

Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum

Managing Data Retrieval in a Multi-Tenant Environment

Bachelor's thesis in Computer Science

Supervisor: Rituka Jaiswal

May 2024

Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum

Managing Data Retrieval in a Multi-Tenant Environment

Bachelor's thesis in Computer Science

Supervisor: Rituka Jaiswal

May 2024

Norwegian University of Science and Technology

Faculty of Information Technology and Electrical Engineering

Department of ICT and Natural Sciences



Norwegian University of
Science and Technology

Abstract

In an era where data accessibility and management are paramount, FiiZK Digital AS sought to enhance the integration and accessibility of aquaculture data across various platforms.

This bachelor thesis describes the development of a full-stack application, designed to explore and implement multiple methods of data retrieval within a multi-tenant environment, ensuring that customers can manage and access their data efficiently.

By utilizing a combination of .NET 8, Vue.js, PostgreSQL, and Docker, the group developed proof-of-concept REST, GraphQL, and Kafka-based solutions to provide three different, robust, secure and scalable methods of data transfer.

This thesis also includes details about the development process, including challenges, technological choices and how we used agile project management principles to organize and effectively develop our final product.

Sammendrag

I en tid hvor datatilgjengelighet og håndtering blir konstant viktigere, har FiiZK Digital AS forsøkt å forbedre integreringen og tilgjengeligheten av akvakulturdata på tvers av ulike plattformer.

Denne bacheloroppgaven beskriver utviklingen av en fullstack-applikasjon, designet for å utforske og implementere flere datainnhentingsmetoder i et multi-tenant-miljø, for å sikre at kunder kan administrere og få tilgang til dataene sine effektivt.

Ved å bruke en kombinasjon av .NET 8, Vue.js, PostgreSQL og Docker, har gruppen utviklet proof-of-concept løsninger basert på REST, GraphQL og Kafka for å gi tre forskjellige, robuste, sikre og skalerbare metoder for å håndtere data.

Denne oppgaven inneholder også detaljer om utviklingsprosessen, inkludert utfordringer og teknologiske valg, og hvordan vi brukte agile metoder og prinsipper for å organisere og effektivt utvikle sluttproduktet vårt.

Preface

About

This project was offered through NTNU along with other projects, where we could request which projects we wanted for our Bachelor's thesis, in order of priority.

We requested this project as our first choice because we wanted a project where we could build on our previous experience regarding developing full-stack applications, which had been relatively surface-level.

The project offered the possibility of using new technologies and exploring different methods of transferring data.

Date of completion: 21.05.2024

Group Members:

Adrian Rennan Dahl

Simen Haug Veum

Ole Tønning Aanderaa

Thanks

We would like to express our heartfelt gratitude to the following individuals:

- Marius Lundbø and Erik Måring at FiiZK, our client, for their availability and valuable feedback during the development of the project.
- Our supervisor, Rituka Jaiswal, for her insightful advice on how to approach our project, assistance with various challenges, and guidance throughout the report writing process.

Table of Contents

Abstract	i
Sammendrag	i
Preface	ii
About	ii
Thanks	ii
List of Figures	vi
List of Tables	viii
Acronyms	1
Terms	1
1 Introduction	3
1.1 Background	3
1.2 Problem Statement	3
1.3 Requirements	3
1.4 Limitations	4
1.5 Subject Areas	4
1.6 Document Structure	4
2 Theory	6
2.1 Programming & Frameworks	6
2.1.1 Object Oriented Programming	6
2.1.2 SOLID Principles	6
2.1.3 .NET	6
2.1.4 C#	7
2.1.5 Single page application	7
2.1.6 Vue	7
2.1.7 JavaScript	7
2.1.8 TypeScript	7
2.1.9 Tailwind CSS	7

2.2	Data Exchange	8
2.2.1	REST API	8
2.2.2	GraphQL	8
2.2.3	Kafka	9
2.3	Data Storage and Security	9
2.3.1	Database	9
2.3.2	PostgreSQL	9
2.3.3	Multi-Tenant Environment	10
2.3.4	Encryption	10
2.3.5	HTTPS	10
2.3.6	Bearer Token	10
2.3.7	GUID	11
2.4	Tools	11
2.4.1	Docker	11
2.4.2	Postman	11
2.4.3	DBeaver	11
2.4.4	ChatGPT	11
2.4.5	GitHub Copilot	12
2.5	Version Control And Documentation	12
2.5.1	Git	12
2.5.2	GitHub	12
2.5.3	NGINX	12
2.5.4	Overleaf	13
2.6	Development Methodologies	13
2.6.1	SCRUM	13
2.6.2	Agile Methods	13
3	Method	14
3.1	Managing the project	14
3.1.1	Meetings	14
3.1.2	Project Organization	14
3.1.3	Dividing Tasks	15
3.2	Planning and Choosing Technologies	16

3.2.1 Frameworks	16
3.2.2 Methods of Data transfer	17
3.2.3 Milestones	18
3.2.4 Wireframes	18
3.3 Development	18
3.3.1 REST API	19
3.3.2 GraphQL	19
3.3.3 Kafka	19
3.3.4 Applications used	19
3.3.5 API overview	20
3.3.6 Project structure	20
3.4 Running the Project	23
4 Result	25
4.1 Engineering Results	25
4.1.1 Demo Video	25
4.1.2 Frontend	29
4.1.3 Backend	33
4.1.4 REST API Solution	37
4.1.5 GraphQL Solution	42
4.1.6 Kafka Solution	48
4.1.7 Containerization	53
4.1.8 Testing	56
4.2 Administrative Results	58
4.2.1 Group Structure	58
4.2.2 Milestone Result	59
5 Discussion	60
5.1 Comparing the solutions	60
5.1.1 Developer Experience	60
5.1.2 Differences in solutions	61
5.1.3 Best Use Case	62
5.2 Administrative Discussion	62

5.3 Unit testing	63
6 Conclusion	64
6.1 Conclusions	64
6.2 Further Work	64
Societal Impact	65
References	65
Appendix	68
A Project Plan	68

List of Figures

1 Example over divided user-stories for a sprint in Jira.	15
2 Example overview over sub-tasks within a user-story in Jira.	15
3 Google search trends for RabbitMQ vs Apache Kafka	17
4 Frontend project structure	21
5 Backend project structure	22
6 Root directory README file	23
7 REST API Use Case Diagram	26
8 GraphQL Use Case Diagram	27
9 Kafka Use Case Diagram	28
10 Architecture diagram of our application	29
11 Customer Page Wireframe	30
12 Keys Page Wireframe	30
13 Theme Page Wireframe	31
14 Login page	31
15 Register page	32
16 Navigation bar after login	32
17 Admin Page when editing a user	33
18 The modified IdentityUser class.	34
19 Sequence diagram of how a key is created and used.	35

20	Example of what an encrypted key could look like.	35
21	ER diagram for our main database	36
22	ER diagram for the private databases assigned to a user	36
23	Frontend GUI of the themes overview.	38
24	Frontend GUI for creating a theme.	39
25	Frontend GUI of the REST key overview.	40
26	Frontend GUI for creating a key	40
27	Frontend GUI for testing a REST key where you can enter a key, see the themes for the key, and test the accessible endpoints.	41
28	How authorization is done using the ASP.NET Core Identity library for a REST API endpoint in the controller, with the accessibility limited to only ADMIN users.	41
29	Overview of the GraphQL Query Schema. Note: The column with <i>encryptedKey</i> are the queries that can take a key as a parameter.	42
30	Example query of fetching all of the fields for the Organizations class.	43
31	Example of a partial GraphQL query, where you only fetch the <i>id</i> and <i>orgNo</i> fields.	43
32	Schema definition of the different fields the objects can return.	44
33	The .NET Entity Framework Core class of Species.	44
34	Overview of the GraphQL mutations. On the right are the parameters.	45
35	GraphQL create key mutation.	45
36	Frontend GUI equivalent for the GraphQL create key mutation.	46
37	Frontend GUI of the GraphQL key overview.	46
38	Frontend GUI for testing a GraphQL key where you can enter a key, see the available class tables and fields that key can make a query of, as well as make a query of your selected fields.	47
39	GraphQL Workaround to authenticate a user.	47
40	Example of body and response when creating a Kafka key using the REST API endpoint.	48
41	Example of GUI dialog and response when creating a Kafka key for a user on the frontend Website.	49
42	Frontend Kafka Keys Overview page.	49
43	Example of REST API endpoints for a sensor in the Mock Sensor .NET Application.	50
44	The Main .NET Applications corresponding REST API endpoints to communicate with the Mock Sensor.	50

45	REST API endpoint and its parameters to start a Mock Sensor.	51
46	Logs from the Mock Sensor Docker Container producing messages to the topic with the <i>userId</i> as the sensor identifier.	51
47	Logs from the main .Net application after a sensor was started and all the messages are sent at once because the <i>SendHistoricalData</i> parameter was set to true.	51
48	Two separate frontend sessions connected to the backend through a Web-Socket, left is watching the live feed, and right requested all of the messages (up until that point).	52
49	Sequence Diagram of the interaction between the .Net backend, mock sensor, Kafka Cluster, Database, and frontend website.	52
50	Frontend table displaying the boat-location-updates logs, sorted by most recent log first.	53
51	The Docker Environment.	54
52	Docker compose using an existing image pulled from docker hub. The <i>:latest</i> annotation means the most recent image is pulled, and every time the project is built it checks for a new version and automatically updates.	55
53	Docker compose setup for the main .Net application.	55
54	Dockerfile for the main .Net application and the test project for the unit-tests.	56
55	URL constants for the main .Net application. <i>mock-sensors:80</i> is used for internal docker-communication with the Mock Sensor .Net application.	56
56	Postman collection run results	57
57	GraphQL AvailableQueries postman tests.	58

List of Tables

1	Initial Project Milestones.	18
2	Group Roles and Responsibilities.	58
3	Project Milestone Result.	59

Acronyms

AES Advanced Encryption Standard. 34

API Application Programming Interface. 7, 8, 11, 13, 17–19, 37, 41, 42, 48, 60, 64

CRUD Create Read Update Delete. 62

GUI Graphical User Interface. 25

IDE Integrated Development Environment. 19

NDA Non-Disclosure Agreement. 4

NTNU Norges Teknisk-Naturvitenskapelige Universitet / Norwegian University of Science and Technology. 14

SOLID Single-responsibility, Open-closed, Liskov Substitution, Interface Segregation and Dependency Inversion principles. 18, 65

Terms

Advanced Encryption Standard A symmetric block cipher chosen by the U.S. government to protect classified information. 34

AspNetCore A framework for building web apps and services with .NET and C#. 16, 19

AspNetCore.Identity A library that manages user info, authentication and authorization, as well as sign-in functionality.. 16, 19

base64 An encoding scheme used to represent binary data in an ASCII string format using 64 characters. 34

Composition API One of the two API styles used to build Vue 3 components. 16

Confluent.Kafka An enterprise-ready platform that completes Kafka with advanced capabilities designed to help accelerate application development and connectivity.. 16, 19

DBeaver A free cross-platform database tool. 19

dependency injection A design pattern where objects are externally passed their dependencies rather than creating them internally. 51

Docker Desktop An application for running the docker engine, making it possible to run docker containers locally. 19, 23, 53

Docker Hub A cloud-based repository service for building and shipping Docker containers. 53

EntityFrameworkCore Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.. 16

Figma A cloud-based design tool used for interface design, incorporating collaboration, and prototyping features. 18

GraphQL A *Graph Query Language* for APIs and a runtime for executing those queries by using a type system defined for your data. 42, 45

hash A function that converts an input (or 'message') into a fixed-size string of bytes. Difficult to decode and often used for password or key storage. 34

Hot Chocolate An open-source GraphQL server for the Microsoft .NET platform. 16, 19, 47

HttpContext Provides HTTP-specific information about an individual HTTP request.. 47

Kafka Cluster A system consisting of ZooKeeper, Kafka brokers, producers, consumers, topics, and partitions.. 48

KsqlDB A database purpose-built for stream processing applications, providing real-time data enrichment and server-side processing. 61

Main .Net Application The main backend application, responsible for authorization, authentication and all interaction with the frontend, user and the Mock Sensor application. 50, 62

microservice architecture A design approach to build an application as a suite of small, independent services modeled around a business domain. 62

Mock Sensor .Net Application A simple and separate .Net application that acts as a "sensor" streaming data to a Kafka cluster. 49, 62

MSSQL A Relational Database Management System developed by Microsoft that uses Structured Query Language (SQL). 16

REST API A protocol for client-server communication that uses HTTP requests to access and manipulate data. 44

shadcn-vue A bunch of highly customize able components. These are Radix Vue components styled using Tailwind CSS. 16

Swagger UI A REST API Documentation Tool. 20

theme A collection of REST API endpoints. 28, 37

TypeScript JavaScript with syntax for types. 16

VS Code A popular and versatile code editor developed by Microsoft. 19

Vue.js A javascript framework. 16

Zookeeper A centralized service, often used to compliment kafka by maintaining and configuration information, naming, providing distributed synchronization, and providing group services. 48, 53

1 Introduction

This chapter will provide an overview of the project that forms the basis of the report. The chapter outlines the project's background, the problem statement, requirements, the scope, and subject areas to help give a clear understanding of the project. In addition, the organization of the report is detailed at the conclusion of this chapter.

1.1 Background

FiiZK is a group of companies providing equipment and digital services for the aquaculture industry. They have developed multiple software solutions and manage significant amounts of biological and economic data. However, despite the sophistication of these tools, the integration and accessibility of data across different platforms remain a complex issue. FiiZK wants to make this data as available and accessible as possible for their costumers, so they have reached out to NTNU. They want a proof of concept that explores various methods of delivering data to their customers.

1.2 Problem Statement

The primary objective is to develop a full-stack application that investigates and evaluates various secure data retrieval techniques. The data retrieval methods in question include REST, GraphQL and Kafka. The application will be containerized using Docker. A user-friendly graphical interface will make using the application easy and enjoyable for the end user. Moreover, the application will incorporate functionality to restrict access to specific data subsets, allowing users to generate and distribute access keys to third parties.

1.3 Requirements

To complete the project as it is outlined by FiiZK the following requirements needs to be fulfilled:

Functional Requirements:

- **Data Retrieval Methods:** The application must support various data retrieval methods to ensure flexibility and accessibility in how data is delivered to users.
- **Graphical User Interface:** Develop a user-friendly interface that allows costumers to manage their data access rights easily.
- **Method To Share Subset Of Data:** Develop a method that enables customers to selectively share subsets of their data with third parties.
- **Containerization:** The application should be containerized and capable of running in multiple containers to support scalable deployment across different environments.

Non-Functional Requirements:

-
- **Scalability:** The architecture must support scalability to accommodate growth in user numbers and data volume without performance degradation.
 - **Security:** Implement robust security measures to protect data integrity and privacy, ensuring that users can manage who accesses their data.
 - **Accessibility:** The application should be designed to be accessible, allowing users to manage data access easily and effectively.

While FiiZK didn't give any strict technical requirements as to which technologies to utilize, they mentioned some preferred technologies. These were .NET 7, Vue, TypeScript, MSSQL and Docker.

1.4 Limitations

There are some limiting factors affecting the project. To avoid signing an NDA, which would prevent us from publishing our project and findings, we have declined the offer to work with real datasets from FiiZK's clients. This is also the preferred option for FiiZK, as it's not necessary for us to have access to the data to develop the application.

Because the project is a proof of concept, the focus is on demonstrating feasibility rather than creating a fully market-ready product. This means some features that we don't consider as critical, such as optimization, may not be included within the scope of this project.

Regarding design and technical solutions for the project we have a high degree of freedom to choose which technologies we would like to use.

1.5 Subject Areas

This project covers a broad range of subject areas within software and computer engineering, with a specific focus on data management and software development methodologies. There are both technical and organizational subject areas which the project covers.

Some of the Technical subject areas include several data retrieval techniques, such as REST, GraphQL and Kafka. It also covers containerization and cloud deployment using Docker, and Software architecture design principles for building full-stack applications.

The organizational subject areas include project management and organization. Using tools like Jira, Confluence and GitHub enhances the management and tracking of the project significantly.

1.6 Document Structure

The document is divided into 6 main chapters:

Chapter 1: Introduction - Provides an overview of the project, including the background, problem statement, requirements, limitations and subject areas.

Chapter 2: Theory - Introduces the theoretical framework and materials relevant to both the development and the solution of the project.

Chapter 3: Method - Describes the scientific and developmental methodologies employed in the project.

Chapter 4: Result - Details the outcomes of the project and showcases what has been accomplished.

Chapter 5: Discussion - Discusses the results in the context of the initial problem statement and reflects on the project's findings.

Chapter 6: Conclusion - Concludes the thesis, summarizing the project's thematic and practical outcomes and discussing the implications of the results and any future work.

2 Theory

2.1 Programming & Frameworks

2.1.1 Object Oriented Programming

Object oriented programming, or OOP is a way of programming “that organizes software design around data, or objects, rather than functions and logic.” [1]

2.1.2 SOLID Principles

The SOLID principle is a set of guidelines for writing maintainable, scalable, and robust object-oriented software and stands for:

- **S**ingle-responsibility Principle
 - “A class should have one and only one reason to change, meaning that a class should have only one job.” [2]
- **O**pen-closed Principle
 - “Objects or entities should be open for extension but closed for modification.” [2]
- **L**iskov Substitution Principle
 - “Let $q(x)$ be a property provable about objects of x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .” [2]
- **I**nterface Segregation Principle
 - “A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.” [2]
- **D**ependency Inversion Principle
 - “Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.” [2]

2.1.3 .NET

.Net was introduced in the year 2000 and is a free, open-source application platform developed and maintained by Microsoft. It primarily uses C# as its programming language of choice, but also supports other programming languages like C++ and Visual Basic. It is used to create a various amount of applications for different platforms, like web-, desktop-, and mobile applications [3].

As of February 2024, the newest version of .NET is .NET 8.

2.1.4 C#

C# (C Sharp) is a modern type-safe object oriented programming language. It is often used for mobile and web applications and game development. C# has its roots in the C family of programming languages, and share similarities with C, C++, JavaScript and Java. It is statically typed, meaning types are checked at compile-time, not at runtime. C# was developed by Microsoft and introduced along side the .NET Framework [4].

2.1.5 Single page application

A single page application is a web app implementation that loads a single web document. It then updates only the part of the document that needs to be updated, instead of loading a whole new page. Using an SPA can increase performance as it saves time during rendering because there is less that needs to be rendered at once [5].

2.1.6 Vue

Vue is a JavaScript framework for developing web applications and user interfaces that builds on top of standard HTML, CSS and JavaScript. Vue is often used for developing single page applications (SPA) [6].

2.1.7 JavaScript

JavaScript (JS) is a high-level, dynamically typed and frequently just-in-time compiled programming language often used in web development. JavaScript also conforms to the ECMAScript standard, which is a standard for scripting languages, ensuring interoperability of all web pages covering all web browsers. All web browsers execute JavaScript via dedicated engines, and JavaScript also supports multiple programming paradigms. These are event-driven, functional and imperative programming styles. JavaScript have APIs (application programming interfaces) for regular expressions, date, text, the Document Object Model (DOM), and data structures [7].

2.1.8 TypeScript

TypeScript is JavaScript with syntax for types. This makes it easier to detect errors earlier in the editor [8].

2.1.9 Tailwind CSS

Tailwind CSS is a utility-first CSS framework which allows for inline styling. With predefined classes for styling it makes it easy to style components directly in the markup. This approach eliminates the need for writing custom CSS by offering predefined classes for virtually any styling feature, from layout and spacing to typography

and color. Files using Tailwind CSS maintain the .html extension, as the framework is implemented within the HTML structure rather than in separate style sheets [9].

2.2 Data Exchange

2.2.1 REST API

REST, or Representational State Transfer is an architectural style for designing networked applications. It relies on a stateless, client-server, cacheable communications protocol — typically HTTP. In the RESTful model, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the web. REST uses a set of operations, mainly HTTP verbs, such as GET, POST, PUT, DELETE, and PATCH, to manipulate the resources [10].

REST APIs adhere to a set of architectural constraints defined by Roy Fielding in his 2000 dissertation. These constraints include: [11]

- Client-server architecture: REST APIs employ a well-defined client-server model, where clients (applications) make requests to servers (providers) for resources (data).
- Stateless communication: Each request-response interaction is independent, and the server does not maintain any client state between requests.
- Cacheable data: Responses can be cached by intermediaries, such as web proxies, to improve performance and reduce server load.
- Uniform interface: Resources are identified by Uniform Resource Identifiers (URIs) and manipulated through a standard set of HTTP methods (GET, POST, PUT, DELETE).
- Layered system: The architecture can be layered to support different levels of functionality, such as security, caching, and load balancing.

2.2.2 GraphQL

GraphQL is a query language for APIs, offering a server-side runtime for executing queries by using a type system defined for your data. Unlike traditional REST API approaches, GraphQL allows clients to request only the data they need, making it an efficient alternative for developing web APIs. Originated by Facebook to address performance and flexibility issues inherent in conventional REST services, GraphQL has been open-sourced and adopted by a diverse array of companies and projects [12].

Some of GraphQL's key features include precise data fetching, single endpoint, a strong type system and a real-time data subscription updates.

GraphQL fetches only the data that is queried, allowing the client to fetch exactly the data it needs. Often times in REST API you over- or underfetch data. Overfetching results in downloading data that is never used and underfetching means you have to

make multiple requests to get all the data that is needed. Comparatively you only need to send a single request with GraphQL [13].

One benefit is when you have multiple consumers of your API. Each consumer can request the data they need and nothing more without bloating the response to satisfy all parties.

2.2.3 Kafka

Apache Kafka is a distributed streaming platform that enables the processing and management of streams of records in real-time. First developed by LinkedIn to manage high volumes of data—exceeding trillions of messages per day—Kafka has evolved into a widely adopted open-source solution, suitable for various data-intensive enterprise environments.

Kafka’s architecture is designed to provide high throughput for both publishing and subscribing to streams of records, facilitating the storage and processing of these streams as they occur. It is distinguished by its ability to handle data from multiple sources and deliver it to multiple consumers simultaneously, effectively moving vast quantities of data across a distributed network.

Key features include real-time data pipelines for streaming data exchange between systems and applications. This is crucial for operational monitoring and analytics. Kafka’s distributed architecture enables effortless scaling for big data workloads. Notably, it’s versatile and valuable for future-proofing applications in data-intensive fields like IoT and social media [14].

2.3 Data Storage and Security

2.3.1 Database

There are many different variations of databases, but what they all have in common is that it’s a structured collection of information, or data, stored in a computer system. This data is usually managed by what is called a database management system, or DBMS for short. A database system, often just referenced as “database”, is the combination of the database, the DBMS and any other application associated with the two. In modern databases data is mostly modeled in columns and rows in a series of different tables. This makes data querying and processing more efficient. They also mostly use SQL (structured query language) when writing and querying data [15].

2.3.2 PostgreSQL

PostgreSQL is a database management system which is used to create and manage relational databases based on the SQL standard, and is both free and open source [16]. PostgreSQL is equipped with robust transactional support, characterized by ACID (atomicity, consistency, isolation, durability) properties, and includes advanced features like updatable views, materialized views, triggers, and stored procedures. PostgreSQL started as a project at UC Berkley called the POSTGRES Project all the way

back in 1986. Today PostgreSQL is one of the most widespread SQL-based relational databases in the world. It is compatible with all the major operating systems such as Linux, macOS and Windows [17].

2.3.3 Multi-Tenant Environment

Multi-Tenant Environment, also known as software multitenancy is a form of software architecture where multiple tenants are using the same instance which operates on a server. The system is shared, which allows different tenant groups, each with its own set of users, to access a similar instance while maintaining their own specific privileges, data isolation and configuration. The multitenancy approach is distinguished from multi-instance architectures, where distinct software instances are deployed for different tenants. Multi-tenant environments is regarded as a cornerstone feature of cloud computing [18].

2.3.4 Encryption

Encryption is a way of encoding information. The data gets scrambled so only authorized parties are able to understand the information. This is a fundamental technique in cryptography, and the scrambled version of the data is known as ciphertext. The process of encryption ensures that the information is only accessible to the intended party who have the appropriate decryption key, which can revert the ciphertext back to plaintext. Encryption does not prevent the interception of data, but it renders the data unintelligible to those without the right decryption key to decode it [19].

Encryption usually relies on the use of a pseudo-random key generated by an algorithm. It is theoretically possible to decrypt the message without the key, but an intelligently designed encryption plan makes this feat practically impossible.

2.3.5 HTTPS

Hypertext Transfer Protocol Secure (HTTPS), is the secure alternative of HTTP. It is a widely used protocol over the internet to send data between a web browser and a website, and uses encryption for secure communication. The encryption is important when users are transmitting sensitive data, like password login information. The encryption protocol is called Transport Layer Security (TLS), formerly known as Secure Sockets Layer (SSL). The TLS protocol secures communications by using asymmetric public key infrastructure. This system uses two dissimilar keys to encrypt the communications between parties. One key is the private key and is controlled by the owner of the website. The other key is the public key, and it is available to everyone who interacts with the server [20].

2.3.6 Bearer Token

A Bearer Token is a token used in Bearer authentication which is an HTTP authentication scheme commonly used with OAuth 2.0. The token is an encrypted string used for authentication and authorization, and is usually generated by a server in response

to a login request. The Bearer token is generally sent via HTTPS over an encrypted connection [21] [22].

2.3.7 GUID

A GUID is short for Globally Unique Identifier. A GUID is to create a *unique* value and is commonly used for when creating ids for class objects, or any other scenario when an entity needs to be uniquely identified [23].

2.4 Tools

2.4.1 Docker

Docker is a set of “Platform as a Service (PaaS)” products that employ OS-level virtualization to distribute software in units called containers. These containers are isolated from each other, each containing its own software, libraries, and configuration files, yet they can communicate through well-defined channels. As all containers share the same operating system kernel, they consume fewer resources than traditional virtual machines. This efficiency allows a single server or virtual machine to run multiple containers simultaneously, enhancing resource utilization and simplifying application deployment [24].

2.4.2 Postman

Postman is a software tool used for testing APIs. It started as a browser extension in 2012 [25], and has since evolved into a standalone application that allows developers to create, share, document and test API endpoints. Postman supports automated testing which helps validate API performance and reliability. It also offers collaboration capabilities that aids developers in collaborating efforts [26].

2.4.3 DBeaver

DBeaver is a free open source database management tool released in 2011. It is a SQL client software application that uses the JDBC application programming interface to connect with relational databases. It also uses proprietary drivers for NoSQL databases. DBeaver comes with an editor which enhances the user experience by supporting code completion and syntax highlighting. Its architecture is based on the eclipse platform which employs a plug-in system that allows users to extend its functionality greatly [27].

2.4.4 ChatGPT

ChatGPT is an advanced chatbot developed by OpenAI, which was founded in 2015 [28].

ChaptGPT is based on large language models (LLMs) and was launched on November 30 2022. The technology allows users to tailor conversations by adjusting length, format, style, level of detail and language, making it highly versatile for various interactive applications. When interacting with ChatGPT each earlier stage of the conversation is incorporated, which enhances the relevance of the responses [29].

2.4.5 GitHub Copilot

GitHub Copilot was developed by GitHub (a Microsoft subsidiary) in collaboration with OpenAI. It is an innovative code completion tool designed to enhance the productivity in software development. Launched on June 29, 2021, Copilot integrates with popular integrated development environments (IDEs) like Visual Studio Code, Visual Studio, Neovim, and JetBrains, providing advanced coding assistance primarily for languages such as Python, JavaScript, TypeScript, Ruby, and Go. This tool is available by subscription to both individual developers and businesses [30].

2.5 Version Control And Documentation

2.5.1 Git

Git is a distributed version control system that is used by almost 9% of professional developers. It's open source, and was first developed by Linus Torvalds in 2005. Git is designed to track changes in computer files, which helps collaboration among developers. Git is primarily used in software development and supports coordinated work efforts, which allows for efficient management of source code. Git's design prioritizes data integrity, speed and the facilitation of non-linear, distributed workflows, which enables thousands of parallel branches to operate at the same time across different systems [31].

2.5.2 GitHub

GitHub is a developer platform based on Git software. It allows developers to store, create, share and manage their code. Since it's based on Git software it provides the distributed version control of Git, but it also gives users access control, software feature request, bug tracking, continuous integration, task management and wikis. These features streamline the development process, which allows for more efficient project management and cooperation among developers [32].

2.5.3 NGINX

NGINX is a high-performance web server and reverse proxy that enhances web application efficiency and security. It facilitates rapid content delivery and is optimized for handling simultaneous connections with minimal resource usage. NGINX supports multiple web technologies, acting as a mediator between client requests and server responses. In our Vue.js project, NGINX was employed to serve static files, man-

age SSL/TLS termination, and route API calls, streamlining both development and production environments [33].

2.5.4 Overleaf

Overleaf is an online LaTeX editor with real-time collaboration and version control used to produce scientific documents [34].

2.6 Development Methodologies

2.6.1 SCRUM

Scrum is a lightweight framework that enhances team collaboration in complex project environments. Originating in the early 1990s, it promotes an empirical approach to problem-solving, emphasizing decision-making based on real-time observations. Scrum structures work into sprints, typically one month long, to foster regular planning, execution, and evaluation within a flexible yet disciplined framework.

A Scrum cycle typically consists of:

1. "A Product Owner orders the work for a complex problem into a Product Backlog." [35]
2. "The Scrum Team turns a selection of the work into an Increment of value during a Sprint." [35]
3. "The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint." [35]
4. "Repeat" [35]

Scrum artifacts, including the Product Backlog, Sprint Backlog, and Increments, come with commitments that enhance transparency and guide the team toward achieving specific goals. These elements together create a dynamic environment where teams can self-manage and adapt to changes rapidly, making Scrum ideal for managing complex projects effectively.

2.6.2 Agile Methods

Agile methods is a philosophy that centers around continuous incremental improvement through small and frequent releases. This makes for a development strategy that is very flexible to change and that can continuously improve and adapt [36].

The Agile manifesto outlines four values:

- "Individuals and interactions over processes and tools" [36]
- "Working software over comprehensive documentation" [36]
- "Customer collaboration over contract negotiation" [36]
- "Responding to change over following a plan" [36]

3 Method

3.1 Managing the project

To organize and manage the project we used an agile methods philosophy with a SCRUM-like approach. Where the the project is divided into week-long sprints. The sprints start with *sprint planning/startup*, where we set goals for the week, and tasks are put into- and distributed from a *sprint backlog*. There is also an overarching *product backlog* with tasks that eventually will be done. The product backlog is continuously updated and refined throughout the development process.

At the end of the sprint, a *sprint retrospective* is held, where we discuss what went well, and what we could have done better. The retrospective is followed by a *sprint review*, where everyone talk about and showcase what they did during the sprint.

3.1.1 Meetings

We met with the supervisor assigned to us by NTNU for the project bi-weekly on Thursdays. During these meetings we discussed our progress regarding the project as a whole.

The group also met with FiiZK bi-weekly on Fridays, where we presented and discussed our progress on the product and got feedback, as well as help and tips if needed.

Communication regarding meetings with the supervisor and FiiZK were over email, and communication between the group members happened through discord on a private server made for the project.

Originally sprint review and retrospective were on Fridays in a separate meeting, and the sprint startup on Monday. However, due to the group members often working on the project also on weekends, the group figured it would be best to just have one big meeting on Monday where we had all of the sprint-discussions. This meant that the sprint didn't end until sprint review and retrospective was completed on Monday, and then the next sprint started directly afterwards.

3.1.2 Project Organization

To organize the project files and links, Confluence was used as a project "homepage". Confluence stored all of the meeting notes, retrospectives and various documents related to the project.

We used Jira to organize the sprints and keep track of the backlog, logged hours, tasks, and user stories.

Jira and Confluence were chosen largely because they were provided free-of-charge by NTNU.

3.1.3 Dividing Tasks

As mentioned in Section 3.1.2, Jira was used to organize the sprints. During the sprint startup phase, a new sprint was created in Jira, and tasks, or *user stories*, were written as “As a [persona], I [want to], [so that].”, from the Atlassian user story examples [37]. A story is usually divided into one or more sub-tasks.

The group considered having the sprints either 1 week or 2 weeks long, but ultimately chose to have 1 week sprints. This is to keep us more engaged, as we felt that with 2 weeks sprints it would be easier to procrastinate.

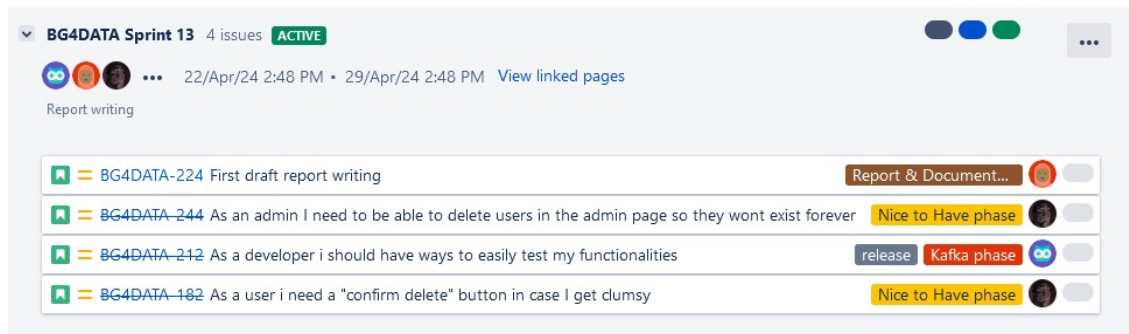


Figure 1: Example over divided user-stories for a sprint in Jira.

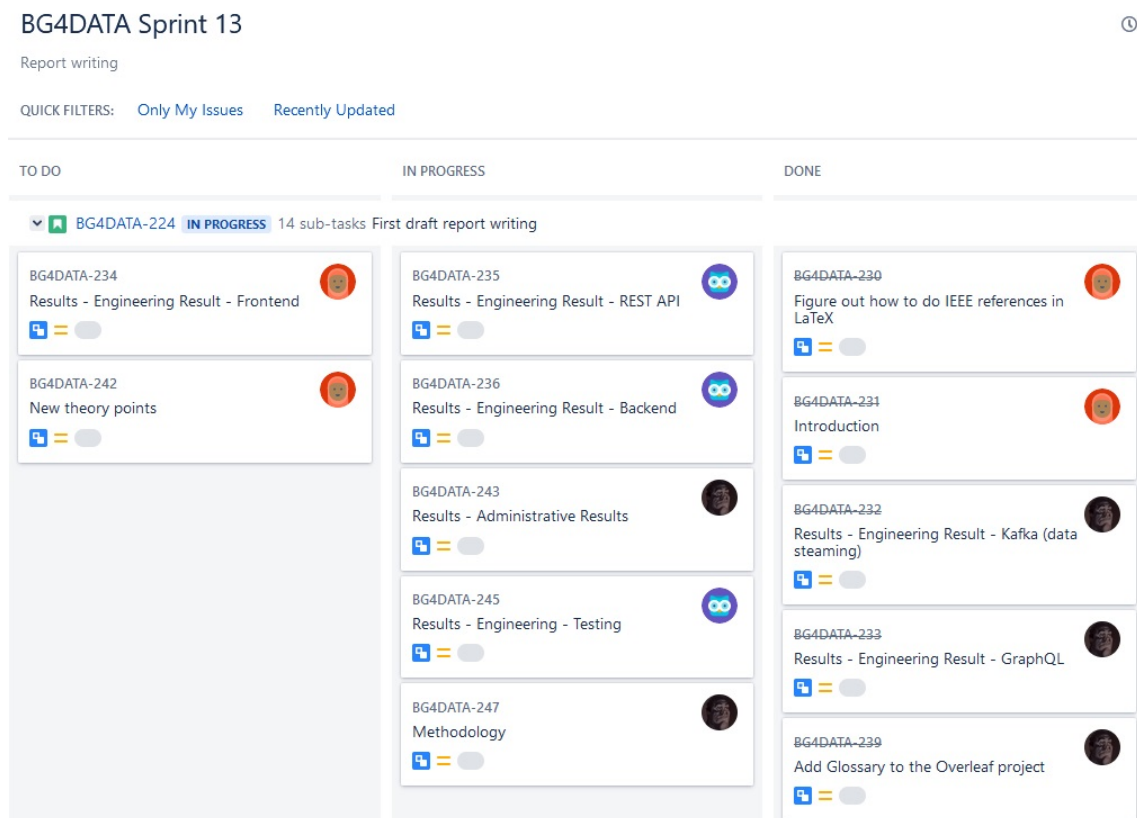


Figure 2: Example overview over sub-tasks within a user-story in Jira.

Note regarding Figure 1 and Figure 2:

The screenshot was taken after the product development was over, and report writing

was the main focus. The “nice to have”-phase was added for for additional and less critical functionality.

3.2 Planning and Choosing Technologies

Before we started the development, we made a preliminary project plan which you can see in Appendix A. The project plan gave an early overview over the group structure, initial milestones, risk assessment as well as a general idea of how we would approach the development process.

The planning process consisted of choosing which frameworks to develop our full-stack application with, choosing methods of data transfer, designing how the website would look and making an outline of when we would reach the different milestones.

3.2.1 Frameworks

The frameworks we chose for the application was almost fully decided upon before our first meeting with FiiZK. In the project description and requirements seen in Section 1.3, they informed us that their preferred technologies were .Net 7, Vue, TypeScript, MSSQL and Docker.

The only changes were that we used .Net 8 and PostgreSQL. We chose .Net 8 due to it being the most recent version of the .Net framework. PostgreSQL was chosen over MSSQL due to FiiZK informing us that they wanted to transition away from MSSQL to PostgreSQL.

3.2.1.1 Framework Overview

Backend

- .Net 8
 - ASP.NET Core
 - EntityFrameworkCore
 - ASP.NET Core.Identity - For authentication
 - Confluent.Kafka - For Kafka
 - Hot Chocolate - For GraphQL
- C#
- PostgreSQL

Frontend

- Vue.js 3
 - TypeScript
 - Composition API
 - Components used from shadcn-vue

Containerization

- Docker

3.2.2 Methods of Data transfer

Similarly to how we chose the frameworks as described in Section 3.2.1, the methods of data transfer were primarily chosen from what was suggested by FiiZK.

We chose to use REST API and GraphQL due to a combination of the suggestion, as well as the group having worked with REST in previous projects. GraphQL was chosen as it is often considered as a popular alternative to REST.

In our 3rd meeting with FiiZK, we started discussing a possible 3rd alternative, due to how quickly we made progress with the REST solution. They suggested an event-based method of data transfer that would use data-streaming, specifically recommending RabbitMQ or Kafka, rather than the request-response model employed by REST and GraphQL.

3.2.2.1 RabbitMQ or Kafka?

None of us were familiar with event based data streaming, so we firstly employed the “google to see what is most relevant”-method. As seen in Figure 3, we found out that according to *google trends*, Kafka was the most popular and relevant of the two.

After researching Kafka more, we realized Kafka probably would be excessive for our use-case, due to its complex architecture and scalability features that exceed our data volume and processing needs. Despite this, we still chose Kafka over RabbitMQ, primarily because we got the impression that learning Kafka would also give us insight into how to use something simpler like RabbitMQ, but not necessarily the other way around.

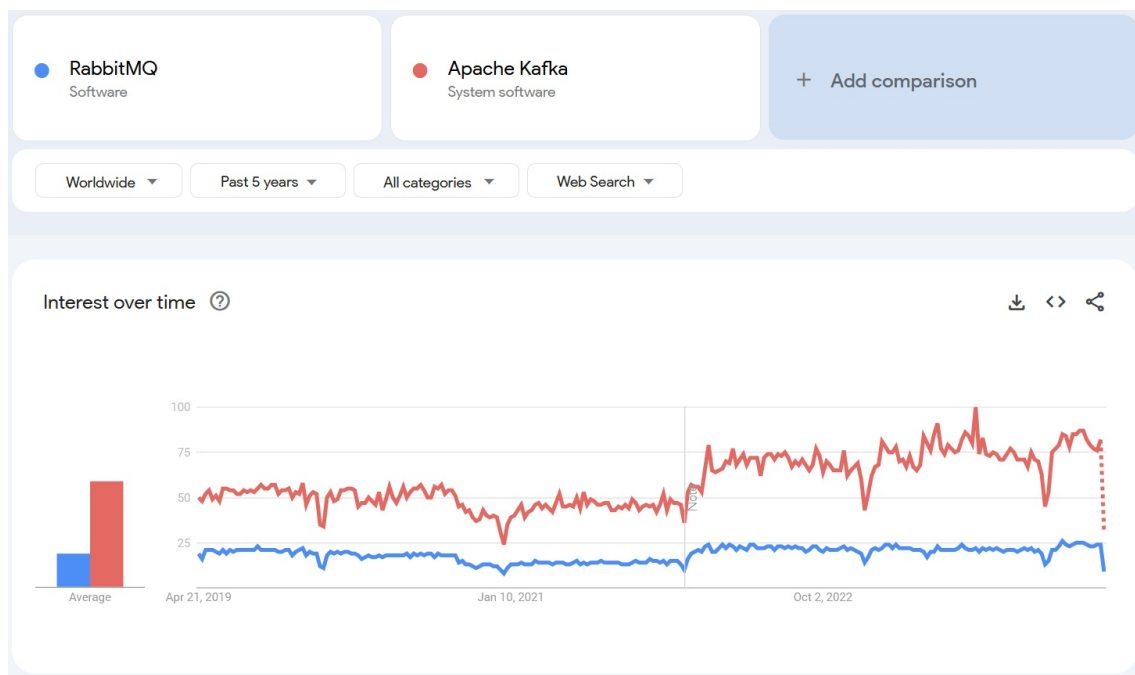


Figure 3: Google search trends for RabbitMQ vs Apache Kafka

3.2.3 Milestones

The project was set up with different milestones for when we were supposed to be done with the different parts of the project, like the date for when REST API solution, GraphQL solution and the undecided 3rd solution which we would do if we saw we had enough time.

The different solutions would have a 3-week development period, with an additional 3 weeks as a "buffer" to clean up and finish the project. The remaining weeks would be spent on project writing.

3.2.3.1 Initial Project Milestones:

Date	Milestone
26.01.2024	Set up Github
26.01.2024	Make Wireframes for the frontend
16.02.2024	Complete REST API solution
08.03.2024	Complete GraphQL solution
29.03.2024	Undecided 3rd solution
23.04.2024	Finish Product Development
07.05.2024	Finish first-draft of report
21.05.2024	Finish Report

Table 1: Initial Project Milestones.

3.2.4 Wireframes

The wireframes for the frontend were completed using Figma. It was important to complete the wireframes quickly, since they would help with the planning process and make sure our group and FiiZK were on the same page when it came to how they wanted the product to look and function.

3.3 Development

The development process was planned from easy to hard, in regards to how complex the solutions would be to implement.

The choice was between developing the REST or the GraphQL solution first. The group chose REST API due to the groups previous experience using REST API.

As mentioned in Section 3.2.2, the third method of data transfer wasn't decided upon until we had already started the development process, and was only chosen after we were confident the GraphQL solution would be completed close to the milestone deadline.

The programming was done with the SOLID principles in mind. This was to ease the development process by ensuring our code is robust, maintainable and scalable.

3.3.1 REST API

For the REST API development, we used the `AspNetCore` library to create the API endpoints. Our backend structure included various layers such as controllers for handling requests, services for business logic, and models for data representation. `AspNetCore.Identity` was utilized for authentication, ensuring secure user access and data protection.

3.3.2 GraphQL

We adopted the `Hot Chocolate` library, a .NET platform for building GraphQL servers. `Hot Chocolate` seamlessly integrated with our existing `AspNetCore` infrastructure, allowing us to develop complex queries and mutations efficiently. This setup enabled a flexible and powerful API layer that could handle varied client data requests dynamically.

`FiiZK` also made a request to create a *secret field*, which users cannot see in the GraphQL schema, and is only accessed by the administrators of the application. This was in case a class table contains sensitive information they do not want to share.

3.3.3 Kafka

For event-based data streaming, we utilized `Kafka` with the `Confluent.Kafka` library. While our project does not require the processing of high volumes of data in real-time, `Kafka`'s robustness makes it an excellent choice for reliably handling data streams. This ensures our system's ability to manage data efficiently, even with lower throughput, providing a stable foundation for future scalability. `Kafka`'s architecture allows us to decouple data streams from system processes, enhancing fault tolerance and improving the overall reliability of our application's communication infrastructure.

3.3.4 Applications used

Postman was used to easily test different endpoints without having to interact with the frontend application. This way we could test the endpoints before we had that part of the frontend developed, and more quickly test a specific endpoint without having to go through the frontend. In the end we made a fully working postman collection that can be run and shows which endpoints are working as intended or not.

VS Code was the IDE of choice for us, because it is versatile and familiar. This was also the recommended IDE for `Vue 3` development. `VS code` extensions that were needed were `Volar (Vue - Official)` for frontend, and `C# Dev Kit` for backend.

We used **DBBeaver** to have a more in depth look at our database. With this program we can for example visualize single tables or a whole ER - diagram for our databases.

To containerize the project we used **Docker Desktop**, which is a program made for building, running, and managing containerized applications on local machines.

3.3.5 API overview

We used Swagger UI and GraphQL's UI that comes with the HotChocolate library during development. This made it very easy to get an overview over the different endpoints and methods, and what schema they use.

3.3.6 Project structure

3.3.6.1 Frontend project structure

The frontend Vue 3 project has a lot of configuration files for vite, prettier, tailwind and ESLint. The vue components are organized into different folders for each of the three data retrieval methods.

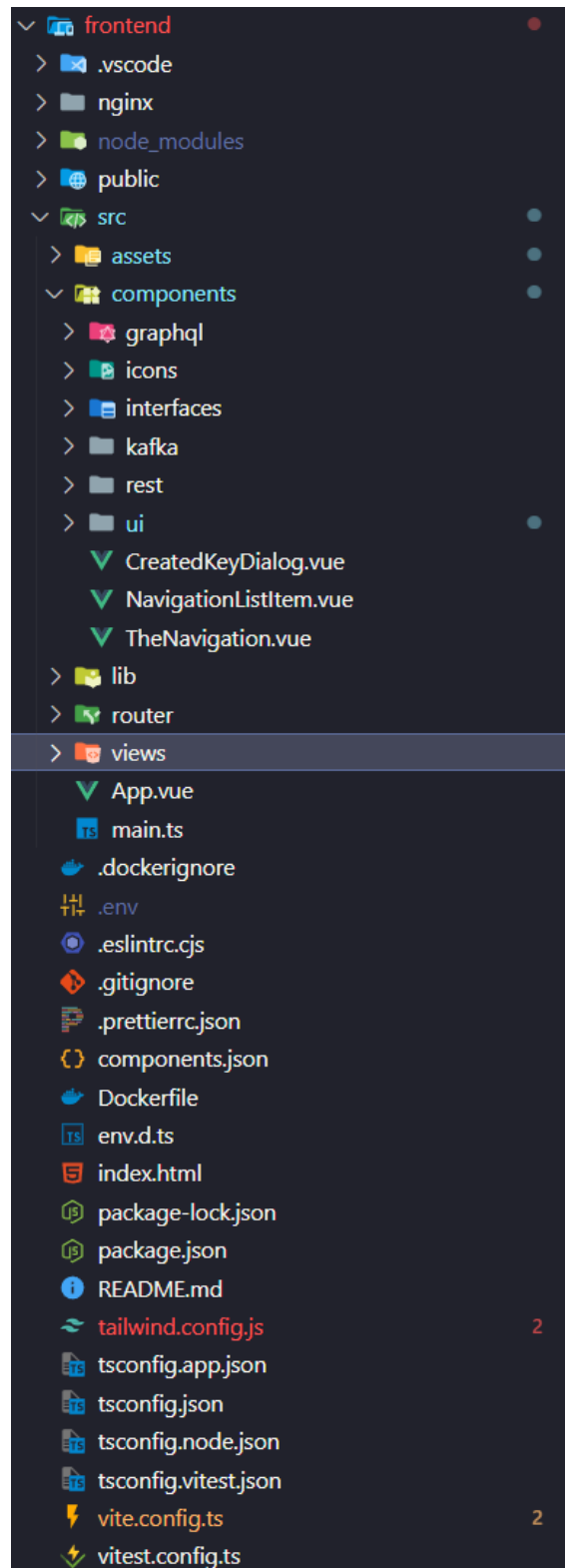


Figure 4: Frontend project structure

3.3.6.2 Backend project structure

The backend folder contains 3 different .NET projects. The main backend, the mock-sensor and the testing project. The backend.sln solution file has references to all of these 3 projects. Inside the backend we try to keep everything nice and organized

into different subfolders for the different parts of the project.

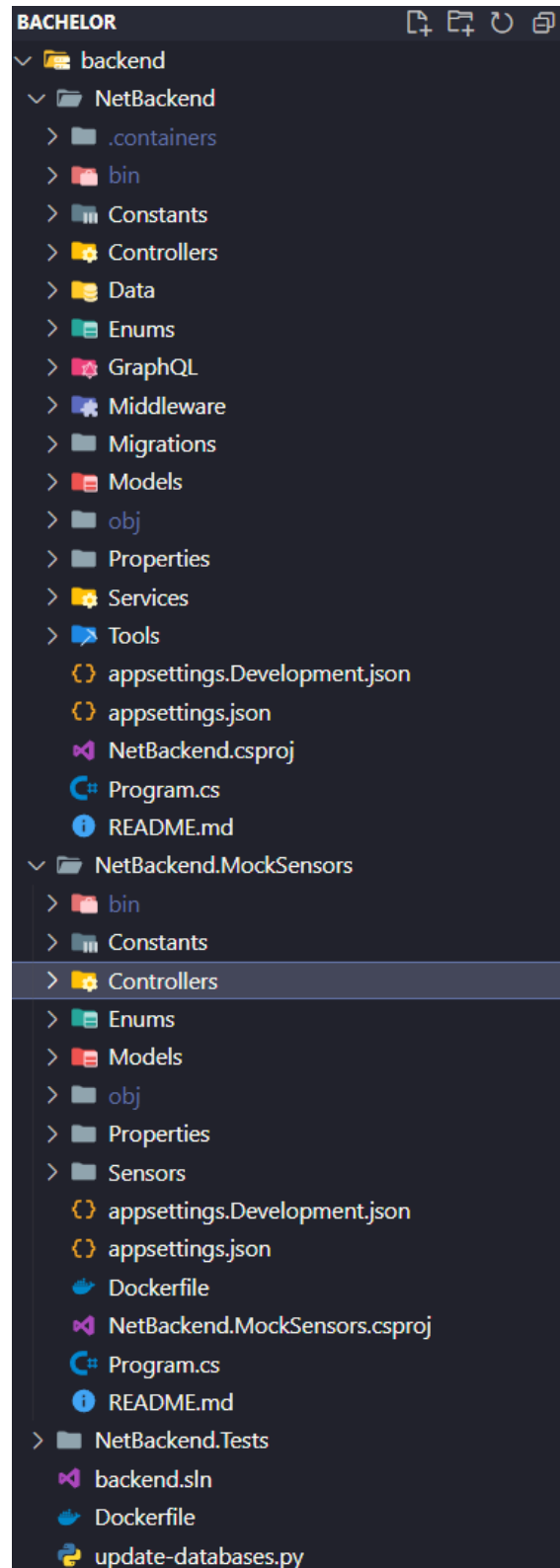


Figure 5: Backend project structure

3.4 Running the Project

For running the project we used docker desktop, which runs all of the different containers the project consists of.

The README.md file shown in Figure 6 explains how to run the project and will be discussed further in this chapter. The README file can also be seen in the GitHub repository found using *this link*, or seen in the references [38].



Figure 6: Root directory README file

3.4.0.1 Prerequisites

Before initiating the project, ensure the following prerequisites are installed on your system:

- **Docker Desktop:** For running containerized applications.
- **Node.js:** Required for the frontend Vue application.
- **.NET 8.0 SDK:** Necessary for backend services.

3.4.0.2 Running the Project with Docker

From the root directory of the project, execute the following command to build and run the project using Docker:

```
docker-compose up --build
```

This command builds the Docker images for the frontend, backend, and any services like Kafka and PostgreSQL, and starts the containers.

3.4.0.3 Accessing Application Components

Once the containers are running, you can access the various components of the application using the following URLs:

- **Main Backend REST API Swagger Documentation:** <http://localhost:8088/swagger> - Provides an interactive interface for testing and documenting the REST API.
- **Backend GraphQL Interface:** <http://localhost:8088/graphql> - Access the GraphQL playground to execute queries and mutations.
- **Frontend Application:** <http://localhost:8080/> - The main interface for the application, built with Vue.
- **Mock Sensor Controls:** <http://localhost:8089/swagger> - Provides swagger docs of the different REST API endpoints for the Mock Sensor.

3.4.0.4 Additional Setup Instructions

For specific setup instructions regarding the frontend and backend components, refer to their respective README files in the GitHub repository.

3.4.0.5 Database Setup

To update or modify the database schemas:

1. **Install dotnet-ef tool:** `dotnet tool install --global dotnet-ef`
2. **Run the database update script:** `python update-databases.py`

3.4.0.6 Logging in

To test different user functionalities, use the pre-configured user credentials:

- **Admin Access:**
 - Email: `admin@mail.com`
 - Password: `Password!1`
- **User Access:**
 - Email: `test@mail.com`
 - Password: `TestPassword1!`

4 Result

4.1 Engineering Results

The engineering results section contains the result of our development. The backend subsection contains information that is shared between all solutions, as well as general information about the the backend architecture.

Each solution has its own subsection that goes more in-depth, with specific results unique to that solution.

The frontend subsection contains information about how we developed the GUI.

This section also contain information about how we containerized the application in Docker, and our testing-results.

Here is a small summary of the problem and requirements outlined in Section 1.2 and Section 1.3:

- Must be a multi-tenant environment of multiple users.
- The solution must offer users various methods of retrieving their data.
- There must be a GUI to manage access to the data.
- A user must be able to share a subset of of their data to third parties.
- The solution should be built as a containerized application.

4.1.1 Demo Video

To showcase the different features of our application, we made a demo-video you can watch using *this link*, or see references [39].

4.1.1.1 Use Case Diagram

Three different use case diagrams were developed to visually illustrate the interactions between the different user roles and the system within the different data retrieval methods. The use case diagrams features three primary actors: Admin, User and Third Party.

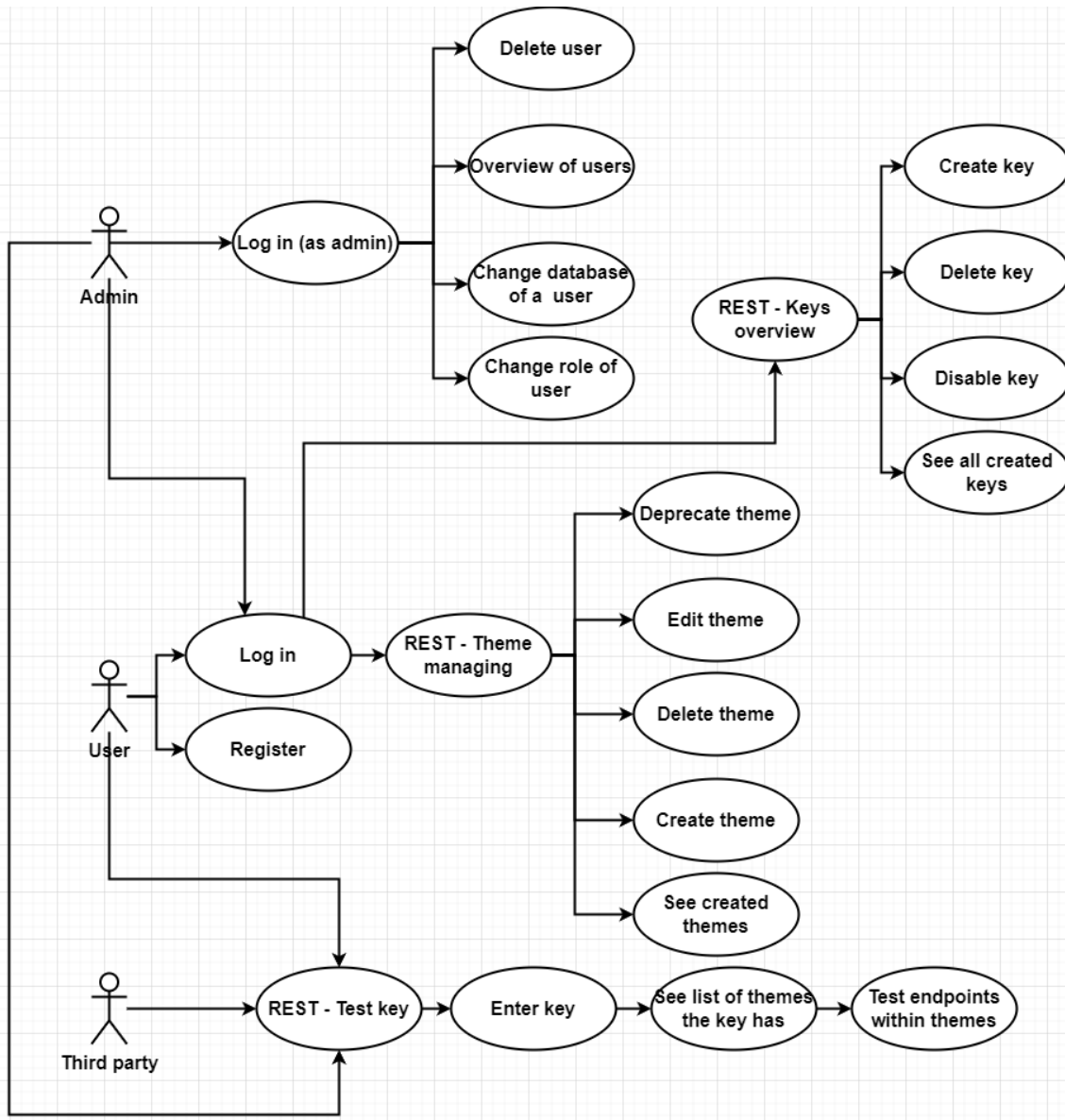


Figure 7: REST API Use Case Diagram

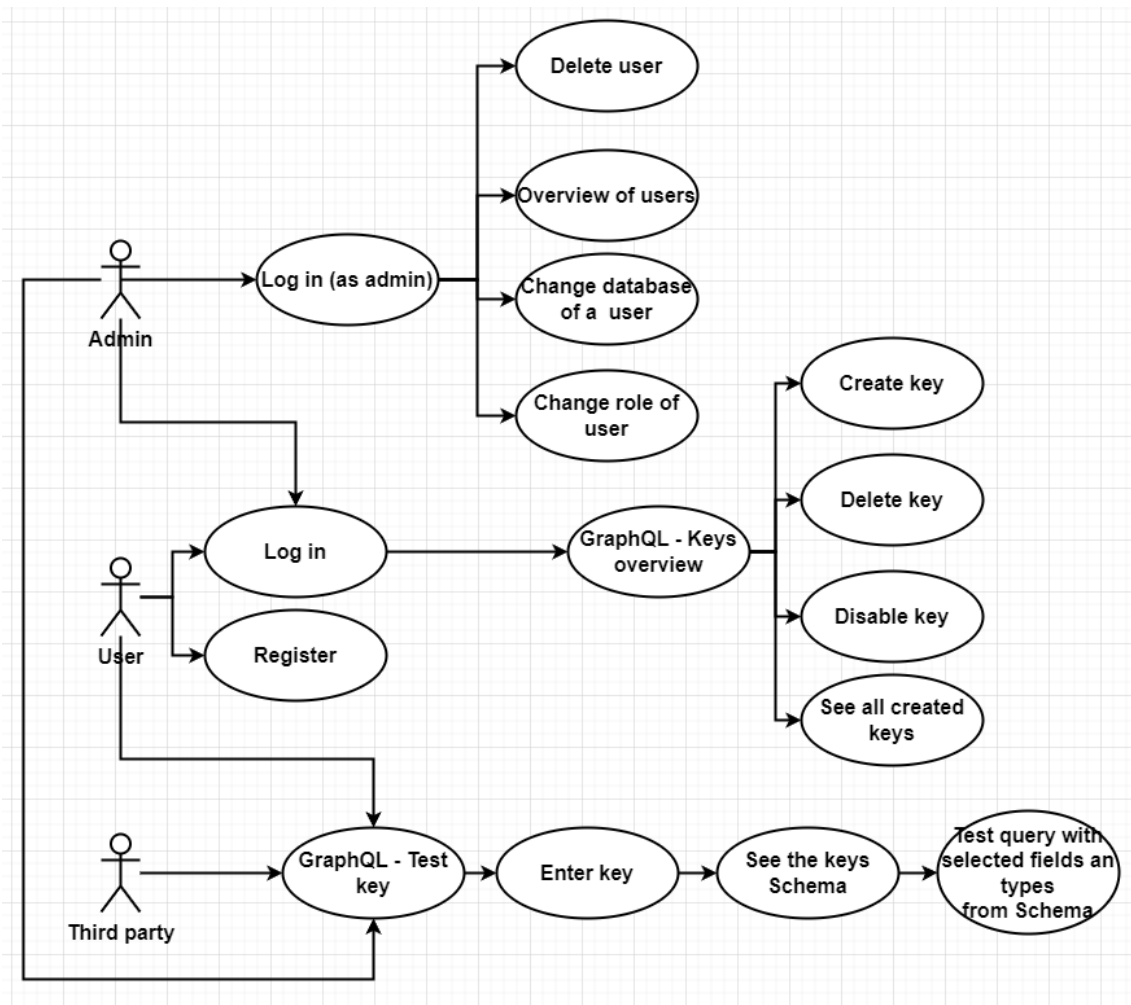


Figure 8: GraphQL Use Case Diagram

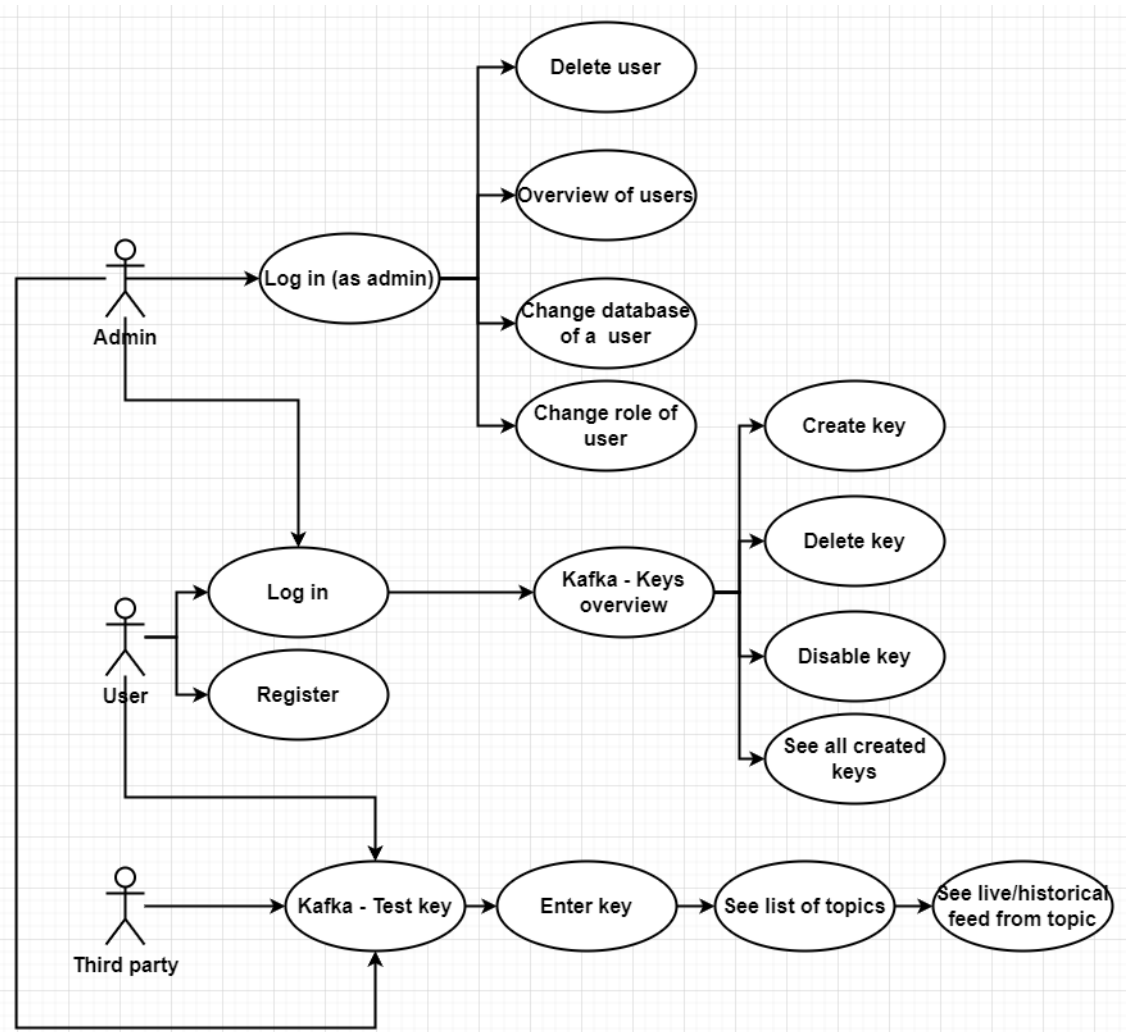


Figure 9: Kafka Use Case Diagram

- Admin: After logging in, the Admin has access to four extra functionalities: "Overview of Users", "Change Database of a User", "Change role of user" and "Delete user". These functions allow the admin to manage all registered users and modify the database access privileges of any user.
- User: Users begin their interaction with the system through the "Register" and "Log in" use case. Once authenticated, they have several actions available for creating and managing keys. For the REST API solution the user can also create and manage themes.
- Third Party: This actor represents external users who have the ability to see what data is provided with a particular key. The "solution - Test Key" use case is their sole interaction with the system, allowing them to test and confirm the data available through their given access key.

The use case diagram helped to further clarify the scope of the project, and it outlined the necessary actions that each type of user can perform within the application. It became a foundational tool for the development process by providing a high-level view of the system's functionality. It also informed the security model, ensuring that appropriate authentication and authorization checks were in place for sensitive functions

such as user management and key creation.

4.1.1.2 Architecture Diagram

Similarly to the use-case diagram, an architecture diagram was developed to visually illustrate the different components in the system, and the interactions between them.

The architecture diagram shows how the main .Net backend acts as the only direct link between the Webserver, and all the different backend components.

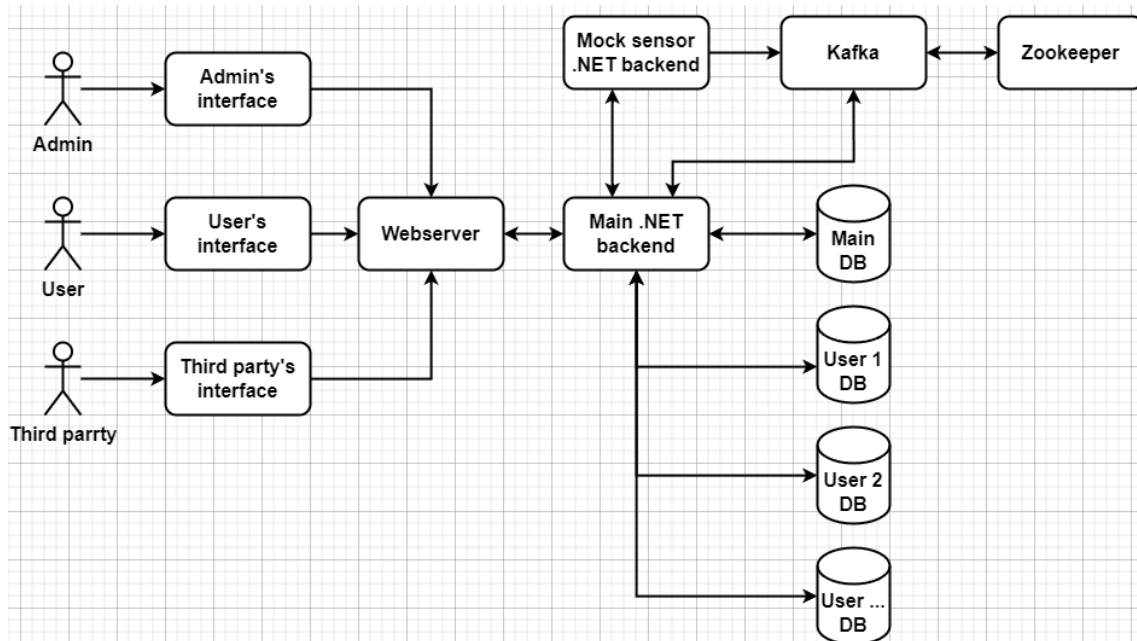


Figure 10: Architecture diagram of our application

4.1.2 Frontend

When designing our frontend solution we first started creating wireframes to get a visual representation of components and the overall aesthetic of the application before delving into coding. These wireframes helped guide the development process.

Customer Page

Fish Health API

Getting started - Components - Documentation -

GET AquaCultureLists X

GET CodSpawningGround X

GET DiseaseHistory X

GET ExportRestrictions X

```
{
  "id": 384,
  "placeName": "Mettefjorden",
  "information": "Lokalt viktig gytefelt",
  "areaDescription": "Lav eggletthet (1), lav retensjon (1)",
  "origin": "Havforskningsinstituttet",
  "bmvalue": "B",
  "value": 4,
  "registeredDate": "2024-01-25T15:42:49.867Z",
  "geometry": {
    "type": "Point",
    "coordinates": [
      10.4,
      60.5
    ]
  }
}
```

Figure 11: Customer Page Wireframe

Admin Page

Home Theme Creator

Sign In/Out

Keys

Name / description	Key
Rema1000000	12j*****g3j
Banan	gg2*****3j
Feesh	6h0*****7j

Create key
Create theme here. Click save when you're done.

Description

Select themes

Themes

- AquaCultureLists
- CodSpawningGround
- DiseaseHistory
- ExportRestrictions

Figure 12: Keys Page Wireframe

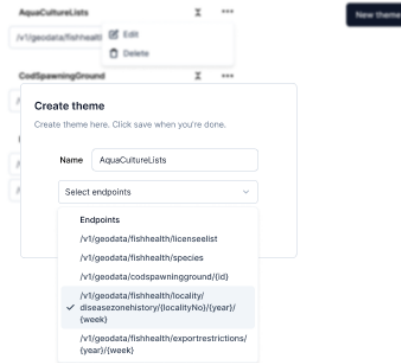


Figure 13: Theme Page Wireframe

Through feedback from FiiZK we were able to refine our approach and align the user interface more closely with their needs and expectations.

The frontend interface is designed to suit the different needs of different user roles through specialized pages and functionalities. Users are first met with a login page where they can log in, register an account or continue using the testing pages as a third party. The page provides a simple and secure entry point for users to access their dashboard.

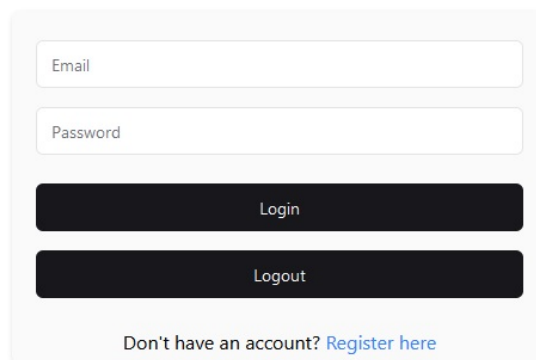


Figure 14: Login page

If users don't have an account and want to register one, they have the ability to do so at the register page.

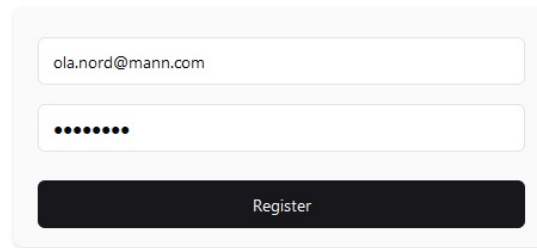
A registration form with a light gray border. It contains two input fields: the first is for an email address, containing 'ola.nord@mann.com'; the second is for a password, shown as a series of dots. Below the fields is a dark gray button with the text 'Register' in white.

Figure 15: Register page

The testing pages are accessible to everyone, including third-party users who do not have to log in to access them. As shown in the pictures above, non-logged in users can access the testing pages from the navigation bar. The purpose of these pages is to allow external users to test the access keys they have obtained. This functionality ensures that third parties can verify what data their keys can retrieve before they integrate their systems or applications.

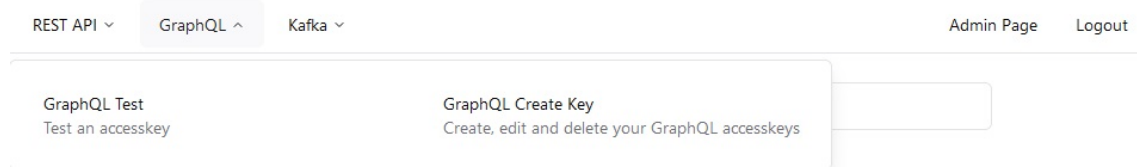


Figure 16: Navigation bar after login

As shown from this dropdown menu from the navigation bar, once users log in, they are provided with a variety of functions related to key management.

The "Create Key" page is one of the crucial features available only to authenticated users. Here, users can generate new access keys based on their needs and permissions. This page includes intuitive forms to input key parameters and immediate validation feedback to ensure users understand their input effects. The ability to see and manage all keys they have created is also provided through a user-friendly interface, allowing for easy tracking and modification of key permissions and details.

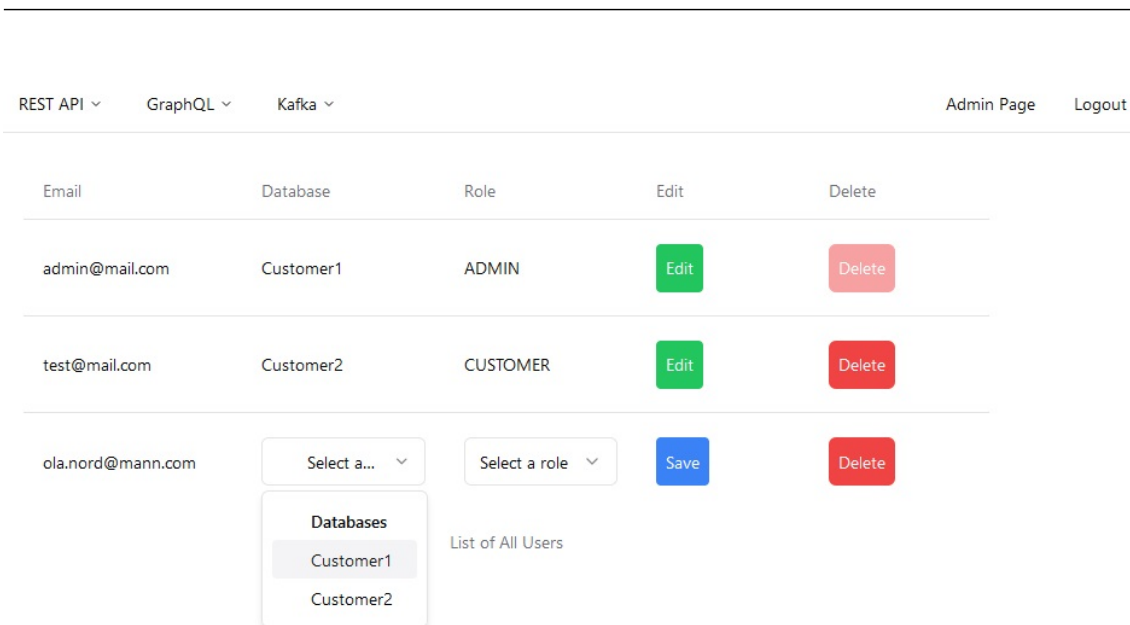


Figure 17: Admin Page when editing a user

The admin page is a central tool for administrative users to manage the entire system. Only accessible after admin login, this page allows for comprehensive management tasks including an "Overview of Users" where admins can see all registered users, edit their details, or delete users as necessary. Additionally, admins can modify user database access privileges through the "Change Database of a User" functionality. These tools are designed to ensure that administrators can effectively oversee the system's operation and maintain user data security.

4.1.3 Backend

4.1.3.1 Backend overview

The backend system consists of multiple components, as seen in Figure 10. These include the main .NET 8 application, PostgreSQL databases, a Kafka cluster and a separate mock sensor .NET application.

User management and authentication is done using the Microsoft ASP.NET Core Identity library. The library is an "Is an API that supports user interface (UI) login functionality." and "Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more." [40].

The Identity library comes with a default *IdentityUser* class, which we slightly modified to contain additional information to tie the user to its personal database, and the keys made by that user, as seen in Figure 18.

Both the Kafka Cluster and mock sensor .Net application has no form of authentication, this is because all authentication is centralized and handled in the main .Net application. There was no security implemented to the mock sensor application or the Kafka Cluster because all of the communication with the frontend goes through the main .Net application, as seen earlier in Figure 10.

```
34 references | You, 2 minutes ago | 2 authors (You and others)
public class UserModel : IdentityUser
{
    11 references
    public string? DatabaseName { get; set; }

    // REST
    1 reference
    public List<RestApiKey>? RestApiKey { get; set; }
    1 reference
    public List<Theme>? Themes { get; set; }

    // GraphQL
    1 reference
    public List<GraphQLApiKey>? GraphQLApiKey { get; set; }

    // Kafka
    1 reference
    public List<KafkaKey>? KafkaKeys { get; set; }
}
```

Figure 18: The modified IdentityUser class.

4.1.3.2 Similarities between the solutions

As mentioned, the solution must give the user the ability to share a subset of their data to third parties. We solved this by giving the user the ability to create keys, that contain the information accessible by anyone using that key.

The user info, as well as the Rest, GraphQL and Kafka keys are stored in the main database, as seen in Figure 21. We have separate databases for each user containing only data related to that user, seen in Figure 22.

A core functionality between the solutions is creating the key. The encrypted keys are made using Advanced Encryption Standard (AES) and converts it into a base64 string. The data that is encrypted is the key objects id and type, either ApiKeys (for REST), GraphQLApiKeys or KafkaKeys seen in Figure 21.

Once created, the encrypted key is converted to a hash, which is stored in the database, and not the encrypted key itself. This is for security reasons, where even if the database was compromised, the encrypted key cannot be retrieved.

Because the hash of the the encrypted key is stored, and not the key itself, the key is lost after its initially created, unless copied and stored somewhere else.

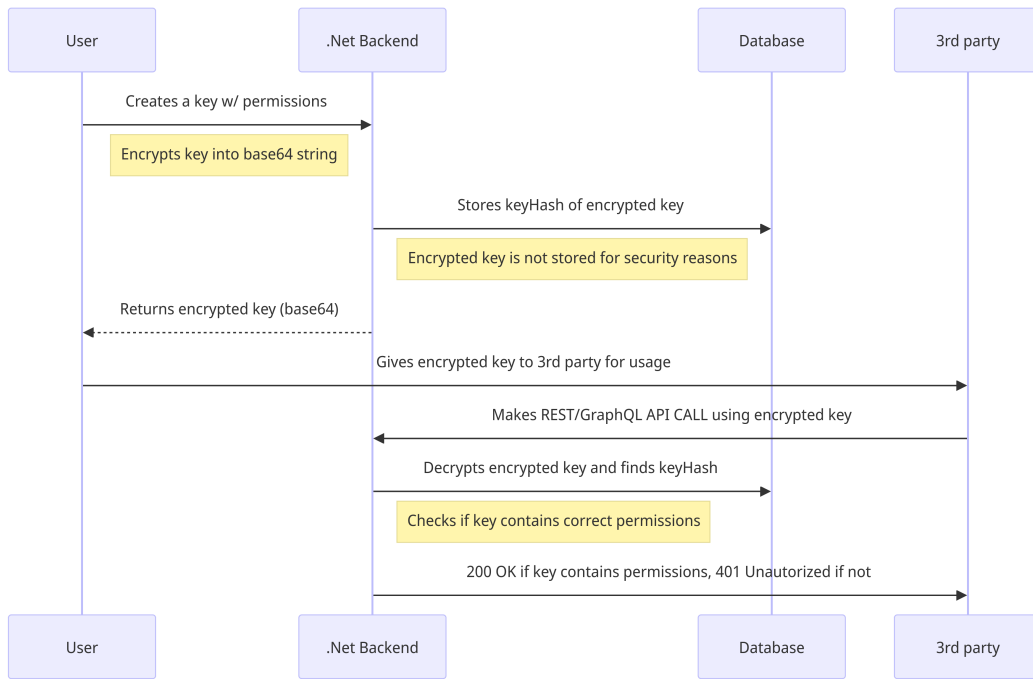


Figure 19: Sequence diagram of how a key is created and used.

***p5QfXJvrTmMMdJx9AZRd90un+LFMQkbsnx811s7AqLyIZvw6c0NCOzS9c
3nOZSYQqcjN/caBIYvjFWtnpHcw==***

Figure 20: Example of what an encrypted key could look like.

Every time the key is used for something, the key is decrypted and the key id and type is found. Then the key object is located in the database, and if found, it checks if the key is used for something it has access to or not.

A default expiration date of 90 days is also added to the keys. This is common in case of keys being forgotten and containing information that isn't meant to be accessible anymore.

The main database, with an ER-diagram seen in Figure 21, already has two users by default, an admin user and a normal "customer" user, their login info can be seen in paragraph 3.4.0.6.

The "customer", or "user" PostgreSQL databases are populated with dummy data, consisting of 100 object of both the Species class and the Organization class seen in Figure 22.



Figure 21: ER diagram for our main database

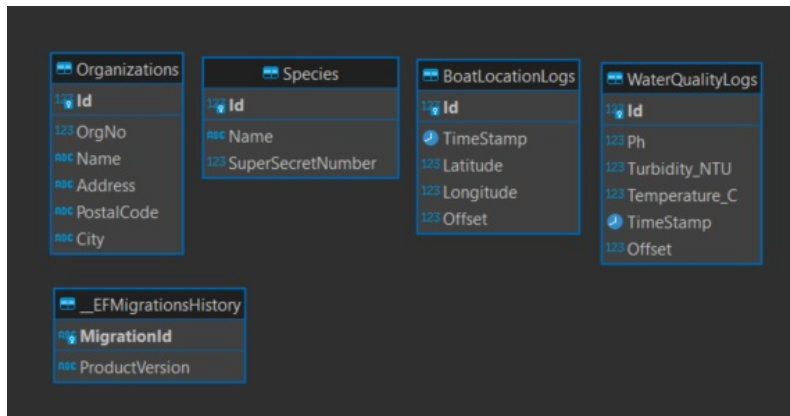


Figure 22: ER diagram for the private databases assigned to a user

4.1.4 REST API Solution

In the REST API solution, we implemented functionality for making keys containing the REST API endpoints, that anyone with access to that key, can use.

To accommodate for scenarios where the application has multiple endpoints, we implemented themes, which are a collection of endpoints. A key can then be created with one or multiple themes and gain access to the endpoints that those themes contain. This way the user can, for example, make a theme called "Aquaculture" and group all the endpoints that fit that theme.

With the rest solution we have created the endpoints that we need for our solution, these include operations for creating, getting, updating and deleting keys and themes.

4.1.4.1 Theme overview

On the frontend theme page you can create themes, see a list of your created themes, edit themes, delete themes and deprecate themes. When creating a theme you choose a name for the theme and which endpoints you want to group with the theme. The list of endpoints you can choose for a theme is the available data retrieval endpoints that the third party would have access to. Deprecating a theme disables the theme from being used when creating new keys, but doesn't delete the theme.

Create Theme

Aquaculturelist



/api/aquaculturelist/fishhealth/licenseelist

/api/aquaculturelist/fishhealth/species

Test endpoints (deprecated)



/api/rest/accesskey-themes

/api/rest/accesskey-rest-endpoints

All endpoints



/api/aquaculturelist/fishhealth/licenseelist

/api/aquaculturelist/fishhealth/species

/api/rest/accesskey-themes

/api/rest/accesskey-rest-endpoints

Figure 23: Frontend GUI of the themes overview.

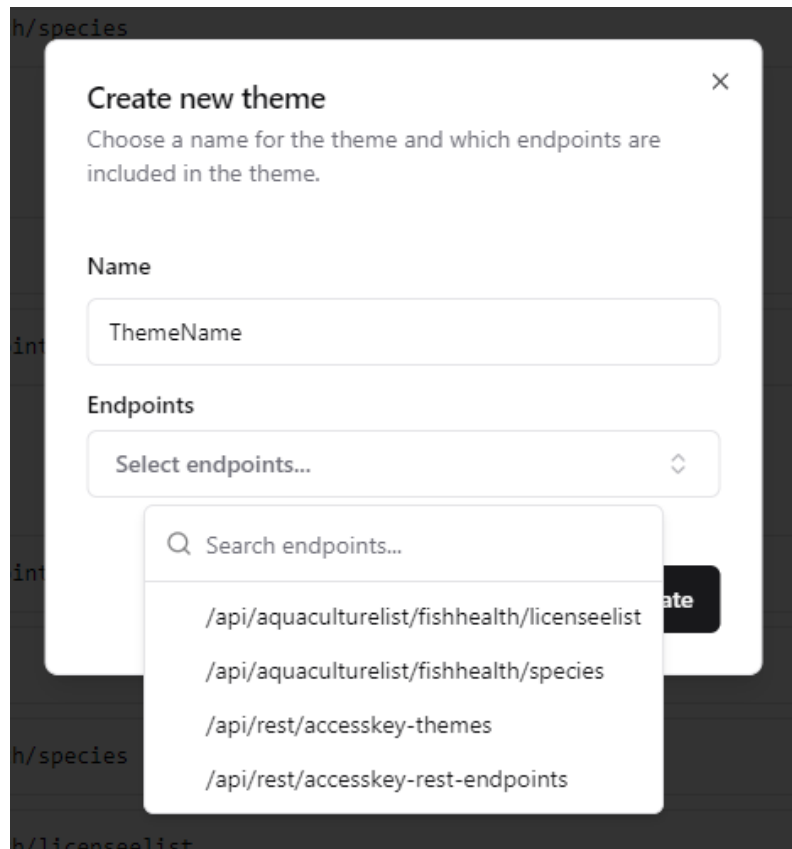


Figure 24: Frontend GUI for creating a theme.

4.1.4.2 Key overview

General information and disable and deleting actions for each key is shown on the REST key overview page. This is also where you are able to create a REST key. After successfully creating a key you get a dialog showing you the encrypted key that was made.

Your API keys

Create new key

Name	Themes	Expires in (days)	Disable	Delete
	2 themes ^			
	Aquaculturelist			
	<code>/api/aquaculturelist/fishhealth/licenseelist</code>			
	<code>/api/aquaculturelist/fishhealth/species</code>			
Key1	All endpoints	90	Disable	Delete
	<code>/api/aquaculturelist/fishhealth/licenseelist</code>			
	<code>/api/aquaculturelist/fishhealth/species</code>			
	<code>/api/rest/accesskey-themes</code>			
	<code>/api/rest/accesskey-rest-endpoints</code>			
Key2	1 theme v	90	Disable	Delete

REST API Keys Overview

Figure 25: Frontend GUI of the REST key overview.

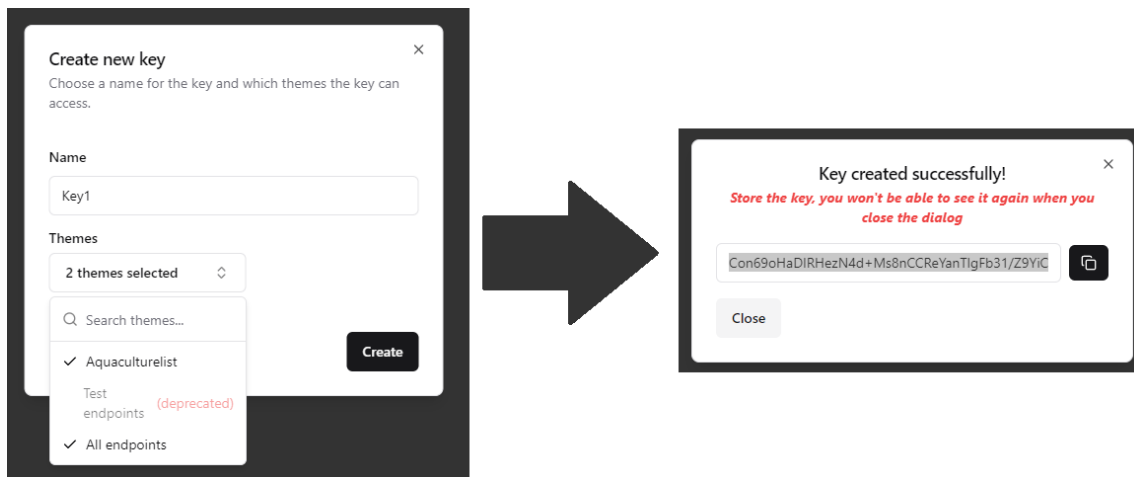


Figure 26: Frontend GUI for creating a key

4.1.4.3 Testing a REST Key

There is a dedicated page for testing the REST key, where once a key is entered into the input, we get a list of the themes this key has access to. Clicking on a theme shows you all the endpoints for that theme, and what schema you can expect to get

as a response. Here you can also test the endpoint and get the actual response shown below.

This page doesn't require a login to be viewed and is available to everyone since its meant to be used by third-parties to test their keys.

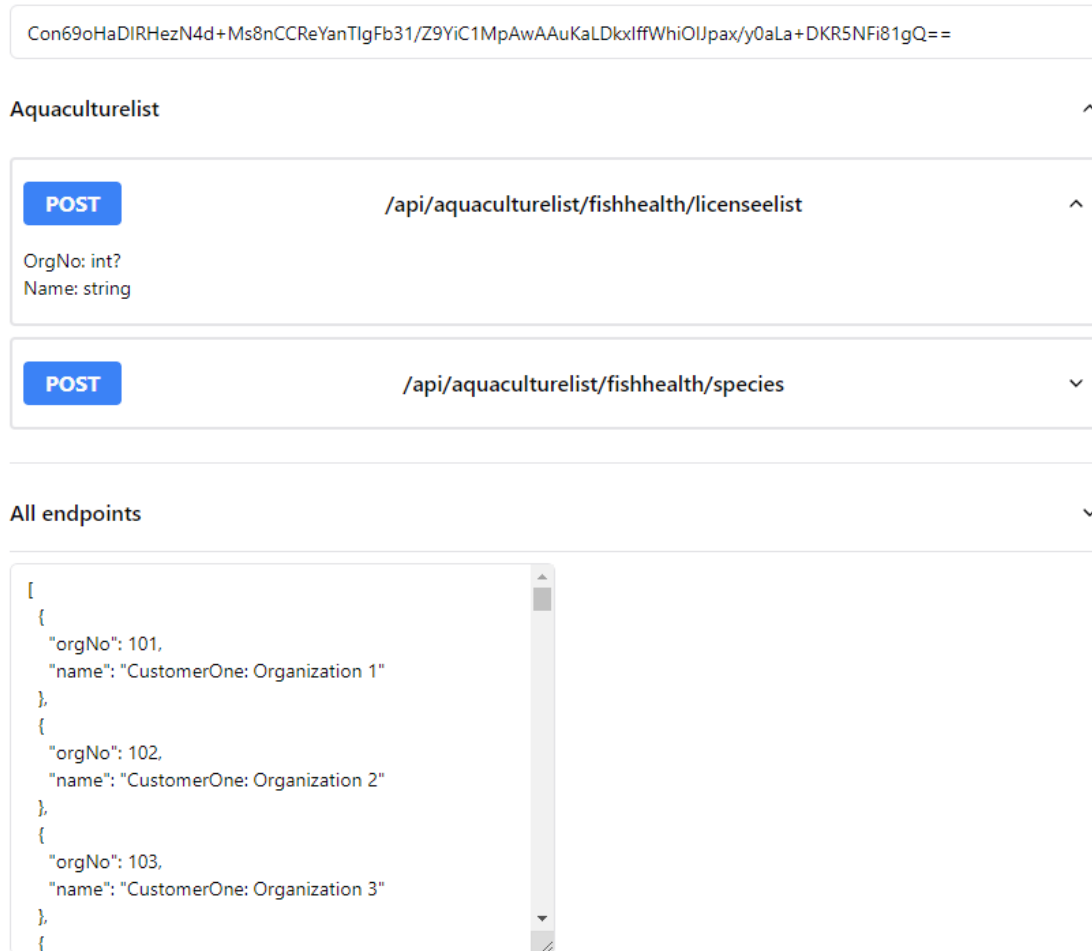


Figure 27: Frontend GUI for testing a REST key where you can enter a key, see the themes for the key, and test the accessible endpoints.

4.1.4.4 Authorization

Authorization for an API endpoint was done using the ASP.NET Core Identity library.

```
67 [HttpPost("{sensorType}/stopAll")]
68 [Authorize(Roles = RoleConstants.AdminRole)]
69 public async Task<IActionResult> StopAllSensors(SensorType sensorType)
70 {
```

Figure 28: How authorization is done using the ASP.NET Core Identity library for a REST API endpoint in the controller, with the accessibility limited to only ADMIN users.

4.1.5 GraphQL Solution

The goal of the GraphQL solution is very similar to the REST API solution, since in practice they are used for similar things: fetching and requesting data. However, all of the communication with the backend is done through GraphQL queries, and mutations.

While the REST API solution gives direct access to a whole endpoint, the GraphQL solution allows the user to be more specific with what data they want to share with third parties. For example, using GraphQL you can choose to only return one field in an object containing multiple fields, for example the *name* field in a Species object. The solution reflects this by giving the user the same amount of control when creating the keys.

To make a GraphQL query for a class table, all of the communication is done on the *localhost:8088/graphql/* url. Directly going to the url also gives you an overview of the Schema Reference, and the Schema Definitions.

The GraphQL query is used for fetching data. A big difference between making a GraphQL Query and a REST API call is that you have to know *beforehand* what kind of information you need, see Figure 30 and Figure 31 for reference. See Figure 29 for an overview of the the queries and Figure 32 for an example of the schema definitions.



Query	Description
Kind of type: Object	The Query type is a special type GraphQL object type, and its
Fields	
availableClassTables: [ClassInfo]!	No description
availableQueries: [[String!]!]!	No description
boatLocationLogs: [BoatLocationLog]!	No description
graphqlApiKeysByUser: [GraphQLApiKeyDto]!	No description
organizations(...): [Organization]!	No description encryptedKey: String
restApiKeysByUser: [RestApiKeyDto]!	No description
species(...): [Species]!	No description encryptedKey: String
waterQualityLogs: [WaterQualityLog]!	No description

Figure 29: Overview of the GraphQL Query Schema. Note: The column with *encryptedKey* are the queries that can take a key as a parameter.

```

Operations Schema Reference Schema Definition
Operations Run ▶ Response
1 Run ▶
2 {
3   organizations {
4     id,
5     orgNo,
6     name,
7     address,
8     postalCode,
9     city
10  }
11 }
12
13 {
14   "data": {
15     "organizations": [
16       {
17         "id": 101,
18         "orgNo": 101,
19         "name": "CustomerOne: Organization 1",
20         "address": "Address 1",
21         "postalCode": "PostalCode 1",
22         "city": "City 1"
23       },
24       {
25         "id": 102,
26         "orgNo": 102,
27         "name": "CustomerOne: Organization 2",
28         "address": "Address 2",
29         "postalCode": "PostalCode 2",
30         "city": "City 2"
31       },
32       {
33         "id": 103,
34         "orgNo": 103,
35         "name": "CustomerOne: Organization 3",
36         "address": "Address 3",
37         "postalCode": "PostalCode 3",
38         "city": "City 3"
39       }
40     ]
41   }
42 }

```

Figure 30: Example query of fetching all of the fields for the Organizations class.

```

Operations Run ▶ Response
1 Run ▶
2 {
3   organizations {
4     id,
5     orgNo
6   }
7 }
8
9 {
10  "data": {
11    "organizations": [
12      {
13        "id": 101,
14        "orgNo": 101
15      },
16      {
17        "id": 102,
18        "orgNo": 102
19      },
20      {
21        "id": 103,
22        "orgNo": 103
23      },
24      {
25        "id": 104,
26        "orgNo": 104
27      },
28      {
29        "id": 105,
30        "orgNo": 105
31      },
32      {
33        "id": 106,
34        "orgNo": 106
35      },
36      {
37        "id": 107,
38        "orgNo": 107
39      }
40    ]
41  }
42 }

```

Figure 31: Example of a partial GraphQL query, where you only fetch the *id* and *orgNo* fields.

As mentioned in Section 3.3.2, FiiZK added a requirement for GraphQL to add a “secret column” for GraphQL, for scenarios where the admin didn’t want to share all fields with a user. As you can see the .Net Entity Framework Core class in Figure 33 and the GraphQL Schema in Figure 32, the *SuperSecretNumber* is ignored by GraphQL and doesn’t show up in the schema, meaning it can’t be retrieved using a GraphQL query.

```

27 type WaterQualityLog {
28   id: Int!
29   offset: Long!
30   timeStamp: DateTime!
31   ph: Float!
32   turbidity: Float!
33   temperature: Float!
34 }
35
36 type Organization {
37   id: Int!
38   orgNo: Int
39   name: String
40   address: String
41   postalCode: String
42   city: String
43 }
44
45 type Species {
46   id: Int!
47   name: String
48 }
49
50 type BoatLocationLog {
51   id: Int!
52   offset: Long!
53   timeStamp: DateTime!
54   latitude: Float!
55   longitude: Float!
56 }

```

Figure 32: Schema definition of the different fields the objects can return.

```

1 using System.ComponentModel.DataAnnotations;
2
3 namespace NetBackend.Models;
4
5 26 references | You, 2 months ago | 2 authors (shv-gitman and others)
6 public class Species
7 {
8   [Key]
9   16 references
10  public int Id { get; set; }
11   19 references
12  public string? Name { get; set; }
13
14   [GraphQLIgnore]
15   4 references
16  public int? SuperSecretNumber { get; set; }
17 }

```

Figure 33: The .Net Entity Framework Core class of Species.

4.1.5.1 GraphQL mutations

The mutations are GraphQLs equivalent to the POST, PUT, PATCH or DELETE methods for REST API. We made GraphQL mutations for creating, deleting and toggling a GraphQL Key, as seen in Figure 34. We only made the bare minimum mutations necessary for the GraphQL Key overview page to function similarly to the REST API key overview page. Meaning there would be no difference in the user experience interacting with the GUI between the two solutions.

ApiKeyMutation	
Kind of type: Object	
The Mutation type is a special type that is used to modify ask for nested fields. It can also contain multiple fields. Ho	
Fields	
<code>createGraphQLAccessKey(...): AccessKeyDto!</code>	No description <code>keyName: String!</code> <code>permissions: [GraphQLAccessKeyPermissionDtoInput!]</code>
<code>deleteGraphQLApiKey(...): ResponseDto!</code>	No description <code>id: UUID!</code>
<code>deleteGraphQLApiKeyByEncryptedKey(...): ResponseDto!</code>	No description <code>encryptedKey: String!</code>
<code>toggleApiKey(...): ResponseDto!</code>	No description <code>toggleApiKeyStatusDto: ToggleApiKeyStatusDtoInput!</code>

Figure 34: Overview of the GraphQL mutations. On the right are the parameters.

4.1.5.2 Creating A GraphQL Key

A GraphQL access key is created using the `createGraphQLAccessKey` mutation with a `keyName` and a list of `permissions`. The permissions hold onto the class table (`queryName`), and the specific fields of that class table the key can access. Similarly to the other solutions, the response body is the final access key (`encryptedKey`), as seen in Figure 35 for the backend, and Figure 36 for the frontend GUI for creating a key.

The frontend GUI for the GraphQL key overview is similar to all the other key overviews, except for under `Permissions`, there is a list of all class tables and fields that key can access, as seen in Figure 37.

```

Operations
1  mutation {
2    createGraphQLAccessKey(
3      keyName: "ExampleKey"
4      permissions: [
5        {
6          queryName: "species"
7          allowedFields: ["id", "name"]
8        },
9        {
10         queryName: "organizations"
11         allowedFields: ["orgNo"]
12       }
13     ]
14     {
15       encryptedKey
16     }
17   }
18 }
Response
1  {
2    "data": {
3      "createGraphQLAccessKey": {
4        "encryptedKey": "y7DwpKSiqQMb56PKkXNthTr81z16NLFctq30v3FXFTvg91LCWoCChzL/q17D0EepMALqw6zekTQzTbqEqTX+Q="
5      }
6    }
7  }

```

Figure 35: GraphQL create key mutation.

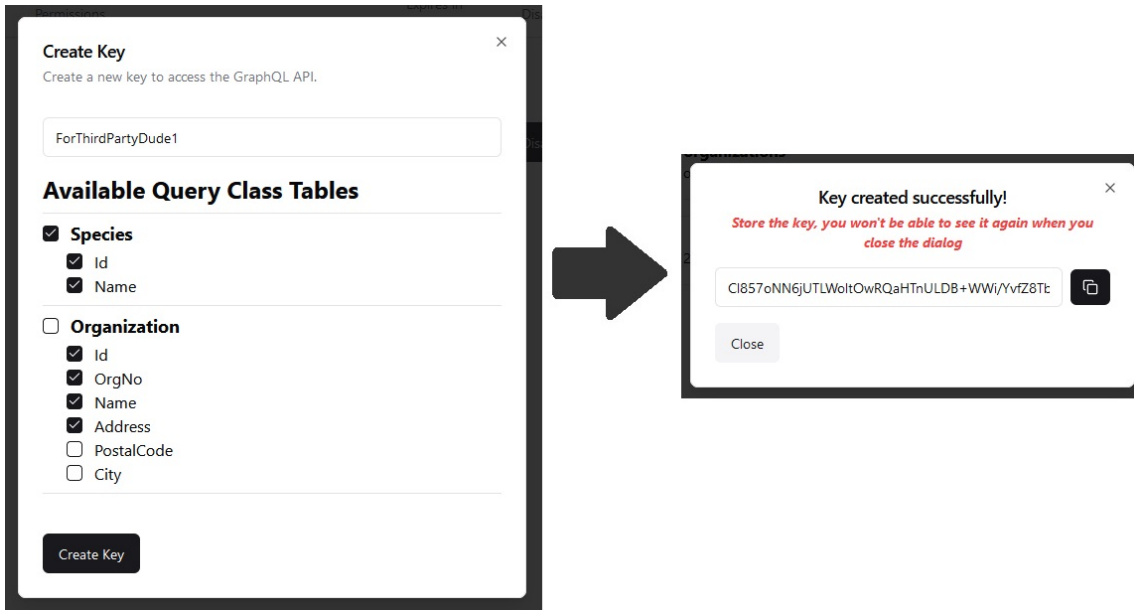


Figure 36: Frontend GUI equivalent for the GraphQL create key mutation.

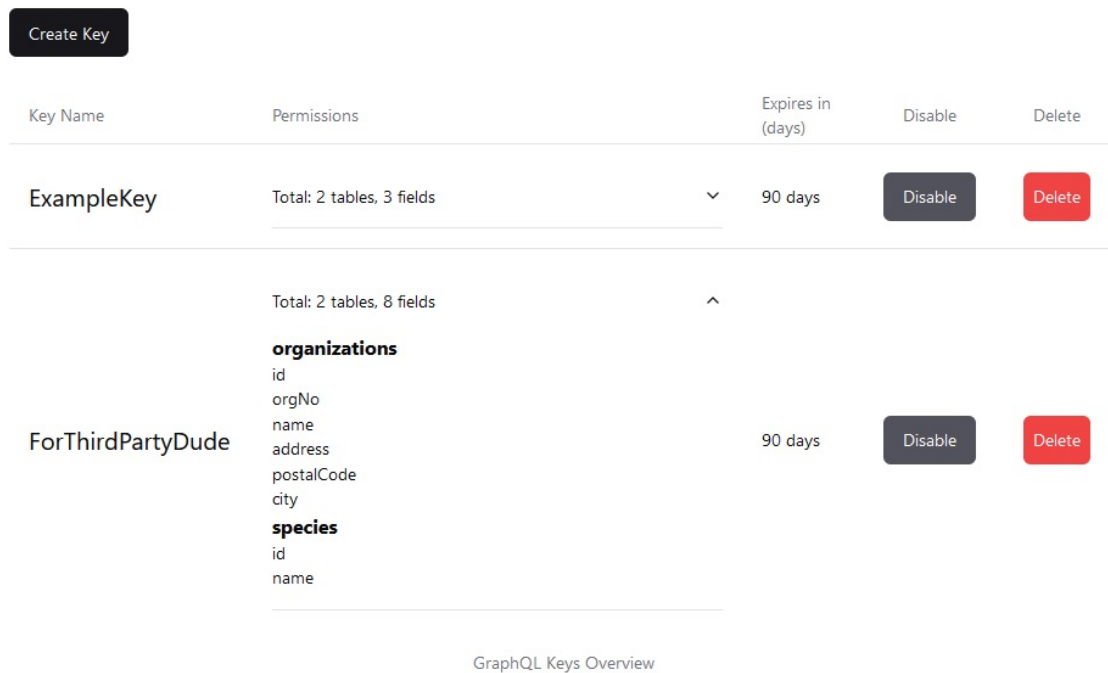


Figure 37: Frontend GUI of the GraphQL key overview.

4.1.5.3 Testing a GraphQL Key

There is a dedicated page for testing the GraphQL key, where once a key is entered into the input, everything that key has access to, is showed. The user can also make a query of the selected fields it want to fetch from the database, as seen in Figure 38. This page doesn't require a login to be viewed and is available to everyone since its meant to be used by third-parties to test their keys.

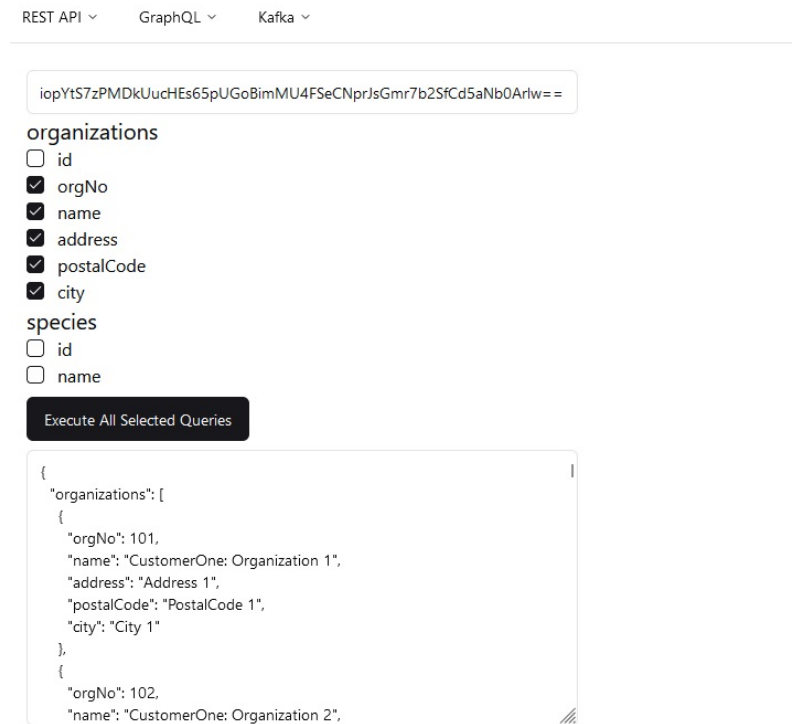


Figure 38: Frontend GUI for testing a GraphQL key where you can enter a key, see the available class tables and fields that key can make a query of, as well as make a query of your selected fields.

4.1.5.4 Authorization

The ASP.NET Core Identity library which is used to create the users and authorize the REST API endpoints, seen earlier in Figure 28, didn't work with the Hot Chocolate library used for GraphQL. A workaround was implemented for GraphQL queries and mutations that should only be accessed by users, where the `HttpContext` is used to check if it's made by an authorized user. The workaround is seen in Figure 39.

Note:

The HotChocolate library comes with its own `[Authorize]` attribute, but we realised the workaround that still use the Identity library would be easier and less complicated to implement, than implementing HotChocolate's method of authorization.

```
0 references
public async Task<ResponseDto?> DeleteGraphQLApiKey(
    [Service] IGraphQLKeyService graphQLKeyService,
    [Service] IUserService userService,
    Guid id)
{
    // NOTE: Workaround to replace [Authorize]
    var (user, error) = await userService.GetUserByHttpContextAsync(_httpContextAccessor.HttpContext ?? throw
        (_httpContextAccessor), "HttpContextAccessor's HttpContext is null. User unauthorized or session expired.");
    if (user == null) return null;
}
```

Figure 39: GraphQL Workaround to authenticate a user.

4.1.6 Kafka Solution

Kafka differs from using REST API and GraphQL for data transfer, because it is used for event-based data streaming. Where a message can be *produced* and sent to a topic in a Kafka Cluster, and then a consumer can *consume* that message by subscribing to the topic.

To satisfy the requirement of allowing users to share a subset of their data with third parties, the Kafka keys can contain the different topics accessible by the user.

Kafka works with Zookeeper to form a Kafka Cluster, as seen in the architecture diagram back in Figure 10. What is relevant to this solution is that Zookeeper manages the brokers and notifies when topics are created or deleted. Kafka on the other hand is what manages the data-stream coming into the Kafka Cluster [41].

4.1.6.1 Creating A Kafka Key

For a user to create a key, they would have to specify the key name, and a list of topics that key would have access to, see Figure 40. The topics would each have the id of the user that created the key added, meaning you created a key that had access to the "water-quality-updates" topic, it would in reality have access to the "water-quality-updates-*{userId}*" topic. The *userId* is a Globally Unique Identifier (GUID). Because of this, two users won't be able to create a key that has access to another users topics.

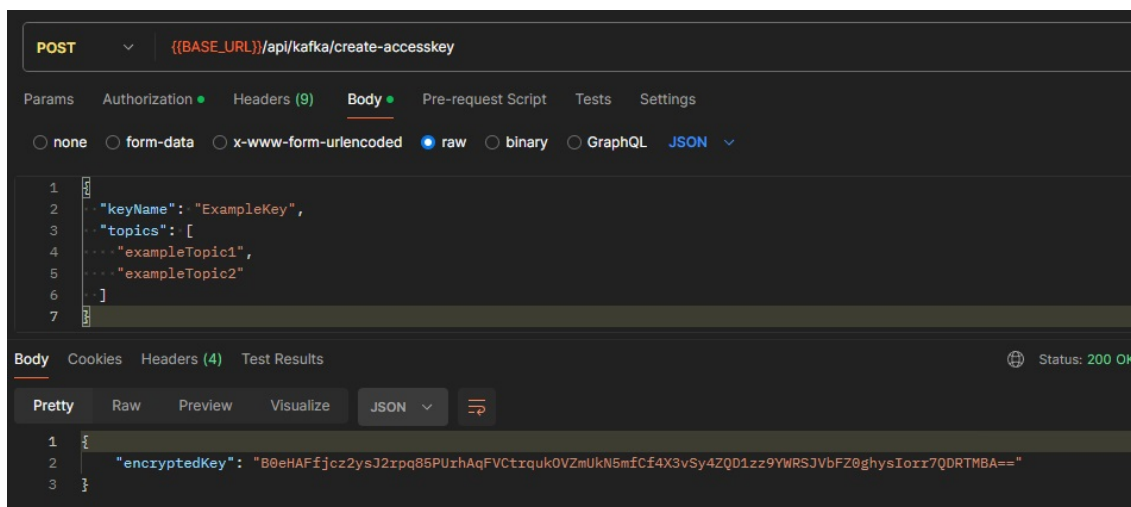


Figure 40: Example of body and response when creating a Kafka key using the REST API endpoint.

A logged in user can also create a key in the frontend website as seen in Figure 41. And similarly to other solutions, can get an overview of all the created keys, with a list of the topics they have access to. From the Keys overview page, a user can also disable and delete the keys, as well as see how many days are left until the keys expire, see Figure 42 for reference.

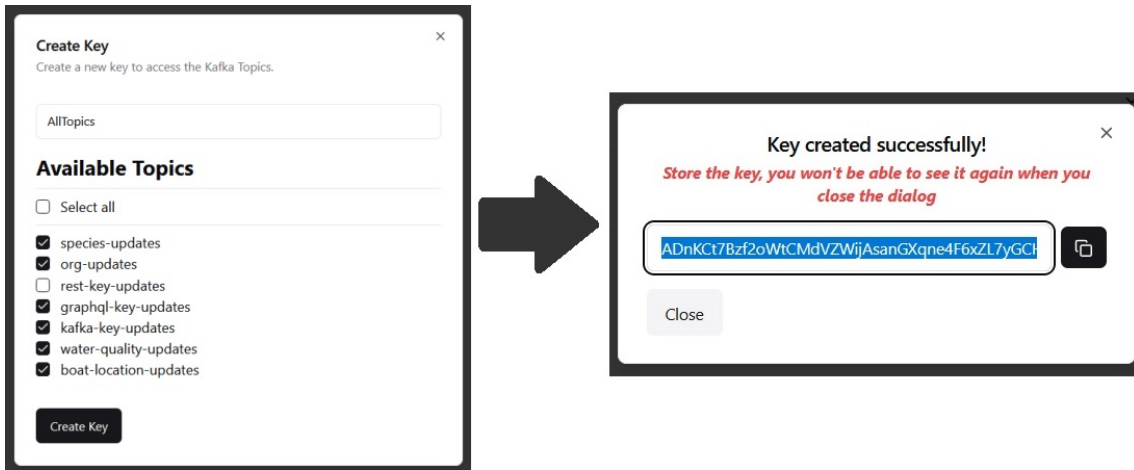


Figure 41: Example of GUI dialog and response when creating a Kafka key for a user on the frontend Website.

REST API ▾ GraphQL ▾ Kafka ▾ Admin Page Logout

[Create Key](#)

Key Name	Topics	Expires in (days)	Disable	Delete
AllTopics	6 topics	90 days	Disable	Delete
BoatLocationUpdatesKey	1 topics boat-location-updates	90 days	Disable	Delete
ActuallyAllTopics	7 topics species-updates org-updates rest-key-updates graphql-key-updates kafka-key-updates water-quality-updates boat-location-updates	90 days	Disable	Delete

Kafka Keys Overview

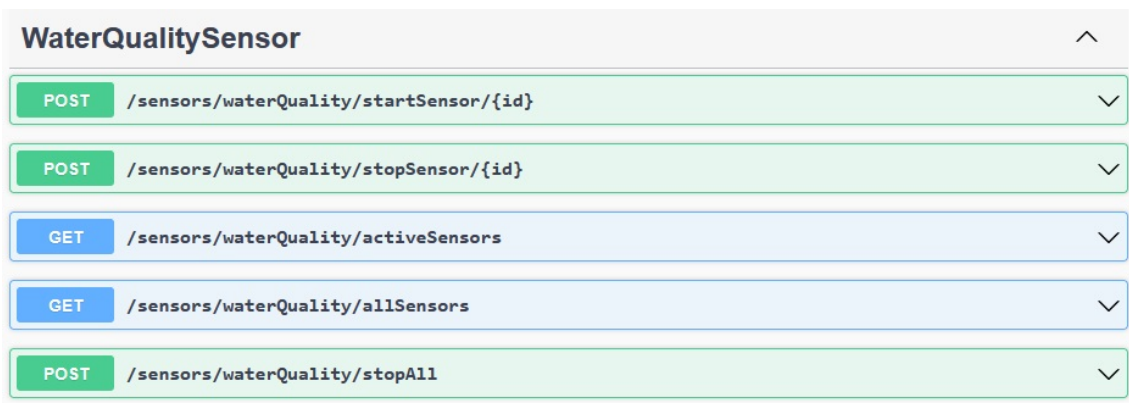
Figure 42: Frontend Kafka Keys Overview page.

4.1.6.2 Data stream proof of concept example

To give an example where an event based messaging system like Kafka would be used, we made a "mock sensor" that would produce a message to a Kafka topic related to a sensor.

To do this, we made a Mock Sensor .Net Application, which is a separate .Net 8

application. The Mock Sensor has two pre-made sensors, that can be created, started and stopped using REST API calls, seen in Figure 43.



The screenshot shows a list of REST API endpoints for the **WaterQualitySensor**. The endpoints are as follows:

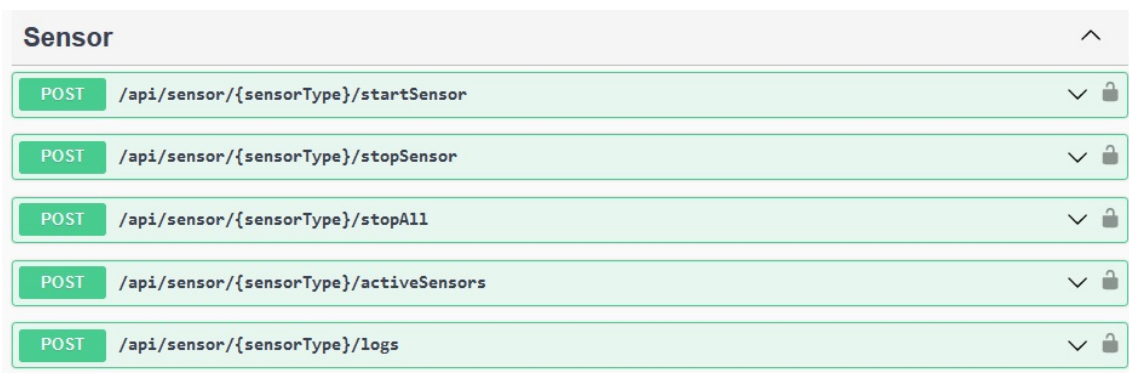
Method	Endpoint
POST	/sensors/waterQuality/startSensor/{id}
POST	/sensors/waterQuality/stopSensor/{id}
GET	/sensors/waterQuality/activeSensors
GET	/sensors/waterQuality/allSensors
POST	/sensors/waterQuality/stopAll

Figure 43: Example of REST API endpoints for a sensor in the Mock Sensor .Net Application.

The Mock sensor produces messages to the Kafka Cluster, that acts as a middle-man between the Main .Net Application and the Mock Sensor.

The mock sensor comes with two pre-made sensors, a water quality sensor that produces messages to the water-quality-updates-`{id}` topic, and a boat location sensor that produces to the boat-location-updates-`{id}` topic.

All of the communication with the frontend goes through the main .Net application, so to start the mock sensor, the main .Net application has corresponding REST API endpoints to communicate with the mock sensor, as seen in Figure 44.



The screenshot shows a list of REST API endpoints for the **Sensor**. The endpoints are as follows:

Method	Endpoint
POST	/api/sensor/{sensorType}/startSensor
POST	/api/sensor/{sensorType}/stopSensor
POST	/api/sensor/{sensorType}/stopAll
POST	/api/sensor/{sensorType}/activeSensors
POST	/api/sensor/{sensorType}/logs

Figure 44: The Main .Net Applications corresponding REST API endpoints to communicate with the Mock Sensor.

Note regarding Figure 43 and Figure 44

The Main .Net Application doesn't need the id parameter, since it is able to extract it from the *bearer access token* of a user, or from the Kafka key, if one is provided in the body.

When making the REST API call to start a sensor, in addition to selecting the sensor type to start, you can specify a *SessionID* which is used to create a private WebSocket connection with the frontend. You can also choose to enable a *SendHistoricalData* boolean parameter which, will notify the consumer in the main .Net application to

send all of the messages so far produced to the topic. See Figure 45 and Figure 46. See Figure 47 for example logs when *SendHistoricalData* is set to true.

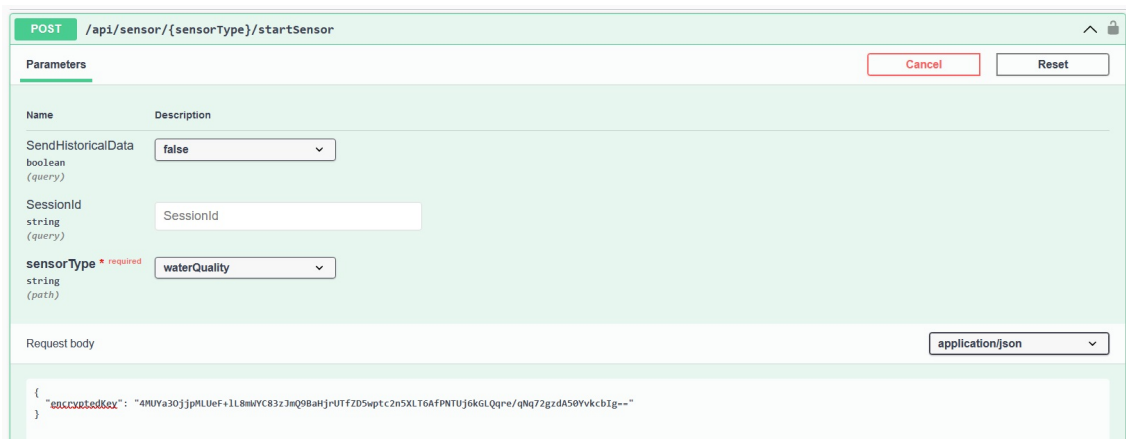


Figure 45: REST API endpoint and its parameters to start a Mock Sensor.

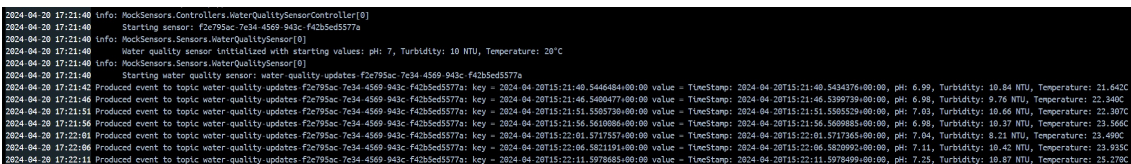


Figure 46: Logs from the Mock Sensor Docker Container producing messages to the topic with the *userId* as the sensor identifier.

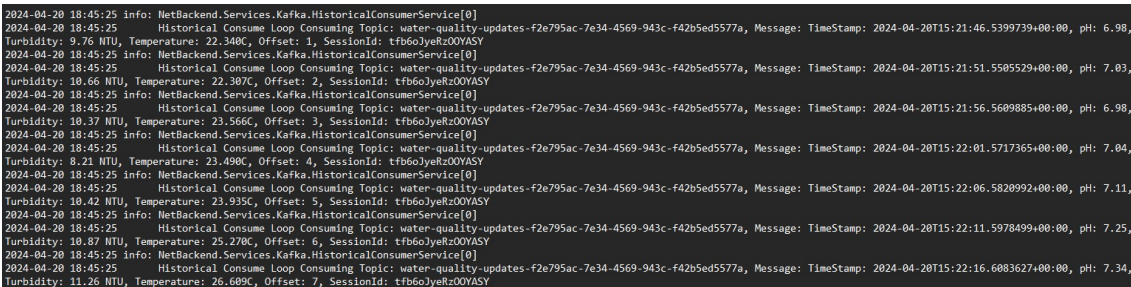


Figure 47: Logs from the main .Net application after a sensor was started and all the messages are sent at once because the *SendHistoricalData* parameter was set to true.

4.1.6.3 Long Term Storage

Every time the backend consumes a message from the *water-quality-updates* topic or the *boat-quality-updates* topic, it stores the message to the users PostgreSQL database.

To efficiently store the messages and not interrupt the consume-loop, the messages are offloaded into a separate thread and then processed in a batch every 10 seconds. Using batch processing improves performance by fetching and disposing of the dependency-injected database context less frequently. Otherwise it would have to fetch the database-context and the different dependency injections every message.

After a message is sent for processing, or for any other topic than *water-quality-updates* and *boat-quality-updates*, the consumer just sends the message through a

WebSocket. If the *SendHistoricalData* boolean is false, it sends it to all WebSocket sessions (identified through the *sessionId* parameter) that are currently listening to that topic, but if *SendHistoricalData* is true, the messages are only sent to that specific session.

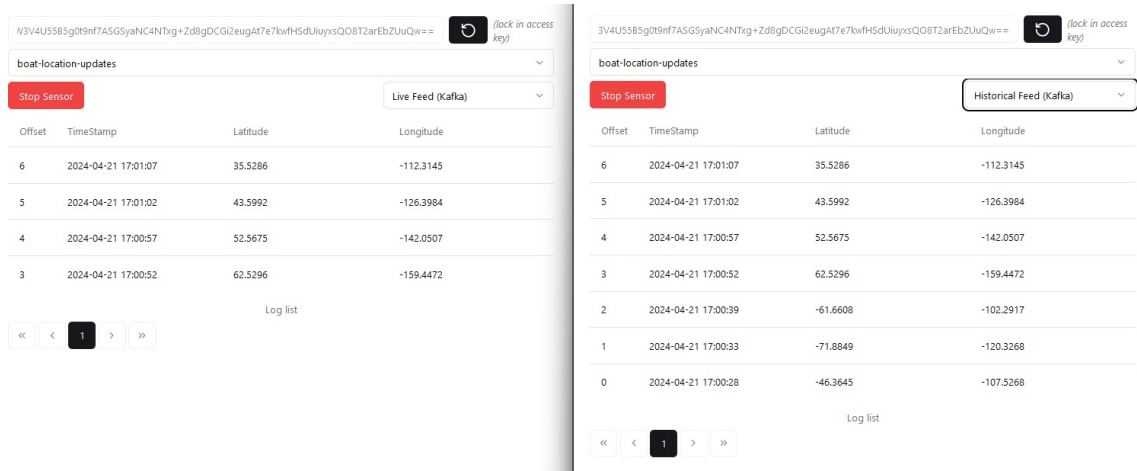


Figure 48: Two separate frontend sessions connected to the backend through a WebSocket, left is watching the live feed, and right requested all of the messages (up until that point).

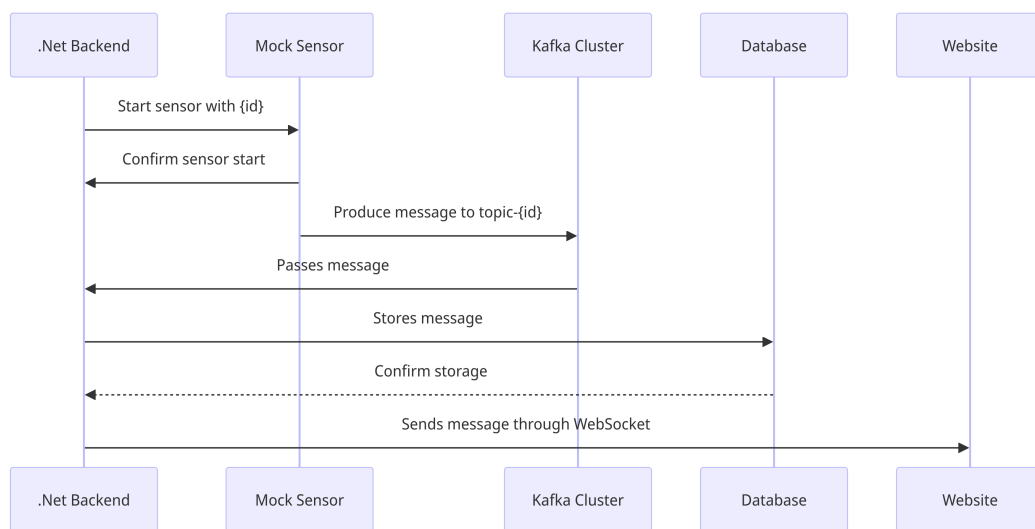


Figure 49: Sequence Diagram of the interaction between the .Net backend, mock sensor, Kafka Cluster, Database, and frontend website.

In Figure 50 you can see how the frontend also can fetch the logs stored in the PostgreSQL database using the */api/sensor/sensorType/logs* endpoint seen in Figure 44.

3V4U55B5g0t9nf7ASGSyaNC4NTxg+Zd8gDCGi2eugAt7e7kwfHSdUiuyxsQO8T2arEbZUuQw== (lock in access key)

boat-location-updates

Stop Sensor Newest Values (REST)

ID	Offset	TimeStamp	Latitude	Longitude
10	9	2024-04-21 17:01:22	15.8468	-77.9642
8	8	2024-04-21 17:01:17	21.7304	-88.232
9	7	2024-04-21 17:01:12	28.2633	-99.6393
6	6	2024-04-21 17:01:07	35.5286	-112.3145
7	5	2024-04-21 17:01:02	43.5992	-126.3984
4	4	2024-04-21 17:00:57	52.5675	-142.0507
5	3	2024-04-21 17:00:52	62.5296	-159.4472
1	2	2024-04-21 17:00:39	-61.6608	-102.2917
2	1	2024-04-21 17:00:33	-71.8849	-120.3268
3	0	2024-04-21 17:00:28	-46.3645	-107.5268

Log list

<< < 1 > >>

Figure 50: Frontend table displaying the boat-location-updates logs, sorted by most recent log first.

4.1.7 Containerization

We containerized the application using Docker to ensure multi-platform compatibility. The project employs multiple containers for both the backend and frontend, utilizing Docker Compose to construct the multi-container environment. Docker Desktop is used for management and to get an overview of the environment. A complete view of the project's setup is shown in Figure 51.

For containers like PostgreSQL databases, Kafka, Zookeeper and Nginx, existing containers are pulled from the Docker Hub, see Figure 52 for an example of using an existing container, and how a PostgreSQL container is made.

To persist data between container builds, which is necessary for the PostgreSQL databases and Kafka Cluster, the data is stored as a volume locally. If this isn't done, the data is lost every time the container is destroyed.

Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
netbackend		Running (9/9)	0.93%		57 minutes ago	
netbackend	docker.io/library/netbackend	Running	0.49%	8088:80	2 hours ago	
mock-sensors	docker.io/library/mock-sensors	Running	0%	8089:80	2 hours ago	
kafka	confluentinc/cp-kafka:7.0.14	Running	0.37%	9092:9092	2 hours ago	
zookeeper	confluentinc/sp-zookeeper:7.0.14	Running	0.07%	2181:2181	2 hours ago	
nginx-app	nginx:alpine	Running	0%	80:80	2 hours ago	
postgresCustomerTwo	postgres:latest	Running	0%	5435:5432	2 hours ago	
postgresCustomerOne	postgres:latest	Running	0%	5434:5432	2 hours ago	
postgresTest	postgres:latest	Running	0%	5433:5432	2 hours ago	
vuefrontend	docker.io/library/vuefrontend	Running	0%	8080:80	57 minutes ago	

Figure 51: The Docker Environment.

Here is a clearer overview over the different containers seen in Figure 51 and their purpose:

Backend:

- **netbackend** container - Main .Net application
- **mock-sensors** container - Mock Sensors .Net Application
- **postgresTest** container - Main PostgreSQL database, user info, keys
- **postgresCustomerOne** container - User tenant database 1, private data
- **postgresCustomerTwo** container - User tenant database 2, private data
- **kafka** container - Manages message queues for real-time data handling and processing.
- **zookeeper** container - Coordinates with Kafka to manage cluster state and configurations.

Frontend:

- **vuefrontend** container - Vite + Vue frontend application
- **nginx-app** container - Serves and routes traffic to the Vue frontend.

The *nginx-app* container acts as the *Webserver* seen in Figure 10. It serves the static pages generated by the *vuefrontend* container, which builds the Vue application into static pages.

```

82     postgresCustomerOne:
83         image: postgres:latest
84         container_name: postgresCustomerOne
85         environment:
86             - POSTGRES_USER=postgres
87             - POSTGRES_PASSWORD=password
88             - POSTGRES_DB=postgresCustomerOneDB
89         ports:
90             - "5434:5432"
91         restart: always
92         volumes:
93             - ./backend/NetBackend/.containers/postgresCustomerOneDb:/var/lib/postgresql/data
94         networks:
95             - dev

```

Figure 52: Docker compose using an existing image pulled from docker hub. The :latest annotation means the most recent image is pulled, and every time the project is built it checks for a new version and automatically updates.

For applications developed locally, like the different .Net applications and the Vue frontend, a Dockerfile is used to build the container and add it to the docker-compose file, seen in Figure 53. The Dockerfile is responsible for copying and building the project. For .Net applications with unit tests, the Dockerfile is also responsible for running the tests, as seen in Figure 54, where the container will fail building if the tests fail.

The containers cannot communicate internally between themselves using the normal *localhost* route, which is only used to communicate with them externally. The containers use an internal network which in this case is named "dev", as seen in Figure 52 and Figure 53. If they are on the "dev" network, they can connect to each other using *container-name:port*, as seen in Figure 55.

```

46     netbackend:
47         image: docker.io/library/netbackend
48         depends_on:
49             - postgresTest
50             - postgresCustomerOne
51             - postgresCustomerTwo
52             - kafka
53         container_name: netbackend
54         ports:
55             - "8088:80"
56         build:
57             context: ./backend/
58             dockerfile: Dockerfile
59         environment:
60             - ConnectionStrings__DefaultConnection=User ID=postgres;Password=password;
61               Server=postgresTest;Database=postgresTestDB;Pooling=true;Port=5432;
62             - ConnectionStrings__CustomerOneConnection=User ID=postgres;Password=password;
63               Server=postgresCustomerOne;Database=postgresCustomerOneDB;Pooling=true;Port=5432;
64             - ConnectionStrings__CustomerTwoConnection=User ID=postgres;Password=password;
65               Server=postgresCustomerTwo;Database=postgresCustomerTwoDB;Pooling=true;Port=5432;
66             - ASPNETCORE_URLS=http://+:80
67         networks:
68             - dev

```

Figure 53: Docker compose setup for the main .Net application.

```

1 # Base image for running the application
2 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
3 WORKDIR /app
4 EXPOSE 80
5 EXPOSE 443
6
7 # Build environment and build the application
8 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
9 WORKDIR /src
10 # Copy the main project and its dependencies
11 COPY ["NetBackend/NetBackend.csproj", "NetBackend/"]
12 COPY ["NetBackend.Tests/NetBackend.Tests.csproj", "NetBackend.Tests/"]
13 # Restore packages for both projects
14 RUN dotnet restore "NetBackend/NetBackend.csproj"
15 RUN dotnet restore "NetBackend.Tests/NetBackend.Tests.csproj"
16 # Copy the rest of the source code
17 COPY . .
18 # Build the main project
19 RUN dotnet build "NetBackend/NetBackend.csproj" -c Release -o /app/build
20 # Test the main project
21 RUN dotnet test "NetBackend.Tests/NetBackend.Tests.csproj" --verbosity normal
22
23 # Publish the application
24 FROM build AS publish
25 RUN dotnet publish "NetBackend/NetBackend.csproj" -c Release -o /app/publish
26
27 # Build the final image
28 FROM base AS final
29 WORKDIR /app
30 COPY --from=publish /app/publish .
31 ENTRYPOINT ["dotnet", "NetBackend.dll"]

```

Figure 54: Dockerfile for the main .Net application and the test project for the unit-tests.

```

3 public static class UrlConstants
4 {
5     1 reference
6     public const string FrontEndURL = "http://localhost:8080";
7
8     // For the docker network
9     1 reference
10    public const string MockSensorURL = "http://mock-sensors:80";
11 }

```

Figure 55: URL constants for the main .Net application. `mock-sensors:80` is used for internal docker-communication with the Mock Sensor .Net application.

Note that the url for the frontend isn't `vuefrontend:80`, like the container. This is because the requests from the frontend website are still made externally from a browser at `localhost:8080`.

4.1.8 Testing

We have a postman test collection that tests the most crucial features of the application.

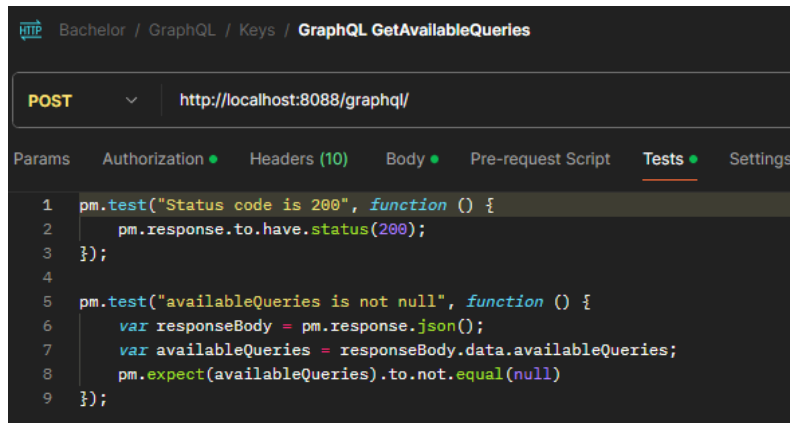
The collection is idempotent (running the collection doesn't change any resources and yields the same result every single time). This is done by deleting any resource that is created and only performing update operations on resources that will be deleted at the end of the collection.

To make the collection dynamic and repeatable we use environment variables to store values we get from the requests, that we later use.

▶	POST	invalid login test123@mail.com	1	0
▶	POST	register	1	0
▶	POST	login with registered registered@mail.com	1	0
▶	POST	login test@mail.com	1	0
▶	GET	invalid user level token to Get all users	1	0
▶	POST	login admin@mail.com	1	0
▶	GET	valid admin level token to Get all users	1	0
▶	GET	User Info	1	0
▶	GET	get-default-endpoints	1	0
▶	POST	create-theme	1	0
▶	GET	get-themes-by-user	1	0
▶	PUT	update-theme	1	0
▶	POST	create-accesskey	1	0
▶	GET	get-aplkeys-by-user	1	0
▶	POST	accesskey-themes	1	0
▶	DELETE	delete-accesskey-by-encryptedkey	1	0
▶	DELETE	delete-theme	1	0
▶	POST	GraphQL GetAvailableQueries	2	0
▶	POST	GraphQL GetAvailableClassTables	2	0
▶	POST	create-accesskey GraphQL	2	0
▶	POST	Specles Key GraphQL	2	0
▶	POST	Specles GraphQL Bearer Token	2	0
▶	POST	Licenseelist GraphQL Key	2	0
▶	POST	Licenseelist GraphQL Bearer Token	2	0
▶	POST	Licenseelist GraphQL no token or key	2	0
▶	POST	Both GraphQL Bearer	3	0
▶	POST	delete-accesskey-by-encryptedkey Grap...	2	0
▶	GET	get-available-topics	1	0
▶	POST	create-accesskey	1	0
▶	GET	get-keys-by-user	1	0
▶	POST	Key topics	1	0
▶	PATCH	subscribe-to-topics	1	0
▶	DELETE	delete-accesskey-by-encryptedkey	1	0
▶	GET	Get all users	1	0
▶	GET	Get roles	1	0
▶	POST	change-role	1	0
▶	GET	get-database-names	1	0
▶	POST	update-database-name	1	0
▶	DELETE	delete-user-by-email registered@mail.com	1	0

Figure 56: Postman collection run results

For most of the requests the testing consists of checking that the status code is 200, but for GraphQL we also have to check that the desired part of the response body isn't *null*.

A screenshot of the Postman interface showing a POST request to http://localhost:8088/graphql/. The 'Tests' tab is active, displaying two test scripts. The first test checks for a 200 status code. The second test checks that the availableQueries field in the response body is not null.

```
1 pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test("availableQueries is not null", function () {
6   var responseBody = pm.response.json();
7   var availableQueries = responseBody.data.availableQueries;
8   pm.expect(availableQueries).to.not.equal(null)
9 });
```

Figure 57: GraphQL AvailableQueries postman tests.

4.2 Administrative Results

As mentioned in the Section 3.1 section, we implemented a SCRUM-like approach with week long sprints, group roles and a milestone plan for the project.

4.2.1 Group Structure

Name	Roles
Simen H. Veum	Team leader, meeting planner
Adrian R. Dahl	Delivery manager
Ole T. Aanderaa	Document manager, quality assurance

Table 2: Group Roles and Responsibilities.

4.2.2 Milestone Result

Date	Milestone
26.01.2024	Set up Github
26.01.2024 05.02.2024	Make Wireframes for the frontend
16.02.2024 17.03.2024	Complete REST API solution
08.03.2024 18.03.2024	Complete GraphQL solution
29.03.2024 22.04.2024	<i>Undecided 3rd solution</i> Complete Kafka solution
23.04.2024 29.04.2024	Finish Product Development
07.05.2024	Finish first-draft of report
21.05.2024	Finish report

Table 3: Project Milestone Result.

5 Discussion

5.1 Comparing the solutions

It is difficult to compare the three solutions, particularly the Kafka solution with the REST and GraphQL due to Kafka serving a different purpose than the other two.

They all provide ways of data transfer, however Kafka is made for real-time data streaming whereas REST and GraphQL are protocols used for API communication between the server and its clients.

Due to these differences, we never focused on comparing the performance between the three, and focused more on the developer experience, learning curve and ease of implementation with the frontend and backend systems when comparing the solutions.

Even our GraphQL and REST solutions had some differences that, although you could use both protocols to complete more-or-less the same tasks, our solutions had some limitations that caused them to only be usable in certain situations, and locked them into different use-cases.

5.1.1 Developer Experience

As mentioned in Section 3.3, we already had an idea of the learning-curve and how difficult the different solutions would be to implement, and due to this we developed the REST solution first, followed by the GraphQL and Kafka solutions.

5.1.1.1 REST

Although we were new to the .Net framework, we assumed that implementing REST API to a .Net application wouldn't be very challenging due to our previous experience, which is why we chose it first. This was so we would quickly have a good baseline and proof-of-concept we could work from, and replicate when creating the other solutions.

Regarding authorization and user management, the REST API controllers had a seamless integration with the *AspNetCore Identity* library, meaning we didn't have to spend a lot of time creating our own methods of authentication and authorization.

For the frontend, the developer experience was also quick and seamless due to the straightforward nature of REST API.

5.1.1.2 GraphQL

The GraphQL development experience was more challenging, and the learning curve was steeper compared to REST API, with a few unfamiliar concepts like how the GraphQL queries and mutations were made, both in the frontend and backend.

As mentioned in paragraph 4.1.5.4, The *AspNetCore Identity* library wasn't directly compatible with the *HotChocolate* library we used for GraphQL when it came to authorization. Due to this we had to make a workaround that added a lot more code compared to the REST API authorization, as seen in Figure 39, compared to the simple [Authorize] attribute for the REST controller in Figure 28.

5.1.1.3 Kafka

Kafka was a huge step up from both REST and GraphQL, with a much steeper learning curve. Kafka had a bunch of new concepts like partitions, topics, brokers, consumers and producers that had to be understood. Even after completing the project, our expertise regarding Kafka would probably still be considered surface-level.

Choosing Kafka was, as mentioned a few times, excessive for our use-case, since its primarily used for scenarios where high throughput and reliable message delivery is important.

The development was slowed down due to not fully understanding how consumers and producers worked with .Net, and also a general lack of knowledge regarding the different new concepts. Because of this, a lot of the development consisted of a *“throwing stuff at the wall and see what sticks”*-type approach, with a varying degree of success. In despite of all this, we still managed to create a solution with the functionality that we wanted.

REST API was used to handle all the communication with the Kafka consumer and producer regarding subscribing to- and creating new topics, as well as communicating with the mock sensor .Net application. REST was used due to our familiarization and confidence using the protocol compared to GraphQL.

The .Net backend also processes and stores messages consumed from the mock sensor topics into the users PostgreSQL database. We encountered a problem where messages were sometimes stored out of order, as seen in Figure 50. In the figure you can see the ID is out of order. Luckily, Kafka sends the messages with an offset value, so the order isn't completely lost, but if you wanted to fetch and display the data in the correct order, you would have to do an additional sorting-process that could become computationally expensive.

If we were to start over we could have looked into using a KsqlDB database, rather than using a PostgreSQL database for handling and storing the kafka messages. KsqlDB is built to handle data streaming messages, and would eliminate the problem of storing messages in the wrong order, while still supporting SQL queries.

5.1.2 Differences in solutions

As mentioned earlier, even though REST API and GraphQL can be used to solve the same problems, especially the GraphQL solution lacked certain features to be fully comparable with the REST API solution.

GraphQL shines in regards to control, giving you the ability to choose specifically what data you want to retrieve, and our implementation still gives you that control when fetching from tables in the database. However, a GraphQL key cannot perform mutations or any form of data manipulation, something the REST API key can do because it simply gives access to the endpoint, and then lets it do its thing.

The Kafka solution is similar in the sense that the keys are created the same way as REST and GraphQL keys are. However, the Kafka keys give you access to topics and its corresponding data stream, rather than access to the data in the PostgreSQL databases.

5.1.3 Best Use Case

Although used to solve very similar problems, the GraphQL and REST has, according to our experience from the project, certain use-cases where one shines over the other.

As mentioned earlier, Kafka was, to put it mildly, overkill for our use case, so we were never able to fully see its true potential and capabilities. Kafka is meant to handle thousands of events a *second*, and dealing with systems far more complex than ours [42]. Because of this the *best use cases* will only be based on what we were able to do with the different technologies, rather than list everything they are good for.

5.1.3.1 REST API

- Easy to understand with a low learning curve.
- Operates well under the HTTP standard, making them a good fit for scenarios requiring CRUD operations.

5.1.3.2 GraphQL

- Scenarios where control over what data to fetch is important, reducing the chance of over- and under-fetching.
- Scenarios where the option to make multiple queries or mutations at the same time is beneficial.

5.1.3.3 Kafka

- Scenarios using inter-service communication in systems that employ microservice architecture, like between the sensor started by the Mock Sensor .Net Application that produces to a topic, which is consumed by the Main .Net Application.
- Scenarios that requires live updates or event-based data transfer, like our sensor proof-of-concept or a logging system.

5.2 Administrative Discussion

5.2.0.1 SCRUM

Our implementation of a SCRUM-like approach was largely successful. We never implemented standup meetings into our schedule, something that is common doing every workday when using SCRUM. This was primarily due to other commitments and the fact we had a different course in parallel with the Bachelor project for the first few weeks of the semester. By the time the other course ended, and our main focus was the Bachelors project, we saw no reason to suddenly add the standup meetings. This was because we felt our approach of a weekly group meeting, and two bi-weekly meetings with the supervisor and FiiZK, was working fine and we never felt we were falling behind in our development process.

There was also a brief discussion, during the start of the project, about having the sprints be two-weeks long, to align with the supervisor and FiiZK meetings, but we felt sprints that long would make us procrastinate, so we went with week-long sprints instead.

5.2.0.2 Milestones

As seen in Table 3, none of the "solution"-milestones were completed on time. However, they all had a proof-of-concept done within the milestone date, and the extra days (or month for the REST API solution) was just fine-tuning, cleaning code and making the frontend look pretty.

Different parts of the project was developed in parallel, where someone would start on the next solution once the proof-of-concept was done on the former, hence the close completion dates for REST API and GraphQL solutions.

5.3 Unit testing

We have a .NET project for testing, but no unit tests were implemented due to time constraints. The testing project was made late in the development process, and we already had a sizeable postman collection that tested most of the REST API endpoints and GraphQL queries and mutations. Because of this we saw a greater benefit from refining our postman collection, over implementing unit tests in the testing project.

6 Conclusion

This chapter contains the conclusions regarding our initial problem statement, compared to our end result. This chapter also contains recommendations for further work, in case anyone decides to continue development on the project.

6.1 Conclusions

This project set out to develop a proof-of-concept full-stack application capable of managing data retrieval in a multi-tenant environment for FiiZK Digital AS.

Throughout this thesis, we have outlined the methodical approach we employed, from using agile methods and a SCRUM-like approach, to selecting technologies and developing methods using REST, GraphQL, and Kafka for data transfer.

Our efforts have resulted in a robust application that meets the requirements set by FiiZK. The application provides a scalable, secure, and user-friendly interface that enhances data accessibility and management for multiple users, including the capability for users to securely manage and and share a subset of their data effectively.

Despite the challenges of adopting new technologies and managing project coordination, the practical experience gained and the successful development of the application demonstrate the project's success.

The application was developed to manage data retrieval in a general sense, giving the application a broader use-case rather than just for the aquaculture industry.

6.2 Further Work

This section consists of suggestions for further work that can be done to the solutions or project as a whole.

We realised towards the end that we could add an almost unlimited amount of features to the project. Because of this we had to put a hard deadline for when we would stop the development when we went over the planned "Finish Product Development" deadline, as detailed in Table 3.

Here are some suggestions that we could implement to further improve our product:

- When creating a new user, also automatically create a new PostgreSQL database and assign it to the user.
- Add the possibility to include GraphQL mutations to the GraphQL key permissions, to align it better with the REST API solution.
- Return the ApiKey object alongside the encrypted key when creating an key, to follow best practices. This would allow the frontend to update the list of keys more easily, compared to having to re-fetch the whole Key list after a key is created.
- Implement a testing scenario to utilize Kafka's potential in regards to handling a high volume of messages per second.

Societal Impact

Our project is licensed under the MIT license. This license allows anyone to use, modify, distribute and commercialize our source code. By contributing to the open-source community we enable others to learn, or develop other useful systems or solutions using our code.

We have also written our code using the SOLID principles, as well as structured our project to be as easy to read and understand as possible, making it easier for others to use, modify or learn from our code.

We used docker containers to containerize our solution. This way we ensure that the environment will be the same for anyone running the application, avoiding common environment issues like different operating systems and installation of packages.

References

- [1] Alexander S. Gillis. What is object-oriented programming. <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>. [Online; accessed 14-May-2024].
- [2] Samuel Oloruntoba and Anish Singh Walia. Solid: The first 5 principles of object oriented design. <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>. [Online; accessed 14-May-2024].
- [3] Microsoft contributors. What is .net? <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>. [Online; accessed 05-May-2024].
- [4] ——. A tour of the c sharp language. <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>. [Online; accessed 05-May-2024].
- [5] MDN contributors. Spa (single-page application). <https://developer.mozilla.org/en-US/docs/Glossary/SPA>. [Online; accessed 18-May-2024].
- [6] Vue contributors. Vue - introduction. <https://vuejs.org/guide/introduction>. [Online; accessed 18-May-2024].
- [7] Wikipedia contributors. Javascript — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/JavaScript>. [Online; accessed 05-May-2024].
- [8] Typescript contributors. Typescript is javascript with syntax for types. <https://www.typescriptlang.org/>. [Online; accessed 18-May-2024].
- [9] geeksforgeeks contributors. Introduction to tailwind css. <https://www.geeksforgeeks.org/introduction-to-tailwind-css/>. [Online; accessed 18-May-2024].
- [10] Red Hat contributors. What is apache kafka? <https://www.redhat.com/en/topics/api/what-is-a-rest-api>. [Online; accessed 05-May-2024].
- [11] Roy Thomas Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

-
- [12] Red Hat contributors. What is graphql? <https://www.redhat.com/en/topics/api/what-is-graphql>. [Online; accessed 05-May-2024].
- [13] How To GraphQL. GraphQL is the better rest? <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>. [Online; accessed 05-May-2024].
- [14] Red Hat contributors. What is apache kafka? <https://www.redhat.com/en/topics/integration/what-is-apache-kafka>. [Online; accessed 05-May-2024].
- [15] Oracle contributors. What is a database? [https://www.oracle.com/database/what-is-database/#:~:text=A%20database%20is%20an%20organized,database%20management%20system%20\(DBMS\)](https://www.oracle.com/database/what-is-database/#:~:text=A%20database%20is%20an%20organized,database%20management%20system%20(DBMS)). [Online; accessed 05-May-2024].
- [16] Wikipedia contributors. PostgreSQL — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/PostgreSQL>. [Online; accessed 05-May-2024].
- [17] PostgreSQL contributors. About. <https://www.postgresql.org/about/>. [Online; accessed 05-May-2024].
- [18] Wikipedia contributors. Multitenancy — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Multitenancy>. [Online; accessed 05-May-2024].
- [19] —. Encryption — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Encryption>. [Online; accessed 05-May-2024].
- [20] —, "Https — Wikipedia, the free encyclopedia," <https://en.wikipedia.org/wiki/HTTPS>, 2024, [Online; accessed 19-April-2024].
- [21] Swagger contributors. Bearer authentication. <https://swagger.io/docs/specification/authentication/bearer-authentication/>. [Online; accessed 05-May-2024].
- [22] APIDOG contributors. What is bearer token and how it works? <https://apidog.com/articles/what-is-bearer-token/>. [Online; accessed 05-May-2024].
- [23] Microsoft contributors. Globally unique identifier (guid). <https://learn.microsoft.com/en-us/previous-versions/windows/desktop/automat/globally-unique-identifier-guid->. [Online; accessed 05-May-2024].
- [24] Wikipedia contributors. Docker (software) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software)). [Online; accessed 05-May-2024].
- [25] —. Postman (software) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Postman_\(software\)](https://en.wikipedia.org/wiki/Postman_(software)). [Online; accessed 05-May-2024].
- [26] —. Api platform. <https://www.postman.com/api-platform/>. [Online; accessed 05-May-2024].
- [27] —. Dbeaver — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/DBeaver>. [Online; accessed 05-May-2024].
- [28] —. Openai — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/OpenAI>. [Online; accessed 05-May-2024].
- [29] —. Chatgpt — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/ChatGPT>. [Online; accessed 05-May-2024].

-
- [30] ——. Github copilot — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/GitHub_Copilot. [Online; accessed 05-May-2024].
- [31] ——. Git — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Git>. [Online; accessed 05-May-2024].
- [32] ——. Github — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/GitHub>. [Online; accessed 05-May-2024].
- [33] ——. Nginx — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Nginx>. [Online; accessed 05-May-2024].
- [34] Overleaf contributors. The secret to better scientific and technical writing. <https://www.overleaf.com/about/why-latex>. [Online; accessed 18-May-2024].
- [35] Ken Schwaber and Jeff Sutherland. The 2020 scrum guide. <https://scrumguides.org/scrum-guide.html>. [Online; accessed 15-May-2024].
- [36] Claire Drumond. What is scrum? <https://www.atlassian.com/agile/scrum>. [Online; accessed 15-May-2024].
- [37] Atlassian. User stories in agile project management. <https://www.atlassian.com/agile/project-management/user-stories>. [Online; accessed 28-April-2024].
- [38] Simen H. Veum, Adrian R. Dahl, Ole T. Aanderaa. Bachelor github repository. <https://github.com/sh-veum/Bachelor>. [Online; accessed 18-May-2024].
- [39] Simen H. Veum. Demo: Managing data retrieval in a multi-tenant environment. <https://youtu.be/o9TnndkU0Yg>. [Online Video; accessed 12-May-2024].
- [40] Rick Anderson. (2024) Asp.net core identity. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio>. [Online; accessed 05-May-2024].
- [41] Joe Carder. How (and why) to use kafka with zookeeper. <https://www.openlogic.com/blog/using-kafka-zookeeper#what-is-zookeeper-01>. [Online; accessed 05-May-2024].
- [42] Apache Kafka. Apache kafka use cases. <https://kafka.apache.org/uses>. [Online; accessed 04-May-2024].

Appendix

A Project Plan

Project number: 01

Managing Data Retrieval in a Multi-tenant Environment

Preliminary Project Plan

Version <1.3>

Project number: 01

Revision History

Dato	Versjon	Beskrivelse	Forfatter
<19/01/24>	<1.0>	Første utkast	Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum
<22/01/24>	<1.1>	Lagt til vedlegg	Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum
<26/01/24>	<1.2>	Lagt til 3-parts avtale og arbeidskontrakt	Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum
<22/02/24>	<1.3>	Rewritten in English and updated	Adrian R. Dahl, Ole T. Aanderaa, Simen H. Veum

Project number: 01

Table of Contents

1. Objectives and Goals	4
1.1 Orientation	4
1.2 Project Description and Outcome	4
1.2.1 Project Description	4
1.2.2 Outcome Objectives	5
1.3 Impact Objectives	5
1.4 Frameworks	5
2. Organization	5
2.1 Project Group	5
2.2 Supervisor	6
2.3 Client	6
3. Implementation	6
3.1. Main Activities	6
3.2. Milestones	7
4. Monitoring and Quality Assurance	7
4.1 Quality Assurance	7
4.2 Reporting	8
5. Risk Assessment	8
6. Appendices	9
6.1 Schedule	9
6.2 Address List	10
6.3 Agreement Documents	11
6.3.1 Arbeidskontrakt for bachelor-gruppen	11
6.3.2 3-partsavtale	13

Project number: 01

1. Objectives and Goals

1.1 Orientation

This project was offered by NTNU along with other bachelor's projects and was selected after a discussion within our group where we concluded that this project appeared to be a sufficient challenge that would allow us to use our existing knowledge and skills, as well as develop them further.

1.2 Project Description and Outcome

1.2.1 Project Description

The objective of this project is to develop a containerized full-stack application to test and evaluate different methods of secure data retrieval, primarily focusing on REST API and GraphQL with the option to explore methods like RabbitMQ and Kafka if we have time.

The application will be in a multi-tenant environment¹ where each user has their own database that only they and system admins can access.

Using a graphical user interface, the users will be able to create keys for third parties that can access a subset of the data in their assigned database. When creating the key, the user will be able to specify what data the key can access. The third party would be able to insert their key in the application to see what data they are able to retrieve. An admin is able to see all the users and is able to change which database a user is linked to.

In the REST API solution, the key will hold information about the API endpoint anyone with that key can access, for example if the key only had access to “api/all-species”, would return all species in the database, but nothing else.

¹ A multi-tenant environment hosts multiple clients (users) on a single instance.

Project number: 01

In the GraphQL solution the key would be able to specify the fields of a table in the database that the key can access. For example, if the database has a table named *Species* with the fields *name* and *id*, the key could limit the access to just the *name* field for *Species*.

1.2.2 Outcome Objectives

The end product will be a containerized full-stack application with a user-friendly graphical user interface where users can manage access to their data. The application will be a proof of concept of a system that will allow users to create keys that will give access to a subset of their data, which can be given to third parties. This is so the users can specify the exact data they want to share with third parties.

1.3 Impact Objectives

The long term goal for the project is to have developed a robust full-stack application for managing data access that is easy to maintain and expand. The project will allow users to easily and securely share subsets of their data to third parties.

1.4 Frameworks

Everyone in the group must have access to a computer and the internet, with the necessary software installed to be able to work on the project.

2. Organization

2.1 Project Group

Simen Haug Veum

Student, Team Leader and Meeting Planner

Adrian Rennan Dahl

Student, Delivery Manager

Ole Tønning Aanderaa

Student, Document Manager and Quality Assurance

Project number: 01

2.2 Supervisor

Rituka Jaiswal

Associate Professor, Department of ICT and Natural Sciences NTNU Ålesund

2.3 Client

Marius Lundbø

Senior Software Developer, FiiZK

3. Implementation

3.1. Main Activities

The entire group will participate in developing the project. We will try to evenly distribute tasks among the various members of the group. Each member will receive their respective issues to work on, but other members can assist when necessary.

The backend will be written in a .NET 8 application that accesses multiple PostgreSQL databases. There will be one *main* database and multiple *user* databases. The *main* database contains information about all of the users (login info) that is only accessible for the administrators. The *user* databases will contain all of the data that the user can access. The .NET 8 application will be responsible for the communication between the databases.

The frontend will be written in Vue with TypeScript, and will give the user an easy way to create the keys they will give to third parties. The frontend will also be a testing-ground for the access keys to ensure they work. The administrators will also be able to use the frontend website to get an overview of the users and their assigned database.

The application will be containerized using Docker, this ensures that the application will run in the same environment on different machines. These technologies were chosen primarily because they are the recommended technologies that FiiZK already uses, and recommended for the project.

Project number: 01

Results will be documented in the final report, which everyone will work on equally. The main focus at the start is to program the full-stack application to have the product completed as early as possible, so we can well before the deadline shift the main focus to the report.

3.2. Milestones

Date	Milestone
26.01.2024	Deliver Project Plan
26.01.2024	Set up Github
26.01.2024	Make Wireframes for the frontend
09.02.2024	MVP Frontend
09.02.2024	MVP Backend
16.02.2024	Complete REST API Solution
08.03.2024	Complete GraphQL Solution
29.03.2024	Complete undecided 3rd Solution
April 2024	Oral Presentation of the Project in English
23.04.2024	Finished Project Development
07.05.2024	Ask Supervisor for report feedback
21.05.2024	Deliver Bachelor Thesis

4. Monitoring and Quality Assurance

4.1 Quality Assurance

Everyone must ensure that what they deliver is of the highest possible quality. We also have a member responsible for quality assurance, who has an extra focus on ensuring that the quality of what is delivered is high.

Project number: 01

Regarding other significant documents and tasks, it is the duty of each member to involve the other team members. This ensures a broader range of input and helps to achieve the highest quality of work, as well as a result that the whole group can unite behind and be satisfied with.

4.2 Reporting

We will have meetings with the advisor and client every other week, where we conduct a sprint review. With the advisor, we will also have a sprint retrospective every other week. In the weeks in between, the group will have an internal sprint review and retrospective.

5. Risk Assessment

Risk analysis that assesses vulnerabilities in the project.

	Harmless	Less severe	Severe	Very severe	Catastrophic
Verylikely	5	10	15	20	25
More likely	4	8	12	16	20
Likely	3	6	9	12	15
Less likely	2	4	6	8	10
Unlikely	1	2	3	4	5

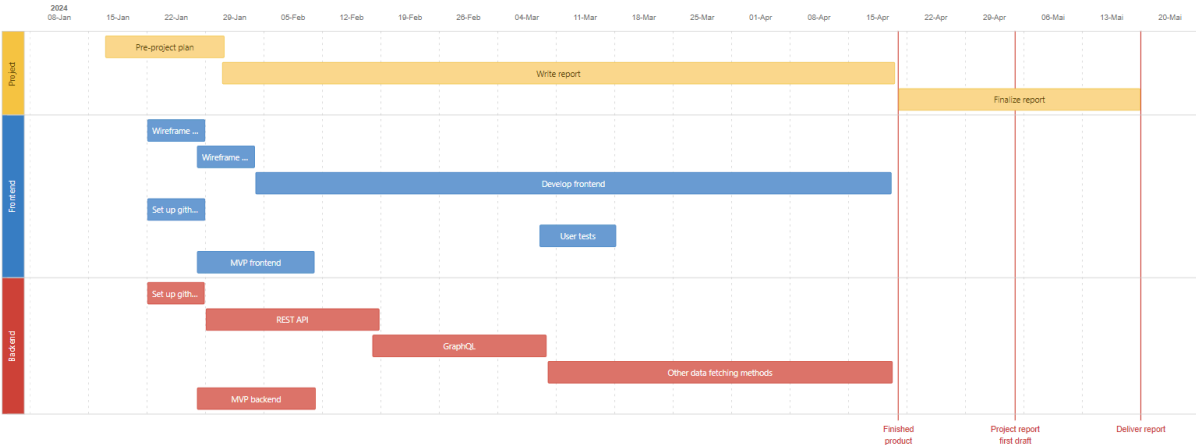
Event	Probabililty	Consequence	Risk factor	Measurement
Dropout of a group member	Unlikely	Very severe	4	Keep working.
Can't finish project in time	Unlikely	Catastrophic	5	Set the scope for the project down to a manageable size. Work consistently and document things along the way. Start writing the report in good time before

Project number: 01

				submission.
Bad communication between group members	Unlikely	Severe	3	Have a meeting and agree on a solution to communicate better
The quality of the finished product is unsatisfactory.	Less likely	Very severe	8	Maintain effective communication with both supervisor and client to discuss the quality of the product and understand their expectations of it.
Group member is ill	Less likely	Severe	6	Try to work together online instead of physical meetings while they are ill
Insufficient overview of deadlines and the required tasks.	Less likely	Less severe	4	Start early with effective planning and identifying essential tasks.

6. Appendices

6.1 Schedule



Project number: 01

6.2 Address List

Navn	Firma	Telefon (+47)	E-post
Simen Haug Veum	NTNU	91595359	simenhv@stud.ntnu.no
Adrian Rennan Dahl	NTNU	99088579	adrianrd@stud.ntnu.no
Ole Tønning Aanderaa	NTNU	92810548	oleton@stud.ntnu.no
Rituka Jaiswal	NTNU	40641567	rituka.jaiswal@ntnu.no
Marius Lundbø	FiizK	45233577	marius.lundbo@fiizk.com

6.3 Agreement Documents

6.3.1 Arbeidskontrakt for bachelor-gruppen

Arbeidskontrakt for Gruppe 4

Medlemmer: Adrian Rennan Dahl, Ole Tonning Aanderaa, Simen Haug Veum

Innledende tekst

Denne arbeidskontrakten bygger på et sett med typiske mål, oppgavefordelinger, prosedyrer og retningslinjer for interaksjoner for studentarbeider. Arbeidskontrakten er utfylt med *egne* fortolkninger av hva man mener med disse og hvordan man skal oppnå dette.

Roller og oppgavefordeling (*Hvordan organiserer man arbeidet?*)

Hvilke roller/ ansvarsområder er formålstjenlig for samarbeidet i prosjekt-gruppen?

Rolle	Beskrivelse	Tildelt
Teamleader	Koordinering, kontaktperson, sørger for at alle bidrar.	Simen Haug Veum
Dokumentansvarlig	Håndterer og organiserer alle dokumenter.	Ole Tonning Aanderaa
Leveringsansvarlig	Sørger for at alt blir levert til rett tid. Holder oversikt over tidsfrister og milepæler.	Adrian Rennan Dahl
Kvalitetssikring	Overvåker kvaliteten på arbeidet som skrivefeil, struktur, og at produktet følger design guidelines.	Ole Tonning Aanderaa
Møteinnkaller	Organiserer og kaller inn til møter, setter opp en agenda og sørger for at nødvendige dokumenter er tilgjengelige for alle deltakere	Simen Haug Veum

Prosedyrer (*hvordan gjør man ting?*)

A. Møteinnkalling

Når skal man ha møter. Hvordan innkalles det.

- Gruppen skal ha en form for møte hver mandag og fredag.
- Gruppen skal kalle inn til møter med veileder og oppdragsgiver annenhver uke.
- Møter med veileder og oppdragsgiver kalles inn via mail, ellers via Discord.

B. Varsling ved fravær eller andre hendelser

Dersom man kommer for sent eller ikke kan møte

- Varsling skjer gjennom Discord i god tid før fraværet

C. Dokumenthåndtering

Prosedyrer for lagring, samskriving, versjonshåndtering

- Lagring av dokumenter skjer i Google Drive og Confluence
- Oversikt over prosjekt, sprints og issues skjer i Atlassian JIRA & Confluence
- Kommunikasjon skjer i Discord
- Versjonshåndtering av prosjektet håndteres i GitHub

Project number: 01

D. *Innleveringer av gruppearbeider*

Ferdigstilling, kvalitetskontroll av innholdet, holde frister

- a. Alt som skal leveres blir kvalitetssikret og det blir gitt tilbakemelding om det er noe som mangler, eller om det er klart til å leveres av leveringsansvarlig.

Interaksjon (*Hvordan opptrer man sammen?*)

A. *Oppmøte og forberedelse*

- a. Alle møter til avtalt tid med mindre det er gitt info om fravær. Helst delta på forelesninger, eventuelt se dem digitalt.
- b. Alle gjør issues og oppgaver til beste evne, og sier ifra om noe ikke blir gjort i tide.

B. *Tilstedeværelse og engasjement*

- a. Alle skal være til stede og bidra til gruppearbeidet.

C. *Hvordan støtte hverandre*

- a. Alle tilbyr å hjelpe til om en i gruppa får trøbbel med en oppgave/issues.

D. *Uenighet, avtalebrudd*

- a. Om uenighet oppstår har vi først et internt møte, om problemet ikke løses blir det tatt til veileder.

Underskrevet av

Adrian R. Dahl

Adrian Rennan Dahl

Ole Tønning Aanderaa

Ole Tønning Aanderaa

Simen H. Veum

Simen Haug Veum

6.3.2 3-partsavtale



Norwegian University of Science and Technology

Approved by the Pro-Rector for Education 10 December 2020

STANDARD AGREEMENT

on student works carried out in cooperation with an external organization

The agreement is mandatory for student works such as master's thesis, bachelor's thesis or project assignment (hereinafter referred to as works) at NTNU that are carried out in cooperation with an external organization.

Explanation of terms

Copyright

Is the right of the creator of a literary, scientific or artistic work to produce copies of the work and make it available to the public. A student thesis or paper is such a work.

Ownership of results

Means that whoever owns the results decides on these. The basic principle is that the student owns the results from their own student work. Students can also transfer their ownership to the external organization.

Right to use results

The owner of the results can give others a right to use the results – for example, the student gives NTNU and the external organization the right to use the results from the student work in their activities.

Project background

What the parties to the agreement bring with them into the project, that is what each party already owns or has rights to and which is used in the further development of the student's work. This may also be material to which third parties (who are not parties to the agreement) have rights.

Delayed publication (embargo)

Means that a work will not be available to the public until a certain period has passed; for example, publication will be delayed for three years. In this case, only the supervisor at NTNU, the examiners and the external organization will have access to the student work for the first three years after the student work has been submitted.

Project number: 01

1. Contracting parties

The Norwegian University of Science and Technology (NTNU) Department: ICT and Natural Sciences Faculty of Information Technology and Electrical Engineering
Supervisor at NTNU: Rituka Jaiswal email and telephone: rituka.jaiswal@ntnu.no , 40641567
External organization: FiiZK Contact person, email address and telephone number of the external organization: Marius Lundbø, marius.lundbo@fiizk.com , 45233577
Student: Simen Haug Veum Date of birth: 23.05.2001
Student: Adrian Rennan Dahl Date of birth: 03.07.2001
Student: Ole Tønning Aanderaa Date of birth: 09.01.1997

The parties are responsible for clearing any intellectual property rights that the student, NTNU, the external organization or third party (which is not a party to the agreement) has to project background before use in connection with completion of the work. Ownership of project background must be set out in a separate annex to the agreement where this may be significant for the completion of the student work.

2. Execution of the work

The student is to complete: (Place an X)

A master's thesis	
A bachelor's thesis	X
A project assignment	
Another student work	

Start date: 08.01.2024
Completion date: 21.05.2024

The working title of the work is: Managing Data Retrieval in a Multi-tenant Environment
--

Project number: 01

The responsible supervisor at NTNU has the overarching academic responsibility for the design and approval of the project description and the student's learning.

3. Duties of the external organization

The external organization must provide a contact person who has the necessary expertise to provide the student with adequate guidance in collaboration with the supervisor at NTNU. The external contact person is specified in Section 1.

The purpose of the work is to carry out a student assignment. The work is performed as part of the programme of study. The student must not receive a salary or similar remuneration from the external organization for the student work. Expenses related to carrying out the work must be covered by the external organization. Examples of relevant expenses include travel, materials for building prototypes, purchasing of samples, tests in a laboratory, chemicals. The student must obtain clearance for coverage of expenses with the external organization in advance.

The external organization must cover the following expenses for carrying out the work:

Coverage of expenses for purposes other than those listed here is to be decided by the external organization during the work process.

4. The student's rights

Students hold the copyright to their works ¹. All results of the work, created by the student alone through their own efforts, is owned by the student with the limitations that follow from sections 5, 6 and 7 below. The right of ownership to the results is to be transferred to the external organization if Section 5 b is checked or in cases as specified in Section 6 (transfer in connection with patentable inventions).

In accordance with the Copyright Act, students always retain the moral rights to their own literary, scientific or artistic work, that is, the right to claim authorship (the right of attribution) and the right to object to any distortion or modification of a work (the right of integrity).

A student has the right to enter into a separate agreement with NTNU on publication of their work in NTNU's institutional repository on the Internet (NTNU Open). The student also has the right to publish the work or parts of it in other connections if no restrictions on the right to publish have been agreed on in this agreement; see Section 8.

¹ See Section 1 of the Norwegian Copyright Act of 15 June 2018 [Lov om opphavsrett til åndsverk]

Project number: 01

5. Rights of the external organization

Where the work is based on or further develops materials and/or methods (project background) owned by the external organization, the project background is still owned by the external organization. If the student is to use results that include the external organization's project background, a prerequisite for this is that a separate agreement on this has been entered into between the student and the external organization.

Alternative a) (Place an X) General rule

<input checked="" type="checkbox"/>	The external organization is to have the right to use the results of the work
-------------------------------------	---

This means that the external organization must have the right to use the results of the work in its own activities. The right is non-exclusive.

Alternative B) (Place an X) Exception

<input type="checkbox"/>	The external organization is to have the right of ownership to the results of the task and the student's contribution to the external organization's project
--------------------------	--

Justification of the external organization's need to have ownership of the results transferred to it:

6. Remuneration for patentable inventions

If the student, in connection with carrying out the work, has achieved a patentable invention, either alone or together with others, the external organization can claim transfer of the right to the invention to itself. A prerequisite for this is that exploitation of the invention falls within the external organization's sphere of activity. If so, the student is entitled to reasonable remuneration. The remuneration is to be determined in accordance with Section 7 of the Employees' Inventions Act. The provisions on deadlines in Section 7 apply correspondingly.

7. NTNU's rights

The submitted files of the work, together with appendices, which are necessary for assessment and archival at NTNU belong to NTNU. NTNU receives a right, free of charge, to use the results of the work, including appendices to this, and can use them for teaching and research purposes with any restrictions as set out in Section 8.

8. Delayed publication (embargo)

The general rule is that student works must be available to the public.

Project number: 01

Place an X

<input checked="" type="checkbox"/>	The work is to be available to the public.
-------------------------------------	--

In special cases, the parties may agree that all or part of the work will be subject to delayed publication for a maximum of three years. If the work is exempted from publication, it will only be available to the student, external organization and supervisor during this period. The assessment committee will have access to the work in connection with assessment. The student, supervisor and examiners have a duty of confidentiality regarding content that is exempt from publication.

The work is to be subject to delayed publication for (place an X if this applies):

Place an X	Specify date
<input type="checkbox"/>	one year
<input type="checkbox"/>	two years
<input type="checkbox"/>	three years

The need for delayed publication is justified on the following basis:

If, after the work is complete, the parties agree that delayed publication is not necessary, this can be changed. If so, this must be agreed in writing.

Appendices to the student work can be exempted for more than three years at the request of the external organization. NTNU (through the department) and the student must accept this if the external organization has objective grounds for requesting that one or more appendices be exempted. The external organization must send the request before the work is delivered.

The parts of the work that are not subject to delayed publication can be published in NTNU's institutional repository – see the last paragraph of Section 4. Even if the work is subject to delayed publication, the external organization must establish a basis for the student to use all or part of the work in connection with job applications as well as continuation in a master's or doctoral thesis.

9. General provisions

This agreement takes precedence over any other agreement(s) that have been or will be entered into by two of the parties mentioned above. If the student and the external organization are to enter into a confidentiality agreement regarding information of which

Project number: 01

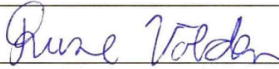

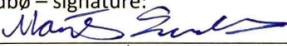

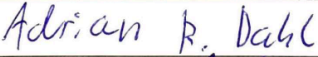

the student becomes aware through the external organization, NTNU's standard template for confidentiality agreements can be used.

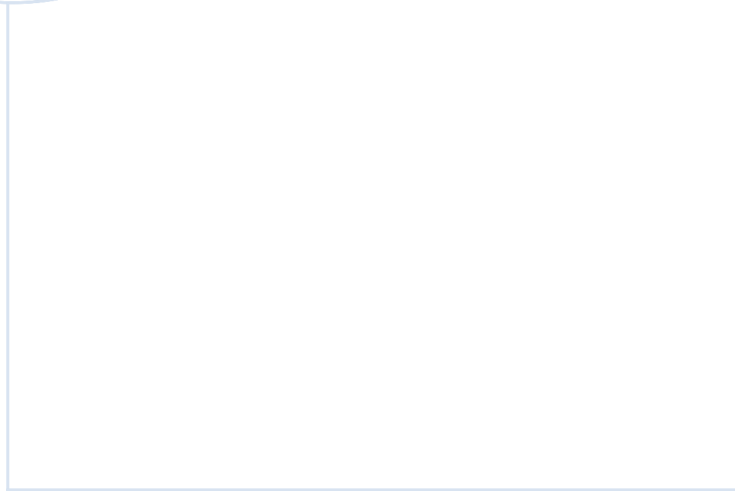
The external organization's own confidentiality agreement, or any confidentiality agreement that the external party has entered into in collaborative projects, can also be used provided that it does not include points in conflict with this agreement (on rights, publication, etc). However, if it emerges that there is a conflict, NTNU's standard contract on carrying out a student work must take precedence. Any agreement on confidentiality must be attached to this agreement.

Should there be any dispute relating to this agreement, efforts must be made to resolve this by negotiations. If this does not lead to a solution, the parties agree to resolution of the dispute by arbitration in accordance with Norwegian law. Any such dispute is to be decided by the chief judge (sorenskriver) at the Sør-Trøndelag District Court or whoever he/she appoints.

This agreement is signed in four copies, where each party to this agreement is to keep one copy. The agreement comes into effect when it has been signed by NTNU, represented by the Head of Department.

Signatures:

Head of Department: Rune Volden – signature: Date: 25.01.2024	
Supervisor at NTNU: Rituka Jaiswal – signature: Date: 25.01.2024	
External organization: FiiZK, Marius Lundbø – signature: Date: 26.01.2024	
Student: Simen Haug Veum – signature: Date: 23.01.2024	
Student: Adrian Rennan Dahl – signature: Date: 23.01.2024	
Student: Ole Tønning Aanderaa – signature: Date: 23.01.2024	



 **NTNU**

Norwegian University of
Science and Technology