Politecnico di Milano

AA 2018/2019



DD – Design Document

Version 1.0 - 11/12/18

*Authors: Professor:*

Emilio Imperiali Elisabetta Di Nitto

Giorgio Labate

Mattia Mancassola

1. **Introduction**

1.1 Purpose

The purpose of this document consists of giving more technical details than the RASD concerning the TrackMe application.

Indeed, if the RASD has as its objective to provide a more abstract view of the system with its functionalities, the Design Document goes deeper into detail about the implementation, providing an overall guidance to the architecture of the project. Here all the components forming part of the system are described, with the related run-time processes, the algorithms which are the basis of the application are explained and all the implementation choices are listed and motivated.

In particular, the following topics are touched by the document:

* The high level architecture;
* The main components and their interfaces;
* The runtime behavior;
* The design patterns;
* The algorithms’ design for the most crucial ones;
* Implementation plan;
* Integration plan;
* Testing plan.

1.2 Scope

<here goes more or less the same general purpose/scope which is in the RASD>

1.3 Definitions, acronyms, Abbreviations

1.3.1 Definitions

<TODO>

1.3.2 Acronyms

* API: Application Programming Interface
* DD: Design Document
* RASD: Requirements Analysis and Specifications Document

1.3.3 Abbreviations

<TODO>

1.4 Revision history

* Version 1.0:
  + First Release

1.5 Reference documents

<TODO>

1.6 Document Structure

**Chapter 1** is an introduction to the design document. Its goal is to explain the purpose of the document and to highlight the differences with the RASD, whilst showing the link between them.

**Chapter 2** aims to provide a description of the architecture design of the system. More precisely, this section is divided in the following parts:

* Overview
* Component view
* Deploying view
* Runtime view
* Component interfaces
* Chosen architectural styles and patterns
* Other design decisions

**Chapter 3** specifies the user interface design. Actually, this part is already contained in the RASD in the mockups’ section, so here will be added only additional behaviors.

**Chapter 4** provides the requirements traceability, namely how the requirements identified in the RASD are linked to the design elements defined in this document.

**Chapter 5** includes the description of the implementation plan, the integration plan and the testing plan, specifying how all these phases are thought to be executed.

**Chapter 6** shows the effort which each member of the group spent working on the project.

1. **Architectural Design**

2.1 Overview: High-level components and their interaction

The application to be developed is a distributed application and the three logic software layers of Presentation (P) that manages the user interaction with the system, Application (A) that handles the business logic of the application and its functionalities and Data access (D) that manages the information are thought to be divided on three different hardware layers (tiers) that represent a machine (or a group of machines) so that any logic layer has, in principle, its own dedicated hardware: we have a so called three-tier architecture. This architecture is thought to guarantee to the system characteristics of scalability and flexibility and to lighten the server side splitting it into two nodes. In particular, the second tier is thought to contain only the business logic to physically separate users and data to guarantee more safety in accessing to data since the system deals with sensitive data.

The following image show the high-level architecture of the system without providing any detail.



Figure 1 – High-level system architecture

For what concerns the third parties, they can communicate through their devices with the application level to make queries and to organize competitions with a synchronous message flow (the server implementing the business logic have to provide an answer). The server of the application level will interact with the customers also when sending notifications (SOS etc.) in an asynchronous way (they don’t wait for an answer). Instead, for what concerns the users, they interact with the application level providing answers to queries and their data (asynchronous messages) and to retrieve their data and stats, enroll themselves for competitions or watch them (synchronous messages). Finally, the server in the application layer communicates synchronously with the database server (data access layer) to retrieve or store information when needed. (TODO -> storing data is asynchronous?)

To augment the system scalability a scale-out approach is followed: performances improvement is obtained through nodes replication. This approach, that requires the adding of a load-balancing system to distribute the working load among the various nodes, allows to exploit the downsizing principle that affirms that low-end servers have minor costs of high-end servers for the same computational power. Moreover, cloning with shared disk configuration: clones share the memory disks (shared nothing configuration is not convenient since the offered service is very ‘write intensive’) is exploited both to distribute computational load and to replicate data for security reasons. So, this technique increase both scalability properties and faults tolerance of the service and is important in particular for the critical service offered through AutomatedSOS. Finally, to guarantee an appropriate packets control, firewalls are installed before and after the application tier to create a DeMilitarized Zone (DMZ) for the application servers so that the external network can access only to the resources exposed in the DMZ. The web servers don’t guarantee the same security level and this is another reason why, in general, they forward the requests to the application servers in the DMZ.

A more detailed, but still informal view of the system is provided in the following image.

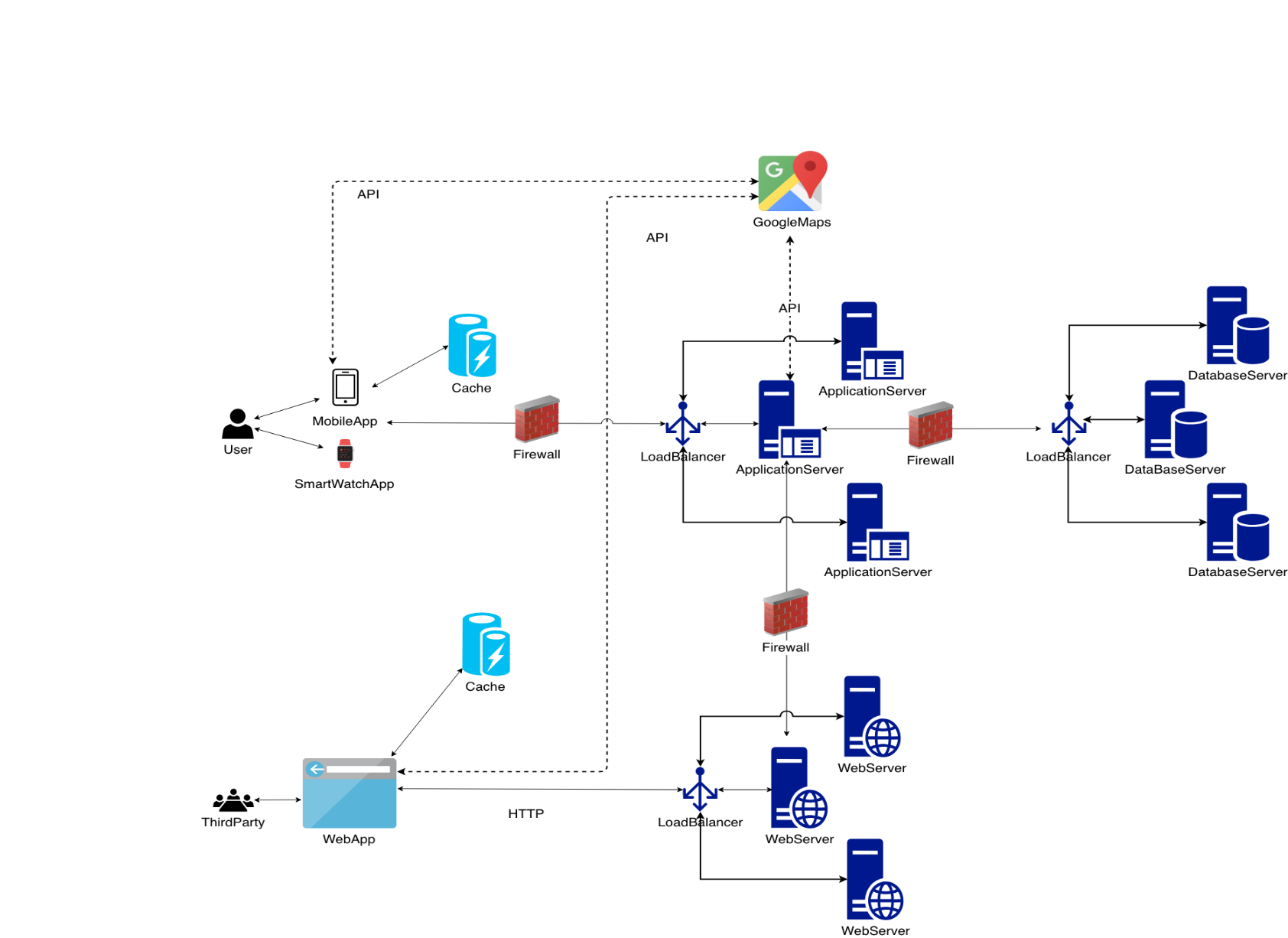


Figure 2 – System Architecture

In the Application tier we have an application server to provide the customers with access to the business logic which generates dynamic content (customable, extensible, interactive and seamless). Then we have a Web server to work with the HTTP standard in case of communication with the third parties’ web application, but to lighten its load the Web server is not supposed to generate dynamic content through plugins for scripting languages: it will forward the requests to the Application server if that is the case. To fasten and lighten the communication, caches are used in front of the Application tier: the cache needs to have an appropriate knowledge of data that make up the business objects data, business logic and changes that can transform them at UI level: this knowledge is required to invalidate data when needed (some alterations occurred). [Moreover the users’ mobile app stores part of the users’ data and stats (most recent ones and manually inserted ones) in their dispositive so that they do not even have to be connected to the Internet to consult their data and stats.] TODO 🡪 non è proprio questo che fanno le caches?

It is worth to note that caches for applications servers can’t be exploited because of node replications: it is not possible to know what have been requested from a certain application server and what from another one so it is impossible to provide a cache that would prevent them from accessing the database server every time.

Finally TrackMe exploits data mining techniques to exploit the big quantity of data available and extract from them relevant information and recurrent patterns that can be useful for the third parties that want, for example, find ‘hidden’ information and characteristics about their own clients or, more generally, about certain sets of TrackMe’s users. This can be obtained through various learning techniques: association rules, classification and clustering.

2.2 Component View

The following diagram contains all the components (logical or physical ones) of the system showing their interactions. The ports that represent the external interface exposed by components are shown only among different subsystem for the sake of simplicity.

In the diagram only the application server subsystem is analyzed in detail because it is the core component of the system: it contains the business logic. The other components of the presentation layer, of the data access layer and the web server are represented (through their software components only) just to represent their interactions with the application server.

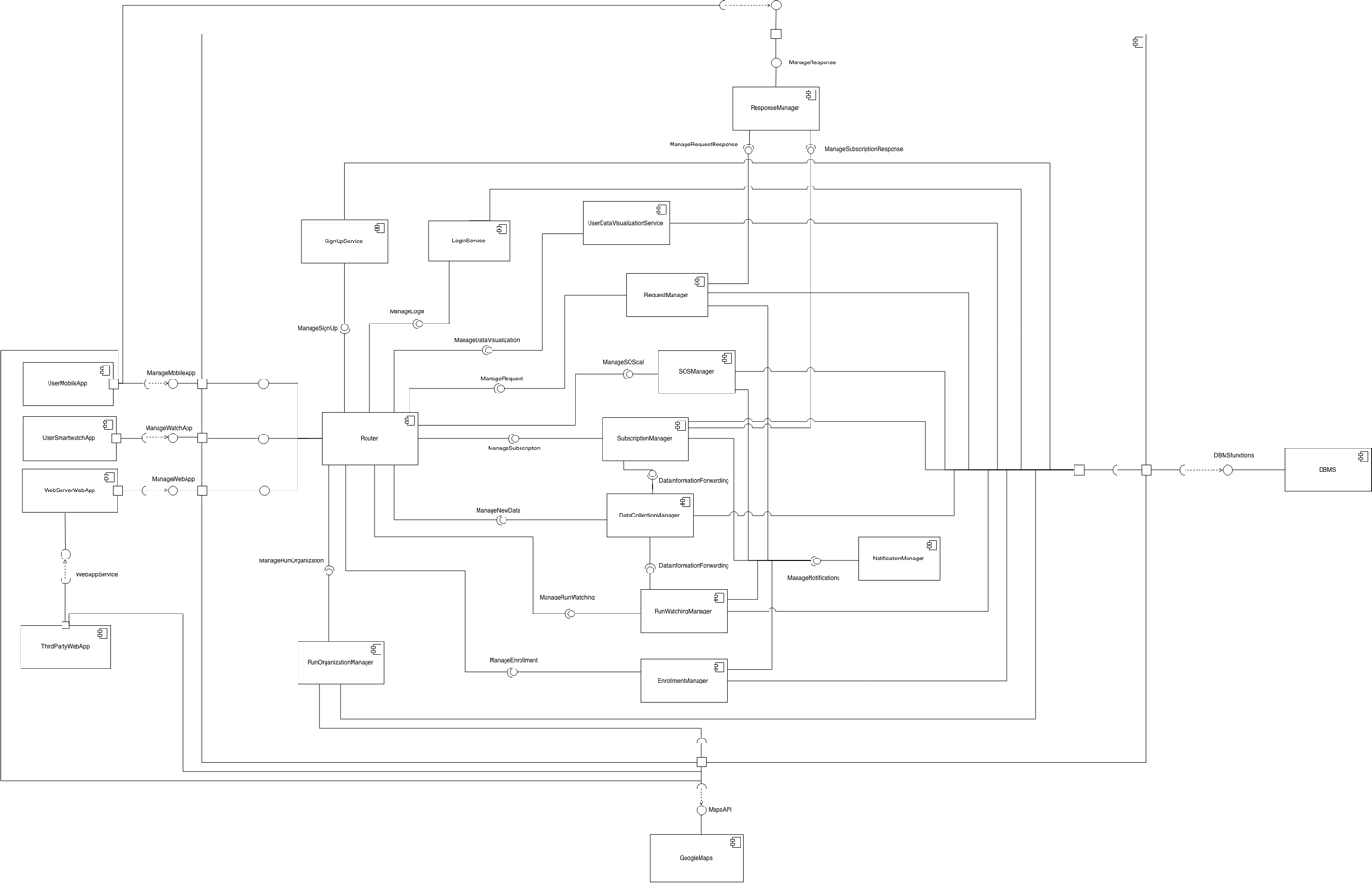


Figure 3 – Component Diagram

The components’ functions contained in the application server are described in the following:

* **Router**: it manages all the ‘messages’ and all the function call coming from the other subsystems to pass the data to the correct component and eventually call the right method/function on it. The router is partitioned according to the type of component it has to interact with because the functionalities offered are quite different among them. The **MobileAppRouter** handles the interaction with the users’ smartphone app: it allows them to sign up, to login, to consult their own data and stats and to watch or enroll for a run. This component also forward the received data (all the data among the required ones that a smartphone is able to collect) to the ‘DataCollectionManager’ in the case that the smartwatch is turned off. The **SmartwatchAppRouter** is indeed concerned with messages coming from the users’ smartwatches (by default data are collected only through them unless the device is turned off): it has to forward the received data to the ‘DataCollectionManager’ and to forward the SOS call in case of emergency to the ‘SOSManager’. Finally the **WebAppRouter** offers the interface for the interaction with third parties’ web application: it must forward to the appropriate components the login and sign up data, the individual and aggregate requests, the subscription requests and the data concerning the organization of a run. A more detailed view of the Router component to show this is provided below.

Figure 4 – Detailed view of the Router component



Figure 5 – Detailed view of the Router component’s interfaces

* **SignUpService**: this component contains all the procedures to allow the customers to register to TrackMe expressing also to which service they want to register for. It has to interact with DBMS to store data about the registration and performing controls about the chosen username and password.
* **LogInService**: it manages all the logic inherent to the authentication of the customers. It interacts with the DBMS to check that the authentication parameters match the stored ones.
* **UserDataVisualizationManager**: it comes into play when the user wants to access to its own data and stats, for example accessing at his own monthly stats about heartbeat. It contains the applicative logic to handle the requests and provide the correct answers and has to invoke the DBMS to retrieve from the database the requested information.
* **RequestManager:** this component deals with requests by third parties about single individuals or groups of individuals. It has to retrieve the requested data from the database (has to interact with the DBMS), make the privacy controls in case of aggregate requests, forward the queries to the requested users, handle their answers in case of individual requests and provide the right feedback to the requesting third parties.
* **SubscriptionManager:** it contains all the logic about subscriptions to users’ data for the third parties. It handles the requests for subscriptions for individual users or groups, the provided answers for the subscription and it is concerned in forwarding the data as soon as they are produced in case of a successful individual or group subscription (it provides an interface to the ‘DataCollectionManager’ to be able to do this).
* **SOSManager:** this component receives the emergency call from the user’s mobile app and is concerned in finding the third party that is nearer to the user to forward to it the SOS. The control to check if parameters are out of the defined bound is performed client side, this component is concerned only in handling the emergency call (this is why it offers its interface only to the ‘Router’ and not to the ‘DataCollectionManager’).
* **RunWatchingManager:** this component is concerned in providing to all the willing users the possibility to follow a desired run. It receives continuously information from the ‘DataCollectionManager’ about the new collected data and, analyzing them, it is able to perceive which of them must be retrieved from the database and to which users they have to be forwarded. It exploits the ‘NotificationManager’ to remind with one hour in advance that the competition for which the user has expressed the will to watch is beginning in short time.
* **EnrollmentManager:** it manages all the requests of enrollment to organized competitions. This component has only to verify the availability for the requested run and send the confirmation to the requesting user providing him with an identifier for the competition he has enrolled for. It exploits the ‘NotificationManager’ to remind with one day in advance about the competitions for which the user has enrolled.
* **RunOrganizationManager:** it deals with the organizations of a competition asked by some third party. This component has to verify that all the inserted data are correct, for example it has to control that the defined path is feasible, that it doesn’t cross an already defined path for another defined competition in the same time slot and this kind of things (that’s why it has to exploit Google Maps APIs).
* **DataCollectionManager:** this component receives all the data transmitted by users and has to forward the proper information about them (meta-information to the other components needing them that are the ‘SubscriptionManager’ and the ‘RunWatchingManager’ so that they can retrieve the eventually needed data from the database).
* **NotificationManager:** this component deals only with the logic for push notifications, it sends to the customers only asynchronous messages (doesn’t expect for any feedback by them). All the other components that have to forward some message to a client (for example the ‘RequestManager’) has to exploit the interface offered by this component. This component has been conceived to maintain single responsibility of the components: the type of messages sent are asynchronous (they are different from the general response messages provided by the server) and so it is more correct to use a different component rather than sending push notifications directly from the components where they were originated. When a notification is sent the dedicated client’s interface in ‘ResponseManager’ for the handling of the client’s response is provided (in our system this happens in case of a request of subscription or individual request). This mechanism of providing the interface of ‘ResponseManager’ only when ‘needed’ is exploited to show the interface only to authorized clients (i.e. clients that have to response to a request sent as an asynchronous notification).
* **ResponseManager:** this component is the handler of the client’s answer to a request of subscription or to an individual request and, in general, for all the responses that a client will eventually be able to provide if some new functionality will be introduced in future versions of the system.

It is important to notice that the only components that exploit the APIs offered by Google Maps are the clients’ ‘UserMobileApp’, ‘ThirdPartyWebApp’ and the ‘RunOrganizationManager’ on the server side: this is because all the visualization and interaction tools offered by Google Maps’ APIs have to be directly encoded on the clients’ app and on the application server the APIs are exploited only for the functionalities concerning the organization of a competition (ex: checking if two paths overlap in some point in the case of simultaneous competitions).

2.3 Deployment View

In the following image the TrackMe deployment diagram is represented: it shows the execution architecture of the system and represents the distribution (deployment) of software artifacts to deployment targets (nodes). Artifacts, in general, represent pieces of information that are used or are produced by a software development process and are deployed on nodes that can represent either hardware devices or software execution environments.

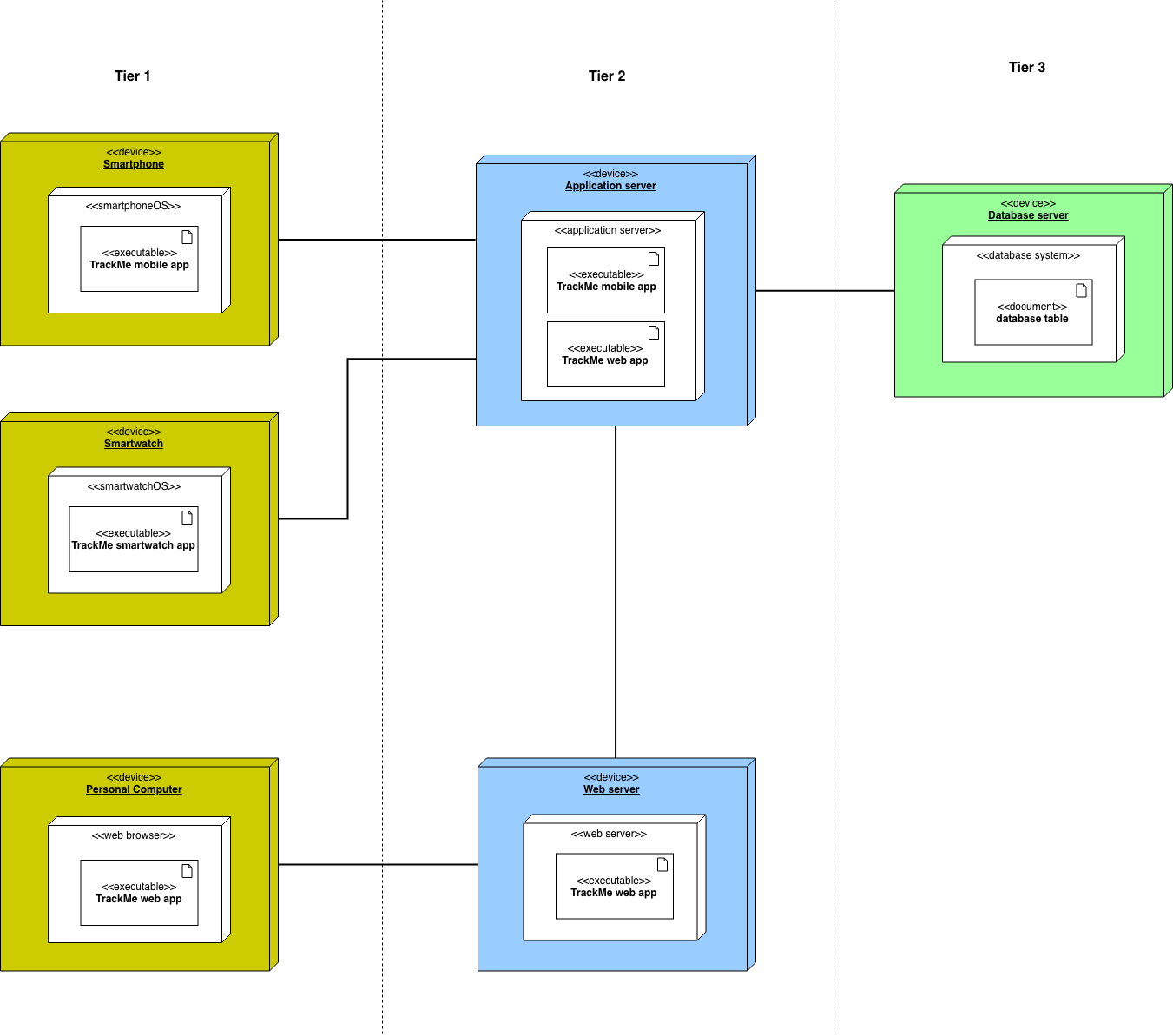


Figure 5 – Deployment Diagram

[TODO: explain why we decided to use mobile app and web app]

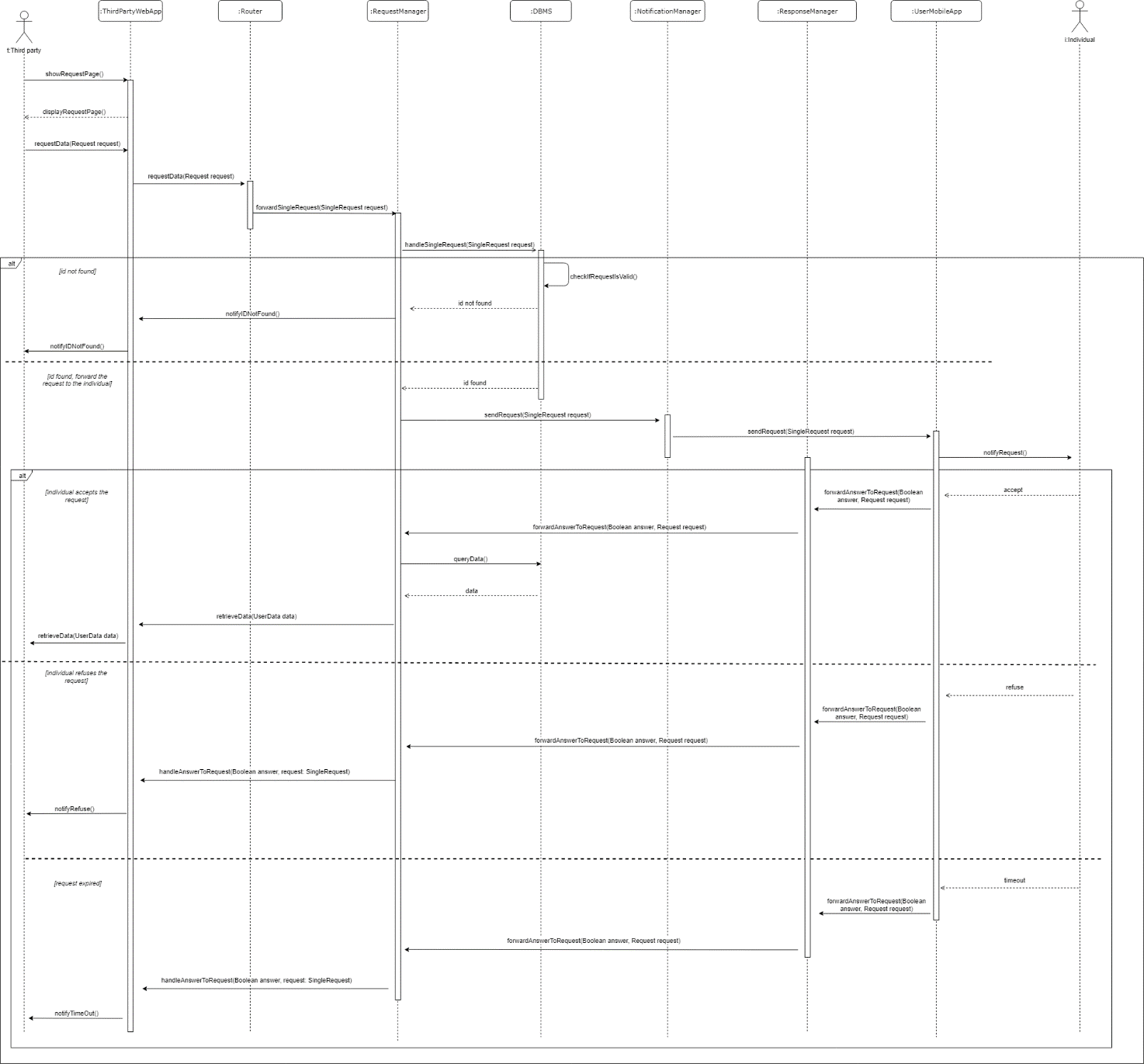
In the diagram ‘<<executable>>’ and ‘<<document>>’ are standard UML stereotypes that apply to artifacts: the former represents a program file that can be executed on a computer system while the latter represents a generic file that is not a «source» (another UML standard stereotype) file or «executable», in this case it represents the tables in the database.

In the diagram external systems as well as some other components (like load balancers and firewalls) are not represented to focus only on the components that host the core functionalities of the application or to components for which the deployment is effectively executed (Google Maps is already built and deployed). The three tier respectively contains:

* Tier 1: here the presentation logic must be deployed. Users must be provided with a mobile application on their smartphone and third parties with a web application accessible from their web browsers. Users communicate with the application server to retrieve their own data and stats, to report an emergency or to enroll or follow a competition. Third parties communicate with the web server to retrieve users’ data, receive emergency reports or organize run competitions.
* Tier 2: here is deployed the application logic. The application server implements all the business logic, handles all the requests and provide the appropriate answers for all the offered services. It is directly addressed by the mobile application and handles also some requests that are forwarded by the web server and sent by the web application in all the cases in which the web server can’t provide information either because it doesn’t have them in its local disk or because some dynamic content has been requested.
* Tier 3: here the data access must be deployed. The database server is conceived to execute a relational DBMS (RDBMS), the database is relational. This is because the structure of a relational database allows to link information from different tables through the use of foreign keys (or indexes), which are used to uniquely identify any atomic piece of data within that table and this is very important for applications that are heavy into data analysis like TrackMe. Moreover the application has to handle a lot of complicated querying, database transactions and routine analysis of data and for all these reasons a relation approach is what is needed. A non-relational database would be inappropriate because it would not be able to represent some structures (as the users’ account etc.): it can’t express rules and constraints and have no fixed structure.

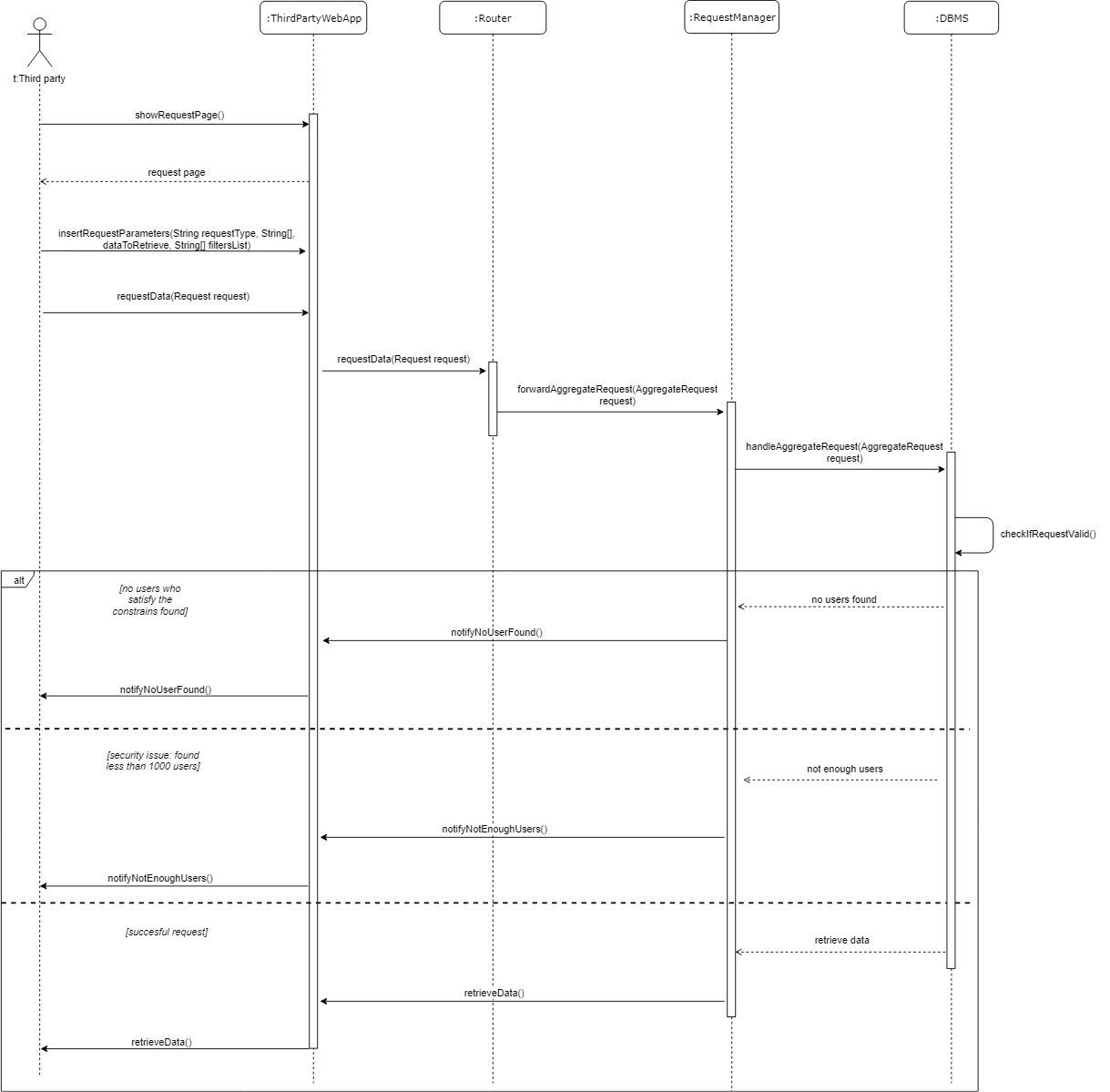
2.4 Runtime View

2.4.1 Individual request



In this sequence diagram the process through which a third party can request the data of an individual is shown. Once the web app has rendered the page for making the request, the third party can insert all the needed input data to perform the action (the data are here thought to be contained in the Request object). When submitted, the request is sent to the Router, which forwards it to the right component, i.e. the Request Manager. The latter is responsible for checking, communicating with the DBMS, if the request is valid: if the requested id does not exist in the database, an error message is sent back to the third party. Otherwise, if the check goes through, the Notification Manager is in charge of warning the individual. At this point, the user can decide whether to accept the request or not. There is a specific component in charge of receiving the user’s response, the Response Manager, which talks to the Request Manager. The Request Manager, then, according to the user’s decision, either queries the database for retrieving data or forwards to the third party a message which contains the user’s refusal. There is also a third case, which plays its part when the user doesn’t answer to the request within a time limit of 24 hours.

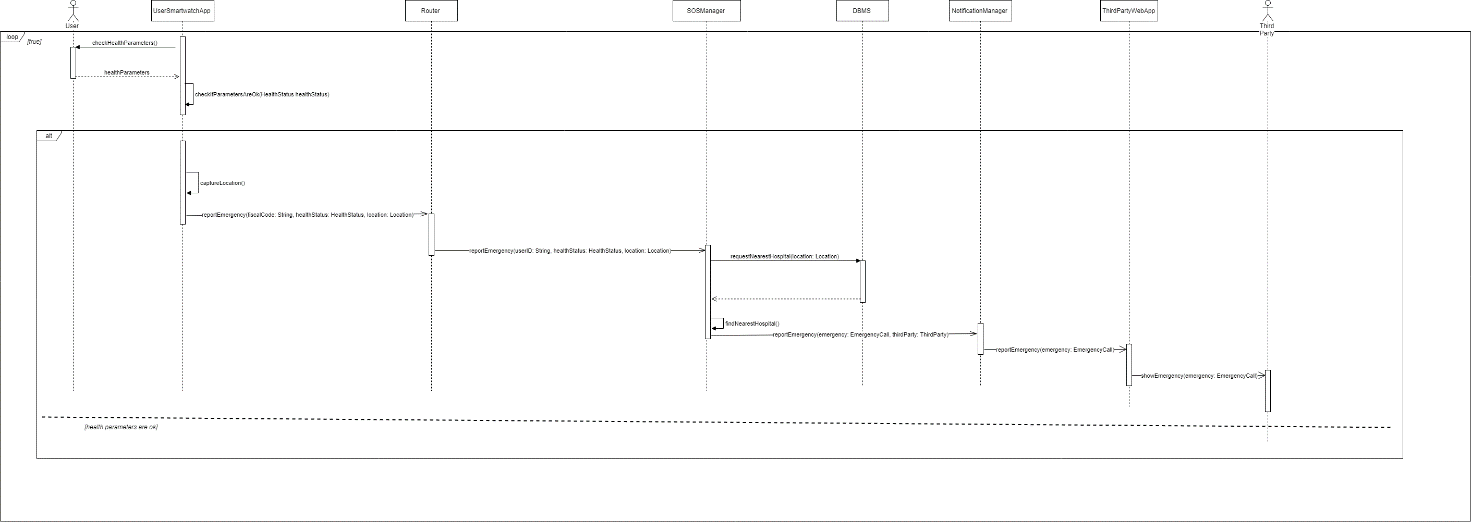
2.4.2 Aggregate request



In this sequence diagram the process through which a third party can request the data of a group of users, on the basis of some criteria, is shown. At first the flow is similar to the individual request one, the third party asks for the request page and when this is rendered, it can insert the parameters. Of course here the difference is that there are no identifiers, but only a bunch of filters, which the third party can choose to select a specific population. Once the needed data are inserted, the request is sent to the Router and then forwarded to the Request Manager, which talks to the DBMS to check its validity. Here three scenarios can occur:

* There is no user who satisfies the selected criteria in the database; in this case the third party is notified with a dedicated message.
* There aren’t enough users who satisfy the selected criteria in order to guarantee their privacy, i.e. the query’s result contains less than 1000 rows.
* The query goes fine and the data are retrieved to the third party.

2.4.3 Report an emergency



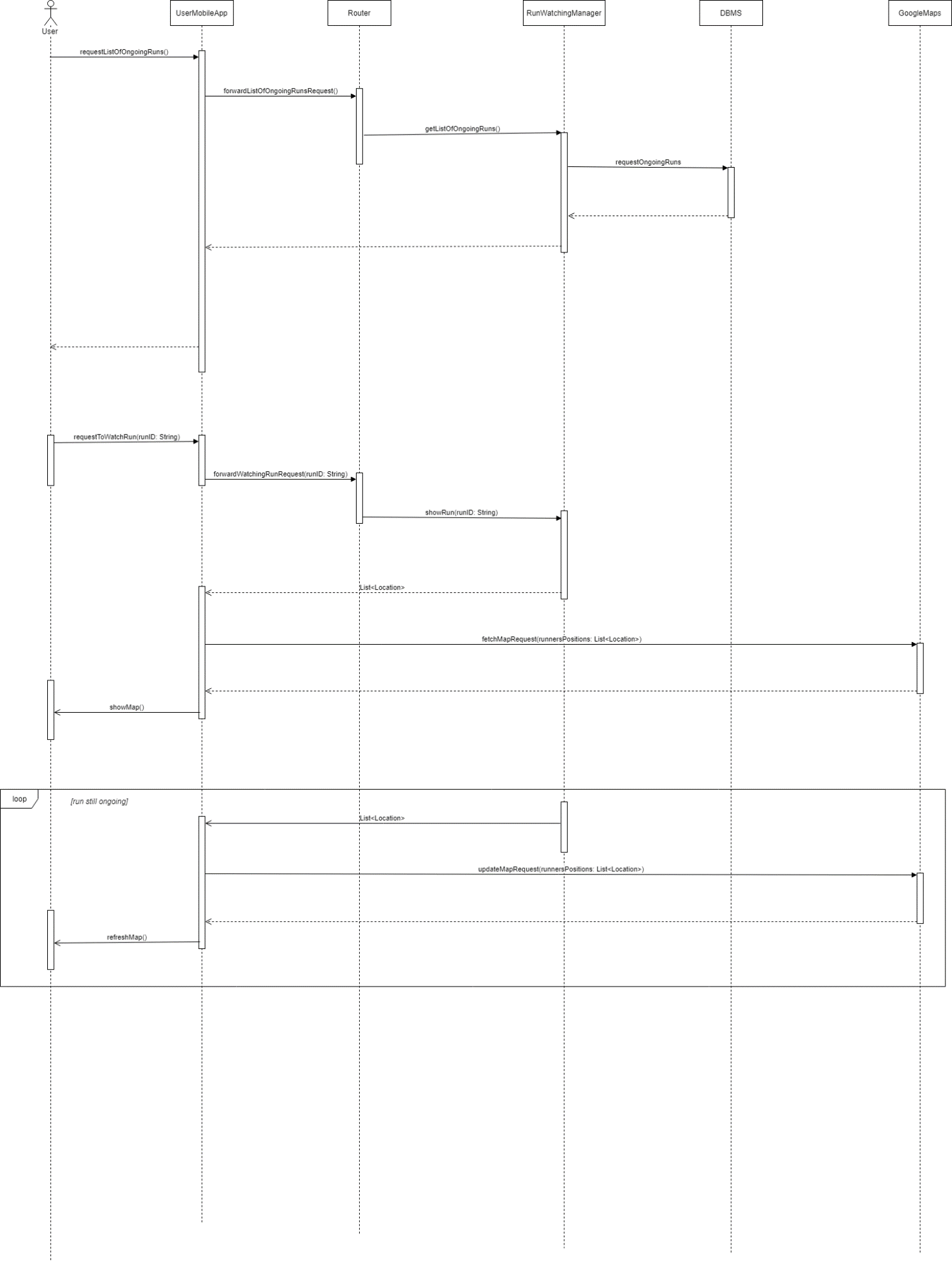
In this sequence diagram the process through which health parameters of the users are checked and, if below some threshold, an emergency is reported, is shown. At first, the app checks the health parameters and this operation is done in loop, until it finds out that those are below the thresholds. In that case, the app gets the location of the user (through GPS) and then reports to Router the fiscal code (that is an identifier), the health status (that contains all the health parameters of the user) and the location. The Router forwards that data to SOSManager, that queries the database in order to get all the third parties that provide emergency assistance services. In particular, SOSManager needs the location of those third parties, because it must find out which one is the nearest to the user. After that, the EmergencyCall, that contains data about the user, his health parameters and his location, is reported to the NotificationManager with the third party that must be contacted. The NotificationManager then sends a push notification to the right third party.

2.4.4 Organization of a run



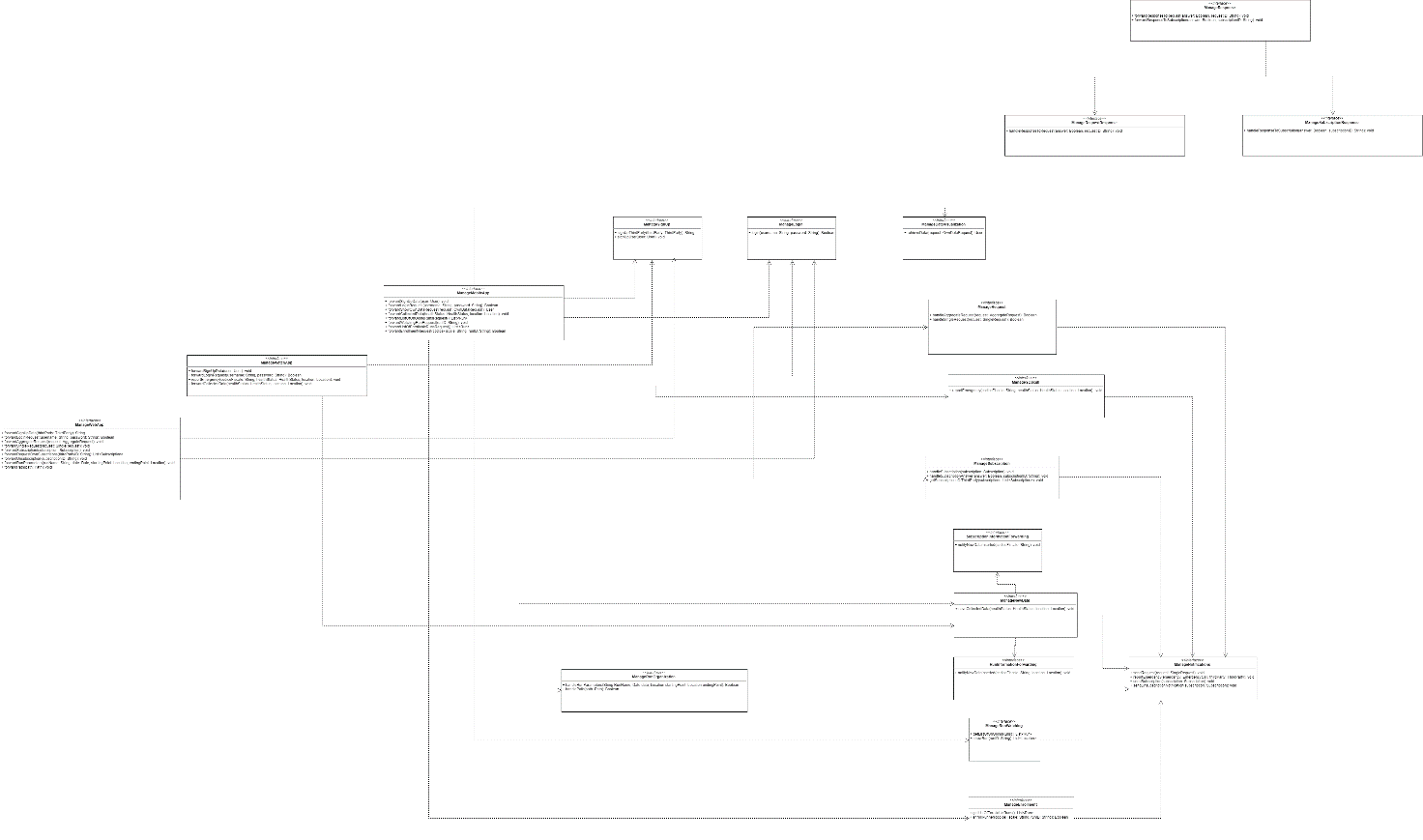
In this sequence diagram the process through a third party organizes a run and it’s registered in the system is shown. At first, the ThirdPartyWebApp forwards the run parameters inserted by the third party to the Router. Then, the Router forwards them to RunOrganizationManager, that queries the database in order to check if the run to be prepared is compatible with the other ones (for example there must not be runs with the same name). If the requested run isn’t ok, ThirdPartyWebApp is notified and the third party is requested to rewrite the run parameters, so the loop restarts. Otherwise, the ThirdPartyWebApp is notified that the run is ok. At this point, the third party will have to provide the path. To do so, the ThirdPartyWebApp uses the GoogleMaps API to fetch a map, providing starting point and ending point of the run, so that the map is centered according to where the run will take place. This map is showed to the third party, that selects the path of the run. All this is then sent to the Router, which forwards it to RunOrganizationManager in order to check if it’s ok (by means of not overlapping paths). RunOrganizationManager queries the database to get the paths of the other runs, and uses the GoogleMaps API to check if there are overlap with the requested path. Similarly to the check of the run, if the path is not ok then the third party will be asked to provide a new path and the loop will restart (in this case GoogleMaps API will just refresh the map, not fetch new one). Otherwise, the Run is created and linked to its Path, saving it in the database, and the third party is notified.

2.4.5 Watching a run



In this sequence diagram the process through a user watches an ongoing run is shown. At first, the user requests to provide the list of ongoing runs through the UserMobileApp, and this request is forwarded to Router. The Router then forwards it to the RunWatchingManager, that queries the database and gets the ongoing runs. It also sends them to the user. The latter chooses which is the run he wants to watch and his choice is forwarded to the Router and then to the RunWatchingManager. RunWatchingManager just provides a list of current location of each runner participating in the run. UserMobileApp then uses GoogleMaps API to fetch the map, using those locations in order to let the user see where the runners are. This process, that starts with RunWatchingManager sending to UserMobleApp the list of refreshed locations, is looped until the run is finished. While looping, the GoogleMaps API will not be used to fetch a new map, but just to refresh the positions of the runners on the already fetched map.

2.5 Component interfaces



Above the component interfaces belonging to the application server, with reference to what was shown in the Component diagram, are represented.

There are some things that are thought to be pointed out, in order to give a better understanding of how those component interfaces are intended:

* Only the interfaces offered by the application server are chosen to be represented.
* Both classes Request and Subscription contain a unique ID, that is also the primary key that identifies them in the database. Those IDs are assigned by the server when they are saved into the database. However, as it can be seen, the interface ManageWebApp offers for example the method forwardSubscription(subscription: Subscription). An instance of Subscription needs to be sent, but it can’t be an ID, since it’s assigned by the SubscriptionManager when it’s saving it in the database. Therefore it’s assumed that in this case the ID is set to null. The same considerations hold for all methods offered by all the interfaces that take as parameter a Subscription’s instance or a Request’s instance, and that are called before the ID was assigned.
* When a customer signs up, the application server saves his data in the database and assigns it a unique ID. The methods offered by the interfaces ManageWebApp and ManageSignUp that are called when a third party signs up (forwardSignUpData(thirdParty: ThirdParty) and signUpThirdParty(thirdParty: ThirdParty) respectively) return a String because they return the uniqueID assigned. This is then saved in the device of the third party, so that it can be used to identify him when other methods are called in the future. As said, regarding the ID of Subscription and Request, when forwardSignUpData is called in the context of ManageWebApp interface, the ID of ThirdParty’s instance passed is set to null, because its ID does not exist yet. The sign up of the user works in a slightly different way: when a user signs up, he must provide his fiscal code, which will be used as his unique ID and as primary key in the database. That is why it’s not needed to return a String containing the assigned ID when a sign up is done.
* The interfaces SubscriptionInformationForwarding and RunInformationForwarding have a similar purpose and it’s important to describe them in detail. Every time new data are collected by the system and arrive to the DataCollectionManager component, the latter saves them in the database. Then the SubscriptionManager component needs to be notified, because it could happen that one or more third parties are subscripted to that data, so they must receive them. That’s why SubscriptionManager offers the SubscriptionInformationForwarding interface to DataCollectionManager: the notifyNewDataInserted method is the one called in this case. The fiscal code is passed, because it’s sufficient for the SubscriptionManager to do the right query to the database and to get the new data to be passed to be forward to third parties (in case there is data to be forwarded). This mechanism can be considered an application of the Observer/Observable pattern.

As said before, RunInformationForwarding has quite the same purpose. In the context of Track4Run service, when a runner is participating in a competition, he sends to the system his location in real time, so that other users can watch the run. When a new location of a runner arrives to DataCollectionManager component, RunWatching needs to be notified, and that’s why RunInformationForwarding is needed. In this case, the location is passed along with the fiscal code, because there is no reason to save all the locations that a runner sends in the database. So RunWatching will not get them by querying the database.

* In general, methods written in the Component Interfaces diagrams are not to be intended exactly as the methods that the developers will write, but they are a logical representation of what component interfaces offer. This consideration is important in particular with regard to the methods offered by the interfaces provided by Router and by ManageResponse. In fact, since RESTful architecture is used, it must be cleared that they are not method called by means of remote invocation, instead they will be implemented following the RESTful principles.

2.6 Selected architectural styles and patterns

1. **RESTful architecture**

To develop the application we decided to use a RESTful architecture, with the goal to reduce the coupling among client and server components as much as possible in mind and also because the centralization of data plays an important role. Moreover, it fits very well for the scope since we are dealing with an application with many clients, on which we don’t have control, while on the contrary we have it on the server and we may want to be able to update it regularly, without touching the client software.

Using a RESTful architecture inevitably leads to an adoption of the following constraints:

* Uniform interface
* Client-server
* Stateless
* Cacheable
* Layered system

Concerning the client-server architecture, the third party will use the appropriate web app, which communicates with the web server, which in turn communicates with the application server. The user, instead, will use the mobile application to access directly the application server.

Finally, the application server acts like a client, querying the database server.

1. **Three Tier Client-Server**

We choose a multitier architecture because it allows to decouple the complexity of the system, making it more flexible and reusable. Indeed, the developers acquire the power of modifying or adding specific layers without disrupting the entire application.

More precisely, we adopted a three tier architecture, composed of a presentation tier, a domain logic tier and a data storage tier.

1. **Model View Controller (MVC)**

We decided to use this pattern in order to guarantee the reusability of code and also to promote a parallel development as much as possible.

MVC, indeed, with the separation of concerns allows flexibility and opens up the doors to other design approaches, which without it would be difficult to use.

In our case, the clients represent the front-end, i.e. the view, that interact with the controller, through which the information flows to and from the database, namely, the model.

**Other design decisions**

Thin client

We opted for a thin client to maintain the line of thought underlined till now. Indeed, this allow us to have an architecture in which the client is designed to communicate with a server, where the real logic is implemented. However, it is true that we don’t rely on external services to collect the data necessary to provide the application’s services (like Google Fit, Fitbit etc.), but instead we have our DataCollectionManager component which plays this role. For this reason, our client does not really respect the thin client paradigm: it could be considered a hybrid version, even if certainly more thin than thick.

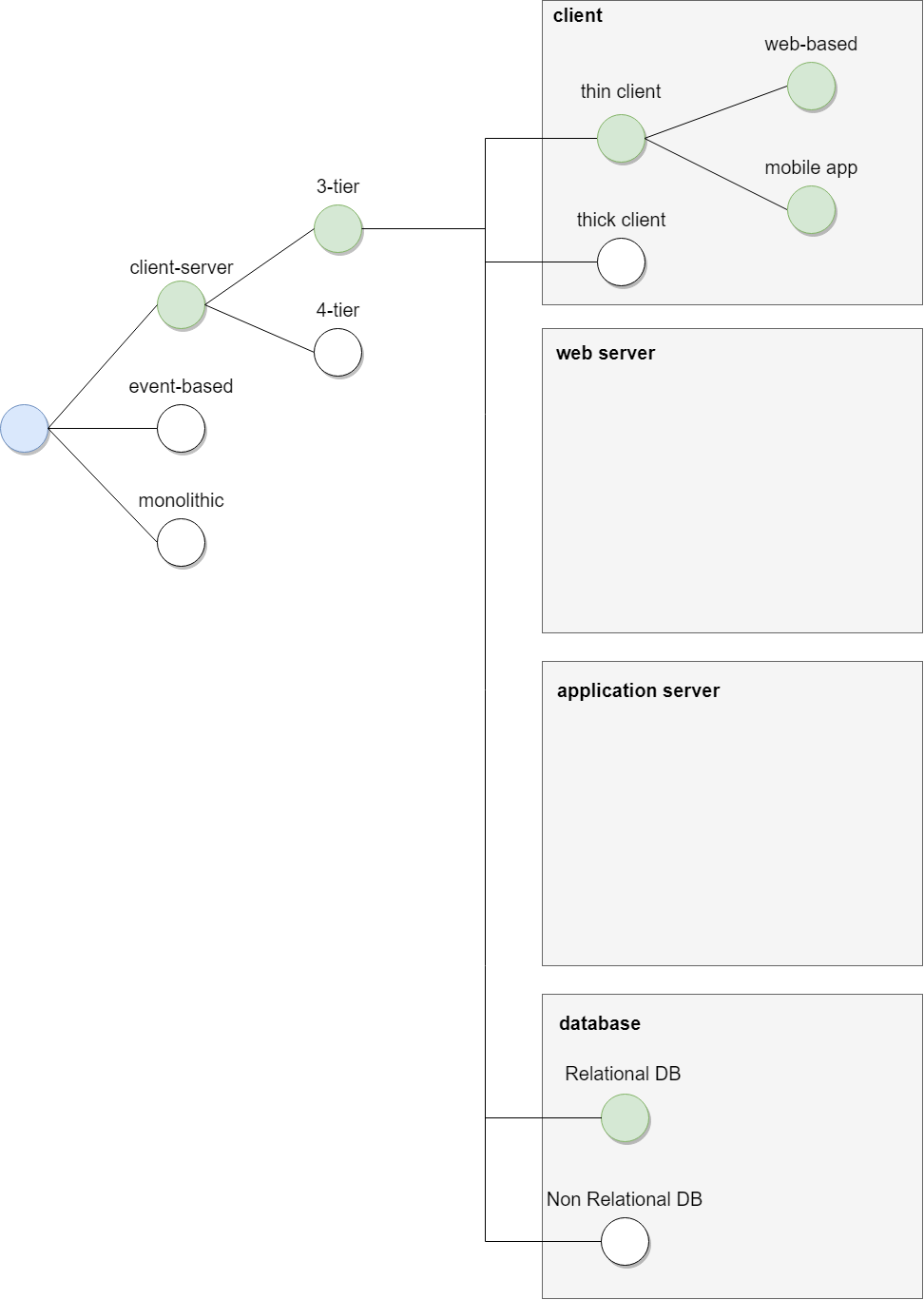
This choice has also its advantages. Thin clients are strictly dependent on a network connection, and in our case this is not an issue, since the application was conceived to operate most of the time online. Nevertheless, especially with mobile devices, it could happen that the network connection is not available, so it is extremely important to guarantee to the users the possibility to use at least some of the offered services offline.

It is worth mentioning that there is also another exception, which consists of using the Google Maps API internally to the client when the service which allows to watch a run and visualize the runners in real time on the map is in use. This has been decided to avoid to overload the server.

Relational database

Relational databases are a good choice when there is the need to deal with several transactions and when the data are linked by some relationships (users, third parties, runs etc.). This fits perfectly for our purposes, which can be catalogued as data analysis.

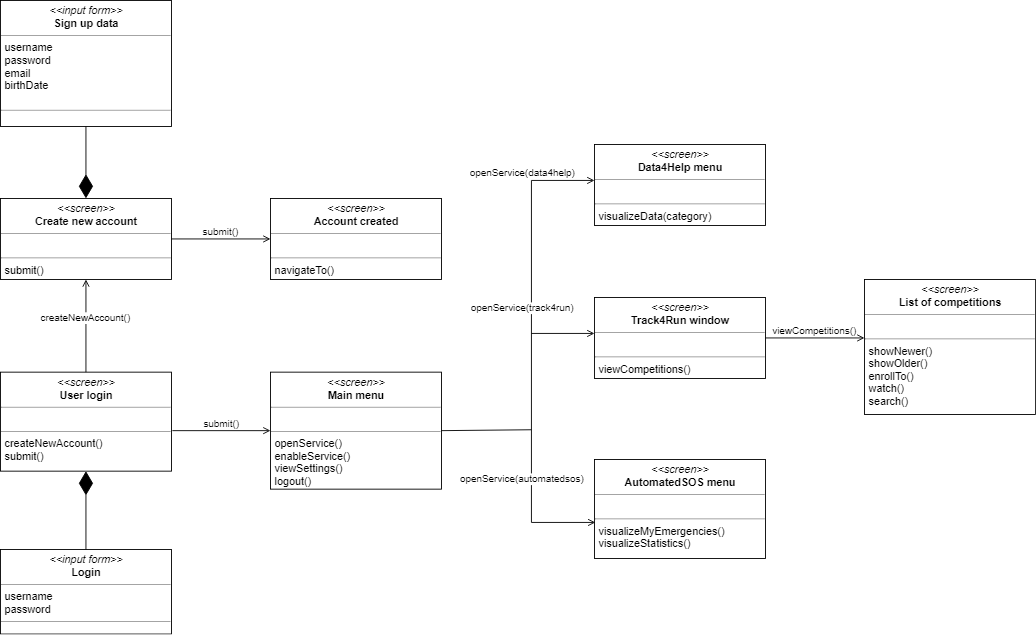
<Other more low level design patterns, like façade or observer>



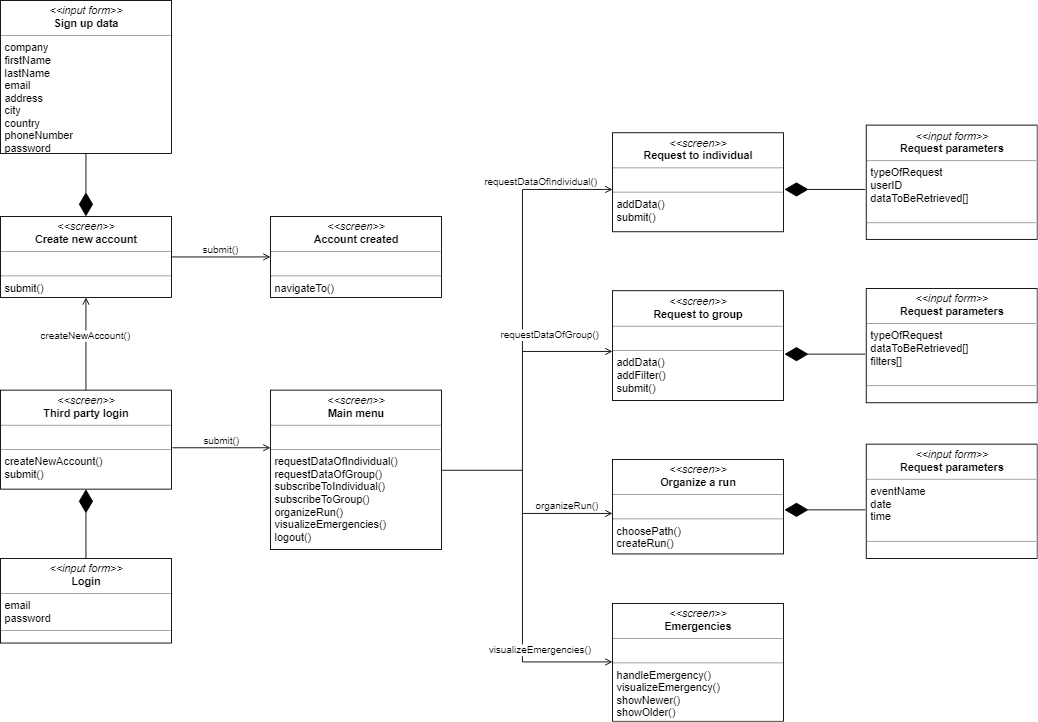
Decision Tree

1. **User interface design**

The mockups of the application were already exposed in the RASD document in the appropriate section. Here we present two UX-diagrams to show how the customer is supposed to navigate inside the application. We chose to expand only some of the possible interactions, to avoid to overcomplicate the diagram.



UX diagram- User

****

UX diagram- Third Party

1. **Requirements Traceability**

The whole design has been thought to guarantee that the system is able to enforce the requirements defined in the RASD (and, as a consequence, to achieve the prefixed goals). Here a mapping between those requirements and the design components in the application server that will ensure their fulfillment in a direct way (some other components that are indirectly needed to enforce some requirements are not directly mentioned in the list, but their role has been made clear through some comment) is shown:

* **R1**: The system must save the collected data of users registered to Data4Help in real time.
  + **DataCollectionManager:** this component manages to save the received data in the database in a correct way.
* **R2**: The system has to allow the third party that wants to retrieve some data to choose between an individual request or an aggregate one.
  + **RequestManager:** manages both individual and aggregate requests, it is able to distinguish the choice made by the third party on the ‘ThirdPartyWebApp’.
* **R3**: In case of a query for data of an individual the system has to ask to the third party the individual’s fiscal code.
  + **RequestManager:** contains the logic about individual requests, it retrieves the data of the right user thanks to his fiscal code. The fiscal code field is shown client-side on the ‘ThirdPartyWebApp’.
* **R4**: In case of a query for aggregate data the system has to ask to the third party which parameters to use to filter data.
  + **RequestManager:** contains the logic about aggregate requests, it retrieves the correct data exploiting the filters provided. The filters are inserted client-side on the ‘ThirdPartyWebApp’.
* **R5**: When a request for a specific individual’s data is made the system must allow the individual to accept the request or not.
  + **NotificationManager**: sends to the user a push notification with the information about the request that has been made and that is provided by the ‘RequestManager’.
  + **ResponseManager**: handles the user’s answer allowing him to accept or refuse the request.
* **R6**: When a request for data is approved the system has to make the previously saved data available to the third party.
  + **RequestManager**: in case of a positive answer by the system (for both individual and aggregate requests), sends the data to the requesting third party.
* **R7**: The system must provide to the third party the possibility to subscribe to new data and receive them as soon as they are produced both for individuals and for groups of individuals.
  + **SubscriptionManager**: handles the subscription requests, the response of the interested user (in case of individual subscription) and it sends the new data as soon as they are produced (that are forwarded by the ‘DataCollectionManager’) to the third party that has done a successful subscription.
* **R8**: The system must allow the user to accept or refuse a possible request for subscription by a third party.
  + **ResponseManager**: it handles the user’s answer allowing him to accept or refuse the request of subscription.
  + **NotificationManager**: it sends to the user a push notification with the information about the request of subscription that has been made (and forwarded by the ‘SubscriptionManager’).
* **R9**: The requests by third parties on aggregate data must provide them if and only if the number of individuals whose data satisfy the request is higher than 1000.
  + **RequestManager:** it contains the logic to make privacy controls on the requested data and provides them to the requesting third party if, and only if, the controls are passed.
* **R10**: When the number of users satisfying an aggregate request (for which a subscription was made) becomes less than 1000 the subscription is automatically canceled until the matching users become again more than or equal to 1000.
  + **SubscriptionManager:** it controls that the privacy controls are passed for subscriptions to data of groups of individuals and provides new data as soon as they are produced only in that case.
* **R11**: The system must allow the User to analyze its own data and stats providing him a way to access to all registered data and stats and giving him the possibility to consult both their aggregate (ex: daily average) values and precise measurements.
  + **UserDataVisualizationService:** provides the requested data and stats requested by the user.
* **R12**: The application on wearable devices, if AutomatedSOS is activated, must send the health parameters and location to the third party as soon as it detects that parameters are out of the defined bounds.
  + **SOSManager**: handles the SOS call and sends it to the ThirdParty providing the AutomatedSOS service that is closer to the user. The problem is detected client-side: it is the ‘UserSmartwatchApp’ that has to analyze user’s data and control if they are in the defined bounds or not.
  + **NotificationManager**: it is concerned in sending the SOS message to the third party chosen by the ‘SOSManager’.
* **R13**: The system has to allow third parties that have activated the Track4Run service to schedule a run providing name, starting and ending point coordinates, the path, the date and time of the competition and the maximum number of participants.
  + **RunOrganizationManager**: handles the interaction with the third party that wills to organize a run competition.
* **R14**: The system has to allow users who have activated the Track4Run service to enroll for an organized run showing them the organized competitions on a calendar.
  + **EnrollmentManager**: handles the users’ requests of enrollment.
* **R15**: The system has to allow users who have activated the Track4Run service to follow the development of a run selecting an ongoing competition from a list that identifies runs by their name.
  + **RunWatchingManager**: contains all the business logic to allow users willing to watch a chosen competition to follow it on a map through their mobile application.

It is worth noting that in the provided mapping the ‘Router’ components are not mentioned for the sake of simplicity, but they are directly or indirectly connected to the fulfillment of the system functionalities because they route to the right component every message coming from the client’s side (except for the ones handled by the ‘ResponseManager’): the interfaces exposed by the application server are the ones provided by the three ‘Router’ components.

1. **Implementation, integration and test plan**

The system is divided in various subsystems:

* UserMobileApp
* UserSmartwatchApp
* ThirdPartyWebApp
* WebServerWebApp
* ApplicationServer
* External systems: DBMS, GoogleMaps

These subsystems will be implemented, but also tested and integrated exploiting a bottom-up strategy: the components building the same subsystems will be implemented, integrated and tested for each subsystem and the integration and testing of the different subsystems will take place at a later stage. It is worth to notice that the external systems’ components need not to be implemented and tested just because they are external and they can be considered reliable.

A table that lists the main features available for a TrackMe customer, their importance and implementation difficulty is shown below to better understand the decisions about implementation, testing and integration that will be taken in the rest of this section.

|  |  |  |
| --- | --- | --- |
| Features | Importance for the customer | Difficulty of implementation |
| Sign up and login | Low | Low |
| Visualize personal data and stats | High | Medium |
| Request data of an individual or of a group of individuals | High | High |
| Subscription to data of an individual or of a group of individuals | Medium | High |
| Handling of emergency calls | High | Medium |
| Organization of run competitions | High | High |
| Enrollment to run competitions | High | Low |
| Watching a competition in live mode | Medium | Medium |

It’s important to stress that the Data4Help functionalities have to be implemented, tested and integrated as a first step: the other services (AutomatedSOS and Track4Run) rely on it, they are built on top of Data4Help.

For this reason and looking at the table that maps the features on their relevance for the customer and their overall complexity, the components have to be implemented and tested (unit tests have to be performed for each component) with the following order for what concerns the application server:

* **Request data of an individual or of a group of individuals**: for this feature it is necessary to implement and test the ‘DataCollectionManager’, the ‘RequestManager’ and the needed parts of the ‘ResponseManager’ and of the ‘NotificationManager’. To provide this functionality, the ‘RequestManager’ has to be integrated with the ‘NotificationManager’ in order to notify a customer about a request and also with the ‘ResponseManager’ to handle in a correct way the customer’s answer: the interaction with these two modules have to be tested.
* **Visualize personal data and stats**: this is another core function of the system and to achieve it the ‘UserDataVisualizationService’ has to be fully implemented and tested.
* **Subscription to data of an individual or of a group of individuals:** to provide this function the ‘SubscriptionManger’ must be implemented and tested and properly integrated with ‘DataCollectionManager’ to allow the third party to receive the data in case of a successful subscription and with ‘NotificationManager’ and ‘ResponseManager’ to forward messages to and from the clients-side. So the implementation of the components with which the ‘SubscriptionManager’ has to be integrated with must continue (the missing parts have to be added) to guarantee a correct behaviour.
* **Sign up and login:** the sign up and login features are obviously an entry condition for the right functioning of the system, but they are not core features and they are not very complex, so the ‘SignUpService’ and ‘LoginService’ components must be implemented and tested only at this point to complete the Data4Help functioning.
* **Handling of emergency calls**: this feature is more critical than the ones related to Track4Run, so the ‘SOSManager’ must be implemented and tested before them. A proper integration with the ‘NotificationManager’ is necessary to guarantee that the SOS message is sent to the third party. So, the code that has to be added to the ‘NotificationManager’ for this integration must be implemented and tested through the integration tests with ‘SOSManager’.

**TODO 🡪 specificare quando i test sono di unità o di integrazione**

* **Organization of run competitions**: this is the core functionality of Track4Run service, so the ‘RunOrganizationManager’ must be implemented and tested at this point.
* **Enrollment to run competitions**: to guarantee this feature the ‘EnrollmentManager’ must be implemented, tested and integrated with the ‘NotificationManager’ and the missing code parts have to be added to the ‘NotificationManager’ for this reason.
* **Watching a competition in live mode**: this feature is provided thanks to the ‘RunWatchingManager’, so this component must be implemented and tested at this point and a correct integration with the ‘NotificationManager’ must be performed and tested. So, again, the missing part of code for the integration has to be added to the ‘NotificationManager’.

Finally, we have to implement and test the integration of the ‘Router’ component with the rest of the components in the application server: this component has only to redirect the clients’ requests and messages and it has no very interesting business logic, although it is obviously a fundamental component for the right behavior of the system and that is why it is implemented and tested only at the end, to make the application server really operative.

It is important that the verification and validation phases start as soon as the development of the system begins in order to find errors as quickly as possible. As mentioned, the program testing to find bugs has to proceed in parallel with the implementation: unit testing has to be performed on the individual components [TODO🡪 scaffolding not needed because we implement the needed components partially, ex: ‘NotificationManager’] and, as soon as the first (partial) versions of two components that have to be integrated is implemented, the integration is performed and tested: we perform integration in an incremental way to facilitate bug tracking.

Since the TrackMe system is a relatively small system, the chosen integration technique is a structural one: bottom-up approach is followed, but taking into account that Data4Help has to be implemented and tested before the other services for the reasons explained before.

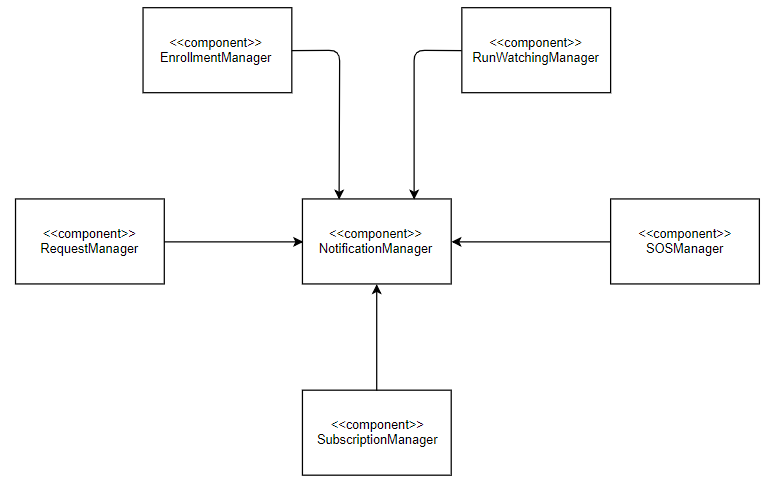
Once the system is completely integrated it must be tested as a whole to verify that functional and non-functional requirements hold, in particular, besides load and stress testing, performance testing has a crucial role: it helps identifying bottlenecks that affects response time and this is very critical for the AutomatedSOS’ requirements; it is concerned in identifying query optimization possibilities and this is also very important since the amount of data handled by the system is very large and the interaction with the database is heavy.

5.1 Component integration

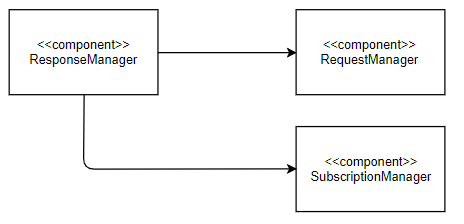
In the following diagrams we are going to show which components will go through the process of integration. The arrows start from the component which uses the other one.

**Integration of the internal components of the Application Server**

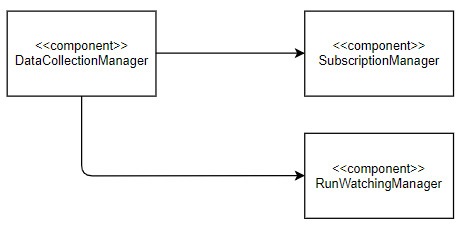
All the components are implemented and unit tested. Subsequently some components are integrated and the integration tested as well.



Notification Manager



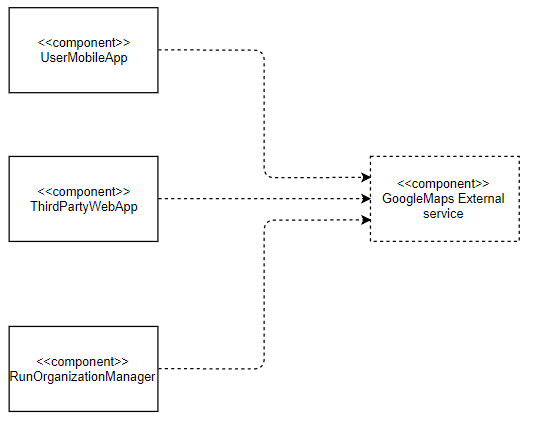
Response Manager



Data Collection Manager

**Integration with the external services**

The integration with the external services is done after all the needed components have been implemented and unit tested.

****

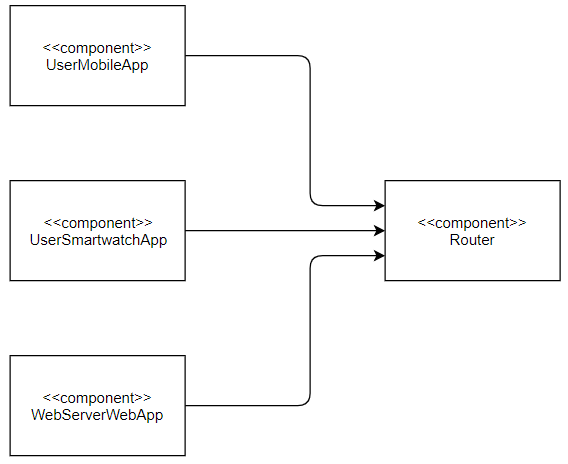
Google Maps API

**Integration of the frontend with the backend**

The integration (and testing) between the frontend and the backend happens only once all the components of the respective parts have been implemented and tested.

****

Third Party Web App



Router