Software Engineering II mandatory project
A.Y. 2018/2019

TrackMe

DD

Version: 1.0
Release date: 10/12/2018

Submitted by:
Claudia Conchetto
Riccardo Corona
Andrea Crivellin

Contents

1 Introduction

1.1 Purpose

This document is intended to provide a deeper functional description of the TrackMe system-to-be by giving technical details and describing the main architectural components as well as their interfaces and their interactions. The relations among the different modules are pointed out using UML standards and other useful diagrams showing the structure of the system.

The document aims to guide the software development team to implement the architecture of the project, by providing a stable reference and a single vision of all parts of the software itself and clearly defining how they work. It also presents in more details the implementation and integration plan, as well as the testing plan.

1.2 Scope

The system aims to support TrackMe services: Data4Help, AutomatedSOS and Track4Run.

The system is structured in a three-layered model, which will be thoroughly described in this document, that adapts to several forms of clients: various types of actors that interact with the system-to-be by generating a client-server dualism, hence a flow of requests-responses.

The architecture must be designed with the intent of being maintainable and extensible, also foreseeing future changes. This document aims to drive the implementation and testing phase so that cohesion and decoupling are increased as much as possible. In order to do so, individual components must not include too many unrelated functionalities and reduce interdependency between one another.

Specific architectural styles and design patterns will be followed in this document and used for future implementation, as well as common design paradigms that combine useful features of said concepts.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Relational data**: data structured according to the relational model.
- **NoSQL data**: data not structured according to the relational model.

1.3.2 Acronyms

- RASD – Requirement Analysis and Specification Document
- DD - Design Document
- API - Application Programming Interface
- REST - REpresentational State Transfer
- HTTPS - HyperText Transfer Protocol over Secure Socket Layer

- SDK - Software Development Kit
- GPS - Global Positioning System
- DBMS - DataBase Management Server
- RDBMS - Relational DataBase Management Server
- ACID - Atomicity, Consistency, Isolation and Durability
- CRUD - Create, Read, Update, Delete (four basic functions of persistent storage)

1.3.3 Abbreviations

- [Gn]: n-th goal
- [Rn]: n-th functional requirement

1.4 Document structure

The document is composed of 7 sections.

Section 1 - Introduction: This section 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.

Section 2 - Architectural design: This section shows the main system components together with sub-components and their relationship. This section is divided into different parts whose focus is mainly on design choices, interactions, architectural styles and patterns.

Section 3 - User Interface: This section is strongly dependant from the section 3 of the RASD, in which mockups are already shown.

Section 4 - Requirements traceability: This section shows how all the functional requirements previously defined in the RASD document are satisfied by architectural choices.

Section 5 - Implementation, integration and test plan: This section identifies the order in which it is planned to implement the subcomponents of the system and the order in which it is planned to integrate such subcomponents and test the integration.

Section 6 - Effort spent: This section shows the effort spent by each group member while working on this document.

Section 7 - References: This section includes the reference documents.

2 Architectural design

2.1 Overview

This section provides a detailed view over the system architecture and its components, describing them at both logical and physical level.

Section 2.2 - High-level components: This subsection provides a description of high-level components and their interactions.

Section 2.3 - Component view: This subsection provides a detailed insight of the components described in the previous section.

Section 2.4 - Deployment view: This subsection provides a set of indications on how to deploy the illustrated components on physical tiers.

Section 2.5 - Runtime view: In this subsection sequence diagrams are used to describe the way components interact to accomplish specific tasks typically related to your use cases.

Section 2.6 - Component interfaces: This subsection provides a description of the different type of interfaces among the various described components.

Section 2.7 - Selected architectural styles and patterns: This subsection provides a list of the architectural styles, design patterns and paradigms adopted in the design phase.

Section 2.8 - Other design decisions: This subsection lists of all other relevant design decisions that were not mentioned before.

2.2 High-level components

The main high-level components of the system are:

- **Database**: The system data layer; it includes all structures and entities responsible for data storage and management. No application logic is found at this level, apart from the DBMS one that must guarantee the correct functioning of the data structures while assuring the ACID properties of transactional databases.
- **Application Server**: This layer encloses all the logic for the system applications, including the logic needed to interface with external systems and the key algorithms.
- **Mobile App Interface**: The client layer dedicated to mobile devices; it communicates directly with the Application Server and only includes presentation logic.
- **Wearable App Interface**: The client layer dedicated to wearable devices equipped with WearOS (Android) and watchOS (Apple). The app requires to be installed also on a smart-phone and to be paired via bluetooth. It just shows dinamically user's data.
- **Script Interface**: The client layer dedicated to third parties who need to get huge quantities of data: our application doesn't provide other visual interfaces besides the mobile one, but we allow companies to get data through scripts and HTTPS requests providing them libraries to include in order to allow the communication (authentication always required). The layer communicates directly with the Application Server.

The described components are structured in three layers, as shown in the following figure. It also includes the interaction with external systems, that is intended to happen at the level of the Application Server.

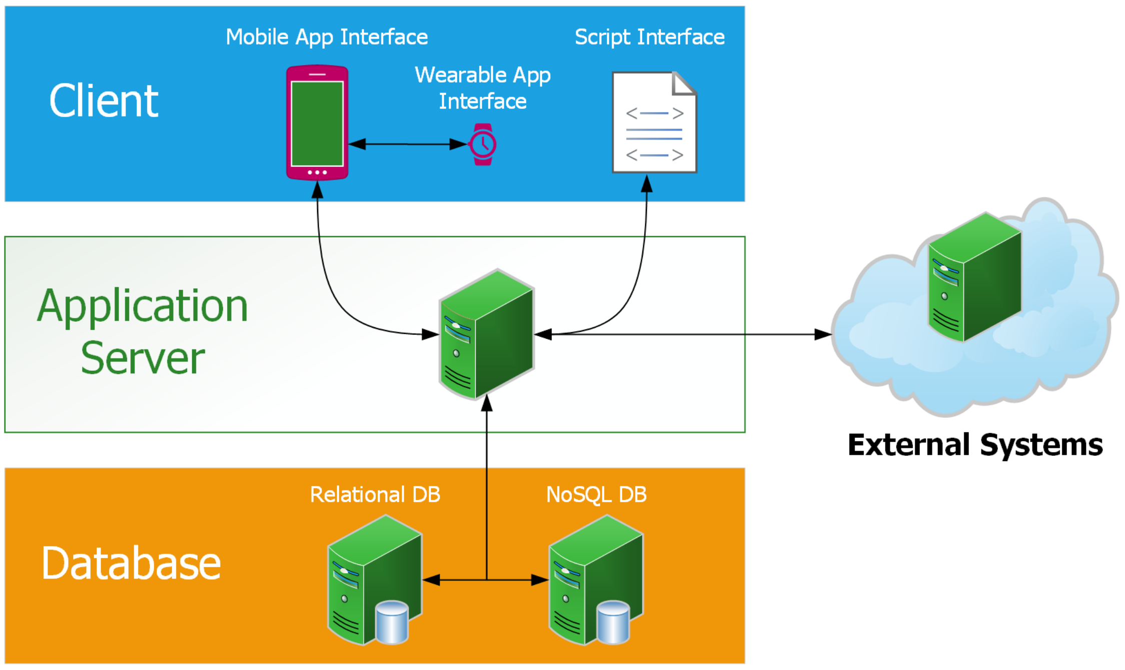

Figure 1: Logical layers of the system

External systems are shown more in detail in the following figure: an high-level overview of the system components.

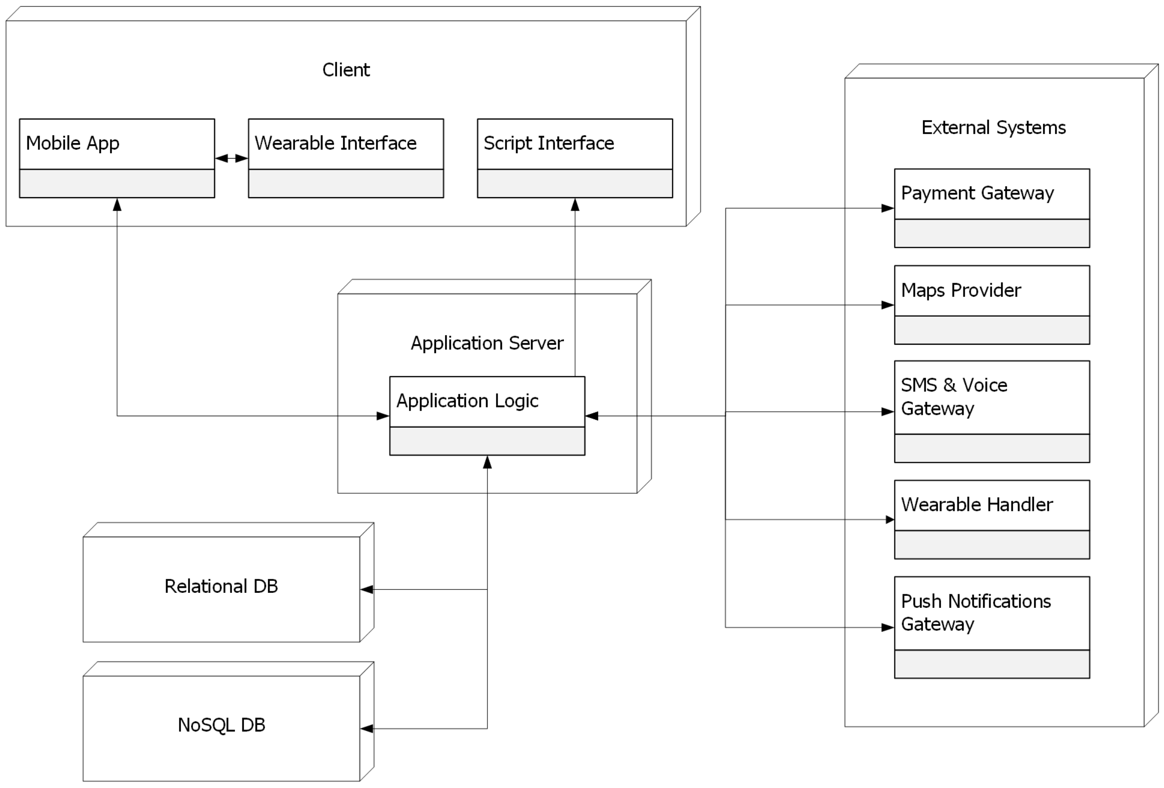

Figure 2: High-level components of the system

2.3 Component view

2.3.1 Database

The Database layer must only be accessible through the Application Server via a dedicated interface. With respect to this, the Application Server must provide a persistence unit to handle the dynamic behaviour of all of the persistent application data.

Besides the Database Engines, this layer is composed by a DBMS, which is in main part relational. The relational approach offers advantages ranging from the easy extensibility to independency from the physical organization, and in general the ACID properties of its transactions. For all these reasons fixed in front data and those that change less frequently are committed to the RDBMS, while for the others a NoSQL is provided. In fact thanks to its well-known scalability it fits better for data accumulated in large numbers every second, which is the case of all the data collected by the application through the wearable device.

The Database layer has to store all the data shown in the following E-R diagram, the sensible of which must be encrypted before being stored.

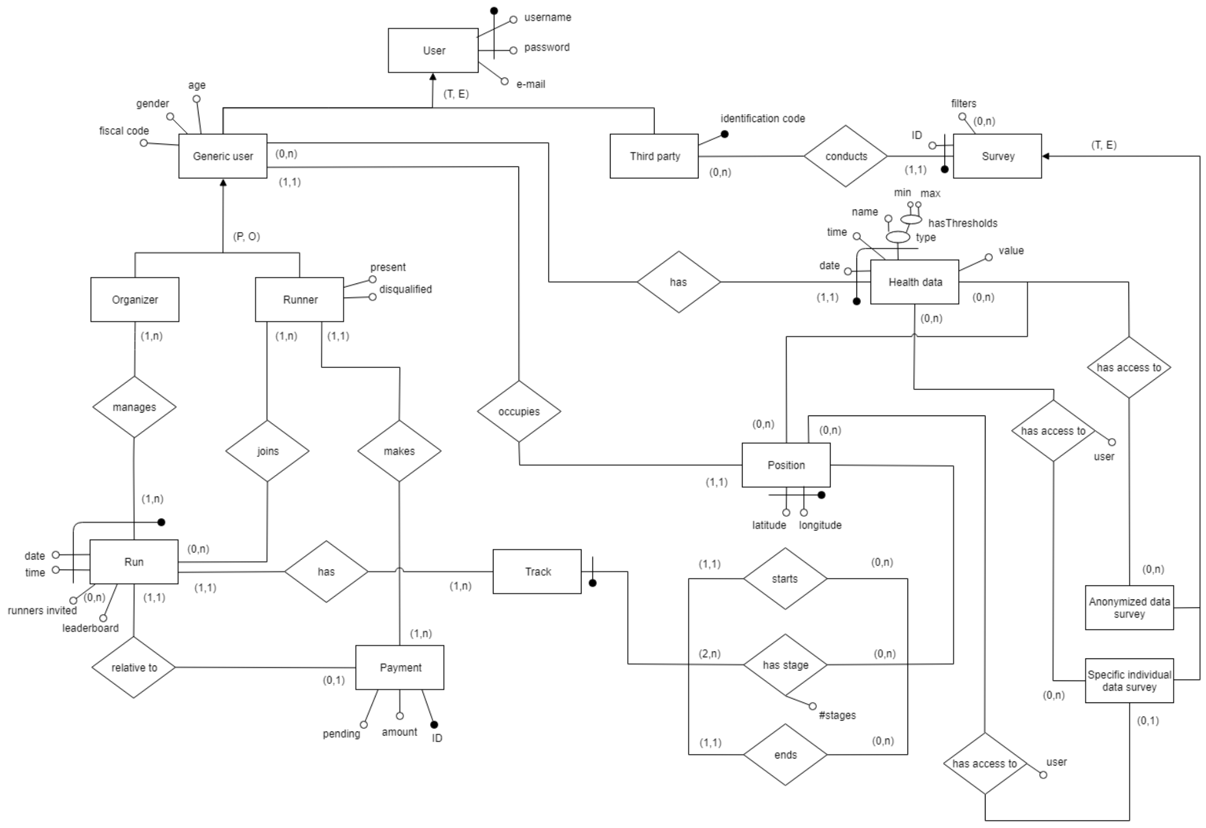

Figure 3: E-R model of the system

2.3.2 Application Server

This layer must handle the business logic as a whole, the connections with the Database Layer and the multiple ways of accessing the application from different clients and external systems. The main feature of the Application Server are the specific modules of business logic, which describe business rules and work-flows for each of the functionalities provided by the application itself.

The interface with the data layer must be handled, as stated before, by a dedicated persistence unit, that will be in charge of the object-relation mapping and dynamic data access and management; this ensures the fact that only the Application Server can access the Database.

The Application Server must provide a means to interface with the mobile clients via specific APIs in order to decouple the different layers with respect to their individual implementation. Moreover, it must provide a way to communicate with external systems by adapting the application to the existing external infrastructures.

The main business logic modules must include:

- **UserManager**: This module manages all the logic involved with user account management, login, registration and profile customization, as well as the generation and provision of user credentials. It also allows to visualize correctly the YOUR RACES screen.

- **DataHandler**: This module manages the logic needed to manage data requests from third parties. It also includes the logic needed to communicate with external wearable devices and get data from them. It must also serve as an interface with the external Wearable Handler (e.g. Health app for iOS, Google Fit for Android, other available apps from various manufacturers).
- **MapManager**: This module includes the logic needed to correctly visualize maps and markers moving on them, and also allows to choose and visualize stages. It must also serve as an interface with the external Maps Provider.
- **PaymentGateway**: The logic involved in the computation of final charges is included in this module; moreover, this unit must stand as an interface with the external Payment Gateway upon the act of the automatic payments.
- **EmergencyManager**: This module includes the logic needed to manage emergency calls and the sending of SMS. It must also serve as an interface with the external SMS & Voice Gateway.
- **NotificationManager**: This module serves as a gateway from the UserManager module, which needs to send an email to the clients, by managing the logic behind the email notification services. It also manages the logic needed to send and receive push notifications serving as an interface with the external Push Notifications Gateway.
- **RunManager**: This module allows users to observe, participate, create and manage running races. It also has to provide and show info about races and it gets data from the MapManager, used to update leaderboards.

2.3.3 Mobile Application Client

As stated in the RASD document, the Mobile Application Client represents the main interface for customers and it should be implemented for both Android and iOS.

The mobile application must be designed in a way that makes communications with the Application Server easy and independent from the implementation of both sides. In order to do so, adequate APIs must be defined and used similarly to what has been described for the interactions between the two server layers.

The mobile application UI must be designed following the guidelines provided by the Android and iOS producers.

The mobile application directly communicates with the business logic using its RESTful API interfaces.

2.3.4 Wearable Application Client

As stated in the RASD document, the Wearable Application Client should be implemented for both WearOS and watchOS.

The software module to be included in the application must manage the GPS locations of the device and the connection with the device sensors, as well as the transmission of all health data to the mobile application via bluetooth.

2.3.5 Desktop Client

As mentioned before, currently TrackMe doesn't offer a desktop or web version of its application. However, companies can get data through scripts and HTTPS requests using provided libraries to include in order to allow the communication (authentication always required).

The Desktop Client directly communicates with the business logic of the Application Server.

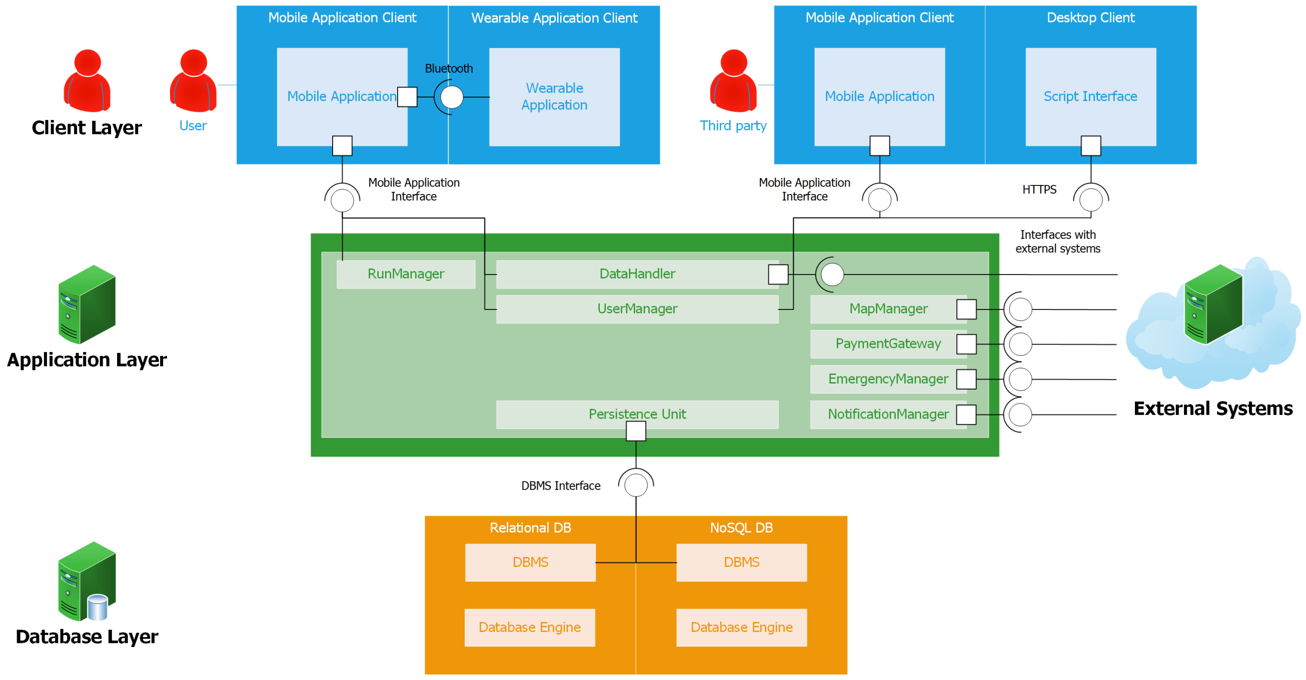

Figure 4: Global component view of the system

2.3.6 Implementation choices

Database implementation

The Database layer will be composed by two different databases: a relational database and a NoSQL database. The implementation choices for the relational one are the following:

- MySQL 5.7 as the relational DBMS.
- InnoDB as the subsiding database engine; InnoDB is a good choice for this application, because it manages concurrent access to the same tables in a very clean and quick way.
- The Java Persistence API (JPA) within the Application Server will serve as an interface with the Database.

The NoSQL Database is in charge of storing all the data generated by the system such as logs and all the kinds of data that will not require any update or delete during the time and that will be accessed to consulted only. The decision of using a NoSQL database instead of a traditional relational database is driven by the need of speed in inserting and accessing a big amount of data.

For this reason, it's not required to ensure all the ACID properties over transaction. The NoSQL Database runs on MongoDB as DBMS and communicates logic tier using TCP/IPO protocol on an arbitrary port. Also, the Eclipse JNoSQL interface is used for communications between the Application Server and the NoSQL Database.

Both databases must be physically protected and duplicated in order to avoid loss. Communication must be encrypted and different users should be created and assigned to the different softwares that access the database in order to always guarantee the minimum required level of privileges per software.

Application Server implementation

The main choice for this layer is the use of Java Enterprise Edition 7 (JEE). This was the most reasonable option for several reasons. First of all, the final product is a large-scope application, and thus needs distribution to great numbers of clients simultaneously. For the same reason, it needs to satisfy continuously evolving functional requirements and customer demands. JEE also allows the developers to focus on the logic behind the main functionalities while being supported by a series of reliable APIs and tools that, among other features, can guarantee the main non-functional requirements of the case (e.g. security, reliability, availability...). Lastly, it can reduce the complexity of the application by using mechanisms and models that easily adapt to a large-scale project.

The specific implementation choices are:

- GlassFish Server as the Application Server implementation.
- Enterprise JavaBeans (EJB) to implement the single business logic modules described in the sections above using Stateless Beans. These will be appropriately subdivided into EJB containers as specified by the JEE documentation.
- Java Persistence API (JPA) as persistence unit to perform the Database access, that is not implemented with direct SQL queries. Entity beans will be used to implement the object representation of the database entities and they are strictly related to the entities of the E-R diagram.
- JAX-RS to implement proper RESTful APIs to interface with clients.
- To interface with external systems, existing RESTful APIs defined by the partners will be used in the communication with the payment handlers, whereas a dedicated one will be provided to the maintenance system to interact properly with the application services.

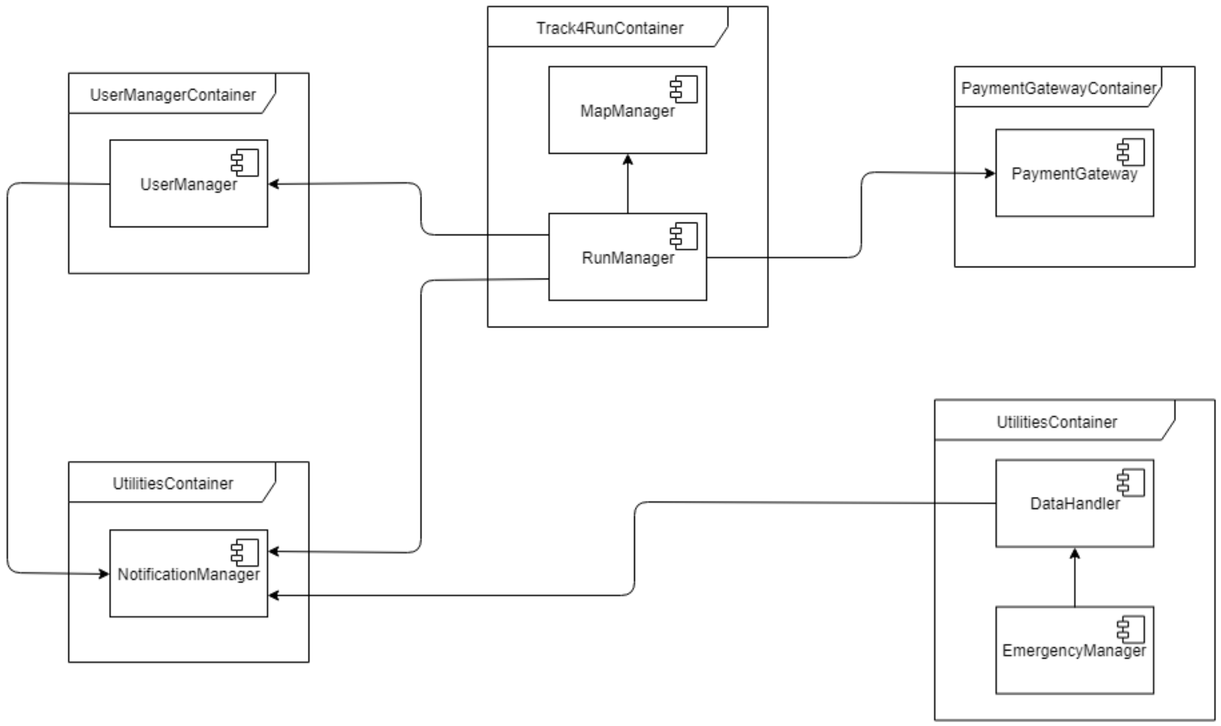

Figure 5: The components of the Application Server implemented as session beans to develop the business logic. An arrow going from component C1 to component C2 means that C1 uses interfacing methods provided by C2

Mobile Application Client implementation

The mobile application UI must be designed following the design guidelines provided by the devices' manufacturers. Two architectures must be supported: iOS and Android. The iOS application must be written in Swift, while the Android one must be implemented in Java.

The core of both application must be a Controller that communicates user inputs (after translating them from the UI input) to the Application Server via RESTful APIs. The access to the GPS of the devices and HealthData of users (only for devices that can collect these kind of data) must be performed through the default frameworks of the respective systems.

If the smartphone is paired with a device equipped with WearOS or watchOS with the TrackMe app installed, communications between the wearable and the mobile app are managed via bluetooth. The Wearable Application Client implementation is based on the mobile application one.

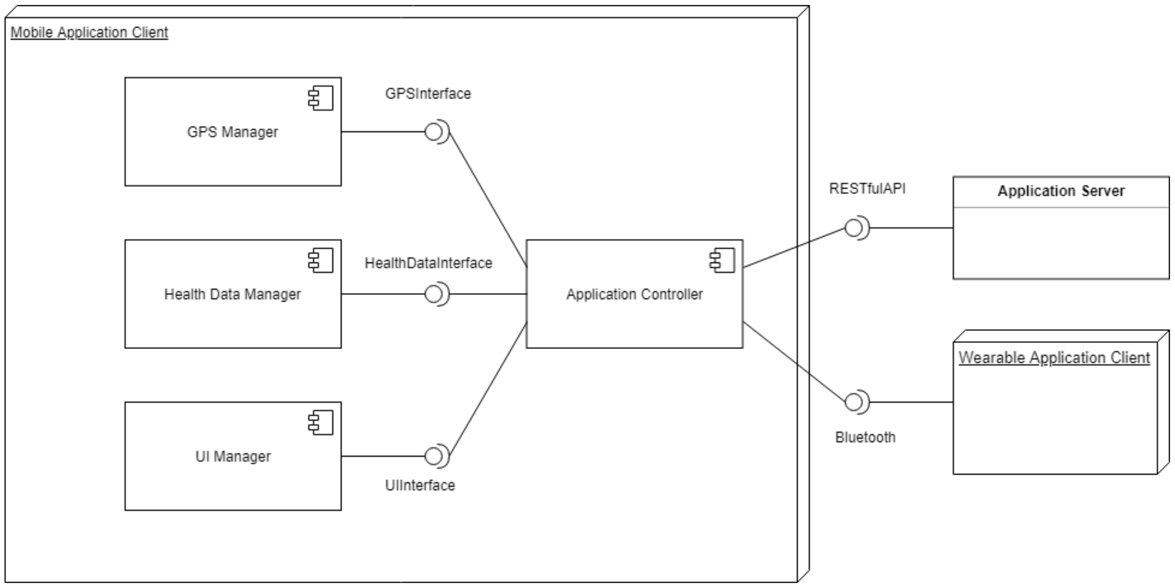

Figure 6: Components of the mobile application

2.4 Deployment view

The physical deployment of the system is done on 3 layers:

- Clients are deployed on different devices: the Desktop Device is related exclusively to third parties, the Mobile Device can be used by both third parties and generic users, the Wearable Device is related exclusively to generic users.
- The main logic of the application will be deployed in the Application Server. This server will communicate with all the other nodes - it will gather information from the external services (not here represented), manage user accounts and saved data from the Database and take requests and send back responses to the users.
- The relational and the NoSQL databases can be deployed on the same physical machine. However, since they are completely independent one from the other, they could be moved on different machines as soon as it will be required for some reason.

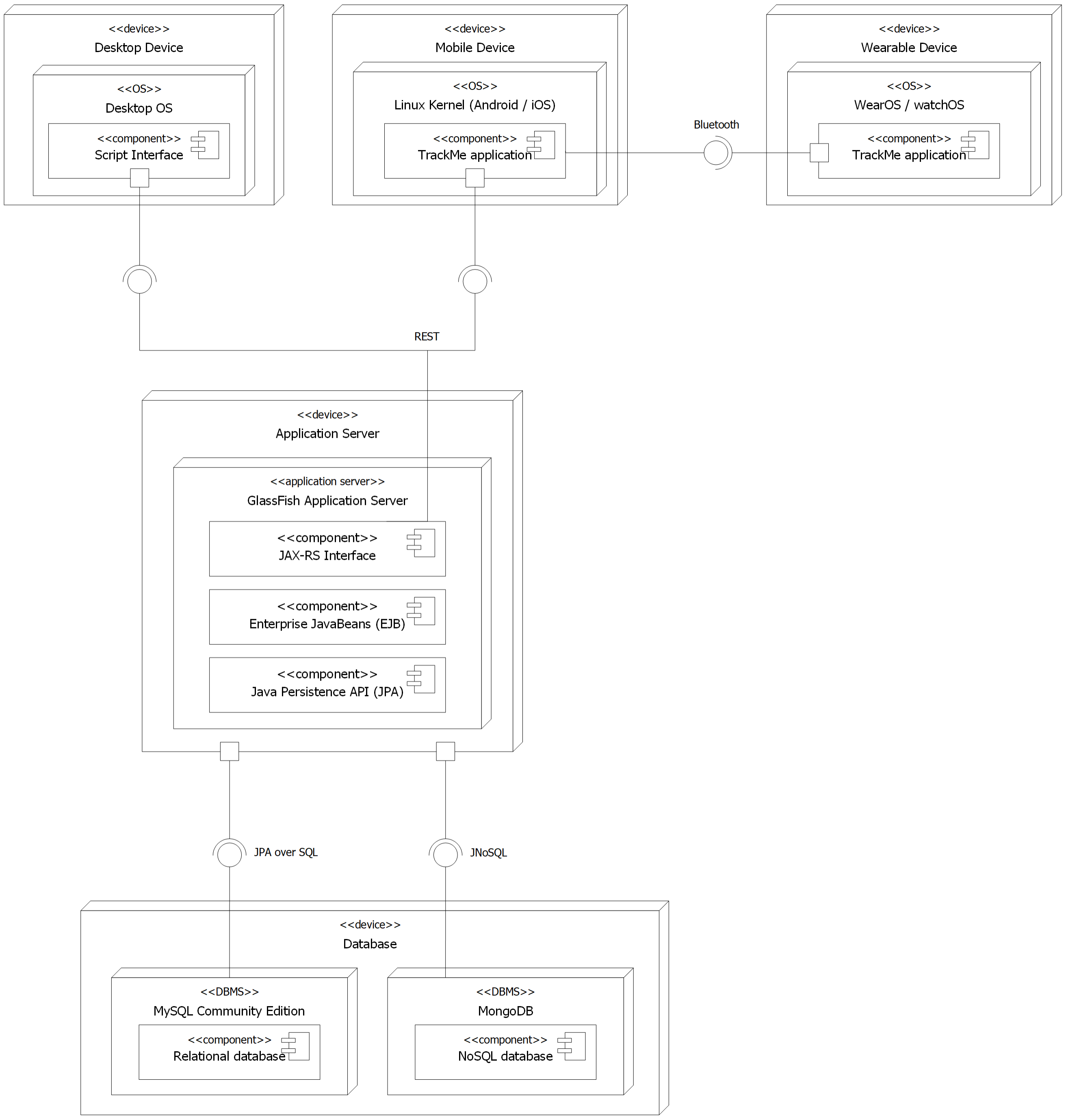

Figure 7: Deployment diagram of the system

2.5 Runtime view

In this section, a detail of the interactions between modules is shown. In particular, it is presented how the server modules, the clients and the database system interact at runtime in the most important scenarios as regarding main functionalities of the system.

The communication between the Application Server and the databases is abstracted by the Persistence Unit, so its data are represented with entity classes; the client is considered as a whole without dividing it into subcomponents.

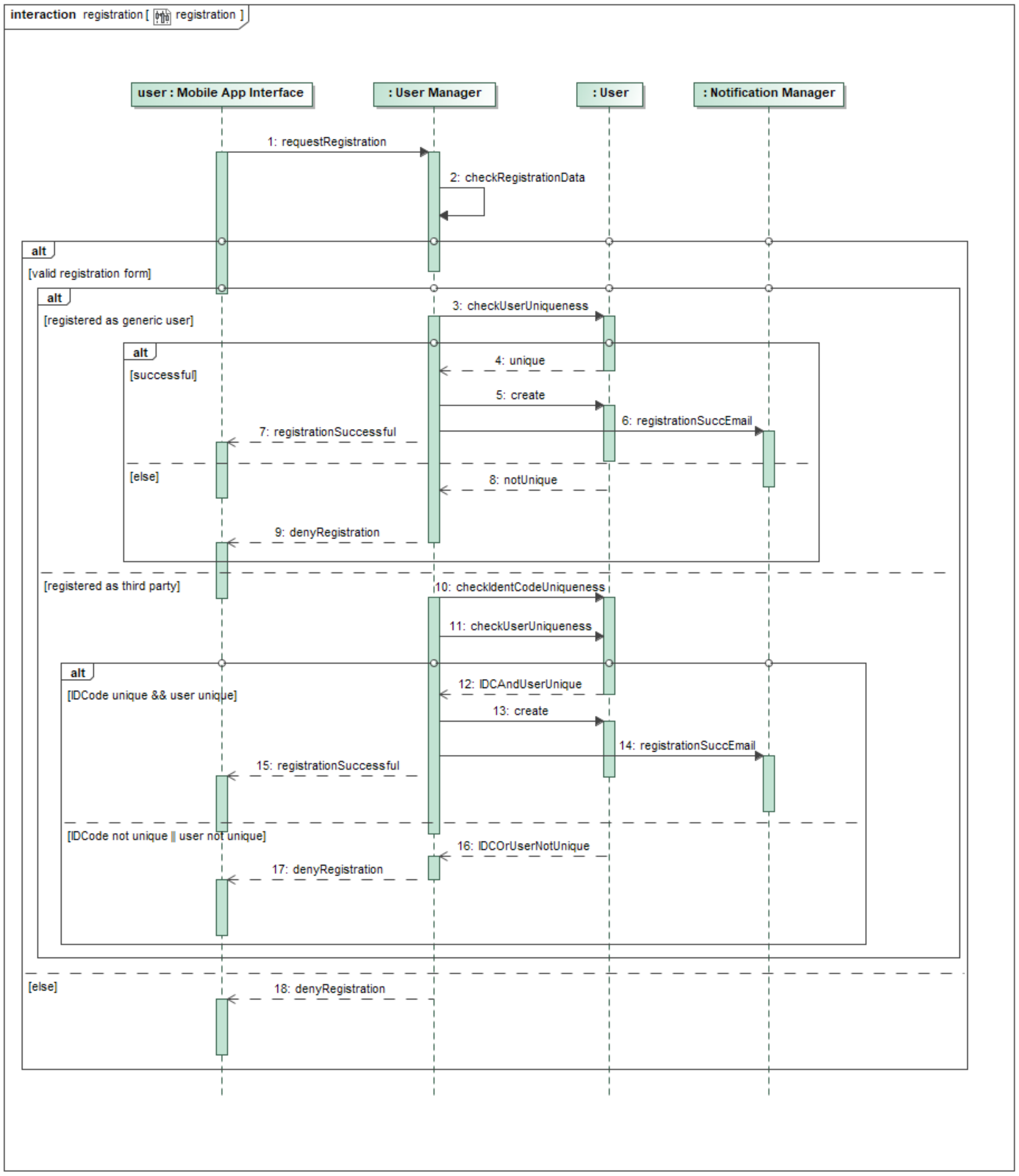

Figure 8: Sequence diagram of the steps to register to the application

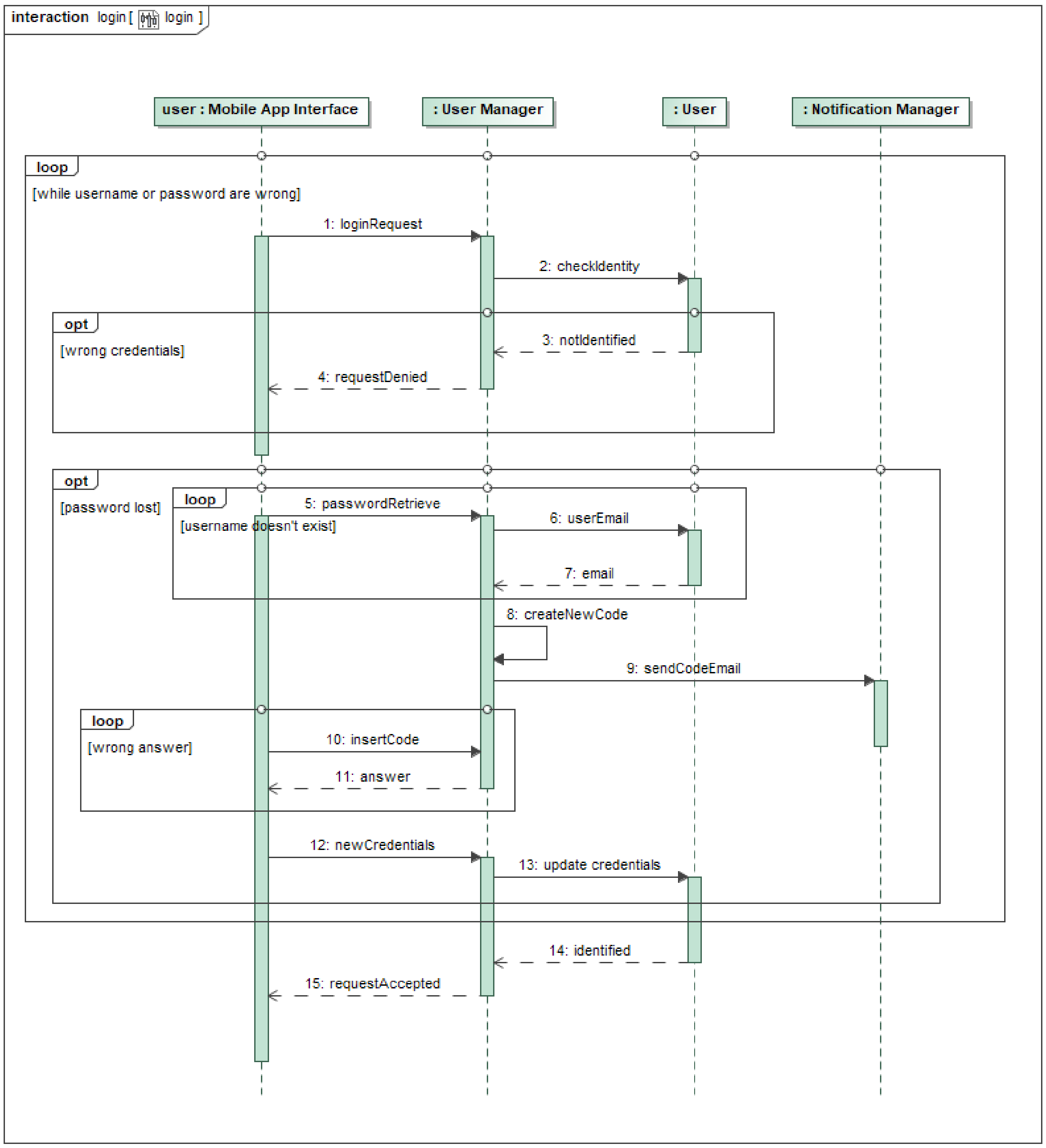

Figure 9: Sequence diagram of the application login

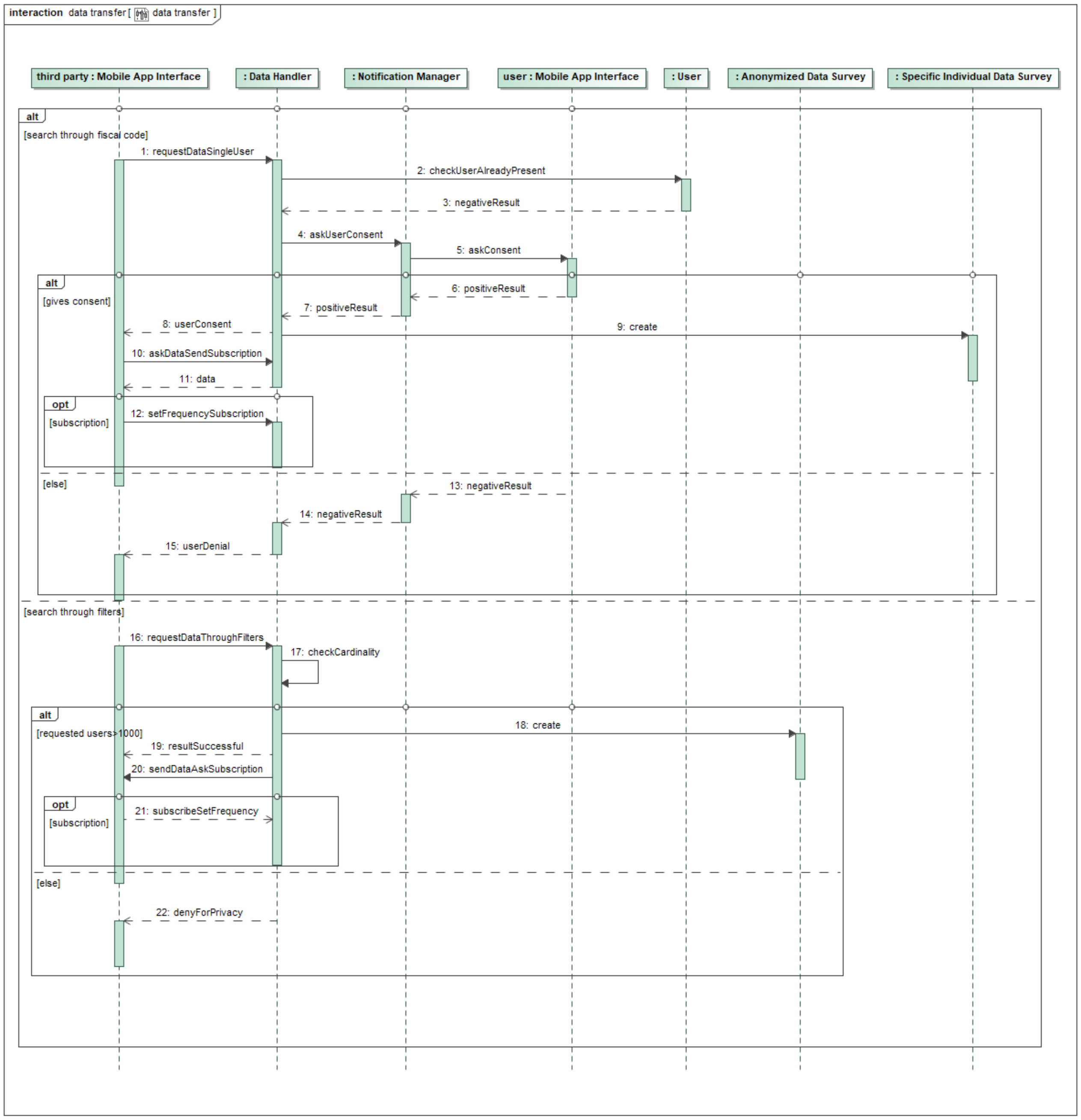

Figure 10: Sequence diagram of a data request from a third party

For sake of legibility in the previous sequence diagram only the case in which the third party doesn't have a survey with the searched user in it is shown. If the third party has already got it, the request for consent isn't forwarded, and the DataHandler gives directly the data requested with the possibility to subscribe to it. This fuctionality is given through the methods `askDataSendSubscription(subscription)` and `setFrequencySubscription(frequency)`, already shown in the diagram.

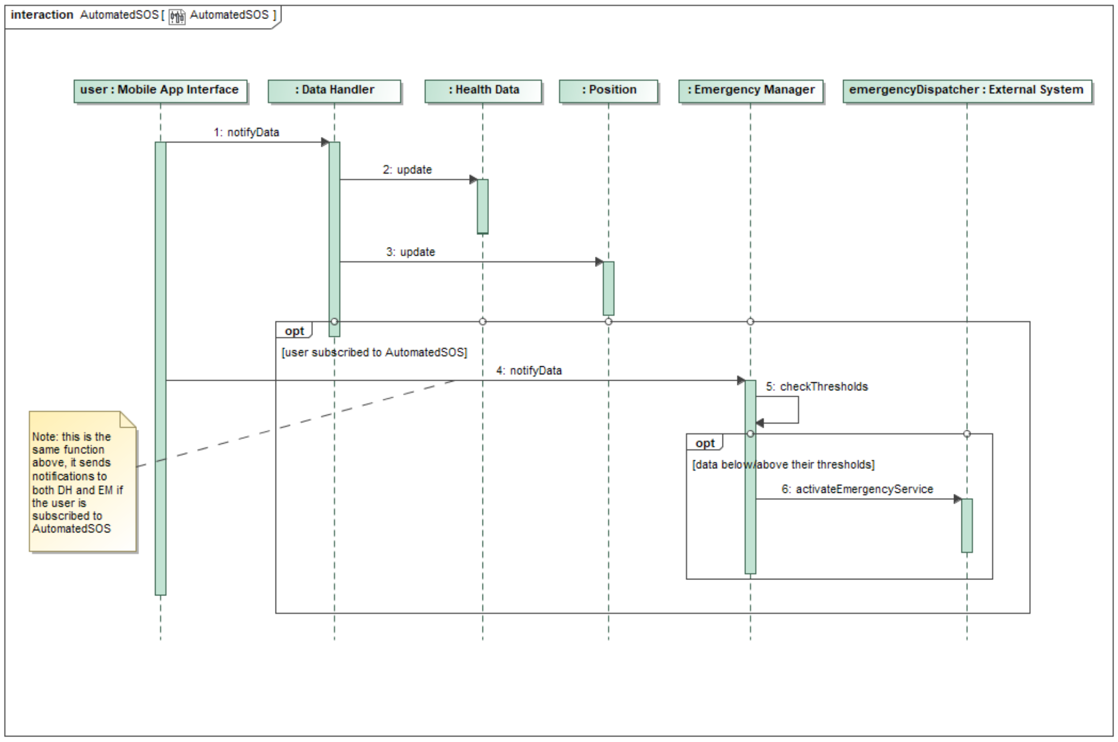
interaction AutomatedSOS [AutomatedSOS]
user : Mobile App Interface
: Data Handler
: Health Data
: Position
: Emergency Manager
emergencyDispatcher : External System
1: notifyData
2: update
3: update
opt
[user subscribed to AutomatedSOS]
4: notifyData
5: checkThresholds
opt
[data below/above their thresholds]
6: activateEmergencyService
Note: this is the same function above, it sends notifications to both DH and EM if the user is subscribed to AutomatedSOS

Figure 11: Sequence diagram of an emergency situation

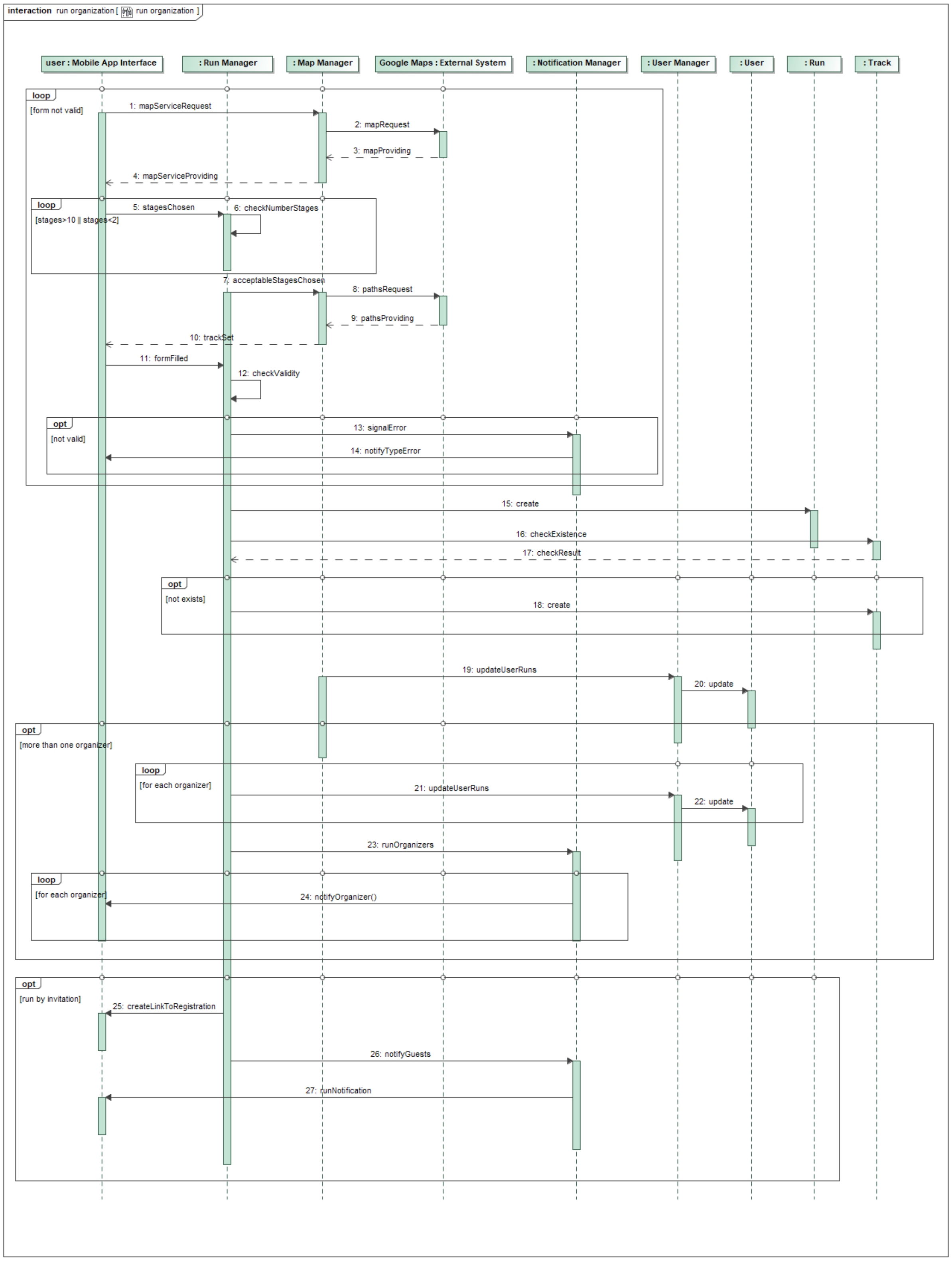

Figure 12: Sequence diagram of a run creation and organization

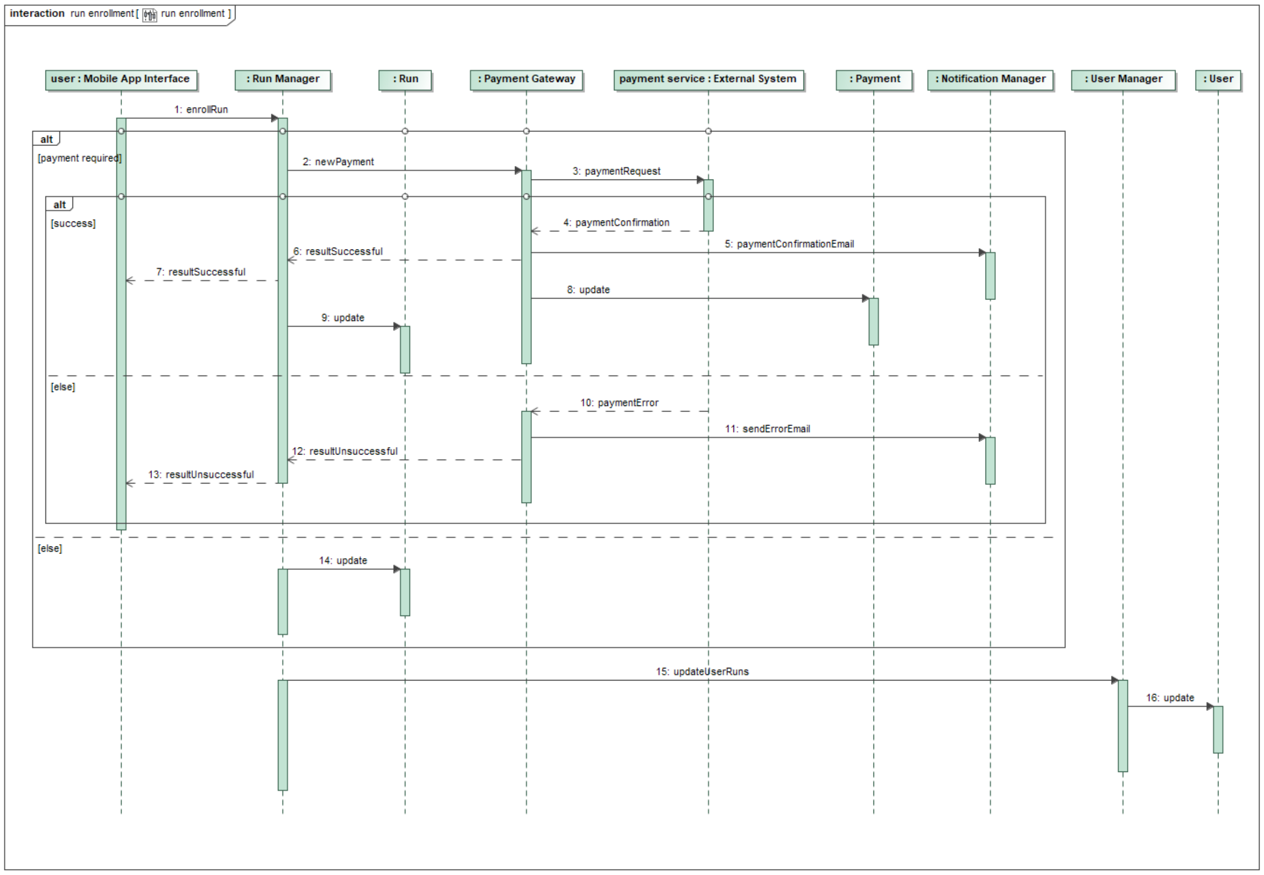

Figure 13: Sequence diagram of a run enrollment from a generic user

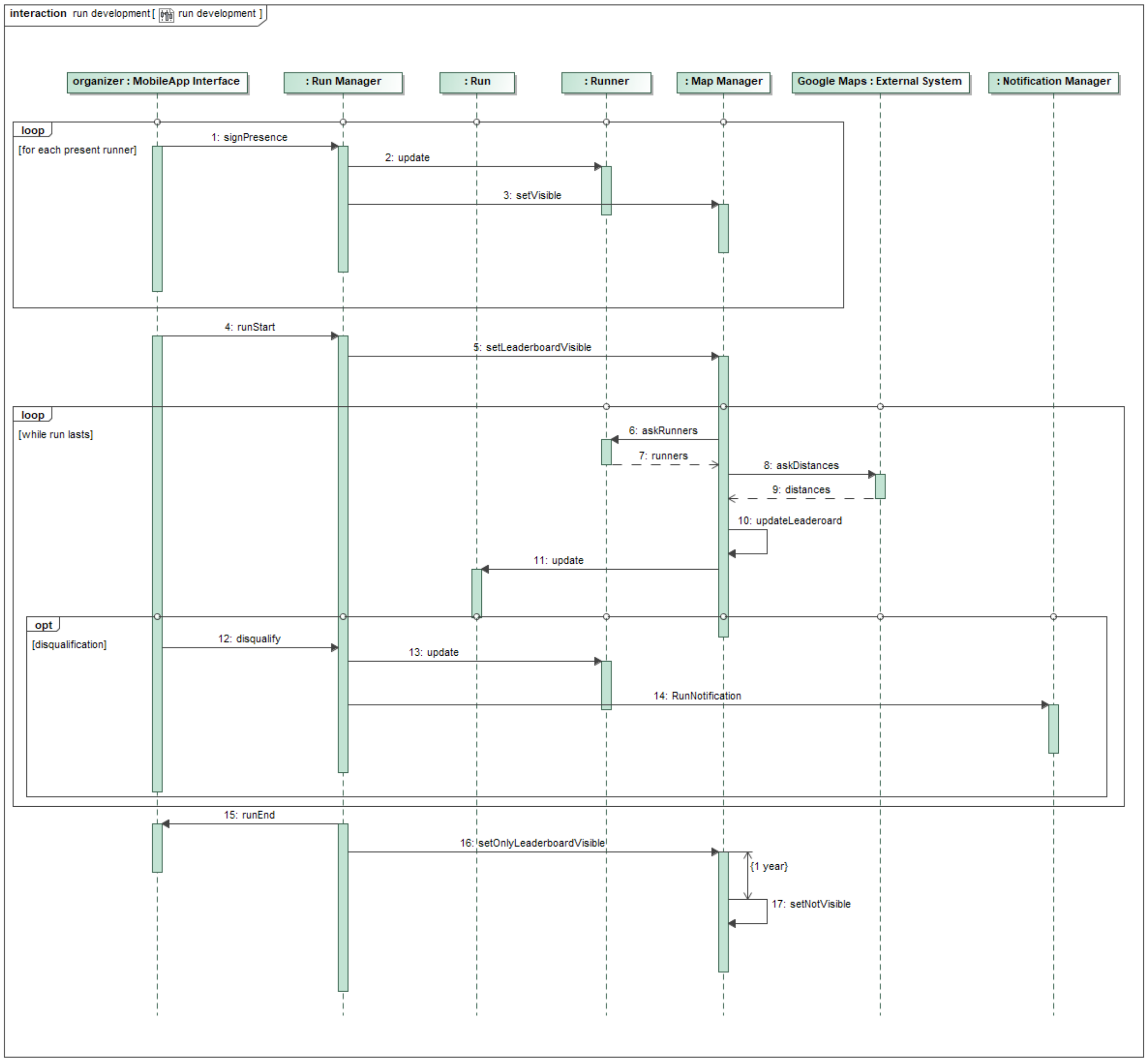

Figure 14: Sequence diagram of a run development from its start to its end

2.6 Component interfaces

This section deals with the comunication between components and subcomponents in terms of interfaces. In fact, the Application Server's interfaces will be described in detail in order to specify exactly how they can be used from the other components and server's subcomponents.

2.6.1 Client

The communication between Client (mobile application or script) and the Application Server is committed to RESTful APIs, that transmit data on HTTPS protocol without further levels. The communication itself is an exchange of HTTPS instructions, URLs and JSON objects.

2.6.2 DBMSs

Since data are mapped into JPA entities, they will be stored persistently in the two databases. The relational one uses JPA over SQL to communicate with the database, while the non relational one uses Eclipse JNoSQL APIs. In order to reply to queries that require both the databeses content two or more distinct queries have to be merged on a more higher level.

2.6.3 External Systems

As regards the mapping service, the Application Server must interface with Google Maps JavaScript API in order to display tracks and runners in real time and in a personalized way, SDK Maps for Android and SDK Maps for iOS, allow an easy way to configure and deploy location services on smartphone.

Payments are committed to a Payment Handler who will provide the appropriate APIs to manage payments of any kind. In particular, the system must support the main payment methods: PayPal and the main credit cards.

As already set in the RASD document, for notifications and messages the Application Server uses the Push Notifications Gateway of own APIs (Push API, to send notification when the application is closed) and a standard SMS & Voice Gateway. An example could be Nexmo, with its SMS & Voice API RESTful service.

Finally, applications that communicate with smart bands and wearable devices other than WearOS and watchOS are considered external systems. Therefore, they will provide their own APIs for communication with the Application Server.

2.6.4 Interfaces of Application Server subcomponents

DataHandler

`requestDataSingleUser(fiscal code)`: method used to forward the request of data of a specific individual; it returns the reply (consent or deny).

`askDataSendSubscription(subscription)`: method used to ask the data requested and assert if he/she wants to subscribe to those data at the same time; it returns the health data and the position.

`setFrequencySubscription(frequency)`: method used to set the frequency on which the caller wants to receive the user's data; it returns nothing.

`requestDataThroughFilters(location, address, age, gender, timeSlot)`: method used to send, in order to submit to check, the request of data of an anonymized group of persons; it returns the check result (success/deny for privacy).

`checkCardinality(specifiedUsersList)`: method used to check if the considered group should be considered anonymized; it returns a boolean result.

`notifyData(user, healthData, position)`: method used to send data to the Data Handler module, that will store them in the NoSQL database: as already stated, only the Application Server can access the databases; it returns nothing.

EmergencyManager

`notifyData(user, healthData, position)`: this method is the same above, it corresponds to the observer pattern's "update", but here it's used to send data in order to check if the user is in danger; it returns nothing.

`checkThresholds(healthData)`: method used to check if each type of health data stays between certain thresholds (which can be taken from health data themselves).

MapManager

`mapServiceRequest(locationName)`: method used to forward a request for a map; it returns the requested map.

`acceptableStagesChosen(positionsSet)`: method used to forward a request for the shortest path between the stages selected; it returns the path.

`setRunnerVisible()`, `setLeaderBoardVisible()`, `setNotVisible()`, `setOnlyLeaderboardVisible()`: these four methods are used to set what is visible on the screen of a run display; they return nothing.

`updateLeaderboard(distancesSet)`: method used to recalculate the leaderboard according to the new runners' distances from the end on the path; it returns the new leaderboard.

NotificationManager

`askUserConsent(user, thirdParty)`: method used to send a notification through which is asked consent for a survey; it returns the user's reply (positive ore negative).

`signalError(user, type` method called when a component wants to send an error notification to the user, specifying the type of error; it returns nothing.

`runOrganizers(organizerList, run)`: method used when in the creation of a run more than one organizer are set and they need to be informed; it returns nothing.

`notifyGuests(runnersList, run)`: method used in order to command to send a notification to all the users invited to a private run; it returns nothing.

`runNotification(user, type)`: method used to create a notification for a runner, which message depends on the type of run notification; it returns nothing.

`paymentConfirmationEmail(user, paymentCode)`: method used to make the NotificationManager module send an e-mail of confirmation for the payment; it returns nothing.

`sendErrorEmail(user, paymentCode)`: method used to make NotificationManager send an e-mail to warn the user his/her payment wasn't successful; it returns nothing.

`registrationSuccEmail(user)`: method used to make NotificationManager send an e-mail to confirm the success of the registration operation; it returns nothing.

`sendCodeEmail(user, code)`: method used to make NotificationManager send an e-mail with an identification code in order to create a new password; it returns nothing.

PaymentGateway

`newPayment(userCredentials, paymentMethod)`: method used to forward a payment request; it returns the payment result (successful or unsuccessful).

RunManager

`stagesChosen(stagesSet)`: method used to send the track stages to submit to check; it returns nothing.

`checkNumberStages(stagesSet)`: method used to check if the number of stages exceeds 10; it returns the check result (positive or negative).

`formFilled(form)`: method used to send the run form filled out; it returns nothing.

`checkValidity(form)`: method used to check if the form is filled correctly and it's therefore possible to create the run; it returns the check result (positive or negative).

UserManager

`requestRegistration(e-mail, username, password)`: method used to forward a request to register to the application, it returns the result (deny or success).

`checkregistrationData(form)`: method used to check if the registration form is complete and its fields have been completed in an acceptable way; it returns the check result.

`loginRequest(username, password)`: method used to require to be identified as a user already registered; it returns the result (deny or acceptance).

`passwordRetrieval(username)`: method used to ask for the possibility to use a new password; it returns nothing.

`createNewCode()`: method that create a temporary identification code in order to make the user choose his/her new password; it returns the code.

`insertCode(code)`: method called to send the code and to check if it matches the code generated previously by the UserManager module; it returns the check result (right or wrong).

`newCredentials(username, newPassword)`: method used to send the new password chosen by the user; it returns nothing.

2.7 Selected architectural styles and patterns

2.7.1 Architectural Styles

Client and Server

The main architectural style adopted for TrackMe system is the Client-Server one, the most well known and used architectural style for distributed applications. It will be adopted in the 3-tier variant, with the Presentation layer on the client (the mobile app), the Application layer on the Application Server and the Data layer on the Database Server.

The main advantages of this choice are the clear decoupling between data and logic, the possibility to increase the portability reaching clients in the most easy way and the availability of a lot of COTS components to develop the system in a very cost-effective way.

Layers

The system will also be divided into 3 Layers:

- Presentation Layer: accessible from users through their mobile application.
- Business Layer: it's on the Application Server.
- Data Layer: it's on the Database Server.

Software is divided in layers for decoupling and for better code maintainability. In this way different developers can focus on different tasks abstracting from other layers.

Thin Client

The thin client approach has been followed while designing the interaction among users' machines and the system itself. All the main logic is implemented by the Application Server that has a sufficient computing power and can manage concurrency issues in an efficient way. On the other hand, the mobile application are in charge of presentation only and they do not involve decision logic.

2.7.2 Design Patterns

Model-View-Controller

The mobile (and wearable) application follow the MVC (Model-View-Controller) software design pattern, that better supports the Client-Server architecture, because it closely follows the division between data (Model), logic (Controller) and presentation (View) present also in our 3-tier architecture.

Observer

The observer or publish-subscribe pattern will also be used, allowing the various components of the system to register themselves and react to event raised by other components. This will be useful in implementing the EmergencyManager that always has to be notified in order to compare data and thresholds.

2.8 Other design decisions

2.8.1 Passwords storage

For security reasons, the user's password is stored using cryptographic hash functions. In addition to that, the password is not only hashed, but also salted. This is a common security choice since many users reuse passwords for multiple sites and a cyber-attack could jeopardize their sensitive information.

3 User Interface design

3.1 UX Diagrams

The RASD already presented the main mobile interfaces for generic users and third parties, in this section the relations among them is shown in detail. UX diagrams below explain which forms have to be completed in which screen, and what flow is followed by the latters. Obviously, only mobile interfaces are dealt with: the script flow is the same as the third parties' UX Diagram.

The two UX Diagrams below are one for the generic user experience and the other for the third party experience.

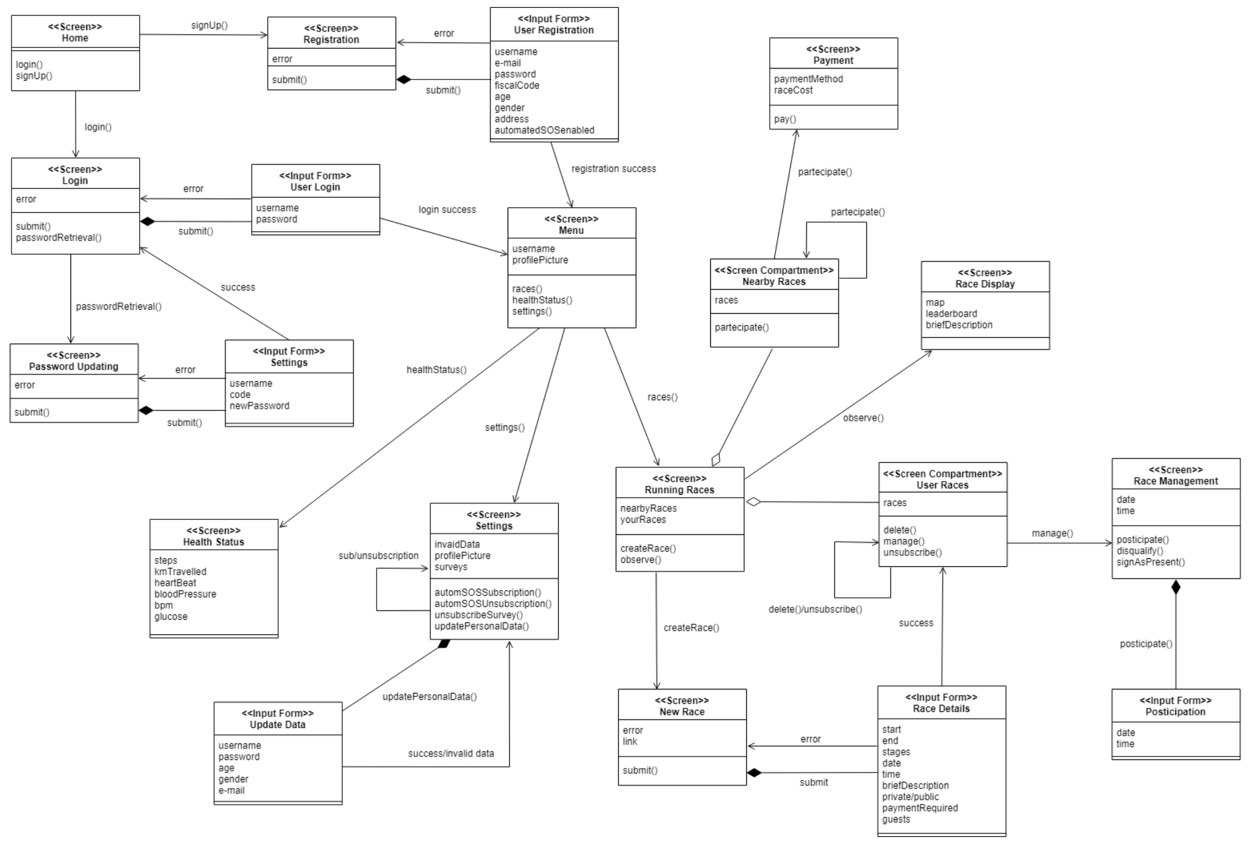

Figure 15: UX Diagram of a the Mobile Application-generic user

As regards the race partecipation, the two "partecipate()" refers to the partecipation when a payment is required and when it doesn't. So, according to the race organization payment request, "partecipate()" will lead only to one screen, exclusively.

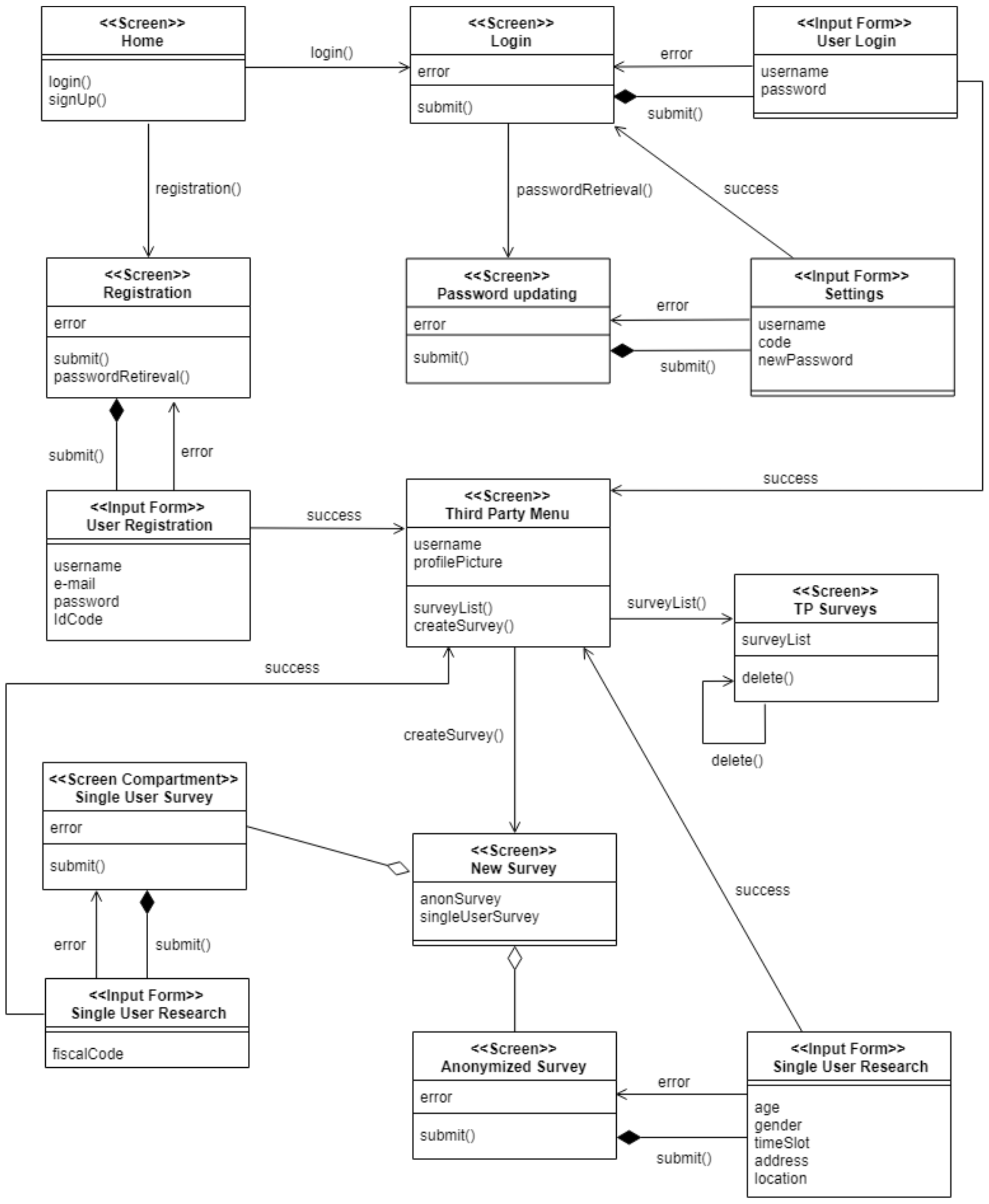

Figure 16: UX Diagram of a the Mobile Application-third party

4 Requirements traceability

The purpose of this section is to show in which modules will be managed functional requirements of the RASD document.

DD Component	Goal	RASD Requirements
UserManager	G1	R1 - The usernames used in the system are unique to every user and third party
		R2 - Users and third parties can create an account by compiling a form
		R2.1 - The form should contain the following: username, password, choice between personal and third party account, other anagraphical info (for the user) or company info (for the third party)
		R3 - Third parties must provide an identification alphanumerical code to confirm their identity
		R4 - Users and third parties can log in to the application by providing the combination of a username and a password that match an account
	G5	R13 - Users can manage the subscription to the AutomatedSOS service through a specific option in the settings menu
		R14 - Old users (60+ years old) are automatically subscribed to the AutomatedSOS service since their registration
	G6	R19 - Users can manage their organized races in a management section in the app
		R19.1 - Organizers can postpone or delete a run. A notification is sent to participants and spectators
EmergencyManager	G5	R15 - The app calls autonomously the NUE (or 911) when such parameters fall below certain thresholds, and sends concurrently user's data (GPS location and health status parameters) via SMS
NotificationManager	G2	R6 - A notify is sent to the selected user, who can accept or refuse the sharing of data with the specific company
		R7 - A message is sent to the third party, containing user's data if he/she allowed the sharing or a notification of refuse otherwise
	G6	R19.1 - Organizers can postpone or delete a run. A notification is sent to participants and spectators

DD Component	**Goal**	**RASD Requirements**
DataHandler	G2	R5 - Third parties can search for specified data by filling the search bar with the fiscal code of a specific user
		R6 - A notify is sent to the selected user, who can accept or refuse the sharing of data with the specific company
		R7 - A message is sent to the third party, containing user's data if he/she allowed the sharing or a notification of refuse otherwise
		R8 - Users under 18 years old can't be shown in the results of a search
	G3	R9 - Third parties can search for anonymized data of a specific group of users by selecting search filters (location, address, age, sex, time slot)
		R10 - Anonymized data are shown just if the research produces at least 1000 results
	G4	R11 - Third parties can flag an option to subscribe to new data of a specific user when they receive his/her data (the user already allowed the data sharing), or they can subscribe to get results of a research with specific filters as soon as they are produced
		R12 - Third parties can manage their subscriptions and set the frequency with which obtaining data for a specific subscription by selecting it and editing preferences
	G9	R27 - Users can visualize their own health status in the home page section of the app
PaymentGateway	G7	R21.1 - If it's specified that some payment is required, it will be committed to an external service
MapManager	G6	R16.3 - The distance between start and arrival points (stages included) cannot exceeds 50kms, otherwise a warning popup is shown and the organizer is invited to change at least one of them before continuing
		R17 - Organizers can choose one of the routes calculated by the system
	G8	R23 - For the duration of the race is always possible to see in the event page the map with markers, associated to runners' name through numbers, and a live leaderboard showing names, order numbers and possible disqualifications
		R24 - A marker is on the track iff it corresponds to a user enrolled, signed as "present" at the run with his/her wearable device with Data4Help installed and not disqualified
		R25 - The track represented is the one chosen by the organizer of the run

DD Component	**Goal**	**RASD Requirements**
RunManager	G6	R16 - Users can create races by defining name, date, time, start point, stages and arrival point, and selecting the method of participation (open or by invitation) and the date and time of expiration for registrations. They can also add other organizers as collaboratos: selected users will receive a notify and they will see the race in their management section
		R16.1 - If a race by invitation is selected, a private link to a registration page is generated
		R16.2 - The organizer can specify a maximum of 10 stages
		R18 - The system allows the organizers to sign an enrolled runner as "present" and to disqualify a runner by signing him/her as "disqualified"
		R20 - Users can visualize nearby races or search for a specific run by searching for the name or location of the race
		R21 - Runners can join a race through the race overview by clicking the "Participate" button. For races by invitation the button is visible just to invited users following the registration link
	G8	R22 - Users can become spectators of a race through the race overview by clicking the "Observe" button

5 Implementation, integration and test plan

5.1 Implementation plan

The implementation of the TrackMe system will be done module by module and component by component. The order in which it is be carried out depends on a number of factors like the complexity of the modules and services, the dependence of other modules on the component being implemented and to the system as a whole, and it should also take into account the possibility of discovering flaws with the proposed design. If such an unfortunate event does happen, the flaws should be found and corrected as soon as possible, to limit the cost of the change of design.

In this sense, the components of TrackMe, could be grouped in the following way, with the order specifying the order of implementation:

1. Model
2. DataHandler and EmergencyManager
3. RunManager, NotificationManager, MapManager
4. UserManager and PaymentGateway

(note that by specifying the names of interfaces of components, we are also considering the concrete implementations, in which ever number they exist)

The Model is proposed to be the first component that is implemented because all parts of the Application Server will be using some element of it and its role in allowing some service to communicate with the DBMSs in the Database Server component is crucial to the whole application. This also eliminates the possibility of every team and developer implementing a part of the Model on the fly, while implementing some service, which could lead to problems with integration and definitions of the same data concepts in different ways.

The second group consists of the DataHandler and EmergencyManager components. They were identified as the most high risk part in the system and it would be advisable to focus on their implementation as soon as possible. They are singled out for the following reasons:

- DataHandler is one of the more complex components of the system, and it's the core of the entire application because it allows the app to support the Data4Help service. It has the most connections to any other part and so it has a vital role of grouping many functionalities of the system and providing access to them to users. The implementation of this module is absolutely a primary concern.
- EmergencyManager allows the app to support the AutomatedSOS service. It also interacts and depends on external services and components and even if its implementation is perhaps not the most complex, it is one of the high priority modules because the AutomatedSOS service is the one which has to guarantee the most high reliability and availability, so it's important to develop this module at the beginning, testing it a lot and eventually improve it in time.

The third group consists of the following:

- RunManager allows the app to support the Track4Run service, and it's one of the most complex modules of the system, even if it's not a primary concern because of the nature of the service itself (Data4Help and AutomatedSOS have a major importance in the general context of the application).
- NotificationManager and MapManager heavily depend on external, already implemented services and components (not including the DBMS component). Because these components already exist, the parts of the system responsible with the task of communicating with them (the ones belonging to this group) have to be adapted to these external services. Although this fact has already been considered in the design of the system, this still represents a point of possible problems that require slight updates and changes to the design. In that sense, their implementation should be done earlier than others.

Finally, the fourth group is composed by:

- UserManager, whose functions, while being some of the most crucial elements of the system, are CRUD operations that are relatively simpler than others that were already mentioned. Considering this, the component is not as risky as others because the time needed to complete it can be relatively accurately predicted.
- PaymentGateway, that despite being a module which heavily depend on external systems, it's absolutely not crucial in the development of the application because payments are related just to some specific running races (not the main focus of the app).

5.2 Integration and Testing

5.2.1 Entry criteria

The integration of components and its testing should start as soon as possible, but it's necessary to specify some conditions before. 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 to our external and already mentioned services: Payment Gateway, Maps Provider, SMS & Voice Gateway, Wearable Handler, Push Notifications Gateway and to DBMSs and the server on which them will be running on.

Next, the modules which are being integrated should have at least the operations concerning one another created, if not completed completely. The operations that have been developed should pass the unit tests in order to be sure that the components are working fine on their own and that if an integration test fails, the problem lies in the in the integration itself.

Considering also the methods already specified, and the relevance of modules in the general context of the application, in order to have integration tests that have some meaning an estimate of the completion of components before the start of testing could be the following:

- UserManager – 20-30% (CRUD operations don't need to be tested so much)
- DataHandler – 90-100% (it's the core of the entire application because of the managing of exchanges of data)

- MapManager – 50%
- PaymentGateway – 20-30% (not crucial in the development of the application)
- EmergencyManager – 70-80% (it's important to test the speed of response in detecting thresholds and communicating outside)
- NotificationManager – 70-80%
- RunManager – 60-70% (interacts with many other modules)

5.2.2 Elements to be integrated

The TrackMe system is composed of a number of components, as already shown, and the integration process can be grouped in the following way:

- Integration of components with DBMSs
- Integration of components with (other) external services
- Integration of components of the Application Server
- Integration of the client (mobile application and wearable application) and the Application Server

Integration of components with the DBMSs - This group includes the integration of every part of the Application Server that uses the external server with the database, in any way (the "DBMS" notation refers to the external server with the database and its DBMSs). The specific integrations in this group are the following ones:

◇ UserManager, DBMS

◇ DataHandler, DBMS

◇ MapManager, DBMS

◇ NotificationManager, DBMS

◇ RunManager, DBMS

Integration of components with the (other) external services - In this group, we include the integration of every part of the system with an already existing and functional external service (without counting the DBMSs). They are the following ones:

◇ PaymentGateway, external Payment Gateway

◇ MapManager, external Maps Provider

◇ EmergencyManager, external SMS & Voice Gateway

◇ DataHandler, external Wearable Handler

◇ NotificationManager, external Push Notifications Gateway

Integration of the components of the Application Server - Here, the integration between the parts of the Application Server is conducted. It is composed of:

- ◇ MapManager, RunManager
- ◇ PaymentManager, RunManager
- ◇ UserManager, RunManager
- ◇ NotificationManager, RunManager
- ◇ NotificationManager, UserManager
- ◇ NotificationManager, DataHandler
- ◇ DataHandler, EmergencyManager

Integration of the client and the Application Server - This is carried out to enable the sending of requests to the server via the mobile application (and interactions with the wearable application) which the user will be utilizing.

5.2.3 Integration testing strategy

The chosen strategy for the integration testing is the incremental integration testing, in particular the bottom-up one is the most suitable for the application. This allows us to start the integration and it's testing while not waiting for the completion of the development and the unit testing of each component in the system. Considering the integration of two components, we would assume that, in best case, they have been implemented fully and that their respectful unit tests pass. However, the integration can, in some cases, start, if necessary, before the implementation has been completed. This can be allowed if the part of the component needed for that specific integration has been completed and tested.

Since the opted solution is to start from the bottom-up, that means that the among the first integrations performed will have the already built external components in them. Since the application rests on these services and the communication with them, this order of integration and testing will enable the earlier detection of errors in these critical parts.

It should be noted that it can be assumed that each integration in the same level of hierarchy (defined by the groups of integrations in the previous chapter) 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.

Finally, black-box testing is suitable for integration, system and acceptance testing. This kind of tests are perfect because they are specs-based and not code-based, and can help identifying missing functionalities in the system, so the idea is to begin with black-box tests as soon as possible.

5.2.4 Sequence of component/function integration

In this section, the list and order of every integration that is performed is shown. As already stated, the integration will be performed with the bottom-up strategy. As a notation used, the arrow pointing from the first component in the integration to the second means that the second one is using the services of the first component.

It should be noted that there will be no explicit integration of the Model with any of the other components. This is because the nature of the component, the extent of the usage and dependency of other components on it and the implementation plan, that clearly states that the Model will be the first part that is implemented, mean that the integration itself is already being done during the implementation phase of the depending components and its correctness will inherently be tested by the unit tests of each component.

It should also be noted, as stated before in this section, that the "DBMS" notation refers to the external server with the database and its DBMSs and that every "service" component considers all of the implementations, whatever they may be, of the specified interface.

Integration of components with DBMSs

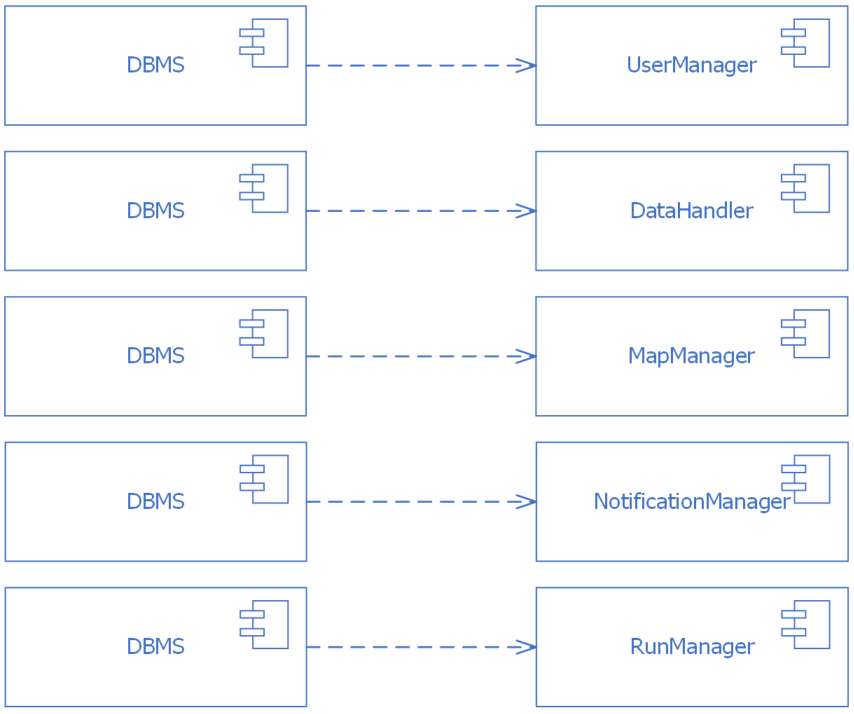

Figure 17: Integration of components with DBMSs. Each of the components shown here has the operations needed to work with the database, concerning the parts of the Model that they are using

Integration of components with (other) external services

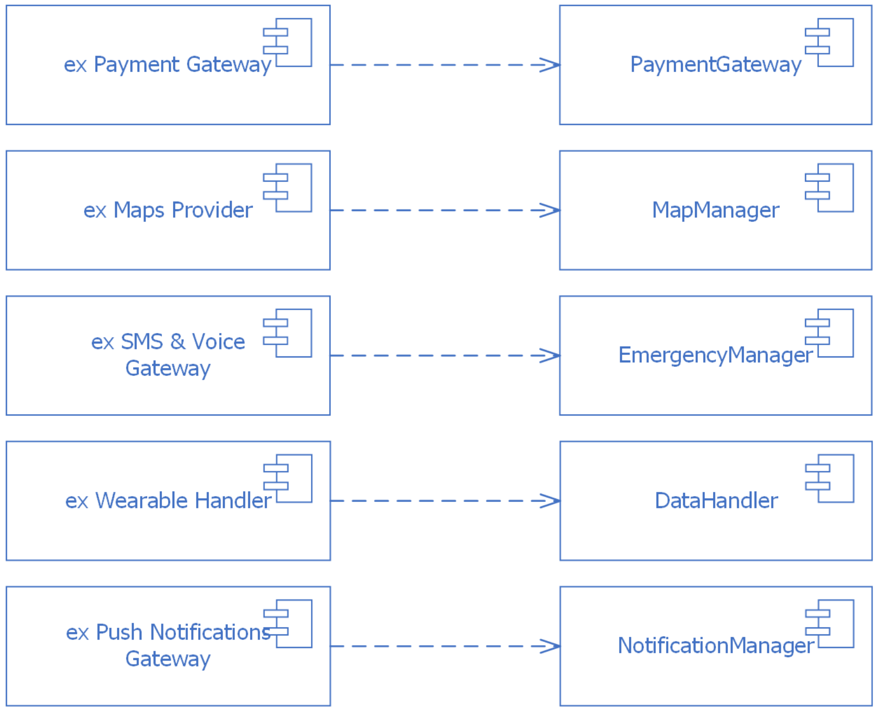

Figure 18: Integration of components with (other) external services. External service components, not considering the database, are integrated with either the components of the Application Server that dedicated to their usage them or the client mobile (or wearable) application

Integration of components of the Application Server

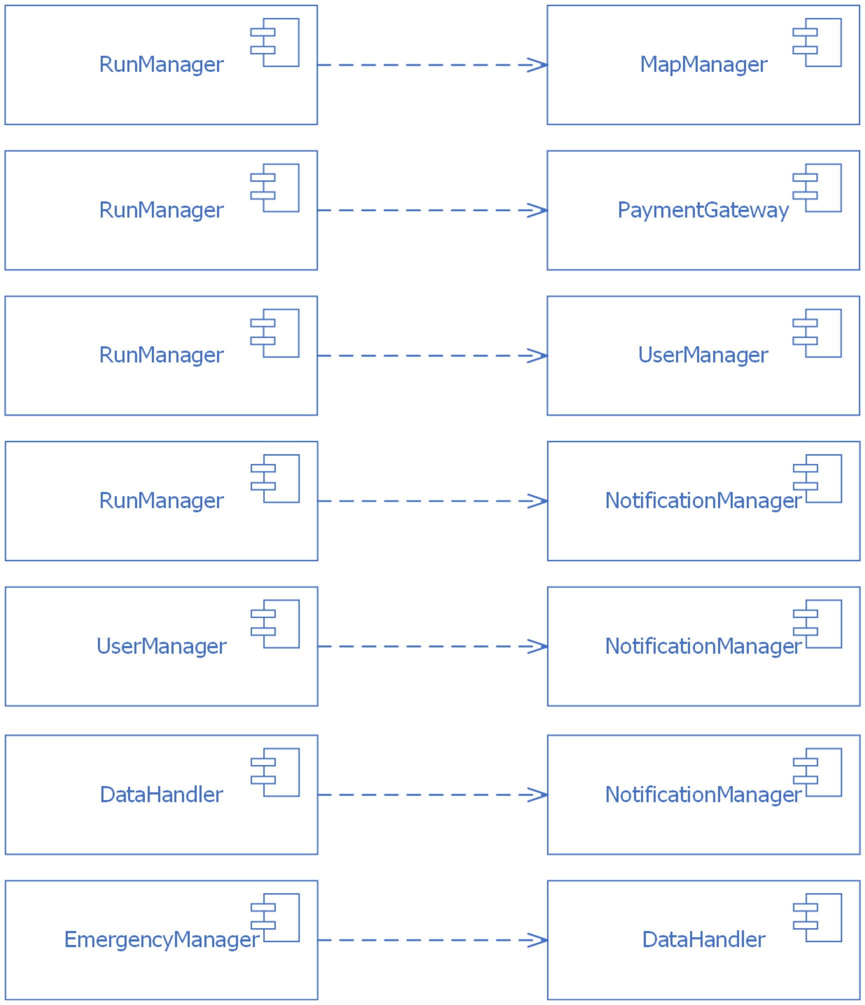

Figure 19: Integration of components of the Application Server. After this phase of integration and testing is complete, the Application Server should be integrated fully, while being connected to the Database Server

Integration of the client (mobile application and wearable application) and the Application Server

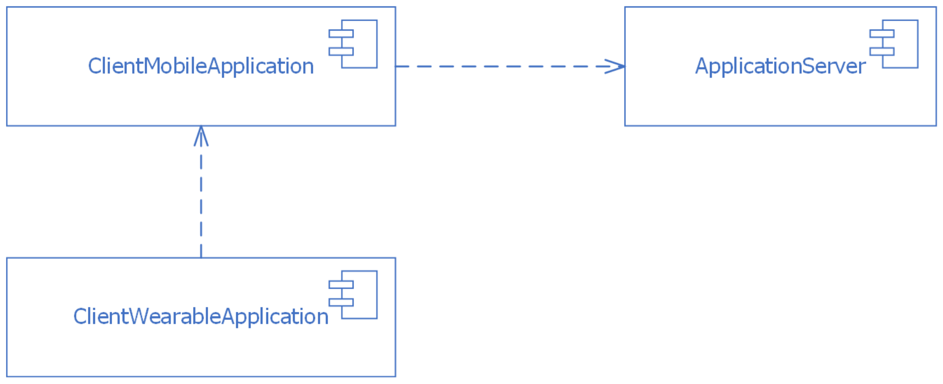

Figure 20: Integration of the client (mobile application and wearable application) and the Application Server. It should be done last, after at least most, if not all, of the Application Server parts have been developed and integrated

6 Effort spent

6.1 Claudia Conchetto

Description of the task	**Approximate hours**
Component view	5
Runtime view	9
Component interfaces	6
User interface design	6

6.2 Riccardo Corona

Description of the task	**Approximate hours**
Introduction	3
High-level components	3
Component view	4
Deployment view	2
Component interfaces	2
Implementation, integration and test plan	6

6.3 Andrea Crivellin

Description of the task	**Approximate hours**
Component view	8
Component interfaces	2
Requirements traceability	3

7 References

- ”Mandatory Project Assignment AY 2018-2019” specification document, available on BeeP’s course channel.
- Slides about design and testing available on BeeP’s course channel.
- Eclipse JNoSQL website with explanation of the interface `https://www.eclipse.org/community/eclipse_newsletter/2018/april/jnosql.php`
- `https://dzone.com/articles/an-introduction-to-dbms-types`, to better understand advantages of NoSQL databases.
- Nexmo website with explanation of the service `https://www.nexmo.com/`
- `http://total-qa.com/rest-services`, to better understand how REST API work.