POLITECNICO DI MILANO

DATA4HELP PRO by TRACKME

Design Document

Paolo Saccani
Diego Riva
Matteo Rubiu

A.Y. 2018-19 Software Engineering 2

INDEX OF CONTENTS

SECTION 1: Introduction

1.1 Purpose

In this document we present the architectural structure of the system, focusing on the following main themes:

- The high-level architecture.
- The design patterns.
- The main components and their interfaces.
- The Runtime behaviour.

In general, the purpose of this document is to provide a more technical guidance to develop a reliable and scalable system, for the implementation phase of the project, with the right trade-offs between cost and future-proof architecture.

1.2 Scope

Data4Help Pro is a software system which aims to offer services to two different types of customers:

- 3rd party companies.
- Single users.

Regarding the first category, the system must provide the possibility to perform queries over groups of users and to access personal data only with a corresponding authorization. In addition to this, 3rd party can offer to its customers to benefit from AutomatedSOS service and the creation and management of a run event through Track4Run services.

Regarding the second category, the system gathers and stores user's biometrical parameters and it offers to the user the representation of his/her data through graphs and diagrams. Users can also enrol to run events or be a spectator to an on ongoing one, watching athletes on a map in "real-time".

1.3 Definitions, Acronyms, Abbreviations

DBMS: Data Base Management System

GUI: Graphic User Interface

SN: Shared Nothing (Architecture)

API: Application Programming Interface

MTTF: Mean Time To Failure

MTBF: Mean Time Between Failure

BU: Bottom-Up Approach

TD: Top-Down Approach

1.4 Revision History

1.0 First Version

1.5 Documents References

1. Scalable Web Architecture and Distributed Systems – Kate Matsudaira: http://aosabook.org/en/distsys.html
2. GoF's Design Patterns in Java – Franco Guidi Polanco: http://eii.pucv.cl/pers/guidi/documentos/Guidi-GoFDesignPatternsInJava.pdf
3. Course Material – Matteo G. Rossi, Elisabetta Di Nitto: https://beep.metid.polimi.it/

SECTION 2: Architectural Design

2.1 Overview

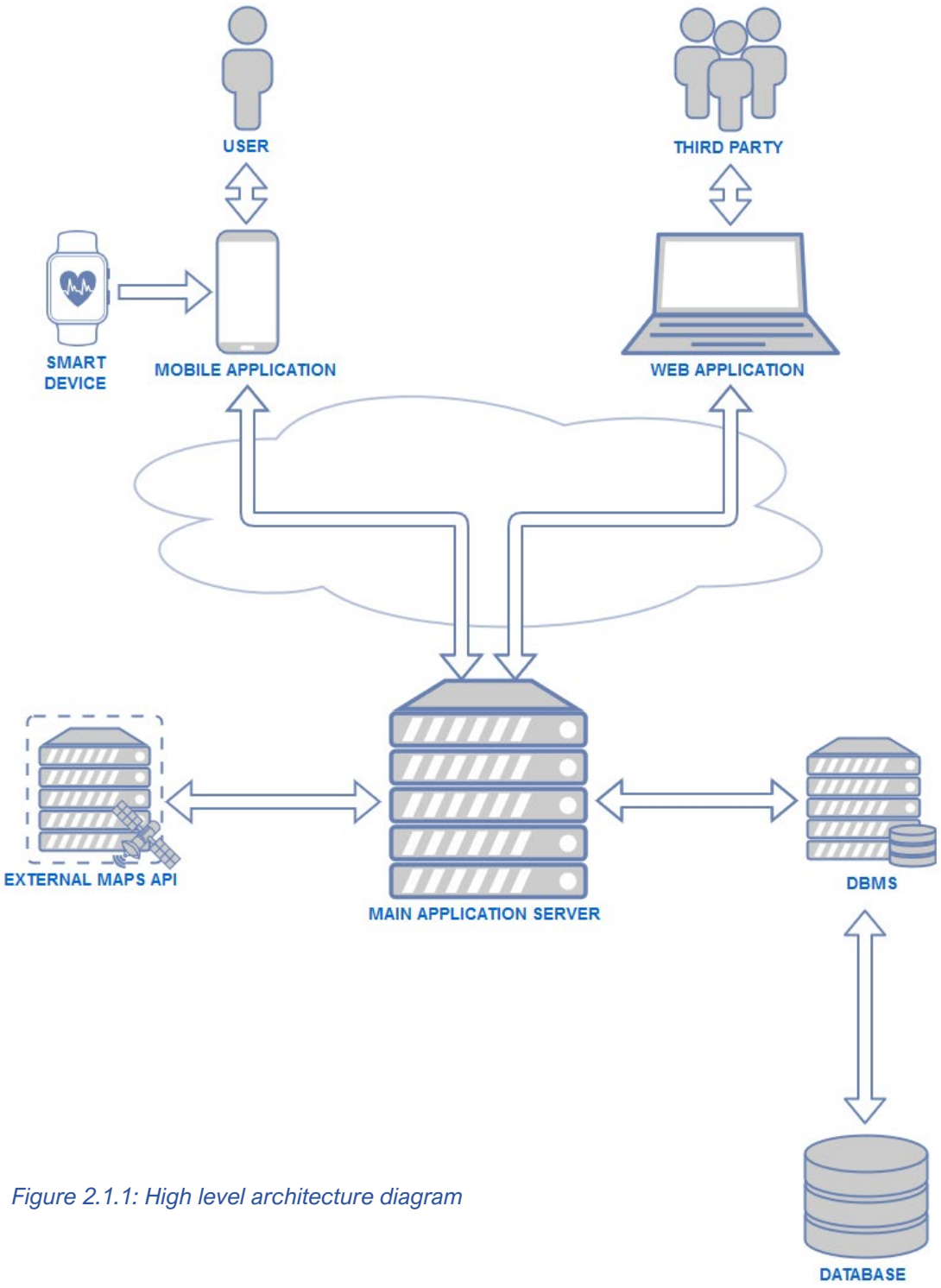


Figure 2.1.1: High level architecture diagram

The system architecture is addressed with in mind a top down approach, here we describe the higher-level concepts we are then going to deepen.

The paradigm is client/server, enabled by a 3-tier architecture: in particular a thin client one. This representation is a first “black box” look without falling into implementation details, which will be exploited further. In the ***Client layer:***

- The user has access to the system via a GUI.
- A 3rd party operator has access via a web application interface in his/her browser.

The business logic is almost completely held by the application server and the system must be designed to react in different ways based on the account/device type of the client, providing two types of interfaces.

The business logic will be implemented in the ***Application Server layer*** with a Shared Nothing architecture in mind: when needed a new node can be “elastically” added to distribute the computing power and improve the horizontal scaling.

Then each node will be connected to the ***Persistence layer*** where will be stored the users’ data and personal information in a relational database, reachable via the DBMS that will handle queries, with a defined policy, using queues divided by read/write actions and data-type.

There are then as layer expansion the external resources accessible via their proprietary API: at the moment maps and the ambulance dispatcher interfaces only. Through them the system gains functionalities and the possibility to exchange information via an agreed protocol.

In the Component View section will be explained in more detail what are the links between these levels and then will be deepen the architectural aspects with an impact on security, reliability and other important system’s attributes.

2.2 Component View

In this Component diagram we exploit the client and server side of our application, with focus on the application logic (figure 2.2.1).

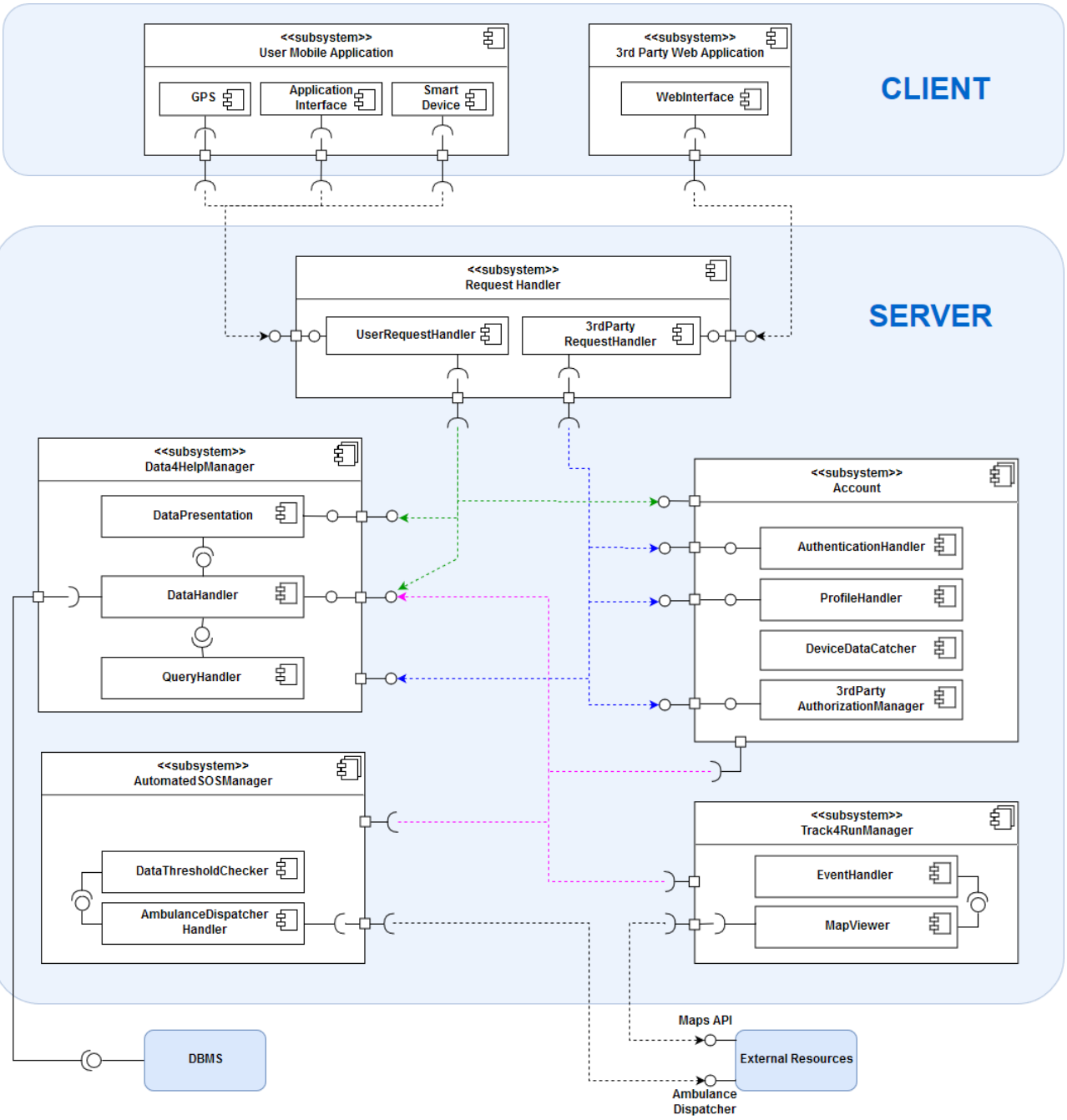


Figure 2.2.1: component diagram

As anticipated, the client side is composed by two subsystems that act as a components for interfacing with our back-end logic, except for the ***UserMobileApplication*** one that is equipped with components able to interpret the data from wearables and GPS information, respectively:

- *SmartDevice*
- *GPS*

Based on clients' actions are then assembled the requests for the system, all of which pass by the entry point subsystem ***RequestHandler*** which already adds a first layer of security and division of permissions through this software design pattern (section 2.3).

The same concept is applied to the exit point towards the Persistence Tier, of which management is in charge the *DataHandler* component.

The server side is divided into subsystems to better distinguish paths and interactions of a service offered by the application. They are the following:

- ***RequestHandler*** - the components act as a controller to redirect to the right interface for the service requested, also dividing the user and the 3rd party logic.
- ***Data4HelpManager*** - this is the base module of the system, along with the account handling one, on top of which the services are added: the components concern here are data aggregation and management.
- ***Account*** - here are contained the components able to administrate the bindings between personal information, devices and authorizations along with a login manager.
- ***AutomatedSOSManager*** - this subsystem adds the interfaces to offer the automated version of the first aid service described before. It handles biomedical data checks and the generation of the right ambulance dispatcher call.
- ***Track4RunManager*** - this subsystem adds the interfaces to offer the management of a run event form both sides: user and organizer, it also provides the connection to the external maps' API, useful for path creation and for following "LIVE" the athletes.

2.3 Deployment View

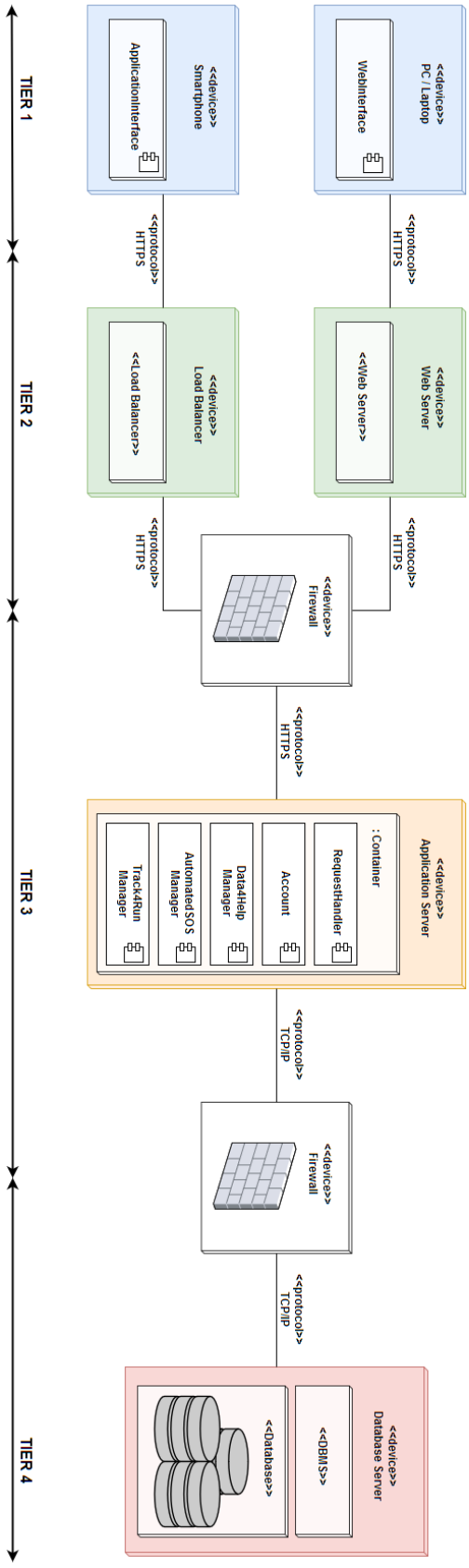


Figure 2.3.1: deployment diagram

In this section we go more into detail for what concern the real world in which the services will be deployed: different artefacts distributed through the different targets' nodes of the architecture. The main difference from the initial overview is that the solution presented here tries to touch also more specific aspect like security or scalability and availability attributes that tend to become of critical importance when the use base grows. In many cases has been proven that proposing a future-proof architecture gives a company a huge business advantage over the competition. We think then that should be implemented, as minimum security's pledge, an entrance and an exit firewall for the Application Server layer.

Our architecture then at this level of detail resembles more a 4-tier one (figure 2.2):

- The first tier is composed by the smartphone application and the web browser through which the respective features and functionalities are provided to clients.
- The second tier contains the web server which will implement servlets for the creation of static and dynamic pages and their rendering, along possibly a load balancer. For sure this last device (hardware or software one) should be a must for the mobile application link to the Server, because the number of users is going to be drastically greater and they will be more active due to the amount of data continuously generated that needs to be uploaded.
- The third tier is represented by the Application Server (or Farm) on which will run a containerized version of the system, easily scalable and redundant to avoid major pitfalls.
- The fourth tier holds the persistence layer of the system: the database system and his DBMS

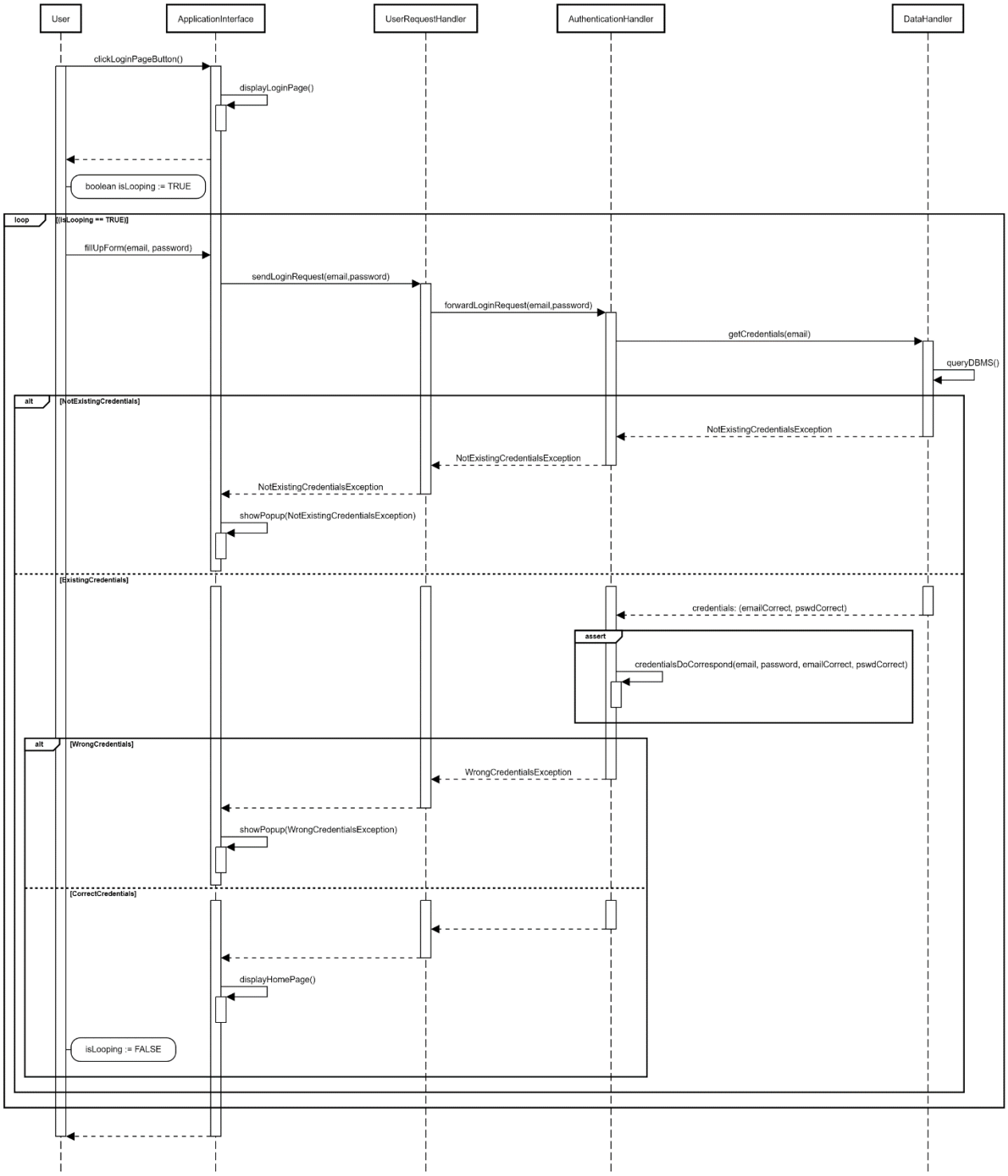
2.4 Runtime View

In the following section we're going to analyse our system through the help of some *Sequence Diagrams,* involving many different interactions between the components described inside section 2.2, and with refer to the use cases defined in the RASD.

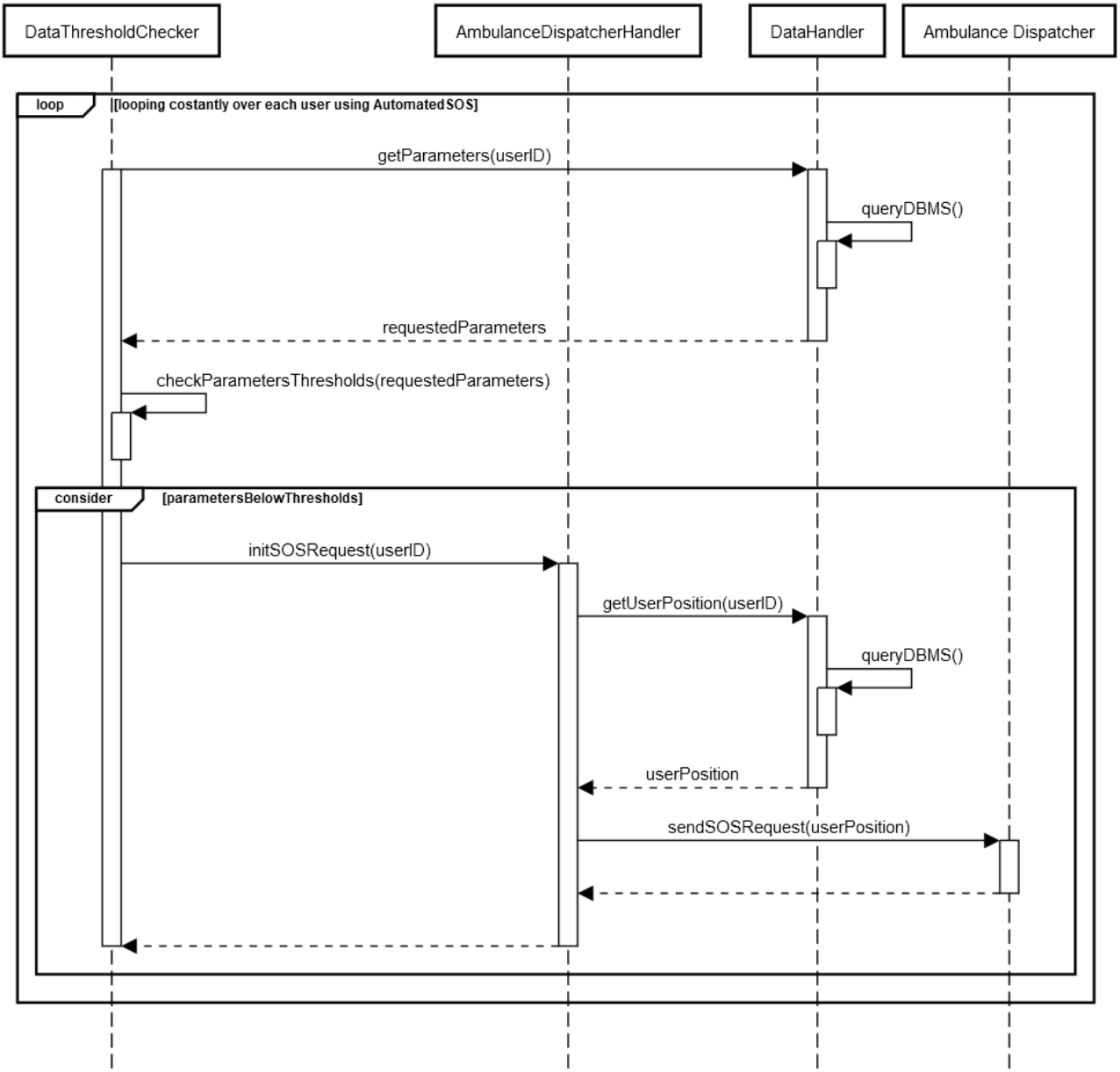
Here's a quick list of all diagrams:

1. **Login**: describe the login process in the application from the User's point of view (it's similar for the 3rd Party, so it's omitted the diagram);
2. **Profile**: describe the profile functionality in the application from the User's point of view (it's similar for the 3rd Party, so it's omitted the diagram);
3. **AutomatedSOS:** describe how does it work the *automatedSOS* functionality, when an SOS call is being sent to the dispatcher;
4. **Smart Device & GPS Data Collecting:** describe how data are collected from the *Smart Devices* and the *GPS* components of our users;
5. **User 3rd Party Authorizations:** describe how the user can manage its own authorizations to 3rd Parties from its *Profile* section on the mobile application, deciding whether revoking them or not;
6. **User View Biological Data:** describe how to user access its own biological data from the mobile application;
7. **User Track4Run:** describe how it's managed the *Track4Run* from the user point of view, inside the mobile application;
8. **3rd Party Track4Run:** describe how it's managed the *Track4Run* from the 3rd Party point of view, inside the web application;
9. **3rd Party View User Data:** describe how the 3rd Party access single user data from the Web Application;
10. **3rd Party Add User:** it's an extension of the previous diagram, describing how a user is added by the 3rd Party, with all the process involving the request and the response directly from the user, which choose to grant or not the permission;
11. **3rd Party Group Query:** describe how 3rd Party performs queries over group of selected users (by parameters), and how our system manages this.

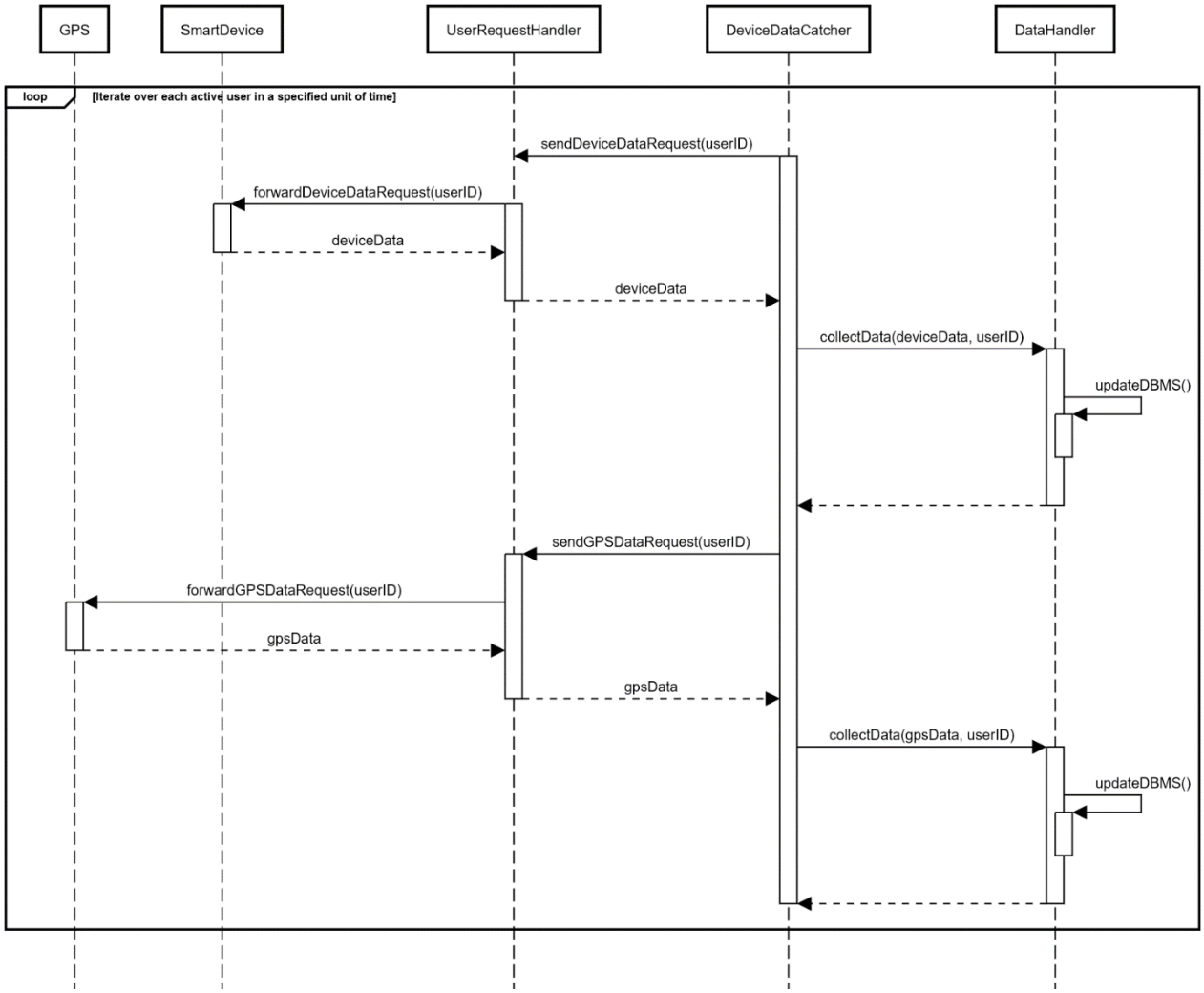
Login Sequence Diagram

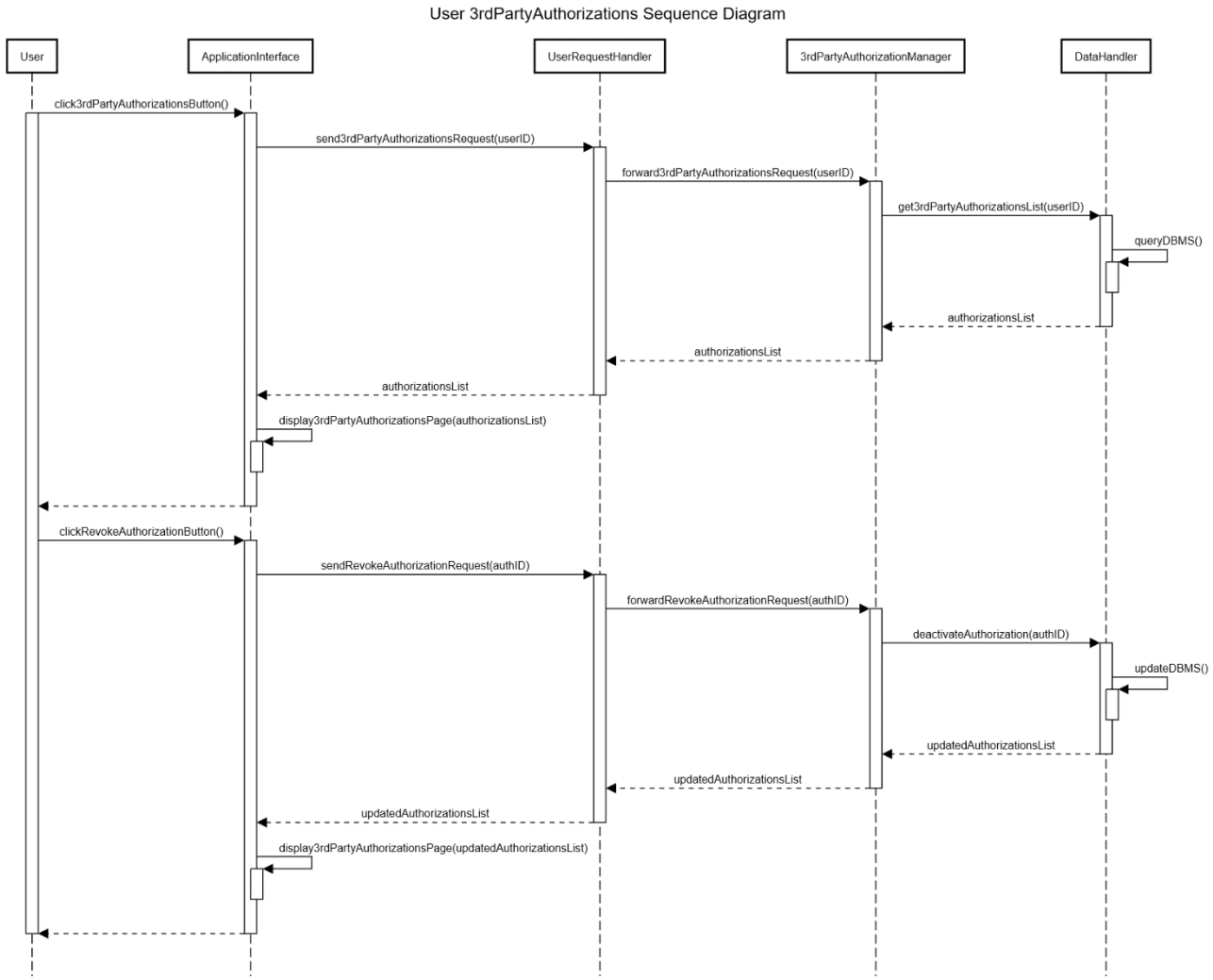


AutomatedSOS Sequence Diagram

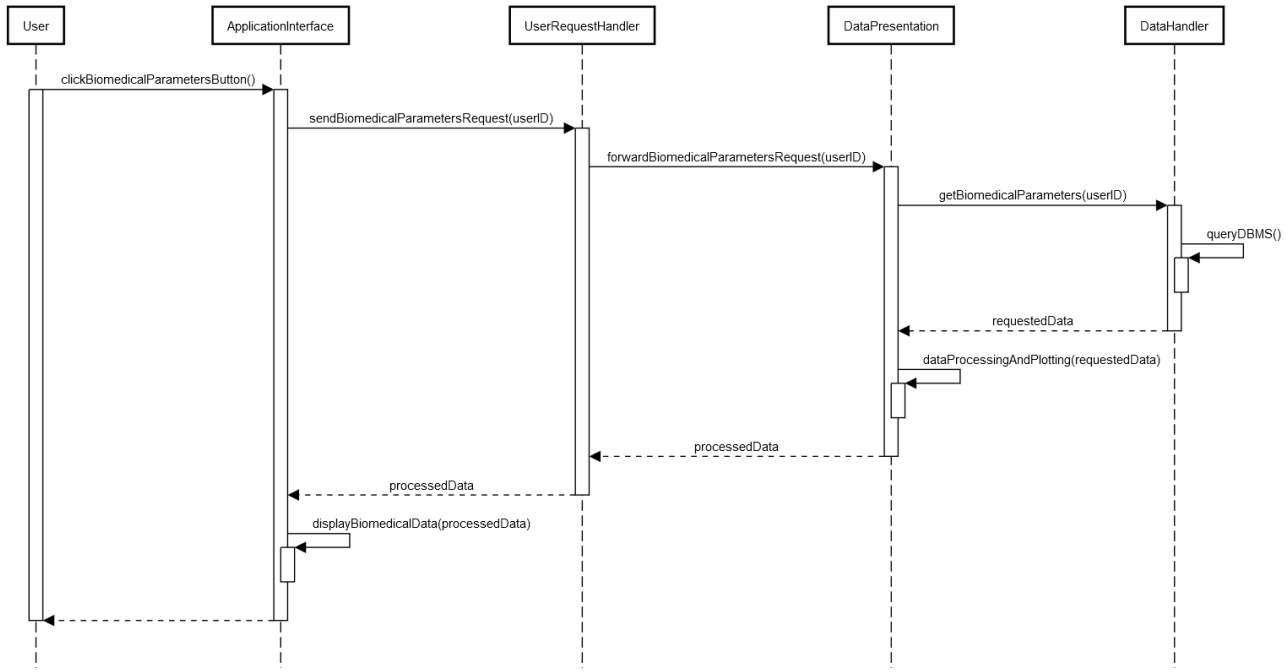


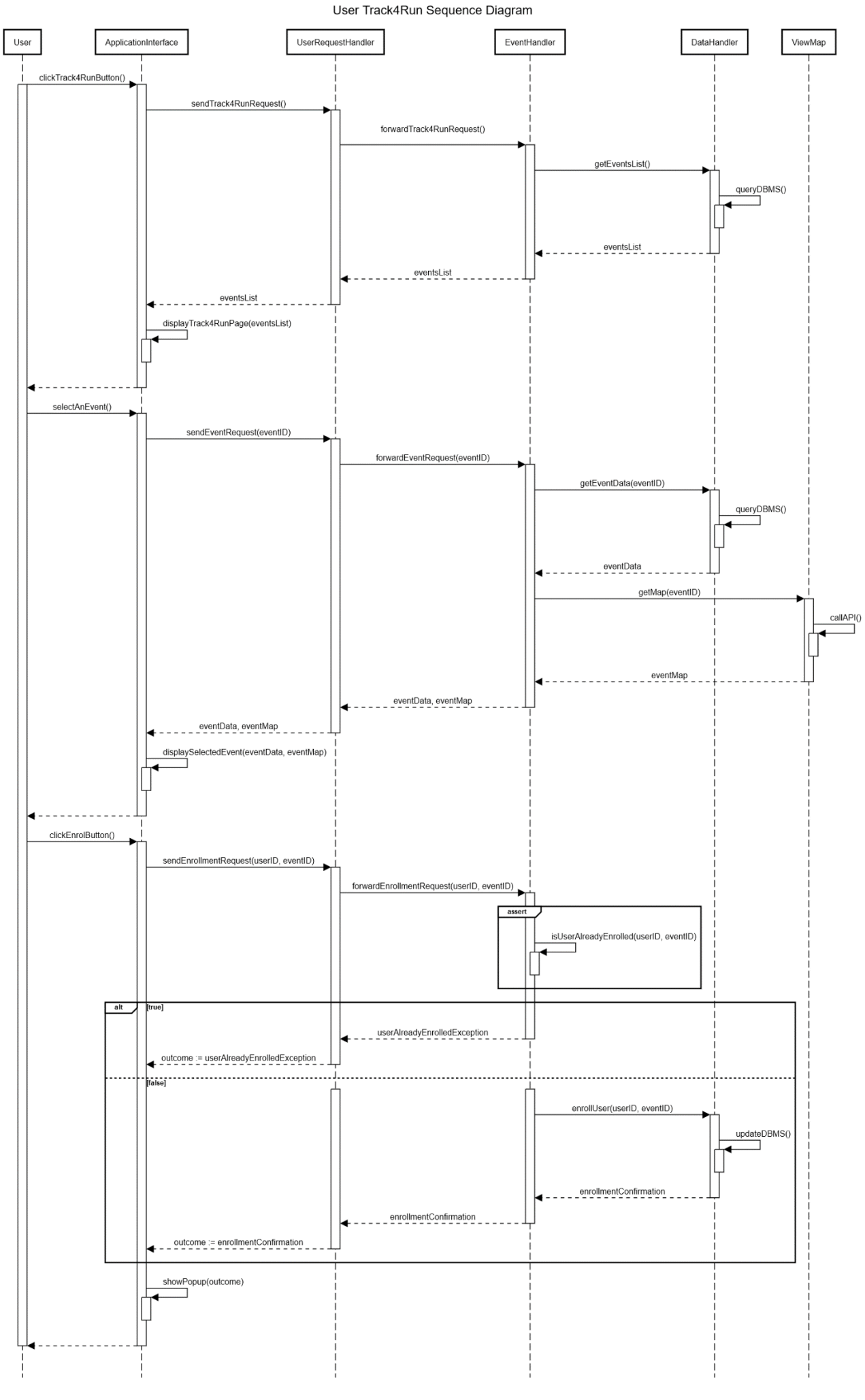
Smart Device & GPS Data Collecting Sequence Diagram

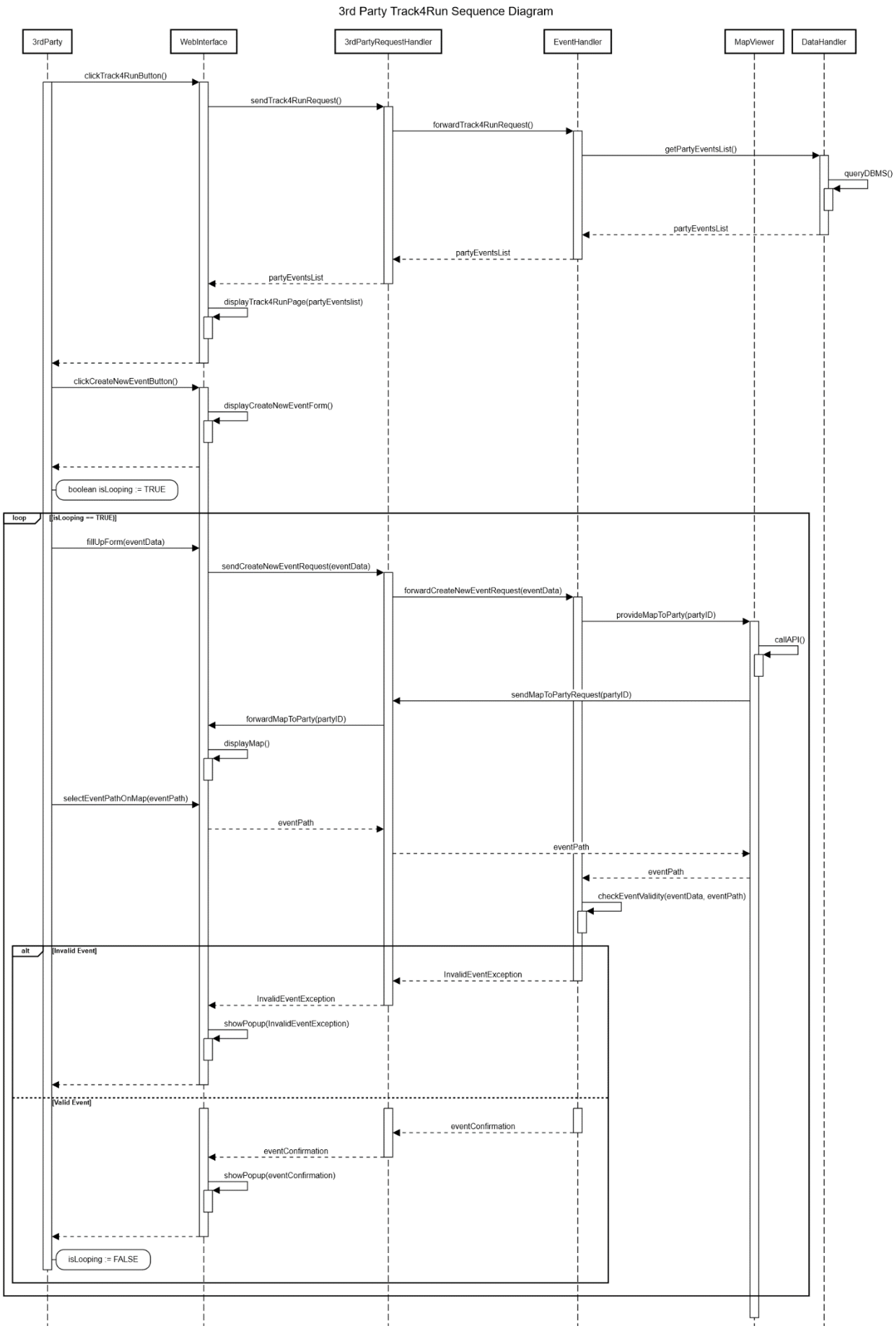


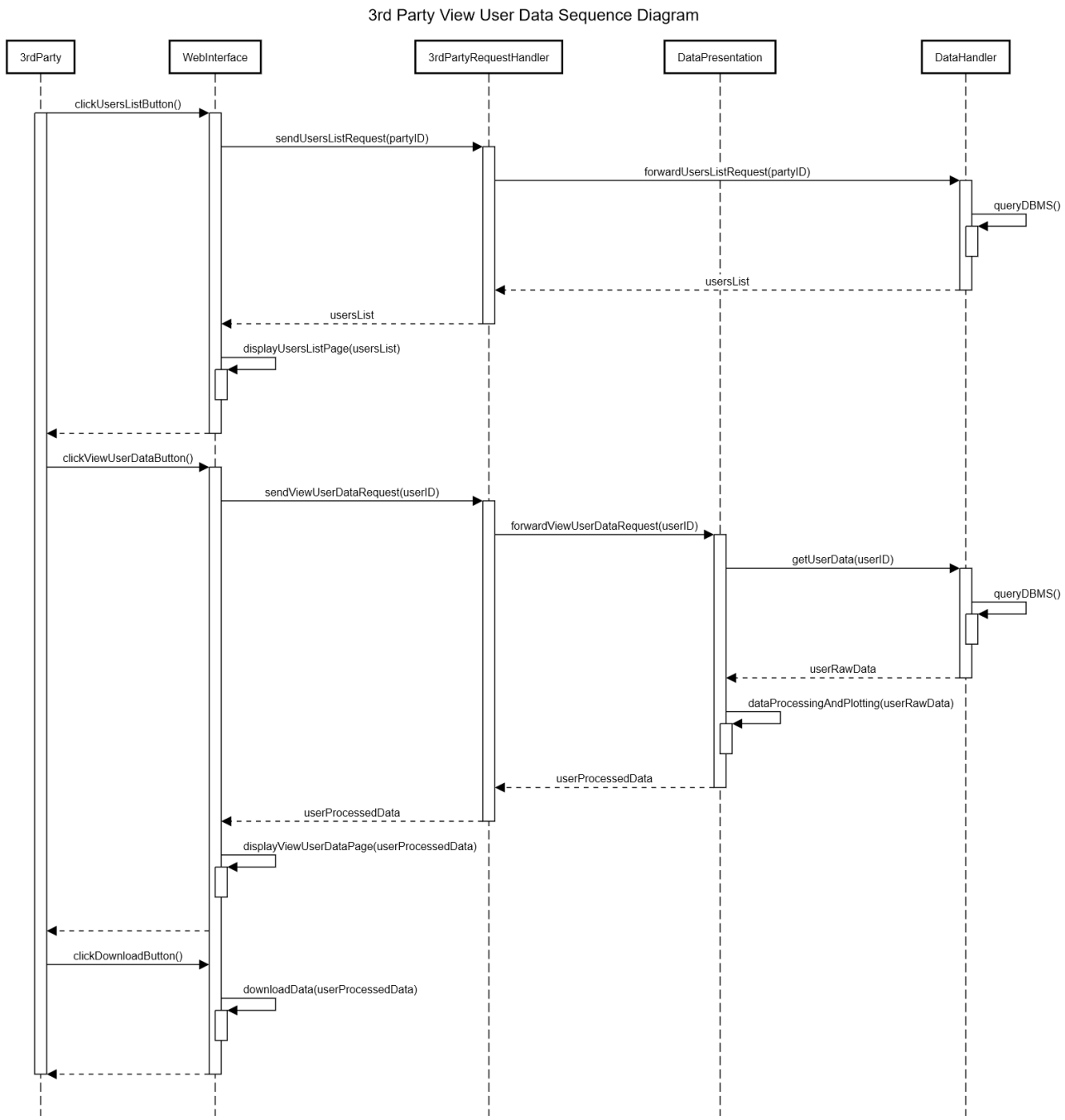

User 3rdPartyAuthorizations Sequence Diagram
User
ApplicationInterface
UserRequestHandler
3rdPartyAuthorizationManager
DataHandler
click3rdPartyAuthorizationsButton()
send3rdPartyAuthorizationsRequest(userID)
forward3rdPartyAuthorizationsRequest(userID)
get3rdPartyAuthorizationsList(userID)
queryDBMS()
authorizationsList
authorizationsList
authorizationsList
display3rdPartyAuthorizationsPage(authorizationsList)
clickRevokeAuthorizationButton()
sendRevokeAuthorizationRequest(authID)
forwardRevokeAuthorizationRequest(authID)
deactivateAuthorization(authID)
updateDBMS()
updatedAuthorizationsList
updatedAuthorizationsList
updatedAuthorizationsList
display3rdPartyAuthorizationsPage(updatedAuthorizationsList)

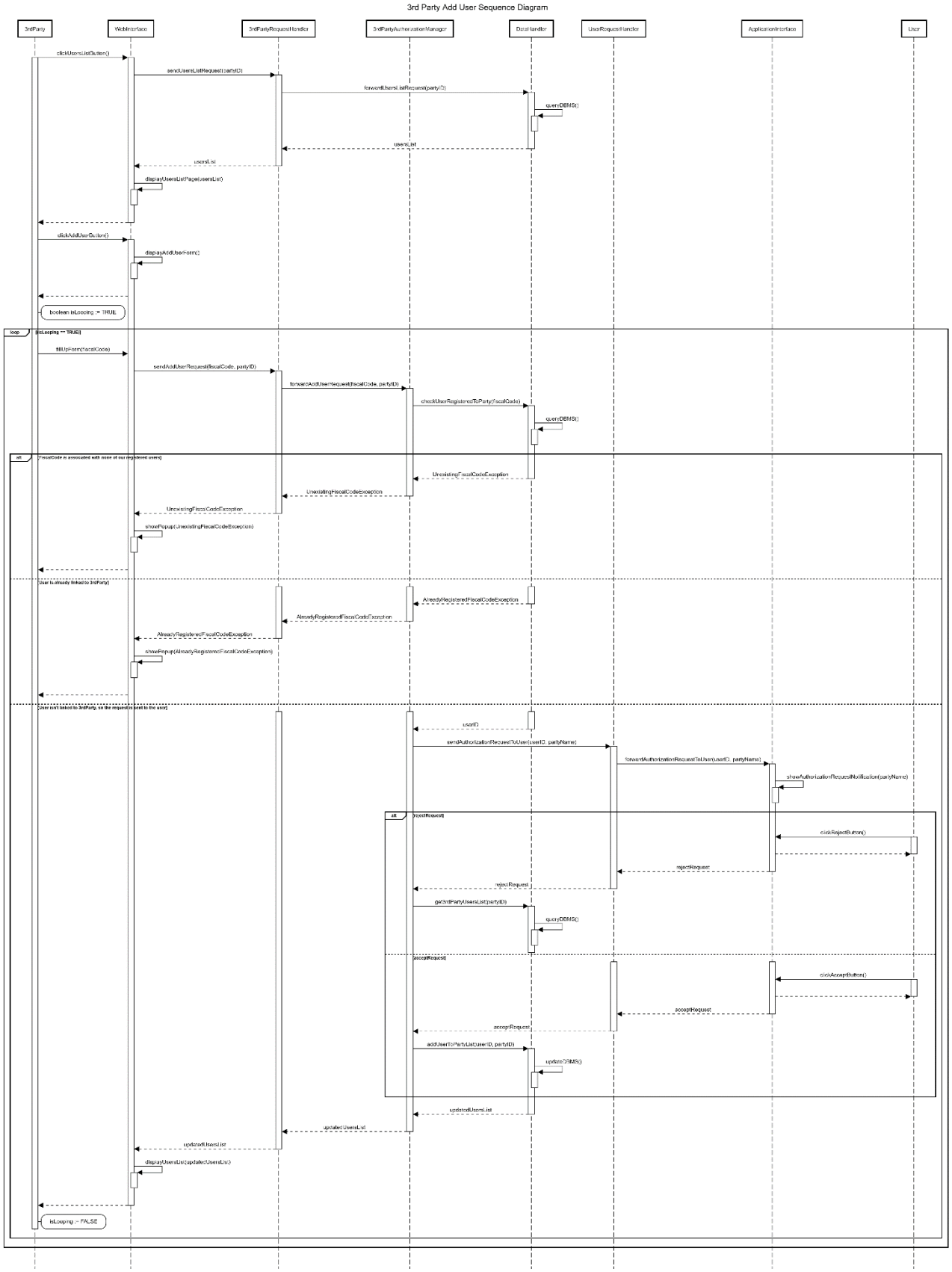
User View Biological Data Sequence Diagram



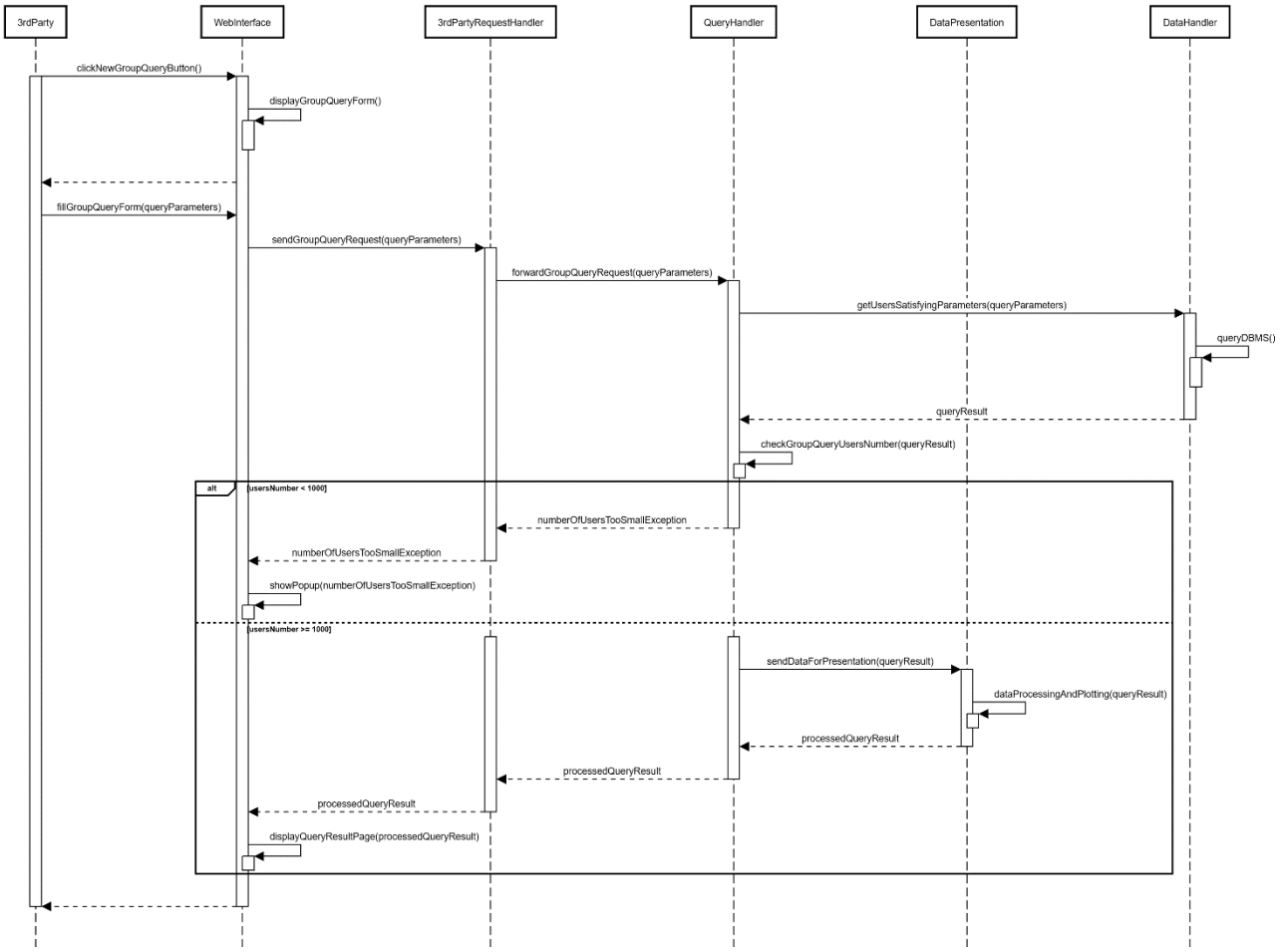

User Track4Run Sequence Diagram
User
ApplicationInterface
UserRequestHandler
EventHandler
DataHandler
ViewMap
clickTrack4RunButton()
sendTrack4RunRequest()
forwardTrack4RunRequest()
getEventsList()
queryDBMS()
eventsList
eventsList
eventsList
displayTrack4RunPage(eventsList)
selectAnEvent()
sendEventRequest(eventID)
forwardEventRequest(eventID)
getEventData(eventID)
queryDBMS()
eventData
getMap(eventID)
callAPI()
eventMap
eventData, eventMap
eventData, eventMap
displaySelectedEvent(eventData, eventMap)
clickEnrolButton()
sendEnrollmentRequest(userID, eventID)
forwardEnrollmentRequest(userID, eventID)
assert
isUserAlreadyEnrolled(userID, eventID)
alt
[true]
userAlreadyEnrolledException
outcome := userAlreadyEnrolledException
[false]
enrollUser(userID, eventID)
updateDBMS()
enrollmentConfirmation
enrollmentConfirmation
outcome := enrollmentConfirmation
showPopup(outcome)


3rd Party Track4Run Sequence Diagram
3rdParty
WebInterface
3rdPartyRequestHandler
EventHandler
MapViewer
DataHandler
clickTrack4RunButton()
sendTrack4RunRequest()
forwardTrack4RunRequest()
getPartyEventsList()
queryDBMS()
partyEventsList
partyEventsList
partyEventsList
displayTrack4RunPage(partyEventslist)
clickCreateNewEventButton()
displayCreateNewEventForm()
boolean isLooping := TRUE
loop
[isLooping == TRUE]
fillUpForm(eventData)
sendCreateNewEventRequest(eventData)
forwardCreateNewEventRequest(eventData)
provideMapToParty(partyID)
callAPI()
sendMapToPartyRequest(partyID)
forwardMapToParty(partyID)
displayMap()
selectEventPathOnMap(eventPath)
eventPath
eventPath
eventPath
checkEventValidity(eventData, eventPath)
alt
[Invalid Event]
InvalidEventException
InvalidEventException
showPopup(InvalidEventException)
[Valid Event]
eventConfirmation
eventConfirmation
showPopup(eventConfirmation)
isLooping := FALSE


3rd Party View User Data Sequence Diagram
3rdParty
WebInterface
3rdPartyRequestHandler
DataPresentation
DataHandler
clickUsersListButton()
sendUsersListRequest(partyID)
forwardUsersListRequest(partyID)
queryDBMS()
usersList
usersList
displayUsersListPage(usersList)
clickViewUserDataButton()
sendViewUserDataRequest(userID)
forwardViewUserDataRequest(userID)
getUserData(userID)
queryDBMS()
userRawData
dataProcessingAndPlotting(userRawData)
userProcessedData
userProcessedData
displayViewUserDataPage(userProcessedData)
clickDownloadButton()
downloadData(userProcessedData)


3rd Party Add User Sequence Diagram
3rdParty
WebInterface
3rdPartyRequestHandler
3rdPartyAuthorizationManager
DataHandler
UserRequestHandler
ApplicationInterface
User
clickUsersListButton()
sendUsersListRequest(partyID)
forwardUsersListRequest(partyID)
queryDBMS()
usersList
usersList
displayUsersListPage(usersList)
clickAddUserButton()
displayAddUserForm()
boolean isLooping := TRUE
loop
[isLooping == TRUE]
fillUpForm(fiscalCode)
sendAddUserRequest(fiscalCode, partyID)
forwardAddUserRequest(fiscalCode, partyID)
checkUserRegisteredToParty(fiscalCode)
queryDBMS()
alt
[FiscalCode is associated with none of our registered users]
UnexistingFiscalCodeException
UnexistingFiscalCodeException
UnexistingFiscalCodeException
showPopup(UnexistingFiscalCodeException)
[user is already linked to 3rdParty]
AlreadyRegisteredFiscalCodeException
AlreadyRegisteredFiscalCodeException
AlreadyRegisteredFiscalCodeException
showPopup(AlreadyRegisteredFiscalCodeException)
[user isn't linked to 3rdParty, so the request is sent to the user]
userID
sendAuthorizationRequestToUser(userID, partyName)
forwardAuthorizationRequestToUser(userID, partyName)
showAuthorizationRequestNotification(partyName)
alt
[rejectRequest]
clickRejectButton()
rejectRequest
rejectRequest
getThirdPartyUsersList(partyID)
queryDBMS()
[acceptRequest]
clickAcceptButton()
acceptRequest
acceptRequest
addUserToPartyList(userID, partyID)
updateDBMS()
updatedUsersList
updatedUsersList
updatedUsersList
displayUsersList(updatedUsersList)
isLooping := FALSE

3rd Party Group Query Sequence Diagram



2.5 Component Interfaces

In this section are presented the Interfaces with their own methods:

Figure 2.5.1: Component Interfaces Diagram

2.6 Selected Architectural Styles and Patterns

2.6.1 Overall Architecture

As expressed in the RASD document, the availability of our system must be a primary concern considering the critical sector in which some of our services operate. We think that will be a key to success to design the architecture with in mind some principles of a distributed systems:

- The services we offer are quite different from one another as for time consume and computational power, so it helps to decouple functionalities and seeing it as a Service-Oriented Architecture (SOA). Database writes will almost always be slower than reads and these last ones can also be asynchronous, so for this reason it will be convenient to design a system that categorizes the request type and scale each service independently.
- In order to handle failure gracefully a web architecture must have some form of redundancy of its logic and data. As anticipated, implementing our application logic containerized in a SN architecture helps a lot: new nodes can be added without special conditions so there won't be a single point of failure in the system and it will scale horizontally. The best way to handle this is via the load balancer: the main purpose is to manage a lot of simultaneous connections and route those connections to one of the request nodes, allowing the system to scale to service more requests by just adding nodes.
- Confidentiality and integrity are guaranteed by the DBMS in the persistence layer of the architecture. Another architectural decision that can this time enhance performance and reliability is to handle the writing on the Database using queues: in this way when a client submits task requests to a queue, he/she is no longer forced to wait for the results working asynchronously, instead he/she need only acknowledgement that the request was properly received.

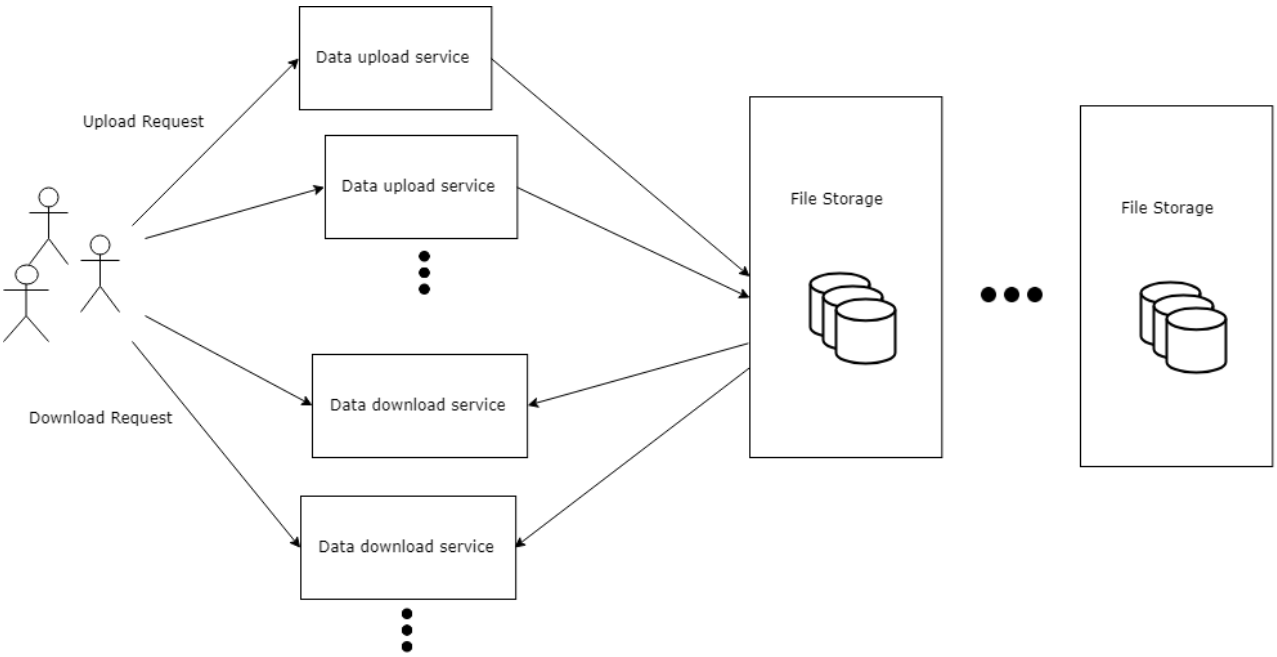


Figure 2.6.1.1: service splitting and redundancy

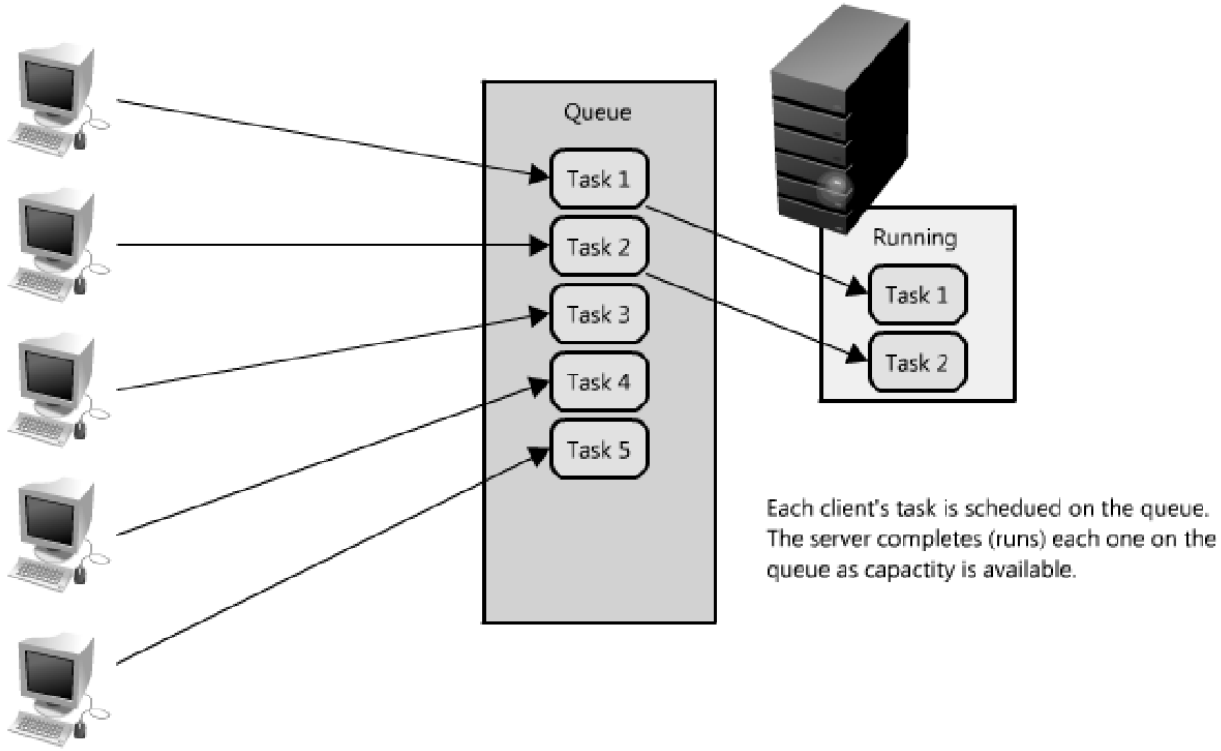
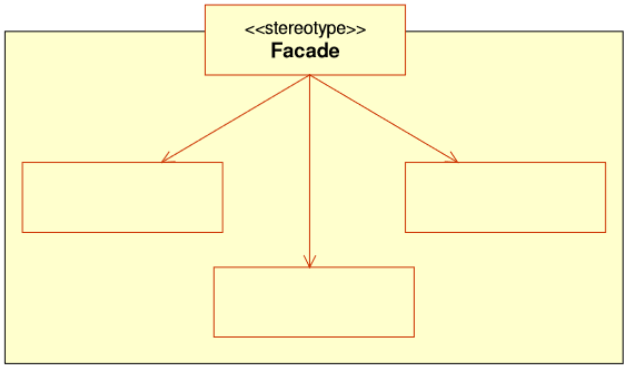


Figure 2.6.1.2: using queues to handle requests

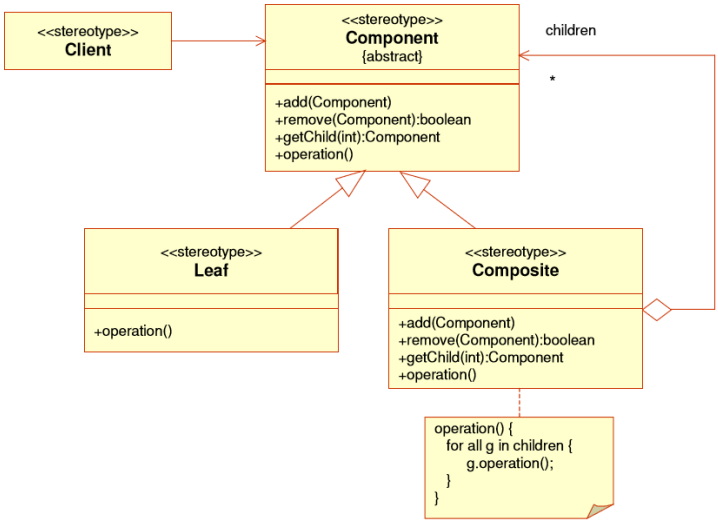
2.6.2 Design Patterns

Recommended Structural patterns:

- ***Façade pattern*** - It supplies a unified interface to a subsystem set of interfaces, making easier to reach and modified them. It delegates to the appropriate objects of the subsystem each request received from the "outside". In our component view its use is evident when we tackle *RequestHandler* and its components.

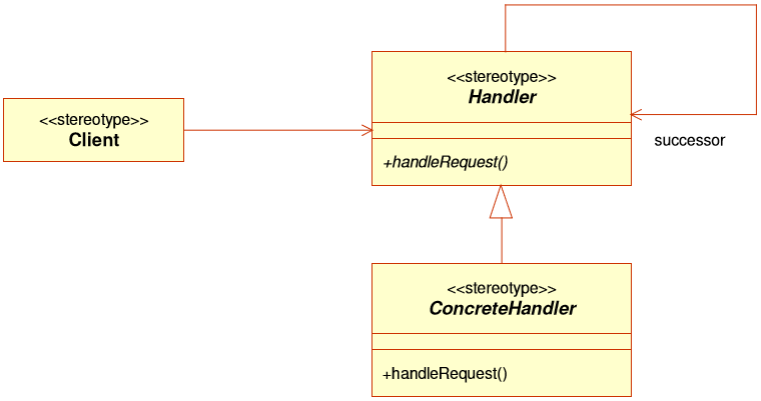


- ***Composite pattern*** - It allows the building of hierarchies of composite objects: aggregation of single objects (Leaf) or other composite ones (Composite). It is particularly useful in the implementation of a GUI as we do in *ApplicationInterface* and *WebInterface* and is the MVC design pattern recommended for the View.

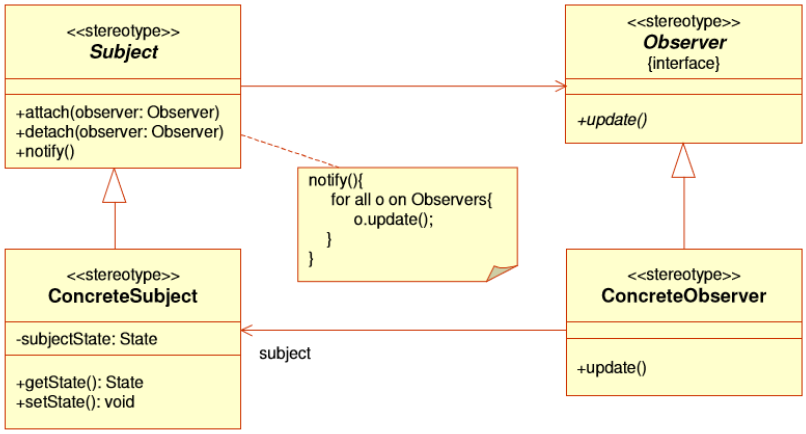


Recommended Behavioural patterns:

- ***Chain of Responsibilities pattern -*** It separates the sender of a request by the recipient, in order to allow more than one object to handle it. The target objects are put in a chain, and the request is transmitted in this chain until it finds one that manages it. We warmly advise to use this pattern whenever is possible in the system because it makes easy to extend components and add services without breaking the code.



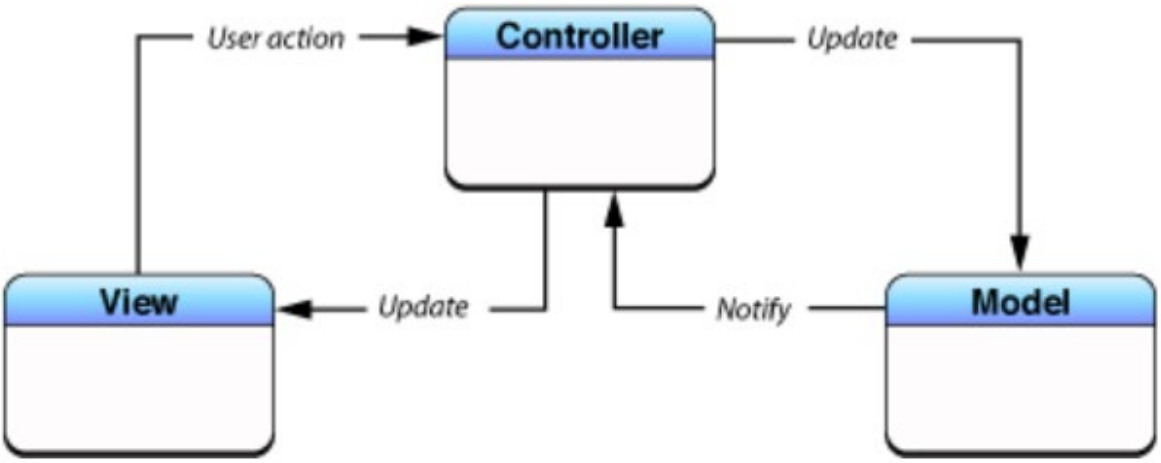
- ***Observer pattern*** – It allows the definition of dependency's associations of many objects to one, so that if the latter changes its status, all the others are notified and updated automatically. This pattern assigns to the monitored object (Subject) the task of recording a reference to the other objects (ConcreteObservers) to be alerted of the Subject's events. Then it notifies them through the invocation of their method, present in the interface that they must implement (Observer). These concepts could be a great help, for example, inside the *ApplicationInterface* component.



Recommended Architectural patterns:

- ***Model-View-Controller*:** It is recognized as an industry standard for web development and it is a compound design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. It is based on 3 fundamental roles:
    - The Model encapsulates the basic behaviours, the correlation between data and exposes functionalities.
    - The Controller defines the application behaviour, maps actions to state updates and deliver the changes to the view.
    - The View renders the model and interprets the user interactions, via the graphical interface.

  This pattern is quite old and there exists a lot of variations of it, but the most suitable for our architecture seems to be the one defined by Apple for its object-oriented languages: the controller acts as a mediator and it oversees updating the view too. It underlines a more recognizable separation of concerns with the aim to reduce high level of coupling and so the misunderstandings between developers.



- **Data Access Object:** the use of *DataHandler* Interface permits us to access our Database through the DBMS; this can be achieved using this pattern, which separates the application in two important parts (the Model and the View of the MVC pattern) that are isolated from each other, guaranteeing also persistency.

- **Data Transfer Object**: a data transfer object (DTO) is an object that carries data between processes. The use of this pattern, together with the DAO, it's realized always thought the use of *DataHandler* Interface.

SECTION 3: User Interface Design

We already defined quite specifically the user interface critic components in the RASD document, here we add only some mock-ups in order to avoid misunderstanding between different development teams.

In the mobile app, from the profile page it is possible to reach the paired devices one, when it is represented what gadget has been connected at least one time and which is currently synchronised. The user can then delete it clicking on the trash icon o add a new device with the button (figure 3.1).

Figure 3.1: Paired devices page

In figure 3.2 instead it is visible how the list of run events is displayed and how it is composed the specific planned run page, where are exploited the event information and the button to register for the run.

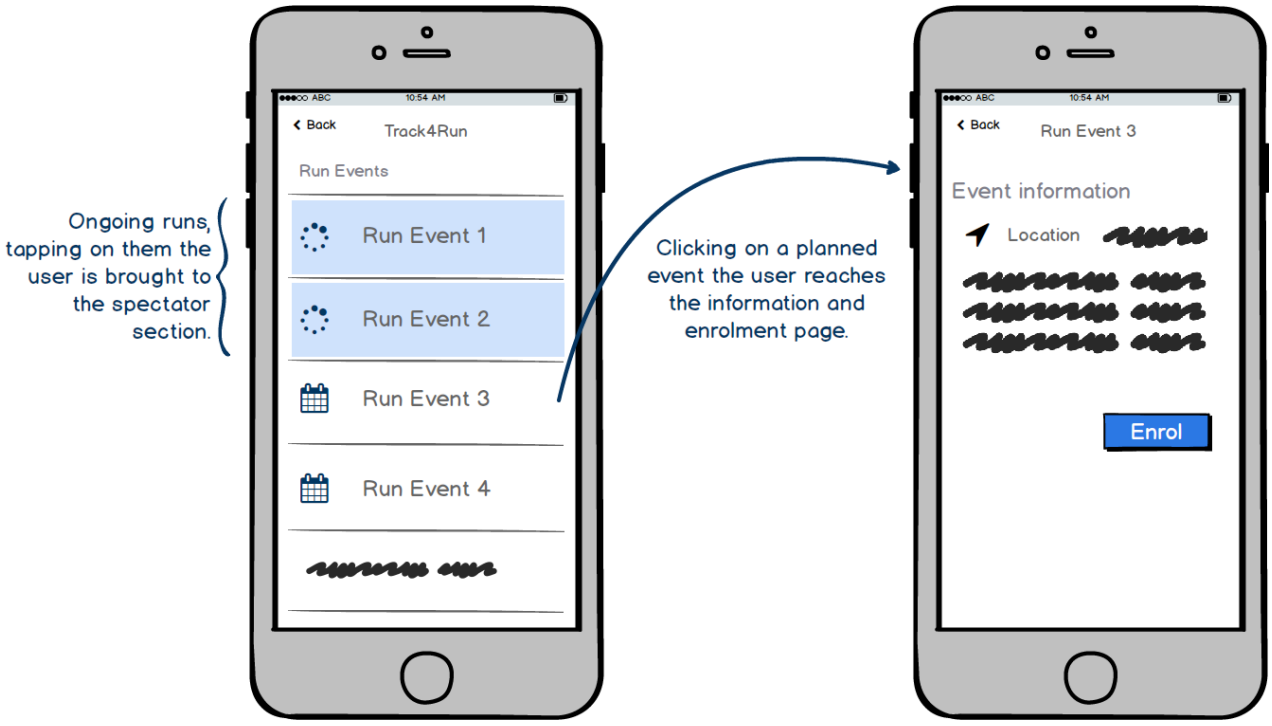


Figure 3.2: Track4Run home and interactions

For the web interface we think that could be of great help to have a sample (figure 3.3) of how the “specific user data” page can be displayed:

- Divided per type of visualization (d/m/y/all).
- In a spreadsheet like form, with the option to order by one of the column values.
- A download button that provide an xml (or similar) version of the table.

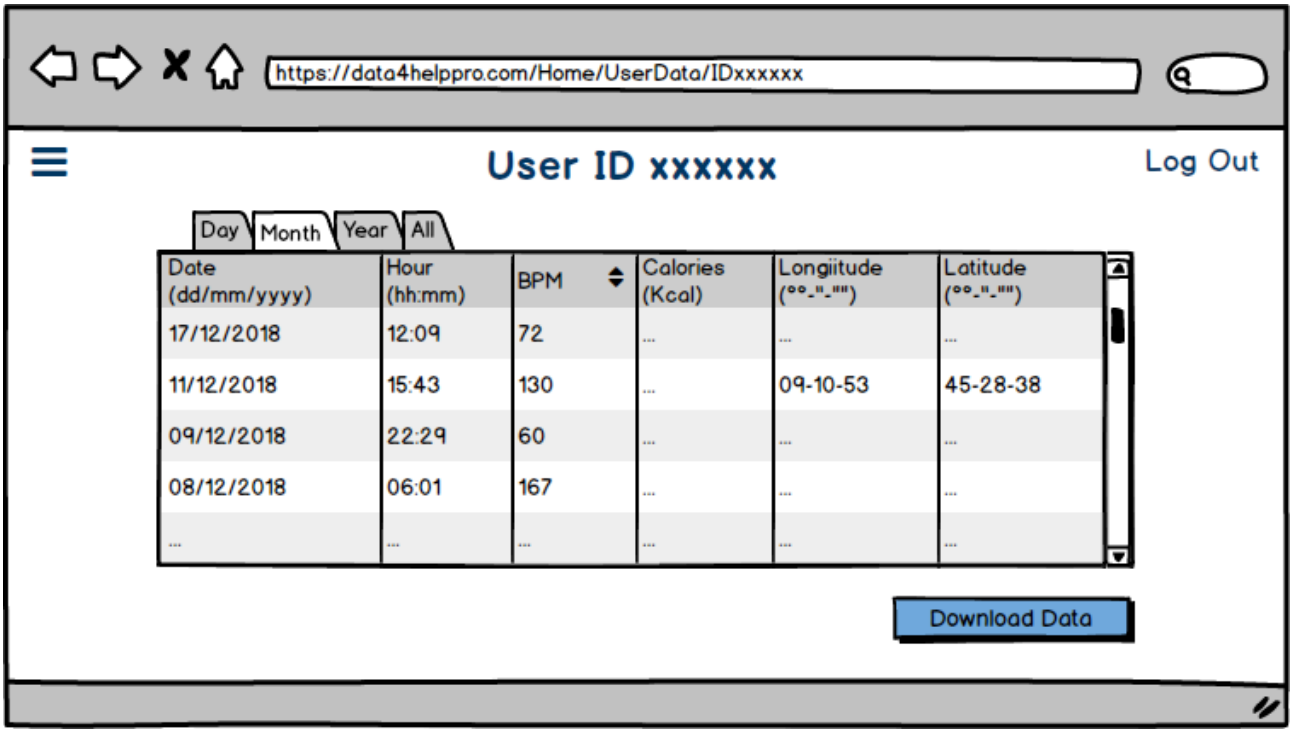


Figure 3.3: specific user data page

SECTION 4: Requirements Traceability

The following table describes how Goals are satisfied with the Requirements and which are the Components that are involved in the satisfaction process:

GOAL	REQUIREMENTS	COMPONENTS
G1	**R1, R2, R3, R4, R5**	ApplicationInterface UserRequestHandler AuthenticationHandler ProfileHandler DataHandler
G2	**R6, R7, R8, R9**	SmartDevice ApplicationInterface UserRequestHandler DataPresentation DataHandler
G3	**R10, R11, R12, R13**	ApplicationInterface UserRequestHandler 3rdPartyAuthorizationManager DataHandler
G4	**R14, R15, R16, R17, R18, R19**	WebInterface 3rdPartyRequestHandler AuthenticationHandler ProfileHandler DataHandler
G5	**R20, R21, R22, R23**	WebInterface 3rdPartyRequestHandler QueryHandler DataHandler
G6	**R24, R25, R26**	WebInterface 3rdPartyRequestHandler QueryHandler DataHandler
G7	**R27, R28, R29**	DataThresholdChecker AmbulanceDispatcherRequestHandler DataHandler
G8, G8.1	**R30, R31**	WebInterface 3rdPartyRequestHandler MapViewer EventHandler DataHandler
G8.2	**R32**	ApplicationInterface UserRequestHandler EventHandler DataHandler
G8.3	**R33**	ApplicationInterface UserRequestHandler EventHandler MapViewer DataHandler

SECTION 5: Implementation, Integration and Test Plan

5.1 Implementation plan

Our modular architecture allows us to parallelize the development of subsystems and there are no constrains on implementation order for the base services of Data4Help. It's different for the ones that are built on top of the core foundation: they will be addressed after implemented and tested this last one. For us the best strategy should be to start with the basic services we must offer and then integrate them, in order of priority:

- ***Data4HelpManager:*** module is crucial for many services related to data handling.
- ***Account:*** module for authentication and personal information management.
- ***RequestHandler:*** module acting as mediator for the request between clients and server.
- The different client interfaces (ApplicationInterface and WebInterface).

Then, having in mind the integration strategy further defined, will be implemented the ***Track4RunManager*** and the ***AutomatedSOSManager*** modules.

5.2 Integration Plan

The pre-requisites of the integration phase are that before any components of two different subsystem, or a component and an external service, are connected they must have been implemented using interface mocks or signatures and tested following the testing strategy.
In this way the system's solidity will be less affected by the integration as much as the order in which it will be performed, that can suit the implementation and testing strategy and be incremental.
Here we exploit the dependencies between components in a suggested order for the application logic, or model in MVC paradigm:

- The internal interfaces between ***Data4HelpManager*** subsystem components.
- The internal interfaces between ***Account*** subsystem components.
- The *DataHandler* component with the ***DBMS*** interface, as the only exit point towards the persistence layer.
- The ***Account*** subsystem with the ***Data4HelpManager*** to complete the core logic.

At this point it is possible to proceed to integrate the controller and the view or to continue the model development:

- If ***AutomatedSOSManager*** is chosen, the order should be:
    - The internal interfaces.
    - The *AmbulanceDispatcherHandler* component and the respective external resource interface.
    - The subsystem interface to the *DataHandler* component.

If ***Track4RunManager*** is chosen, the order should be:

- The internal interfaces.
- The *MapViewer* component and the *MapsAPI* interface.
- The subsystem interface to the *DataHandler* provided one.

At the end of every subsystem integration and testing is possible to proceed, as written before, to connect the ***RequestHandler*** interface to them and to the client's GUI subsystems provided ones.

5.3 Test Plan

We recommend a mix of top-down and bottom-up approach:

- BU: we inspect the functionalities offered by a new component.
- TD: we inspect the route of the request through the sub-system.

The TD-tests are those that run on the design: their purpose is not to meet functional verification requirements, but to establish that the basic functionality of the design operates correctly before performing more exhaustive tests.

To test functionalities in a unit we will use mocks, mainly for external services as maps and data in order to begin the testing phase from the beginning of development.

As the basic model is tested, we will start to test the in integration between the request from the UI all the way to the application logic, using an integration testing framework for containers, where our system runs.

To find the most effective test cases we suggest then to use a Model-Based Testing approach: transition and state coverage on the base of state diagrams defined in the RASD for user and 3rd party interactions. As visible in figures 5.3.1/5.3.2, we re-display the state diagrams with changed colours to underline the diverse subsystems:

- Orange for ***Data4HelpManager*** subsystem.
- Azure for ***Account*** subsystem.
- Pink for ***Track4RunManager*** subsystem.

Different is for ***AutomatedSOSManager*** components because it is a service running in "background", with small interactions with third actors, so it will need a cooperative approach with ambulance dispatchers and healthcare experts.

As last step we can start to test non-functional requirements, like performance, at various level of the architecture:

- Load tests.
- Availability through MTBF and MTTF.
- Security properties for example via dependencies or SQL injections.

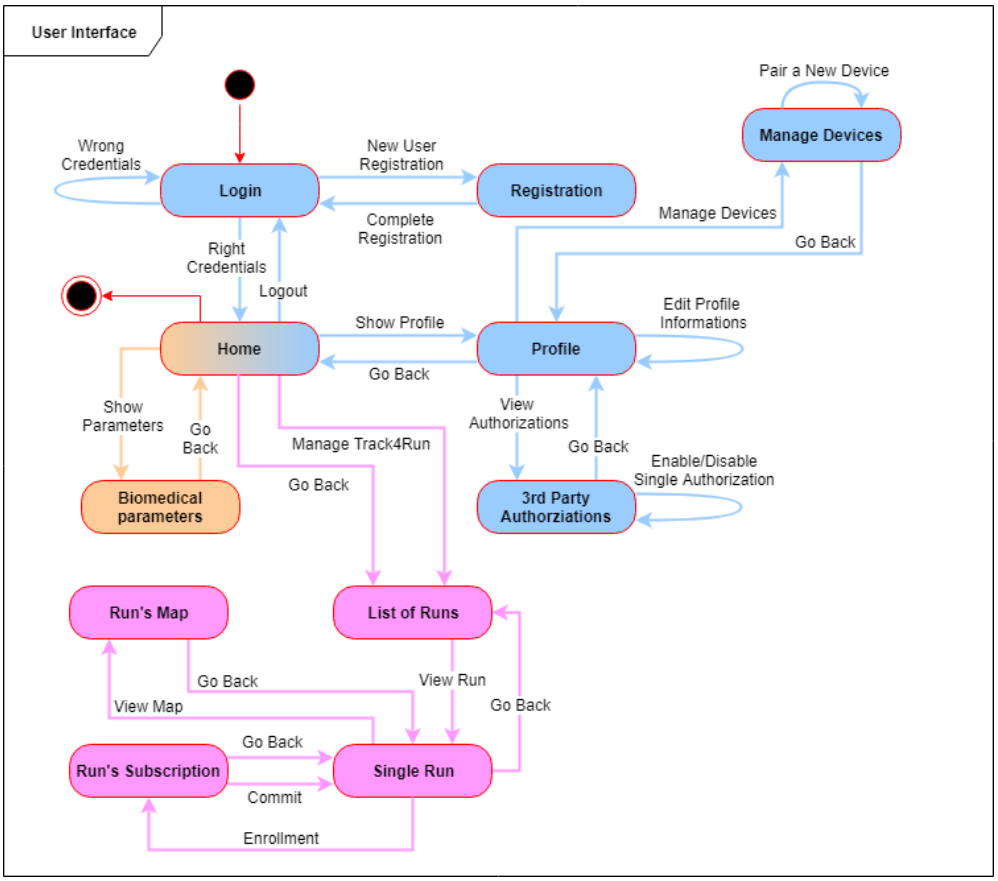


Figure 5.3.1: User interface state

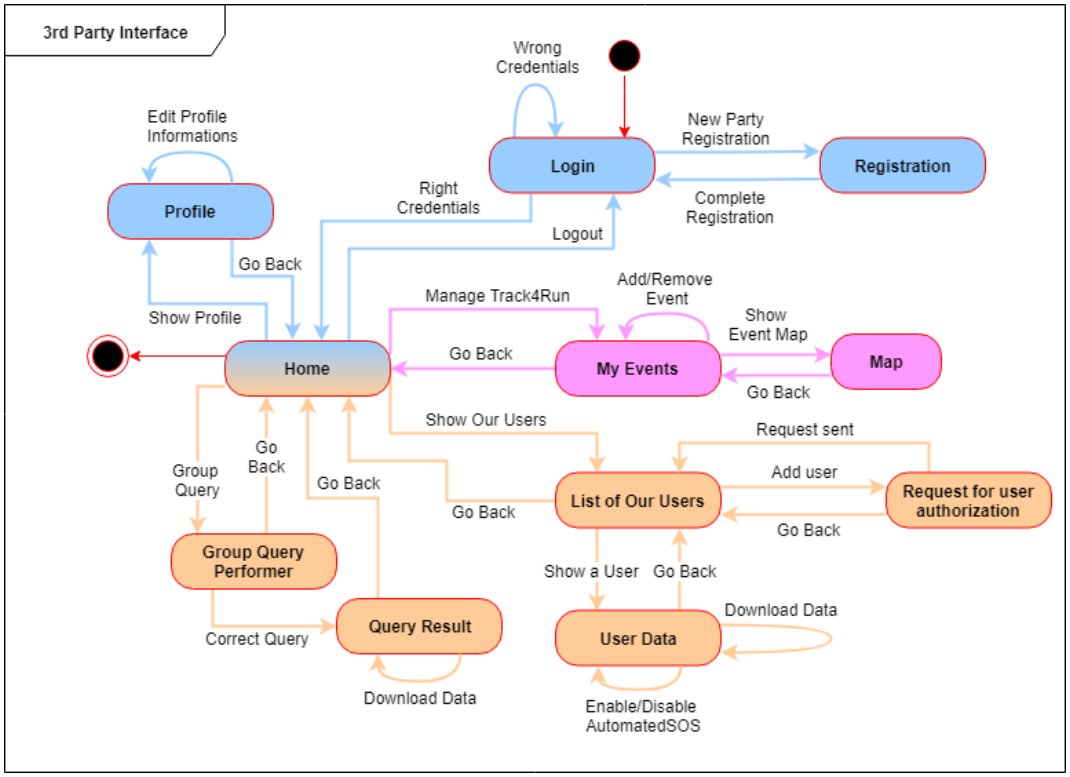


Figure 5.3.2: 3rd party interface state diagram

SECTION 6: Effort Spent

Tools Used

The tools used for the realization of this document are the following:

- **Microsoft Word**: text editor with Live Collaboration feature;
- **Microsoft OneDrive**: cloud space for synchronizing files between us;
- **Draw.io**: all diagrams (except SequenceDiagrams);
- **SequenceDiagram.com:** tool for drawing SequenceDiagrams;
- **BalsamiqMockups3:** tool for realizing mockups.

NB: we haven't done GitHub commits because we preferred keeping the document on our shared OneDrive in order to edit it live between different

Hours Spent

The time we spent is equally divided between us:

- Paolo Saccani: 35 hours;
- Diego Riva: 35 hours;
- Matteo Rubiu: 35 hours.