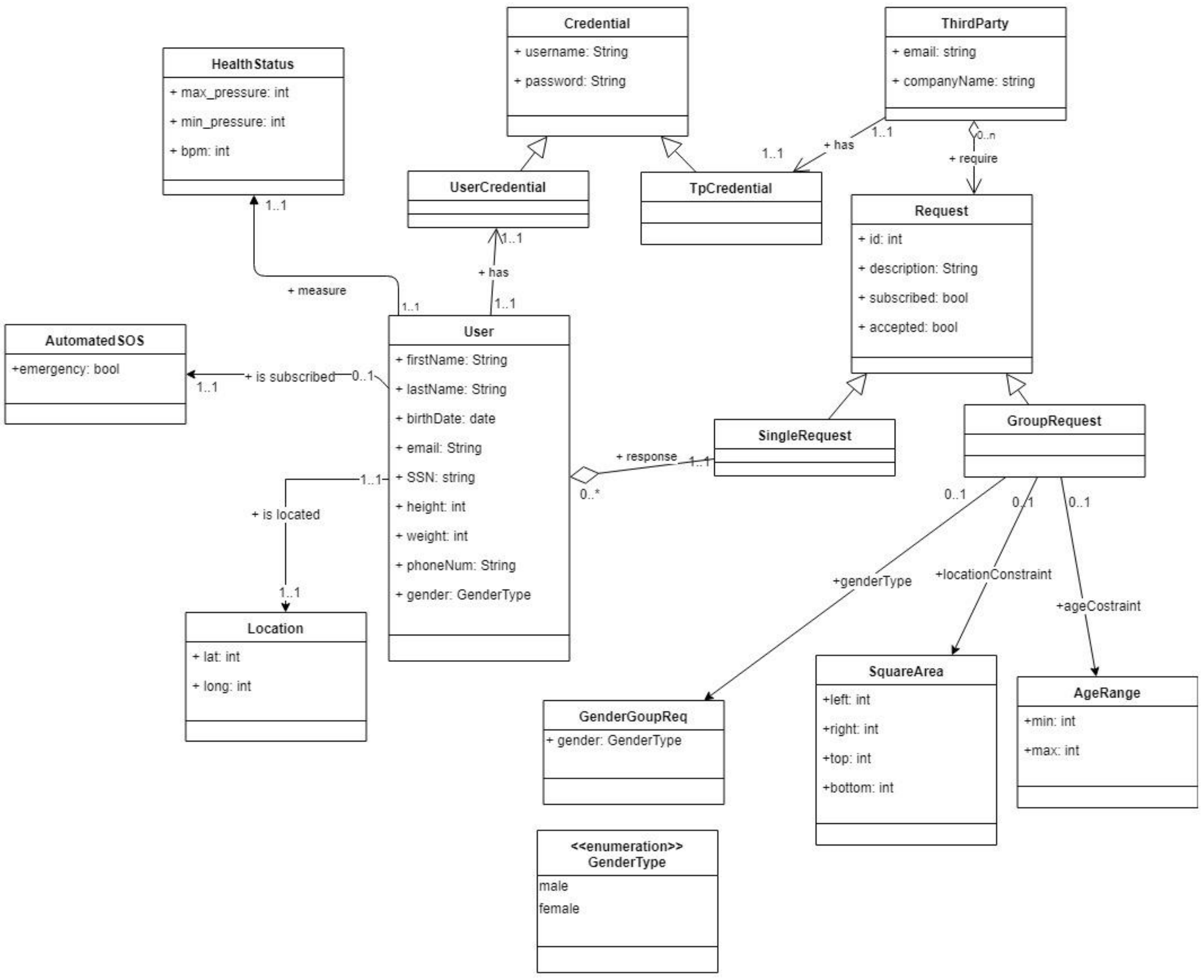
token associate to that user. This token will be subsequently used by the mobile app to interact with the other components of the application server.

- **TpProfileMng**: the component responsible for the authentication request by a Third Party (both registration and log in requests).
  It works the same as the UserProfileMng to provide a token to the web app.
- **RequestMng**: the component responsible for managing requests.
  It co-works with the Database in order to handle each data request made by the Third Parties. More precisely, it deals with sending each new single request to the relative User and validating the group requests. It also deals with notifying the Third Party about the result of the data request, i.e. it notifies if a request has been accepted or not.
  Furthermore, it deals with providing all the requests (pending or accepted request) to each User, and to handle the withdrawing processes.
  It also co-works with the SubscriptionMng component in order to handle the request for which a Third-Party is subscribed.
- **SubscriptionMng**: the component responsible for the subscription.
  It co-works with other three components (RequestMng, UserDataMng and APIsMng) for providing the correct data to the Third-Parties subscribed to some requests.
- **UserDataMng**: the component responsible for the data collection.
  It co-works with the DBMS in order to give its the data collected from the mobile application.
  It communicates with other components which need to operate with these data. More precisely, it co-works with the SosMng component for guarantying the proper functioning of the AutomatedSOS service and with the SubscriptionMng for carrying out the subscription processes.
- **SosMng**: the component responsible for the AutomatedSOS service.
  For each update of an AutomatedSOS User’s health status, this component compares the update data provided by the UserDataMng with some threshold values.
  It provides an interface to the external Ambulance Service to handle the emergencies.
- **APIsMng**: the component responsible for providing data (directly taken from the Database) to the external Third-Parties’ APIs portals.

It also communicates with the SubscriptionMng component in order to send data as soon as they are produced to the Third-Parties subscribed to the requests.

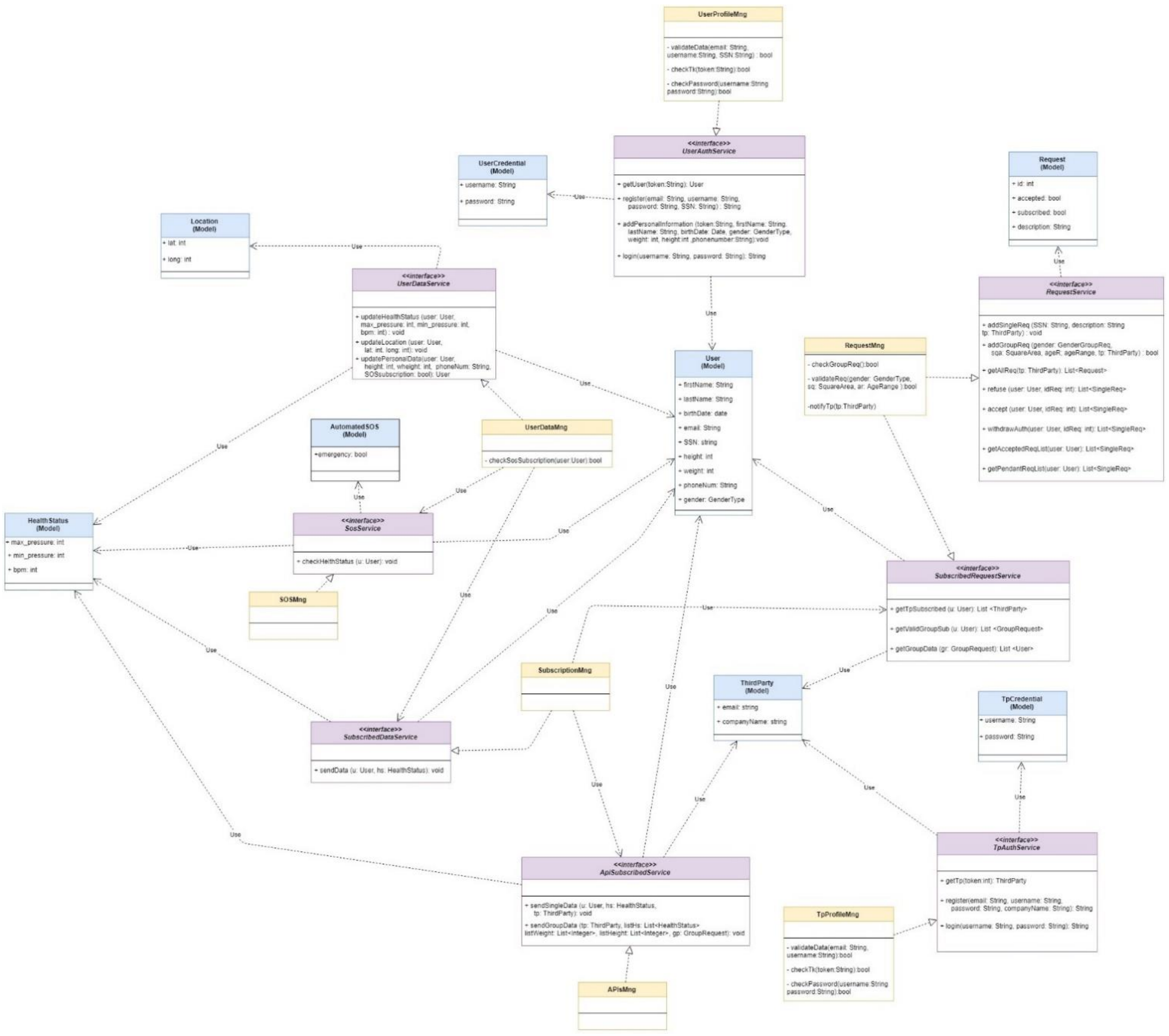
The role of the **Router** is to dispatch each required service to the component involved into the process of that specific service.

In order to better describe the components involved in the application server two other diagrams have been provided as follows.
The first one concerns the Model considered during the development.



Model Class Diagram

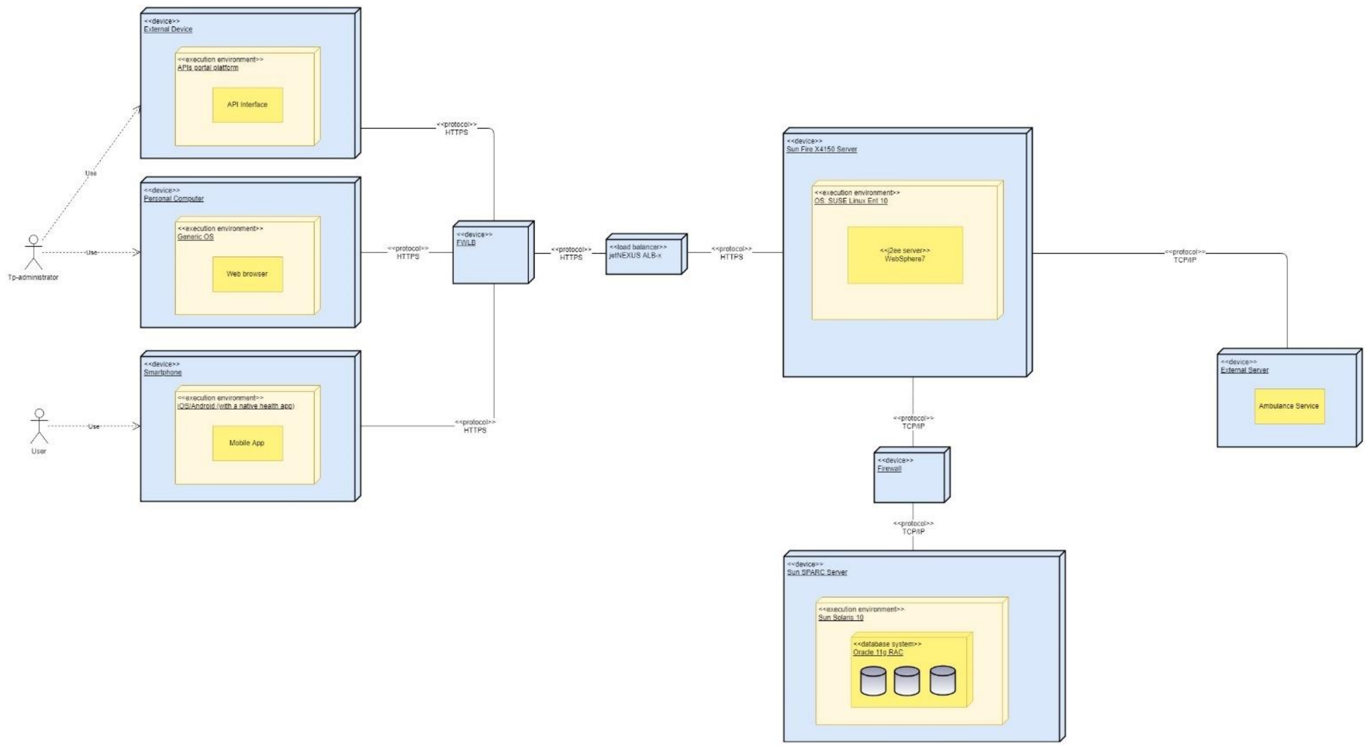
The second one describes the relationship between the services provided by each component and the Model previously described. It is assumed that each service component will be able to invoke operations on the selected database concerning the part of the Model that it is using.


UserProfileMng
- validateData(email: String, username:String, SSN:String) : bool
- checkTk(token:String):bool
- checkPassword(username:String password:String):bool
<<interface>>
UserAuthService
+ getUser(token:String): User
+ register(email: String, username: String, password: String, SSN: String) : String
+ addPersonalInformation (token:String, firstName: String, lastName: String, birthDate: Date, gender: GenderType, weight: int, height:int ,phonenumber:String):void
+ login(username: String, password: String): String
UserCredential (Model)
+ username: String
+ password: String
Location (Model)
+ lat: int
+ long: int
<<interface>>
UserDataService
+ updateHealthStatus (user: User, max_pressure: int, min_pressure: int, bpm: int) : void
+ updateLocation (user: User, lat: int, long: int): void
+ updatePersonalData(user: User, height: int, wheight: int, phoneNum: String, SOSsubscription: bool): User
User (Model)
+ firstName: String
+ lastName: String
+ birthDate: date
+ email: String
+ SSN: string
+ height: int
+ weight: int
+ phoneNum: String
+ gender: GenderType
Request (Model)
+ id: int
+ accepted: bool
+ subscribed: bool
+ description: String
<<interface>>
RequestService
+ addSingleReq (SSN: String, description: String tp: ThirdParty) : void
+ addGroupReq (gender: GenderGroupReq, sqa: SquareArea, ageR: ageRange, tp: ThirdParty) : bool
+ getAllReq(tp: ThirdParty): List<Request>
+ refuse (user: User, idReq: int): List<SingleReq>
+ accept (user: User, idReq: int): List<SingleReq>
+ withdrawAuth(user: User, idReq: int): List<SingleReq>
+ getAcceptedReqList(user: User): List<SingleReq>
+ getPendantReqList(user: User): List<SingleReq>
RequestMng
- checkGroupReq():bool
- validateReq(gender: GenderType, sq: SquareArea, ar: AgeRange):bool
-notifyTp(tp:ThirdParty)
AutomatedSOS (Model)
+emergency: bool
UserDataMng
- checkSosSubscription(user:User):bool
HealthStatus (Model)
+ max_pressure: int
+ min_pressure: int
+ bpm: int
<<interface>>
SosService
+ checkHelthStatus (u: User): void
SOSMng
<<interface>>
SubscribedRequestService
+ getTpSubscribed (u: User): List <ThirdParty>
+ getValidGroupSub (u: User): List <GroupRequest>
+ getGroupData (gr: GroupRequest): List <User>
SubscriptionMng
ThirdParty (Model)
+ email: string
+ companyName: string
TpCredential (Model)
+ username: String
+ password: String
<<interface>>
SubscribedDataService
+ sendData (u: User, hs: HealthStatus): void
<<interface>>
ApiSubscribedService
+ sendSingleData (u: User, hs: HealthStatus, tp: ThirdParty): void
+ sendGroupData (tp: ThirdParty, listHs: List<HealthStatus> listWeight: List<Integer>, listHeight: List<Integer>, gp: GroupRequest): void
APIsMng
<<interface>>
TpAuthService
+ getTp(token:int): ThirdParty
+ register(email: String, username: String, password: String, companyName: String): String
+ login(username: String, password: String): String
TpProfileMng
- validateData(email: String, username:String):bool
- checkTk(token:String):bool
- checkPassword(username:String password:String):bool
Use

Class diagram for Model and Services relationships

2.3.Deployment view

The following diagram illustrates the system’s architecture as deployment and the distribution of processing across the nodes that constitutes the TrackMe project.



As follows all the nodes are better described:

- **Smartphone**: this is the device used by the User to run the mobile application on. The user will be able to communicate with the main Application Server in order to avail of the services it provides.
- **Personal Computer:** this is the device used by the Third-Parties to run the web application on.
  Third-Parties will be able to communicate with the main Application server in order to avail of the services it provides.
- **Application server**: the main logic of the application will be deployed here.
  It communicates with all the other nodes: it gathers information from the external services, handles users’ data and saved them on the DB server.
  As previously mentioned, it takes Customers’ requests (both from Users and Third Parties) and send back responses to them.
- **DB server**: it will store all the persistent data for the users such as credentials data, health status data, etc,
- **External server**: it is the server of the Ambulance Service.

It communicates directly with the Application Server for the AutomatedSOS Services. will only provide services to enrich the application.

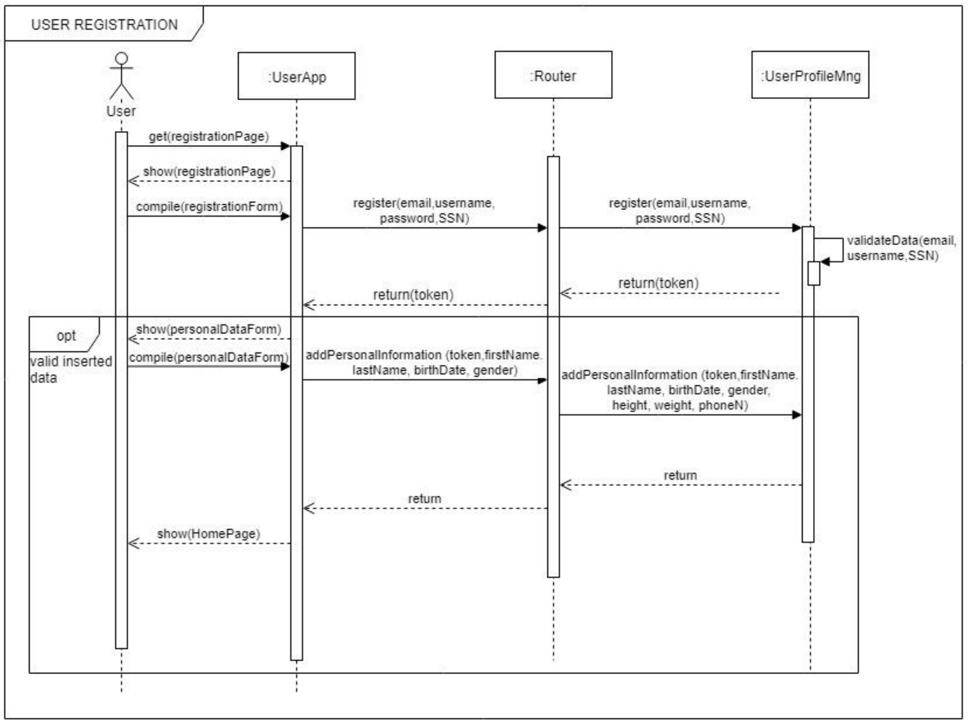
- **External Device:** it is the device used by Third-Parties to access to the required data.
- **Firewall Load Balancing (FWLB):** an array of firewall systems between the Presentation Tier and the Application Server (for how it will be implemented see *section 5.1.3*)
- **Load Balancer**: a device which distributes Customers' requests and network load efficiently (for how it will be implemented see *section 5.1.2*)

2.4.Runtime view

In this section several diagrams are provided to the reader for clarifying on how actions are performed at runtime.
The TrackMe components' interaction will be shown by using sequence diagrams.
Each sequence diagram will be accompanied with a short description in order to help to make it more readable.

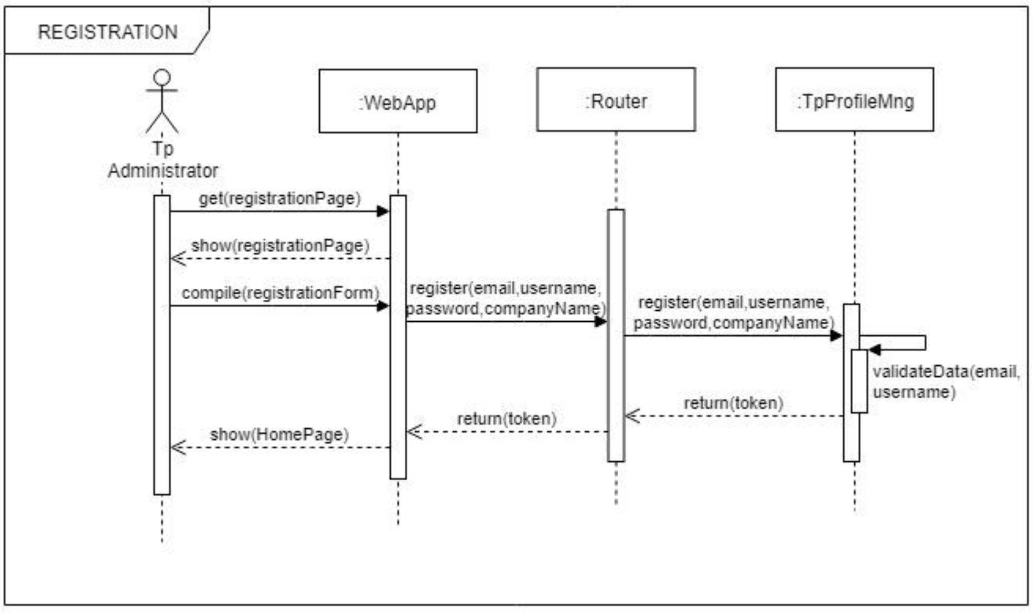
2.4.1. User Registration

The following Sequence Diagram describes the interactions between components to register a new user. First, the user has to insert his credentials which has to be validated to proceed to insert personal information like the name, the birth date and the gender.
A token is provided to the UserApp after the registration completion to remain logged in.


USER REGISTRATION
User
:UserApp
:Router
:UserProfileMng
get(registrationPage)
show(registrationPage)
compile(registrationForm)
register(email,username, password,SSN)
register(email,username, password,SSN)
validateData(email, username,SSN)
return(token)
return(token)
opt
valid inserted data
show(personalDataForm)
compile(personalDataForm)
addPersonalInformation (token,firstName. lastName, birthDate, gender)
addPersonalInformation (token,firstName. lastName, birthDate, gender, height, weight, phoneN)
return
return
show(HomePage)

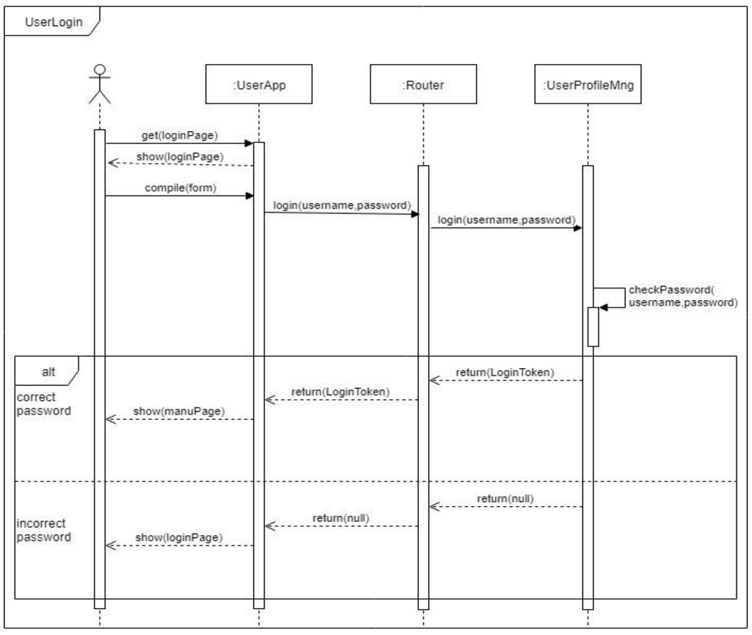
2.4.2. Third Party Registration

The following Sequence Diagram describes the interactions between components to register a new Third Party. The Third-Party Administrator has to insert the credentials which has to be validate because there can't be two Third-Party with the same username or email.
After the registration completion, a token will be provided to the WebApp to remain logged in.


REGISTRATION
Tp Administrator
:WebApp
:Router
:TpProfileMng
get(registrationPage)
show(registrationPage)
compile(registrationForm)
register(email,username, password,companyName)
register(email,username, password,companyName)
validateData(email, username)
return(token)
return(token)
show(HomePage)

2.4.3. User Log In

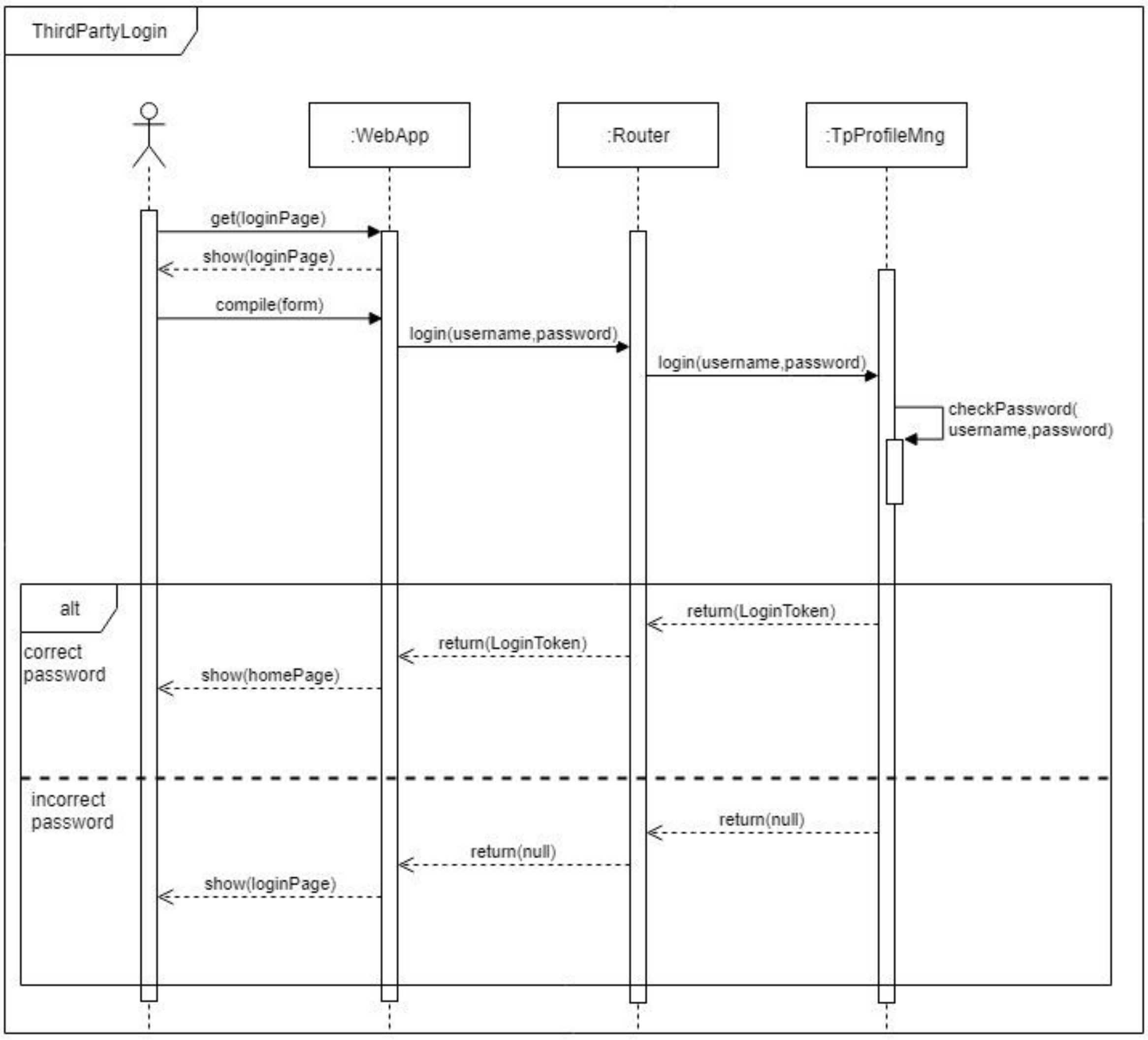
The following sequence diagram describes the steps a user has to take to log into the system. If the inserted password is correct than a token will be provided to the UserApp to carry out requests to the server, otherwise the login page will be reloaded



2.4.4. Third Party Log In

The following sequence diagram describes the steps a Third-party has to take to log into the system.

If the inserted password is correct than a token will be provided to the WebApp to carry out requests to the server, otherwise the login page will be reloaded

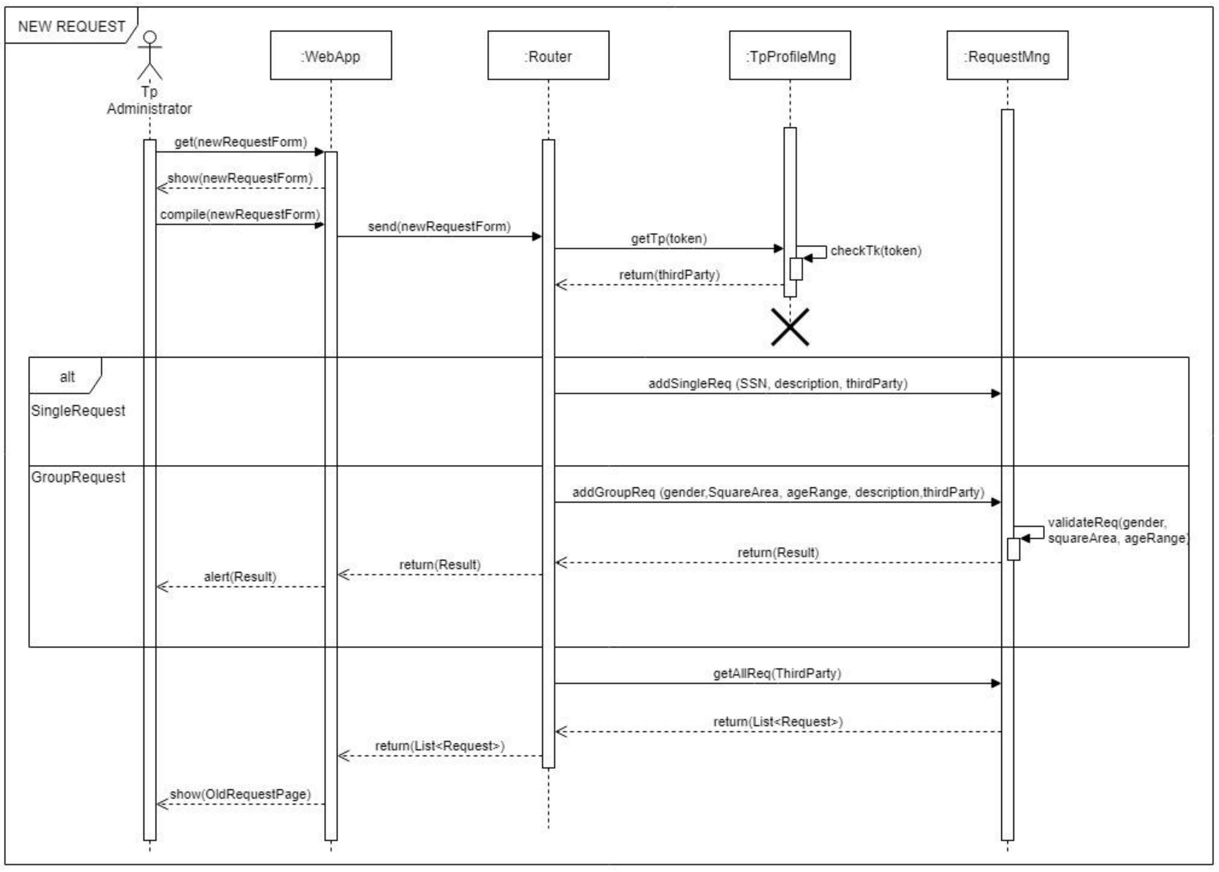


2.4.5. New Data Request from a Third Party

The following sequence diagram describes the steps a Third-Party administrator has to take to make new requests.

The acceptance of a new group request will be immediately shown.

When the request has been successfully made, the WebApp will show the old requests page to the administrator.

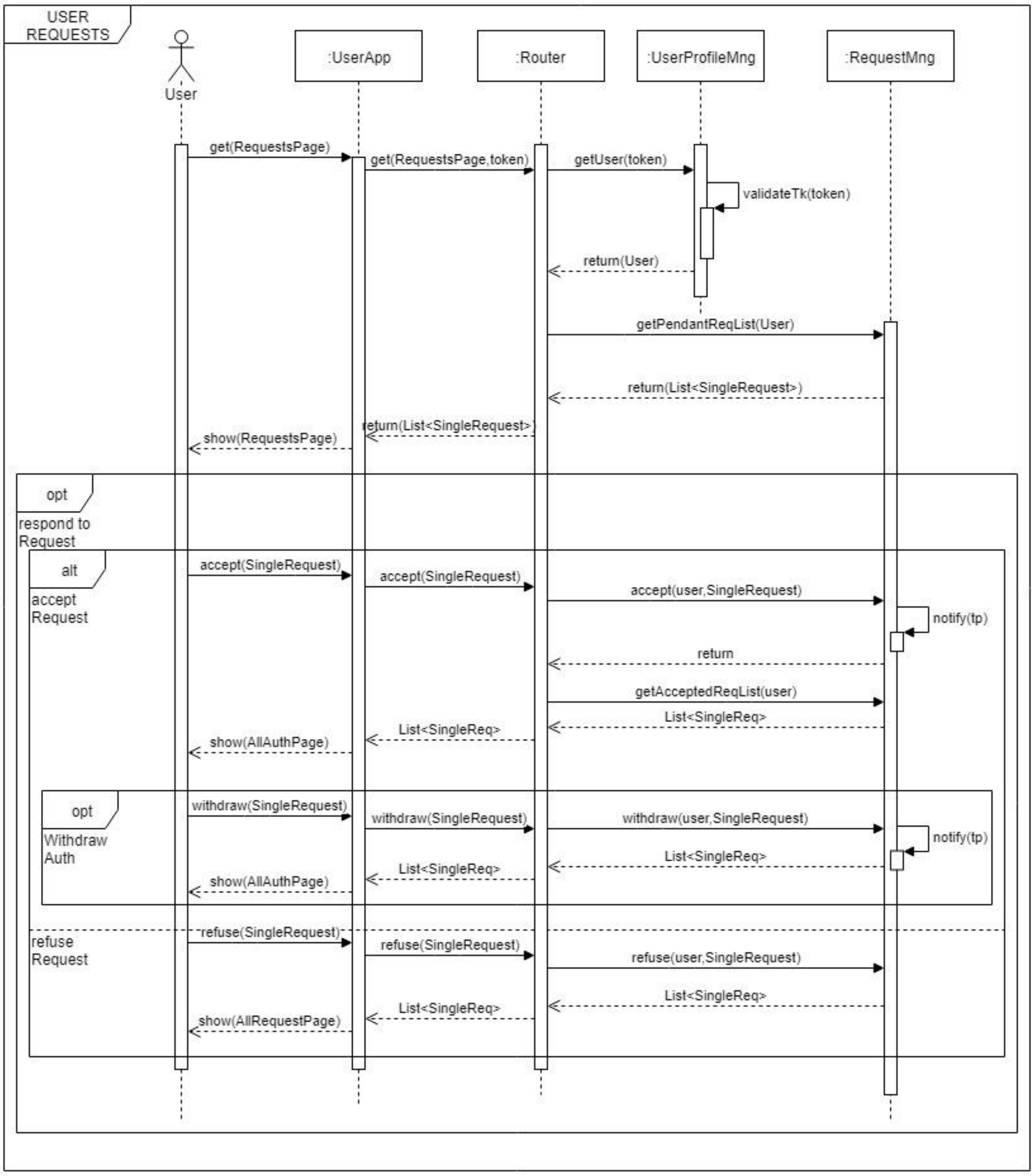


2.4.6. Management of the requests by the Users

The following diagram describe the steps a user has to take to see all the requests sent by Third Parties and interact with them.

After a User accepts a request, the page with all the accepted requests will be shown and the interested Third-Party will be notified by email.

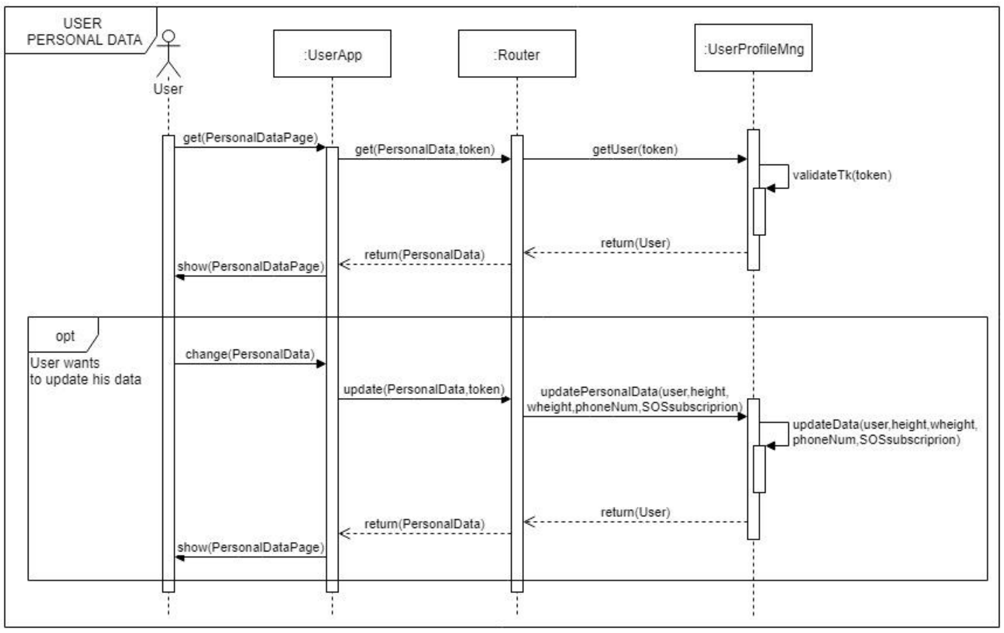
In this page the User can withdraw the authorization to any already accepted request and the involved Third-Party will be notified by email.



2.4.7. User Update Personal Data

The following sequence diagram describes every step each component takes when a user wants to view and update his personal data.

This process involves only the Router and the UserProfileMng component which update the new data into the database.



2.4.8. Health Status Updates and relative Data Flows

The following sequence diagram describes the interactions between components when new health status data of a user arrive into the server.

The new data arrives with the login token which has to be validate by the UserProfileMng component. This component will identify the corresponding User object.

After that, the current location is required to the UserApp and both the health status and the location will be updated by the UserDataMng component.

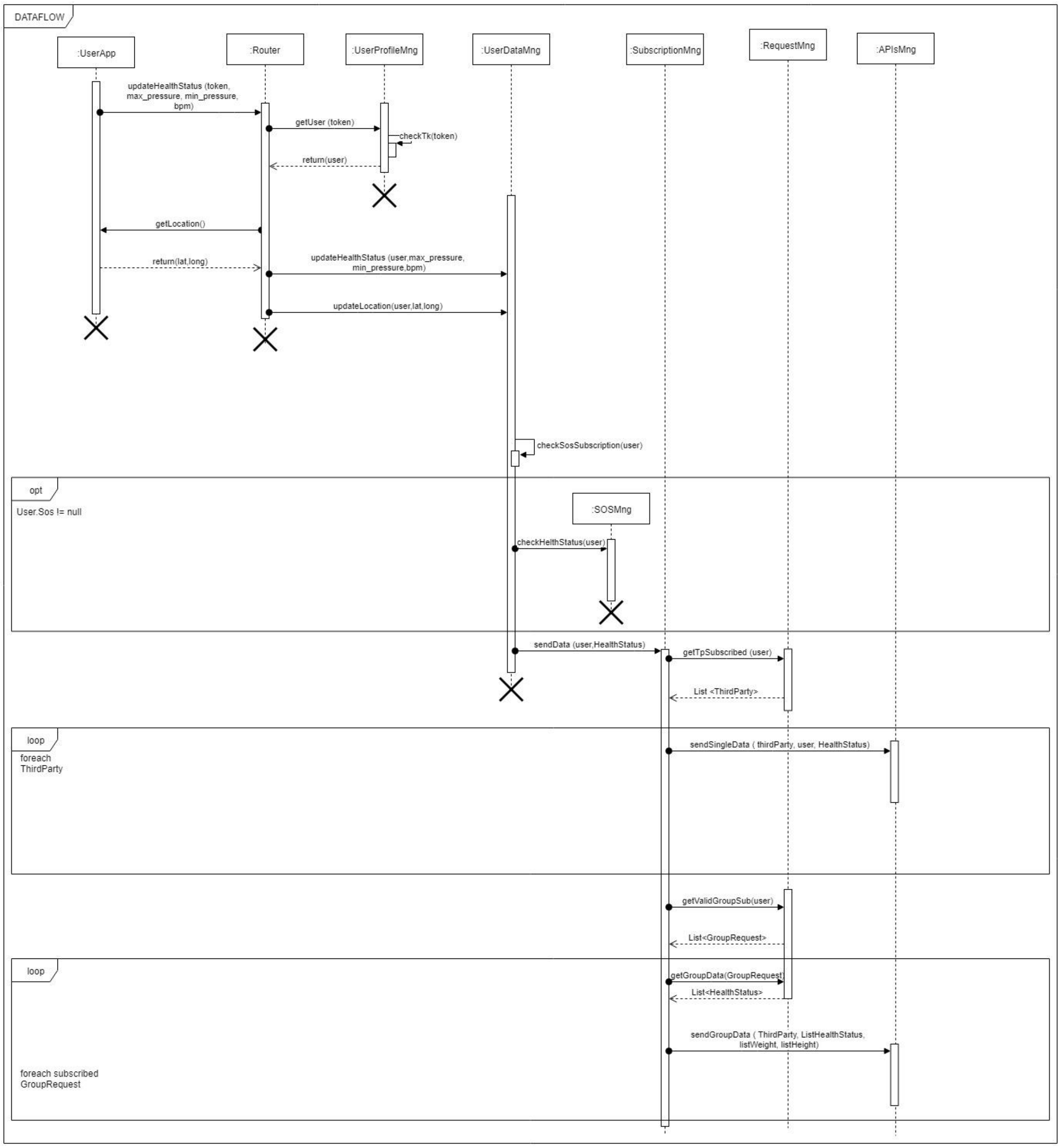
If the User is subscribed to the AutomatedSOS service, the new data will be sent to the SosMng component which will check them and alert the Ambulance Service if these data are below the thresholds.

Then the new data will be sent to the SubscriptionMng component.

First this component will interact with the RequestMng component to get all the Third-Parties subscribed to the User and will send them all the data using the APIsMng component.

After that it will interact with the RequestMng component to get all the group requests interested in his data and to which the Third-Parties are subscribed.

For each request it will ask the RequestMng component to get the group of data and send them to the associated Third-Party with the APIsMng component

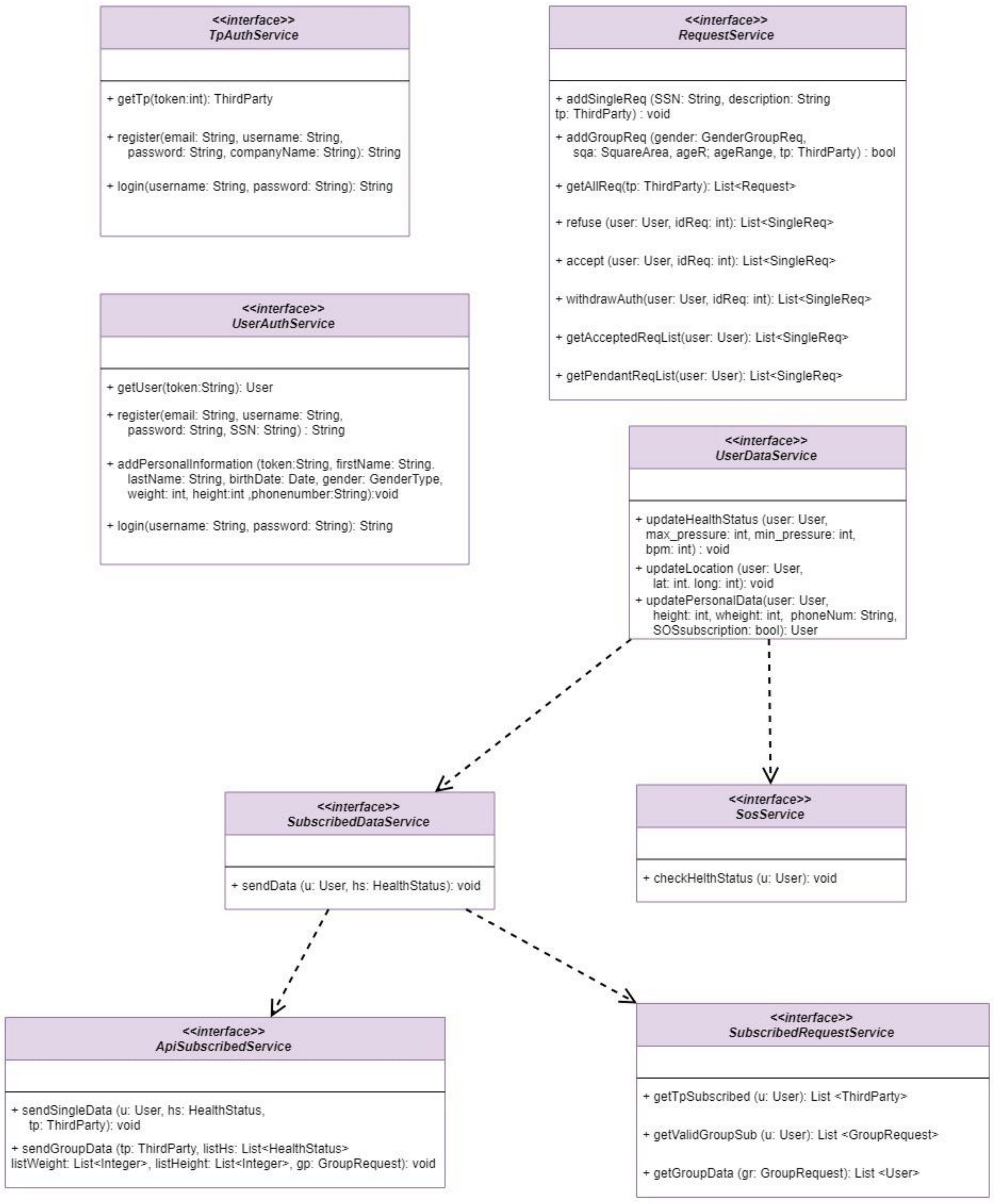

DATAFLOW
:UserApp
:Router
:UserProfileMng
:UserDataMng
:SubscriptionMng
:RequestMng
:APIsMng
updateHealthStatus (token, max_pressure, min_pressure, bpm)
getUser (token)
checkTk(token)
return(user)
getLocation()
return(lat,long)
updateHealthStatus (user,max_pressure, min_pressure,bpm)
updateLocation(user,lat,long)
checkSosSubscription(user)
opt
User.Sos != null
:SOSMng
checkHelthStatus(user)
sendData (user,HealthStatus)
getTpSubscribed (user)
List <ThirdParty>
loop
foreach ThirdParty
sendSingleData (thirdParty, user, HealthStatus)
getValidGroupSub(user)
List<GroupRequest>
loop
getGroupData(GroupRequest)
List<HealthStatus>
sendGroupData (ThirdParty, ListHealthStatus, listWeight, listHeight)
foreach subscribed GroupRequest

2.5.Component interfaces

The following diagram is meant to better describe the component interfaces provided by the components of the Application Server, previously presented in the component diagram (*see section 2.2*).

Moreover, this diagram is used to show more clearly the dependencies between the parts of the application server, previously presented in the *second class-diagram of the section 2.2.*

Finally, there will be a description in order to clarify the role of each interface and what services it provides.


<<interface>>
TpAuthService
+ getTp(token:int): ThirdParty
+ register(email: String, username: String, password: String, companyName: String): String
+ login(username: String, password: String): String
<<interface>>
RequestService
+ addSingleReq (SSN: String, description: String tp: ThirdParty) : void
+ addGroupReq (gender: GenderGroupReq, sqa: SquareArea, ageR; ageRange, tp: ThirdParty) : bool
+ getAllReq(tp: ThirdParty): List<Request>
+ refuse (user: User, idReq: int): List<SingleReq>
+ accept (user: User, idReq: int): List<SingleReq>
+ withdrawAuth(user: User, idReq: int): List<SingleReq>
+ getAcceptedReqList(user: User): List<SingleReq>
+ getPendantReqList(user: User): List<SingleReq>
<<interface>>
UserAuthService
+ getUser(token:String): User
+ register(email: String, username: String, password: String, SSN: String) : String
+ addPersonalInformation (token:String, firstName: String. lastName: String, birthDate: Date, gender: GenderType, weight: int, height:int ,phonenumber:String):void
+ login(username: String, password: String): String
<<interface>>
UserDataService
+ updateHealthStatus (user: User, max_pressure: int, min_pressure: int, bpm: int) : void
+ updateLocation (user: User, lat: int. long: int): void
+ updatePersonalData(user: User, height: int, wheight: int, phoneNum: String, SOSsubscription: bool): User
<<interface>>
SubscribedDataService
+ sendData (u: User, hs: HealthStatus): void
<<interface>>
SosService
+ checkHelthStatus (u: User): void
<<interface>>
ApiSubscribedService
+ sendSingleData (u: User, hs: HealthStatus, tp: ThirdParty): void
+ sendGroupData (tp: ThirdParty, listHs: List<HealthStatus> listWeight: List<Integer>, listHeight: List<Integer>, gp: GroupRequest): void
<<interface>>
SubscribedRequestService
+ getTpSubscribed (u: User): List <ThirdParty>
+ getValidGroupSub (u: User): List <GroupRequest>
+ getGroupData (gr: GroupRequest): List <User>

- **UserAuthService**

  This interface is implemented by the <u>UserProfileMng</u> component and provide the functionalities to authenticate a User.

  - **register (email: String, username: String, password: String, SSN: String): String**

    This method registers a new user into the system and validates the fields which can't be used for other users. It returns the token to keep the User logged in.
  - **addPersonalInformation (token:String firstName: String. lastName: String, birthDate: Date, gender: GenderType, weight: int, height:int , phoneNumber: String):void**

    This method uses the registration token to add the mandatory data of the user into the system.

    The last three fields can be update later at any time.
  - **login (username: String, password: String): String**

    This method is used to log in a User with his credentials. It returns the token to keep the user logged in.
  - **getUser (token:String): User**

    This method is used to get the User object from the corresponding token.

- **TpAuthService**

  This interface is implemented by the <u>TpProfileMng</u> component and provide the functionalities to authenticate a Third-Party.

  - **register (email: String, username: String, password: String, companyName: String): String**

    This method registers a new Third-Party into the system, it also validates the fields which can't be used for other Third-Parties. It returns the token to keep the Third-Party logged in.
  - **login (username: String, password: String): String**

    This method is used to log in a Third-Party with his credentials. It returns the token to keep the Third-Party logged in.
  - **getTp(token:int): ThirdParty**

    This method is used to get the Third-Party object from the corresponding token.

- **UserDataService**

  This interface is implemented by the UserDataMng component and provide the functionalities to update Users' data.

  - **updateHealthStatus (user: User, max_pressure: int, min_pressure: int, bpm: int): void**

    This method is used to update the Health Status of the User.

    It notifies the other components about the new data arrived.
  - **updateLocation (user: User, lat: int. long: int): void**

    This method is used to update the location of the user once new data arrives.
  - **updatePersonalData(user: User, height: int, weight: int, phoneNum: String, SOSsubscription: bool): User**

    This method is used to update User's personal data.

- **RequestService**

  This interface is implemented by the RequestMng component and provide the functionalities to interact with the requests.

  - **addSingleReq (SSN: String, description: String, tp: ThirdParty): void**

    This method is used to add a new single request from a Third-Party.
  - **addGroupReq (gender:GenderGroupReq, sqa:SquareArea, ageR;ageRange, tp: ThirdParty): bool**

    This method is used to add a group request from a Third-Party.

    It returns the acceptance of the request.
  - **getAllReq (tp: ThirdParty): List<Request>**

    This method is used to get all the requests made by a specific Third-Party
  - **refuse (user: User, idReq: int): List<SingleReq>**

    This method is used by a User to refuse a specific request.

    It returns the list of all the User's pending requests.
  - **accept (user: User, idReq: int): List<SingleReq>**

    This method is used by a User to accept a specific request.

    It returns the list of all the accepted requests of the user.
  - **withdrawAuth (user: User, idReq: int): List<SingleReq>**

    This method is used by a user to withdraw a specific accepted request.

    It returns the list of all the accepted requests of the user.
  - **getAcceptedReqList (user: User): List<SingleReq>**

This method is used to get the list of all the accepted request of a specific user.

- **getPendantReqList (user: User): List<SingleReq>**
  This method is used to get the list of all the pendant request of a specific user.

- **SubscribedRequestService**
  This interface is implemented by the RequestMng component. It provides the functionalities to get information about subscribed requests.
  - **getTpSubscribed (u: User): List <ThirdParty>**
    This method is used to get the list of all the Third Parties subscribed to the new data of a specific user.
  - **getValidGroupSub (u: User): List <GroupRequest>**
    This method is used to get the list of all the GroupRequests for which a specific user is in
  - **getGroupData (gr: GroupRequest): List <User>**
    This method is used to get the list of data from a group request.

- **SubscribedDataService:**
  This interface is implemented by the SubscriptionMng and provide the functionalities to notify the arrive of new data.
  - **sendData (u: User, hs: HealthStatus): void**
    Notify the arrive of new health status data of the specific user

- **ApiSubscribedService:**
  This interface is implemented by the APIsMng component and provide the functionalities to send new data to a Third-Party.
  - **sendSingleData (u: User, hs: HealthStatus,tp: ThirdParty): void**
    This method is used to send a user data and health status to a third party subscribed to it.
  - **sendGroupData (tp: ThirdParty, listHs: List<HealthStatus> listWeight: List<Integer>, listHeight: List<Integer>, gp: GroupRequest): void**
    This method is used to send a set of data and health status to a third party subscribed to the corresponding group request.

- **SosService:**
  This interface is implemented by the SosMng component and provides the functionalities to notify the arrive of new data of a user subscribed to the AutomatedSos service.
  - **checkHelthStatus (u: User): void**
    This method is used to check the health status of a specific user.

2.6.Selected architectural styles and patterns

2.6.1. Overall architecture

The most suitable architecture for TrackMe would be three tier architecture.
A three-tier architecture is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on separate platforms.
Three-tier architecture is a software design pattern and a well-established software architecture that enable the distribution of application functionality across three independent systems, typically:

- Client components running on local workstations (tier one)
- Processes running on remote servers (tier two)
- A discrete collection of databases, resource managers, and mainframe applications (tier three)

More precisely:

- **Presentation Tier (first tier):** Responsibility for presentation and user interaction resides with the first-tier components. These client components enable the user to interact with the second-tier processes in a secure and intuitive manner through a set of REST API calls.
  In the TrackMe system the presentation tier is a graphical user interface accessible either through a mobile application by individual Users and through a web app by Third-Parties.
  Furthermore, the mobile application communicates with a native health app in order to collect constantly health status values, while the web app use the APIs provided by a Maps Platform.

- **Business Logic Tier (second tier):** it manages the business logic of the application, and permits access to the third-tier services. The application logic layer is where most of the processing work occurs. Multiple client components can access the second-tier processes simultaneously, so this application logic layer must manage its own transactions.
  Without an application logic layer, client components access the database directly. The database is required to manage its own connections, typically locking out a record that is being accessed.
  Separating the second and third tiers reduces the load on the third-tier services, supports more effective connection management, and can improve overall network performance.
  In TrackMe system this tier communicates also with external services (Ambulance Service) for guarantying the AutomatedSOS service and with the Third-Parties' APIs portals for sending the required data.
- **Data Persistence Tier (third tier):** The third-tier services are protected from direct access by the client components residing within a secure network. Interaction must occur through the second-tier processes.

There are many benefits to using a 3-layer architecture including speed of development, scalability, performance, and availability. As mentioned, modularizing different tiers of an application gives development teams the ability to develop and enhance a product with greater speed than developing a singular code base because a specific layer can be upgraded with minimal impact on the other layers.
Scalability is another great advantage of a 3-layer architecture. By separating out the different layers you can scale each independently depending on the need at any given time. This allows to load balance each layer independently, improving overall performance with minimal resources. Additionally, the independence created from modularizing the different tiers provides many deployment options.
As already mentioned, data corruption through client applications can be eliminated since the data passed in the middle tier for database updates ensures its validity. Moreover, the implementation of several layers makes the data more secure. As clients do not interact with the database directly, it provides less risk and confliction with unauthorized data.
Finally, the actual structure of the database often remains hidden from requesters enabling any change of the database to be transparent. Thus, a process in the middle tier

which exchanges data with other applications can sustain its current interface while a modification of the underlying database structure

2.6.2. Design patterns

- **Proxy Pattern**

  It is a structural design pattern that provides a substitute or placeholder for another object. The router component functions as a proxy which controls access to the original objects, allowing to perform something either before or after the request gets through to them, represented in the TrackMe system by the other components.

  Benefits:

  - It controls the service objects without clients knowing about it
  - It can manage the lifecycle of the service objects
  - It increases the performance of the application, by avoiding duplication of objects which might be huge size and memory intensive
  - It enhances scalability, since it is possible to introduce new proxies without changing the services or clients

- **Event-driven architecture style**

  This architecture consists of event producers that generate a stream of events, and event consumers that listen for the events.

  Events are delivered in real time, so consumers can respond immediately to events as they occur.

  In our scope this architecture is used to handle the sending in real time of new produced data to the subscribed Third-Parties.

  TrackMe system uses the **pub/sub** model for this architecture. When an event is published, it sends the event to each subscriber. After an event is received, it cannot be replayed, and new subscribers do not see the event.

  More precisely, in TrackMe:

  - The event is the update of the health status of a specific User
  - The event producer is the mobile application
  - The event consumers are the Third-Parties' APIs portals

  Benefits:

  - Producers and consumers are decoupled.

  - No point-to point-integrations. It's easy to add new consumers to the system.
  - Consumers can respond to events immediately as they arrive.
  - Highly scalable and distributed.
  - Subsystems have independent views of the event stream.

2.6.3. Other Design Decisions

The web application needs an integration with a map service in order to allow Third-Parties to select the geographical area in group requests. It will be integrated using the APIs provided by the Maps platform.
The mobile needs an integration with a native health app installed on the smartphone in order to allows Users' health status collection.

Others design pattern widely recommended:

- **Model View Controller (MVC)**
  For the implementation of the mobile application and the web application is recommended to adopt this commonly architectural pattern which is used for developing user interfaces that divides an application into three interconnected parts. The mainly benefits are:
  - **Faster development process:** MVC supports rapid and parallel development.
  - **Ability to provide multiple views:** In the MVC Model, you can create multiple views for a model. Code duplication is very limited in MVC because it separates data and business logic from the display.
  - **Modification does not affect the entire model:** Modification does not affect the entire model because model part does not depend on the views part. Therefore, any changes in the Model will not affect the entire architecture.

3. User interface design

The mock-ups for this application were presented in the RASD Document in section “3.1.1. User Interfaces”.

At this point, there are no new functionalities that can be presented with new mock-ups for the application.

4. Requirements traceability

The design of this application aims to meet all the goals and requirements that have been previously specified in RASD document.

Below are listed the design components to which TrackMe requirements and goals are mapped:

- **[G1]**: Allow individuals to become registered users of Data4Help (requirements **[R1]**, **[R2]**, **[R3]**):
  - UserProfileMng
- **[G2]**: Allow users to sign up for AutomatedSOS service (requirements **[R4]**, **[R5]**):
  - UserProfileMng
- **[G3]**: Provide the registration to third parties who want access to users’ data (requirements **[R6]**):
  - TpProfileMng
- **[G4.1.]**: Give third parties access to data of a specific user (requirements **[R7]**, **[R8]**):
  - RequestMng
  - APIsMng
- **[G4.2]**: Give third parties access to anonymized data of group of users (requirements **[R9]**, **[R10]**):
  - RequestMng
  - APIsMng
- **[G5]**: Allow users to accept or refuse the requests from third-parties to access their own data and their location (requirements **[R11]**):
  - RequestMng
- **[G6]**: Allow third parties to subscribe to new data and to receive them as soon they are produced (requirements **[R12]**, **[R13]**):
  - UserDataMng
  - SubscriptionMng
  - RequestMng
  - APIsMng

- **[G7]**: Allow customers to insert or update their own personal data and information about their body measurements (e.g. weight, height, etc.) (requirements **[R14]**):
    - UserProfileMng
- **[G8]**: If some parameters of health status are below certain threshold, send an ambulance to the user location, with a reaction time less than 5 seconds (requirements **[R15]**, **[R16]**):
    - UserDataMng
    - SosMng
- **[G9]**: Allow users to keep track of their health status at any time (requirements **[R17]**):
    - UserDataMng
- **[G10]**: Allow users to withdraw the authorisation to third parties to access their data (requirements **[R18]**, **[R19]**):
    - RequestMng

5. Implementation, integration and test plan

5.1. Implementation plan

5.1.1. Introduction

The purpose of this section is to describe how the implementation plan process has been designed in order to widely determine the project success and ensure that the strategic goals are actionable.
Next sections will explain the order in which the implementation will be carried out and the implementation strategies adopted.

5.1.2. Implementation Schedule

Before starting with the implementation of the whole TrackMe project, it is assumed that the DBMS in the DataBase Server component is almost entirely developed, since it represents a crucial point in the implementation process.
The starting point will be the implementation of the Model in order to eliminate potential integration problems, since all the Server components use some Model objects.
After that, the implementation steps will focus on the Client and Server sub-systems. These two macro-components will be developed in parallel adding step-by-step on the client side the management of each function implemented in the Server side. This ensures higher reliability and facilitates the integration process.
As previously mentioned in *section "Other design patterns" (2.6.3)* the client application (both mobile app and web app) will be developed following the MVC pattern, where:

- The model consists of the data that the User or the Tp-a needs to interact with.
- The view represents the user interface which will be update by the model.
- The controller deals with interacting with the Server in order to carry out Clients' operations.

On the other side, for the application Server a mix of top-down and bottom-up approaches will be used.
More precisely, the components for which the features they provide are completely independent from each other are:

- UserProfileMng
- TpProfileMng
- APIsMng

- SosMng
- RequestMng

These components will be implemented in parallel using the bottom-up approach, by which the components are almost fully developed and tested separately and subsequently integrated with each other.

While, a top-down approach better suits the components for which the functioning as a whole is the first priority.
Since the remain components interface with some of the others above described, the implementation process of the following components will start from the development of the functionalities needed to carry out the co-working between them.
These components are:

- UserDataMng
- SubscriptionMng

This permits to proceed hand in hand with the implementation, the testing and the integration processes.

The last component to be developed will be the Router.
As widely described in the previous sections, its main function is to offer to the Clients a way to communicate with the modules and services which will be carrying out by the Server.
Since it represents a critical point for the efficient and reliable functioning of the Server, it will be implemented in a distributed way, by running simultaneously the main role on several independent devices.
The communication between the Client and the distributed router system will be handled by a load balancing algorithm.
The benefits of Load Balancing are to provide scalability, optimize service reliability and availability and increase overall manageability.

5.1.3. Security and Privacy

To ensure the security of the TrackMe system every connection with the Clients will be filtered by a Load Balancer Firewall.
Firewall Load Balancing is a deployment architecture where multiple firewall systems are placed behind Server Load Balancers. Network traffic through the firewall systems

is load balanced to the group of firewalls providing a scalable and highly available security infrastructure.
A Firewall Load Balancing is an array of firewall systems which are configured in a load balanced configuration.
This solution ensures:

- Scalability: additional firewalls can be added dynamically to increase capacity. Firewall capacity can be added live without affecting the existing firewall systems
- Reliability: when multiple firewalls are load balanced, any single firewall failure does not cause serious outages
- Maintainability: firewall maintenance is easier in the load balanced environments than in the non-load balanced ones, where changing security policies can easily cause unforeseen issues and outages.

Finally, a simple firewall device will be implemented between the Server and the DataBase for ensuring the correct security of the data flow.

5.2. Integration and testing

5.2.1. Entry Criteria

Some critical conditions have to been verified before starting with the integration of components and its testing.
First of all, the external services and their APIs that are going to be used in the application should be available and ready. This applies also to the DBMS and the DataBase Server on which it will be running on.
Moreover, the components for which a bottom-up approach in the implementation plan has been used have to pass completely the unit tests in order to ensure the proper functioning of the components themselves. In this way, if the integration test fails, the problem will need to be found in the integration itself.
For these components the code and the corresponding functional test cases will be produced simultaneously in order to find out possible code flaws as soon as possible.

5.2.2. Components to be integrated

As widely described in previous sections, the TrackMe system is composed of several macro-components.

For a more efficient integration process, it can be grouped in four sections:

1. **Integration of components with the DBMS**

   This integration is meant for the components of the application Server which use the DataBase Server to carry out their functionalities. These components are:

   - UserProfileMng
   - TpProfileMng
   - RequestMng
   - APIsMng
   - UserDataMng

2. **Integration of components with the external services**

   In this group, we include the integration of the TrackMe components which use an external service. These components are:

   - UserApp has to be integrated with the Native Health app
   - TpWebApp has to be integrated with the Maps APIs Platform
   - SosMng has to be integrated with the Ambulance Service interface

3. **Integration of the components of the application server**

   This group concerns the integration between the application server components. These components are:

   - SubscriptionMng with RequestMng
   - UserDataMng with SubscriptionMng
   - SubscriptionMng with APIsMng
   - UserDataMng with SosMng

   NOTE: *In the list above, the first component uses the interface implemented by the second one.*

4. **Integration of the client and the application server**

   The integration necessary for carrying out the communication between the client and the server components.

   More precisely:

   - the mobile application will be integrated with UserProfileMng, UserDataMng and RequestMng
   - the web application will be integrated with TpProfileMng and with RequestMng

5.2.3. Integration Testing Strategy

Considering the implementation plan and the overall architecture of the TrackMe system, a Bottom-up approach is the most suitable solution.
This strategy starts from the lowest or the innermost unit of the application, and gradually progresses towards the upper modules of the application.
The integration will continue until all the modules are integrated and the entire application is tested as a single unit.
The advantage of this approach is that, if a major fault exists at the lowest unit of the program, it is easier to detect it, and corrective measures can be taken

Each integration in the same group-level defined in *section 5.2.2* is independent and there is no specific order in which to complete them. In this way, the integration process and its testing are more flexible and efficient.

5.2.4. Functional System Testing

Finally, once the integration has been completed, additional test cases are considered for functional system testing.
The main object of the system testing is to check the expected behaviour of the system as a whole.
It will be performed on a complete system to verify that if it works as expected once all the components are integrated.
Two functional system testing techniques will be mainly adopted:

- **Decision-based Tests**
  These tests are focused on the possible outcomes of the system when a particular condition is met (for example the system behaviour after inserting wrong credentials)
- **Boundary Value Tests**
  These tests imply data limits to the application and validate how it behaves (for example for testing the AutomatedSOS emergency)

Finally, when most of the bugs are uncovered through the above techniques, **ad-hoc tests** will be used to uncover any discrepancies not observed earlier. These are performed with the mindset of breaking the system and see if it responds gracefully.

6. Effort Spent

Section	**Student**	**Hours of work**
Introduction	Enrico Voltan	1
Introduction	Antonio Urbano	3
Architectural Design	Enrico Voltan	17
Architectural Design	Antonio Urbano	14
Requirements Traceability	Enrico Voltan	1
Requirements Traceability	Antonio Urbano	1
Implementation, Integration and Test Plan	Enrico Voltan	9
Implementation, Integration and Test Plan	Antonio Urbano	9

7. References

- Specification document *"A.Y. 2018-2019 Software Engineering 2 Mandatory Project: goal, schedule, and rules"*
- RASD documents (version 1.0 and 1.1)
- Old Design Documents: "*DD to be analysed*" and more previous DDs.
- Slides provided on Beep channel
- Architecture styles: "https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/"