SDU
University of
Southern Denmark

Bachelor Project

Replacement of a PAAS Backend with a Custom Real-time Solution

Author

Troels Kaldau

Project Supervisor

Kamrul Islam Shahin

Liaison

Daniel Holst

Product Owner

MyEpi Aps

Examination number: 497065

June 2023

CONTENTS

LIST OF FIGURES

I. Abstract

Firebase is a popular server-less solution for a wide range of software products, particularly aimed at mobile applications. It offers a simple setup and covers a wide range of responsibilities which would otherwise need to be developed from the ground up, or covered by a range of other services, each with their own setup requirements.

While this solution offers significant advantages, using Firebase limits the developers flexibility and control of their service. They are constrained to the features which Firebase offers and combining these with custom solutions can cause issues based on increased complexity and incompatibility. This project has sought to design a replacement service which covers an array of the responsibilities otherwise covered by Firebase. The project takes basis in the concrete case of MyEpi, a company seeking to replace their implementation of Firebase. Therefore, the responsibilities in focus are derived from the current use of Firebase within the software product offered by MyEpi.

To reach a design, the responsibilities which the replacement service needed to cover were first elicited through source code of the MyEpi product in question as well as through stakeholder meetings. They were thereafter analysed for the obtainable value from deriving new solutions compared to known solutions within the organisation. The responsibilities chosen as the focus of this project were stream Management, Code Generation, and Authentication. Concrete functional requirements were elicited and prioritised to demonstrate the design for covering the selected responsibilities. Use cases were extrapolated from the requirements and class diagrams as well as component diagrams were derived from the use cases. The Design was implemented and tested using both unit tests and integration tests to validate the functionality. Thereafter a performance test was conducted to test and improve the average and maximum response time of the service.

The outcome of the process was an implemented service which demonstrates the coverage of the selected responsibilities. It has proved the viability of the selected technologies, hereunder the framework of NestJS, selected transport protocol of gRPC, and the state management method of MongoDB ChangeStream's. The most noticeable results have been the limitations of the MongoDB ChangeStream's. Performance implications have been linked to both excessive MongoDB connections as well as the method in which the used MongoDB driver handles connection limitations. The final service showed an average response time of 102.5 milliseconds for a single service servicing 500 subscribed clients, each making 1 change request per second and expecting a stream event in return.

The final findings of this project has primarily shown the promise of gRPC. It has proven it to be an efficient transport protocol, both in implementation and performance.

II. Introduction

This section is dedicated to clarifying the scope and purpose of the project described in this report. It provides a description of the case, the problems that should be solved, the intended focus and by whom they are defined, as well as the project formulation and project limitations.

This project is based on a business case offered by the company MyEpi Aps and defined by the affiliate company LittleGiants Aps. LittleGiants is responsible for the product development for MyEpi. MyEpi was founded in February 2021 by Jakob Junker. The company is a lean Start-up working closely with its affiliate company LittleGiants, within the same offices. Its mission statement is defined as:

"Ensure safety and security for families greatly affected by epilepsy" [1]

The company has sought to do this through a software application for the Apple Smartwatch series. The product uses information available from the smartwatch sensors to detect epileptic seizures through an algorithm commissioned from a professor at The University of Southern Denmark.

LittleGiants was founded in March 2018 by the current owners Daniel Holst, Camilla Holst, Carsten Holt, and Steffen Aagaard. The company is primarily a cross platform application consultancy agency; however, they have offered a wide range of services in an attempt to find their ideal market. Beyond experimenting with different services, they have experimented with many different technologies to find the most suitable tech stack. They have chosen Flutter as their preferred front-end framework. The back-end solution has been more unstable, with different frameworks, languages, transfer protocols, etc., considered, tested, and used. The current preferred solution is NestJS as framework, Typescript as language, MongoDB as database, and OpenAPI for code generation. The experimentation has slowed down, however, the company is still considering other solutions.

Currently, the MyEpi software application is server-less and based on Firebase. However, interest in expanding the software service beyond the current product has proven the server-less solution too inflexible. Therefore, the company would like to replace this solution with a back-end service, of which they can have more control. However, it is important that the current service does not lose functionality. Therefore, the new solution must offer real-time data in the same form as Firebase, to avoid the need for new state maintenance solutions implemented in the front-end. As Firebase has its own client library for front-end implementation, the new solution cannot offer full compatibility with the current front-end implementation. However, to increase the ease of implementation, the back-end should be developed as close to the current Firebase setup as possible in terms of offered endpoints and data structuring. Alternative implementations can be considered when sufficiently justified, but the implications must be weight against how much value it brings to the table.

The focus of this project is to design a solution for the project, and so implementing any concrete endpoints is not the focus of the project. The design should be clearly defined, defended, and verified to fulfil the requirements and needs defined in the case. The design includes the technologies used, architectural templates for domain implementation, as well as concrete architectural pattern implementations for selected non-domain specific services.

The project is written for MyEpi and is responsible only to this company. However, due to the affiliation between MyEpi and LittleGiants, it has been requested that the project take the interests of both companies into consideration.

Problem Statement:

"The product owner wants more control of its back-end service to avoid restrictions for future product development, but does not want to lose the functionality that they are currently utilising"

III. Method

This section is dedicated to clarifying the method used for fulfilling the project formulation. It outlines the use of processes and tools in partially chronological order. It is a guideline for the method used, however, as software engineering is a fluid and iterative process, deviations from the method are inevitable.

A. System Analysis and Design

The first step is to define the system which is to be developed. This includes identifying the requirements of the system, prioritising set requirements, and converting the requirements into an implementable design.

1) Elicitation

First, the current system was analysed for use of Firebase functionality. This was done with a dependency analysis. Files containing dependencies of Firebase services were first identified and listed throughout the project. Thereafter, each file was analysed to identify the functionality provided by it, and the way in which Firebase played into the functionality. This resulted in a list of concrete functionalities provided by Firebase, as well as an overview of the dependency distribution. The result was formed in a list of functional requirements reflecting the functionality that would need to be provided to replicate the Firebase service as it is currently used. To ensure compatibility between prospects of the company and the developed solution, documentation of future functional requirements of the user interface was analysed. Each requirement was evaluated for its dependency on back-end functionality. This resulted in a revised requirement specification with support for planed functionality.

2) Verification

To verify the captured requirements of the project a meeting was scheduled with the project manager. Before the meeting, an agenda was written as well as a list of questions for clarification of written and intended functionality of the analysed system. The meeting started with an explanation of the elicited requirements to establish a common ground, followed by discussion of the correctness and relevance of these. The meeting resulted in a clear list of requirements, with added sources for requirements elicitation.

3) Responsibility Analysis

Once the list of requirements was final, a responsibility analysis was carried out. This sought to identify central areas of responsibility from the requirements to investigate their risk of complications in regards to the project. The method included five steps:

- Identifying the responsibility
- Identifying the desired trades of a future solution
- Identifying the systems current solution
- Identifying known solutions within the organisation
- Identifying central design decisions of the solution.

This method revealed the distance between the desired solution, the known solutions, and the currently used solution. It revealed the value aimed to be gained by the replacement service. The output revealed the requirements with the most value and with the least certain solutions. High value solutions would be prioritised in the requirement selection. uncertain solutions would be important to solve early in the process, as their outcome could have a great impact on the further proceedings of the project.

4) Prioritisation

Following the responsibility analysis, a new meeting was setup with the project manager. This had the purpose of discussing the results of the analysis to agree upon the most relevant requirements. Beyond the value proposed by the analysis, the focus was to select diverse requirements. The intend was to cover the service as broadly as possible, and thereby create a structure which could be extrapolated to the rest of the requirements. This would ease and accelerate the complete service implementation as the remaining functionality would have a similar structure to the already implemented functionality. It would also reduce the risk of the project, as the main architecture and technologies would already have been tested and validated. Diverse requirements would therefor bring the most value to the stakeholders. The final prioritisation was documented with the MoSCoW principle. From the MoSCoW prioritisation, all Must requirements were selected as well as a selection of the Should requirements. The scope was kept relatively small to ensure adequate time for completion.

5) Use-case Description

Based on the functional requirements, the use cases were described with four fields:

- Which requirements does it fulfil
- What is the intended outcome of the use case
- What are the intended steps for fulfilling the use case
- What are the potential error states that can arise, and how should they be handled

This provided a frame for fulfilling the intended functionality.

6) Activity Diagrams

Based on the requirement expansion, two activity diagrams were formed. One showed separate flows modelling the most important use cases, as well as examples of similar use cases. This gave a visual and easily digestible overview of the use cases. The second diagram showed a single flow which incorporated the use cases involved in starting, maintaining, and ending an authentication session. This was identified as the most valuable and intrinsic flow of the service. The diagram was therefore made to give an understandable view of it. It was supplemented by a state machine showing the flow in an even simpler diagram.

7) Sequence Diagrams
To provide a more concrete and in-depth overview of the process flow for the use cases, a sequence diagram was made for each one. This showed the flow between the user, the system, and the persistence method of the system. From this, four implementable concepts could be identified:

- Data objects – encapsulations of semantically or functionally coherent data
- Internal processes – processes either creating or modifying data
- Persistence processes – communication with the persistence method
- Errors – potential error states which the user would be informed of

By identifying these concepts from every sequence diagram, a union could be made for each area of responsibility. This removed repeated concepts from a single area of responsibility. Afterwards, an intersection of the concepts could be made across the areas of responsibility. This gave a list of common concepts which were reusable across areas of responsibility.

8) Class Diagrams
The elicited concepts from the sequence diagrams gave the fundamental building blocks for creating class diagrams. The concepts provided both data classes and interfaces, and the sequence diagrams documented the dependencies between the classes. The class diagrams provided the general structure for the code implementation

9) Component Diagrams
From the class diagrams, two component diagrams could be derived. This demonstrated the organisation of modules in the system as well as the provided and required interfaces and the methods for connecting internal and external services.

10) Design Decisions
From the responsibility analysis, a range of design decisions could be made. This included the general tech-stack, the service distribution, and the method for stream management. These were all key areas of concern and creates a frame for the implementation of the project.

B. Implementation

Once the system has been designed it must be implemented diligently to ensure that the final product follows the system design. For this, a work process and fitting management tools must be selected. This should ensure that the design is implemented systematically and efficiently in small iterations with sufficient verification.

1) Implementation Division
For the implementational process an iterative process was chosen with a sprint duration of one week. This gave approximately two and a half workdays per sprint. For each sprint a section was written including a purpose and a list of objectives. The objectives reflected the requirements it would fulfil, but also included technical challenges that would need to be overcome. After each sprint, a result section was written documenting what was accomplished, what challenges were meet, and what needed to be kept in mind or finished in the next sprint.

2) Documentation Methods
Beyond the written sprint sections, the code progress was documented through GitHub. For each sprint a branch was made based on the master branch. The sprint was implemented here, and changes committed with descriptive messages. At the end of a sprint a merge request was made to the master branch with a more concise description of what changes had been made. After the merge with master a release version was produced with a description of the state of the release. This made it easier to identify the major feature changes made throughout the project.

3) Task Selection
The tasks for the sprint scope were prioritised based on the perceived difficulty of the implementation, the understanding of the implementation, and the effect that the used implementation would have on other tasks. The goal was to avoid redoing work by completing the tasks with the biggest impact on the project structure first. As such, the first sprint was a prototype with the single purpose of validating the compatibility of the tech-stack and the basic features it needed to provide. Once this was done, there was a basic structure for the core feature types which could be copied for the implementation of the functional requirements.

4) Unit Testing
The purpose of the project was not just to make a functional service, but to create an architecture upon which the final service, and potentially other services, could be based. Therefore, validating the individual components was of high priority. The functionality had been divided into 3 main layers: endpoints, business logic, and persistence. For each module, a test file was made for the business logic and persistence layers. Each public function was tested with the possible states and the returning values or errors were validated. This ensured a 100% test coverage of the implemented logic. Due to the business logic layers use of the persistence layer the persistence layer was tested twice. The reason for this was to quickly identify the root of a given issue. The deeper nested an error occurs from the point of discovery, the harder it is to detect the source. With the separation of business logic and persistence, there was also the opportunity of switching the persistence method. For this, a dedicated persistence test suit would make the switch easier to validate.

5) Integration Testing
To test the endpoints an integration test suite was setup. This included the creation of a client service which could call the endpoints. As the final service would be used by an

application written in dart, dart was chosen as the language for the client, to ensure that successful communication could not be contributed to the NestJS framework. The integration test focused on the possible input values and return types. It did not seek to test every possible state, as this was covered in the unit tests. The purpose was not to test the service logic, but to ensure correct data transfer between client and service.

6) Performance Testing

To test the performance of the service multiple performance tests were run. The performance tests were based on the same test client as the integration tests. For each test, a number of clients were created, each in their own process implemented with the concept of isolates [2]. Each client would subscribe to a stream exposed by the service. Thereafter they would emit 10 requests for change with a random frequency of between 500 and 1500 milliseconds. Each message would trigger a response through the subscribed stream. To test the performance of the service, each client would measure the delay between an emitted request for change and a corresponding received stream event. Tests were conducted with a different number of clients and changes to the parameters of the service to identify performance inhibitors. Some tests were run several times on the same service instance to test for accumulating delays. Other tests were run a single time were after the service instance was restarted. After an inhibitor had been identified, it was attempted solved and the test which identified it was run again. Successful solutions to identified inhibitors were kept for subsequent tests. Thereby the process iteratively zeroed in on the best performing solution.

IV. System Analysis and Design

This section is dedicated to documenting the process used for identifying and prioritising requirements as well as converting them to implementable designs for the implementations process. It outlines the results of each process as well as the thought process and information upon which it was based.

A. Requirements Elicitation

First, the requirements must be elicited from available documentation as well as stakeholders. The elicited requirements contain all the currently available expectations of the finished product, which goes beyond what will be implemented in this project.

1) Functional Requirements

As mentioned in the Method, the functional requirements were elicited through a combination of concept location, dependency analysis, stakeholder meetings, and front-end requirement analysis. The requirements have been iteratively elicited and represent the full functionality of the final product. These can be seen in Appendix A.

2) Non-functional Requirements

The nonfunctional requirements were elicited solely through stakeholder meetings, and so, they reflect the stakeholders' expectations of the product characteristics. An overview of the selected requirements can be seen in Appendix A.

In terms of scalability, the service will be run in a containerised environment. The service needs to be stateless so that it can be scaled horizontally. Due to the containerised hosting, the internal scalability of the service is of little concern. In terms of availability, there has not been set a concrete expectation of up-time. This is due to the difficulty of testing this without having a service in production.

In terms of performance, there has been set an expected number of active users per service instance of 500. This is based on the following details: The expected number of active users in the short-term future of the service is 1000. An active user is expected to have a maximum activity level of 1 modification per second. There is expected to be a minimum of four service instances. A state of two killed services is considered an undesired but possible state. Two services must therefore be able to handle the currently expected load. One service must therefore be able to handle 500 active users.

In terms of modifiability, there will be a high focus on code separation due to high experimentation within the organisation. Technologies should be kept separate to ensure easy replacement.

In terms of testability the highest risk in the system is a service killing endpoint state, as this could sequentially and continuously shut down services faster than they can be created, rendering the system unavailable. This is also known as a cascading failure. To avoid this, there should be a complete test coverage of the endpoint states. The total expected internal test coverage is at 95%.

In terms of security, it is expected that common, modern, and defended standards are used for security measures. Bcrypt is expected for password encryption.

B. Responsibility Analysis

As described in the introduction, the aim of this project is not implementation of functional requirements but design and verification of the architecture and tech stack meant to support the system. The prioritisation of the requirements is therefore tied to the analysis and design as it is here the technical challenges will be revealed. This section is meant to uncover these technical challenges. The prioritisation will be based on the most valuable and risk prone requirements found in the analysis and design of the system. To compare the current solutions with known solutions within the organisation, a project named MealBuilder has been selected as a reference project.

1) Authentication

The authentication solution must allow for login with email and password. The connection must be encrypted with asymmetric encryption to ensure security. It must allow the authentication to last between sessions. It must ensure that the open authentication cannot be violated by other actors. Authentication is also a domain-less responsibility shared by many applications. It is therefore an obvious responsibility to extract into a reusable component. This will maximise the value of the project to the stakeholders at LittleGiants.

Currently, the authentication is handled solely by Firebase through FirebaseAuth [3]. FirebaseAuth uses JWT tokens to manage users. The Flutter FirebaseAuth package abstracts the handling of JWT tokens from the developer. The current solution only incorporates sign-in with email and password, but FirebaseAuth offers both OAuth2.0, supporting a long list of integrated authentication providers, as well as 2-factor authentication.

Within the MealBuilder back-end authentication has been implemented as a separate controller in the NestJS application. It handles the creation of users, authentication of users, creation of authentication sessions, and tear-down of authentication sessions. The created *User* object only contains authentication data and a reference ID. The id is later associated with the creation of a user profile, containing domain specific data. The implemented authentication strategy is a two-factor authentication using SMS verification codes. The Authentication session is managed with the use of two tokens – an access token and a refresh token. The access token is a signed token containing the user's id and phone number. The access token is generated using the *JWTService* which produces a JWT token. The refresh token is a unique random string of 16 bytes in hex format. The token is stored in the database with the user's id and an expiration date set to six months. The access token is

used to validate the user with the domain endpoints but it is short lived. To re-validate, the user can send the refresh token and obtain a new access token. This reduces the exposure time of the access tokens and reduces the exposure frequency of the refresh token. This authentication method is tightly coupled with the overall authentication controller and replacing the method or adding additional authentication methods is not as easy as it could be. There is a distinct separation of concerns between the handling of authentication and handling of an authentication session. Separating the concerns could simplify the implementation and increase modifiability.

In summary, for handling responsibilities tied to authentication, the service must maintain and have access to authentication information. This is compared to the information sent by the user. If the user is authenticated, a session is created. The user must receive a reference to the session (session token), which can be given to other services for validation. Other services must therefore have a way of validating the given session token. They must also be able to get a unique user identifier from the session token to maintain the user's state.

2) Authorisation

The highest responsibility of authorisation is ensuring confidentiality and integrity of the users' data. The authorisation should rely completely on the back-end system and should treat the front-end as an untrusted entity. It should be possible to define authorisation guards at multiple levels to avoid scattering general guards unnecessarily. The guards should be highly visible and manageable to ensure that developers add the correct guards and avoid over exposing resources.

Currently, the front-end system uses unguarded resources. This means, the authorisation is handled in the front-end. This is done primarily by getting the current user id from the FirebaseAuth library and using this for queries. This ensures that the data collected belongs to the authenticated user. However, handling authorisation on the client device is generally not safe, as it exposes the authorisation logic to potentially malicious users.

Within the MealBuilder back-end authorisation has been implemented with the NestJS UseGuard Library. This allows middle-ware injection on endpoints which is run before the endpoint function. The middle-ware accepts the endpoint context containing the payload and parameters of the request. It can then carry out any validation on the request before allowing the endpoint function to execute or it can throw an error if the validation fails. However, this solution is only effective for REST-endpoints and WebSocket connection request, but not for active WebSocket communication. Currently the active WebSocket authorisation is placed in domain entity specific functions, wherein it is determined which channels the data object should be sent to. This handling is significantly different from the REST-endpoint authorisation strategy and is less efficient at conveying the intent of authorisation logic.

The tight coupling to the domain discourages the generalisation of an authorisation service. However, the logic can be abstracted into modules or packages to keep the implementation as clean and manageable as possible. The implementation should ideally be short readable statements clearly defining the authorisation requirements of a particular resource in the system. There should be a consistent tactic for handling authorisation, as consistency makes it easier to verify endpoint compliance with trans-trust-boundary and general permission documentation.

3) Stream Management

The objective is to make domain state management obsolete in the front-end application, and it is therefore paramount that the clients can rely upon the back-end to provide the correct current state of the system. Beyond this, the overhead of implementing streams in the service must also be as small as possible. Ideally, the subscription endpoints should be code generated. The needed state maintenance in the front-end should also be minimised and made simple to avoid state inconsistency due to front-end misconfiguration.

Currently, streams are managed by Firebase which abstracts the state management from the front-end developers. Firebase downloads and maintains data which the user is subscribed to and returns data from the in-memory stores. It also maintains a queue of requests when the client is offline [4].

Within the MealBuilder back-end the real-time data has been implemented with the use of WebSocket services. There is one service for each domain entity. The services are based on the NestJS library GateWay and consist of 2 primary functions: *handleEvent* and *emit*. The *handleEvent* function is informed of attempted client subscriptions to the domain entity and is given a reference to the socket. The socket can then be subscribed to additional sub-channels based on the client type and authorisations.

The *emit* function handles the emission of messages to the subscription channels of the associated domain entity. The function emits to the main channel and sub-channel based on available information on the given data object. Within the function, guards are written to determine who should receive what.

The *emit* function of a domain object is called within any function which manipulates the domain object. This, however, places the responsibility of ensuring correct emission of data from each endpoint on the developer. If they don't, the clients won't be informed of the change and will have an invalid state. Ideally, the stream of data should be directly and implicitly connected to the actual state of the back-end data storage solution. This would simplify the responsibility of the developers and reduce the possibility of errors.

Streamed data can be provided in multiple ways, with varying need for maintenance. Data can be directly updated by transferring the results of the entire subscribed query when state has changed. This minimises the need for maintenance by the

client but increases the data usage. Alternatively, only changed entities can be transferred, which requires the client to insert the updated entity. The most efficient method is transferring the changes made, also called the ”delta”, rather than the full object. This requires the client to perform the changes on the stored entities, and thereby replicate the changes made to state, rather than the state itself. Beyond this, a minimal stream implementation would be to simply inform the front-end of a changed resource and leave it up to the front-end to update this resource through requests. This would ensure real time state management with a minimal back-end implementation; however, it would be slower and more data intensive.

A client can connect to multiple WebSocket's, however, each WebSocket has an overhead. It must therefore be decided whether to use a single WebSocket with more complex emission handling or individual sockets each with a singular purpose. The former will increase performance; however, the latter will decrease complexity and simplify authorisation.

The solution should be based on streams. A secure guard system must be implemented to safeguard the stream channels. The streams should ideally be directly connected to the data state, instead of being handled as a step in the state manipulation process to avoid faults in the implementation. A data transfer strategy must be chosen to balance the simplicity of client state management with data usage.

4) Code Generation

The desired system is an API back-end to which clients can connect over a network connection. Due to the connection type, there is not a predetermined communications protocol, beyond the basics of http restrictions. Therefore, a communications structure must be created and documented to ensure compliance between client and server. This compliance requires 3-way interpretation, as it needs to be written both for the server API, the client connection, and the documentation. This is the grounds for slow unnecessary implementational work which allows for many errors due to communication inconsistencies. To solve this issue, the strategy of code generation is desired. This allows both client and server communication to be generated based on a common API documentation scheme. The responsibility is to ensure complete communication compliance while minimising manual code duplication. Every back-end endpoint should be clearly defined and documented to give a clean overview of the available functionality and possible error states that may occur. Interfaces and shared data entities between client and server should ideally only be defined once, and from there be automatically generated anywhere else it is needed. There should ideally only be one method of definition for both REST-endpoints and WebSocket endpoints.

The current Firebase solution is completely implemented in the front-end. However, it does have cloud functions defined which must be compliant with the client functions which calls them. This consistency is manually maintained by the developers.

Within the MealBuilder back-end, decorators are appended to controllers and endpoints. These provide the specification of the service. From these decorators a package called ”nestjs/swagger” [5] can generate an OpenAPI specifications document. This document can be read by a multitude of other services, which can present it for easy writing and generate front-end and back-end code from it. This allows for generating the front-end client automatically, avoiding implementation mistakes. The OpenAPI protocol [6] Allows for defining endpoints, attribute types, attribute ranges, endpoint errors, as well as other identifiers. The point is to convey the correct usage of a service in as simple a format as possible. The format is kept both human and machine readable to be as versatile as possible. The OpenAPI protocol is not yet compatible with streaming endpoints and is only meant for restful endpoints. As streaming endpoints is a requirement for this project another documentation must be found for the streaming endpoints. As a single documentation style is preferred for the project, it should be examined whether another documentation style, supporting stream endpoints, can also be used for code generation on par with the supported choices for OpenAPI specifications.

AsyncAPI [7] is a protocol which builds on the principles of OpenAPI to add support for asynchronous endpoints. It is supported by swagger code-gen which supports code generation for a long line of languages. However, AsyncAPI is designed specifically for Asynchronous endpoints and does not offer support for rest endpoints. This means, all communication, including CRUD requests, would have to be implemented in Asynchronous endpoints, or a combination of AsyncAPI and OpenAPI would have to be used. As Swagger supports both, the same code generation tool could be used for a combination of the documentation protocols.

gRPC is a WebSocket based transfer protocol designed by google, primarily to support fast communication between micro-services. It was not designed for front-end communication; however, it is one of the fastest protocols available. It relies on transferring binary encoded objects rather than text encoded objects such as JSON. This significantly reduces the size of messages and thereby increases the transfer speed. It only uses a single stream connection created in the beginning of a client correspondence, which reduces the connection overhead. It also has code-generation support for a long line of languages. Beyond this, there is the possibility for adding http support with JSON payload to the gRPC endpoints, which would allow non-gRPC clients to have access to the service [8]. These http endpoints could be described in an OpenAPI document to provide documentation for third party services [9].

The gRPC protocol seems to offer better performance and more flexibility than a combination of AsyncAPI and Swagger. However, it is also much further from the method that is currently used by LittleGiants. It must therefore be decided whether the benefits of gRPC warrants the risk of implementing a new technology. Should AsyncAPI be chosen, it must be

decided whether the service will be asynchronous only or if it will mix AsyncAPI with OpenAPI, which could also present complications.

5) Contact Information Validation
It is currently a requirement that the contact information of both users and their contacts are validated. This entails verifying that the owner of the contact information has access to the communication channel bound to the contact information, and that they accept the systems use of them.

It is determined that the current solutions within LittleGiants is sufficiently modular to be implemented with little risk of complications due to a new project structure. The implementation is therefore of little interest to the objective of this project and is not explored further

6) Contact Alerts
The system must have a way of alerting the contacts of a user when the user is experiencing a seizure.

The current solution fits the desired trades and is implementable in a long line of languages. It is not expected to provide any complications with a new project structure and is therefore not explored further.

7) Notification Management
The system must have a way of sending notifications to the client, both when the client application is active and when it is inactive. The system must be able to send information necessary for forming notifications on both Android and iOS which the client application supports.

It has been requested by the stakeholders that the current solution for notification management is maintained. This has been done in previous projects withing LittleGiants and is not expected to give any complications in a new project structure. It is therefore not implemented or explored further in this project.

C. Prioritisation

After the responsibility analysis it has been concluded that the most valuable and risk prone responsibilities of the new service is going to be Stream management, Code Generation, and Authentication prioritised in that order. The global requirements have been prioritised by how they implement these responsibilities as can be seen in Appendix A.

The authentication requirements have been included to demonstrate the responsibility area of Authentication Management. Profiles and Seizures have been chosen to demonstrate the responsibilities of Code Generation and Stream management. For the profile, creation and updating of the profile has been chosen as must requirements, as they allow state changes which will trigger the stream management. The same goes for Seizures, where creation and deletion has been chosen as must requirements. For the seizure stream, filtering on seizure attributes has also been included to demonstrate this capability.

D. Use-case Analysis

The use case analysis has the purpose of mapping the external system requirements to internal system requirements and concepts. This moves the focus from *what* the system needs to provide for the user to *how* the system intends to provide this functionality. The result is a list of abstract concepts which a system design can be based upon.

1) Method
The use case analysis builds upon the functional requirements identified and prioritised in the Requirements Analysis section. Select functional requirements are mapped to use cases. The system communicates with the user through endpoints and every use case is therefore an API endpoint.

The objective of the analysis is to identify the needed sequence of steps to fulfil the goal of the use case, as well as error states which can occur from misuse of the use case or from data corruption. This is collected in a use case specification table defining both the requirements that the use case fulfils, the goal of the use case, the sequence of steps to reach the goal, and the possible error states.

From the steps and error states an abstract model of the system can be created. This is used to identify system actions needed to fulfil the use case specification. The abstract model is built through sequence diagrams. The sequence diagrams are external, meaning, they model the system as a single unit, as the system components have yet to be identified. The model is therefore divided into the 3 known components - The user connection, the system, and the method of persistence called the “database”.

From the sequence diagrams four primary concepts can be identified:

- Internal processes of the system which either validates or creates data
- External database processes which either modifies or collects data from the database
- Error state messages sent to the user
- Abstract data models

These concepts are central to the development of the system structure. They each represent an Area of Responsibility (AoR) which must be fulfilled to provide the intended functionality. As each concept carry different responsibilities, they must be logically divided in the architecture of the system. This ensures easier implementation and maintainability.

Beyond concept identification of each use case, a union of the identified concepts can be created for each AoR. This leaves the entire list of abstract concepts for each AoR. From this an intersection of the concepts for the entire system can be made, which leaves a list of commonly used concepts which can be shared between different AoR’s. The table of concepts for each AoR as well as the execution sequence of each use case is the primary output of this analysis.

2) Results

The process resulted in 13 sequence diagrams – four to model the authentication endpoints, four to model the profile endpoints, and five to model the seizure endpoints. From each section of models a table of concepts was derived from the diagrams.

Figure 1 shows the sequence diagram of the endpoint for starting an authentication session. To identify concepts this was analysed in the following way. Messages from the system lifeline to the user lifeline was either "data objects" or "error types". Self-calling processes of the system lifeline were "internal processes". Calls from the system to the database lifeline were "database calls". Beyond this, data objects could be identified from the return types and arguments of processes.

E. Class Analysis

The output of the Use case analysis provided the required elements for a high-level overview of the project implementation. This was modelled with a Class diagram to ensure consistency in the implementation. It provided a better overview of shared resources, dependencies within the system, and the responsibilities of different components.

Other structural views were disregarded due to the low complexity of the system. Artefacts produced either need to feed the further implementation of the developed prototype system or be useful for later implementation of the full system. For these purposes, the class structure view was deemed sufficient.

Due to the separation between the authentication endpoints and the domain specific endpoints, they were modelled in their own class diagrams. This also enforces the desired separation between the AoR's, as any dependencies between them would warrant a complete reconfiguration of the modelled structure.

Further, the authentication responsibility was separated into two AoR's – Authentication and Authentication Sessions. This resulted in 4 distinct AoR's. Each AoR has the responsibility of communication, validation, manipulation, and persistence of its own data object. These responsibilities are divided into 3 classes, which is a controller class for communication with the client, a service class for data manipulation, and a persistence class for communication with the database. The validation is distributed across all classes.

Figure 2 shows the authentication class diagram, which models both the Authentication and Authentication Session AoR's. It shows one dependency from the *AuthService* to the *AuthSessionService* (D1) and one interdependency on the *AuthTokens* data model from the *AuthService* and *AuthSessionService* (D2).

The authentication controller has the need to create an authentication session. This could be done by implementing the persistence functionality in the *AuthPersistenceService*. However, this would mean fragmenting the responsibility for persistence of the *AuthSession* object. To avoid this, the dependency D1 was introduced.

Figure 3 shows the domain class diagram which models the Profile and Seizure AoR's. It can be seen that there are only interdependencies between the AoR's but no direct dependencies. The primary interdependency is on the *JwtService*, which is used by the service classes of both AoR's. This reflects the shared internal processes shown in Appendix B

F. Component Analysis

The output of the class structure analysis provided the needed information to create a component structure. This shows the interface connections between modules within the system, as well as actors outside of it.

Two distinct component diagrams have been made – one showing a monolithic system and one showing a distributed system. Due to the low interdependency of the system, multiple configurations are possible. The monolithic system is the easiest to implement, but the distributed system allows for plugging the authentication module into other systems.

Figure 4 shows the monolithic system. It contains four modules, each exposing their own interface which is exposed out of the system through a port. Each module also has a dependency on the same interface exposed by the database, which is outside the system boundary. The dependency between the *AuthModule* and the *AuthSessionModule* which was defined in the class structure analysis, is also modelled in the component diagram. It can be seen as a direct dependency without any mediaries.

Figure 5 shows the distributed system. It contains two systems, each containing two modules. Like in Figure 4, each module exposes an interface, and each module has a dependency on an interface exposed by an external database. The dependency between *AuthModule* and *AuthSessionModule* is here mediated by exposing ports and negotiated by an internal interface. This has an increased overhead, but the separation allows easier swapping of *AuthModule*, and potentially multiple *AuthModules* which can be introduced to the system without change of source code. The distributed system could be further decorated. The two systems could have separate databases to increase security. The exposed interfaces of the modules, which are currently exposed on two different ports, could be included in a load balancer to expose them over one port.

G. Design Decisions

This section clarifies the design decision which have been made for this project. They rely heavily upon the information gathered in the Responsibility Analysis as well as the questions which was raised during this analysis.

1) Tech Stack

The database used for all the projects of LittleGiants is MongoDB supported by Redis. MongoDB does not hinder any of the proposed technologies or strategies and the use of it is therefore not protested. Redis is not suspected to be utilised for this project but will be available in the production environment due to organisational convention within LittleGiants.

Based on the analysis and meetings with LittleGiants gRPC has been chosen as the stream method, which also offers code generation. It has been chosen as it is one of the most performant transfer protocols available and offers code generation for both streams and endpoints. It works over http 2.0, which makes it incompatible with web browsers, but configurations are offered for it to work over standard http 1.1 calls with JSON serialised payloads [10].

The use of JWT is widespread. It is used both by the current solution (Firebase) and within the most recent projects of LittleGiants. It also minimises the needed communication between the authentication management and domain services, as each token contains the user's authentication level. JWT will be used for the access token and will be protected with a refresh token.

The current preferred back-end framework at LittleGiants is NestJS which is based on Typescript. This framework does not seem to hinder any of the other chosen technologies and no other framework offers crucial advantages over NestJS. It is therefore chosen for consistency and reusability.

2) Service Distribution

Containerisation is a great method for increasing reusability of services. It would be possible to separate the authentication service into its own container. However, the current projects at LittleGiants are relatively small projects. The overhead of the container communication would most likely supersede the gained reusability from containerising their systems. The service will run in a container wherein support systems could be included in their own containers, but the boilerplate code will be kept in an isolated service within one container. However, the authentication and domain code will be separated as much as possible, and communication between them will be managed and limit. This should make it easy to separate the authentication code into its own service later.

3) Stream State Management

The chosen stream state management strategy is sending the entire updated element along with the type of CRUD change that has been performed. The client is therefore responsible for carrying out the change on the existing data. If all changes are performed, the client and back-end state will match.

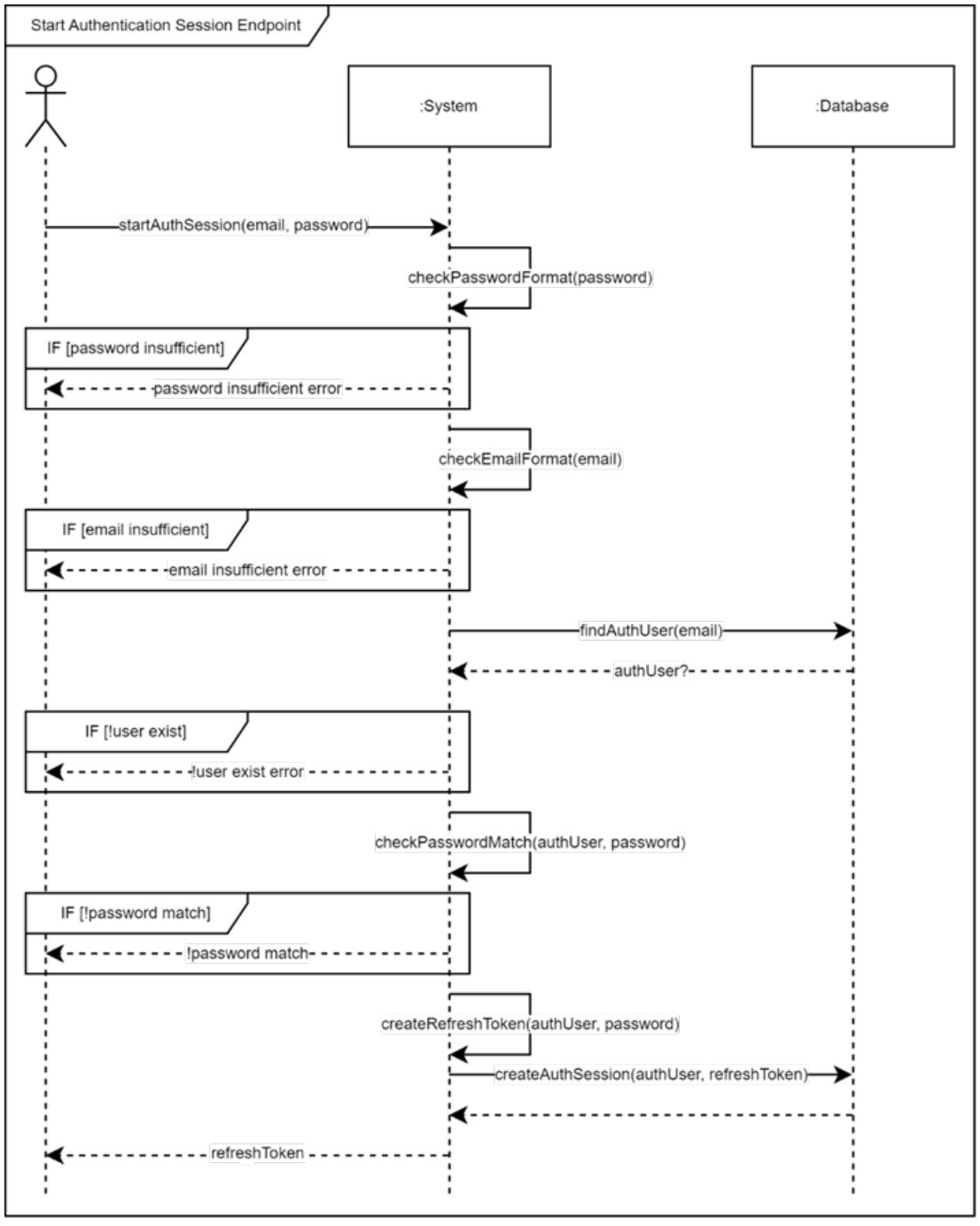


Fig. 1: Start authentication session endpoint sequence diagram

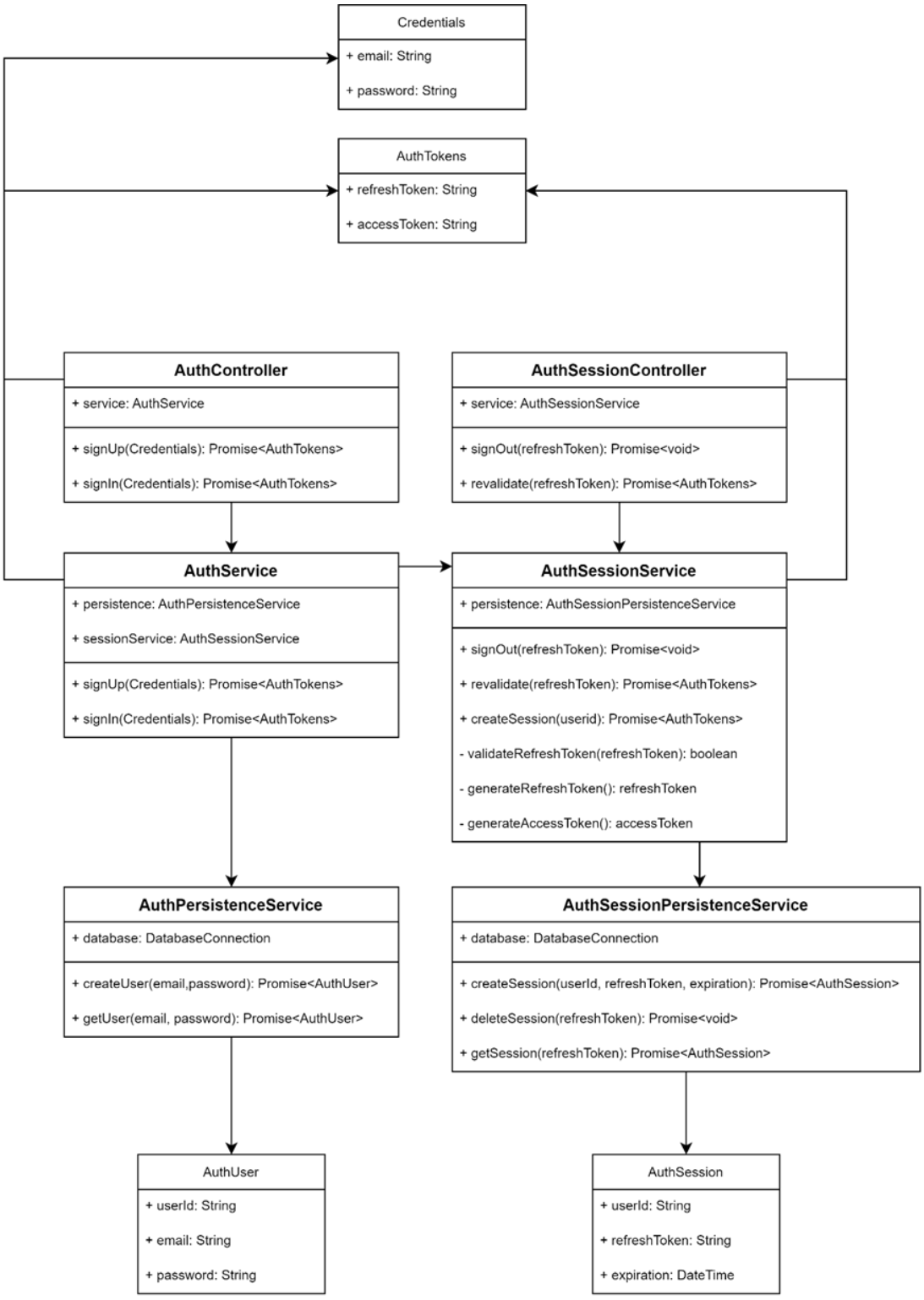


Fig. 2: Authentication Class Diagram

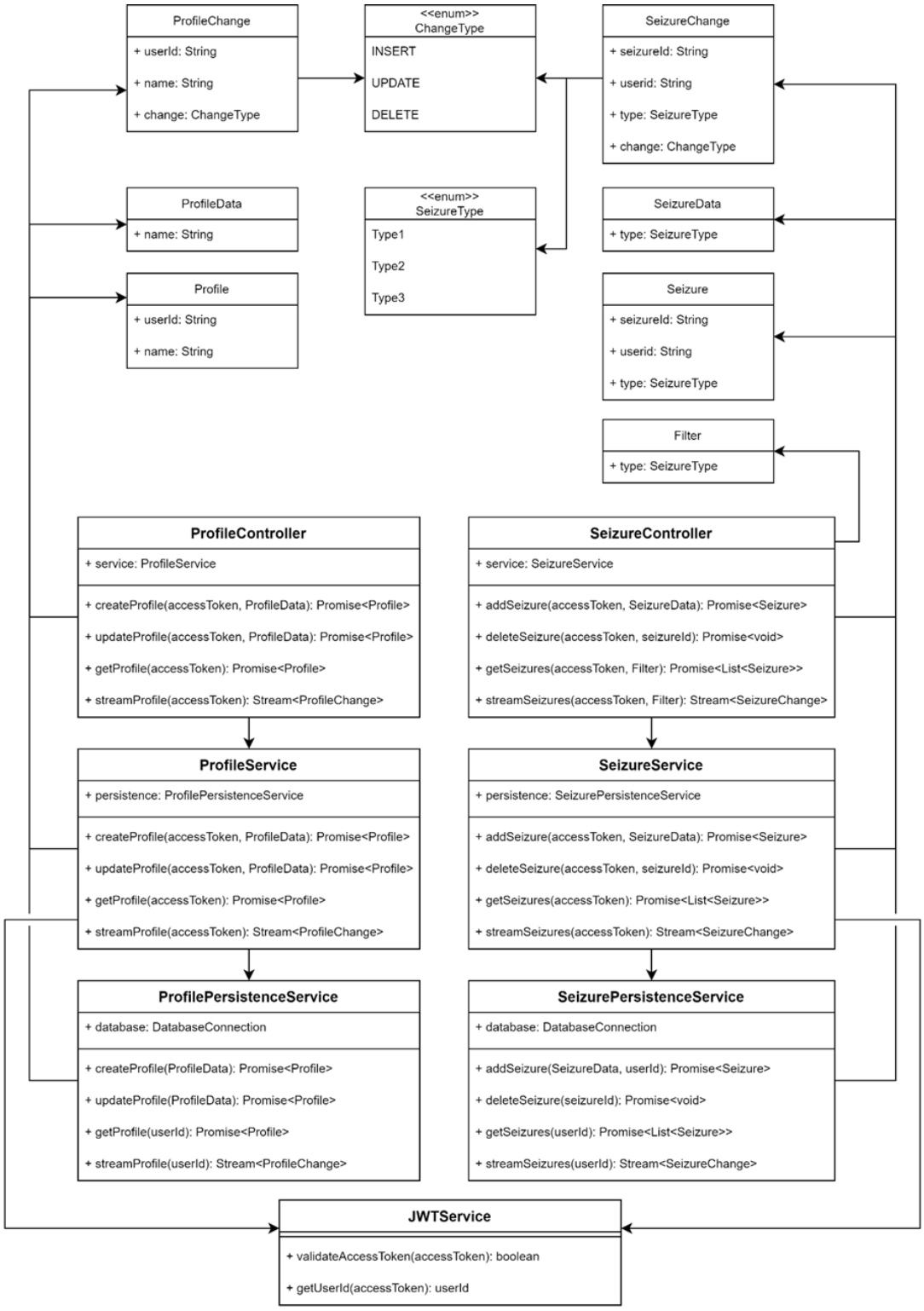


Fig. 3: Domain Class Diagram

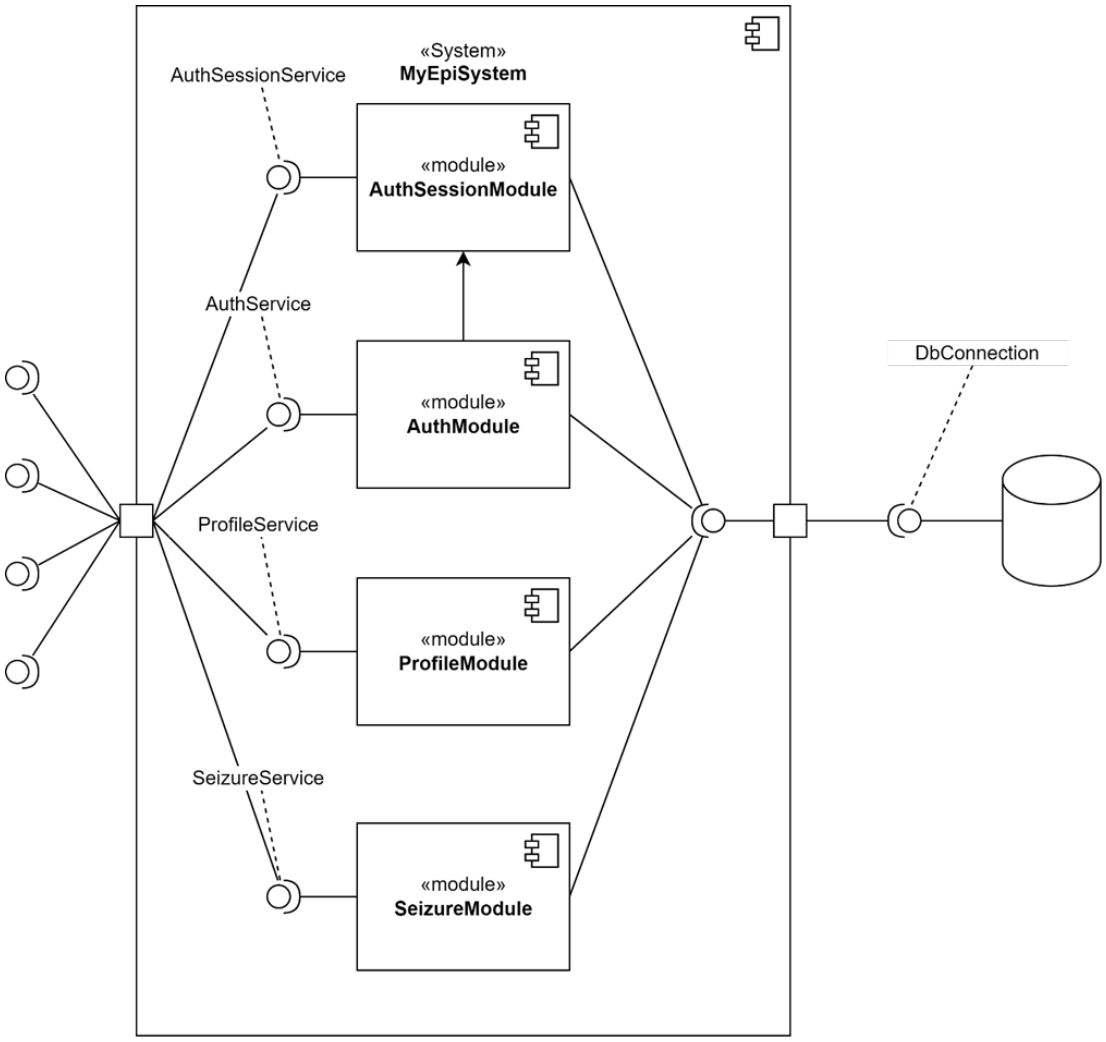


Fig. 4: Monolithic Component Diagram

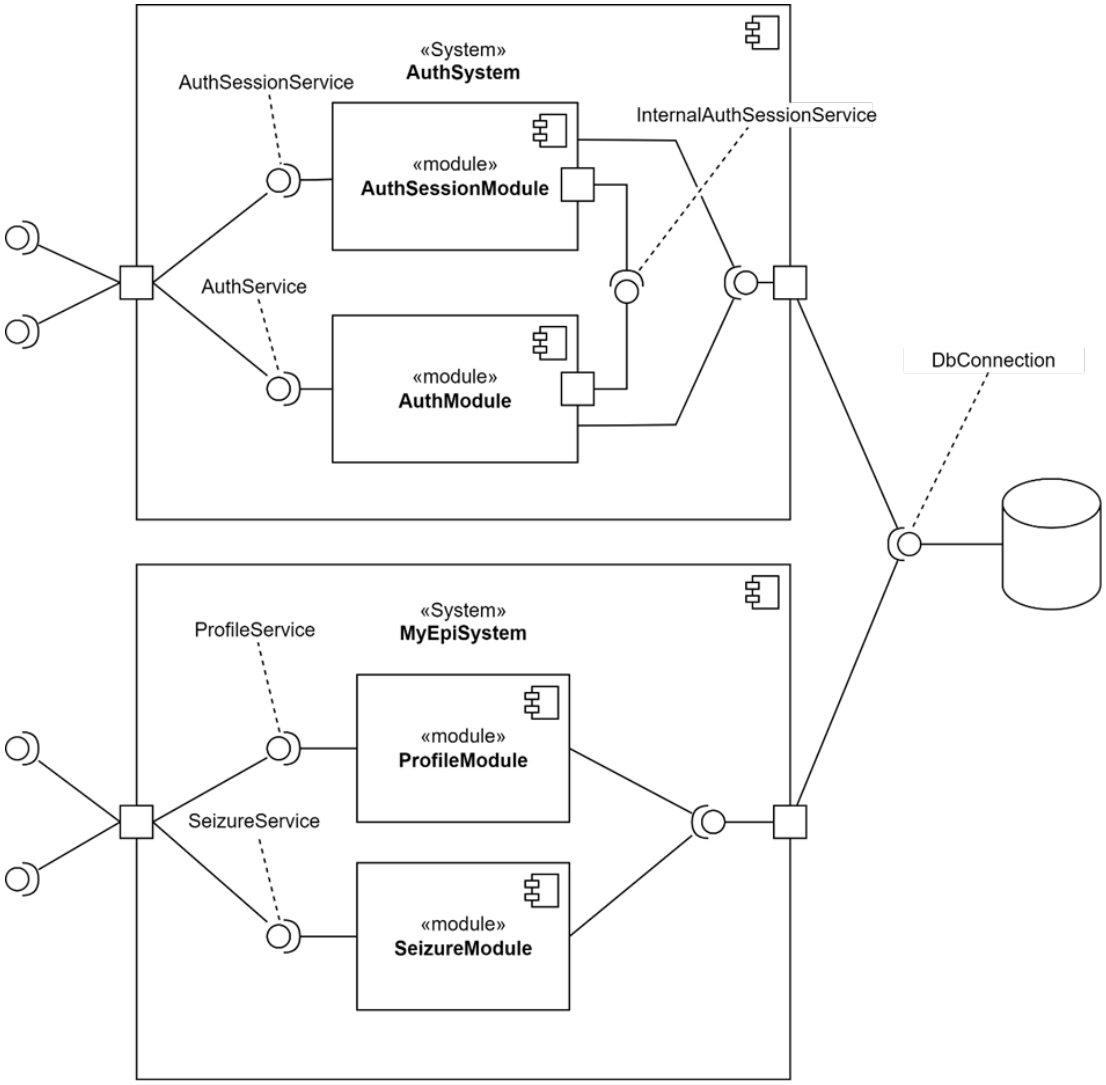


Fig. 5: Distributed Component Diagram

V. Implementation

This section is dedicated to give a clear understanding of the product which resulted from the documented design as well as the development process behind it.

A. Process

This section gives an overview of the sprints which were carried out during the main development phase. Each sprint included the creation of a module, unit tests of set module, and integration tests of set module. It did not, however, include performance testing, which was carried out after the functional verification of the entire project.

1) Sprint 1

Time period: 2023-03-06 till 2023-03-12 (Week 10)

The purpose of the sprint is to create an MVP prototype which can document the fundamental aspects of the tech-stack. These are the aspects which are necessary for the design to move forward and is agnostic of the domain. It should demonstrate the:

- Compatibility of gRPC and NestJS
- Compatibility of gRPC and Dart
- Code Generation within both NestJS and Dart
- Single call capabilities of gRPC
- Streaming capabilities of gRPC
- Streaming capabilities of MongoDB

The main objectives where to:

- Create a NestJS project
- Create a Proto file with a single call and a streaming point
- Auto-generate data objects and interfaces from a proto file within the NestJS project
- Implement endpoints for both single call and streaming, and streaming from MongoDB
- Create Dart client project
- Auto-generate data objects and interfaces from proto file within Dart project
- Create tests of endpoints within Dart project
- Create integration test file

By the end of the sprint a GitHub project was created and a NestJS project and a Dart project was setup within. Auto-generation worked however, it was not able to auto-generate NestJS modules. Instead, Typescript Interfaces were generated which can be implemented manually in the NestJS project. This is slightly more work than full auto-generation but gives the same type-safety for the endpoints, avoiding different implementations of client and server. The prototype successfully proved the core aspects of the project.

2) Sprint 2

Time period: 2023-03-13 till 2023-03-19 (Week 11)

The purpose of the sprint is to implement the session part of the authentication responsibilities in gRPC. This includes the setup of a complete Auth Session module with 2 endpoints. The endpoints will utilize only single calls and will not implement streaming for either gRPC or MongoDB. It will, however, depend on JWT for access token creation.

The main objectives where to:

- Create the module setup with a module file, a controller file, a service file, and a persistence file.
- Implement the necessary persistence functions
- Write unit tests for persistence functions
- Implement the necessary service functions
- Write unit tests for service functions
- Implement endpoints “signOut” and “revalidate” in the controller.
- Write integration tests for endpoints

By the end of the sprint the project was successfully updated with the new Auth Session module and all functions were successfully unit tested. However, the integration test was not possible as the state of the service depends on the authentication endpoints for the auth session endpoints to be tested.

3) Sprint 3

Time period: 2023-03-20 till 2023-03-26 (Week 12)

The purpose of the sprint is to implement the auth part of the authentication responsibilities in gRPC. This includes the setup of a complete Auth module with 2 endpoints. Like the Auth Session module, the endpoints will utilize only single calls and will not implement streaming. It will not depend on JWT for token creation; however, it will depend on Crypto for password hashing.

The main objectives where to:

- Create the module setup with a module file, a controller file, a service file, and a persistence file.
- Implement the necessary persistence functions
- Write unit tests for persistence functions
- Implement the necessary service functions
- Write unit tests for service functions
- Implement endpoints “signUp” and “signIn” in the controller.
- Write integration tests for endpoints

By the end of the sprint the project had been successfully updated with the new Auth module and all functions were successfully unit tested. However, the integration test uncovered a problem with the implementation of Promise call-back which had not been tested by unit tests, as it was written in the controller. By moving the logic to the controller, it was able to be unit tested which quickly uncovered the root problem.

4) Sprint 4

Time period: 2023-04-03 till 2023-04-09 (Week 14)

The purpose of this sprint is to setup the profile module. This includes functionality for creating, updating, fetching,

and listening for changes to a profile associated with a user's unique auth user id. The module will introduce two new concepts to the project, namely access Token decoding and streaming. However, streaming was prototyped in the first sprint and the implementation is expected to be eased by this.

The main objectives where to:

- Create the module setup with a module file, a controller file, a service file, and a persistence file.
- Implement the necessary persistence functions
- Write unit tests for persistence functions
- Implement the necessary service functions
- Write unit tests for service functions
- Implement endpoints "create"," update", "get" and "listen in the controller.
- Write integration tests for endpoints

The use of streams in Mongo requires the Mongo database to be sharded or replicated and be connected with a read concern of Majority. This had the unexpected effect of introducing a lag in the database updates. As a result, a database update followed by an immediate fetch of set data would return without the update. The tests of the rest of the project therefor failed as the read concern was changed from "local" to "majority". To solve the testing problems, artificial delays were injected in between the update and fetching of data. The module also introduced the need for input data validation, and a fitting solution needed to be found. This solution needed to be easy to replicate, implement, and be easy to identify when glossing over endpoints. The best solution was identified as "Pipes" [11] which can intercept the input value of a function and throw an error if it does not meet requirements. Per the NestJS documentation, Pipes should work with RPC connections, but it was not possible to enable them. A temporary and sub-optimal solution was therefore implemented. This places the validation logic in line with the endpoint logic, making it more difficult to ensure correct validation when glossing over endpoints.

5) Sprint 5
Time period: 2023-04-10 till 2023-04-16 (Week 15)

The purpose of this sprint is to setup the seizure module. This includes functionality for creating, updating, fetching, and listening for changes to seizures associated with a user's unique "auth user id. The sprint introduces filtering of seizures, both when fetching and listening for change.

The main objectives where to:

- Create the module setup with a module file, a controller file, a service file, and a persistence file.
- Implement the necessary persistence functions
- Write unit tests for persistence functions
- Implement the necessary service functions
- Write unit tests for service functions
- Implement endpoints "create"," update", "get" and "listen in the controller.
- Write integration tests for endpoints

The profile service included Mongo streams, however, it did not include a delete stream, as there is no delete endpoint. The seizure stream does include a delete endpoint, which exposed a problem with the Mongo streams. The change events from the Mongo stream return the full document that was changed on insertion and updating documents, but not when deleting documents. Here you only have access to the document id. As the seizure document contain a reference to the "userId", and a user should only listen for seizures containing their "userId", filtering was not possible. A workaround was implemented, which included a "deleted" flag attribute in the seizure document. On deletion, this flag would first be updated, triggering an update event in the Mongo stream. If the flag was set to true, a Delete event could be returned to the user. According to Mongo documentation, it appears that the option for returning the deleted document is possible, but it requires extensive configuration of the database and only works on Mongo version 6.0. As the project currently uses 5.2, there is the possibility for utilising the correct method in the future, but for now, the workaround is the best solution.

B. Unit Testing

The modules are set up with a controller containing the endpoints, a service containing the business logic, and a persistence service containing the connection with the database. The controller contains most of the error state handling, implemented as guards and inline validation. These endpoints could be unit tested with a more complex setup; however, those tests would be identical to the integration tests and were therefore not implemented. The endpoint validation is therefore tested through the integration testing. Any error states that could get passed to the endpoints should therefore be handled in the business logic and persistence layer. These handling methods are tested with unit tests for each module.

```
Test Suites: 8 passed, 8 total
Tests:       48 passed, 48 total
Snapshots:   0 total
Time:        9.216 s
Ran all test suites.
```

Fig. 6: Unit test output

Figure 6 shows a complete run of the unit tests in the system. It can be seen that there are 8 test files. The test files each has a very similar running time, suggesting that the majority of the time consumption is due to setup and tear-down, which is identical across the test files. It can also be seen that the files are run in parallel, dramatically reducing the testing time.

The finished tests serve as documentation for the project, as it shows the outcomes which can be expected in different scenarios. Beyond this, the tests have been the main driver of bug fixing. In the development process, an entire service file was written, where after its entire testing file was written. The testing file was then used to fix the errors in the

implementation. This is not strictly test driven development, but it shares the focus on function outputs rather than function implementation for validation and development.

C. Integration Testing

The integration testing is meant to test the functionality which is specific to the endpoints. This means, ensuring that all endpoint attributes and error states are correctly conveyed to the client. The purpose is not to revalidate the business logic of the endpoints, and so, not all system states are tested. The integration testing is done with a testing client written in dart. This was done to mimic the conditions in which the service is going to be used. The test client has an auto-generated connection created from the same proto file as the API. Each endpoint is tested for the different outputs it can return.

The unit tests are tested with an in memory Mongo database. The integration tests are tested with a MongoDB docker container. This should mimic the working conditions of the service more closely. Both the client, the API, and the Mongo database are run in containers, defined in a docker-compose file. Due to timing issues, a bash and bat script were made to setup the compose file.

```
bachelor-project-api-client-1  | 00:16 +31: All tests passed!
bachelor-project-api-client-1  |
bachelor-project-api-client-1 exited with code 0
```

Fig. 7: Integration test output

Figure 7 shows a complete run of the integration tests. It can be seen that there are 35 integration tests, and they were completed in 16 seconds. However, for each test the services are removed, rebuild, and spun up, and the database volumes are deleted. A hot run of this process therefore takes 42 seconds on the machine used.

D. Performance Testing

The nonfunctional requirement NF2 in Appendix A states that a service must be able to handle 500 users each making one modification per second. To validate this, a performance test was setup. This uses the same client and setup as the integration tests but focuses on the delay times of the service.

The test setup used creates each client in a separate process called and Isolate [2]. Each client creates a separate connection to the service. Each client will subscribe to the *Seizure* stream endpoint which will emit a *SeizureChange* event every time a new *Seizure* is created or deleted. The client will then proceed to emit 10 *Seizure* messages with a random frequency between 500 and 1500 milliseconds. This is to avoid every client sending their message at the exact same moment. Each client will measure the time laps from a *Seizure* creation method is sent till a *SeizureChange* event is received. After the last message is sent, the process will return the best, worst, and average timing from the 10 messages given in milliseconds. The testing unit will then calculate a total average, as well as find the overall best and worst timings from the clients.

The testing unit is setup to run 25 iterations of a test with a given number of clients. Running multiple iterations allows for monitoring of changes in performance as clients are created and removed. Running different number of clients allows for detecting sharp changes in performance between different loads.

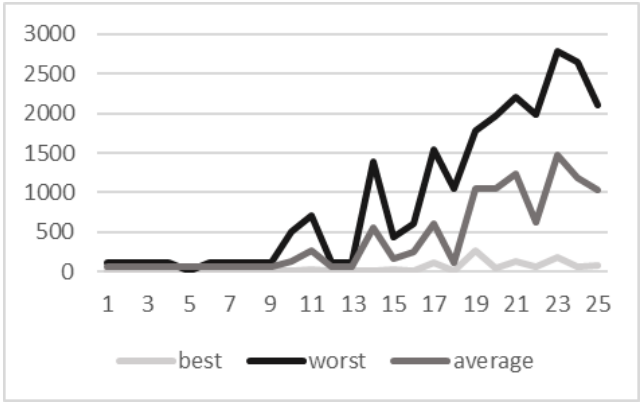


Fig. 8: Performance test - 100 clients, pool size 100, no modificiations

Figure 8 is one of the first tests runs before any modifications. It clearly shows an uneven performance degradation as the test progresses. This indicates a resources leak. This was traced back to the failed assumption that all connected streams would automatically shut down upon a client disconnecting. Due to the use of *Observable* as the return type, there is no way of detecting when the client has disconnected, as only the publisher has power over the connection but not the subscriber. This means that the processes started for a streaming endpoint will continue to run after a client shuts down. No clear solution for this issue could be found, and a temporary solution was therefore implemented. The input for the *Seizure* stream endpoint was changed from a *SeizureFilter* object to a *SeizureFilter* stream object. This allowed the service to listen for a disconnect message of the client. Thereby, the service can dispose of resources when a client disconnects. It is not an elegant solution, and it is presumed that an alternative strategy is available and built in to gRPC, but it is not certain whether this is available in the NestJS implementation of gRPC.

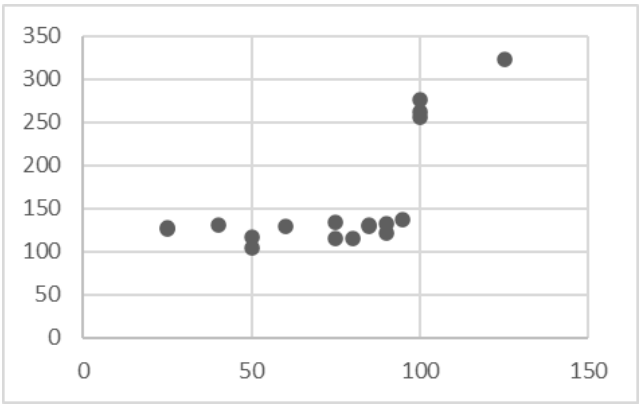


Fig. 9: Performance test - single iteration, variable client numbers

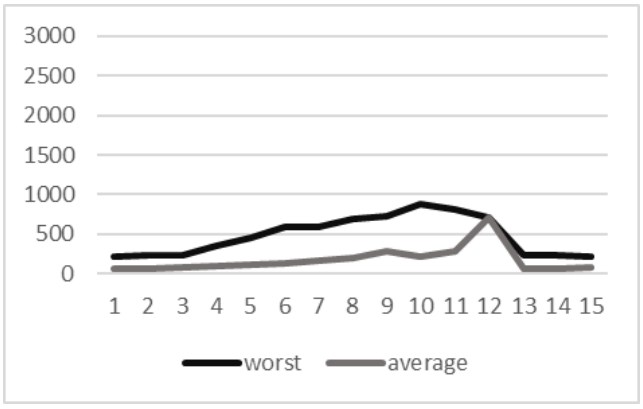


Fig. 11: Performance test - 100 clients, pool size 2000, ping modificiation

Figure 9 shows an alternative performance test. Each test was run with a single iteration, restarting the service after each. Here, focus was on performance change as a function of clients. The test showed a sharp change in performance for runs with over 100 clients. Upon inspecting the service, it was seen that the connections to the database never rose above 102, despite client numbers going above 102. The Connections also never fell below three despite there being no connected clients. This indicated a connection limitation on the database. This was found to be due to the concept of *Connection Pools* [12]. This is a way of managing and optimising when connections are created and removed from the database. The default cap is set to 100 (presumably with two extra connections for global database management).

Figure 10 and Figure 11 shows the two tests which were run to examin the effect of changing the pool size. Both tests were run for 15 iterations. Figure 10 shows the test with the default pool size and Figure 11 shows the test with an extreme increase in pool size. The tests show a clear improvement after increasing the pool size both in the overall performance and the uniformity of the performance.

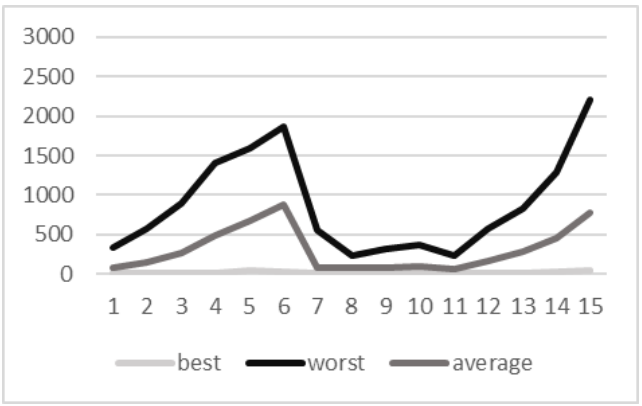


Fig. 10: Performance test - 100 clients, pool size 100, ping modification

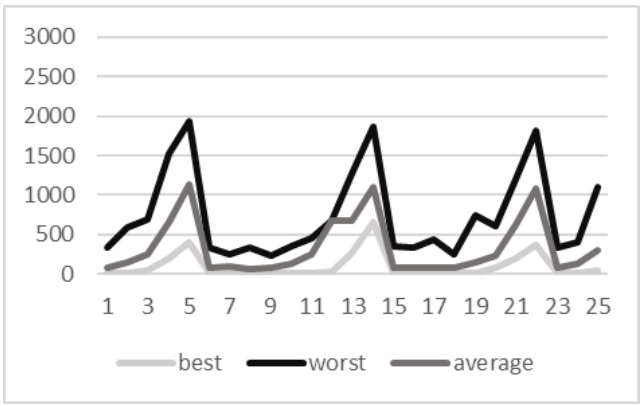


Fig. 12: Performance test - 200 clients, pool size 550, ping modification

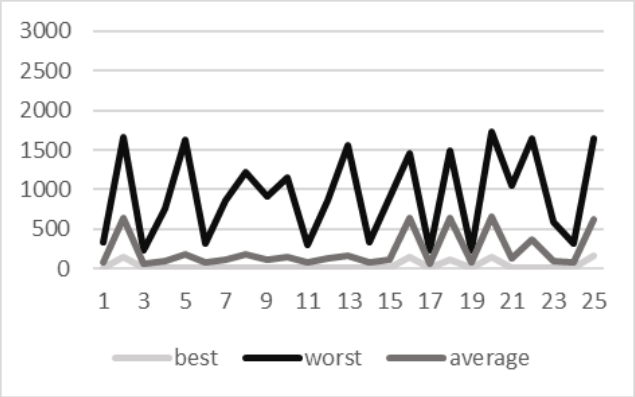


Fig. 13: Performance test - 200 clients, pool size 250, ping modification

To fine tune the pool size as well as look for further performance limitations and irregularities, additional scenarios were tested. Figure 12 and Figure 13 shows that increasing the connection pool also has a detrimental effect. After analysing the database, it could be seen that connections are opened as clients are added. However, connections not in use are only purged when the connection pool reaches its limit. It can be seen in Figure 12 that a high number of connections reduces the performance. The best number of connections is therefore the exact number of clients. The management of the pool can be further adjusted with parameters on the MongoDB driver used in the service [12].

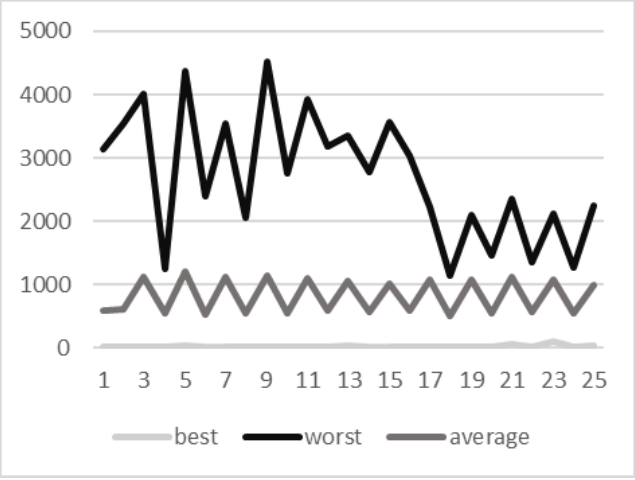


Fig. 14: Performance test - 500 clients, pool size 750, ping modification

Figure 14 shows the modified service running with 500 clients. It can handle the load but with unimpressive results. The average response time is 800 milliseconds. All services are run on a local machine, and network timings therefore must be added to this for production. Beyond this, there are timings of up to 4 seconds, which would cause frustration with many users [13]. To tackle this issue, an alternative tactic was therefore tested. Instead of creating a database stream for each user, a single database stream was created for the service listening on the *Seizure* collection. Client connections could then listen on this stream internally in the service and apply a custom filter.

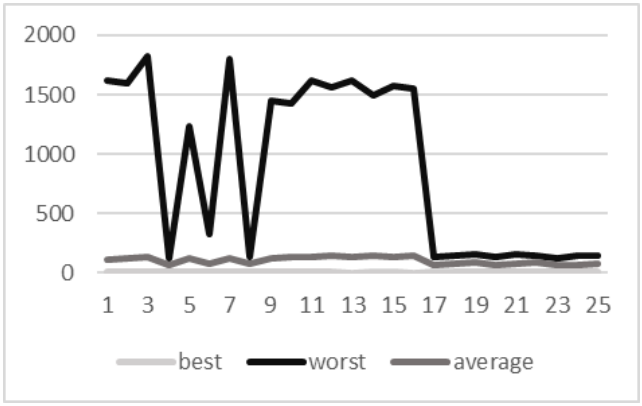


Fig. 15: Performance test - 500 clients, pool size 750, ping modification, global stream modification

Figure 15 shows that the global stream tactic had a drastic improvement. The new average timing fell to 102.5 milliseconds, and the worst timings fell to under 2 seconds. The strategy means that each service instance will receive all *Seizure* events and must filter these before returning them to the clients. Once the client number becomes large enough, this tactic must be modified. However, this is not expected to be a problem for MyEpi as of now. It simply needs to be kept in mind and monitored. Furthermore, the drop off in worst timings, as seen at iteration 17 on Figure 15, could not be explained. However, the test was run multiple times and the pattern repeats itself. It is therefore presumed that either MongoDB or the Mongo driver has a number of startup processes which needs to run their course before the optimal performance is achieved. Either way, it is suspected that the last iterations of the test is representative for real world usage with occasional increases in worst timings.

E. End Product

This section will give an overview of the resulting system from development process. It will start with a general overview, and thereafter dive into solutions and compromises which were made in the system.

1) Service Endpoint Overview

The functionality of the product is defined by the resulting endpoints. This is documented in the proto file of the project. The file is used to generate endpoints in the back-end service and to generate a client connection in the front-end. This ensures that the client and service match. LittleGiants currently uses OpenAPI to document their endpoints which contains more detail, such as possible error codes and allowed value ranges. However, there are tools for converting proto files into OpenAPI for further documentation if that is desired [14].

```
service AuthService:
    function SignUp(Credentials) returns AuthTokens;
    function SignIn(Credentials) returns AuthTokens;

service AuthSessionService:
    function SignOut();
    function RefreshAccessToken() returns AuthTokens;

service ProfileService:
    function create(Profile);
    function update(Profile);
    function get() returns Profile;
    function stream(stream of void) returns stream of Profile;

service SeizureService:
    function create(Seizure);
    function update(Seizure);
    function get(Filter) returns List of Seizures;
    function stream(stream of Filters)
        returns stream of Seizures;
```

API 1: Service endpoints

There are two major differences between the implemented endpoints and the interfaces which were defined in Figure 2 and Figure 3. This is the decision to not return the created object form the *Create* endpoints and the decision to make the *Stream* endpoints accept streams instead of objects. The decision not to return objects from *Create* was made to "push" the use of the implemented streams. If the client relies on state changes from multiple sources, they are more likely to implement a faulty state management system. State changes should therefore solely come from the stream method. The get method should be seen as a state of its own and should not be used to update a maintained state in the client system. The decision to accept streams from the *Stream* endpoints was described in the Performance Test section.

2) Code Structure Overview

The code repository is divided into two projects – the service, which is the product, and the client, which is the integration and performance testing module.

The service contains meta files in its root, including the proto file defining the endpoints. The source folder contains the source code, which is divided into 5 modules which is the 4 main modules (auth, auth-session, profile, seizure) and a common module. The common folder contains shared errors, validation logic, and testing logic. It also contains 3 services – a gRPC service for extracting auth tokens, a JWT service for encoding, validating, and decoding tokens, and a Mongo service for injecting Mongo collections.

Each main module contains 4 root files and up to 4 folders. The root files are:

- **Module**: defining and configuring the imports and exports of the module
- **Controller**: implementing the endpoints and handling gRPC logic
- **Service**: containing business logic and data validation logic
- **Persistence**: containing communication logic with the database

The folders are guards, containing endpoint validation classes, tests, containing unit test files, types, containing non-gRPC entities, and errors, containing both standard and gRPC specific errors.

3) gRPC Code Generation

Code generation was one of the requested features of this system. This was achieved by using the gRPC protocol along with a .proto endpoint definition. The specific code generation tool used is the *protoc-gen-ts* [15]. This has wide support and compiles into human readable Typescript files. This allows developers to understand the code from its source rather than through documentation. The code gen tool produces plain Typescript enums and types from the proto definition and interfaces for the services. It also produces a service decorator which implements the needed decorator configuration of the NestJS Framework.

4) gRPC Object Separation

As stated in the introduction the company maintaining the system tends to experiment with different technologies. It was therefore seen as important to keep a good separation between the business logic and the used technologies. One of these technologies was the gRPC protocol and the generated code which could be replaced in a future version.

One method of separation was being mindful about what logic went into the controller files. These should only contain logic related to the gRPC protocol. In this way, the transport layer could easily be switched out. The second method was the explosion of gRPC generated objects. These were particularly important to isolate, as the generative tool is community supported, and a replacement could therefore be necessary in future versions. Generated objects were therefore kept in the controllers and mapped to objects used in the service and persistence layer, which had no ties to specific technologies.

5) Endpoint Context Documentation

The documentation for a service needs to provide the entire context needed to operate the service efficiently. An important part of this is the error states which can occur when calling an endpoint. These states will either need to be avoided or handle by the client. The endpoint documentation currently used by LittleGiants is OpenAPI which does provide a way of documenting possible error states for each endpoint. They also provide ways of defining the accepted range of values that an endpoint accepts. The documentation is auto-generated from their service by adding annotations to the endpoints.

Vanilla proto has limited nuance in its schema definition which is a conscious decision to keep the language simple and lightweight [16]. It therefore has no standard way to define custom error states or value ranges. There is multiple community extension which attempt to add these features, but it is difficult to evaluate their stability.

The implemented solution was providing an enum for each service with the possible error states. However, this does not document which endpoint might throw which error. The current documentation therefore requires some context and is not complete. Further context could be given by having an error state enum for each endpoint or by using comments in the proto file.

It is possible that the current solution with OpenAPI could work with gRPC, adding the same decorators to the gRPC endpoints. In this way, an OpenAPI documentation could be generated, with a more complete context for the provided service. However, this would require that the proto file and OpenAPI decorators are both kept up to date, which could result in inconsistencies.

6) Input Validation System

The gRPC protocol can be used to define input value types as each endpoint accepts exactly one predefined data object. However, as mentioned in the Endpoint Context Documentation section the proto protocol definition does not allow defining value ranges. A method for validating input data therefore had to be implemented.

It was expressed by stakeholders that it was preferred to have all validation statements implemented as decorators on endpoints. This has the aim of keeping the endpoint logic clean and simple to read, to make correct configuration easy to double check. The NestJS framework has a method for this called Pipes [11]. These allow for defining a middle-ware function which receives the data before the actual endpoint function. This middle-ware function can then validate and reject the data with a given error message.

In the NestJS documentation it is stated that Pipes should work identically for REST based and RPC based projects [17]. They have therefore not provided much RPC specific documentation on pipes. However, when employed, the principles did not seem to work. Either the documentation is lacking necessary information, or the framework version used has a bug for this particular concept. If the first suspicion holds true, further experimentation and web searching should give a working solution. If the second suspicion holds true, a future fix is suspected. However, as validation decorators are not strictly necessary for the functionality of the service a different solution was found instead.

The data validation logic has been defined in functions in the service files of each module. These functions validate the data and throws a descriptive error upon rejection. The functions are called directly in the endpoint functions rather than before their execution. The solution is not as visible as using decorators but still keeps the endpoint flow unobscured and relatively easy to understand.

```
@UseGuards(HasNoProfileGuard)
create(
  request: Profile,
  metadata: Metadata,
): Void | Promise<Void> | Observable<Void> {
  return new Promise<Void>(async (resolve, reject) => {
    try {
      this.service.validation(request.name);

      const accessToken = this.grpcService.extractToken(metadata);

      await this.service.create(accessToken, request.name);

      resolve({});
    } catch (e) {
      if (e.message === new ProfileDataInsufficientError().message) {
        reject(new ProfileDataInsufficientGrpcError());
      } else {
        reject(new ProfileInternalGrpcError());
      }
    }
  });
}
```

Syntax 1: Create profile endpoint

```
IF user has no profile:
    RETURN descriptive error;
ELSE:
    ASYNC:
        TRY:
            Validate input parameter;
            Extract access token from metadata;
            Create profile with given data;
            RETURN;
        CATCH:
            IF data insufficient:
                RETURN descriptive error;
            ELSE:
                RETURN generic error;
```

Pseudo code 1: Create profile endpoint

Syntax 1 shows the general setup for endpoints. The first line shows the use of guards, which are able to validate the call based on metadata. As auth tokens are placed in metadata, the user's auth status can be validated here. As the auth user id is placed in the auth token, validation based on the user's profile can also be implemented here. On line 3 the proto input type validation can be seen, as the input of the endpoint called request is cast to the type of Profile, which is an auto generated data class based on the proto definition. On Line 8, the call to the validation function on the service class can be seen. This is where the data value range can be validated.

Pseudo code 1 shows the flow of the endpoint in plain text.

7) Mongo Stream Management

The most central aspect to this service is the implementation of real time streaming as this is the prominent feature of the Firebase service it is replacing. Real time streaming simply moves a part of the state management from the client to the service. However, state management still needs to be handled and this can be done in many ways. It is desired to generalise this procedure to avoid having to write it for every state variable. This saves time but more importantly avoids

implementational errors.

The utilised database, MongoDB, already has a state management utility baked in, called watch [18]. By using the watch functionality on a collection, you get notified on changes to the collection and can react to these changes by informing relevant clients. By utilising this functionality, a great deal of responsibility is removed from the implementation of the service.

To enable the watch functionality the Mongo instance used must either be sharded [19] or replicated [20]. This is because the watch functionality depends on an *oplog* [21]. The *oplog* registers all changes to share these between multiple Mongo instances. Beyond that the Read Concern [22] must be set to "majority". This means, when communicating with the database, the "truth" of the database cluster is the state most widely held within the cluster. This is opposed to "local" which accepts the state of the database instance you have direct contact with.

As assured by stakeholders, the database used in the final service would be both sharded and replicated, and so the configuration was not a problem. However, the watch function needed to have access to the data of the affected document to filter which clients needed to be updated. This data was available when creating or updating a document, but not when deleting a document. The documentation does explain a method for getting the deleted document [23], but it seems this method is very new and not compatible with the versions of Mongo used. A workaround therefore had to be found.

The solution was adding a Boolean field to all document schema's. On creation this would be set to "false". Just before deletion, the document would be updated, and the field would be set to "true". The update would trigger the watch function. Upon an update event, the call-back checks the Boolean and sends an update event to the client if the Boolean is false and a delete event if the Boolean is true. The workaround has the drawback of increasing complexity of both documents, the delete function, and the watch call-back. It also requires an extra call to the database on deletion with an extra resource cost. However, the drawbacks are estimated to be acceptable. They are also presumed to be temporary as the solution mentioned in the documentation could be implemented in future versions.

As discovered in the performance test the Mongo change streams must be handled correctly to perform efficiently. To this end, change streams are created per service instead of per client. Each stream returns all events on the collection for all clients. When a client subscribes to a stream, a subscription is made within the service to the global Mongo change stream, and the events relevant to the client is returned to them. The solution proved efficient for a number of 500 active clients on one service but it is expected to cause problems as the active client number grows. For a scenario of 10.000 active users distributed on 20 service instances, the received change events from the Mongo server would be 20 times higher than what is needed. This is due to the fact that events for a single client on a single service instance will be sent to all 20 service instances.

VI. Conclusion

A. Conclusion

The basis of this project was to address the inflexibility of Firebase, which was used as the back-end support for an existing front-end application. The goal was to develop and demonstrate a template design which would serve as a foundation for developing a replacement service.The main focus of the template was to address the most uncertain features and provide standardised solution patterns for handling repetitive logic efficiently. While the project sought to optimise the product attributes it also sought to remain aligned with the products already developed and maintained by the organisation responsible for implementing the complete service.

To identify repetitive logic, functional requirements were elicited from the current front-end project, providing an overview of the required feature types. To identify the most uncertain features, a list of central responsibilities were elicited from the functional requirements. These were analysed for their desired properties and compared with properties offered by existing solutions. The disparity between the desired properties and the offered properties showed the attainable value of each responsibility. Based on the comparison, the selected responsibilities were Stream Management, Code Generation, and Authentication. Thereafter functional requirements were selected based on their coverage of identified feature types and their ability to demonstrate the selected responsibilities.

Following standard software methodologies, the selected requirements were extrapolated into broader use-cases. From there, a concrete product design could be developed to serve as the basis for the implementation. Based on information attained from stakeholder meetings, the existing project, and information gathered during the responsibility analysis, a tech stack was selected. The tech-stack was primarily based on technologies already used within the involved organisation. The primary deviance was the implementation of a new transport protocol - gRPC. This had the promise of broad code generation, rapid transfer speeds, and the ability to offer both static endpoints as well as stream endpoints.

During the implementation a list of central assumptions were verified - the compatibility of the chosen tech-stack, the ability to generate code with both the chosen framework and the front-end programming language, and the ability to communicate between the chosen framework and the front-end project programming language with the chosen transport protocol. This provided the bare minimum validation for proceeding with the implementation.

To prove the implementation of the business logic for the functional requirements a suite of unit tests was developed during the implementation. This covers the possible states of each function call from the endpoints. It documents the intended behaviour of each business logic function to ensure compliance with the identified functional requirements. Furthermore, to prove the usability of the endpoints, an integration test suite was developed. This proves the correct actuation of business logic as an endpoint is called as well as the correct return of data and error states.

Authentication was selected as a central responsibility both because of its overall importance and because of the potential need for a different approach compared to previous projects. However, during the implementation it was discovered that the chosen tech-stack supported a nearly identical approach to what had already been used in the involved organisation.

Code Generation was another central responsibility as requested from stakeholder meetings. This was achieved through the chosen transport protocol as it offered native support for code generation in an array of programming languages. The currently used method in the involved organisation only offers code generation for static endpoints. The implemented method improves on this by also offering code generation for stream endpoints. However, the used method does limit the nuance of information which can be conveyed in the generated documentation, as the method does not offer native support for endpoint errors or input range documentation.

Stream management was the most important responsibility as it is the central selling point of the currently used Firebase service. The implemented method demonstrates implicit real-time filtered stream management. However, the project only covers stream management in the back-end service and a client stream management approach still needs to be developed.

The resulting product contains a selective array of endpoints which demonstrate the key feature types needed in the final product. It proves the viability of the selected tech-stack and includes the selected central responsibilities. It closely builds upon the existing projects within the involved organisation, and restricts its deviation to areas with a strongly presumed improvement in desired qualities. It provides a solid foundation for the implementation of a complete replacement service.

B. Discussion and Perspective

Though the methodology used in the project is sound, an error was found in the sequence. The performance testing was performed at the end of the implementation after all functionalities had been implemented and tested. It revealed deep seeded structural errors which had to be rectified for all occurrences. This resulted in re-work and re-testing of two out of five modules in the project. Instead, another sprint should have followed the initial prototype sprint to test the performance of the streaming implementation. This would have saved time throughout the implementation process.

Furthermore, the performance tests have shown a list of parameters which can significantly affect the performance of the product. It is suggested that additional testing is done in production environments before further development takes place. This should ensure that the results of this paper remain valid as the active user count grows.

To increase the value offered by the resulting product there are

five main improvements which could be implemented in future iterations: data stream optimisation, Separation of Authentication into isolated services, improvement of validation logic for readability, improvement of documentation, extraction of logic patterns.

The data stream is currently broadcasting all data changes to every service, which is going to yield an exponential data consumption as a function of active users. Implementing a method for filtering the data before it is broadcast to the endpoint services is assumed necessary for the long term viability of the proposed solution as active user count increases.

The authentication responsibility has been separated into two distinct responsibilities - authentication handling and session handling. Session handling could be reused regardless of the implemented authentication method and authentication methods could be reused across multiple project. To reduce the work of implementing these elements into new projects they could be separated into micro-services. This would allow reuse without meddling with the source code.

The validation of the input parameters is currently highly manual and prone to incorrect implementation. Ideally, a more implicit method should be found to make input guards easier to identify, thereby reducing the risk of bugs caused by validation logic.

The current documentation of the implemented service is the gRPC specifications document. This lacks the required information for a complete implementation of the service. There are multiple methods upon which the total information of the service could be conveyed, hereunder comments in the gRPC specification or the use of OpenAPI along side the gRPC specification. A method should be chosen to provide clear information on the use of the service and the possible states it can enter.

The code contains multiple reused patterns, especially in the stream endpoint data flow. These patterns should be extracted to increase code reuse and decrease the source of bugs.

The primary technology of interest in this project has been the use of gRPC as a transport protocol for a back-end service. gRPC was originally designed to support inter service communication, but through the project, its efficient data transfer and broad code generation has proved it to be a solid choice for client supporting services as well. It is a promising concept able to support the growing demand of services which require real-time data delivery with minimal bandwidth requirements.

REFERENCES

[1] "Myepi," myepi.dk, [Accessed: 9 May 2023].

[2] "Isolate class," https://api.flutter.dev/flutter/dart-isolate/Isolate-class.html, March 2023, [Accessed: 9 May 2023].

[3] "Documentation authentication," https://firebase.google.com/docs/auth, May 2023, [Accessed: 9 May 2023].

[4] "Enabling offline capabilities," https://firebase.google.com/docs/database/flutter/offline-capabilities, Juli 2022, [Accessed: 9 May 2023].

[5] "Openapi introduction," https://docs.nestjs.com/openapi/introduction, [Accessed: 9 May 2023].

[6] "Openapi specification v3.1.0," https://spec.openapis.org/oas/latest.html, February 2021, [Accessed: 9 May 2023].

[7] "Comming from openapi," https://www.asyncapi.com/docs/tutorials/getting-started/coming-from-openapi, April 2019, [Accessed: 9 May 2023].

[8] J. Brandhorst, "The state of grpc in the browser," https://grpc.io/blog/state-of-grpc-web, January 2019, [Accessed: 9 May 2023].

[9] B. Phillips, "grpc with rest and open apis," https://grpc.io/blog/coreos/, May 2016, [Accessed: 9 May 2023].

[10] T. Peskens, "Why choose between grpc and rest," https://medium.com/@thatcher/why-choose-between-grpc-and-rest-bc0d351f2f84, February 2018, [Accessed: 9 May 2023].

[11] "Overview pipes," https://docs.nestjs.com/pipes, [Accessed: 9 May 2023].

[12] "Connection pool overview," https://www.mongodb.com/docs/manual/administration/connection-pool-overview, 2023, [Accessed: 9 May 2023].

[13] J. Nielsen, "Response times: The 3 important limits," https://www.nngroup.com/articles/response-times-3-important-limits/, 1993, [Accessed: 12 May 2023].

[14] J. Ji, "protobuf2swagger," https://www.npmjs.com/package/protobuf2swagger, 2022, [Accessed: 9 May 2023].

[15] "Protoc gen typescript," https://www.npmjs.com/package/protoc-gen-ts, 2023, [Accessed: 9 May 2023].

[16] "Protocol buffers v3," https://cloud.google.com/apis/design/proto3, May 2023, [Accessed: 9 May 2023].

[17] "Websockets pipes," https://docs.nestjs.com/websockets/pipes, [Accessed: 9 May 2023].

[18] "db.collection.watch()," https://www.mongodb.com/docs/manual/reference/method/db.collection.watch, 2023, [Accessed: 9 May 2023].

[19] "Sharding," https://www.mongodb.com/docs/manual/sharding, 2023, [Accessed: 9 May 2023].

[20] "Replication," https://www.mongodb.com/docs/manual/replication, 2023, [Accessed: 9 May 2023].

[21] "Replica set oplog," https://www.mongodb.com/docs/manual/core/replica-set-oplog, 2023, [Accessed: 9 May 2023].

[22] "Read concern," https://www.mongodb.com/docs/manual/reference/read-concern, 2023, [Accessed: 9 May 2023].

[23] "Change streams," https://www.mongodb.com/docs/manual/changeStreams, 2023, [Accessed: 9 May 2023].

APPENDIX A
REQUIREMENTS TABLES

ID	Prioritisation	Requirement
F1	Must	Profile
F1.1	Must	A user shall be able to create a profile
F1.1.1	Could	The profile must be given name, email, phone number, date of birth, address, password, user type, seizure type
F1.1.2	Could	The given email must be verified to belong to the user
F1.1.3	Could	The given phone number must be verified to belong to the user
F1.2	Should	A verified user shall be able to delete their profile along with all connected data
F1.3	Must	A verified user shall be able to update their profiles non-identifying data
F1.4	Should	A verified user shall be able to get their profile
F1.5	Must	A verified user shall be able to listen for and receive changes of their profile
F1.6	Could	A verified user shall be able to set and update their contact alert settings
F1.7	Would	A verified user shall be able to upgrade to a paid user type
F2	Would	Contacts
F2.1	Would	A verified user shall be able to add contacts to their profile
F2.1.1	Would	The contact must be given name, type, and phone number
F2.1.2	Would	A verified user shall be able to add 2 contacts without payment
F2.1.3	Would	A verified user shall be able to add additional contacts with payment
F2.1.4	Would	A given contacts phone number must be verified
F2.1.5	Would	A given contacts phone number must not be used for the user's own profile
F2.1.6	Would	A given contacts phone number must only be used for one of the user's contacts
F2.2	Would	A verified user shall be able to delete their contacts
F2.3	Would	A verified user shall be able to listen for and receive changes to their contacts
F3	Must	Seizures
F3.1	Must	A verified user shall be able to add seizure to their profile
F3.1.1	Could	The seizure shall contain date and time, duration, description, plus additional information
F3.1.2	Could	The seizure can contain location
F3.1.3	Would	A verified user shall be able to add seizure with alert for contacts
F3.1.4	Must	A verified user shall be able to add seizure without alert for contacts
F3.2	Should	A verified user shall be able to update a seizure tied to their profile
F3.3	Must	A verified user shall be able to delete a seizure tied to their profile
F3.4	Should	A verified user shall be able to get seizures tied to their profile matching a given filter
F3.5	Must	A verified user shall be able to listen for and receive changes to seizures tied to their profile matching a given filter
F4	Would	Notification stream
F4.1	Would	A verified user shall be able to receive notifications
F5	Must	Authentication
F5.1	Must	A user shall be able to verify with email and password
F5.2	Would	A user shall be able to reset password with email
F5.3	Must	A verified user shall be able to remain logged in across sessions
F5.4	Must	A verified user shall be able to end authentication session

TABLE I: Functional requirements

	Category	Requirement	Test parameter
NF1	Scalability	Service must be stateless	N/A
NF2	Availability	Must handle 500 active users per service	500 clients with 1 modification / second
NF3	Modifiability	Must allow easy implementation of defined expected future features	N/A
NF4	Testability	Must have 100% test coverage of endpoint call states	Test coverage
NF5	Testability	Must have 95% test coverage of complete service	Test coverage
NF6	Security	Must use Bcrypt for passwords	N/A

TABLE II: Non-functional requirements

APPENDIX B
USE-CASE ANALYSIS DERIVED CONCEPTS TABLE

AoR	Data Objects	Errors	Internal processes	Database calls
Authentication	Email Password AuthUser RefreshToken AuthSession AccessToken	Password insufficient Email insufficient User exists !User exits !Password match user !Access token Verified !Refresh token verified	Check password format Check email format Check password match user Verify token Create access token Create refresh token	Find user from email Create user from email and password Create session from user and refresh token Delete session from user Get session from refresh token
Profile	ProfileData AccessToken UserId Profile ChangeType	!Access token Verified !Profile data verified	Verify token Verify profile data Get user id from access token	Create profile from data and user id Update profile from data and user id Get profile from user id Listen to profile from user id
Seizure	SeizureData AccessToken UserId Seizure ChangeType SeizureId FilterData	!Access token Verified !seizure exist !Seizure data verified !filter data verified	Verify token Verify seizure data Verify filter data Get user id from access token	Create seizure from data and user id Delete seizure from seizure id Get seizures from filter and user id Listen to seizures from filter and user id
Cross Section	AccessToken UserId ChangeType	!Access token Verified	Verify token Get user id from access token	

APPENDIX C
UNIT TESTS TABLES

Service	Function	Expect	Description
Persistence	createSession	Pass	When called, should insert session in database
Persistence	deleteSession	Pass	When called, should remove session form database
Service	createSession	Pass	When ObjectId is given, should return session with refresh token and expiration
Service	signOut	Pass	When refresh token of existing session is given, should remove session from database
Service	signOut	Pass	When non existing token is given, should complete without error
Service	refreshAccessToken	Pass	When refresh token of existing session is given, should return JWT token containing user id
Service	refreshTokenGuard	Fail	When invalid token given, should throw error
Service	refreshTokenGuard	Fail	When expired token given, should throw error
Service	refreshTokenGuard	Pass	When valid token given, should not throw error

TABLE III: Auth Session Module unit tests

Service	Function	Pass/Fail	Description
Persistence	createUser	Pass	When valid information given, should create user and return it
Persistence	getUserFromEmail	Pass	When user exists, should create user, and return it
Persistence	getUserFromEmail	Pass	When user does not exist, should return null
Service	signUp	Pass	When valid credentials given, should create user and return auth tokens
Service	signUp	Fail	When in use email is given, should throw error
Service	signIn	Fail	When invalid email is given, should throw error
Service	signIn	Fail	When invalid password is given, should throw error
Service	signIn	Pass	When session already exists, should return new session

TABLE IV: Auth Module unit tests

Service	Function	Expect	Description
Persistence	create	Pass	When required attributes given, should insert profile to database
Persistence	update	Pass	When profile exist and required attributes given, should update profile with attributes
Persistence	get	Pass	When profile exist, should return profile
Persistence	stream	Pass	When profile is created, should return ProfileChange with type CREATE and correct profile data
Persistence	stream	Pass	When profile is updated, should, return ProfileChange with type UPDATE and correct profile data
Persistence	stream	Pass	When one user subscribes to stream, and other user changes data, should not return change event
Service	create	Pass	When valid AccessToken given, should insert profile in database
Service	update	Pass	When valid AccessToken given, should update profile in database
Service	get	Pass	When valid AccessToken given, should return profile
Service	stream	Pass	When valid AccessToken given and user is created, should return ProfileChange of type CREATE

TABLE V: Profile Module unit tests

Service	Function	Pass/Fail	Description
Persistence	create	Pass	When valid input given should create new seizure
Persistence	create	Fail	When invalid input is given, should throw error
Persistence	delete	Pass	When valid id is given, should delete associated seizure
Persistence	get	Pass	When no id is given, should return empty
Persistence	get	Pass	When filter is given, should return all seizures
Persistence	get	Pass	When only lower bound duration is given, should return seizures with same or longer duration
Persistence	get	Pass	When only upper bound duration is given, should return seizures with same or shorter duration
Persistence	get	Pass	When all filters are given, should return all matching seizures
Persistence	get	Pass	Should not return seizures from other users
Persistence	stream	Pass	When only lower bound duration is given, should return CREATE change event for any seizure
Persistence	stream	Pass	When only upper bound duration is given, should return CREATE change event for seizure with same or lower duration
Persistence	stream	Pass	When all filters are given, should return CREATE change event for matching seizures only
Persistence	stream	Pass	When one user subscribes to stream, and other user changes data, should not return change event
Service	validation	Fail	When negative duration is given, should throw error
Service	Validation	Fail	When invalid enum is given, should throw error
Service	Validation	Pass	When valid input is given, should throw error
Service	create	Pass	When valid input given, should create seizure
Service	delete	Pass	When valid id is given, should delete associated seizure
Service	get	Pass	When seizure exist and valid input given, should return seizure
Service	Stream	Pass	When seizure is created, should return CREATE change event
Service	stream	Pass	When seizure is deleted, should return DELETE change event

TABLE VI: Seizure Module unit tests

APPENDIX D
STAKEHOLDER MEETINGS

A. Requirement Clarification

1) Formalities
The meeting was held the 10^{th} of February 2023. It was attended by Daniel, the project manager of MyEpi. The intention of the meeting was to show the analysis of the current system and the resulting elicited requirements. There were several unused functions and libraries which may or may not have needed to be included in the service. Beyond this, a list of non-functional requirements was desired for the project, as well as goals and hopes for the project.

2) Notes
Clarifications

- Crash reporting will be done by LittleGiants with third party software
- Patients does not exist
- User creation must have FCM token for Firebase
- FCM token must be added for new devices on login and removed on logout
- Seizures need pagination and search

Requirements

- containerised
- all real-time data
- Bcrypt for password
- expected test coverage 95
- 1 modification per second, 1000 active clients
- Build in admin support

Other

- Perform load test for verification

A requirements spec, both for the current system as well as future features, had been created beforehand. Daniel needed to get permission to share it with me. This may have a great impact on the current work.

B. Analysis Presentation

1) Formalities
The meeting was held the 24^{th} of February 2023. It was attended by Daniel, the project manager of MyEpi. The intention of the meeting is to show the analysis of both the current problem domain, the current solution, and the currently used solutions. The goal is to get feedback on what identified qualities, trade-offs, and side effects, matters most to MyEpi and LittleGiants. Design decisions will be based mainly on the current analysis and the given feedback of the meeting.

Questions/Information

- AsyncAPI and OpenAPI vs gRPC and REST
- Stream state management
- Potential separation of Authentication and Authentication Session Management.
- Current use of Access and Refresh tokens are encouraged
- Authentication within WebSocket or not
- Fragile WebSocket Authorisation management
- Current payment accommodation
- Gateway API vs Twilio

2) Notes
- The current Redis setup will be difficult to configure for MongoDB streams
- LittleGiants is open to trying out gRPC and have been working with it before

C. Implementation Presentation

1) Formalities

The meeting was held the 31^{th} of March 2023. It was attended by Daniel, the project manager of MyEpi. The Intention of the meeting was to present the current architecture decision of the project and get feedback on how it aligned with their expectations. The goal was to see how similar it was to their current project structure, and whether there were any concerns that needed to be addressed.

D. Notes

- The current state of the project has demonstrated all concepts used in the current project structure of LittleGiants.
- There is an interest in a cleaner and more detailed presentation of the endpoints than the raw proto file.
- There is an interest in generating or clearly documenting permitted value ranges and patterns for the endpoints

APPENDIX E
CONCEPT LOCATION

A. Method

The aim of this project is to replace the Firebase service with a custom solution. The first step is to locate and analyse the functionality used from the Firebase packages, as this is the functionality to replace.

To do this, the used Firebase libraries are first identified. Their general usage and responsibility are looked up. Thereafter, all the files using the libraries are identified and analysed to identify the usage of each library. Only functionality relevant to the application is in focus. Firebase specific setup is disregarded.

The functions using the Firebase libraries will be recursively investigated. This is to ensure that data given to the function, and used from the functions return value, is identified. This includes side-effects of stateful functions.

The goal is gaining a list of functions, function responsibilities, and needed data models from the existing implementation of Firebase.

B. Import Analysis

First, the Firebase packages are identified from the pubspec.yaml file, where imports of a flutter project are defined:

- firebase_database
- firebase_remote_config
- firebase_core
- firebase_crashlytics
- firebase_auth
- cloud_firestore
- cloud_functions
- firebase_messaging

1) firebase_database

The Firebase database package is a package that allows the user to communicate with a Firebase Real-time No-SQL database. However, it appears that it is not used anywhere in the front-end and is a dead import.

2) firebase_remote_config

Remote config is a package which allows you to configure the appearance and behaviour of your app through cloud hosted values which your app responds to. This allows you to update the app through the cloud hosted values without rebuilding and uploading the through any app store. The remote config package is only reference in the versionCheck.dart file. However, the file and its contained static functions are not referenced, and the package is therefor currently not in use.

3) firebase_core

The package is used only in the main.dart file on which the function initializeApp is called. This appears to handle configuration and is therefore not relevant to the replacement system.

4) firebase_crashlytics

Crashlytics is a package used for customising the collected crash reports of the Firebase setup. It is referenced only in the main.dart file, where it is used to disable crash reporting when debug mode is enabled and enable the crash reporting when debug mode is disabled.

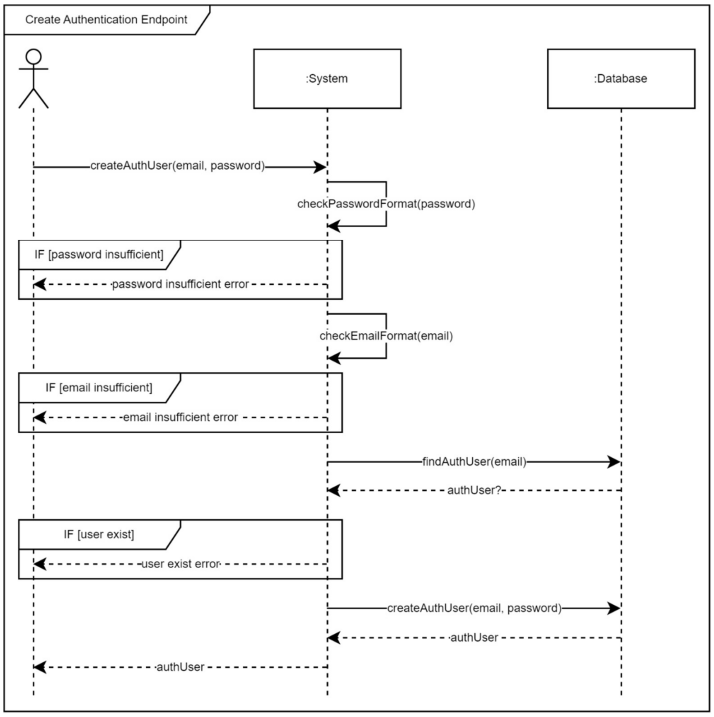
5) Remaining Packages

The remaining packages appear in a number of files and appear to be central for the main functionality of the Firebase setup. Each will be investigated for its use in the project. If the file provides functionality in use, which is not solely relevant to a Firebase implementation, it will be considered in the requirements elicitation process.

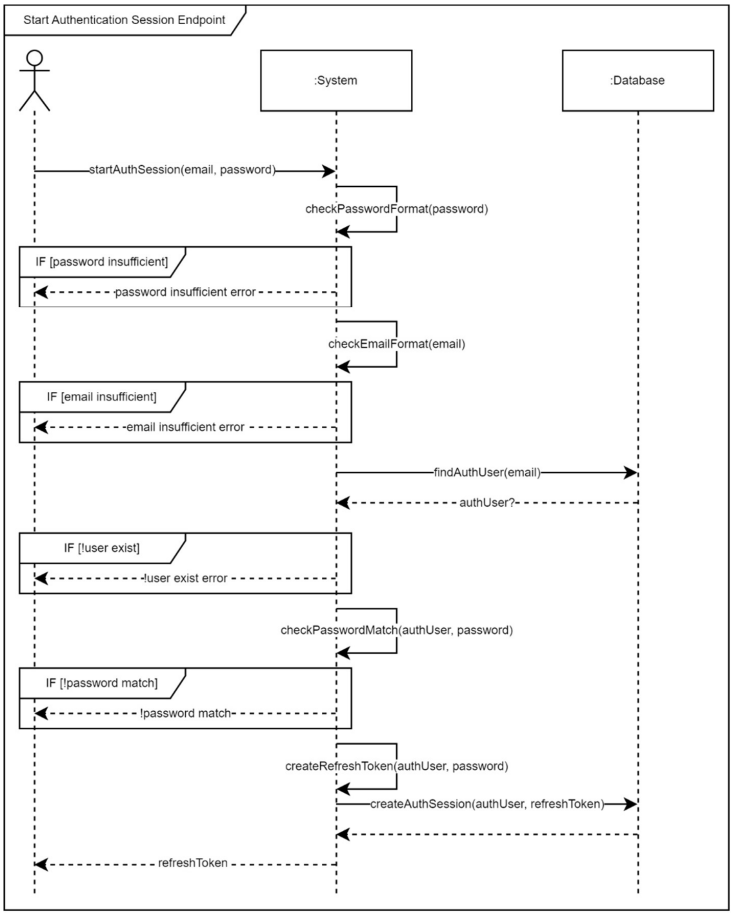
1. Use-case Analysis

1.1. Authentication

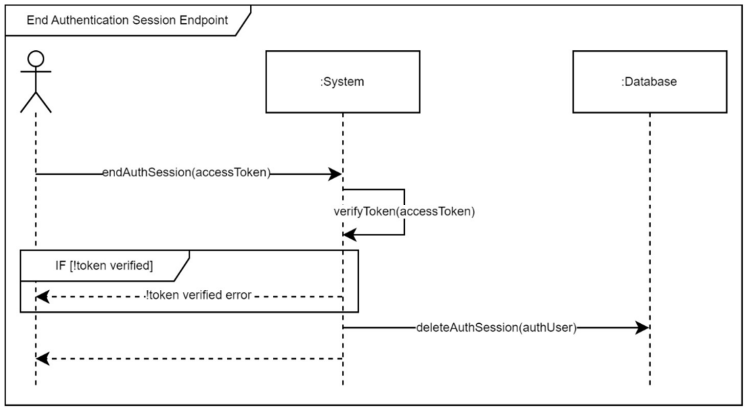
Create Authentication User	
Fulfilled requirements	
Goal	Actor has a user associated with email
Sequence	1. Actor request user creation with email and password 2. Email and password are checked for format adherence 3. User is created with email and password 4. Actor is informed of successful user creation
Error states	2.a. email or password does not adhere to format requirements, actor is informed of error, transaction terminates 3.a. user exists, actor is informed, transaction terminates



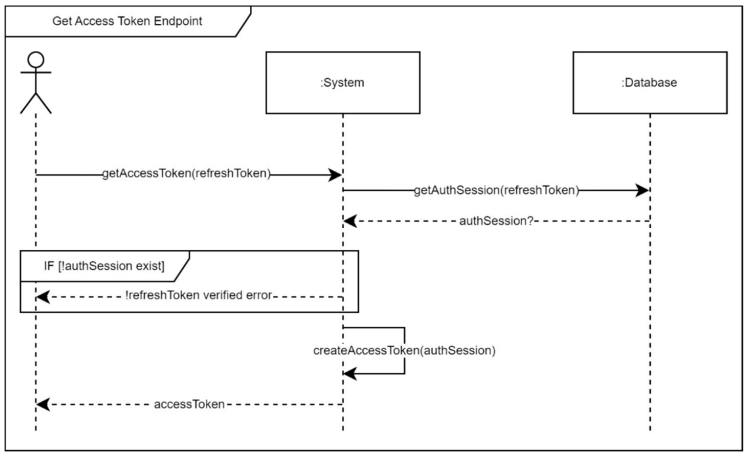
Start Authentication Session	
ID	F6.1
Precondition	Actor has created a user
Postcondition	Actor is verified and has proof of authentication session
Sequence	1. Actor request user verification with email and password 2. Email and password are verified 3. Authentication session is started 4. Proof of authentication session is returned to actor
Alternatives	2.a.1. Email does not match any user, actor is informed, transaction terminates 2.a.2. Password does not match user, actor is informed, transaction terminates 3.a. authentication session is already active, transaction continues



End Authentication Session	
ID	F6.4
Precondition	Actor has created a user and is verified
Postcondition	Actor is no longer verified
Sequence	1. Actor request authentication session terminations with proof of verification 2. Verification data is verified 3. Authentication session is terminated 4. Actor is informed of successful termination
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



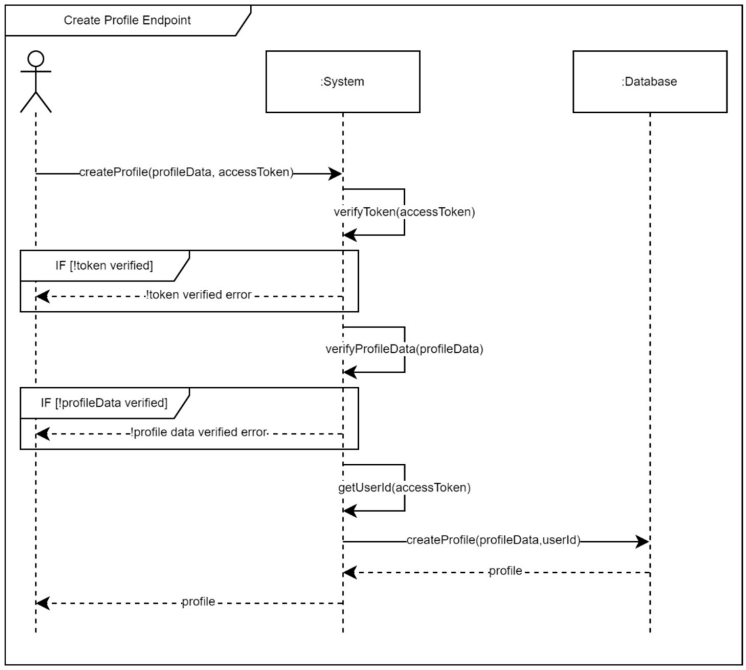
Get Authentication Token	
ID	F6.3
Precondition	The actor has created a user and is verified
Postcondition	The actor has proof of verification
Sequence	1. Actor request proof of verification with proof of authentication session 2. Authentication session is verified 3. Verification proof is created 4. Verification proof is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



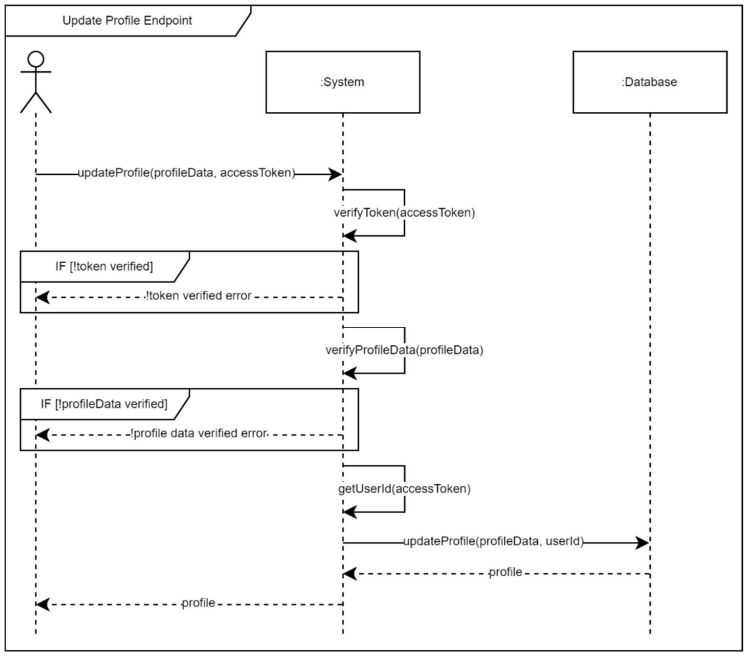
Errors	Internal processes	Database calls
Password insufficient Email insufficient User exists !User exits !Password match user !Access token Verified !Refresh token verified	Check password format Check email format Check password match user Verify token Create access token Create refresh token	Find user from email Create user from email and password Create session from user and refresh token Delete session from user Get session from refresh token

1.2. Profile

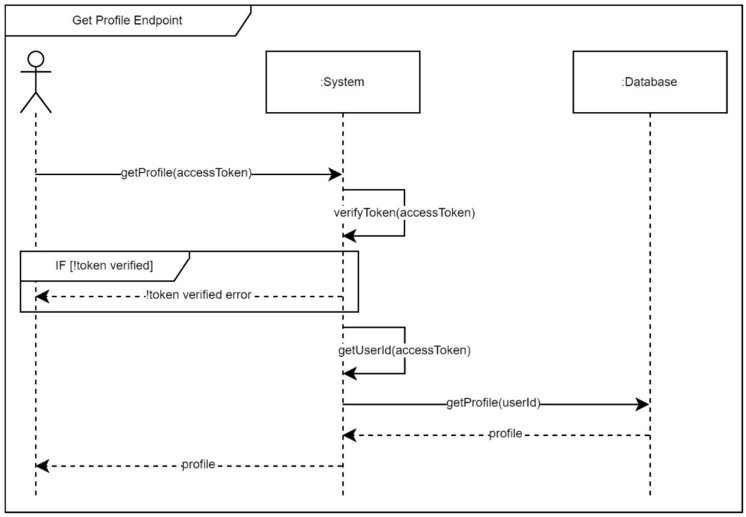
Create Seizure	
ID	F1.1
Precondition	The actor has created a user and is verified
Postcondition	The actor has a profile associated to their existing user
Sequence	1. Actor request profile creation with profile data and proof of verification 2. Verification data is verified 3. Profile data is verified 4. Profile is created 5. Created profile is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates. 3.a. Profile Data is rejected, actor is informed of error, transaction terminates.



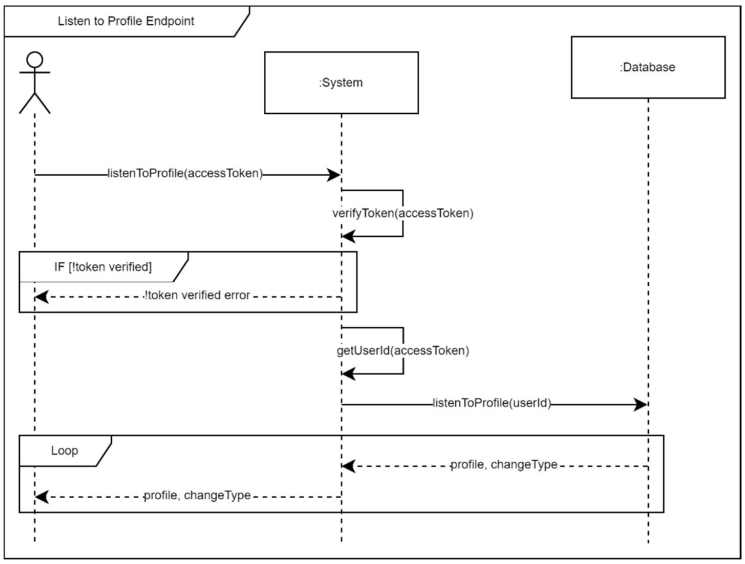
Update Profile	
ID	F1.3
Precondition	The actor has created a user, a profile, and is verified
Postcondition	The actor's associated profile is updated with new data
Sequence	1. Actor request profile update with profile data and proof of verification 2. Verification data is verified 3. Profile data is verified 4. Profile matching user is found 5. Profile is updated 6. Updated profile is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates. 3.a. Profile Data is rejected, actor is informed of error, transaction terminates.



Get Profile	
ID	F1.4
Precondition	The actor has created a user, a profile, and is verified
Postcondition	
Sequence	1. Actor request profile with proof of verification 2. Verification data is verified 3. Profile matching verified user is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



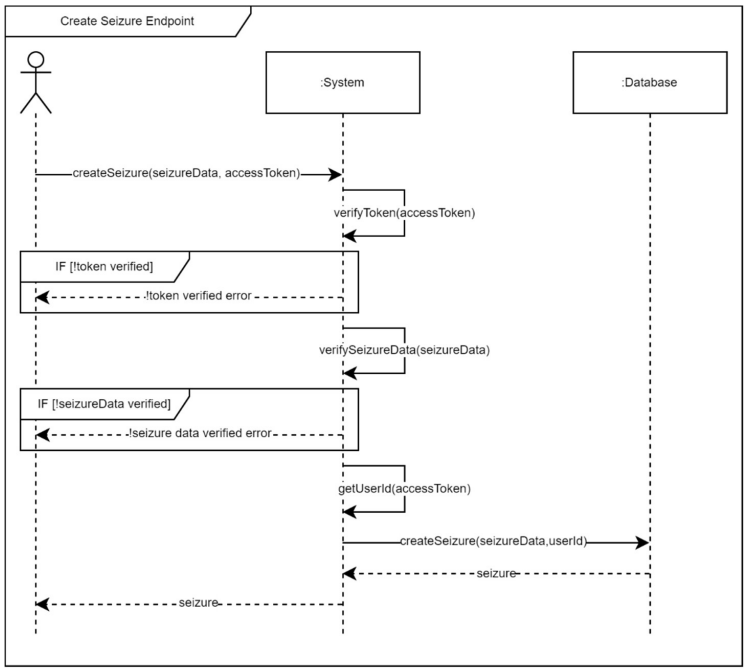
Listen to Profile	
ID	F1.5
Precondition	The actor has created a user, a profile, and is verified
Postcondition	
Sequence	1. Actor request updates on profile state change with proof of verification 2. Verification data is verified 3. Changes to profile matching verified user are sent to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



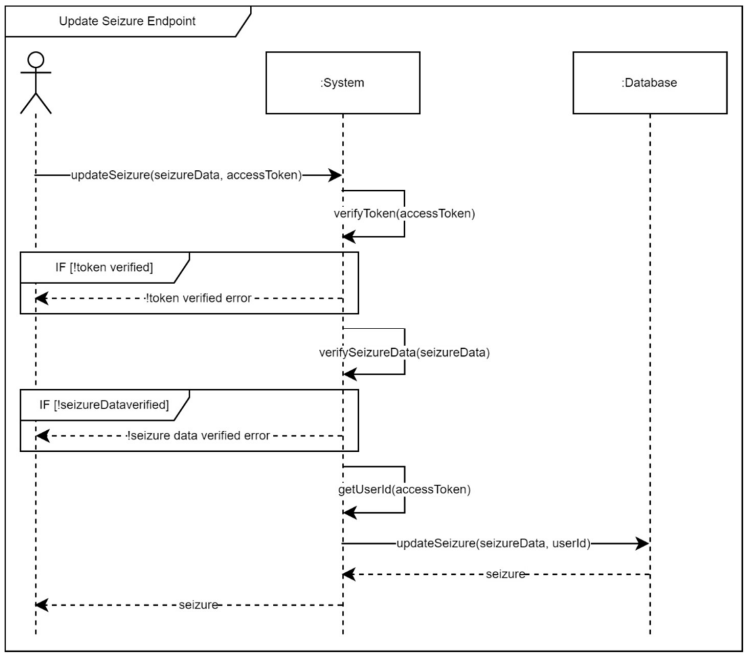
Errors	Internal processes	Database calls
!Access token Verified !Profile data verified	Verify token Verify profile data Get user id from access token	Create profile from data and user id Update profile from data and user id Get profile from user id Listen to profile from user id

1.3. Seizures

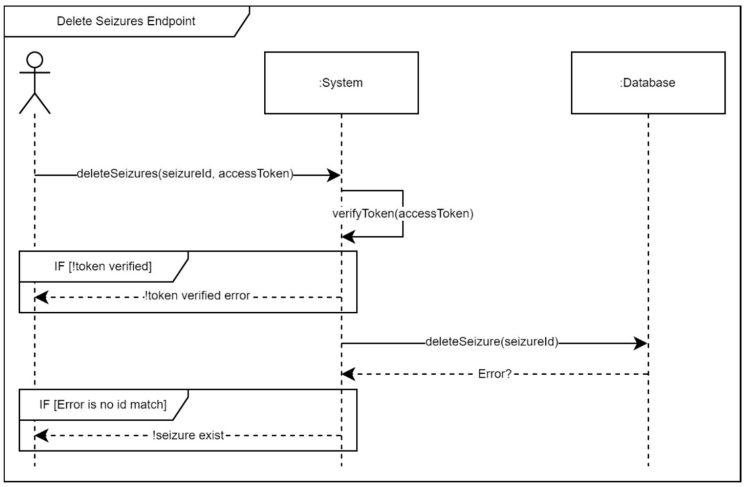
Create Profile	
ID	F3.1
Precondition	The actor has a profile and is verified
Postcondition	The actor has a new seizure associated to their existing user
Sequence	1. Actor request seizure creation with seizure data and proof of verification 2. Verification data is verified 3. Seizure data is verified 4. Seizure is created 5. Created seizure is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates. 3.a. Seizure Data is rejected, actor is informed of error, transaction terminates.



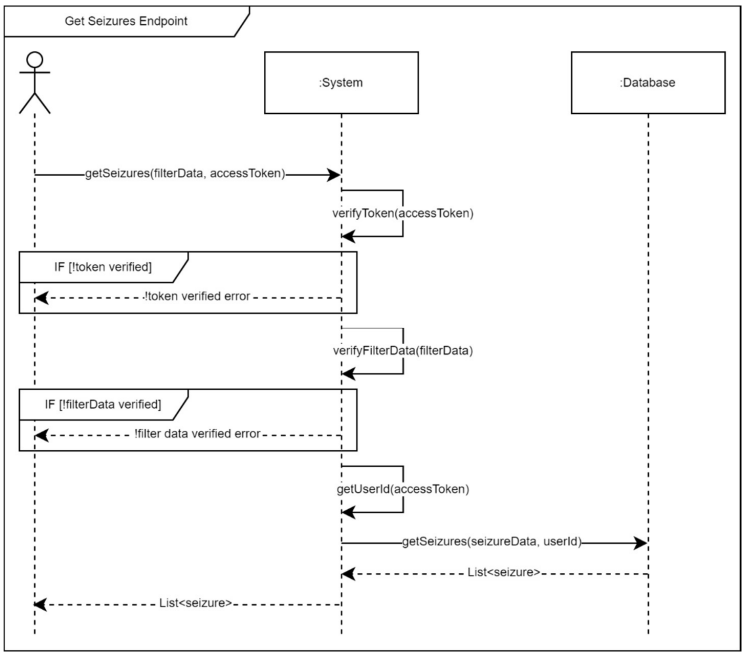
Update Seizure	
ID	F3.2
Precondition	The actor has created a user, a profile, a seizure, and is verified
Postcondition	The selected seizure is updated with new data
Sequence	1. Actor request seizure update with seizure data, id, and proof of verification 2. Verification data is verified 3. Seizure data is verified 4. Seizure matching id is found 5. Seizure is updated 6. Updated seizure is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates. 3.a. Seizure data is rejected, actor is informed of error, transaction terminates. 4.a. No seizure is found matching id, actor is informed, transaction terminates.



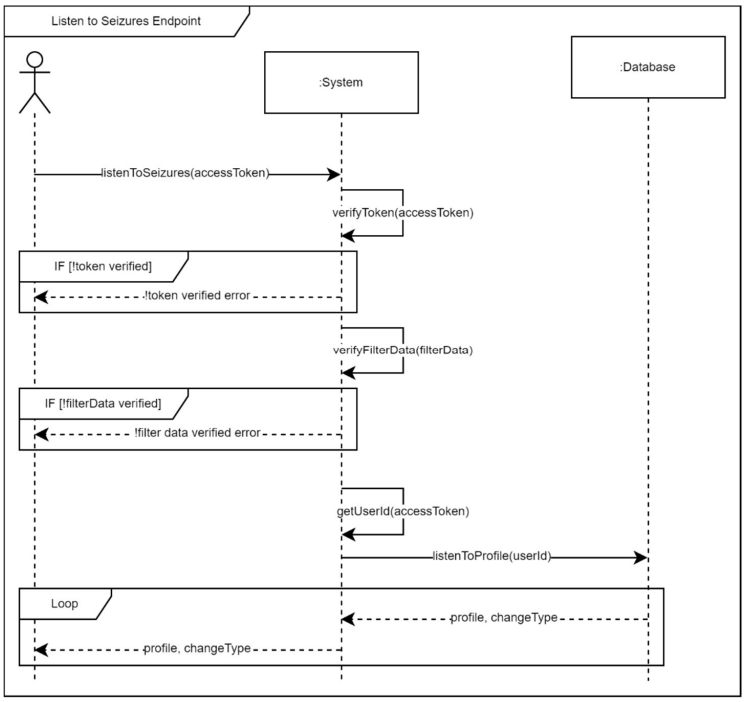
Delete Seizure	
ID	F3.3
Precondition	The actor has created a user, a profile, a seizure, and is verified
Postcondition	The selected seizure is deleted
Sequence	1. Actor request seizure deletion with seizure id and proof of verification 2. Verification data is verified 3. Seizure matching id is found 4. Seizure is deleted 5. Confirmation of deletion is returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates. 3.a. No seizure is found matching id, actor is informed, transaction terminates.



Get Seizures	
ID	F3.4
Precondition	The actor has created a user, a profile, and is verified
Postcondition	
Sequence	1. Actor request seizure with filter data and proof of verification 2. Verification data is verified 3. Seizures matching filter and verified user is found 4. Seizures are returned to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



Listen to Seizure	
ID	F3.5
Precondition	The actor has created a user, a profile, and is verified
Postcondition	
Sequence	1. Actor request updates on user seizures with filter and proof of verification 2. Verification data is verified 3. Changes to seizures matching filter and verified user are sent to actor
Alternatives	2.a. Verification data is rejected, actor is informed, transaction terminates.



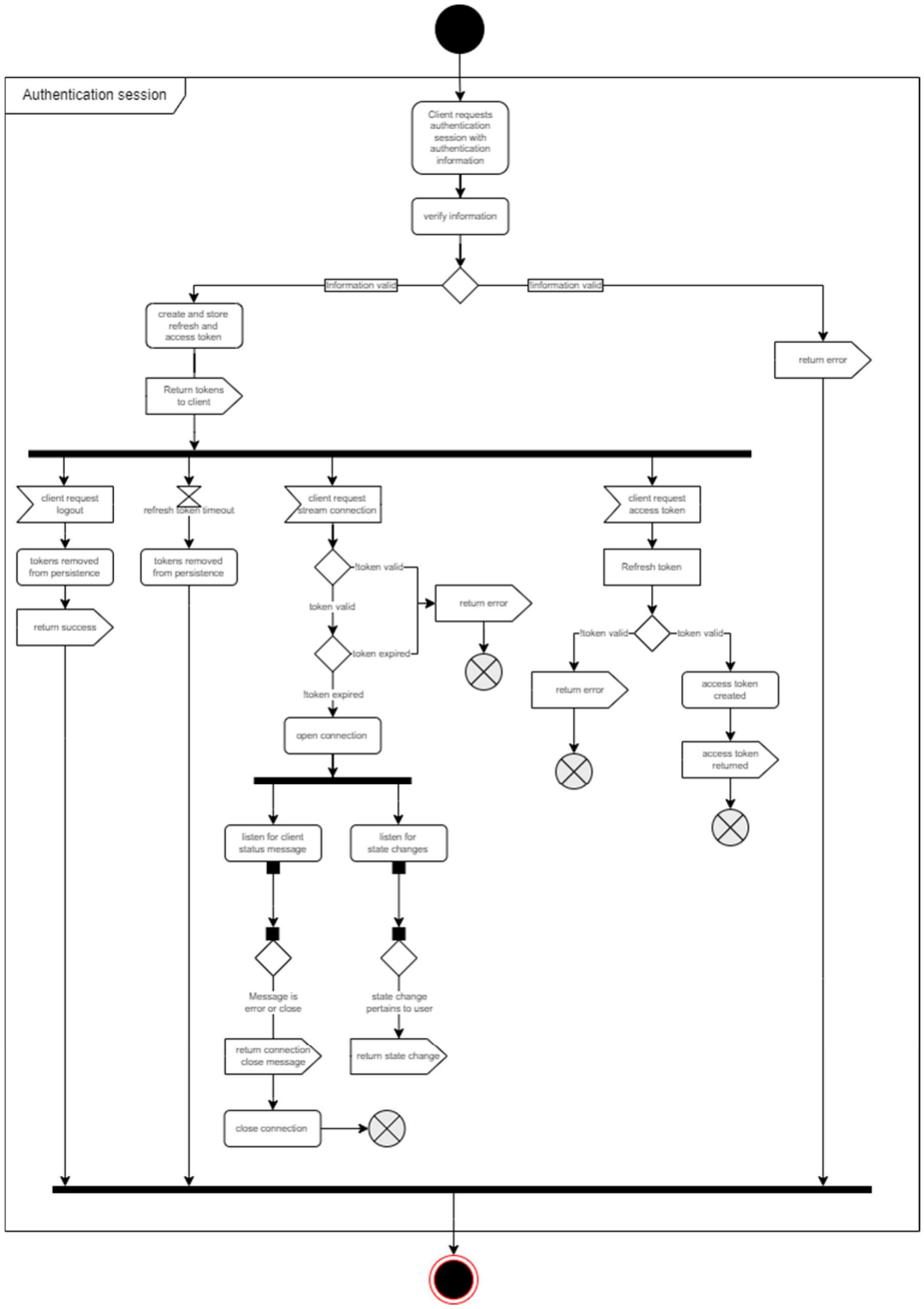
Errors	Internal processes	Database calls
!Access token Verified !Profile data verified !seizure exist !Seizure data verified !filter data verified	Verify token Verify seizure data Verify filter data Get user id from access token	Create seizure from data and user id Update profile from data and seizure id Delete seizure from seizure id Get seizures from filter and user id Listen to seizures from filter and user id

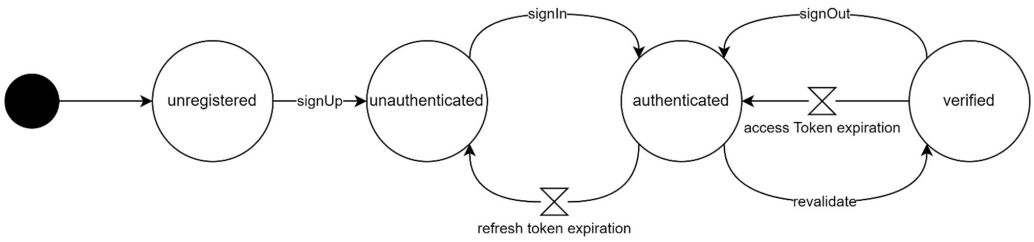
1.4. Results

Errors	Internal processes	Database calls
!Access token Verified !Profile data verified	Verify token Get user id from access token	

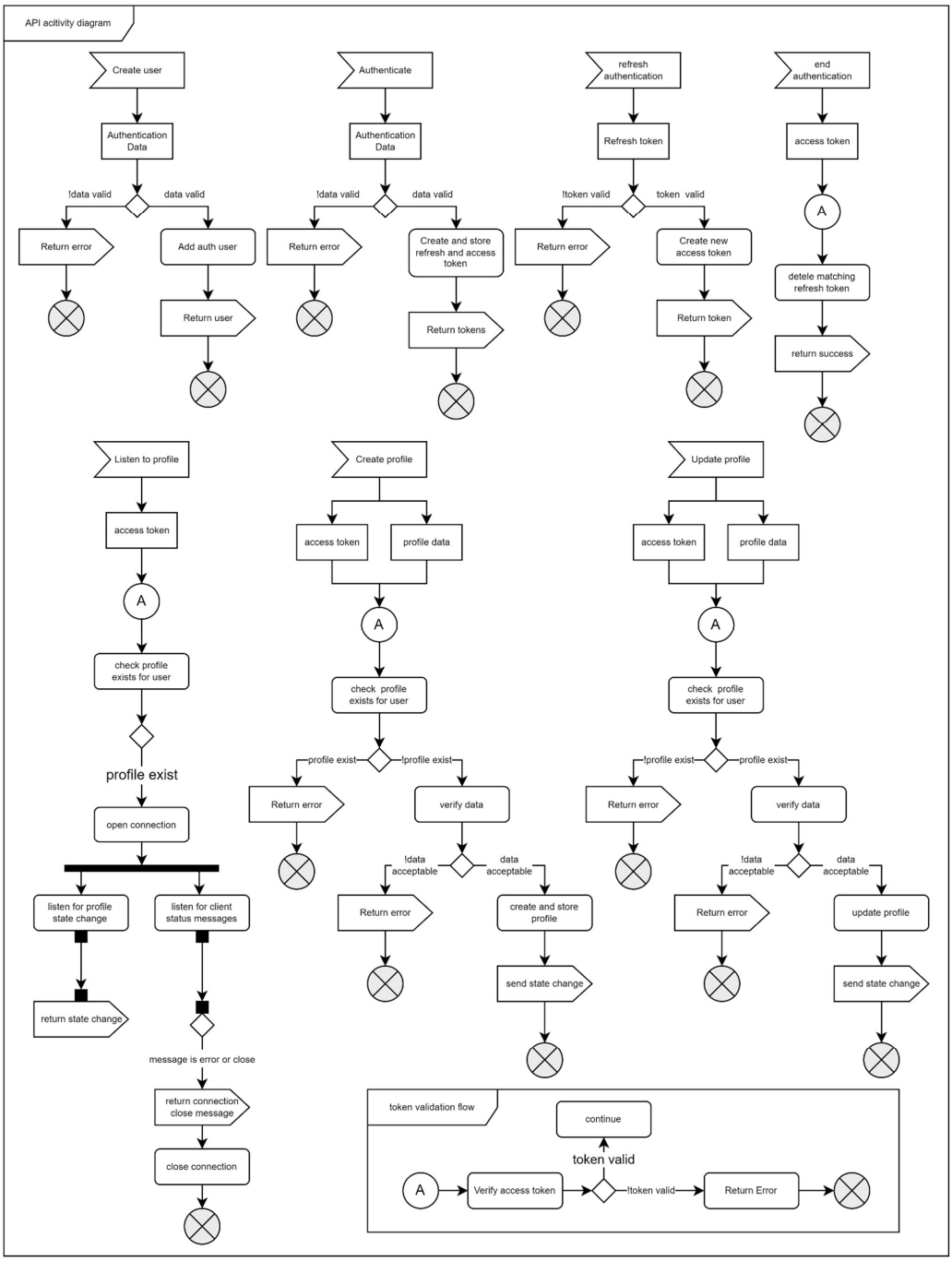
2. Activity Diagrams

2.1. Auth Session




unregistered
signUp
unauthenticated
signIn
authenticated
signOut
verified
access Token expiration
refresh token expiration
revalidate

2.2. API


API acitivity diagram
Create user
Authentication Data
!data valid
data valid
Return error
Add auth user
Return user
Authenticate
Authentication Data
!data valid
data valid
Return error
Create and store refresh and access token
Return tokens
refresh authentication
Refresh token
!token valid
token valid
Return error
Create new access token
Return token
end authentication
access token
A
detele matching refresh token
return success
Listen to profile
access token
A
check profile exists for user
profile exist
open connection
listen for profile state change
listen for client status messages
return state change
message is error or close
return connection close message
close connection
Create profile
access token
profile data
A
check profile exists for user
profile exist
!profile exist
Return error
verify data
!data acceptable
data acceptable
Return error
create and store profile
send state change
Update profile
access token
profile data
A
check profile exists for user
!profile exist
profile exist
Return error
verify data
!data acceptable
data acceptable
Return error
update profile
send state change
token validation flow
continue
token valid
A
Verify access token
!token valid
Return Error