POLITECNICO DI MILANO

SOFTWARE ENGINEERING 2 COURSE

The TrackMe project

Design Document

Kevin Mato, Antonio Mazzeo

Github: https://github.com/antoniomazzeo/MatoMazzeo

10/12/2018

Summary

1 Introduction

1.1 Purpose

The purpose of this document is to give more technical details than the RASD about the TrackMe system. While the RASD presented a general view of the system and what functions the system is supposed to execute, this document aims to present the implementation of the system including components, run-time processes and deployment. It also presents in more details the implementation and integration plan, as well as the testing plan.

1.2 Scope

The extent of the descriptions in this document are listed in the purpose section,

More precisely, the document presents:

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

The purpose of this document is to provide an overall guidance to the architecture of the software product.

For further details about what the system must be able to do see the RASD document.

1.3 Definitions, Acronyms, Abbreviations

Definitions

- Reverse proxy: a reverse proxy server retrieves resources on behalf of a client from one or more servers
- Message broker: an intermediary platform when it comes to processing communication between two applications

Acronyms

- RASD: Requirements Analysis and Specification Document
- DD: Design Document
- DB: Database
- REST: Representational State Transfer
- JWT: JSON Web Tokens
- CI/CD: Continuous integration and Continuous delivery

- DBMS: Database Management System
- CQRS: Command Query Responsibility Segregation
- DNS: Domain Name System
- OS: Operating System
- HTML: Hypertext Markup Language
- CSS: Cascading Style Sheets
- JS: JavaScript
- JSON: JavaScript Object Notation
- API: Application Programming Environment
- HTTP: Hypertext Transfer Protocol
- HTTPS: Hypertext Transfer Protocol Secure
- TCP: Transmission Control Protocol

Abbreviations

- [Gn]: n-th goal
- [Rn]: n-th functional requirement
- [Dn]: n-th domain assumption

1.4 Revision History

Version, date and summary

Version	Date	Summary
1.0.0	10/12/2018	First Release

1.5 Reference Documents

The following documents were used:

- The document of the assignment: https://beep.metid.polimi.it/documents/121843524/3f744351-7378-4162-86b0-45eddaf10713
- IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications.
- IEEE Std 1016tm-2009 Standard for Information Technology-System Design-Software Design Description.

1.6 Document Structure

The document is divided in the following sections:

- **Section 1:** General introduction of the design document. It contains the purpose and the scope of the document, as well as some abbreviation in order to provide a better understanding of the document to the reader.
- **Section 2:** this section describes the platform’s architecture structure and provides detailed information about component interaction.
- **Section 3:** here it’s listed the mapping between the requirements in the RASD and the components defined in this document.
- **Section 4:** presents the way in which the implementation and the integration should be done and the aspects to focus the testing part on.

2. Architectural Design

2.1 Overview

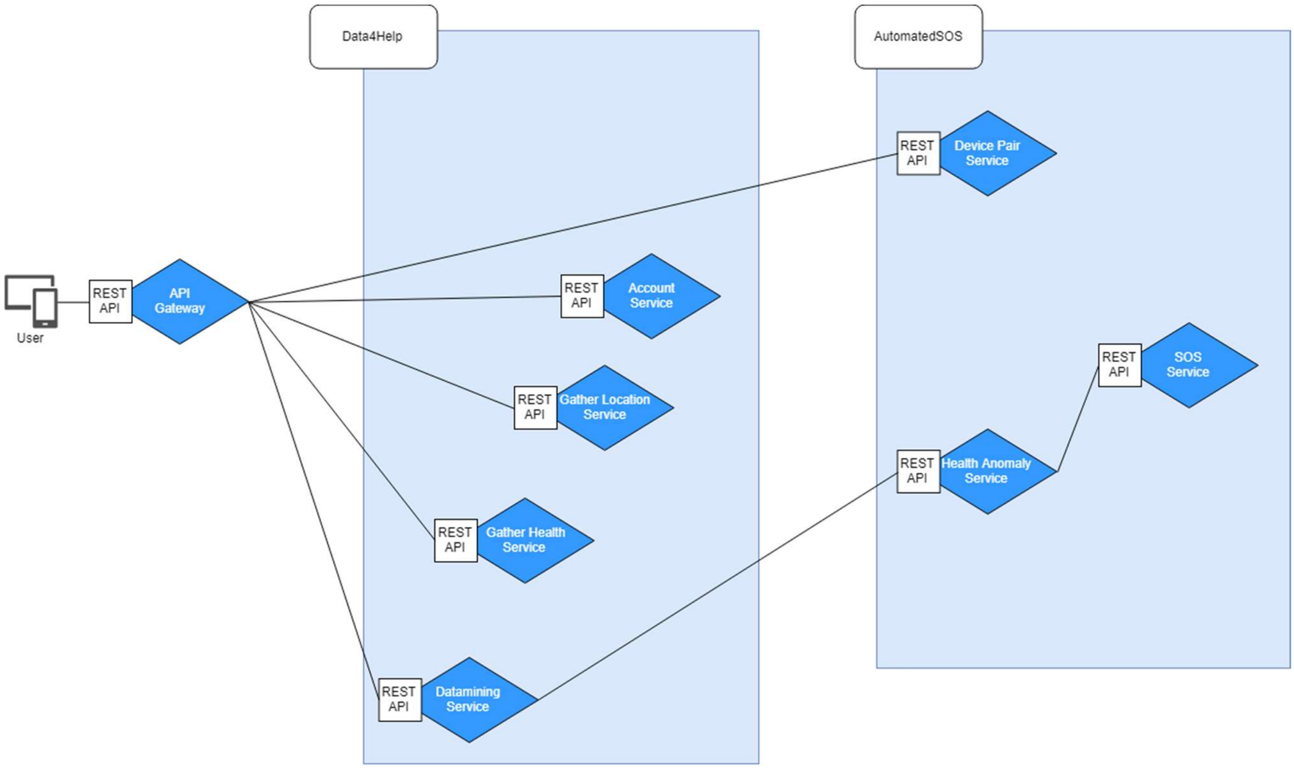

Figure 1 System Overview

The whole system is composed of multiple microservices able to communicate one to another as described in the picture above, every one of them shows a REST interface in order to keep invisible the actual service location (i.e. The client makes no difference between services of Data4Help and AutomatedSOS).

The API Gateway behaves as a reverse proxy between the Client and the TrackMe backend (Data4Help + AutomatedSOS), where the entry point is responsible for the 1:N mapping between the REST API exposed externally and the ones offered internally by each Microservice, in other words it’s responsible to forward and route the request to the internal system, reaching the appropriate microservice for that request. Each microservice can perform a specific task:

- Gather Location Service: Acquires and stores location data from the user.
- Account Service: Acquires and stores account information (data provided during the sign-up process from the user).
- Gather Health Service: Acquires and stores health data from the user.
- Health Anomaly Service: Manages the detection of possible anomalies on health data and notifies SOS Service in case of emergencies.
- Datamining Service: Acquires, processes and analyse all the data managed from the system; this service is also responsible for the detection of anomalies. The results will respond to third parties’ requests, and will ensure the smooth functioning of AutomatedSOS.

- SOS Service: Contacts the emergency office in case of problems (triggered by Health Anomaly Service).
- Device Pair Service: This service is responsible of the smart watch/band connection to the application.

This microservices division provides us with many advantages, such as:

- High level of modularity, which helps the developers to follow the CI/CD practices in an intuitive and fast manner.
- Possibility to develop service independently, with different deployment rates and the ability to include in each service the most updated technologies and framework without the need to take the whole system down.
- Improved scalability control, to satisfy capacity and availability constraints in an easy and immediate fashion.
- High supervising level over the Hardware used to best match the resource requirements for each service.

The data storage system is distributed across the multiple services, where this encapsulation ensures that the microservices are loosely coupled and can evolve independently one of another.

The data partitioning is splitted as follows:

Data4Help:

- Account related informations are stored in the Account Service.
- Health related informations are stored in the Gather Health Service.
- Location related informations are stored in the Gather Location Service.

AutomatedSOS:

- Paired devices informations are stored in the Pair Device Service.

Other needed data gets retrieved from Data4Help through REST interfaces, for example the SOS Service needs Location, Health and Account data.

In order to maintain data consistency across different services, data management will be implemented in an event-driven architecture through the **event sourcing pattern**. Every service publishes an event as a part of an atomic operation, which is concluded after all the interested services subscribed finally consume that event. **CQRS** is used to implement the queries: as a result, the consistency is eventually guaranteed. Eventual consistency must be considered by the developers for a proper implementation.

Docker is used as the packaging and deployment tool needed to develop the Microservices structure.

Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called *containerization*.

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.

- Stackable: You can stack services vertically and on-the-fly.

Every component shown in the Component diagram (section 2.2) will be deployed in a single ad-hoc container, built on a container image: a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it (code, runtime, system tools, system libraries, settings).

Therefore, we have different containers to manage. In order to deploy, manage and scale this containerized application, Google Container engine will be used, powered by the **Kubernetes** system.

Kubernetes orchestrates the containers using different levels of abstraction and provides us with a set of development and maintenance features, such as:

- Automatic horizontal scaling : this lets the system grow without having to manage manually hardware and software changes.
- Service discovery and load balancing : Kubernetes gives containers their own IP addresses and a single DNS name for a set of containers and provides a load-balancing system that addresses both the external requests and the internal calls to the various services.
- Automated rollouts and rollbacks : Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.
- Smart configuration management: deploy and update application configurations without the need to rebuild docker images.
- Self-healing and automatic logging : with built-in integration of logging tools and monitoring systems, it is possible to manage containers accordingly to health-checks feedback and to have an insight of how the application is running, providing a stable service able to heal itself in order to run 24/7.

2.2 Component view

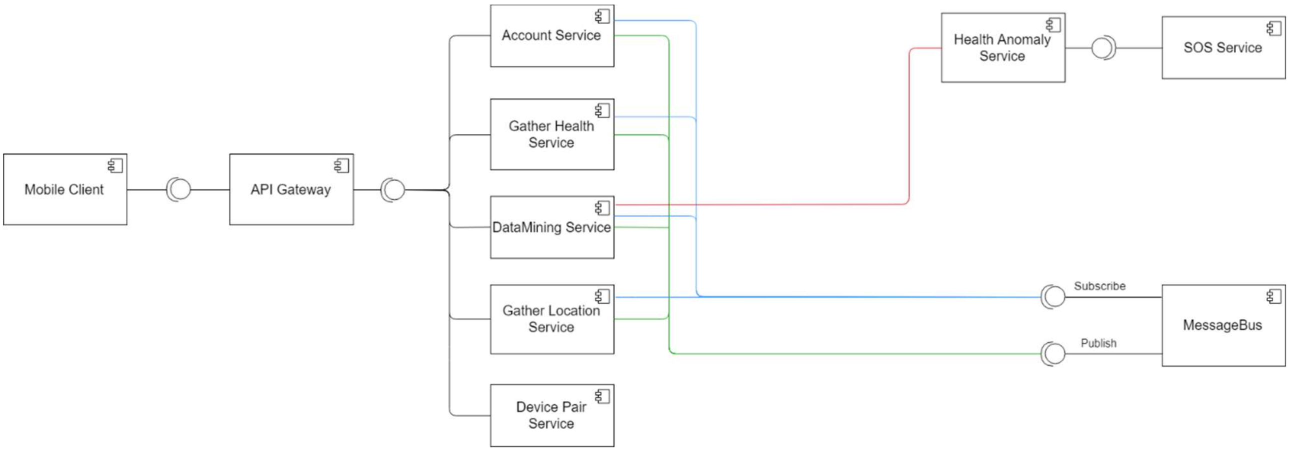

Figure 2 Component View

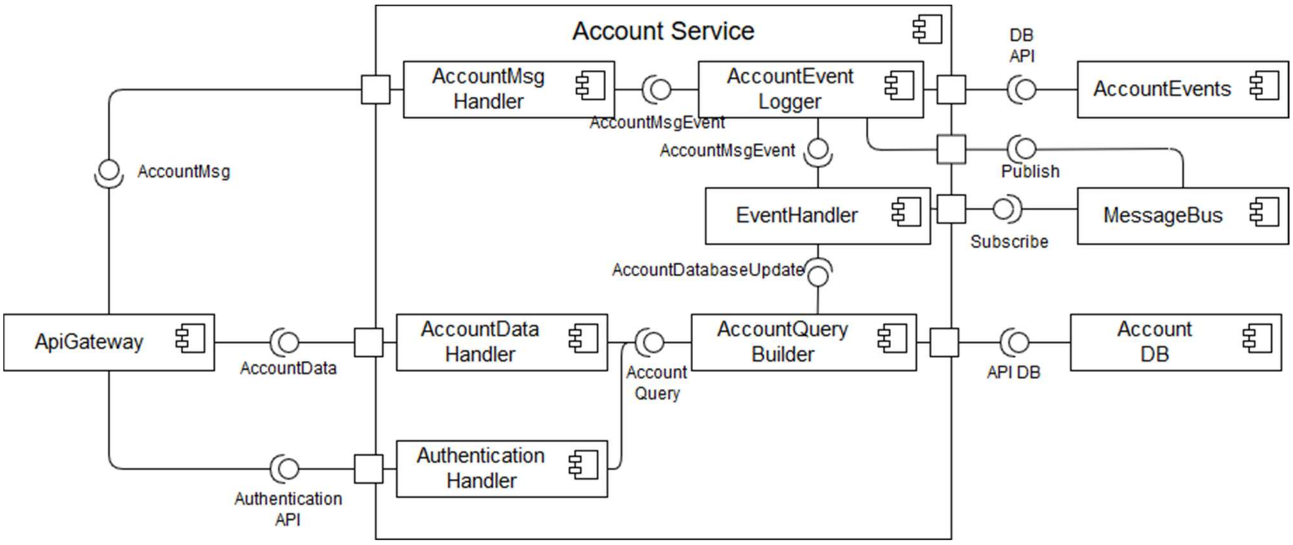

Figure 3 Account Service Component

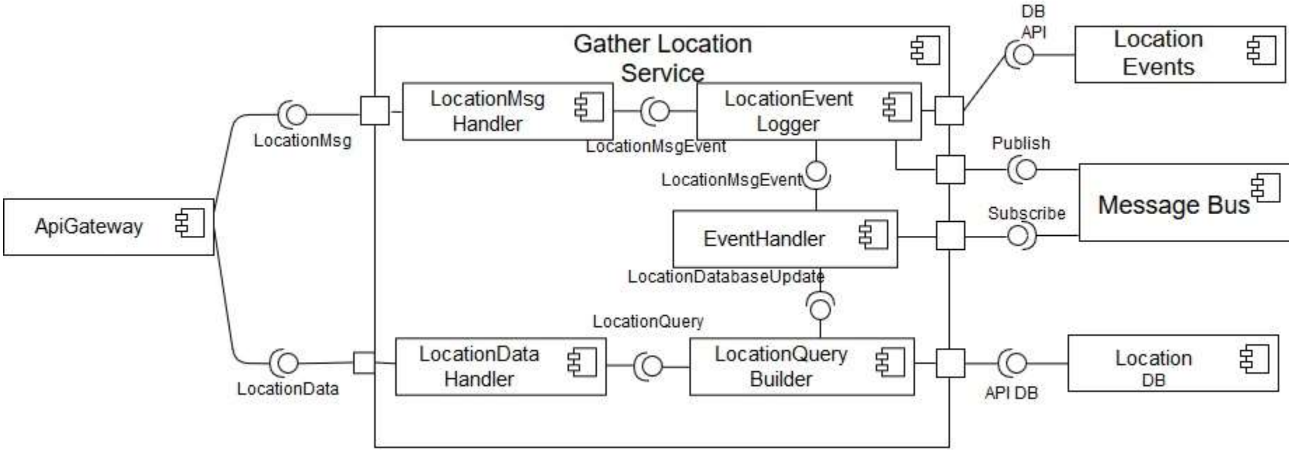

Figure 4 Gather Location Service

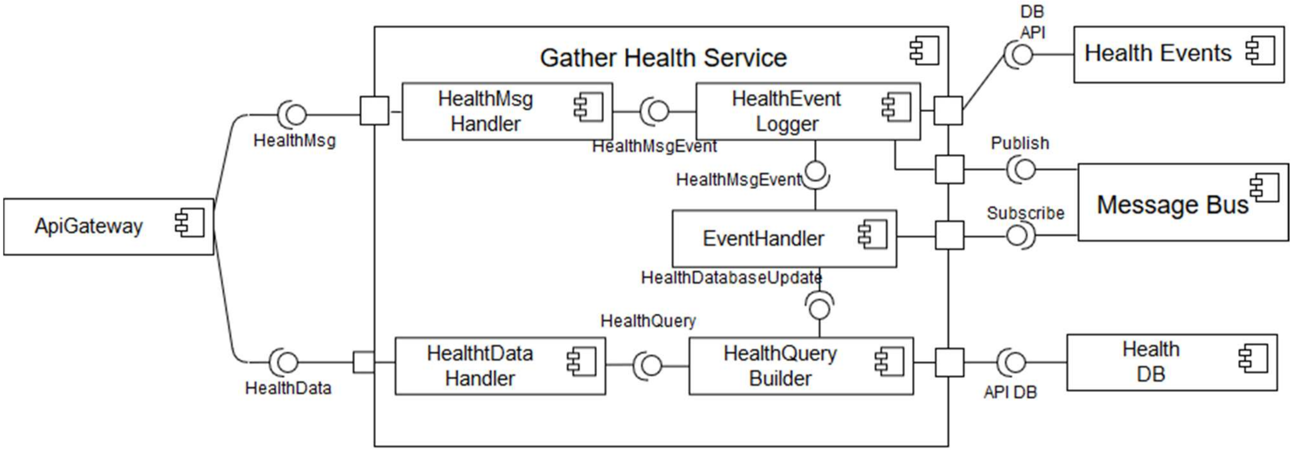

Figure 5 Gather Health Service

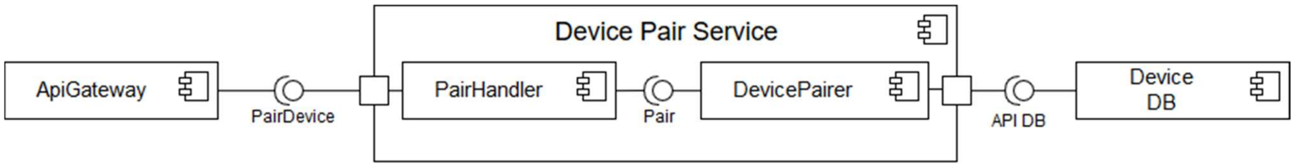

Figure 6 Device Pair

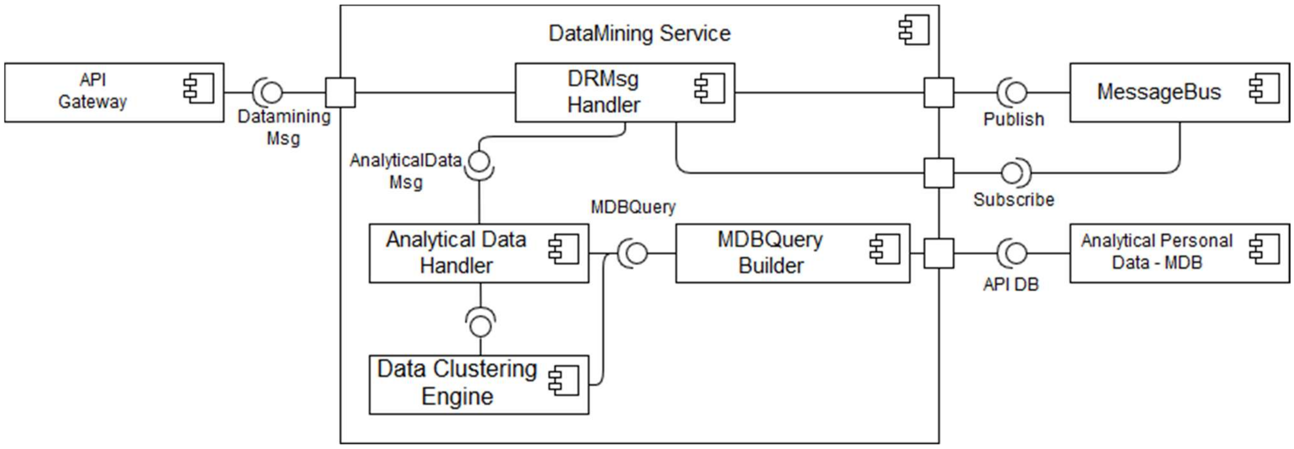

Figure 7 Datamining Service

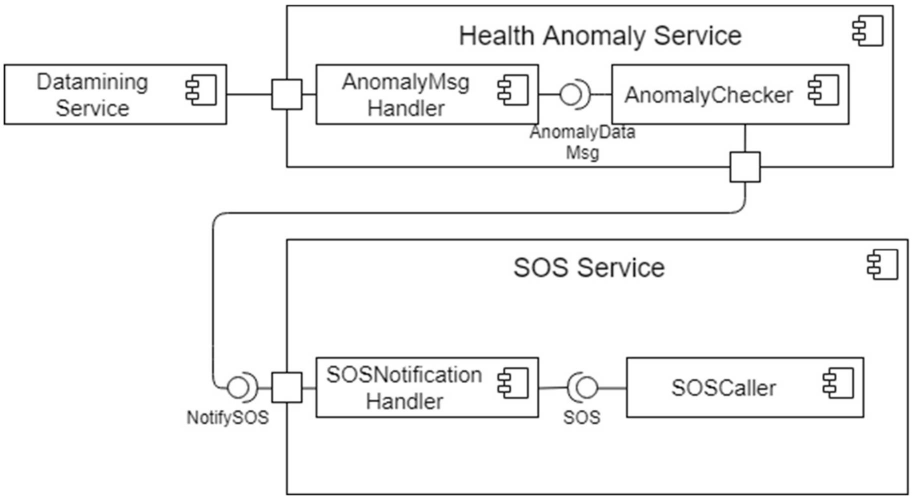

Figure 8 SOS Component

2.3 Deployment view

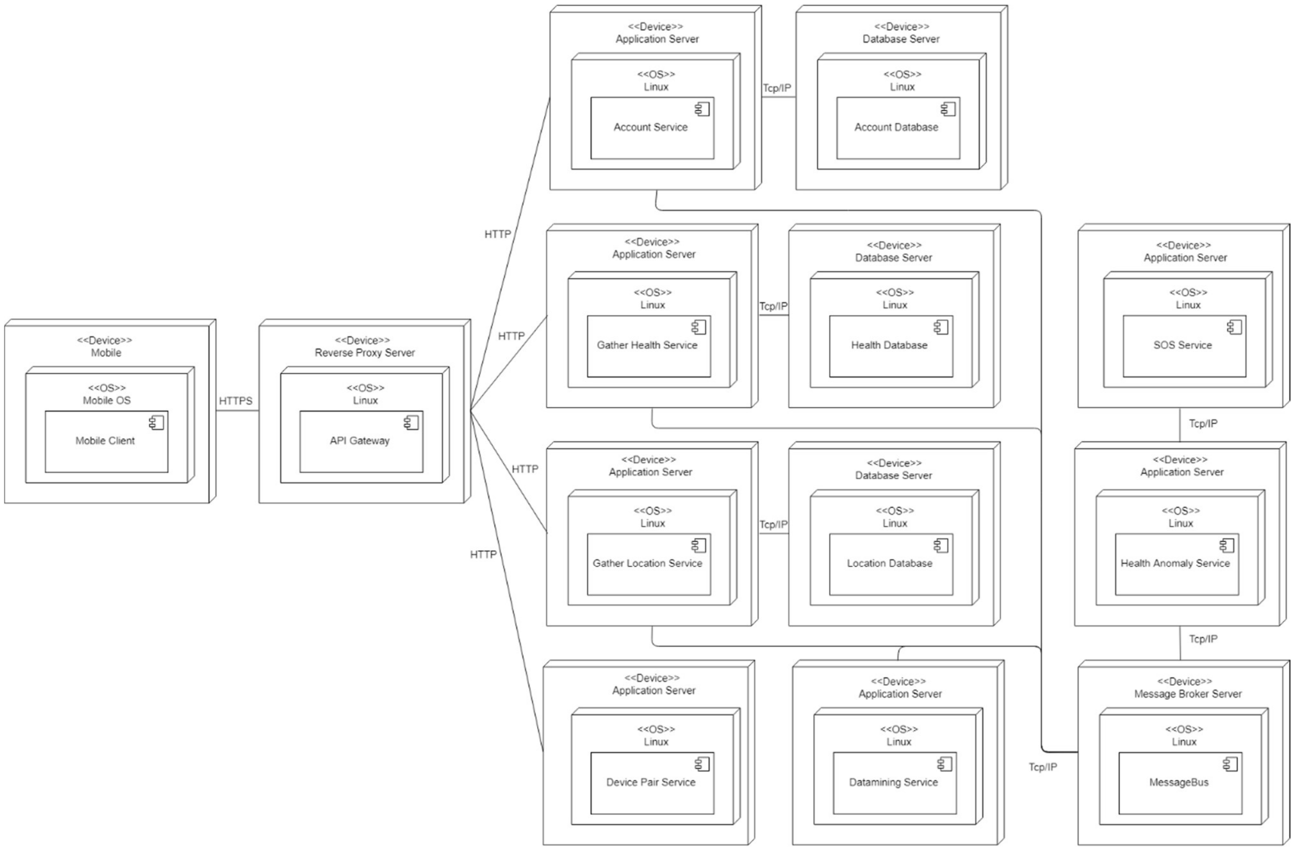

Figure 9 Deployment view

2.4 Component Interfaces

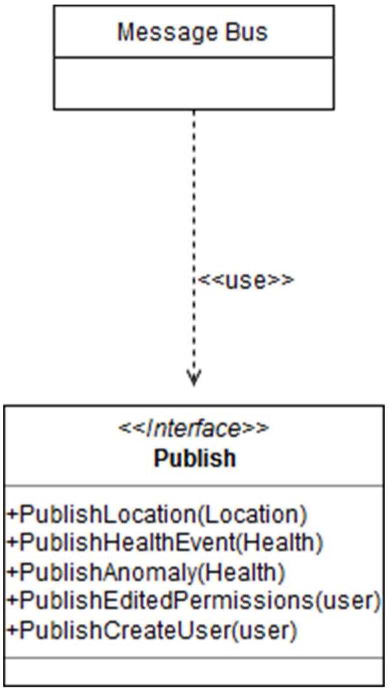

Figure 11 Message Bus Interface

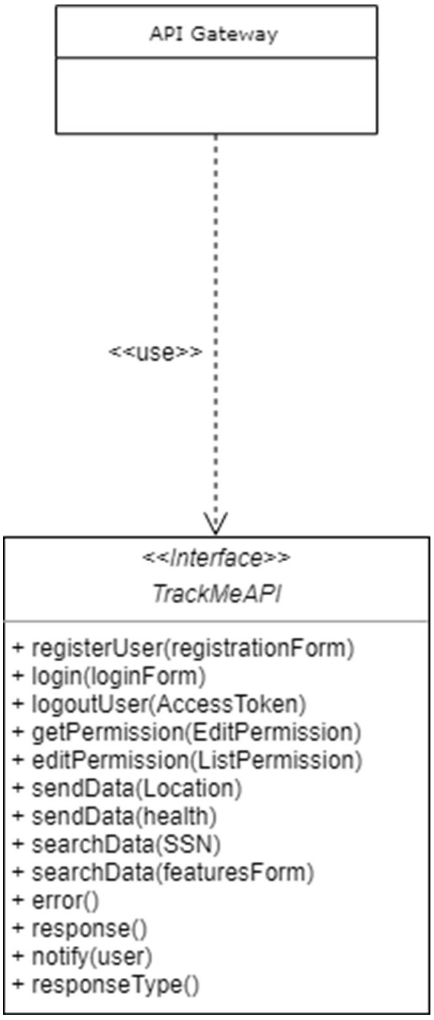

Figure 10 API Gateway Interface

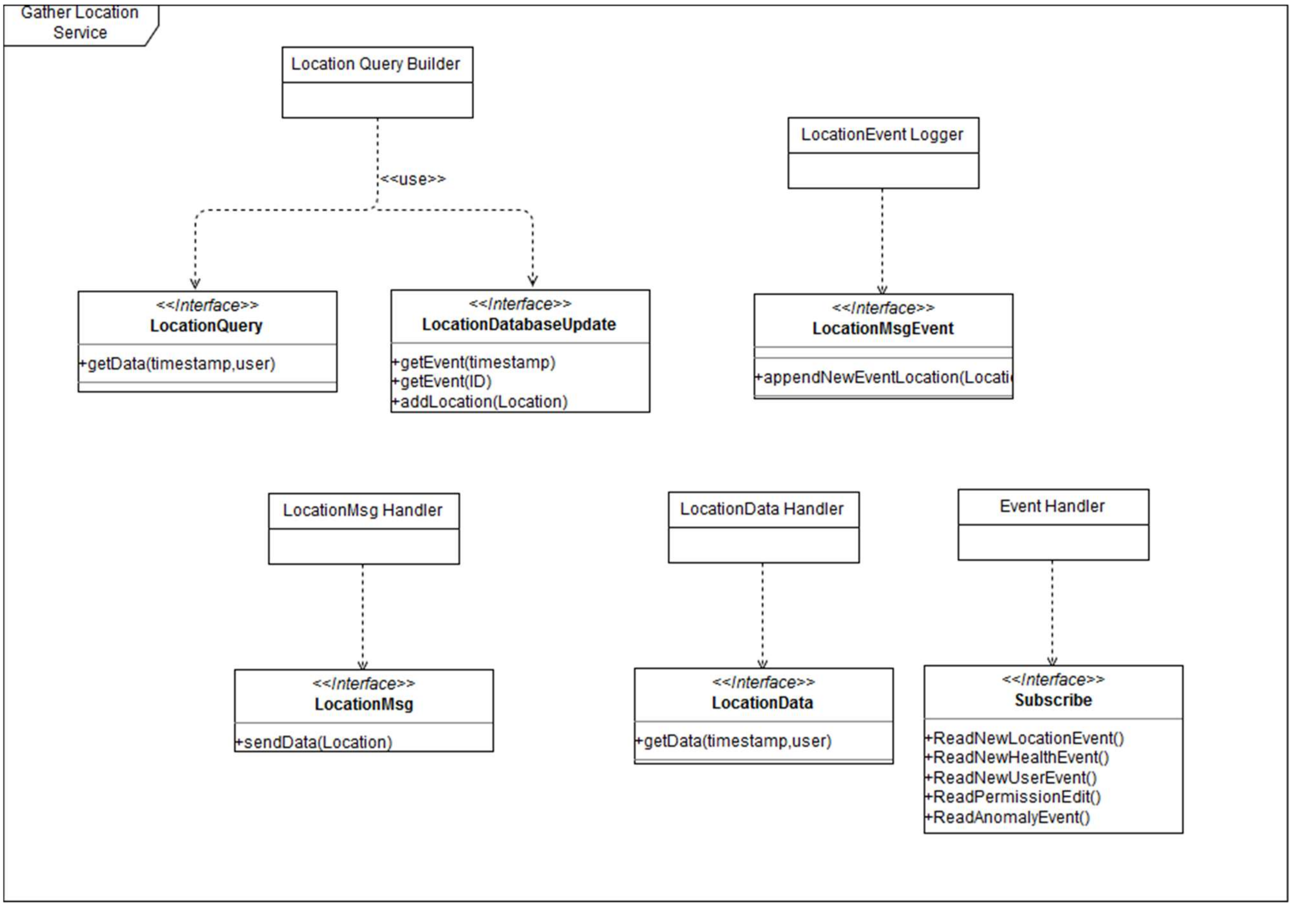

Figure 12 Gather Location Service Interfaces

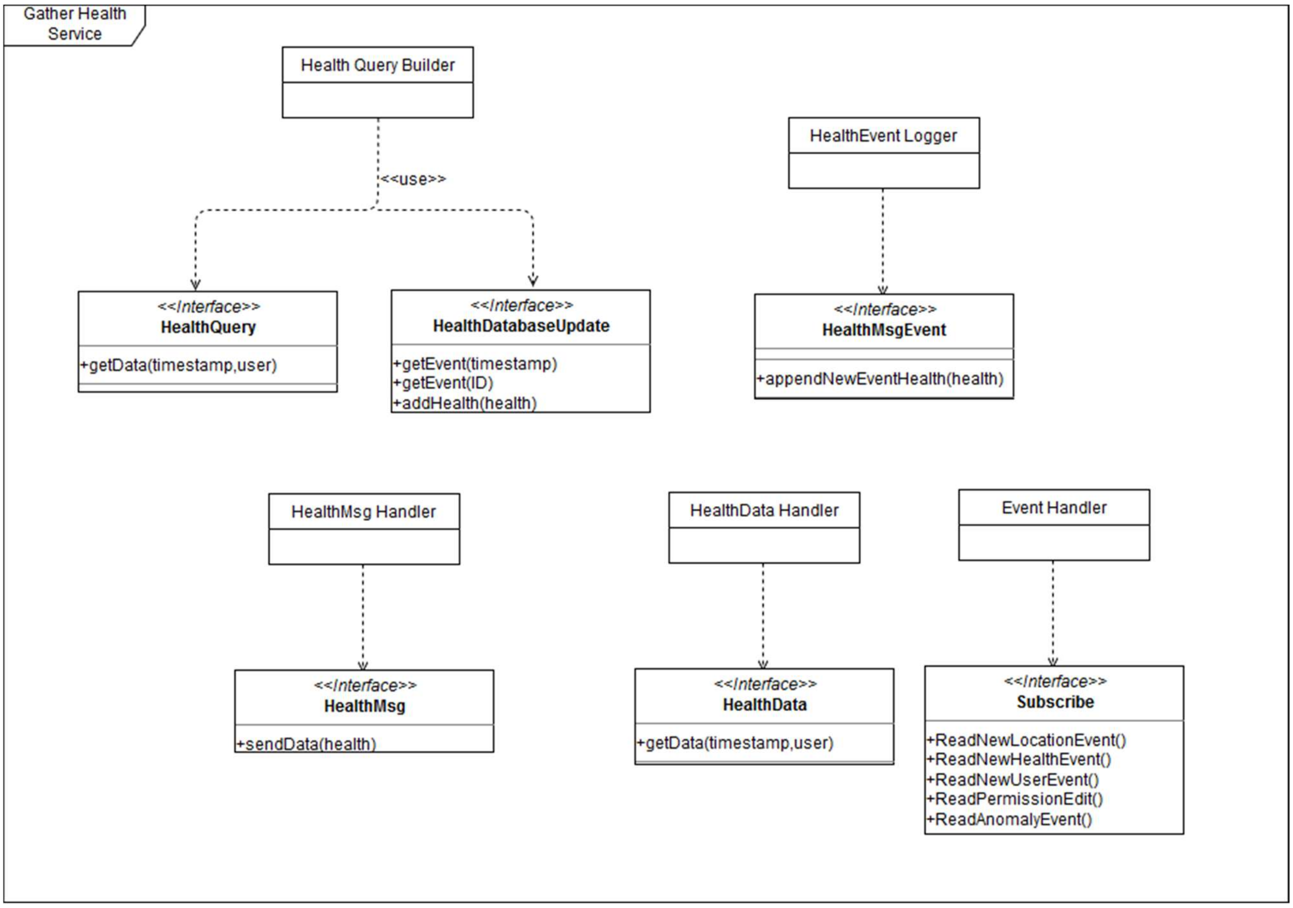

Figure 13 Gather Health Service Interfaces

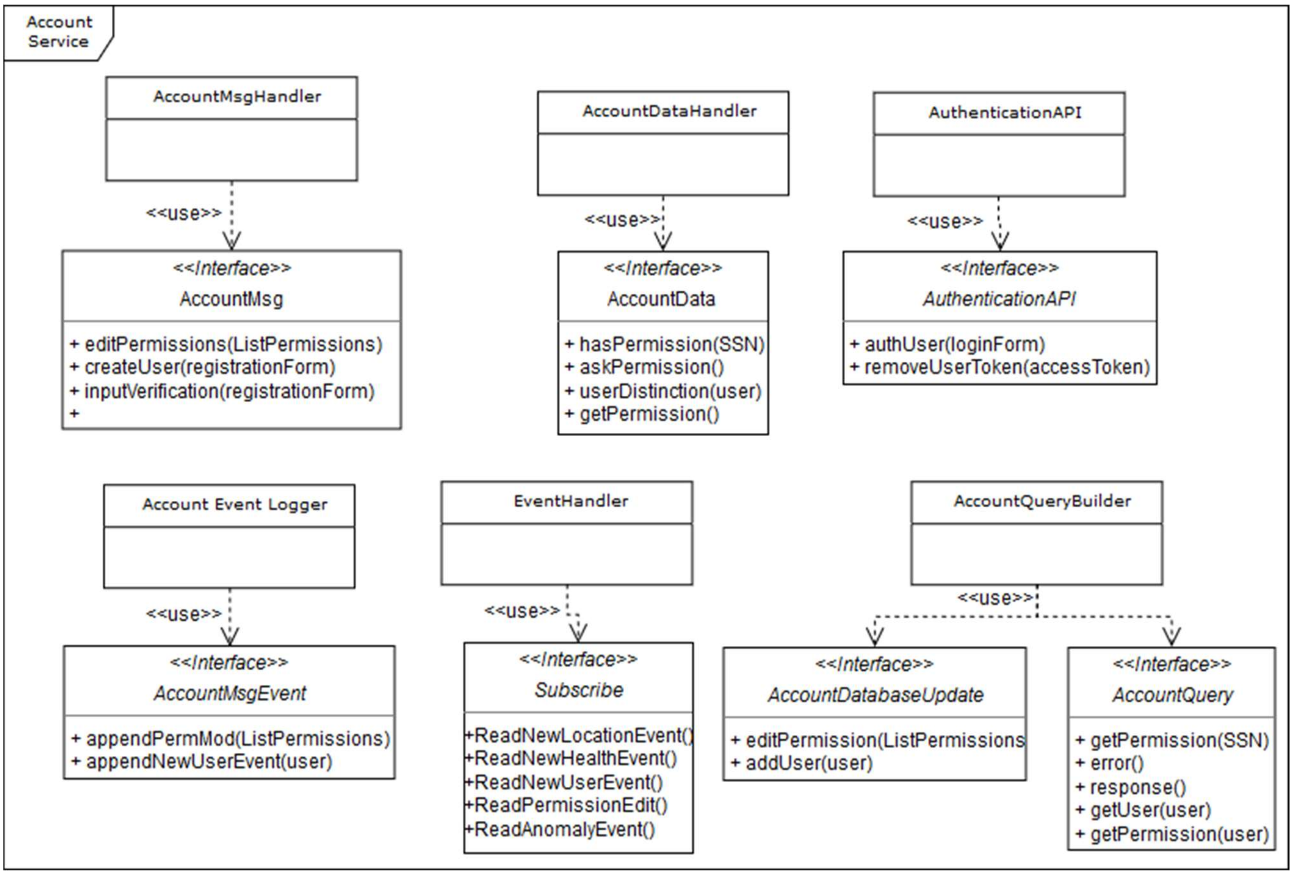

Figure 14 Account Service Interfaces

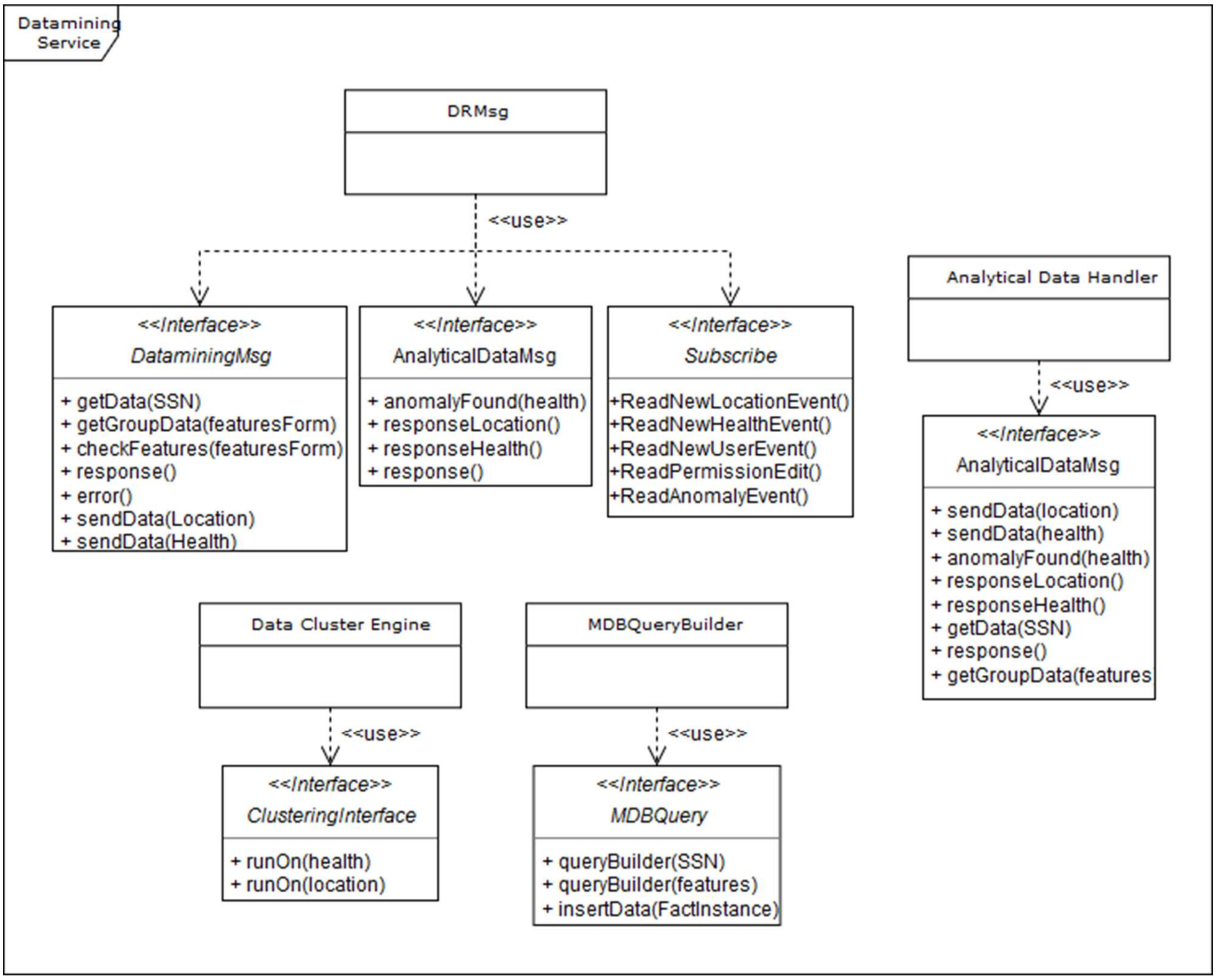

Figure 15 Datamining Service Interfaces

2.5 Runtime view

Guest Sign Up

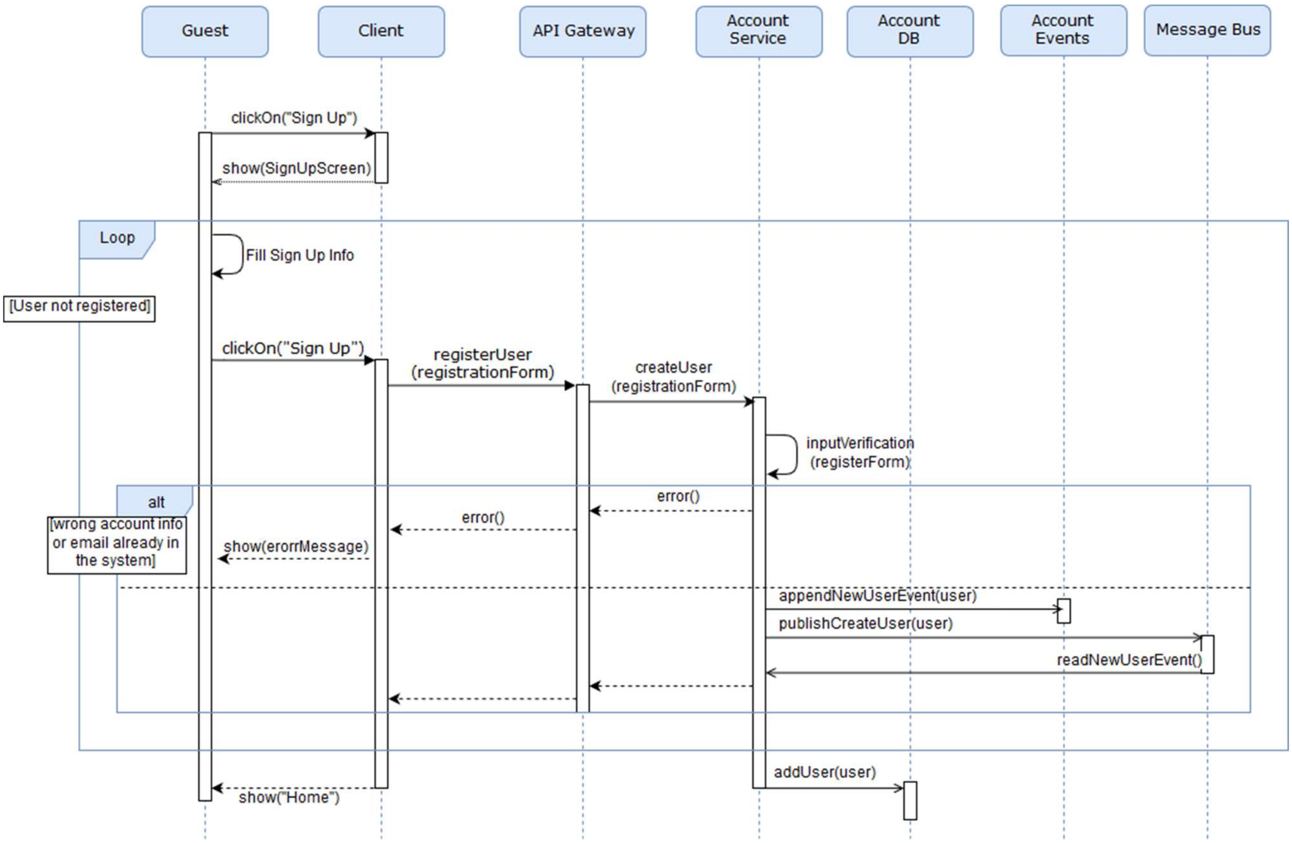

Figure 16 Sign Up Sequence Diagram

User Login

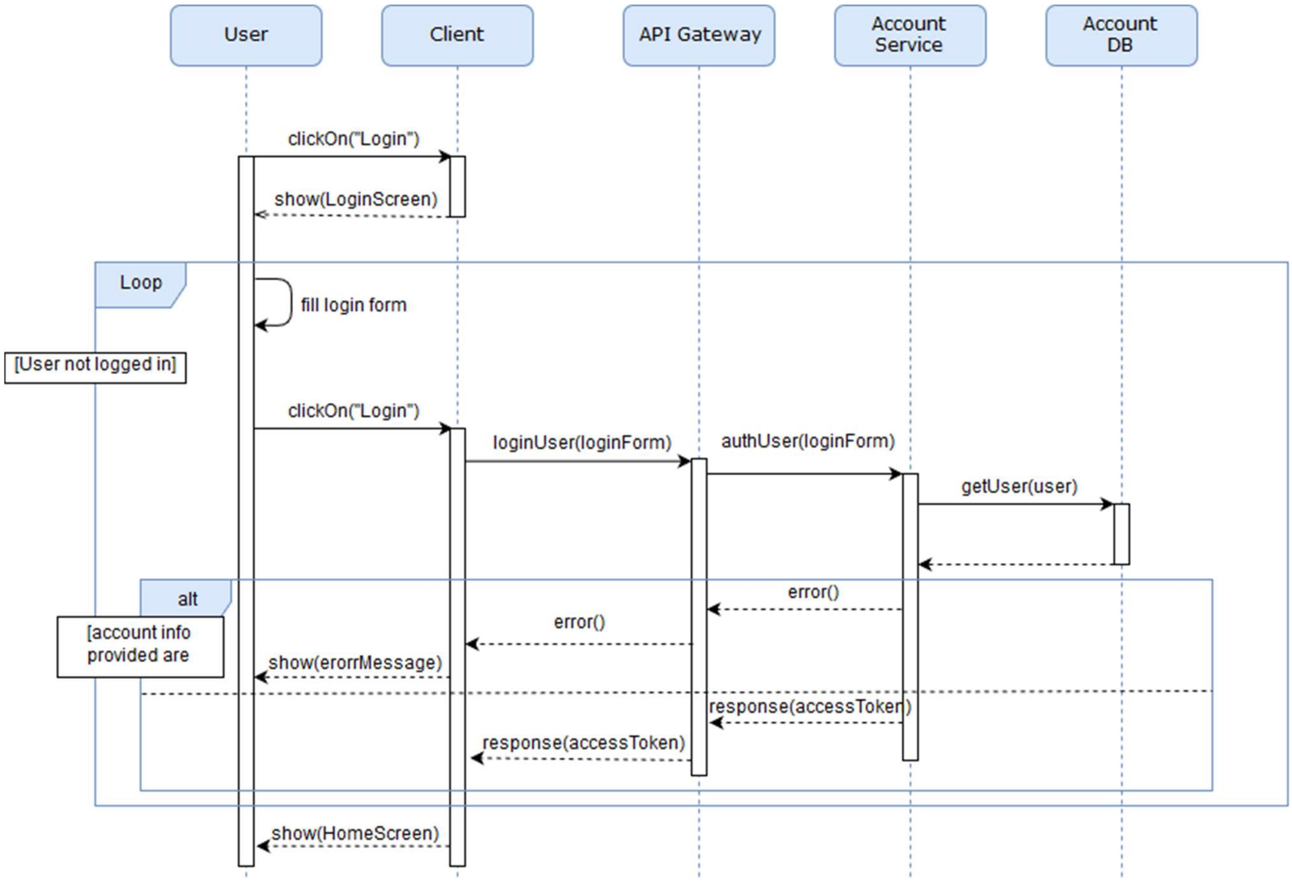

Figure 17 Log In Sequence Diagram

User Logout

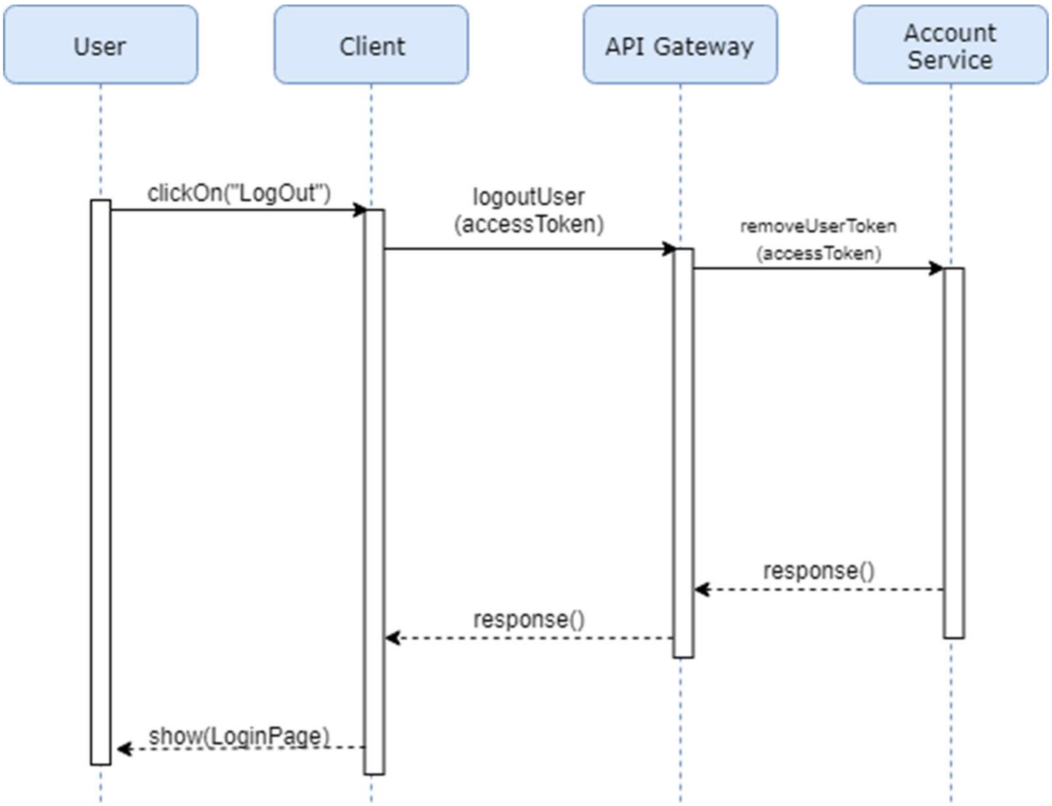

Figure 18 User Logout

Retrieve Data and Anomaly

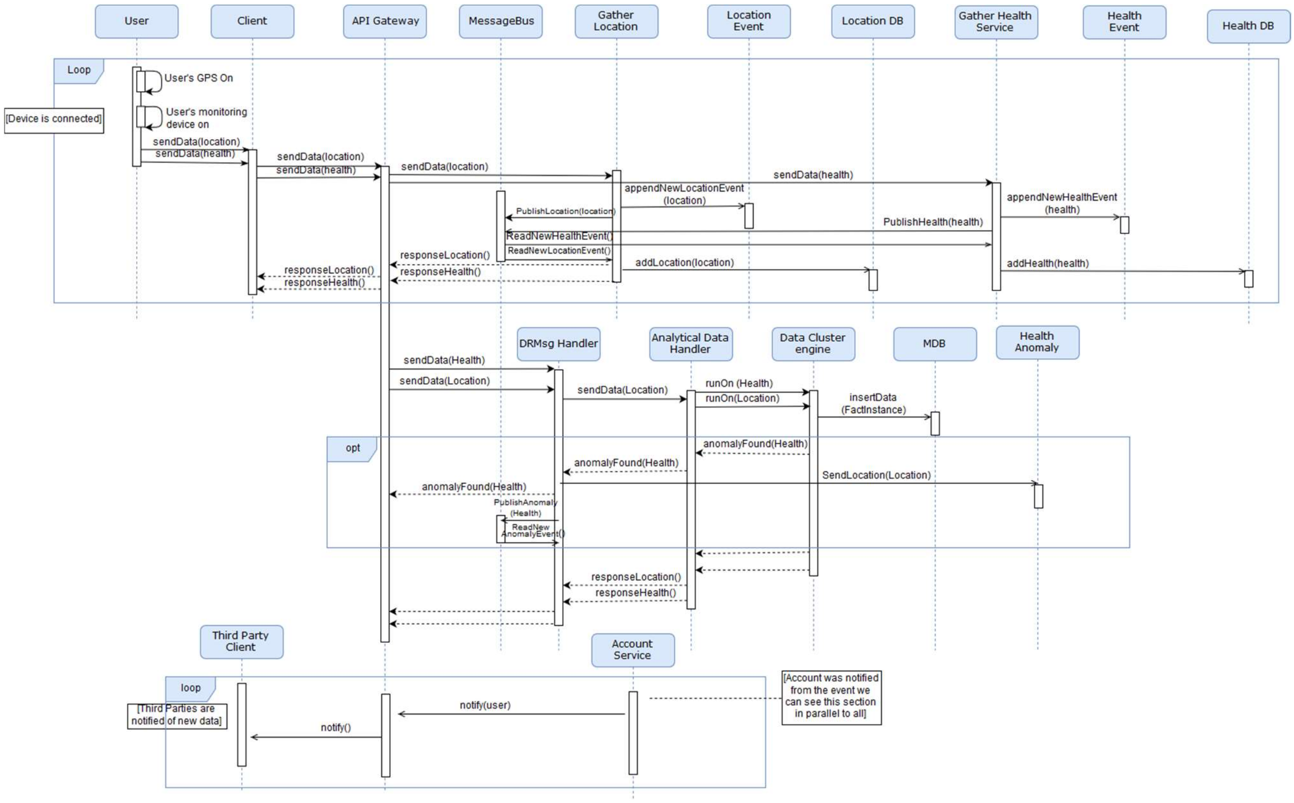

Figure 19 Retrieve Data and Anomaly Sequence Diagram

Review Permissions

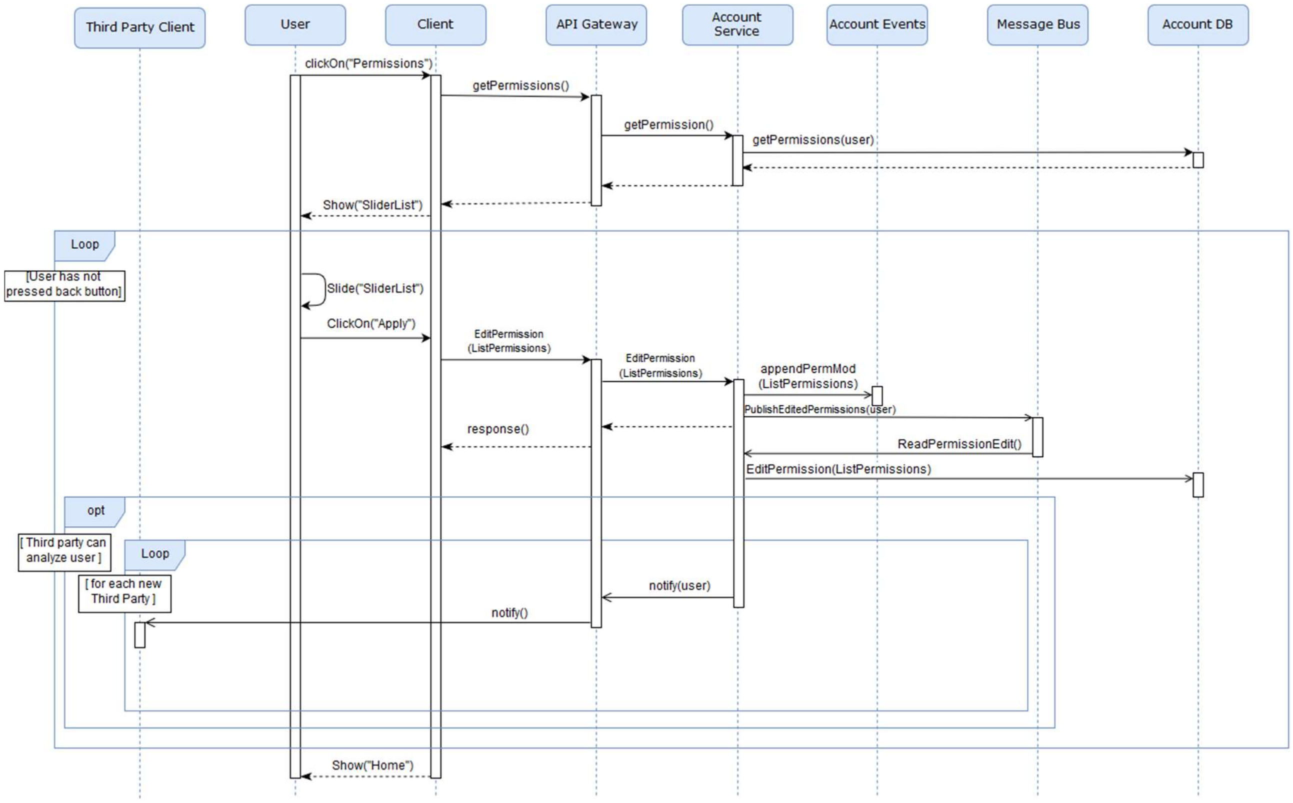

Figure 20 Edit Permissions Sequence Diagram

Private Request

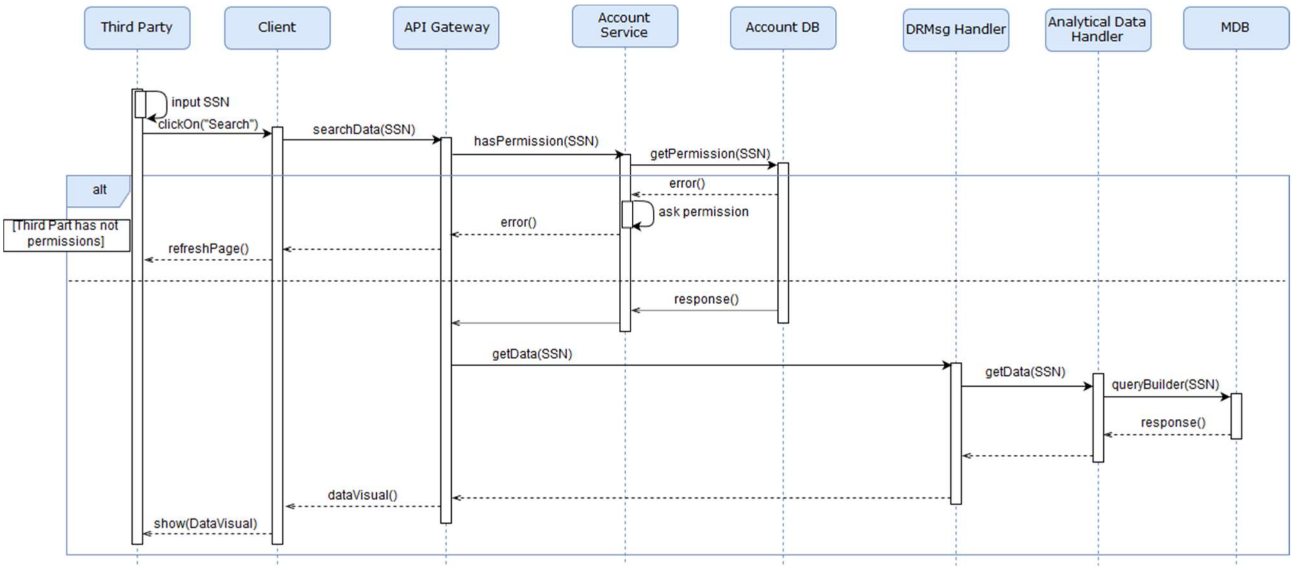

Figure 21 Private Request Sequence Diagram

Group Request

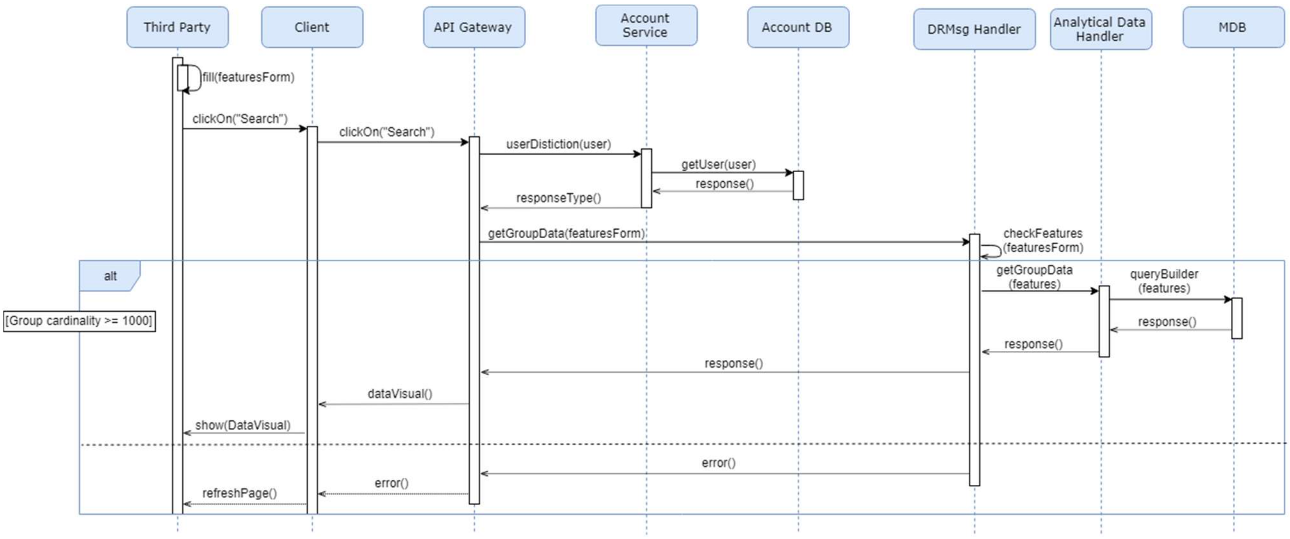

Figure 22 Group Request Sequence Diagram

2.6 Selected architectural styles and patterns

The choice of the following pattern is based on the innovative impact they have had lately while they are relatively well-established.

Primary Pattern: Microservices Pattern

When it comes to the design of a large scale system, with multiple clients and a complex back-end application, it's mandatory to think in advance of the maintainability of the solution and the future deployments of this one. The project is based on the assumption that the service will need upgrades and successive improvements to the architecture to ensure the maximum flexibility and robustness of the service, while the number of user may grow quickly, if the solution is proved to be effective. We will need the internal separation of the logic components and because of their mandatory interaction each of them will have to show different APIs, using HTTPS verbs. This will help in keeping of a high quality the performance constraints presented in the RASD. Besides being very intuitive, this architectural solution offers several benefits:

- Isolation: the group of components is made by several different developers that can work independently and moreover can scale and deploy their corresponding component with a pace different from the others. A consequence of this isolation is the low rate of total down of the system, because when you have such a division, the failure of a component of the system doesn't imply the failure of other parts; this problem is typical of more traditional architectures monolithic mainframes.

- Independence from technology: when is encountered the need of scaling-up, it's possible to just do the migration of the application and of the databases, if it's the scaling is just on the number of features, the system will just build this functionalities on the latest technologies with no constraint on the old architecture's components.

- Small size components: the small dimension of each service makes more clear the task of each developer and makes simpler the verification, the validation and makes really simple to find errors once all the pieces of component are integrated all together. This increases the speed of deployments and improves the overall quality of the project.

It becomes mandatory to organize the deployment of the system with the most suitable methodology for this pattern. We decided that every service will be put inside of a container. This will ensure the isolation and increase the modularity. Moreover the container can be monitored for performance and costs, and it's easy to scale, move or remove.

Security Related Pattern

Since the system implements an API gateway that takes all the client requests, and then redirects them to the right internal services, it is necessary that every request must be verified and approved based on the recognition of the author of the action. The system will be implemented with Access Token. This means that every request must contain in the body all the information necessary to the identification.

External API: Gateway API

The system developed has to manage the collection of data from numerous devices, and these know only the REST end-points of the application. The users can perform some action that need the collaboration of the system and its services. The user to benefit of this service will be asked to go through an application gateway, to ensure flexibility of the solution. The gateway can work as reverse proxy or a request router depending of what kind of action the user wants to accomplish, or after creating an event can broadcast it to the services. An advantage of this architectural choice is the low dependency between the end-points and the single components logic, resulting in something similar to an adapter.

Data Management

Persistent and Persistent Database per Service

Since the decoupling of the components between each other still remains one of the key aspects, the data of every service will remain private and accessible only through the API.

Event Sourcing

Since every service has its own data related, it becomes difficult to keep the consistency between databases and trying to keep operations atomic. The most coherent solution to what has already been described is implying a component as an event bus, that should support the event-driven nature of the system. We would use the event sourcing pattern. The solution found with respect to the lack of atomicity doesn’t involve the use of distributed commit protocols as 2PC but it’ achieved through the use of events. The event sourcing pattern has the peculiarity of being event-centric and of keeping persistent the logical entities. The beauty of the pattern is that you don’t have to save the current state of any entity, instead we will save every event that compete to the modification of the state of an entity. The only way to reconstruct the current state, or any in general (this can be seen as a snapshot of the system), will be by going through all the events of that entity till the desired point. The creation and storing operation of an event, are really simple and naturally atomic, in fact the storing of the event is just appending the event to the list of those that attempted to change the state of the entity. The pattern fits the problem of getting news on the data as well. Every time a client produces a new change in the state it will just communicate through the API, letting the corresponding service subscribe for them, and the system will be also able to do the reverse path by communicating the event to the service interested.

It’s important to say that the pattern is useful in our case because of the static schemas, and of the non-changing objective attributes of the entities, and that the consistency of the system is only eventual, because there’s a “window of inconsistency” between the firing of the event and the consumption of the event through a CQRS query. The second problem does not affect the result of what the system is asked to do. The third parties always visualize data mined on the events related to a single user, and the anomalies are detected as they come.

Command Query Responsibility Segregation (CQRS)

The use of the event sourcing pattern implies the use of CQRS. With the use of event sourcing there is no way to reliably query the data, in fact the for the solution the business value relies primarily on analyzing the data. CQRS describes the concept of having two different models one to change information and one to read it, completely separated from each other.

Having the write model separated from the read model enables the use of the most appropriate strategy to each model and allows both the write and read model to be scalable independently.

Event sourcing is a particularly efficient write model since it works basically as an append log where new information is always added enabling minimal locking. Since each event is irremovable and immutable there are no updates or deletes enabling good write performance. On the other hand since the read model is completely independent allows the freedom to choose the most adequate technology to optimize for queries, which can even be a completely different technology from the write side.

3. Requirement Traceability

In this section every requirement defined in the RASD, functional and non-functional, will be mapped to the corresponding design choice.

Account Service	[R3][R4][R6][R7][R8][R13][R14]
Datamining Service	[R1][R10]
Client	[R2]
Gather Health Service and Gather Location Service	[R5]
Health Anomaly Service and SOS Service	[R11]
Microservices Architecture	[R9]
Every design choice	Availability, Security, Maintainability, Portability, Scalability and Reliability

The mapping made is not strict since every requirement is achieved thanks to the collaboration between all the microservices, but we've linked the service that has more impact to a requirement with the requirement itself.

4. Implementation, Integration and Test Plan

4.1 Implementation plan

The microservices architecture allows the simultaneous work of different developer teams at the same time. Every team can have different and asynchronous deployment rates as well, granting them a high level of freedom, therefore, the focus of this section is in the set-up phase:

Setup the containerization platform (**Docker**).

- Create a container for each service.
  A container is launched by running an image. An image is an executable package that includes everything needed to run an application (a code, a runtime, libraries, environment variables and configuration files).
- Setup communication ports for each container.

Setup the deployment and management platform (**Kubernetes**).

- Setup the **Kubernetes** cluster (configure pods, IP addresses, ports, etc.).
  A pod is a group of one or more containers with shared storage/network and a specification for how to run the containers. Pod's contents are always co-located and co-scheduled and run in a shared context.

- Configure **Liveness** and **Readiness Probes**.
  Liveness probes are used to know when to restart a container, for example, liveness probes could catch an error or a deadlock, where an application is running but unable to make progress.
  Readiness probes are used to know when a container is ready to start accepting traffic. A pod is considered ready when all its Containers are ready.

- Communication: API gateway, Security protocols, Event sourcing

Once the steps stated above are completed, the teams can work independently on their service(s) without the need to follow a strict implementation order plan.

4.2 Integration and testing plan

Integration:

The starting point of the development are the services, therefore a **bottom up** approach is used.

The **bottom up** implementation paradigm fits perfectly with the microservices architectural pattern, when the single services are completed, they will be linked together to build the entire system.

Going into the implementation detail, we have:

1. API Gateway
2. Account service (with parallel implementation of security and authentication protocols via JWTs)
3. Health/Location gathering service
4. Datamining service
5. Health Anomaly service
6. SOS service
7. Device Pair service

The order of the points above is hierarchical and is the most intuitive sequence (even though with the Microservices pattern is perfectly reasonable to integrate the services in any order).

Testing plan:

For each service we have two testing phases:

- **Internal testing** : this is very similar to the standard testing procedure in monolithic systems, where the inner components of the system are tested, focusing on **code** and **branch coverage.**
- **External testing**: this phase consists of integrating the new functionalities added into the service with the publisher-subscribe system, focusing on **API coverage** and **Event sourcing**.

Some standard testing techniques can be applied as well, keeping in mind the cluster structure of the microservices architecture:

- **SOAK testing:** Soak Testing is a non-functional testing in which, the System under Load (SUL, in our case every service first and then the whole system) is tested and verified so that it can withstand a huge volume of load for an extended period of time. Soak Testing is nothing but a type of a performance test, capable to find whether the system will stand up to a very high volume of usage and to see what would happen outside its design expectations.

  A system may behave normally when used for 2 hours, but when the same system is used continuously for 10 hours or more, it may fail or behave abnormally, randomly or may crash. To predict such failure Soak Testing is performed.
  Since we're dealing with a critical application, SOAK tests are of utmost importance.

  Specifically, a series of multiple requests is forwarded to the system's entry point, testing if the load balancing functionality provided by Kubernetes works properly and the various requests are distributed correctly across the pods present.

- **Burn-in testing:** The burn-in test is complimentary to the SOAK test, the system is run for an extended length of time in order to identify any potential problems. It aims to reveal any problems or defects within a system by operating it in the most rigorous, extreme or extended working conditions. Here we're testing Kubernetes reaction after single node and\or entire service failure.

- **Kubernetes performance testing**: testing the ability of Kubernetes to increase replicas of pods inside the cluster nodes accordingly to workload growth. Going into further detail inside the Event Sourcing pattern mentioned above, which necessary to keep consistent the data across all services, it will be developed in a "step-by-step" fashion. Every time a service has an update that requires a new event to be added in the publisher subscribe schema, this will be integrated and tested. Thus, the event-driven architecture will expand as the functionalities implemented in the services increase.

- **Incremental Testing:** with this test we will verify the interfaces and interaction between modules and services. Using Incremental integration testing, we will integrate the services one by one using **stubs** or **drivers** to uncover the defects.

When all the services will be completed and integrated, **alpha** and **beta** tests will be performed:

- **Alpha testing** is a type of acceptance testing, performed to identify all possible issues/bugs before releasing the product to everyday users or public. The focus of this testing is to simulate real users by using Black-box and White-box techniques (Black-box without peering into the system internal structures or workings, White-box the contrary) . The aim is to carry out the tasks that a typical user might perform. Alpha testing is carried out in a lab environment and the testers will be internal employees of the TrackMe organization.

- **Beta testing** is performed by "real users" of the software application in a "real environment" and can be considered as a form of external User Acceptance Testing.
  Beta version of the software will be released to 1000 end-users to obtain feedback on the product quality. Beta testing reduces product failure risks and provides increased quality of the product through customer validation.
  It is the final test before shipping the product to the customers. Direct feedback from customers is a major advantage of Beta Testing. This testing helps to tests the product in real time environment.

5. Effort Spent

Name	Effort
Kevin Mato	25h group and 20h alone
Antonio Mazzeo	25h group and 20h alone

6. References

1. **Docker Documentation:** https://docs.docker.com/ee/

2. **Kubernetes Documentation:** https://kubernetes.io/docs/home/?path=users&persona=app-developer&level=foundational

3. **Microservices:** https://smartbear.com/learn/api-design/what-are-microservices/

4. **CQRS:** https://martinfowler.com/bliki/CQRS.html

5. **Event Sourcing:** https://martinfowler.com/eaaDev/EventSourcing.html

6. **Notes from:** http://emanueledellavalle.org/teaching/digital-project-management-2018-19/