Politecnico di Milano
AA 2018-2019

POLITECNICO
MILANO 1863

Computer Science and Engineering
Software engineering 2

Data4Help Design Document

Design Document
Version 1.0
9/12/18

Diego Piccinotti, Umberto Pietroni, Loris Rossi

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to further analyze the design and architectural choices for the system to be. Whereas the RASD presented a general view of the system and its features, this document will detail those concepts by showing components of the system, its run-time behavior, deployment plan and algorithm design.

The document will therefore contain a presentation of:

- Overview of the high-level architecture
- Main components and their interfaces provided one for another
- Runtime behavior
- Architectural design
- Implementation plan
- Integration plan
- Testing plan

This document is intended to be used by development teams as a guidance in the development process, by testing teams to write automated testing, and also for maintenance and extension to avoid structural degradation of the system through future modifications.

1.2 Scope

This document presents all the design decisions taken during the design process to satisfy the requirements stated in the RASD. It describes main components' architecture and interactions that have been selected to implement the AutomatedSOS, Data4Help and Track4Run services. To this end, it also identifies subsystems and their subcomponents.

1.3 Definitions, Acronyms, Abbreviation

1.3.1 Definitions

- **User**: individual who allows *Data4Help* to monitor his location and health status.
- **Third party**: individual or organization registered to *Data4Help* which can request users' data.

- **Data collection**: gathering of users' data through a wearable device.
- **Anonymized data**: data about more than 1000 users whose personal information has been previously removed so that they are not directly relatable to the system's users.
- **Elderly**: user who is subscribed to *AutomatedSOS* and is older than 60 years old.
- **Risk threshold**: Set of boundary health parameters defined for each elderly. If monitored values of the user's health parameters get below these boundaries, an SOS request is placed to an external ambulance provider.
- **Athlete**: user who participates in a run.
- **Run organizer**: third party who can manage runs for athletes.
- **Spectator**: non-registered individual who follows a run through a map with runners' positions.
- **Wearable**: a personal device provided with biometric sensors and GPS given to each user for free after the registration process.

1.3.2 Acronyms

- **API**: Application Programming Interface
- **DB**: Database
- **DBMS**: Database Management System
- **DD**: Design Document
- **GUI**: Graphical User Interface
- **RASD**: Requirements Analysis and Specifications Document
- **GPS**: Global Positioning Systemra

1.3.3 Abbreviations

- **Gn**: n^{th} goal.
- **Dn**: n^{th} domain assumption.
- **Rn**: n^{th} functional requirement.
- **Rn-NF**: n^{th} nonfunctional requirement.

1.4 Reference Documents

- IEEE Standard on Software Design Descriptions (*IEEE Std 1016-2009*)
- [*https://www.uml-diagrams.org/*](https://www.uml-diagrams.org/)

1.5 Document Structure

Chapter 1 introduces the purpose and scope of this document, the definitions and acronyms used, and the reference documents used while working on this project.

Chapter 2 presents the architectural choices made during the design process, along with an overview of the system and a more detailed component-by-component analysis.

Chapter 3 shows off some of the main GUIs of the web application.

Chapter 4 links all the requirements listed in the RASD with components elicitated during the design process.

Chapter 5 discusses the implementation and integration plan of the system to be, with specific focus on subsystems integration.

Chapter 6 shows the effort spent by each group member while working on this project.

2 Architectural Design

2.1 Overview

This system is based upon a three-tier architecture, consisting of three logical computing "layers". The first one is *Presentation* tier which is the front-end and consists of the web-app's user interface and client logic. The second layer is represented by the *Application* tier, which contains the functional business logic. The last layer, the *Data* tier, is responsible for the database storage system and data access.

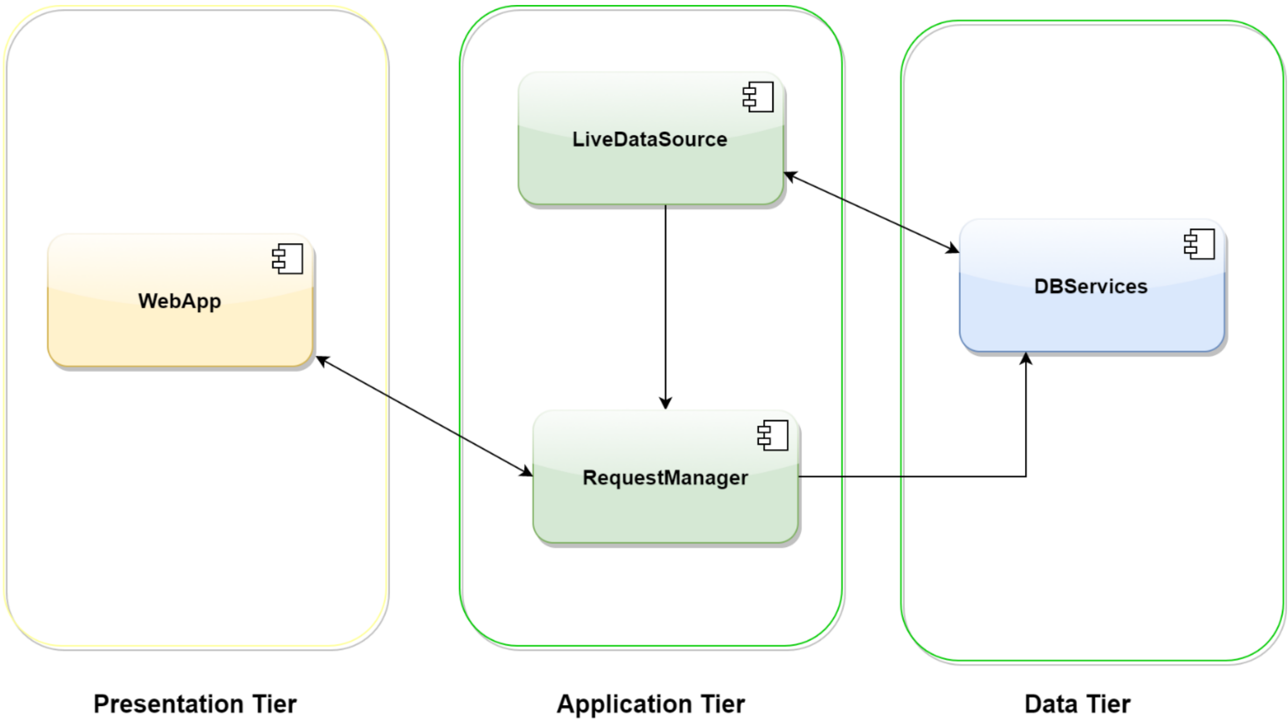
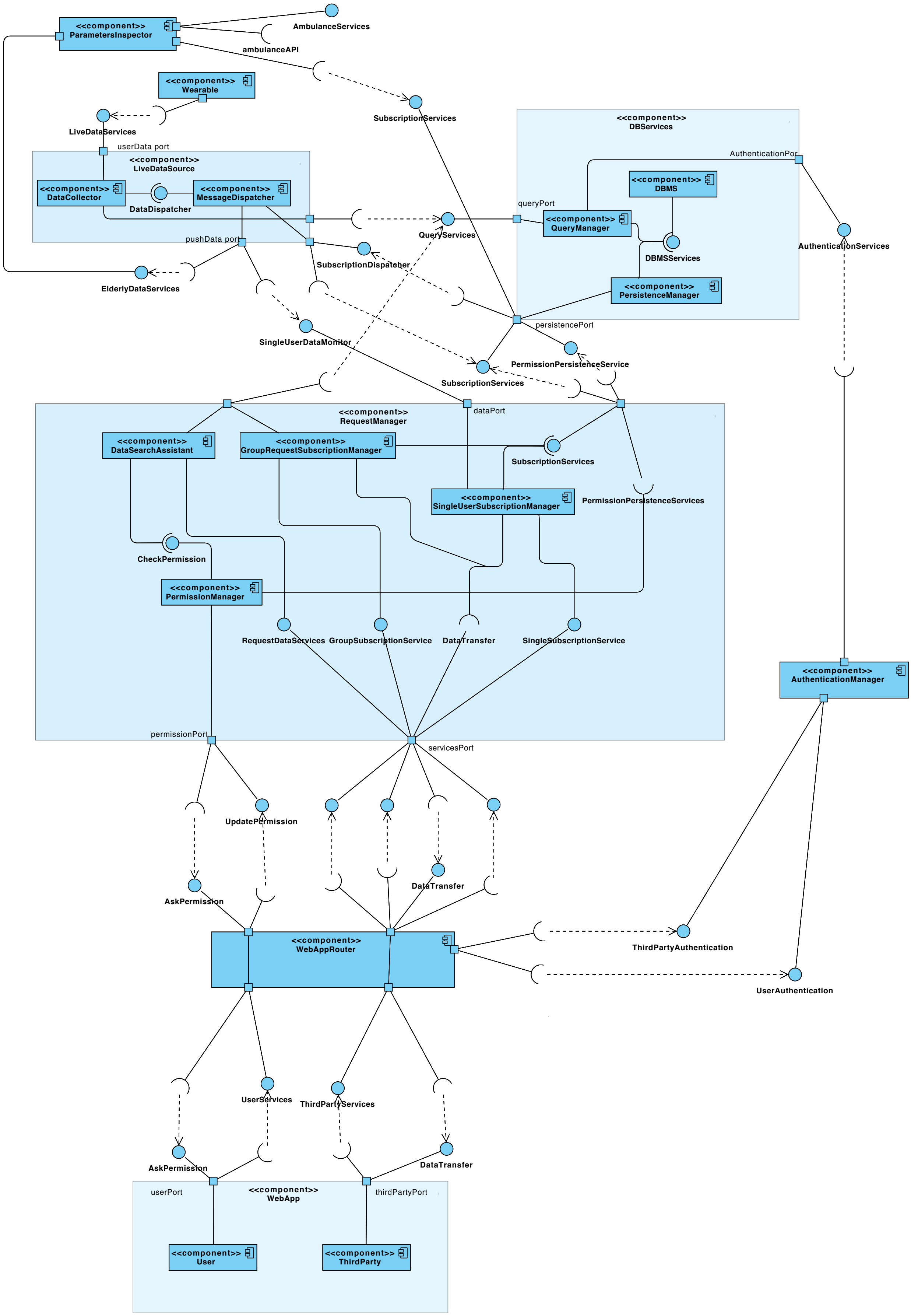
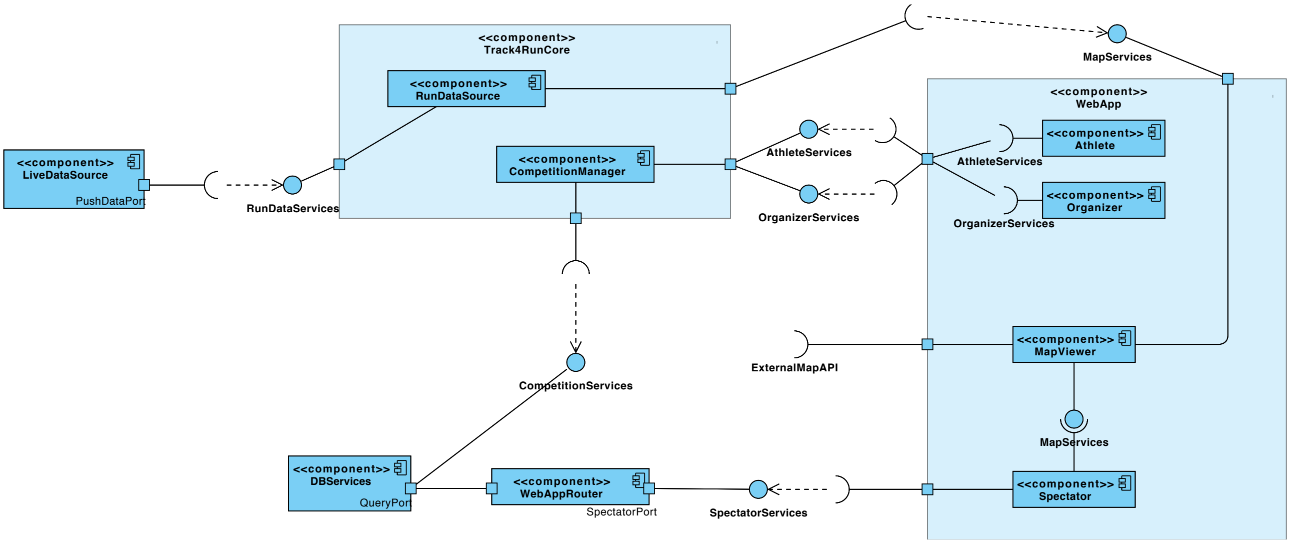


Figure 1: Data4Help High-level Architecture

WebApp is the high-level component related to the *Presentation* layer. Its task is to present the user interface and react to user input.
Data tier corresponds to the *DBServices* component, which stores data in the database and makes queries.
Application tier is composed by two main components: *LiveDataSource*, which is responsible for collecting users' data and then redistributing them immediately to third parties or submit them to the *DBServices*, and *RequestManager*, which contains all the logic related to third parties' requests and subscriptions.
Track4Run and *AutomatedSOS* services are built on top of this architecture, extending *Data4Help* core functionalities.

2.2 Component View





SubscriptionServices and MapViewer interfaces are replicated for readibility reasons

2.2.1 DataCollector

This component receives biometric and GPS data from wearable devices and holds responsibility of saving them to the database and simultaneously relaying them to the *MessageDispatcher* component.

2.2.2 MessageDispatcher

This component is in charge of relaying live data through the system, thus needing to be informed of message recipients. In order to do this:

- It is able to load data in case of cold restart or component failure through the *SubscriptionServices* interface.
- It is informed of every new subscription, be it an elderly subscribing to *AutomatedSOS* or a single user's data subscription from a Third Party, through the *SubscriptionDispatcher* interface.
- It is able to send (push paradigm) messages to *ParametersInspector* and *SingleSubscriptionManager* through their respective interfaces, *ElderlyDataServices* and *SingleUserDataMonitor*

2.2.3 PersistenceManager

The component is responsible of allowing persistence of the system's data, translating them to the query language offered by the DBMS.
In order to do this, the component exposes two interfaces:

- *PermissionPersistenceServices*, which allows to save new user permissions on the database.

- *SubscriptionServices*, which similarly allows to save new data subscriptions and to restore them from the database in case of failure or cold restart.

2.2.4 QueryManager

This component is responsible of allowing other components to make queries and retrieve data stored in the database.

2.2.5 Single and Group Subscription Manager

These components are event listeners that wait for requests for new data subscriptions. When they receive these requests, they use the *SubscriptionServices* interface to store the new information in the DB.

These components are also in charge of sending new data coming from (user or group) subscriptions to third parties.
In order to do this, *SingleUserSubscriptionManager* receives new single users' data from *MessageDispatcher*.
GroupRequestSubscriptionManager instead, maintains an internal list of group requests, each one associated with an update interval. Whenever one of the update timeouts expires, the component repeats the group data request to the DB and sends the result to Third Party, only if it can be properly anonymized.

2.2.6 DataSearchAssistant

This component is activated each time a third party executes a new search about an individual or group. The component holds two main responsibilities: checking if the third party has the permissions to access each individual data and checking if it is possible to anonymize group data. If all conditions are fulfilled, *DataSearchAssistant* will send back the desired data to *ThirdParty*.

2.2.7 PermissionManager

This component is responsible of checking if a third party has permission to access certain data, thus it needs to be informed of past granted permissions. In order to do this, it exploits *PersistenceManager* interface to load permissions in case of cold restart or component failure and also to store new permission granted by the user.

2.2.8 WebAppRouter

The *WebApp Router* is the API exposed by the application server to the client. It handles the client-server interface and relays calls to the methods of the external interface to the single components implementing them.

2.2.9 ThirdParty

This component is part of *WebApp* and it is responsible of offering a GUI to third parties that allows them to search for and subscribe to data and also to see the result of their search.

2.2.10 User

This component is part of *WebApp* and it is responsible of providing a GUI whereby users are able to manage data request permissions and also to subscribe to other services such as *AutomatedSOS* or *Track4Run.*

2.2.11 ParametersInspector

This component is responsible of monitoring elderly subscribers' health status and, if necessary, to request the dispatch of an ambulance. To accomplish the latter:

- It is able to contact the ambulance provider and send it data through the *AmbulanceAPI* interface, provided by the external provider's API.
- It is able to be informed on user being picked up by the ambulance through the *AmbulanceServices* interface, in order to stop sending data to the ambulance provider.

2.2.12 RunDataSource

RunDataSource acts as a relay between *MessageDispatcher* and *MapViewer*. Its main role is to pre-process data received from *MessageDispatcher*, and then send them to *MapViewer*.

2.2.13 CompetitionManager

This component is responsible of managing all the functionalities offered to *Track4Run* users. Specifically, it allows athletes to see the runs' list and subscribe to one of them. It also allows organizers to add and manage new runs.

2.2.14 Athlete, Organizer, Spectator

These components are part of *Track4Run*'s *WebApp* and offer GUI to athletes, organizers or spectators through which they can exploit all the services provided by *Track4Run*.

2.2.15 MapViewer

This component is in charge of showing live runners' position to spectators, thus needing to:

- Be informed of new athletes' position through *MapServices* interface.
- Use an external map provider API.
- Provide a map to each spectator for the desired run.

2.3 Deployment View

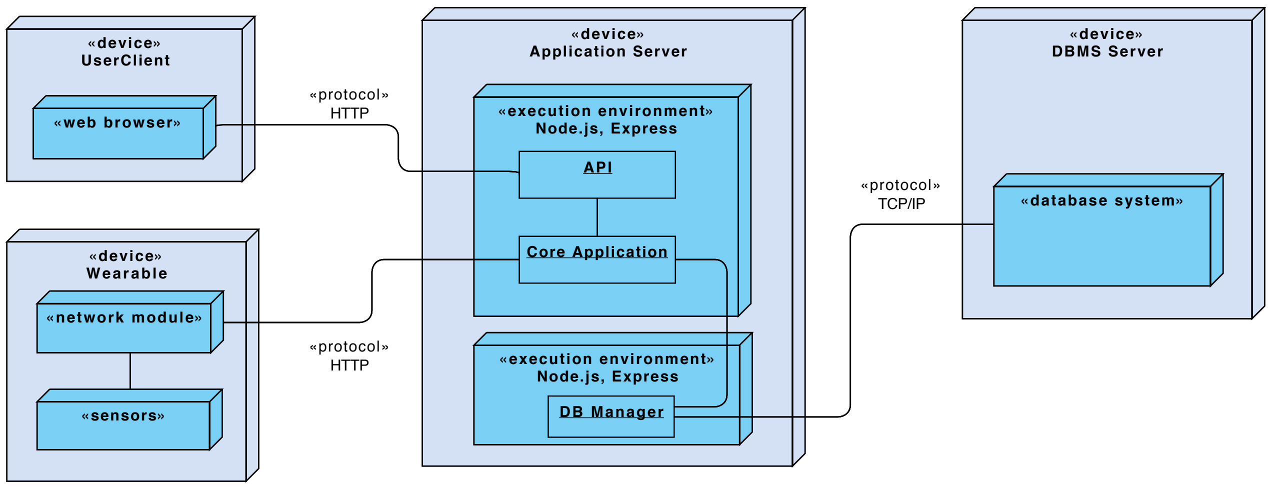
The deployment diagram shows the physical topology of the system.
We can see the three-tier architecture explicitly.

All users can benefit from *Data4Help* services by accessing its web application with their own devices. The web application sends requests to the Application Server API. The Application Server handles all the business logic and the communication with the DB and sends the requested data to the client.

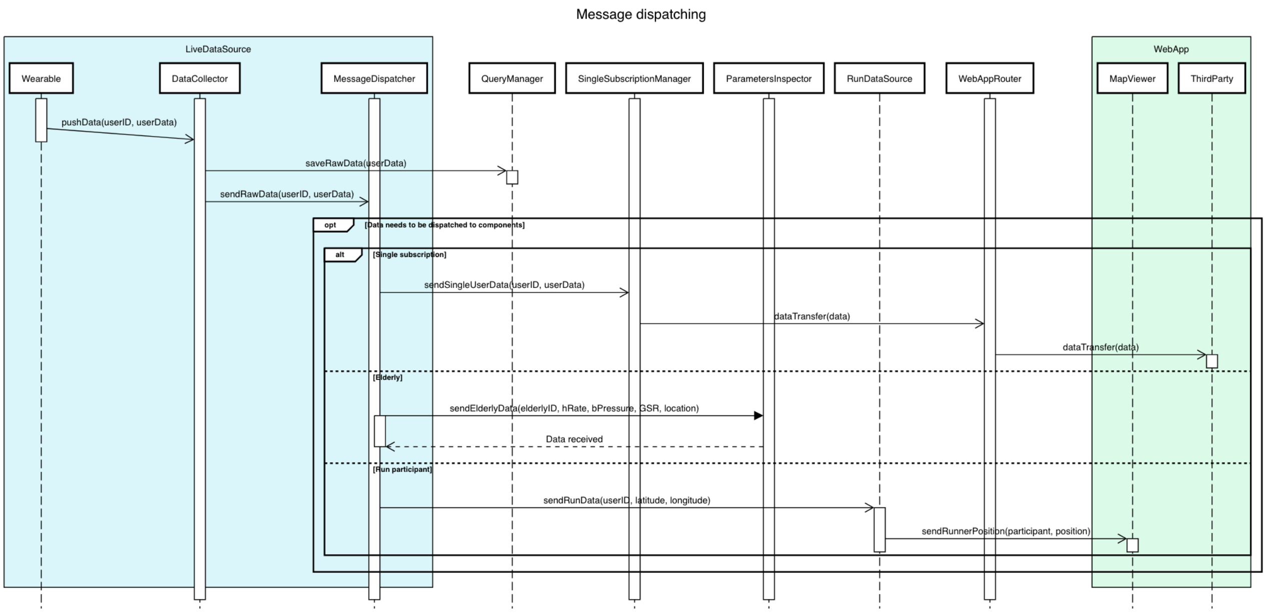
In the Application Server, there is a dedicated node to deploy the DB Manager. This way there is less coupling with the Application Server, and it is possible to maintain and update the two modules independently.

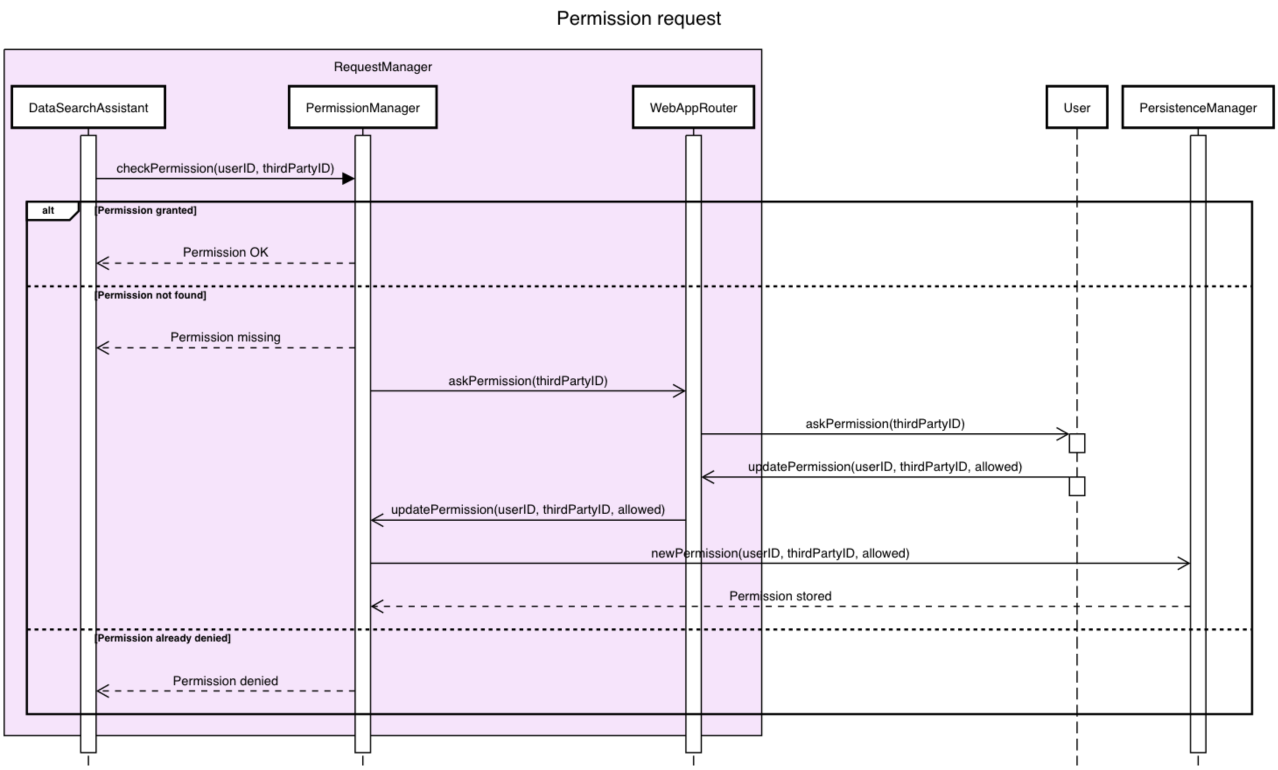
The DBMS Server is in charge to store all the data of the application. Deploying it in a dedicated server increases decoupling between system components.

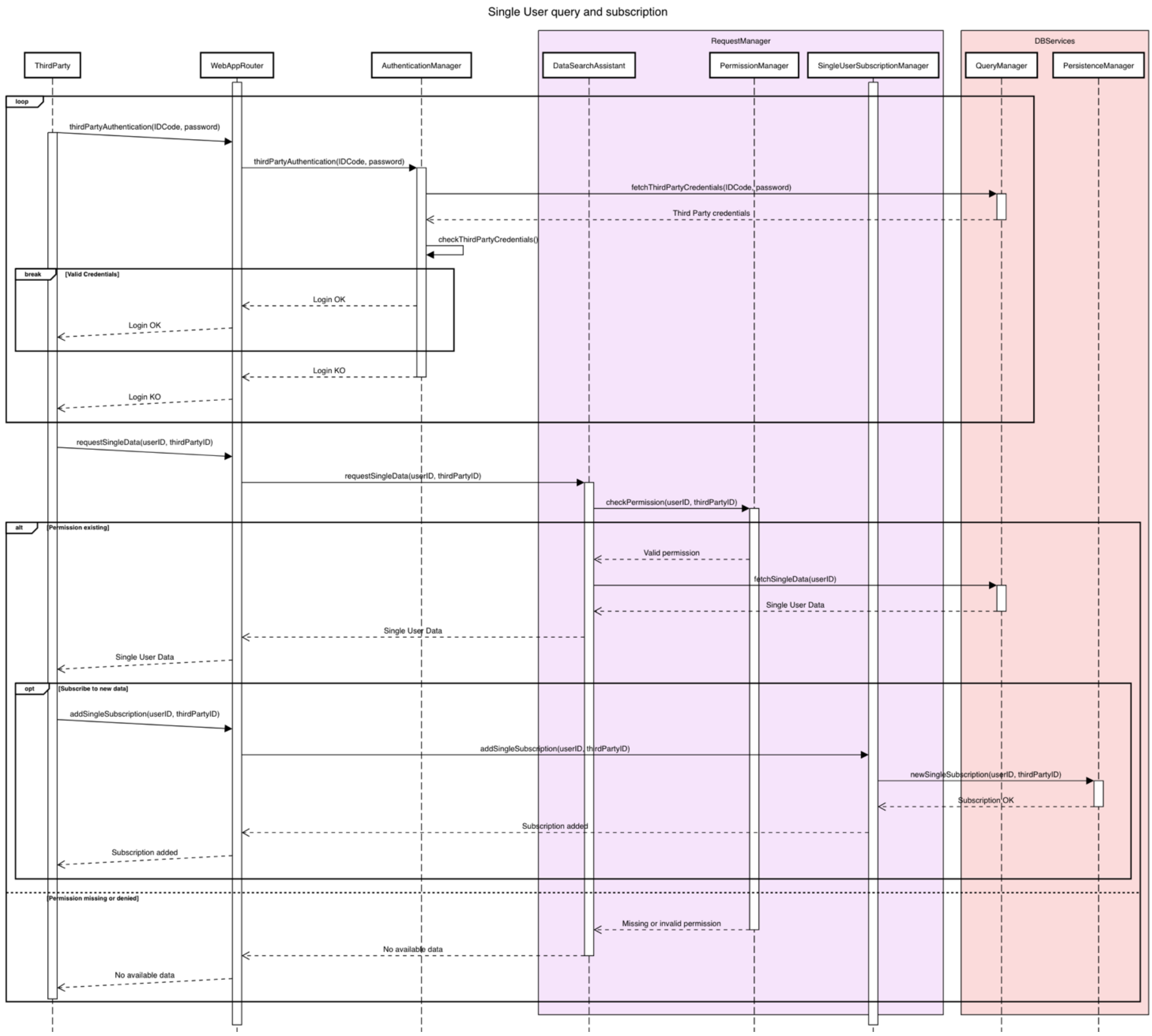
The wearable device runs simple code that reads data from sensors and sends them to the Application Server through the wearable's network module.

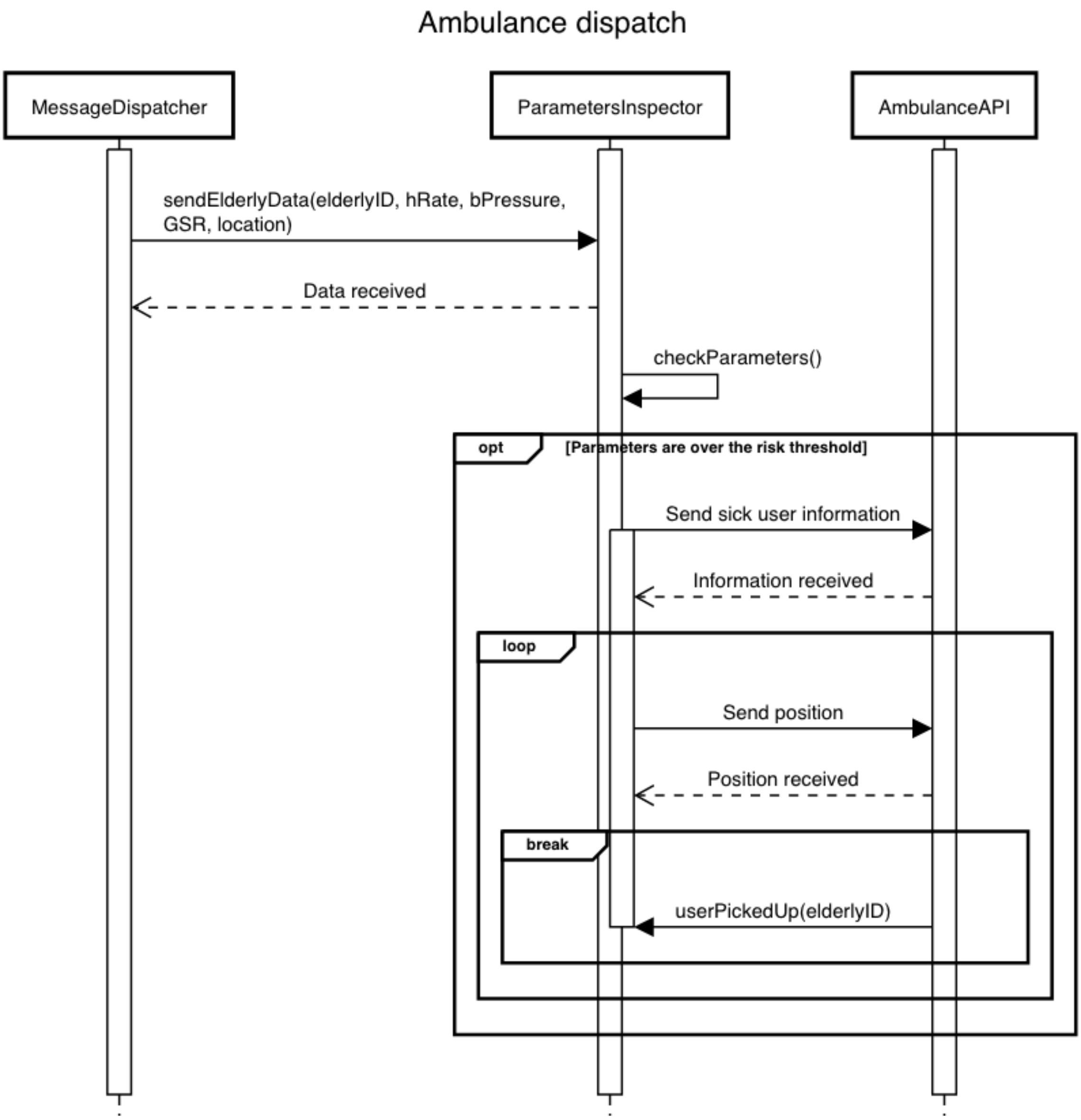


2.4 Runtime View

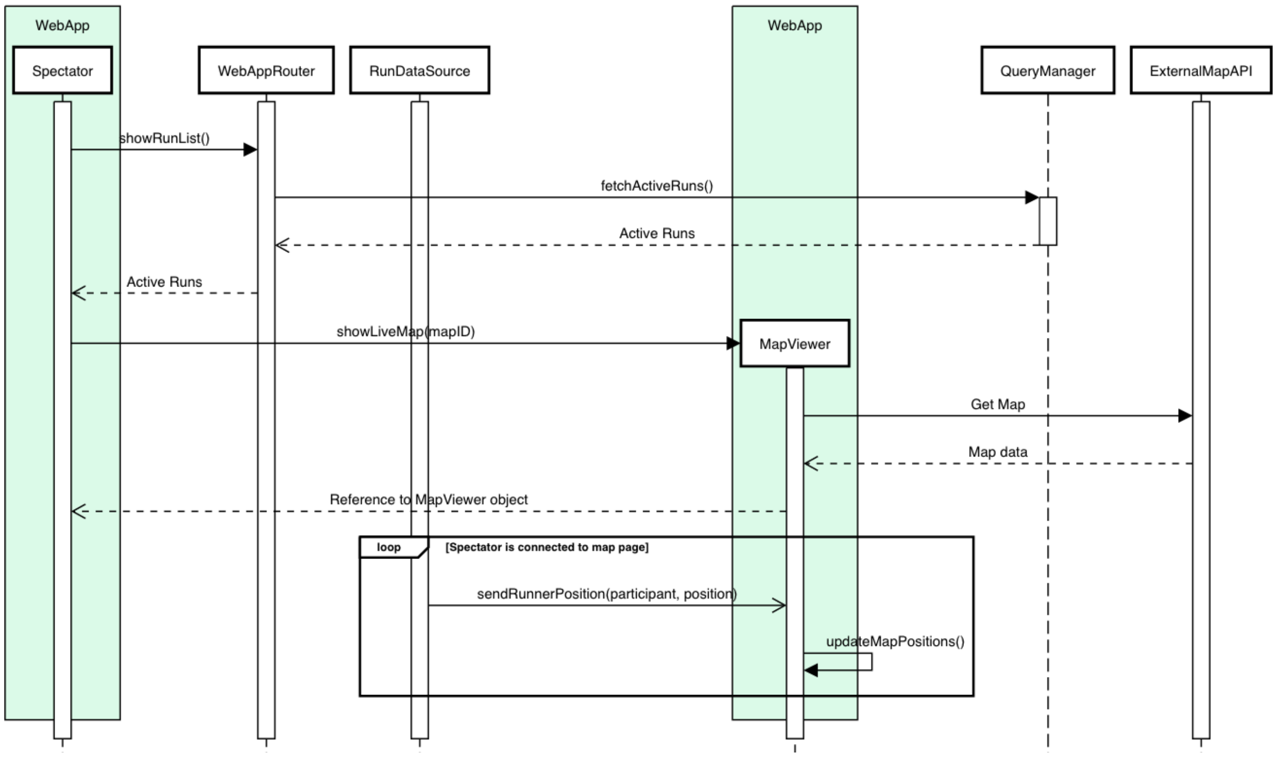




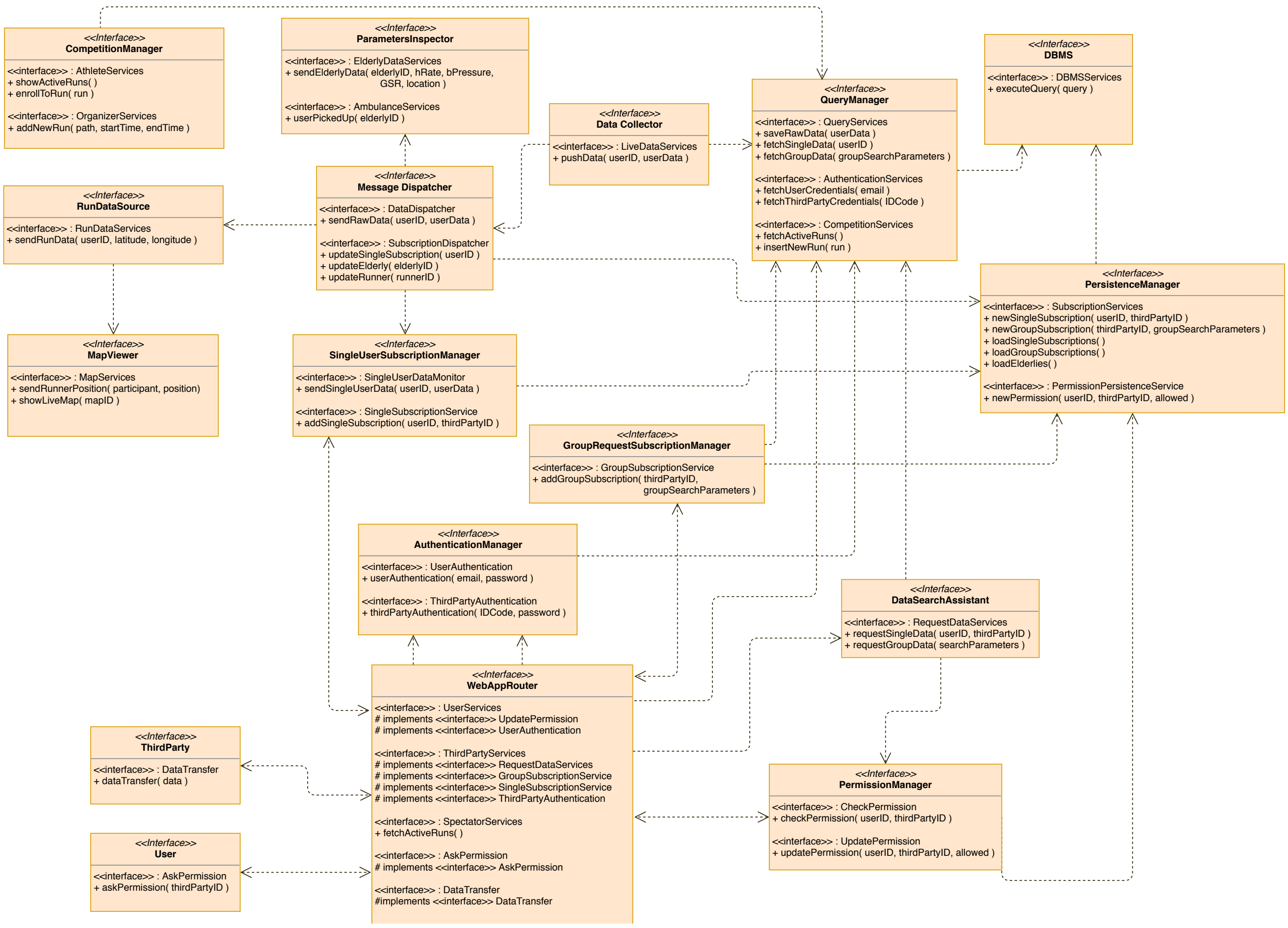

Single User query and subscription
RequestManager
DBServices
ThirdParty
WebAppRouter
AuthenticationManager
DataSearchAssistant
PermissionManager
SingleUserSubscriptionManager
QueryManager
PersistenceManager
loop
thirdPartyAuthentication(IDCode, password)
thirdPartyAuthentication(IDCode, password)
fetchThirdPartyCredentials(IDCode, password)
Third Party credentials
checkThirdPartyCredentials()
break
[Valid Credentials]
Login OK
Login OK
Login KO
Login KO
requestSingleData(userID, thirdPartyID)
requestSingleData(userID, thirdPartyID)
checkPermission(userID, thirdPartyID)
alt
[Permission existing]
Valid permission
fetchSingleData(userID)
Single User Data
Single User Data
Single User Data
opt
[Subscribe to new data]
addSingleSubscription(userID, thirdPartyID)
addSingleSubscription(userID, thirdPartyID)
newSingleSubscription(userID, thirdPartyID)
Subscription OK
Subscription added
Subscription added
[Permission missing or denied]
Missing or invalid permission
No available data
No available data


Ambulance dispatch
MessageDispatcher
ParametersInspector
AmbulanceAPI
sendElderlyData(elderlyID, hRate, bPressure, GSR, location)
Data received
checkParameters()
opt
[Parameters are over the risk threshold]
Send sick user information
Information received
loop
Send position
Position received
break
userPickedUp(elderlyID)

Spectate run



2.5 Component Interfaces



2.6 Selected Architectural Styles and Patterns

2.6.1 General architecture

To allow an easy logical distinction of subsystems' responsibilities, for this system we propose an architecture consisting of three tiers.
This kind of architecture well suits a client-server approach mixed with a thin client design, leaving to the client the sole responsibility of presenting the data to the user and acting as an interface to the server functionalities (*Presentation* tier).
Furthermore, the system to be relies heavily on data storage and later retrieval of this data, making a distinct database entity necessary. Encapsulating the DBMS and its related services into a logical persistence tier allows us to decouple the persistence structure from the rest of the system and to perform easier maintenance and changes, like modifying the DB infrastructure or services, without having the application tier to be informed of the real implementation of the persistence tier but just of its interfaces.

The business logic is handled by the Application tier, which manages every interaction between the system and the external world (users' clients, wearable, external services), and communicates with the Data Persistence tier, which stores all the application data.
Considering the topological deployment of the system, the Application Server exposes an API for the client, and the DB manager inside the application server makes requests to the DBMS.

2.6.2 Subsystems architecture

- **Data Collector**
  The *Data Collector* can be implemented following the event-based paradigm. The *Data Collector* itself acts as the event dispatcher: it receives data from users' wearables, and then broadcasts this data to the DBMS (in order to save the data) and to the *Message Dispatcher*.
  This paradigm allows a high modularity between different components of the system, significantly increasing decoupling and enhancing flexibility.
  Since all the data has its own timestamp, it's not relevant if the data collector dispatches not sorted data.
  The weakness of this paradigm is that *Data Collector* can become a bottleneck of the system. In order to address this problem, it is possible to set a policy for which, once a defined amount of wearable is subscribed to the data collector, a new instance of *Data Collector* is spawned, and all new wearables will subscribe to this new one.

- **Message Dispatcher**
  The *Message Dispatcher* uses the same architecture of the *Data Collector*.
  Its role is to send real-time data to the components that need them, specifically *Parameters Inspector*, *SingleUserSubscriptionManager* and *RunDataSource*.
  Considering this, we could describe the message dispatcher as an event handler, receiving data from *Data Collector* and broadcasting it to the components mentioned above.

  The *Message Dispatcher* stores internally a list of all *AutomatedSOS* users, single user subscriptions and the list of runners currently engaged in a competition. When some modifications of these data structures occur on the DB, the *Message Dispatcher* memory is updated accordingly. The *Message Dispatcher* can also load such data from the DB in case of reboot or failure of the component.

  The *Message Dispatcher* shall be very reliable, since the *Parameters Inspector* depends on it. For this reason, it is necessary to have multiple parallel instances of this component in order to increase robustness to failures.
  Regarding scalability, it is possible to use a load balancer which receives data from the *Data Collector* and distributes it to multiple instances of *Message Dispatcher*, accordingly to their requests load. These multiple instances shall use shared memory, ensuring that they all know to which component to redirect data.

- **Parameters inspector**

  The *Parameters Inspector* is an event listener too. It waits for users' data from *Message Dispatcher*, and then checks if the contained health parameters exceed the owner's risk thresholds.

  The *Parameters Inspector* stores internally a list of every *AutomatedSOS* user's risk thresholds, which needs to be consistent with the data on the DB. To ensure consistency with new subscriptions, whenever the component receives data about a user missing from the maintained list, it loads from the database the corresponding risk thresholds. In case of component failure or reboot, the *Parameters Inspector* is able to load from the database the complete list of elderlies and their threshold.

  If some user's health parameters exceed risk thresholds, the *Parameters Inspector* begins a rescue procedure, where it contacts the ambulance provider and provides the health and position data of the ill user.

  Due to the criticality of this component, it is necessary to have multiple instances in parallel, in order to increase fault tolerance.
  Also, since R1-NF requires the system-to-be to place a SOS request to the external provider within 5 seconds after detecting data exceeding risk thresholds, it is possible to use an elastic load balancer controlling the provisioning of *Parameters Inspector* instances, based upon the amount of data received, allowing the satisfaction of R1-NF.

- **WebApp Router**

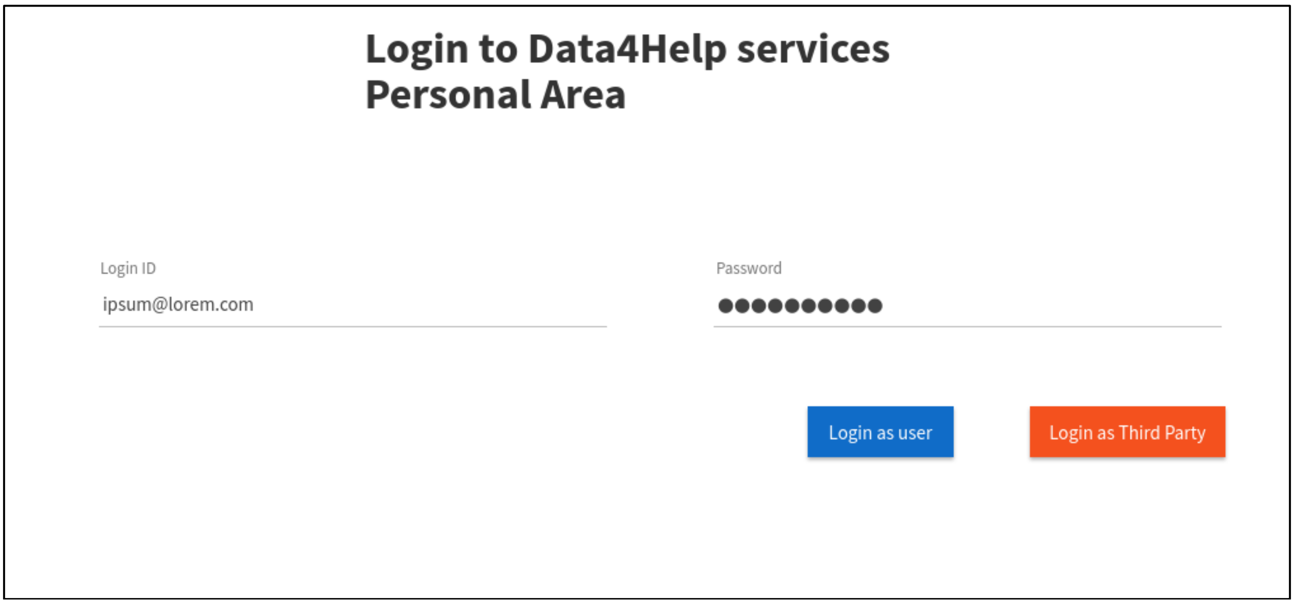
  The *WebApp Router* is the API exposed by the application server to the client. It handles the client-server interface and relays calls to the methods of the external interface to the single components implementing them.

  This kind of behaviour can be implemented through a façade pattern, allowing for a lightweight implementation and reducing the number of distinct ports and interfaces exposed to the client.

  Due to its "single access point" nature, this component could easily become a bottleneck to the system, so it shall be replicated and run in parallel. The amount of replication and the provisioning of new instances responsibilities could be assigned to a load balancer to adjust to the user client load and avoid severe failures.

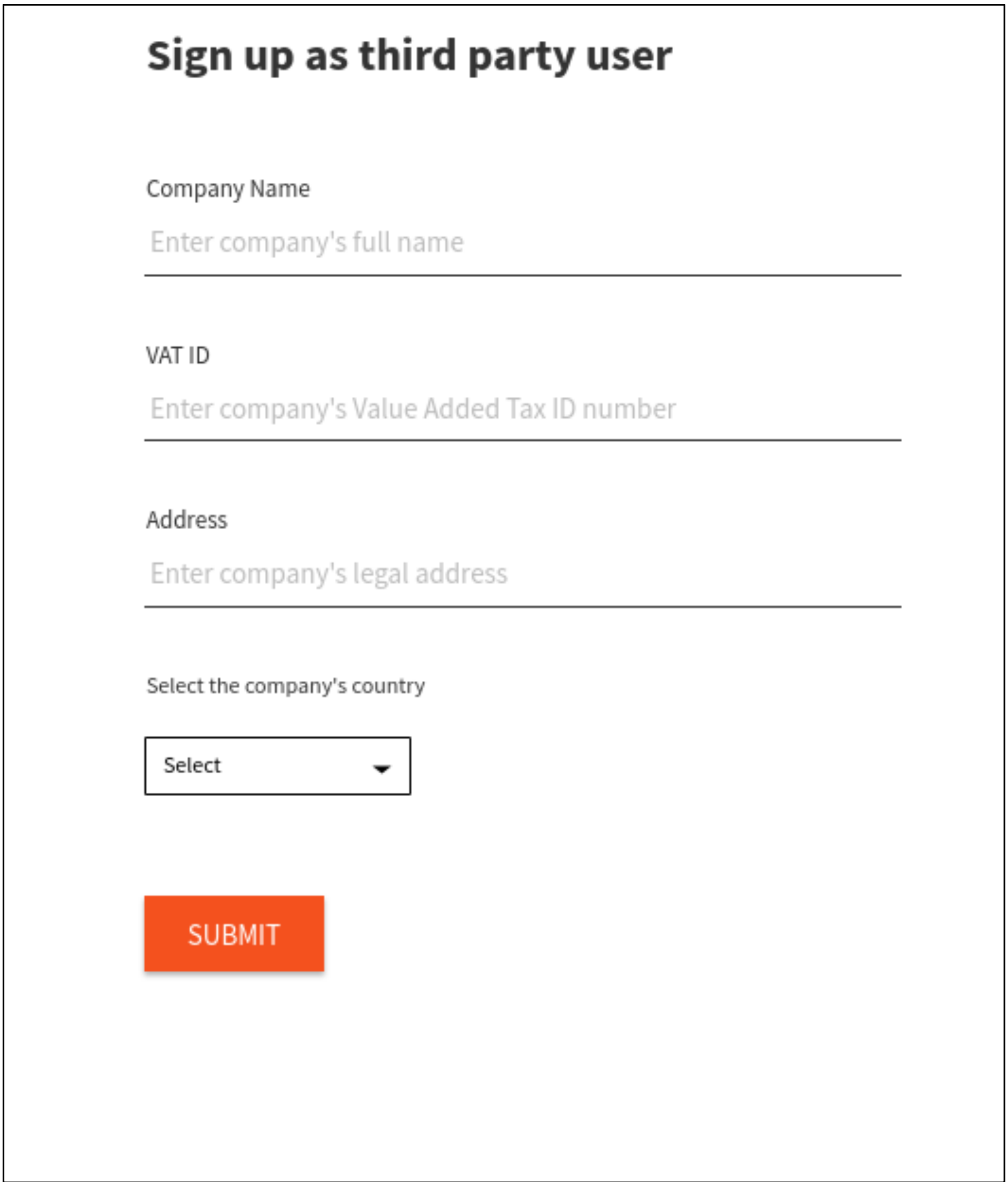
3 User Interface Design

3.1 Login

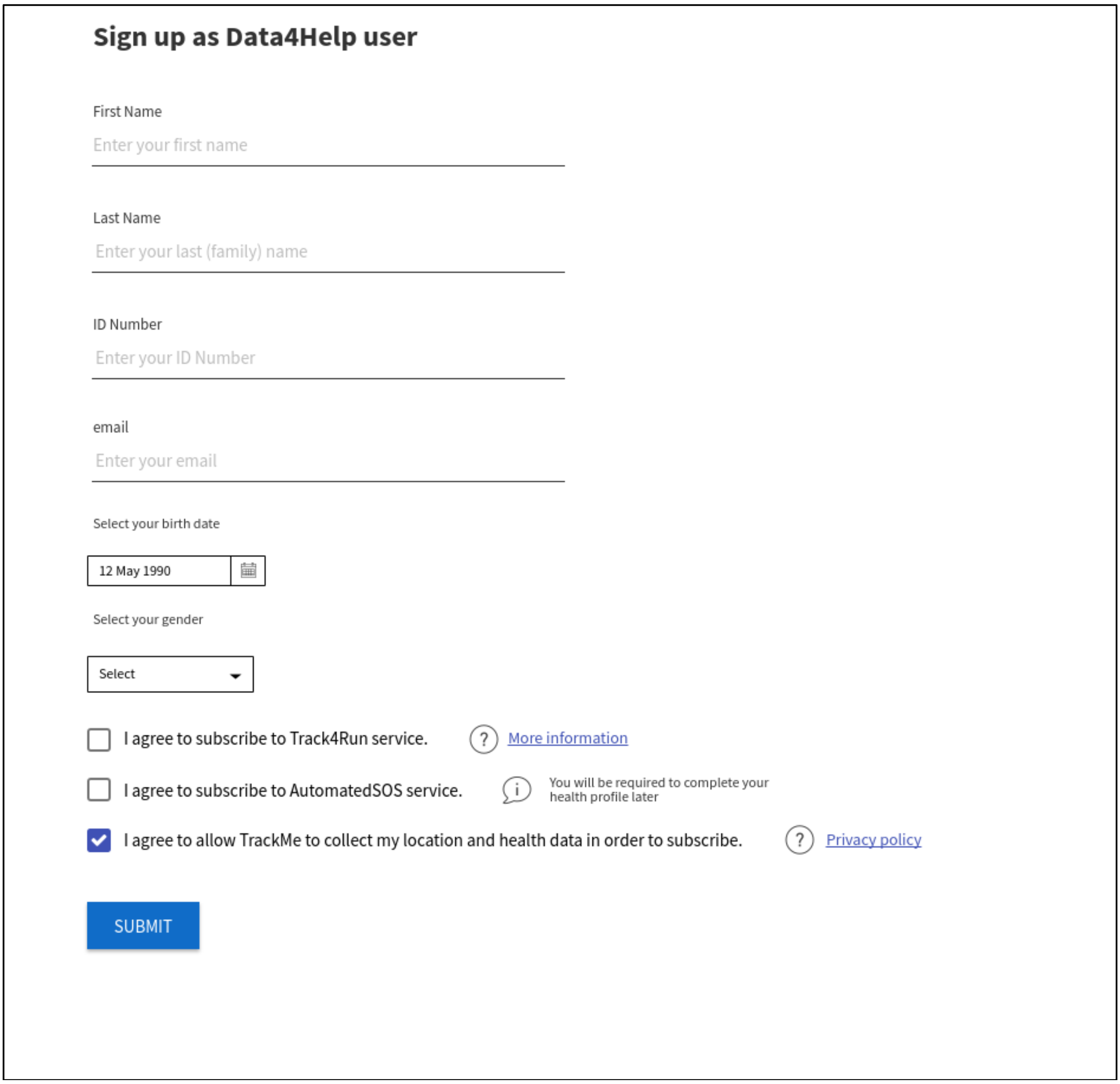


3.2 Signup

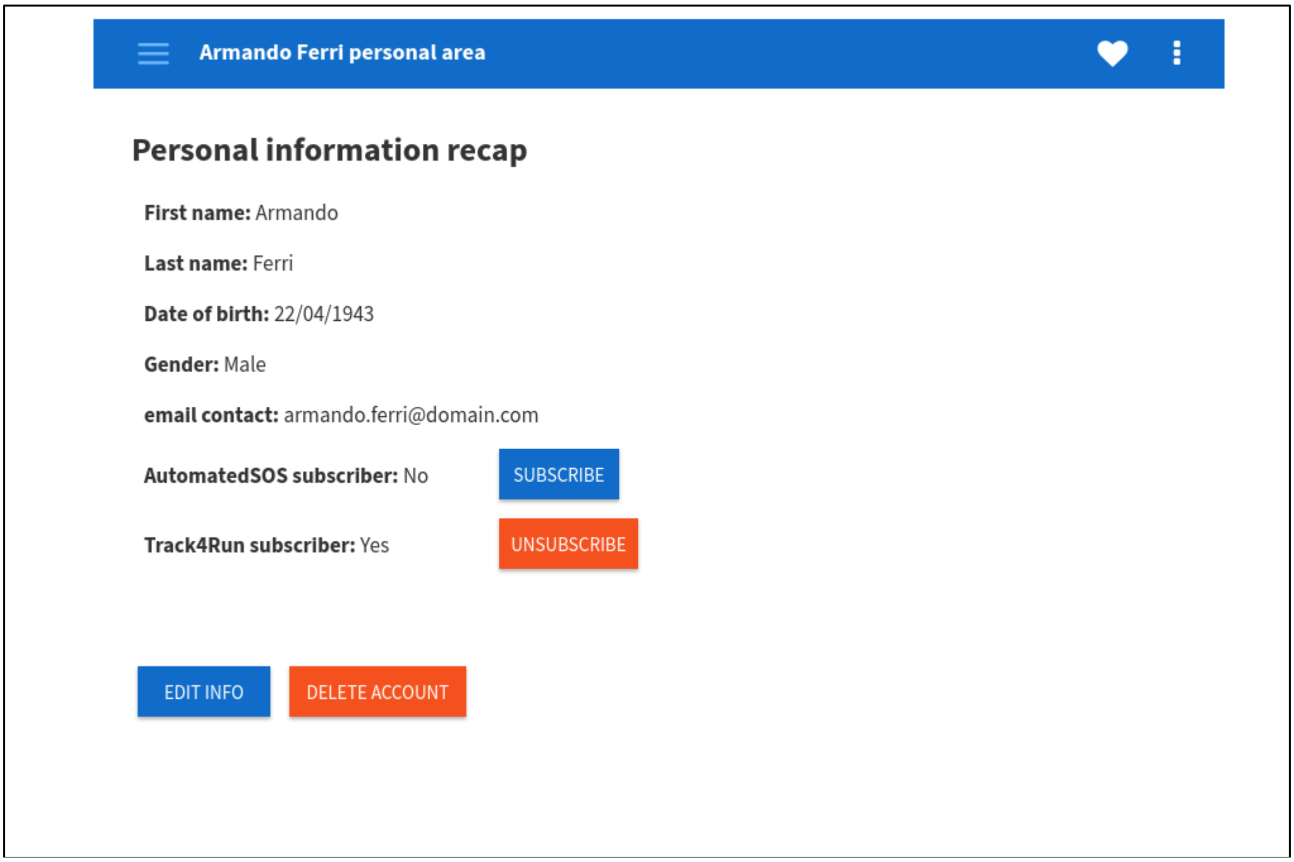
3.2.1 Third Party

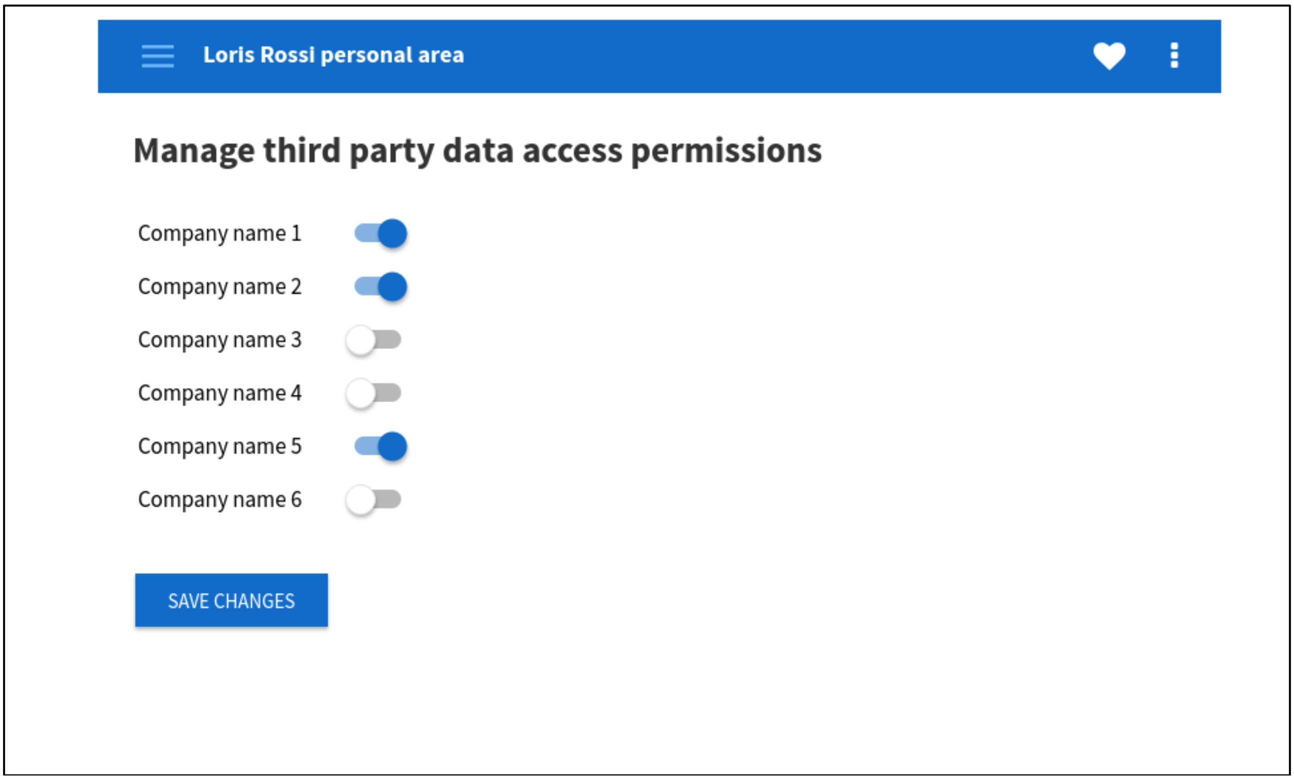


3.2.2 User

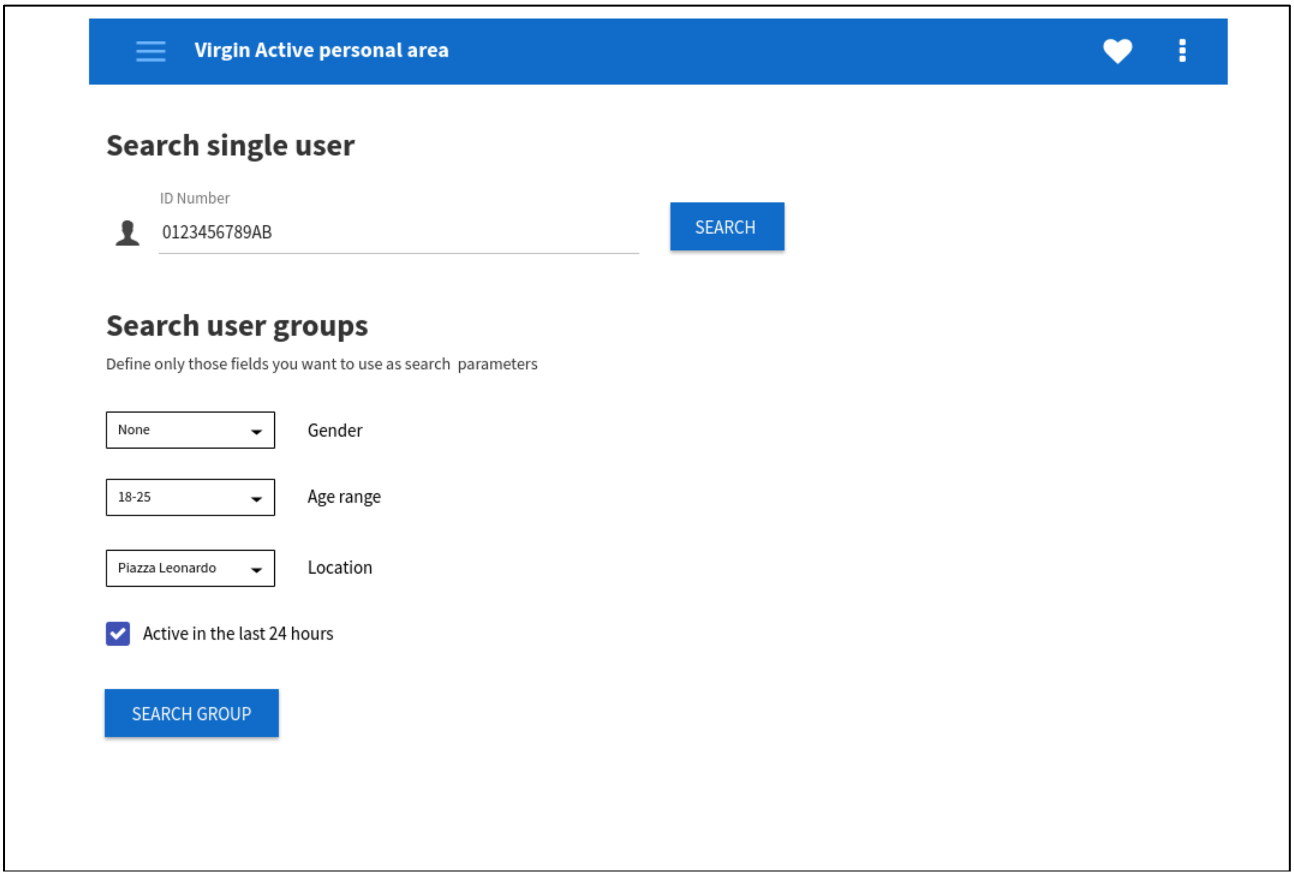

Sign up as Data4Help user
First Name
Enter your first name
Last Name
Enter your last (family) name
ID Number
Enter your ID Number
email
Enter your email
Select your birth date
12 May 1990
Select your gender
Select
I agree to subscribe to Track4Run service.
More information
I agree to subscribe to AutomatedSOS service.
You will be required to complete your health profile later
I agree to allow TrackMe to collect my location and health data in order to subscribe.
Privacy policy
SUBMIT

3.3 Users personal area

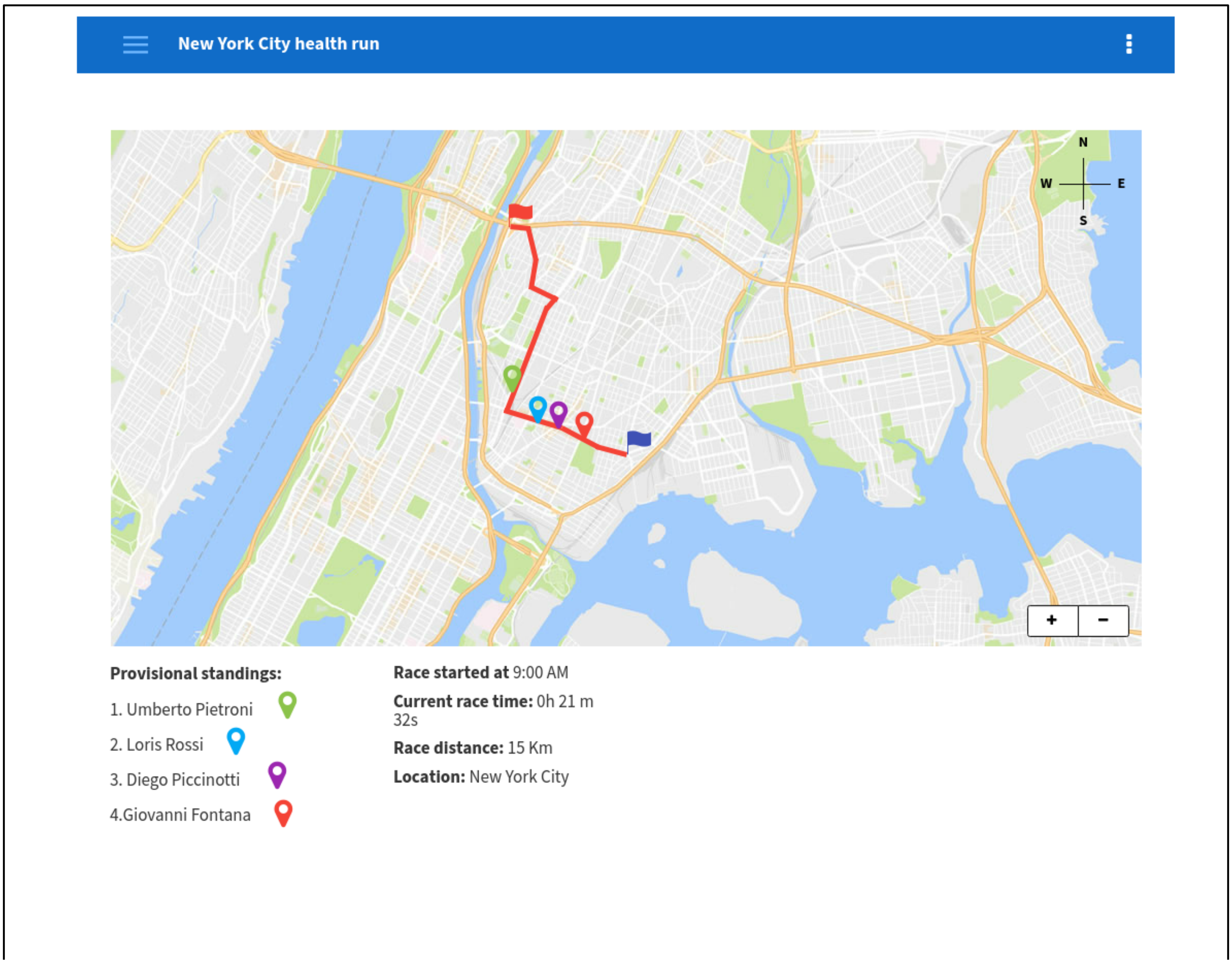




3.4 Third Party search page



3.5 Map view


New York City health run
N
W
E
S
+
−
Provisional standings:
1. Umberto Pietroni
2. Loris Rossi
3. Diego Piccinotti
4.Giovanni Fontana
Race started at 9:00 AM
Current race time: 0h 21 m 32s
Race distance: 15 Km
Location: New York City

4 Requirements Traceability

G1: The user can be recognized by providing a form of identification.

- *WebApp* **[R1, R3, R4]**
- *AuthenticationManager* **[R1, R2]**

[R1]: The system shall allow registration of individuals through an email and a password.
[R2]: The system shall guarantee the unicity of email addresses.
[R3]: The system shall ask users to provide their personal data (birthdate, gender, residency address, ID number).
[R4]: The system shall ask the user to agree to a policy that specifies that, by registering, users agree that TrackMe acquires their data.

G2: Allow third parties to monitor data about location and health status of individuals.

- *DataCollector* **[R5]**
- *MessageDispatcher* **[R7]**
- *DBMS* **[R5]**
- *Wearable* **[R5]**
- *WebApp* **[R6, R30]**
- *ThirdParty* **[R7]**
- *AuthenticationManager* **[R6, R30]**
- *GroupRequestSubscriptionManager* **[R7]**
- *SingleUserSubscriptionManager* **[R7]**

[R5]: The system shall store past position and health data of every single user.
[R6]: The system shall support the registration of third parties and provide them with a unique identifier (ID).
[R30]: The system shall support the login of third parties with their ID and password.
[R7]: Third parties shall be allowed to subscribe to new data and the system shall send data as soon as they are produced.

G3: Allow third parties to access data relative to specific users.

- *ThirdParty* **[R8]**
- *DataSearchAssistant* **[R8]**
- *PermissionManager* **[R9]**
- *User* **[R9, R32]**

[R30]: see G2.
[R7]: see G2.

[R8]: Third parties shall be able to request and receive a specific user's data by providing user's ID number.
[R9]: Upon every data collection request, the system shall check if the permission to access that data has already been granted by the user, otherwise it shall ask them

G4: Allow third parties to access anonymized data of groups of users.

- *ThirdParty* **[R10]**
- *DataSearchAssistant* **[R10, R11]**
- *GroupRequestSubscriptionManager* **[R12]**
- *MessageDispatcher* **[R12]**

[R30]: see G2.
[R7]: see G2.
[R10]: The system shall allow third parties to search for the desired group of individuals.
[R11]: The system shall accept a group data request only if the data can be properly anonymized. Anonymization is considered proper if the number of people involved in the request is greater than 1000.
[R12]: The system shall allow third parties to subscribe to periodic updates relative to a certain group of individuals provided that the data contained in each update can be properly anonymized.

G5: Allow third parties to offer a personalized and non-intrusive SOS service to elderly people so that an ambulance arrives to the location of the customer in case of emergency.

- *WebApp* **[R31]**
- *User* **[R13]**
- *AuthenticationManager* **[R31, R13]**
- *ParametersInspector* **[R14, R15, R16, R17]**
- *MessageDispatcher* **[R14]**

[R31] The system shall support the login of user with their email and password.
[R13]: The system, in order to allow Data4Help users to subscribe to AutomatedSOS, shall ask them to input their risk thresholds.
[R14]: Frequently enough, health parameters of each user are monitored by the system and compared against their threshold to detect risk situations.
[R15]: If a user's parameters get below risk thresholds the system shall contact the ambulance provider and require the dispatch of an ambulance.
[R16]: After an emergency request has been placed the system shall send data about the user and keep sending them regularly to the ambulance provider.
[R17]: The system shall allow the ambulance provider to notify when the user has been picked up, in order to stop data transmission.

G6: Allow athletes to enroll in a run.

- *User* **[R18]**
- *AuthenticationManager* **[R18]**
- *Athlete* **[R19, R20]**
- *CompetitionManager* **[R19, R20, R21]**

[R18]: The system shall allow athletes to register to the system.
[R31]: see G5
[R19]: The system shall allow athletes to check a list of available runs.
[R20]: The system shall provide the ability to enroll to the desired run only to registered athletes.
[R21]: The system shall allow enrolling only if the user has already agreed to share publicly his location for the duration of the run.

G7: Allow organizers to manage runs.

- *Organizer* **[R22, R23, R24, R25, R26]**
- *AuthenticationManager* **[R22]**
- *CompetitionManager* **[R23, R24, R25, R26]**

[R30]: see G2.
[R22] The system shall allow organizers to register to Track4Run platform.
[R23] The system shall allow organizers to create and delete races.
[R24] The system shall allow organizers to add a path for the run.
[R25] The system shall allow organizers to check a participants list.
[R26] The system shall allow organizers to add or remove participants before the start of the race.

G8: Allow spectators to see on a map the position of all runners during the run.

- *Spectator* **[R27, R28]**
- *DBMS* **[R27]**
- *MapViewer* **[R28, R29]**
- *RunDataSource* **[R28, R29]**

[R27] The system shall provide a public list of live runs and allow the spectator to select one of them.
[R28] The system shall allow the spectator to see a map of the desired run, with live participants' position.
[R29] The system shall update the positions of the runners on the map as soon as new data is received.

5 Implementation, Integration and Test Plan

5.1 Implementation Plan

The implementation of this system will be conducted in a bottom-up way by first implementing the low-level components which are used by other components and then going upwards in the system levels. Using this approach stubs won't be needed while testing and several subsystems will be already working and, eventually, they will be integrated into a single system.
However, the low-level components inside a higher-level component (for instance *DataCollector* and *MessageDispatcher* inside *LiveDataSource*) will be implemented in parallel and will be tested concurrently following incremental integration and testing approach (this will be better explained in the following paragraphs).

Following this approach, the components order of implementation shall be:

1. *DBServices (DBMS, QueryManager, PersistenceManager)*
2. *LiveDataSource (DataCollector, MessageDispatcher), Wearable*
3. *RequestManager (DataSearchAssistant, GroupRequestSubscriptionManager, SingleUserSubscriptionManager, PermissionManager)*
4. *WebApp (User and ThirdParty), WebAppRouter, AuthenticationManager*
5. *ParametersInspector*
6. *Track4RunCore*
7. *Track4Run WebApp (MapViewer, Athlete, Organizer, Spectator)*

The first component to be implemented will be *DBServices* because all the other components interact with it and it is a vital part of the system which must store and retrieve data from the database. Its sub-components *(DBMS, QueryManager, PersistenceManager)* are to be implemented concurrently, in order to integrate and test them as soon as a new version is available.
Another crucial component of the system is *LiveDataSource* which has the responsibility to collect each users' data and distribute it to *DBServices* and the third parties subscribed to them. For these reasons *LiveDataSource* is the second component that will be implemented together with *Wearable*.
When *DBServices* and *LiveDataSource* are fully implemented, the flow of data from the wearable of each user to the database is active, leaving only the need to implement the connection with third parties. *RequestManager* is the component that realizes this connection, exploiting the interfaces provided by the previously implemented components. Specifically, it uses *DBServices* interfaces in order to fetch data for the third parties' requests or subscriptions, and it takes advantage of *LiveDataSource* to provide data in real time. Since *RequestManager*'s sub-components don't interact with each other, it's not necessary to implement them in parallel. For instance, one choice could be to implement first

SingleUserSubscriptionManager and then test and integrate it with *DBServices* and *LiveDataSource* before starting implementing other components. Only *DataSearchAssistant* requires some functionalities of another component of *RequestManager*, *PermissionManager*, and for this reason it would be better to integrate them incrementally.
The last component needed to complete *Data4Help*'s functionalities is the *WebApp*. This component offers to the third parties and the users a way to communicate with others module with a GUI. During this implementation phase, other components such as *WebAppRouter and AuthenticationManager* will be implemented, since their functionalities are exploited by *WebApp* sub-components.
After these steps, the major functionalities of *Data4Help* will be available and on top of them *AutomatedSOS and Track4Run* services will be built separately. For the first one it is enough to implement *ParametersInspector* which uses an external API, exploits an interface of *DBServices* and provides an interface for *MessageDispatcher.*
Regarding *Track4Run,* first *Track4RunCore* component will be implemented and then the *Track4Run WebApp (MapViewer, Athlete, Organizer, Spectator),* which is a high-level component.

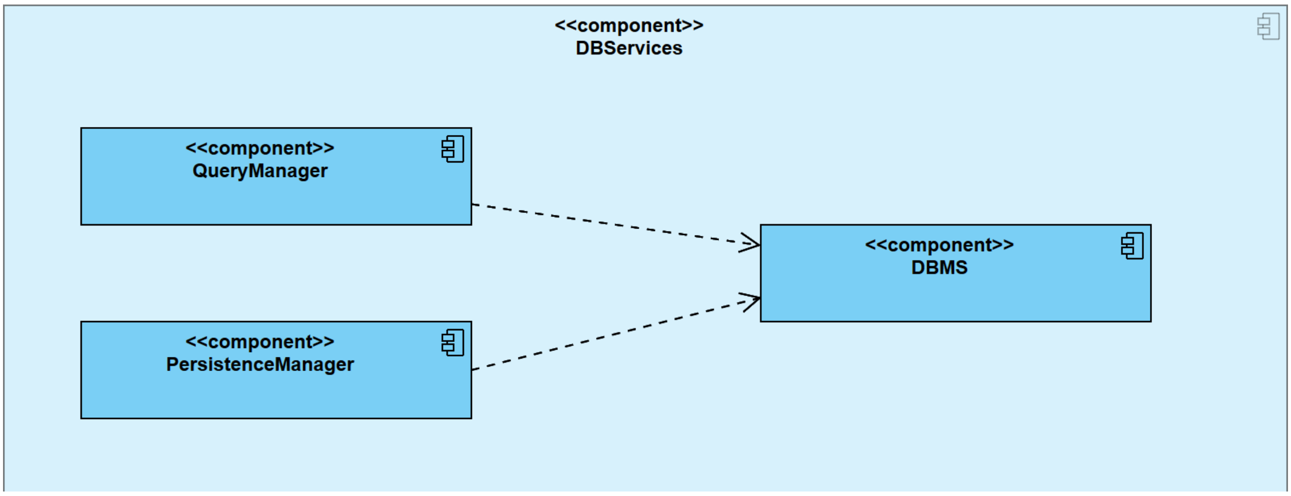
5.2 Integration and Test Plan

5.2.1 Integration and Testing Strategy

As it is stated in the previous section, the testing of this system follows a bottom-up strategy: it begins at the leaves of the "uses" hierarchy and goes upwards until several working subsystems are obtained and then integrated. To start testing at the bottom level of hierarchy means to test critical modules or functionalities at an early stage. This helps in early discovery of errors.
With this approach there is seldom need for stubs when performing unit-testing of each component, since the modules providing functions used by the component are already implemented. However, unit-testing requires the creation of drivers to invoke the component methods.
While high-level components like *DBServices, LiveDataSource, RequestManager, WebApp* are tested following a bottom-up strategy, the subcomponents inside each of them will be integrated by means of Incremental Integration Testing approach to check interfaces and flow of information between modules. As soon as new versions of components are available, they are integrated and tested, making it possible to obtain higher observability, diagnosability and less cost of repair.

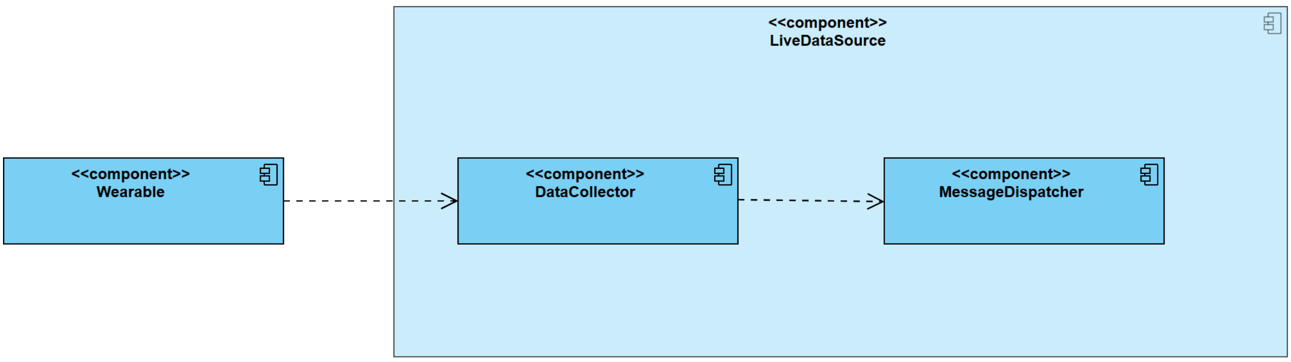
5.2.2 Integration Process

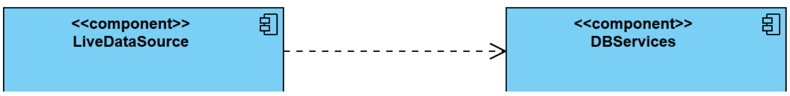
Integration of components inside DBServices



The first high-level component to be implemented and tested, as stated before, is *DBServices.* Inside *DBServices*, however, are three subcomponents interacting with each other.
The dashed arrow represents the dependency relation between two components. Here, for instance, both *QueryManager* and *PersistenceManager* use methods provided by *DBMS*'s interface.

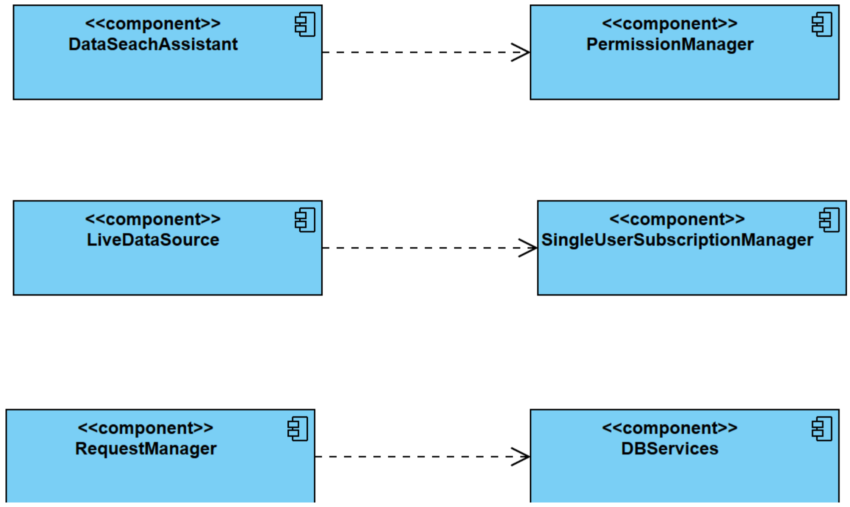
Integration of LiveDataSource components





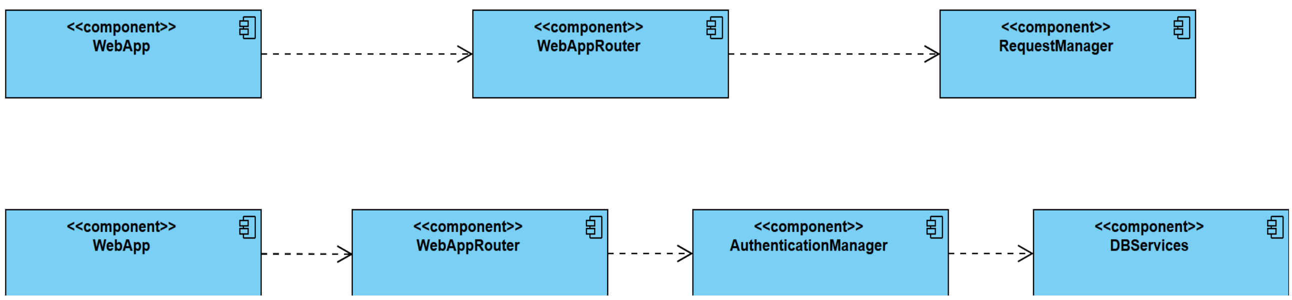
As the components inside *DBServices,* also those contained in *LiveDataSource* are integrated incrementally until all their functionalities are correctly implemented and tested.
Since *DBServices* is already implemented, *LiveDataSource*'s components are integrated and tested with it during this phase.

Integration of RequestManager components



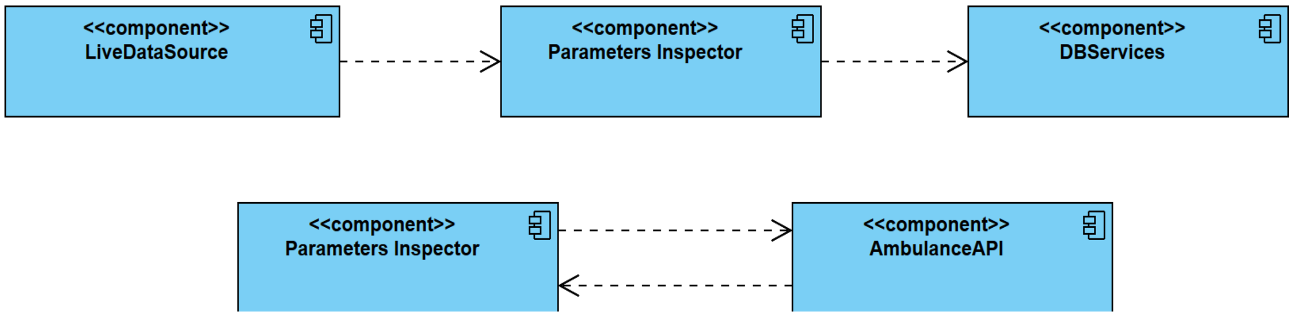
Components inside *RequestManager* don't interact with one another so they don't need to be integrated and tested together. The only exception is represented by *DataSearchAssistant*, which uses *PermissionManager*'s interface in order to check if a third party is allowed to search a specific user's data, therefore they should be integrated incrementally.
Another interaction which should be tested is the one between *MessageDispatcher*, which is a *LiveDataSource*'s component, and *SingleUserSubscriptionManager*.
During the implementation of each *RequestManager* component, it is possible to integrate and test it with *DBServices* which is already implemented.

Integration of WebApp components



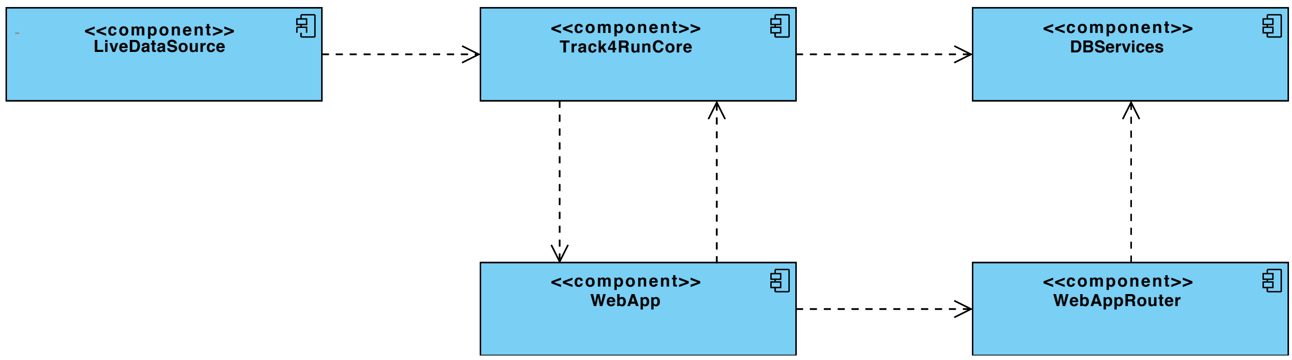
The last high-level component of *Data4Help* to be integrated is *WebApp*.
Due to their independency, its subcomponents *ThirdParty* and *User* don't need to be tested together. However, they will be integrated and subsequently tested with *RequestManager* and *DBServices*, the former by means of *WebAppRouter's* interface and the latter exploiting the functionalities offered by *AuthenticationManager*.

Integration of AutomatedSOS services



In order to implement *AutomatedSOS* services, the related component *ParametersInspector* must be integrated and tested with *LiveDataSource* and *DBServices* components. Furthermore, it is vital to test it with the external ambulance provider API (which is assumed to provide a sandbox environment for this purpose) to send and manage emergency requests.

Integration of Track4Run services



The two *Track4Run* components, *Track4RunCore* and *WebApp*, exploit each other's interfaces and therefore it's better to incrementally integrate and test them. Concurrently, to carry out their functionalities they will be integrated with *LiveDataSource*, *DBServices*, and *WebAppRouter*.

6 Effort Spent

6.1 Piccinotti Diego

Description of the task	Hours
Purpose, Scope, Definition	3
High-level Components and their Interaction	3
Component View	11.5
Deployment View	
Runtime View - Sequence Diagrams	5.5
Selected Architectural Styles and Patterns	1
Component Interfaces	3
User Interface Design	4
Requirements Traceability	1
Implementation, Integration and Test Plan	2

6.2 Pietroni Umberto

Description of the task	Hours
Purpose, Scope, Definition	2
High-level Components and their Interaction	4
Component View	6
Deployment View	
Runtime View - Sequence Diagrams	2
Selected Architectural Styles and Patterns	2
Component Interfaces	3
User Interface Design	
Requirements Traceability	3
Implementation, Integration and Test Plan	8

6.3 Rossi Loris

Description of the task	Hours
Purpose, Scope, Definition	3
High-level Components and their Interaction	3
Component View	5
Deployment View	2
Runtime View - Sequence Diagrams	4
Selected Architectural Styles and Patterns	3
Component Interfaces	6
User Interface Design	
Requirements Traceability	1
Implementation, Integration and Test Plan	2