TrackMe: Implementation document
Software Engineer 2 - 2018/2019

Riccardo Poiani, Mattia Tibaldi, Tang-Tang Zhou
Politecnico di Milano

Version 1.0
*†

January 13, 2019

*Link to source code: `https://github.com/tangtang95/PoianiTibaldiZhou/tree/master/Implementation`

†Link to what has to be installed: `https://github.com/tangtang95/PoianiTibaldiZhou/releases` and the software specified in the installation instruction.

Contents

1 Introduction

The purpose of this document is to provide all the information regarding the implementation of a viable product of the TrackMe project: in particular it regards the services of Data4Help and AutomatedSOS. It follows a brief description of the structure of the document:

- First of all, in the front page it is possible to find links to the source code and to what needs to be installed
- The second section illustrates what are the requirements that have been actually implemented, providing some useful motivations in order to understand the choices that were made
- The third one takes into consideration the frameworks adopted, recapping and introducing further comments on what was already mentioned in the Design Document. Moreover, benefits and drawbacks are better analyzed, and ulterior decisions are discussed
- The fourth chapter analyzes the structure of the source code and diagrams are present to illustrate a precise structure of the written code
- The fifth section provides information on how tests were written. Coverage is here presented and system tests is presented and commented
- The final chapter helps in understanding what is necessary to do in order to install and run the software, with all the necessary prerequisites

2 Requirements and functions implemented

As already mentioned, in this section it is possible to find information regarding the requirements and the functions that are actually implemented with some motivations.

2.1 Core requirements and functions

All the core requirements from R1 to R9 have been implemented, since, as the name suggests, they are fundamental.

2.2 Data4Help requirements and functions

All the requirements related to G1, G2, G8, G9, G12 and G13 have been implemented, basically because they are considered as the main features and components of Data4Help. This is true for all the mentioned goals except for what concerns G9, that is the blocking of third party customers, that can be considered as an additional functions, but can always be useful for engaging future users. The excluded ones are the following

- G7, that is the management of elapsing requests
- G14, that is the subscription to non existing data

The motivation that stands behind this choice is basically a constraint on development time and the fact that these were considered more as a nice feature to have, and not something really critical. Indeed, it is still possible to have a good prototype of the service, even without these features. It should be noted, however, that they do not require big efforts and can be easily integrated into the project in a second time: in particular, G7 can be considered as a periodical task to be scheduled that checks and manages the time stamp of the pending requests. For what concerns G14, instead, the discussion is a bit more complex, but it basically consists in introducing a new status for the requests and a task that operates with the requests that are in such status and that performs some checks w.r.t. to the provided dates (that are the starting date and the ending date of a request: this will be clear after inspecting the source code of the entities that regard the requests).
A final note on the requirements of Data4Help is the following and it regards the type of aggregated data that third parties can request. All the filters mentioned in the design document have been developed, from a server-side point of view. For clarifying this statement, that may sound obscure, consider that every available filter, except the one that regards GPS position data, is only related to some plain input that a third party customer inserts in the application and that is sent to the system. However, for filtering on GPS position a third party should define the interested region by specifying the coordinates that specify the bounds of the interested region. Since, of course, this is totally no user friendly, the application could provide a list of possible cities and region and automatically translates it. Of course, this feature could also be deployed directly on the server.
Nevertheless, up to now, what is present is the possibility of inserting group requests specifying the GPS filters, but this has not been implemented in the mobile application for the discussed reason, and, because, at this stage, it is considered sufficient to have all the other filters.

2.3 AutomatedSOS

The goals, and the related requirements, that involves AutomatedSOS are G3 and G4. However, in this case, not many of the requirements have been implemented, also due to some external limitations: indeed, considering the Android system and hardware, it is not possible to just intercept the phone call and access the microphone (for automating the phone call) and the speakers (to retrieve the response). Therefore, in order to fully develop the requirements it would be necessary to use VoIP API, that requires some sort of payments. However, some requirements such as R12, R13, R15 and R17 have been implemented anyway, with the difference that the VoIP calls are mocked with normal phone calls and no automation is present for interacting with the emergency room operator. This has been considered enough for a viable product also because it has been chosen to give more importance to the core business of the company, that is considered to be Data4Help. Another few words should be spent on R17, that is "during phone calls, the GPS is set on high precision": this is automatically performed by the Android system when calling emergency services [1].
Finally, for what concerns the part regarding the 5 seconds the following shrewdness have been adopted: the maxitum timeout for retrieving the position takes in the worst case 1 second. After that, the more critical part is parsing a

JSON that contains the emergency numbers of the various countries, but this weights only 120kB. It is thought that this should be fast enough to satisfies the requirements of the five seconds. However, the process speed of code is a problem always related with the performance of the user device and the consumed resources of other applications, therefore being 100% sure that the task is accomplished within that time is impossible. Nevertheless, some tests are perfomed and the average speed satisfies this requirement.

2.4 Non functional requirements

For what concerns the non functional requirements, as mentioned in the Design document, a microservice architecture has been developed. In particular, the main implemented features of the architecture are the following:

- Communication between microservices, since this is crucial in order to have a viable product. Indeed, without this, almost no requirement could be fully accomplished
- API Gateway that performs also authentication and authorization: this is also a core feature, since it is the entry point for accessing all the REST API that the various microservices provide
- Service registry, because otherwise, one have to set up all the network communication in a more static way, and also the management of forwarding requests from the gateway would have been more complex
- A basic load balancer, called Ribbon, is already present in the API gateway that have been used (i.e. see Zuul in the adopted developed framework chapter)

Some basic security feature have been developed also: indeed, all the passwords are stored in the database with bcrypt and the only type of communication allowed between clients and the API gateway, is HTTPS. In order to do this, a custom SSL certificate has been generated using the java keytool [2].
The authentication have been implemented by means of an UUID random token and the APIs have been secured in such a way that a certain client can access only the method that he should access: for example, a user cannot access methods that regards a third party customer, but, also a user cannot access API that regards another user (e.g. a user can access only his pending request, and not the pending requests of any user)
As one can claim, using UUID random tokens is for sure not the best way of achieving authentication, however, the code has been designed in such a way that is easily possible to upgrade this to the usage of JWT just implementing a single interface. This choice has been done in order to simplify a bit the security issue and to focus more on the business core of the project.

2.5 Other comments

The database regarding the collected health and position data, has not been deployed on the cloud at this early stage, because the integration was not considered necessary. Indeed, it is more something that regards the deploy, rather than the implementation itself.

The same reasoning has been applied also for the deployment of JARs in docker containers.
Some checks, from a server side point of view, have not been implemented: for example, when registering a third party customer have to provide an email: it is not checked that the string constitutes a valid email. Things like this have been considered not crucial for a prototype, compared, for example, with the set up of the distributed system or to the business function. Nevertheless, some controls have been made in the Android application.
As regards the andorid application, additional features such as the setting screen and the schematic of the user profile have not been implemented due to lack of time. Furthermore, the legal notices in the use of the application have not been explicated in the appropriate popUp.

3 Adopted development framework

Most of the choices that regards the frameworks were already briefly introduced and motivated in the section 2.7 of the design document (that is, other design decision). However, here frameworks and other APIs are recapped:

- Spring[3] Boot has been adopted since it fits well for microservices and the set up of pattern components of the architecture (e.g. service registry) can be easily integrated with the usage of Spring Cloud. The main drawback of this choice is that spring boot does not offer much control to the developer: this "of course" limit the development time, but when something goes wrong, it may take time to fix the issue.

- Spring Security has been used to develop the authentication and the access control of the application. It is basically the de-facto standard for securing Spring-based application.

- Spring Cloud Netflix has been used to integrate and set up the API Gateway and the service registry. In particular, Zuul has been set up as the API gateway integrated with ribbon and the load balancer, which is already incorporated in Zuul and Eureka as the service registry. These were of a great help because it is only necessary to find the right configuration settings and then everything works as expected.

- RabbitMQ[4] is used to set up the communications among microservices. This was chosen because of its properties: durable, open-source, asynchrony and mostly for the reliability. With the help of RabbitMQ it has been achieved a communication between microservices which is asynchronous, durable and reliable. Even if RabbitMQ crashes, once it restarts the queues and messages are saved and, therefore, nothing is lost. Moreover, in the remote case where a microservice cannot handle the message due to an unavailability of other machines such as databases, the message is re-pushed into the same queue and it will be re-pulled until the operation has finished correctly. In conclusion, RabbitMQ is an essential piece to achieve the property of "Eventually Consistency".

- Android: the mobile application has been developed for android, targets SDK version 27 and min SDK version 24. Here, Butter Knife has been

used in order to easily bind the layout with the activities. It also enables to automatically configure listener to onClick methods.

Moreover, not already cited APIs has been adopted, and, therefore, are listed:

- Project Lombok, that is a java library that automatically, by use of annotators, creates certain code in order to reduce the amount of boilerplate code that one write
- Jayway JsonPath for manipulating JSON
- MySQL has been adopted as a DBMS
- JPA for the management of persistence and object/relation mapping
- Google guava for the usage of immutable maps
- Jackson, that is an high performance JSON processor for Java has been used, since it the default library used by Spring boot to convert object to json and viceversa. This has been very useful in the set up of the controllers: indeed, it was possible to define POJOs as controller attributes, and the conversion between HTTP requests and Java is perfectly handled. Jackson has been used also to define views in controller method: this allows to set up that in certain controller methods only certain attributes of a POJO are used. For instance one can specify that when the user is accessing its own information, his password is not sent back, but, when registering, of course the password is needed. Using different views in different methods helps in achieving what mentioned before
- HATEOS is used to provide hypermedia contents to the clients. This helps the client mobile application in accessing the right methods and it makes the APIs RESTful
- QueryDSL has been adopted to perform the custom query to access the group request data
- GeoNames has been used in order to find the country codes, if the Android geocoder doesn't work properly
- JaCoCo has been used to generate reports on the test coverage (they will be shown in the chapter related to testing)
- JMeter has been used to test the system as a whole: functional testing, load testing, and a simulation of a real load testing where there is some kind of think time for each user thread
- Room, a library about SQLite DB has been used to save data regarding health, position and emergency calls.
- Android bluetooth to communicate with the device collecting health data
- Android espresso to test mainly the correct execution of the emergency calls (e.g. if a real call is performed or not)
- support library package: adds support for the Action Bar user interface design pattern and it includes support for material design user interface implementations.

4 Structure of the source code

4.1 Microservices

Here the code regarding the microservices architecture is explained.
The source code that meets the requirements mentioned above has been organized in the following way: for each microservices a project has been set up. Indeed, when dealing with this type of architecture, one should think of a microservice as a project that should be as much independent as possible from the others: this is the reason that stands behind the choice that has been made. Of course, in this way, it is possible to easily generate the single jars that will be deployed, when necessary, with, as mentioned in the design document, dockers. Therefore, the following projects are present: API gateway, service registry, group individual request service, individual request service and share data service. As one may notice, the one containing the set up of the API gateway also accesses all the information related with the accounts, and, therefore, authentication and authorization functions are coded here. In the following sections the structure of the single projects are analyzed.

4.1.1 API Gateway

In the figure 1, it is shown the root of the project structure. An analysis of the various elements follows below the figure.

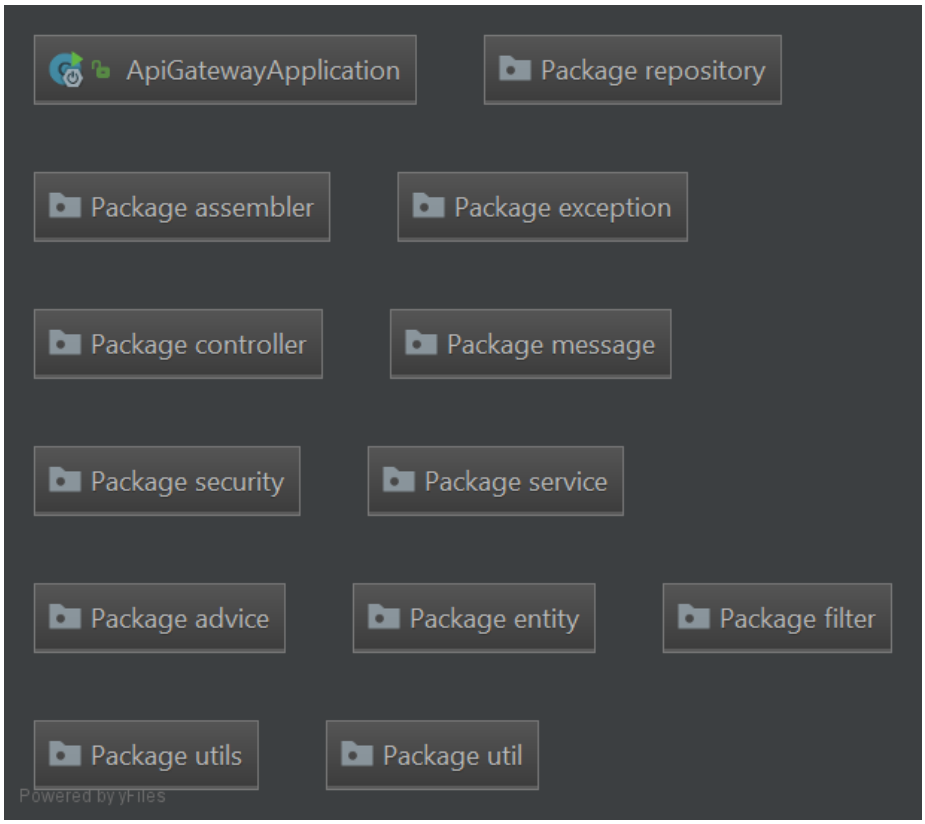

Figure 1: API gateway

- Package repository: contains the JPA repository for accessing the persistent data necessary to this service. In particular information regarding the accounts of users and third party customers are present. For what concerns the third party customers, since they can register both as private and as related with a company the following repositories are present: company details, private details, third party customers and users. An additional repository is present, and it accesses information regarding the API. Indeed, all the accessible APIs that are available are stored in a database, in order to provide access control. For better specifying this choice, one may consider the fact that it is not necessary to search on the service registry non-existing API or to forward requests that can be already classified as rejected (e.g. user A that is trying to access an API available only for third party customers)
- Package assembler: this class contains components useful to build HATEOAS resources of entities that are returned to clients, adding hypermedia contents
- Package exception: this contains the custom exceptions defined during the development
- Package controller: contains the controller that defines the APIs that regards the management of the users and third party customer accounts.

From here, it is possible to access the business functions that regards the account (e.g. login, logout, registration). Inside here, controller are split into secure and public controllers: the public one are accessible to everyone, while for accessing the secured ones, it is necessary to perform the log in

- Package message: it provides the functions of communicating with other microservices, by means of RabbitMQ. In particular, three subpackages are present here. The package publisher contains classes that helps in publishing the events of creation of new accounts to other services. The package protocol defines a way of communicating the interested pieces of information, and, finally the configuration package specifies the configuration settings needed to convert object from JSON (and viceversa) during the propagation of data and also RabbitMQ exchanges and queues[5] .

- Package security: contains the main features that have been introduced above regarding the security

- Package service: contains the services that implements the business functions of the account service, with all the APIs that it exposes. The interfaces present here map the component interfaces of the account service

- Package advice: this encompasses the handling of server side exceptions in order to show to clients useful messages without exposing the structure of the project and low level errors

- Package entity: here classes that are mapped to the databases are present

- Package filter: it includes filters that the API gateway applies during the management of "external" requests (i.e. requests that need to be processed by other services). In particular, it is further subdivided into three packages: pre, route and post. In this case, the pre filter that is present perform access control on the external API that is being requested, the route filter is a translation filter that adds header in order to identify the clients in the other microservices, and the post filter fixes the hypermedia content that is sent as a response to the original request

- Package util: various utility needed in the development

The routes to the other microservices available are present in the application.properties file. Here it follows a list of the API available:

- /public/users/authorize
  This requires two request parameters, that are username and password. This API is necessary to login the users. The method is POST.

- /public/users/{ssn}
  This is the entry point for registering a new user. Here, ssn is a path variable that specifies the social security number of the user that will register. A request body that specifies the necessary fields of the JSON are specified in the user entity.
  Here it is reminded that one should check the needed fields paying attention also to the defined JSON views defined.

This holds also for the other methods in which a request body is needed. The method is POST.

- /public/thirdparties/authorize
  This requires two parameters, that are email and password. This API is necessary to login the third party customers. The method is POST.
- /public/thirdparties/companies is necessary to register a third party customer that is related with a company.
  A request body is needed to specify all the information related to that customer. The method is POST.
- /public/thirdparties/privates is necessary to register a third party customer that is not related with a company. A request body is needed to specify all the information related to that customer. The method is POST.
- /users/info returns the information of the user that is accessing that API. The method is GET.
- /users/logout performs the logout of the user from the system. The method is POST.
- /thirdparties/info returns the information of the third party customer that is accessing the method. The method is GET.
- /thirdparties/logout performs the logout of the third party customer from the system. The method is POST.

4.1.2 Service registry

The project of the service registry is almost empty, no package is present, but just an application class annotated with @EnableEurekaServer. The configuration of the service registry is set in the application.properties file.

4.1.3 Group request service

In the next figure, the package diagram of the group request project is shown 2.
The same comments hold here except for the fact that the business function mapped in the component interfaces are the one regarding the group request service.

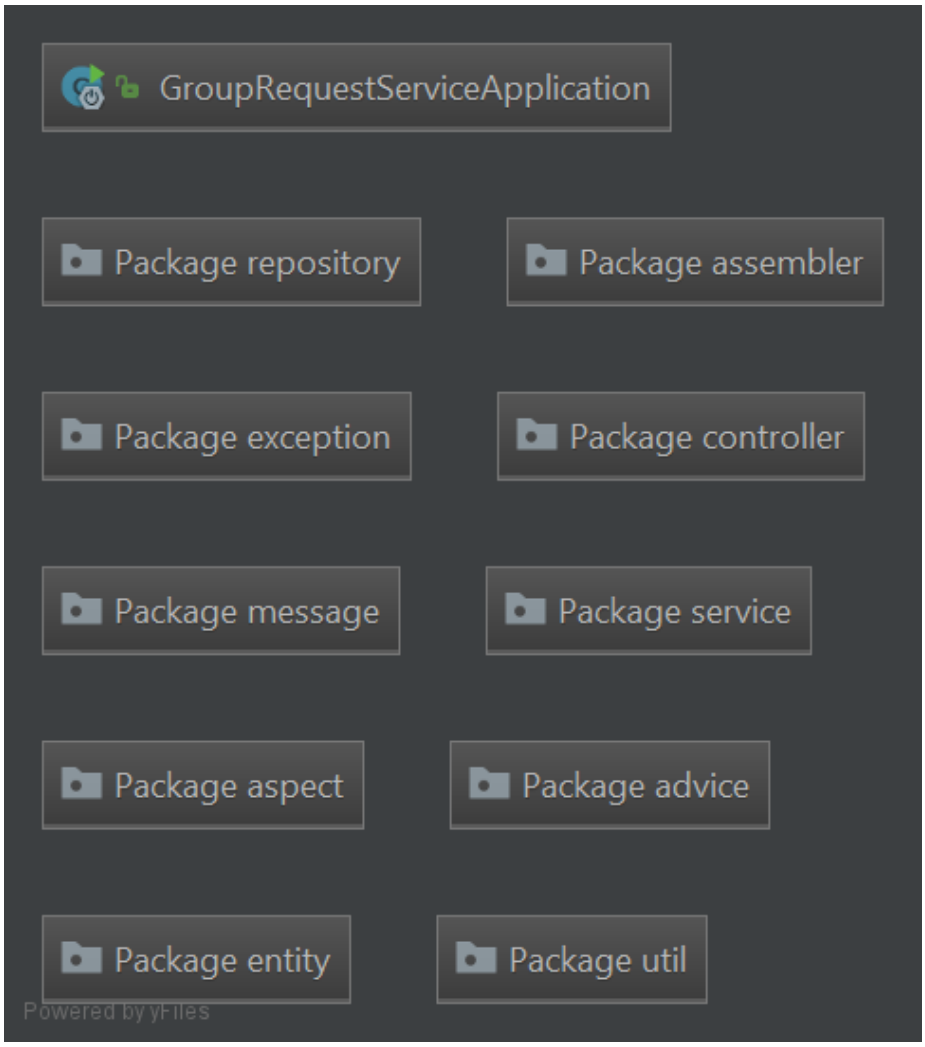

Figure 2: Group request service

Here it follows a list of the API available:

- /grouprequestservice/grouprequests/id/{id}
  It retrieves the information related with the group request specified by the id in the path variable. The method is GET.
- /grouprequestservice/grouprequests/thirdparties
  It collects all the group requests related with the third party that is accessing the method. The method is GET.
- /grouprequestservice/grouprequests/thirdparties
  This adds a new group requests. It requires a request body that contains all the details of the group request that will be added. The method is POST.

The prefix /grouprequestservice is necessary in order to access the APIs from the API gateway. Note, that by inspecting the controller, the prefix won't be

shown at all: this is because the API gateway does this translation and mapping autonomously. Instead, one may notice some request headers: they are added at runtime by the gateway, and so the client does not have to specify them, and, if it does, they will be overwritten.
This comment holds, as well, for all the other microservices.

4.1.4 Individual request service

Here it follows the package diagram of the individual request service project 3. As one may notice, the structure is very similar to the one explained in the api gateway project. Of course, here, the controllers provide access to the APIs that regards the individual request service. The business functions of this project are present in the service package, and the interfaces that are present, are mapped with the component interfaces of the Design document.
Another difference w.r.t. the API gateway that is worth to point out is that here the package message contains also a subpackages that defines listeners: these are charged of listening to new events that are forwarded from RabbitMQ.

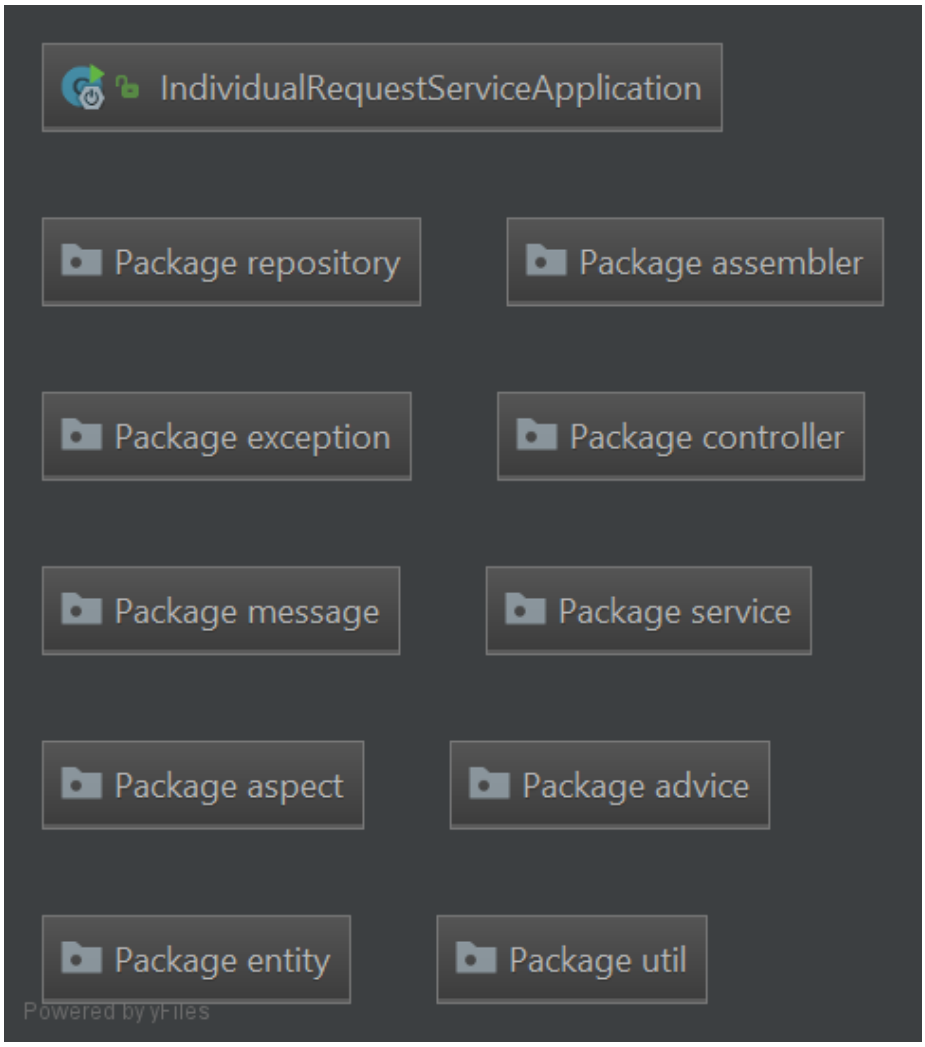

Figure 3: Individual request service

Here it follows a description of the available API:

- /individualrequestservice/requests/id/{id}
  This retrieves an individual request identified by means of a certain id. The method is GET.
- /individualrequestservice/requests/users
  This retrieves the pending requests of the user that is accessing the method. The method is GET.
- /individualrequestservice/requests/thirdparties
  This retrieves the requests performed by the third party customer that is accessing the method. The method is GET.
- /individualrequestservice/requests/{ssn}
  This adds a new individual request toward the user specified in the path

variable. A request body specifies the information requested to define the individual request. The method is POST.

- /individualrequestservice/responses/requests/{requestID}
  It is used in the case in which the user that is accessing the method sends a response to the request identified by requestID in the path variable.
- /individualrequestservice/responses/blockedThirdParty/thirdparties/{thirdParty}
  It is used in the case in which the user that is accessing the method blocks the third party identified by the id specified in the path variable

4.1.5 Share data service

The structure of the share data service, as shown by the following picture, is also very similar to the previous ones 10.

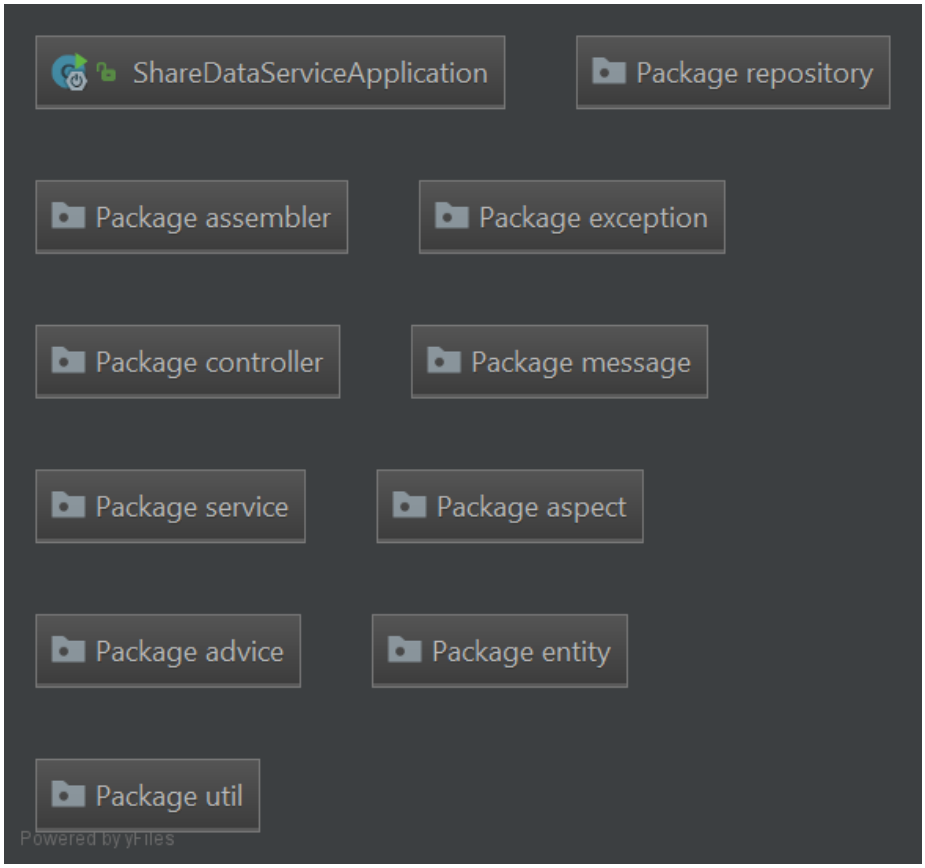

Figure 4: Share data service

However, it is important to point out the structure of the code that manages the personalized query specified by the group requests on anonymized data. Basically, the custom query is realized with the help of QueryDSL, a sort of unified library to make queries; this choice comes from a limitation of Spring JPA: the impossibility to make union with dynamic queries. Therefore, after

researches of other libraries supporting JPA, the best one and mostly supported by the community was QueryDSL which offers a method to makes JPA SQL queries. First of all it is necessary to say that JPA creates connection between objects of classes and tuples of the DB; this has great advantages when it comes to use only JPA, but mixing it with other methods of retrieving data such as raw SQL queries could introduce some inconsistency. But thanks to JPA SQL queries it was possible to avoid this problem. In conclusion, the custom query is generally something with this simplified template:

- SELECT AGG_OP(Column) FROM User JOIN ON Union(HealthData dynamic filtered, PositionData dynamic filtered) WHERE (sameSSN) AND "other filters of User table"

Here it follows a list of the main API that have been developed in the project, and that are located within the various microservices. It is possible to find these mapping in the controller packages.
The methods are expressed from a client point of view: for example, takes into consideration the methods that starts with s

Here it follows a description of the available API

- /sharedataservice/dataretrieval/individualrequests/{request_id}
  This method retrieves the data related to the individual request identified by the path variable. The method is GET.

- /sharedataservice/dataretrieval/grouprequests/{request_id}
  This method retrieves the data related to the group request identified by the path variable. The method is GET.

- /sharedataservice/dataretrieval/users
  This method retrieves the own data of the user. In this case, two request parameters are necessary in order to specify an interval of time that involves the interested data. The method is GET.

- /sharedataservice/datacollection/healthdata
  This allows the user to send data regarding the health status to the system. The method is POST. A request body that specifies the data required.

- /sharedataservice/datacollection/positiondata
  This allows the user to send data regarding his position data to the system. The method is POST. A request body that specifies the data required.

- /sharedataservice/datacollection/clusterdata
  This allows the user to send cluster of data (both health and position data). The method is POST. A request body that specifies the data required.

4.2 Mobile code

4.2.1 Data4Help

The source code of the mobile application has been divided in different packages. The main idea is to build an application where each component represents a microservice's view: they are loaded dynamically and they are assembled for drawing the complete user interface. This solution, even if it uses a monolithic

approach, works well for relative simple mobile applications and it is more easy to implement. Of course, in this way, it is possible to easily generate the single APK that will be installed on the mobile device.
It follows an analysis that will focus on what is a service component and how it has been implemented in the code. In the project, a user interface component is a container, that is represented by an activity class. This classes follow the MVP pattern with delegation, thus allowing to have activities less complex and well structured, since the base structure of an android application with the only uses of simple activities is a bad practice. The MVP pattern is matched against the baseUtility package and his main components are: contract (basePresenterInterface and baseViewInterface), basePresenter, baseActivity and baseDelegate. The fragment that truly represents the microservice's view are loaded into the activities.

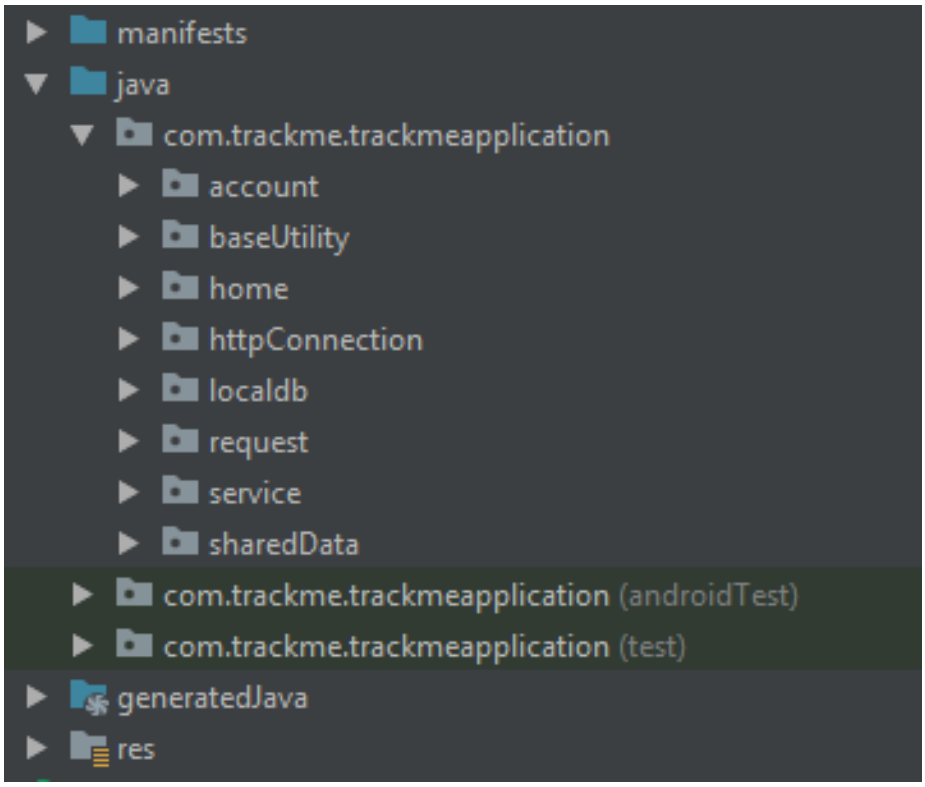

Figure 5: Code structure

In the following paragraph the structure of each package is analyzed.

Account In the next figure, it is shown the account package structure and, after that, comes an explanation of the single parts.

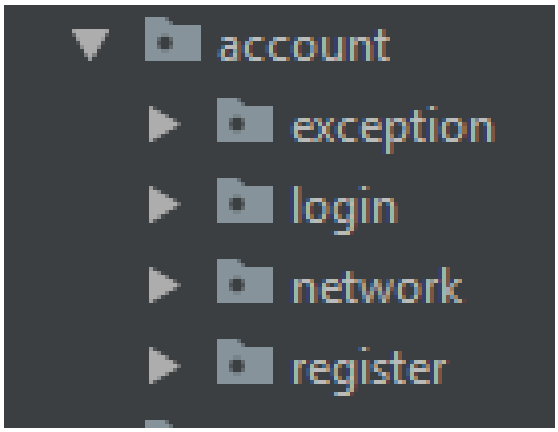

Figure 6: Package account

1. Package exception: this contains the custom exceptions defined during the development for the account service.

2. Package login: this contains the activity for the user login and the third party login. The user login is also the first activity when the application starts. It saves the user context in a persistent way by means of SharedPreferences class, and it requires the user all the permission that the application needs to work properly, like: access to the location, internet, call phone, bluetooth and the possibility of writing on external storage.

3. Package network: this package contains the account controller called accountNetworkInterface that exposes all the functions for the communication with the account service on the server side.

4. Package register: here all the register forms are implemented to allow third parties (i.e. company and private third party) and users to register into the application.

Base Utility The package base utility contains the implementation of the MVP pattern and the main utility used in the application. Furthermore, it contains the constant class with all the useful constants.

Home The home package contains the two main activities of the application. The UserHomeActivity shows the home view to the user, with the possibility of switching through three fragments: the home fragment that allows the user to see his last health status, by pressing the check status button; the history fragment with all the health data registered in the last week; and, finally, the request fragment that allows the user to see the individual requests received and to accept or refuse some of them.
The user home provides also a lateral menu, with the logout function and the possibility of activating two services: health and location service. Note that the user settings and the user profile are only fake options, added for the completeness of the menu, but they are not so important, and therefore they have not

been implemented: more time has been devoted to other pieces of functionality. Also Term and Condition is a fake view.
The second activity that is present in this package is the business home: it is similar to the first, but it is composed of only two fragments: one for showing to third parties the individual requests that they have sent and allow them create additional requests, and a second one that does the same, but for the group requests.
In this package the synchronization between the slider that activates health and location service with the running of the service itself has not been implemented since it is not so important to the prototype application.

HTTP Connection This package contains the main class for supporting a secure connection with the server. Connection thread is the only way to connect to the server and it loads, for all the connections, the SSL context with the certificate and the host verifier that checks if the server IP corresponds to an address known.

Local db The structure of the local db realized with Room framework is now exposed.

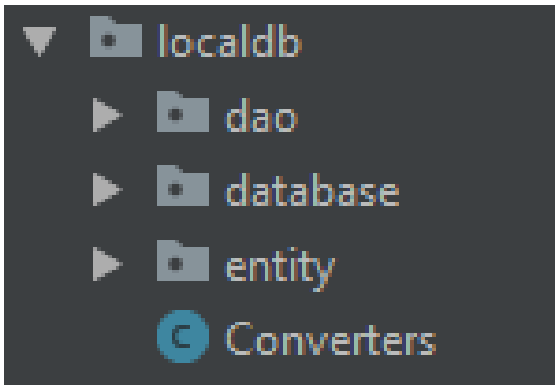

Figure 7: Package local db

1. Package dao: It contains interfaces that maps java function in SQL query.
2. Package database: this class contains the database abstract class.
3. Package entity: here classes that are mapped to the databases are present

Request In this paragraph the request package is presented. This package implements the client side of individual request service and group request service.

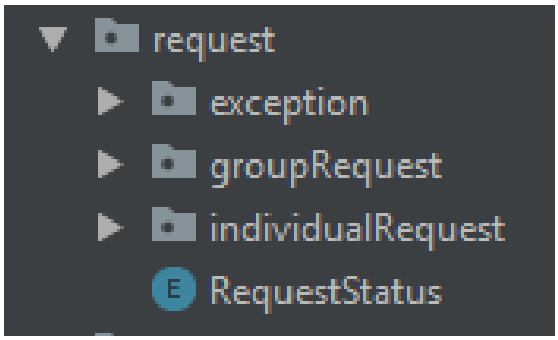

Figure 8: Package request

1. Package exception: this contains the custom exceptions defined during the development.
2. Package individual request: here it is implemented the individual request user interface with fragments that can be loaded in the home containers and the wrapper object that allows Object mapper to map immediately JSON strings to request objects.
3. Package group request: same as above, but for group request.

Service In this package, the health service and the location service are implemented in order to receive data about health and position from, respectively, smartwatch or other similar devices and GPS. The application exploits the bluetooth functionality of the mobile device to receive the health data from the smartwatch, while for what concerns position data, it just receives it from the Location Manager of the device.

This package regards also AutomatedSOS and it is completely developed to work locally on the smartphone of the user.

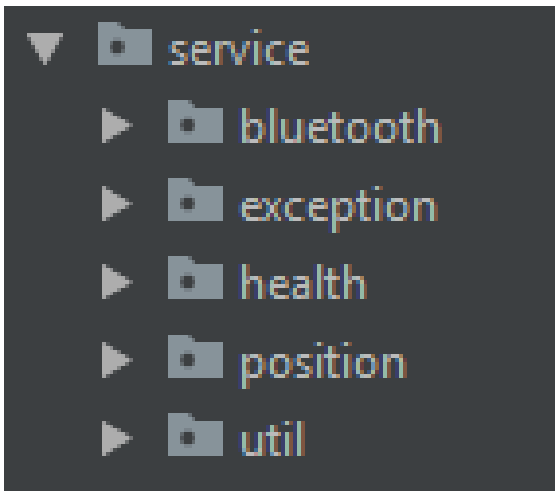

Figure 9: package service

1. Package bluetooth: it contains a Bluetooth Server which is responsible of managing Bluetooth clients

2. Package exception: it contains all the custom exceptions used in the package

3. Package health: it contains the logic behind the health service which is based, basically, on the following: receiving the message; saving the data if it is correct w.r.t. standard values; checking with simple threshold if the health is grave or not; if it is grave and there are no recent emergency call, then it will get the user location and search for the emergency number within a JSON and call it (afterwards, it saves the call on the local database to avoid flood calling)

4. Package util: it contains all the utilities class necessary for the health and position package. Here, it is important to highlight the HealthDataInspectorImpl class in which it is defined the threshold of each field of health data:

   - heartbeat: the standard interval is $[30, 220 - age]$;
   - minimum pressure: an interval of $[40, 100]$;
   - maximum pressure: an interval of $[80, 200]$;
   - blood oxygen level: it is a percentage of interval $[80, 100]$.

   All these threshold are collected by an expert. If one of these thresholds are violated (excluded the case in which the health data is incorrect, i.e. bloody oxygen level not a percentage from 0 to 100), then the health data is considered to be grave and an emergency point should be called.

5. Package location: it contains the logic behind the location service, which just simply saves the position data on the local database every time a new last location is present

Share data Finally, the share data package is commented. The package structure is the same: it contains an exception package with the custom exceptions; it contains a network package with the controller interface that exposes the function that allows the communication with share data service, and, finally, all the wrapper classes for the mapping of the objects with JSON strings and vice versa.

Manifest folder This folder contains the application manifest that contains the properties of the application, like the definition of activities and user permission needed.

Res folder Here are located all XML definitions files presented into the software. This files defines the aspect of the user interface and all the components styles.

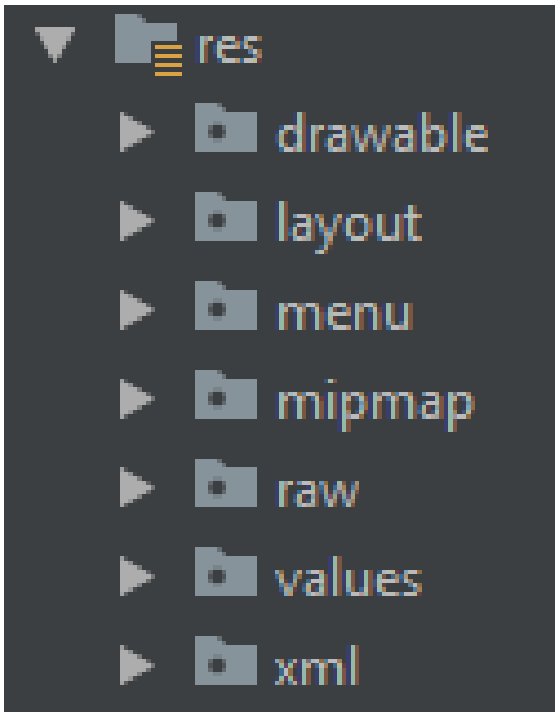

Figure 10: Res folder

1. Package drawable: this package contains the XML definition of the drawable objects like images and icons.
2. Package layout: as the name says, it contains every activity layout. Note that the application is developed with automatically resizable layout and it supports portrait and landscape orientations.
3. Package menu: here there is the menu layout for the drawable menu on the left of the application.
4. Package mipmap: it contains the application icon.
5. Package raw: It contains two particular files. A JSON file with the last emergency call made by the user with Data4Help and keystore with the keys for secure communication.
6. Package value: here are stored string values, colors and component styles.

5 Testing

5.1 Microservices

5.1.1 Tests of a single microservices

For managing the tests, in general, during the development, once a feature was ready it was tested. The procedure described in the Design document was followed; however, due to the fact that many spring components and classes

do already many things, unit tests of single class sometimes didn't make much sense, because it would have been necessary to mock many Spring features, that are supposed to be already tested and working properly.
For the projects, the process of testing usually followed this life cycle: repositories have been tested firsts, then the services, that cover the business functions of the project under consideration, and then the controllers, for which two kinds of tests were performed: a unit and an integration test. These integration tests involves all the stacks that were developed in the projects: indeed, from here, the connection to the APIs with HTTP/ HTTPS requests are tested: this methods usually call the business functions, that modifies the databases by means of repositories. If error occurs, the classes in the advice packages come into help, and, therefore, they are tested as well.
For what concerns the tests that regards interfaces that allows to share events with other microservices, as soon as it was possible to have them working they were tested as well. However, it is worth to note that the real integration between microservices have been tested with JMeter. This is because there were multiple projects and an integration test with JUnit should have been done on one of the two or more projects to be integrated. This was a little bit difficult to approach and moreover, what the test would have checked was the correct execution of the message broker which is already reliable. The same things holds for the test of the service registry and for the forwarding of requests to other microservices by means of the Zuul gateway.
The approach taken for all the tests mentioned above (the ones with JMeter excluded), is white box testing. Here it follows a brief report on the coverage, with some comments. The images are taken from the report generated by JaCoCo.

First of all the coverage of the API gateway is shown.

api-gateway

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.poianitibaldizhou.trackme.apigateway.service		96%		94%	4	53	0	130	2	34	0	4
com.poianitibaldizhou.trackme.apigateway.util		89%		100%	3	9	4	23	3	7	2	5
com.poianitibaldizhou.trackme.apigateway.controller		98%		75%	1	17	1	52	0	15	0	4
com.poianitibaldizhou.trackme.apigateway.security		100%		n/a	0	21	0	60	0	21	0	5
com.poianitibaldizhou.trackme.apigateway.message.publisher		100%		n/a	0	7	0	38	0	7	0	2
com.poianitibaldizhou.trackme.apigateway.advice		100%		n/a	0	14	0	40	0	14	0	7
com.poianitibaldizhou.trackme.apigateway.message.protocol		100%		n/a	0	3	0	6	0	3	0	2
com.poianitibaldizhou.trackme.apigateway.message.configuration		100%		n/a	0	17	0	23	0	17	0	3
com.poianitibaldizhou.trackme.apigateway.assembler		100%		n/a	0	6	0	11	0	6	0	3
com.poianitibaldizhou.trackme.apigateway.exception		100%		n/a	0	5	0	10	0	5	0	5
Total	37 of 1.730	97%	3 of 46	93%	8	152	5	393	5	129	2	40

Figure 11: API gateway coverage

As one may notice, the entity and the filter packages are not present. Indeed, filters have been tested with JMeter, and therefore are not present here. Furthermore, entity classes are basically only attributes and methods that are not shown since they are auto generated with lombok.
Finally, all the other lombok auto generated methods have been excluded from the coverage.

Similar reasoning applies also to the coverage of the individual request service.

individual-request-service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.poianitibaldizhou.trackme.individualrequestservice.message.protocol		65%		n/a	1	4	2	8	1	4	1	3
com.poianitibaldizhou.trackme.individualrequestservice.service		97%		97%	3	47	0	106	2	26	0	3
com.poianitibaldizhou.trackme.individualrequestservice.advice		93%		n/a	1	24	4	70	1	24	0	12
com.poianitibaldizhou.trackme.individualrequestservice.controller		100%		91%	2	20	0	52	0	8	0	2
com.poianitibaldizhou.trackme.individualrequestservice.assembler		100%		n/a	0	6	0	32	0	6	0	3
com.poianitibaldizhou.trackme.individualrequestservice.util		100%		n/a	0	6	0	32	0	6	0	5
com.poianitibaldizhou.trackme.individualrequestservice.message.configuration		100%		n/a	0	22	0	26	0	22	0	4
com.poianitibaldizhou.trackme.individualrequestservice.exception		100%		n/a	0	10	0	20	0	10	0	10
com.poianitibaldizhou.trackme.individualrequestservice.message.publisher		100%		n/a	0	4	0	24	0	4	0	1
com.poianitibaldizhou.trackme.individualrequestservice.message.listener		100%		100%	0	10	0	22	0	6	0	2
com.poianitibaldizhou.trackme.individualrequestservice.aspect		100%		n/a	0	3	0	10	0	3	0	1
com.poianitibaldizhou.trackme.individualrequestservice.message.protocol.enumerator		100%		n/a	0	1	0	2	0	1	0	1
Total	74 of 1.872	96%	3 of 74	95%	7	157	6	404	4	120	1	47

Figure 12: Individual request service coverage

Same things hold also for the group request service.

individual-request-service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.poianitibaldizhou.trackme.individualrequestservice.message.protocol		65%		n/a	1	4	2	8	1	4	1	3
com.poianitibaldizhou.trackme.individualrequestservice.service		97%		97%	3	47	0	106	2	26	0	3
com.poianitibaldizhou.trackme.individualrequestservice.advice		93%		n/a	1	24	4	70	1	24	0	12
com.poianitibaldizhou.trackme.individualrequestservice.controller		100%		91%	2	20	0	52	0	8	0	2
com.poianitibaldizhou.trackme.individualrequestservice.assembler		100%		n/a	0	6	0	32	0	6	0	3
com.poianitibaldizhou.trackme.individualrequestservice.util		100%		n/a	0	6	0	32	0	6	0	5
com.poianitibaldizhou.trackme.individualrequestservice.message.configuration		100%		n/a	0	22	0	26	0	22	0	4
com.poianitibaldizhou.trackme.individualrequestservice.exception		100%		n/a	0	10	0	20	0	10	0	10
com.poianitibaldizhou.trackme.individualrequestservice.message.publisher		100%		n/a	0	4	0	24	0	4	0	1
com.poianitibaldizhou.trackme.individualrequestservice.message.listener		100%		100%	0	10	0	22	0	6	0	2
com.poianitibaldizhou.trackme.individualrequestservice.aspect		100%		n/a	0	3	0	10	0	3	0	1
com.poianitibaldizhou.trackme.individualrequestservice.message.protocol.enumerator		100%		n/a	0	1	0	2	0	1	0	1
Total	74 of 1.872	96%	3 of 74	95%	7	157	6	404	4	120	1	47

Figure 13: Group request individual service coverage

For what concerns the share data service, here follows the image of the coverage. The same comments are valid also in this situation.

share-data-service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.poianitibaldizhou.trackme.sharedataservice.util		99%		100%	2	31	2	124	2	29	2	14
com.poianitibaldizhou.trackme.sharedataservice.service		98%		100%	1	32	0	124	1	31	0	3
com.poianitibaldizhou.trackme.sharedataservice.message.protocol.enumerator		100%		n/a	0	6	0	13	0	6	0	6
com.poianitibaldizhou.trackme.sharedataservice.repository		100%		n/a	0	7	0	51	0	7	0	1
com.poianitibaldizhou.trackme.sharedataservice.controller		100%		100%	0	14	0	40	0	8	0	2
com.poianitibaldizhou.trackme.sharedataservice.message.listener		100%		100%	0	17	0	49	0	10	0	3
com.poianitibaldizhou.trackme.sharedataservice.message.protocol		100%		75%	1	7	0	11	0	5	0	5
com.poianitibaldizhou.trackme.sharedataservice.message.configuration		100%		n/a	0	27	0	34	0	27	0	5
com.poianitibaldizhou.trackme.sharedataservice.assembler		100%		n/a	0	8	0	17	0	8	0	4
com.poianitibaldizhou.trackme.sharedataservice.advice		100%		n/a	0	8	0	24	0	8	0	4
com.poianitibaldizhou.trackme.sharedataservice.aspect		100%		n/a	0	3	0	10	0	3	0	1
com.poianitibaldizhou.trackme.sharedataservice.exception		100%		n/a	0	4	0	11	0	4	0	4
com.poianitibaldizhou.trackme.sharedataservice.message.publisher		100%		n/a	0	3	0	11	0	3	0	1
Total	12 of 3.040	99%	1 of 36	97%	4	167	2	519	3	149	2	53

Figure 14: Share data service coverage

5.1.2 JMeter

In order to do system testing on the server side of the project, JMeter has been adopted. There are many motivation for this choice: open-source application, user friendly (GUI), and many functionality such as the possibility of doing functional testing, load testing. Moreover it provides the possibility of testing almost every kind of resources: databases, message brokers, HTTP REST APIs and many other. In the end, to check if the system as a whole was working as expected, the following test plan has been written:

- Functional testing: without taking into consideration the concurrency and performance problems, the test plan focuses on the expected return value of each request, to test if the system works as intended. It has also been checked the group request accepted case when a group request is correlated

to more than 1000 users. The test, basically, does not check the response body of each request, but it just checks the correct flow of the server application by making request in which the status code of each request is of type 20X.

- Load testing: with the help of the "Random Controller", a random choice in the HTTP requests is applied and many thread users have been launched to test the maximum load of the system. In the GUI mode, the test results in a latency of more than 20 seconds which the API Gateway discards due to a timeout. If the test has been done in a non-GUI mode, the average latency of a request would be less.

- Real simulation testing: it is just a variation of the load testing with the ThinkTime added: a simulation of a possible think time of the user in order to make a request.

There are some considerations taken during the execution of the test plans:

- Since the server application is an eventually consistent system, then for example if a user register the api-gateway add immediately the user, but the other services will be consistent with this new information only when they will receive the message from the message broker (RabbitMQ) and after consumed it. Therefore some heuristics waiting time has been added on the test to wait for the eventually consistency of the system.

- The test has been done locally with many applications opened, which could have interfered with the load testing.

- The services has been deployed all on the same machine which was different from the deployment diagram.

- Every time a test is launched, all the queues of the message broker are purged and the all tuples inside the DBs are deleted.

- Remember that the server has a timeout of one minute for each request, after that the server will stop the request.

- The time and date are not synchronized on a unique timezone or something like that with the clients (jmeter or app), due to complexity, therefore it is important to note that if the test is done within one hour before the midnight some tests could fail since the timestamp saved on the server could leap by one hour.

5.2 Android application

The test that has been done here are only about the health service which it is responsible to the Automated SOS service. This is the most important test since it could be a problem of life and death for the person using the application. Therefore, many test has been done:

- Unit tests of simple POJO classes that can be found in the folder app/src/test

- Android tests, which are more complicated tests that need to be launched on real devices or emulators: from these test, there are unit tests, integrated tests and performance test. The last one it simply checks the average time to execute a call when a bluetooth message containing a grave health data is received. Other tests just checks the functionality of the classes with the exception of bluetooth which is really difficult to test since the emulator cannot simulate a bluetooth connection.

6 Installation instruction

6.1 Microservices

First of all, let's check all the prerequisites needed to install and run the software:

- RabbitMQ is necessary, since a RabbitMQ server is needed to make the communication between microservices properly working. It is possible to download it from there: `https://www.rabbitmq.com/download.html`. Once the page is opened, click on the links on the right, selecting the right operative system and download the installer. When working with windows, it may be possible that the installation process throws an error that expresses the necessity of downloading the Erlang programming language. In this case, just follow `http://www.erlang.org/downloads` and download Erlang for the version of Windows that you are using

- MySQL is another critical component since it has been used as a DBMS for managing persistent data. For having this working, it is possible to follow the guide at : `https://beep.metid.polimi.it/documents/121843524/b5d81926-98f6-496f-bf45-a2a8e897228c` (ignore all the chapter regarding NetBeans and follow the ones that deal with the installation of MySQL). Once that the server has been set up, it is necessary to create the databases and its admin: to do that, first of all, go to the MySQL folder and login with root. After that, write:

  ```
  create user 'trackmeadmin'@'%' identified by 'datecallmeeting95';
  ```

  Now, the admin for the database has been created, and it is necessary to create all the databases used by the various microservices.
  In order to accomplish this, run:

  ```
  create database share_data_db;
  create database group_request_db;
  create database individual_request_db;
  create database account_service_db;
  ```

  Finally, for each database created, it is necessary to run this command:

  ```
  grant all on db_name.* to 'trackmeadmin'@'%';
  ```

where db_name has to be substituted with share_data_db, group_request_db, account_service_db and individual_request_db.

Once that all the previous steps have been accomplished, and once you have downloaded all the jars from the links present in the front page of this document, start MySQL server (following the guide mentioned above), and after that start the RabbitMQ server. In order to do that, navigate to that folder in which RabbitMQ have been installed, search for "rabbitmq-server" and launch it (if you prefer other ways to launch the server you may check the link provided before: indeed, the links to the installation instruction of the various operative systems, provide also information on how to set up the server).
After that, it is possible to start the downloaded jars by means of:

```
java -jar path/to/file.jar
```

It is necessary to start first of the service registry, and then all the other services (and, of course, also the API gateway), in the order you prefer. The port that are needed to be free are the following: 8443 (API gateway), 8761 (service registry), 8089 (share data service), 8081 (group request service), 8082 (individual request service). However, if for some reasons one of these ports is busy on the machine in which the jar will be ran, it is possible to modify the port by adding "–server.port=PORT_NUMBER", while launching the jar. Moreover, to expose the server to the mobile application it is important to change the server address with the local IP of the machine by adding to the api gateway jar launch "–server.address=LOCAL_IP"
Furthermore, is also possible to modify any of the settings that are located within the application.properties of the various project. One can find these files in the various folders of the source code, within the folder "main". However, in order to avoid useless errors we suggest to not modify critical parts, such as the routes used by Zuul (API gateway settings).

For running the tests, it is necessary to start again the RabbitMQ server and the MySQL server. After that, it is possible to go into the root of interested project and run "mvn test".
Keep in mind that running the tests may take some time, due to the fact that some tests require to launch the entire spring application, or to connect to the RabbitMQ server, or to load data into the database. In case there has been some errors due to integration tests with RabbitMQ, it can help to open the web server of RabbitMQ: localhost:15762, enter as guest with the password "guest" and purge the queues if necessary.

Here it follows a list of further considerations:

- If, for some reason, it is necessary to open the source code in some IDE, keep in mind that it may be necessary to download the Lombok plugin. In the case in which you are using Intellij, this link may be useful: `https://projectlombok.org/setup/intellij`
- Another thing that one may take into consideration, is the fact that when downloading the source code, one may have problem with too long paths when extracting the folder. This is a problem that have been encountered only extracting the zip, and that is fixed by using tar.gz format.

- When running tests that are provided with the source code, remember to not having started the application as a whole, or at least not the one that involves the test that is going to start. This is because the same service which is running could steal the messages from the test execution environment and, therefore, conflicts can happen, leading to some bugs. Furthermore, there is a timeout set to 1 minute, regarding the exchange of messages in the message queue. Keep in mind that you are probably running an entire architecture on a single machine, therefore don't get surprised if things get stuck a bit. However, we didn't encountered many problems in this sense.

- If you are opening the source code in your IDE (e.g. IntelliJ) and you run the program from there, you should also configure the databases inside the IDE. In this sense, since all the microservices are different projects, it is necessary to link each database to its related project.
  A final remark is that the account service db is related to the API gateway; for what concerns the others, instead, the binding is trivial.

- Due to setting of the application.properties file, every time that a jar is launched the database will be created and therefore it will need to be re-populated. A little exception holds for what concerns the API gateway: indeed a SQL script named "data.sql" will be executed loading a table that contains information regarding the available API.
  If there is the necessity to avoid creation of a new database, then when launching the jar it is possible to add "–spring.jpa.hibernate.ddl-auto=update" to set the hibernate to update mode.

In order to run the test plans of JMeter it is necessary to have JMeter installed. Then since RabbitMQ queues has to be purged and the DBs has to be deleted, it is necessary to add some libraries:

- mysql-connector.jar (`https://dev.mysql.com/downloads/connector/j/`) to be moved into the lib folder of JMeter

- amqp-client-3.x.x.jar (`https://www.rabbitmq.com/java-client.html` to be moved into the lib folder of JMeter; remember to download a version starting with 3, e.g. 3.6.6 (`http://repo1.maven.org/maven2/com/rabbitmq/amqp-client/`)

- Download also the project `https://github.com/jlavallee/JMeter-Rabbit-AMQP` and build it with Ant ("brew install ant" for Mac users). Then move the generated jar file, which can be found in /target/dist, into the lib/ext folder of JMeter. To avoid searching for this jar, it is possible to download it directly on the release section of GitHub

After this configuration, it is possible to run the test plans of JMeter on GUI/non-GUI mode:

- GUI Mode: to launch this mode, just double-click on the jmeter executable or launch it with the terminal; in order to run the test, it is possible to just click on the green play button on the top bar. After that it is possible to see the results by clicking on the node: "View Results Tree" or "View Results on Table" located in every thread group

- non-GUI Mode: To launch this mode, just run this code on the terminal: "jmeter -n -t [jmx file] -l [results file] -e -o [Path to web report folder]" which will output a result file of the test and a html web page report on the folder chosen where there will be graphic representation of the results of the test. If the command does not work properly it is possible to run the code separately: first "jmeter -n -t [jmx file] -l [results file]" and then run "jmeter -g [results file] -o [Path to web report folder]".

In case the api-gateway service is running on a IP different from localhost, then it is necessary to change the IP address of the test from the GUI: just click on the HTTP request default that can be found on the left side of the graphical interface and change the IP address with the one chosen.

6.2 Android application

In order to install the application onto your device it is necessary to allow the installation of applications from unknown sources. This setting can be found in your device's system settings under: Settings > Applications > Unknown sources.
Now it is possible to download the apk from the project folder on GitHub and after that, it is possible to install it by launching the Downloads application. The Downloads application can be launched from web browser using the menu, or by launching the Downloads applications directly by the download folder on your device. During the first launch of the application the user must accept all the permission required. Note that if some are refused then the application will be closed. For the first execution the user should set the ip address of the server by pressing on the TrackMe logo in the user login. A window with the current ip address is shown and here it is possible to change the address with the one of the machine in which you are running the jars (it is important to have all the jars launched: api-gateway must be running on the local ip address e.g. 192.168.1.X and the mobile device has to be on the same local network).

The possible operations at this point are:

- Register as third party: just click register
- Register as user: click on register and then it is important that the ssn is in a specific form: AAA-BB-CCCC which is composed by only numbers (e.g. 000-01-9830)

After this it is possible to login in the application as third party or user. Afterwards the user interface is quite friendly.

For simulating the smartwatch it is also necessary to download the bluetooth client APK from the GitHub folder and install it, by following the same instruction reported above.

Here it follows a list of further considerations:

- If, for some reason, it is necessary to open the source code it may be necessary to download Android Studio and set up the emulator.

If the reader does not have a mobile phone with android he can use an emulator on his pc, but the bluetooth connection cannot be simulated with an emulator.

Regarding the tests of the health service, all the tests should be launched on an android simulator with some tweak configurations to do:

- Enable the developer settings by tapping a lot of time on the build settings of the simulator.
- Enable the mock position for the application through the developer settings.

Moreover, the test regarding the bluetooth does not exist due to complexity: the simulator does not have the capacity to simulate bluetooth connections. Instead, there is a stub application bluetooth client which can be used to test if the communication between bluetooth devices is working as expected. Therefore, it is necessary to have two devices supporting bluetooth which are paired. Then after starting the health service on the device containing TrackMe application, a bluetooth server is running in background waiting for other devices. Now after starting the other application, it should be possible to insert the name of the device running the server and establish the connection. Once the connection is established it is possible to simulate the sending operation of health data; to test the call procedure of the emergency point it is necessary to send a health data which is grave (i.e. one of the threshold is violated). After sending the data, the main phone running TrackMe application should make a call to a number where the suffix of the number is 118 if in Italy (due to testing reason, it was not possible to truly call the emergency point since this is a prototype application). Since the bluetooth client application is just a stub, it is not so user friendly, therefore, it can be seen that the connection is active only when the send data button is active, otherwise if it is not, check if the devices are paired and the bluetooth server is activated or just retry to connect.

References

[1] Android emergency location service, URL https://crisisresponse.google/emergencylocationservice/how-it-works/

[2] Working with certificates and SSL, URL https://docs.oracle.com/cd/E19830-01/819-4712/ablqw/index.html

[3] Spring framework, URL https://spring.io

[4] RabbitMQ, URL https://www.rabbitmq.com

[5] RabbitMQ concepts about exchanges and queues, URL https://www.rabbitmq.com/tutorials/amqp-concepts.html