

Torbjørn Bakke
Sindre August Strøm
Henrik Tengs Hafsvang

System development of dashboard application

Visualizing customer energy data

Bachelor's thesis in Computer Engineering
Supervisor: Jonathan Jørgensen
May 2022

Torbjørn Bakke
Sindre August Strøm
Henrik Tengs Hafsø

System development of dashboard application

Visualizing customer energy data

Bachelor's thesis in Computer Engineering
Supervisor: Jonathan Jørgensen
May 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Preface

This report is written in conjunction with the bachelor thesis of three students from the Computer Engineering program of the Department of Computer Science at NTNU. The assignment is given by Enoco AS, a company working with automation and energy control based in Stjørdal, Trøndelag.

The assignment was chosen by the students because it focused on system development and frontend design, as well as providing the chance to work with a forward-looking mix of familiar and unfamiliar technology. The students were also eager to get the experience of working on an assignment with an external company, as opposed to an internal assignment from NTNU.

Working with the assignment has given a lot of experience when it comes to combining multiple frameworks to make an application. It has been helpful and educational to receive feedback from the engineering department at Enoco during the developing process. The process of working with new people taught the team how to work together as a team towards a shared goal.


In the end, the team would like to thank our advisors from Enoco, Bjørn Kristian Punsvik and Simon Lilleeng, for their support and assistance, and especially for the workshops introducing us to their technology stack. We also want to thank our supervisor, Jonathan Jørgensen, for helping us with the startup of the project as well as providing valuable feedback on the documentation side of the project.



Torbjørn Bakke



Sindre August Strøm



Henrik Tengs Hafso

Project description

Before the start of the project, the team was given the following project description. The initial project description was written in Norwegian, this is a translated version.

Purpose of the task

Enoco's customers have expressed a need for an external dashboard that they can use in their lobbies or offices to visualize effectivity in a different way than with their main product, Eurora. This could for example be energy usage, CO2 equivalents and how they improve these values over time.

Short description of suggested task

The task will be to develop a standalone frontend application for an existing API, to visualize what the customer has need for. This could entail contact with the end users to uncover customer needs, independently come up with requirement specifications and conduct user tests. This application has not been designed before, and will function as an MVP.

Additional comments about what the task entails

Requirements and conditions for the task

- Use our Backend (MySQL, TypeORM, Typescript, GraphQL), access to relevant source code will be given.
- Can expand Backend as required.
- Frontend in VueJS, Vuetify is used as UI library

Abstract

This bachelor project deals with the design and development of a standalone web application on the behalf of Enoco AS, a company that deliver solutions for automating, analyzing, visualizing and increasing efficiency of energy usage for their clients' buildings. This application is meant to serve as a complimentary product to their existing product, Eurora. Eurora is a complex tool for analyzing and controlling the energy usage in your buildings. The application designed in this project is a much simpler dashboard application, which will primarily be used to visualize data from sensors, for example on a big screen in lobbies or offices.

The thesis question was formulated as such: How can Enoco's customers display the energy usage and other related data from their sites in a presentable way, on big screens in offices or lobbies, with a web application, and how can this application be made accessible, intuitive and user-friendly.

This was solved by developing a Nuxt/Vue based single page web application with a focus on simplicity, both in design, navigation and functionality. This application only consists of a login page and a main page. On the main page, the user can view all the sensors in their sites, and start a presentation mode with selected sensors. In terms of design, the team is happy with the resulting product, which has a simple and intuitive layout with easily understandable functionality.

After the end of the project, there are still multiple avenues for improvement and further development. In addition to some technical problems with the presentation mode, which does not always load correctly when sensor data is updated, the team also would have liked to add some options for customization of the presentation slides.

Sammendrag

Denne bachelorprosjektet omhandler design og utvikling av en frittstående web-applikasjon på vegne av Enoco AS, et selskap som leverer løsninger for automatisering, analyse, visualisering og effektivisering av energibruken til sine kunders bygninger. Denne applikasjonen skal fungere som et komplementært produkt til deres eksisterende produkt, Eurora. Eurora er et komplekst verktøy for å analysere og kontrollere energibruken i bygningene dine. Applikasjonen utviklet i dette prosjektet er en mye enklere dashboard-applikasjon, som primært skal brukes til å visualisere data fra sensorer, for eksempel på en storskjerm i lobbyer eller kontorer.

Problemstillingen for oppgaven ble formulert som følger: Hvordan kan Enocos kunder vise energibruken og annen relatert data fra sine bygninger på en presentabel måte, på storskjermer i kontorer eller lobbyer, med en web-applikasjon, og hvordan kan denne applikasjonen gjøres tilgjengelig, intuitiv og brukervennlig.

Dette ble løst ved å utvikle en Nuxt/Vue-basert ensideapplikasjon med fokus på enkelhet, både i utforming, navigasjon og funksjonalitet. Denne applikasjonen består kun av en påloggingsside og en hovedside. På hovedsiden kan brukeren se alle sensorene i bygningene deres, og starte en presentasjonsmodus med utvalgte sensorer. Utformingsmessig er teamet fornøyd med det resulterende produktet, som har et enkelt og intuitivt layout med lett forståelig funksjonalitet.

Etter prosjektets slutt er det fortsatt flere muligheter for forbedring og videreutvikling. I tillegg til noen tekniske problemer med presentasjonsmodusen, som ikke alltid lastes inn riktig når sensordata oppdateres, så ville teamet ha likt å legge til noen alternativer for tilpasning av presentasjonslysbildene.

Glossary

- API** Application programming interface, an endpoint of for example a database with which different applications can interact and make use of the application. ii, 12, 14, 18, 23
- Backend** The part of the system that is not contained or loaded into the application of the user, usually the API, server and databases. ii, 4, 10, 12, 14, 15, 17, 18, 23, 25
- CLI** Command Line Interface, terminal based user interface for a system or application. 5
- CRUD** Create/Request/Update/Destroy, the four basic operations on persistent storage like databases. 2, 11
- Daemon** A daemon is a program that only runs as a background process serving some utility, rather than being directly used by users. 5
- DevOps** A set of practices that combine software development and IT operations, with the goal of improving the development life cycle, both in terms of quality and speed. 3
- Framework** A big library with extensive functionality, often relying on a large number of other libraries. 4, 5, 10, 11, 23
- Frontend** The part of the system that the user has direct access to, generally contains UI and application logic. ii, 5, 10, 12, 14, 15, 17, 18, 23, 25
- Gantt** A Gantt chart is a type of bar chart that illustrates a project schedule, named after Henry Gantt. 14, 21, 29, 30
- IoT** Internet of Things, broad term describing physical devices, like sensors, that connect and exchange data with other devices and systems over the internet, or other networks. 3
- Library** An external collection of functionality that can be imported and utilized in your code. 5, 11–13
- MVP** Minimum viable product, the minimum functionality and polish for the product to be viable. Often used as a first draft. ii, 14, 22, 24, 26
- Stack** The technology stack, in software development, is the combination of all the programming languages, frameworks, libraries and other tools that are used to make an application or system. 10, 14, 26
- UI** User interface, the part of an application that a user can directly see and interact with, such as text, buttons, menus and so forth. ii, 4, 5, 7, 11
- Web application** An application that is stored on a remote server, and is accessed through a browser. 1, 4, 5, 10, 19
- Wireframe** A simplified version of the application, either made with digital tools or with paper, used to show and test the general layout and UI. 14, 19, 24, 33, 34

Contents

Preface	i
Project description	ii
Abstract	iii
Sammendrag	iv
Glossary	v
Table of contents	vi
1 Introduction	1
1.1 Background	1
1.2 Thesis problem	1
1.3 Existing products	1
2 Theory and background	2
2.1 Databases	2
2.1.1 Relational databases	2
2.1.2 Time series databases	3
2.1.3 Object Relational Mapping	3
2.2 Frontend web frameworks	4
2.2.1 Model-View-Viewmodel	4
2.2.2 Popular frameworks	5
2.3 OS virtualization with containers	5
2.4 Design Theory	6
2.4.1 Color theory	6
2.4.2 Material Design	7
2.4.3 Universal Design	7
2.5 Development process	7
2.5.1 Software development models	7
2.5.2 Version control	9
3 Method & Technology	10
3.1 Technology stack	10
3.1.1 Vue	10
3.1.2 Nuxt	11
3.1.3 Vuetify	11
3.1.4 ChartJS	11
3.1.5 Pinia	11
3.1.6 TypeORM	11
3.1.7 InfluxDB	12
3.1.8 GraphQL	12
3.1.9 Apollo Server and Client	12
3.1.10 Amazon Web Services	13
3.1.11 Test data	13
3.2 Method & process	14
3.2.1 Development process	14
3.2.2 Development method	15
3.2.3 Division of Labor and Roles	15
4 Results	16
4.1 Software development results	16
4.1.1 Functional requirements	16
4.1.2 Non-functional requirements	16
4.1.3 API	18
4.1.4 Web application	18
4.2 Scientific results	19

4.2.1	Application design	19
4.2.2	User testing	21
4.3	Administrative results	21
4.3.1	Working methodology	21
4.3.2	Progress plan	21
4.3.3	Time sheets	21
5	Discussions	22
5.1	Software development results	22
5.1.1	Functional requirements	22
5.1.2	Non-functional requirements	23
5.1.3	API	23
5.2	Scientific results	23
5.2.1	Application design	23
5.2.2	Testing	24
5.3	Administrative results	25
5.3.1	Working methodology	25
5.3.2	Progress	25
6	Conclusions & further development	26
6.1	Conclusions	26
6.2	Further work	26
A	Gantt diagram	29
A.1	Version 1	29
A.2	Version 2	30
B	Main view versions	31
B.1	Version 1	31
B.2	Version 2	32
C	Wireframe	33
C.1	Login view	33
C.2	Main view	33
C.3	Presentation view	34
D	External process documentation	34

1 Introduction

1.1 Background

Enoco is a company that deliver solutions for automating, analyzing, visualizing and increasing efficiency of energy usage for their customers' buildings.[1] Their main digital product is the cloud platform Eurora, which is an application used to control and oversee the customer's sites via web interfaces connected to different sensors at each site.[2] Eurora is complex tool with deep functionality, which gives the user an array of options for monitoring, analyzing and controlling their energy usage.

However, some customers have expressed a need for an alternative interface to their data, for example to display simple visualizations of their data in offices or lobbies. Eurora is not designed with this purpose in mind. In addition to being complicated and primarily designed for active use, it also has access to control functionality. It would be desirable with an application that only has access to select data and no control functionality. Because of this, Enoco wishes to develop a simplified standalone Web application that can work as complimentary product to Eurora. This web application would have read-only access to the databases, have a simple interface and a view or mode specifically designed to present data.

Enoco will in the rest of this report be referred to as the client of this project.

1.2 Thesis problem

How can the client's customers display the energy usage and other related data from their sites in a presentable way, on big screens in offices or lobbies, with a web application, and how can this application be made accessible, intuitive and user-friendly.

1.3 Existing products

Eurora is a Web application provided by Enoco, which is what their customers are currently using for displaying their energy usage and comparisons. Because this is an application that all of Enoco's customers are using, the team will draw a lot of inspiration from the design elements of Eurora when creating the external dashboard application. This will make design choices easier to make, and at the same time give a sense of familiarity to customers which makes it more intuitive to use.

InfluxDB provides a web interface where one can view and edit the data, make custom queries with various chart types, and even make a custom dashboard of selected queries with various chart types. While not being a directly comparable product, it did provide a lot of insight and inspiration that the team could make use of.

2 Theory and background

In this chapter, the theoretical background for choices made during the thesis will be provided. This will include the most important technologies and methods used during development, as well as theory linked to the thesis problem.

2.1 Databases

In the most general terms, a database is an organized collection of structured data stored in a computer system.[3, 4] Most modern databases use a Database Management System to let the users interact with the data, either directly through commands, or via other software. A Database Management System, DBMS for short, is a software system that is used to interact with the database itself. Most databases are also modeled as tables, with rows and columns of data. This makes organizing and accessing the data simple, and it also makes it easy to keep track of the relationships between different data. There is a large number of different types of databases, and an even bigger number of different DBMSs.[4]

A **database query language** is the programming language used as an interface for making CRUD(Create/Request/Update/Destroy) requests to a database. One of the most well known query languages is SQL (Structured Query Language), especially when it comes to relational databases. SQL enables high performance and internally consistent querying of the databases.[5] In recent years newer query languages such as GraphQL have become more popular due to more choices and easier usage, but despite this, SQL is still one of, if not the most, widely used querying languages in the world.[5, 6]

2.1.1 Relational databases

A relational database is a type of database where the data points have predefined relationships between them.[7] Relational databases are based on the relational model, which is a way of representing data with relations as a table of values. Each row in the relational database table holds a record, and each column represents an attribute. The columns of the tables hold the attributes of the data, and each record holds a value for each attribute. One or more attributes function a key for the database table, which is used to identify each record in a table.[8]

The relational data model provides a standard way of representing and querying data. One of the most useful aspects of the relational database model is in its use of tables, which is an intuitive, efficient and flexible way to store and access structured data from the database.[5] Relational databases excel at maintaining data consistency across applications and database instances, and are thus capable of ensuring that multiple instances of a database has uniform data across each of them.[5, 7]

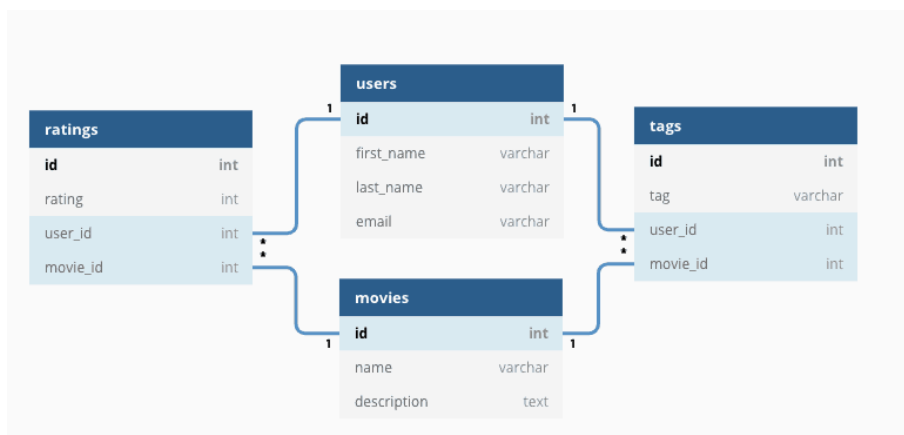


Figure 1: Relational model functions.¹

¹<https://dev.to/jasmcaus/want-to-learn-sql-here-s-everything-you-need-to-know-46co>

2.1.2 Time series databases

Time series databases, or TSDB for short, are databases optimized for collecting, storing, retrieving and processing time stamped or time series data. Time series data is sequential data collected over time intervals.[9, 10] When working with times series data it is common that you only need a specific time frame of data in high resolution, so you want to be able to manage the life cycle of your data. This allows you to retire data or compress it down to a smaller format.[10]

The main advantage of TSDBs is that they are designed to handle data that is both large in scope, and also accumulates or is queried in very quick succession.[9] TSDBs also provides efficient summarising techniques, allowing the user to summarize for example a value for the last hour in a quick manner. Another feature of TSDBs is the ability to perform large range scans of many records, meaning you are able to comb through billions of records and present the results to the client while still having acceptably short query times.[10]

The primary use cases for time series databases are IoT(Internet of Things), DevOps and real-time analytics. In the IoT space you see factories, oil and gas, agriculture, smart roads and infrastructure that want to have some metrics associated with what they are doing. When it comes to DevOps, common use cases would be custom monitoring solutions to track servers, virtual machines, applications, users or events in some format. Real-time analytics, as the name implies, is about apps that uses business, social or development metrics in real-time.[11]

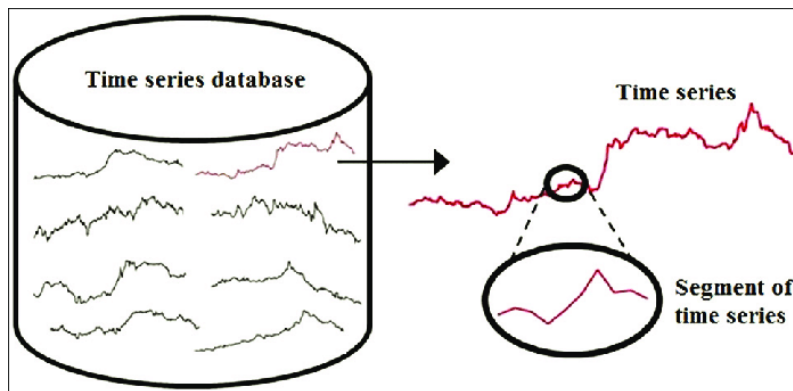


Figure 2: Relationships between time series database, time series and a segment of time series.²

2.1.3 Object Relational Mapping

Object Relational Mapping, or ORM for short, is a technique where you map entity models, for example in an API, with database tables. In this context, an entity model is an object oriented representation of some data entity. The reason why you would want to do this is because while the entities are represented as objects, most databases are unable to handle complex objects. Instead, each record in a relational database table is represented by a tuple of simple data values, like numbers or strings.[12] Because of this, using ORMs significantly simplifies the development process in most cases because it automates the object-to-table and table-to-object conversions.[13] Once an entity model is created, it becomes easier to update, maintain and reuse the code in other parts of the project.

²https://www.researchgate.net/figure/Relationships-between-time-series-database-time-series-and-a-segment-of-time-series_fig1_287897622

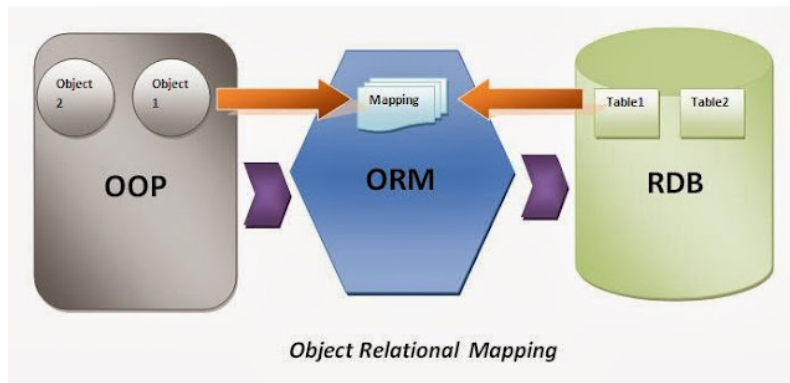


Figure 3: Relation between Object Oriented Programming language, ORM and Relational DataBase.³

2.2 Frontend web frameworks

When developing a web browser application, chances are high that you will need a lot of functionality and design elements that are commonly used in other Web applications. Rather than designing everything from scratch, it is often helpful to use one or more Frameworks that offer much of this out of the box. Not only will this save development time and costs, but it will also ensure that the application code will be more comparable and readable by other developers that are familiar with the framework(s) used.[14]

2.2.1 Model-View-Viewmodel

Because of this, its common to use web Frameworks such as React, Vue or Angular as the backbone of modern Web applications. These are all frameworks that use a Model-View-Viewmodel (MVVM) design pattern. Here, model represents the Backend business logic, view represents the UI layout, and viewmodel represents the intermediary code between the two, which handles the view logic. See figure 4.

The main purpose of a MVVM implementation is separating the logic of the UI from the business logic. This has several advantages, one of them being separation of concerns. This means that you avoid tightly coupled, change resistant code, which often can cause issues when maintaining or updating. Another important advantage is that it enables a developer-designer workflow, which, for example, means that a single developer can develop and test a full application without having much experience with UI design.[15]

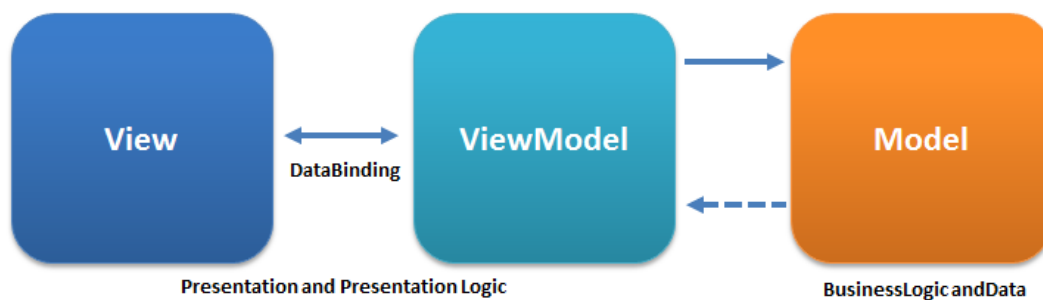


Figure 4: Model-View-Viewmodel structure.⁴

³<http://interviewquestionjava.blogspot.com/2014/01/framework-orm-object-relational-mapping.html>

⁴<https://commons.wikimedia.org/wiki/File:MVVMPattern.png>

One of the key features of MVVM Frameworks is data binding between the data in the view and the logic in the viewmodel. Data binding is the practice of linking a data provider and data consumer, and with this enabling a two-way connection. This is very useful in the context of Frontend development because you want to link the UI to the logic, but these are generally in two different languages, for example HTML and JavaScript. By using data binding, the UI elements in the view can be dynamically changed from the logic in the viewmodel.[16]

2.2.2 Popular frameworks

The most common MVVM based JavaScript/TypeScript web frameworks are, as previously mentioned, React, Vue and Angular. These Frameworks all similarly work as a backbone and a toolbox to build a Web application with, and have many similarities in functionality and implementation, but also some key distinctions and different philosophies.[17, 18]

For **Angular**, this philosophy is to be an all-in-one solution. It is a large Framework with a lot of built in features. Because of this Angular, is often considered a platform rather than just a Frontend framework. Angular is also in an ecosystem managed by Google, which have created different tools to work alongside the platform, such as the Angular CLI and Angular Progressive Web App (PWA).[19] Angular is generally considered to have a steeper learning curve than the two others, in part due to the large amount of features.[17, 18]

React has taken the opposite approach to Angular, with a minimalistic approach, and focuses on UI building. React is designed for gradual adoption, so you can use as little or much of it as you want. React also refers to itself as Library and not a Framework. This is reflected by the fact that React in many ways is stripped down in terms of features, and instead relies on third party libraries for much functionality.[20] This simplicity makes React easier to pick up and learn.[18]

Vue is a Framework that in many ways is a middleground between Angular and React. Much like React, Vue is designed to be incrementally adoptable. The framework has some of the features that Angular offers built in, but not all of them. The focus is on certain core features, for example routing and state management, which are both built in. There are however many features that are only supported by community development, like for example foreign validation.[21] While Vue has wider built in functionality than React, it still has an easier learning curve in certain ways.[17, 18]

Frontend web Frameworks in general all work towards making the development of Frontend applications easier, quicker and more consistent. It makes code maintainable, easier for the programmer who made it and others who wants to test or contribute to the project. It makes it easier to reuse code, with complex interfaces usually getting made out of smaller components that can be reused elsewhere. Frontend frameworks also provides a consistent and intuitive UI, and make it easier to solve problems as they come up in development by having ready-made solutions. This avoids wasting time developing solutions to common issues.[22]

2.3 OS virtualization with containers

Operating System virtualization, or OS virtualization, is the practice of having multiple isolated user-space instances in the OS. These instances are called containers. This isolates the programs running inside the container from files, resources and other programs outside the container, which can be very beneficial for a number of reasons. This includes security, resource management and being able to run code under identical conditions on different computers or environments.[23, 24]

One of the most common OS virtualization platforms is Docker. Docker uses a client-server architecture, with most of the work being handled by the Docker Daemon, such as managing the Docker images and containers. See figure 5. A Docker image is a blueprint for creating a Docker container, where the container is a virtual space for some program, system or platform. Each such container usually runs on specific task, such as for example hosting a database instance.[24, 25]

⁵<https://docs.docker.com/get-started/overview/>

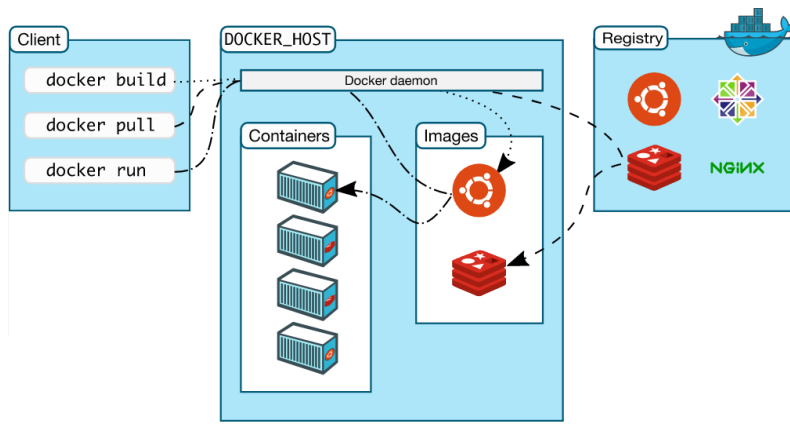


Figure 5: The Docker architecture, where the client interacts with the containers and images via the daemon.⁵

2.4 Design Theory

2.4.1 Color theory

Color theory is the rules and guidelines designers use to choose color schemes in visual interfaces. When choosing colors for a design, it is common to have four different colors in the scheme. When deciding what type of color scheme to use, it depends on what you want your users to see. An example of a color scheme is Monochromatic, which takes one color and fills the scheme with different shades of that color. Neutral colors like black, white and grey can be used with this scheme as it helps the primary color stand out. Another type of color scheme is Analogous, which uses a color as well as its neighbours to fill the palette.[26]

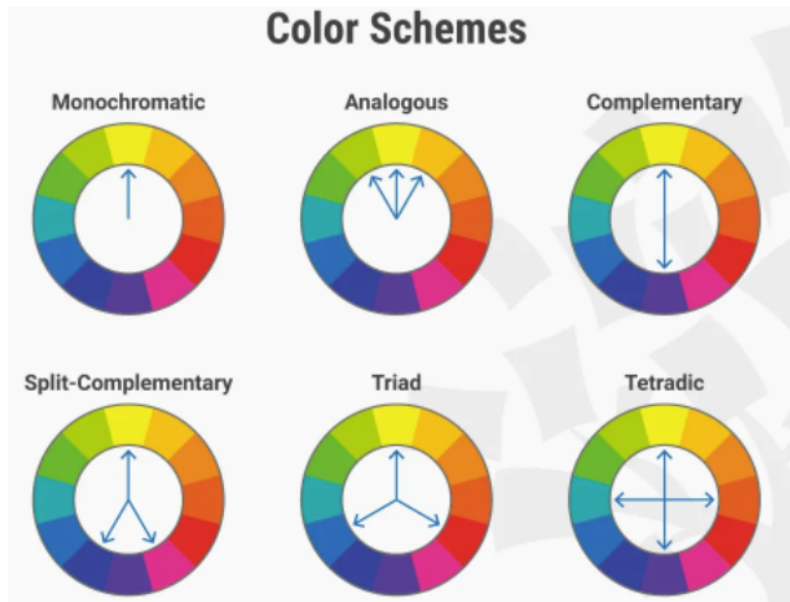


Figure 6: Color schemes.⁶

⁶<https://www.interaction-design.org/literature/topics/color-theory>

2.4.2 Material Design

Material design is a visual and interactive design concept from Google that is based around a hypothetical material, out of which all the UI elements, buttons, drawers, icons and themes are constructed. Material must behave in according with a number of guidelines set out in Google's "Material design spec sheet", a continuously updated document that was created to guide artist and programmers to make a more unified look for UI elements.[27] Design philosophies like this extend further than just rules for how a user interface or application layout should look like. It is a whole set of guidelines on how users interact with the interface, including small UI animations as a response to interactions. This can, along with other techniques, draw the users eyes to the correct part of the screen, which in turn enables more natural interaction.

2.4.3 Universal Design

Universal design is the design and composition of a product, which is accessible to people with a wide range of abilities, disabilities and other characteristics. A product following the principle of universal design communicates information effectively in a way that allows all people who wish to use it, to be able to. Products are typically designed to be most suitable for the average user, while universal design is aimed to be inclusive for all potential users.[28] An examples of universal design could be making the design simple and intuitive to use by the use of icons for different menu options.

2.5 Development process

When creating any kind of software, one has to go through some sort of system development process. This is the process of defining, designing, implementing and testing the software.[29] This process is often also called the software development life cycle (SDLC), and can be classified into many different models based on how the workflow of the development process is structured.[30]

2.5.1 Software development models

The software development models can be categorized by how 'formal' and how 'sequential' the development process is. Formal here refers to how much input the customer has during the development process, with informal implying frequent input and communication, and formal implying input only at the beginning of the process. How sequential a model is describes whether it is iterative or linear, or in other words how flexible it is. A model that is highly sequential is rigidly pre-planned, and you do each part of the development process in a specific order. This makes the process easier to plan, but also makes accounting for changes in the middle of the process much more difficult.[30] A chart illustrating how sequential and how formal popular models are can be found in figure 7.

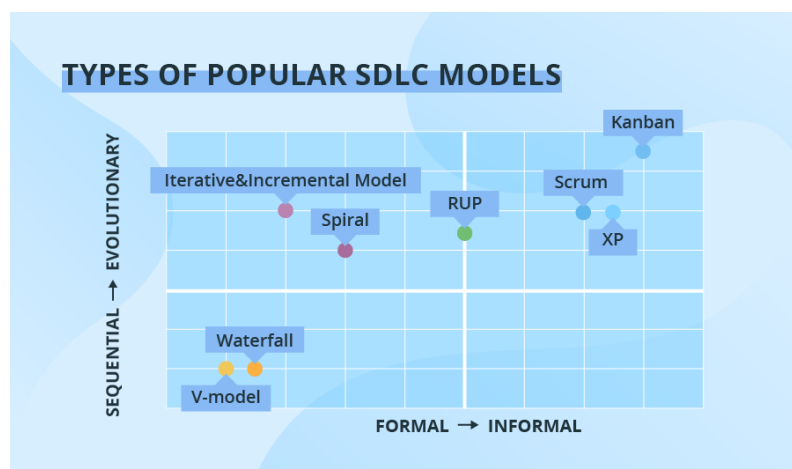


Figure 7: Popular software development models charted by how formal and how sequential they are.⁷

Two of the most famous models are the Waterfall model and the Scrum model. The Waterfall model is an older way of doing software development that can be categorized as formal and sequential. Scrum is a more modern model, and is on the opposite side of the spectrum, being informal and iterative. An illustration of the software development life cycle of both models can be seen in figure 8.

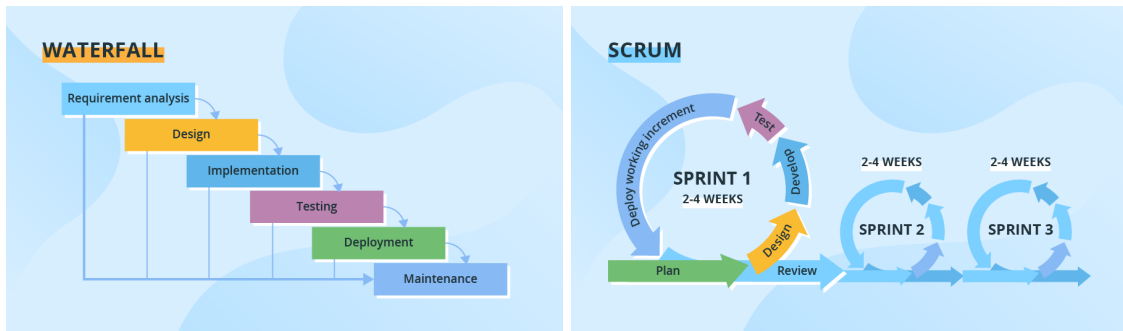


Figure 8: Waterfall model SDLC on the left, Scrum model SDLC on the right.⁷

More towards the middle, you can find moderate models such as the Iterative and Incremental model. These models are relatively incremental and flexible, but less informal than Scrum or Kanban. The Incremental model is split into iterations, where each iteration adds new software modules. These iterations can run sequentially or in parallel. The Iterative model is more flexible, with a focus on developing iterations that builds on the previous, which in many cases can allow for somewhat more client input during the process. See figure 9.

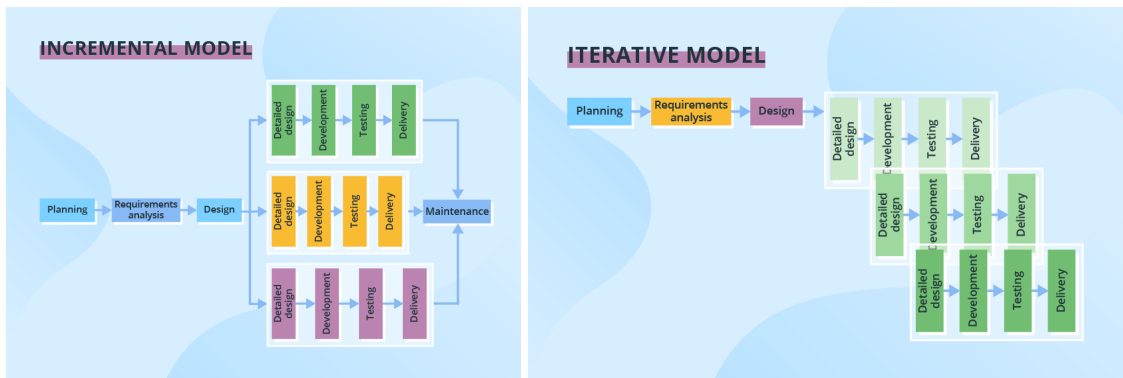


Figure 9: Incremental model SDLC on the left, Iterative model SDLC on the right.⁷

Generally, most developers are moving away from formal and sequential models such as Waterfall and V-model, and towards more informal and iterative models such as Scrum and Kanban. Models such as Scrum and Kanban are often referred to as Agile models.^[30]

⁷<https://www.scnsoft.com/blog/software-development-models>

2.5.2 Version control

A version control system (VCS) of some sort is almost always used in software development processes. The VCS is used to keep track of changes and different versions of the code. This is especially useful when multiple people are working on the same system, and when working with large systems over an extended period of time.[31] There are many different VCSs, such as SVN and Mercurial, but the most popular and widespread system today is Git.[32]

Git is a distributed version control system, which means that all versions are stored on every client, not just on a single centralized server. See figure 10. This means that all clients have access to every version at all times, as long as they keep their repository updated, even if the main server is unavailable.[31] While most VCSs have a unsaved working directory and a saved repository, Git also has a staging area in between. This allows you to stage only certain files or even parts of files for committing to the repository, while still working on other things.[33]

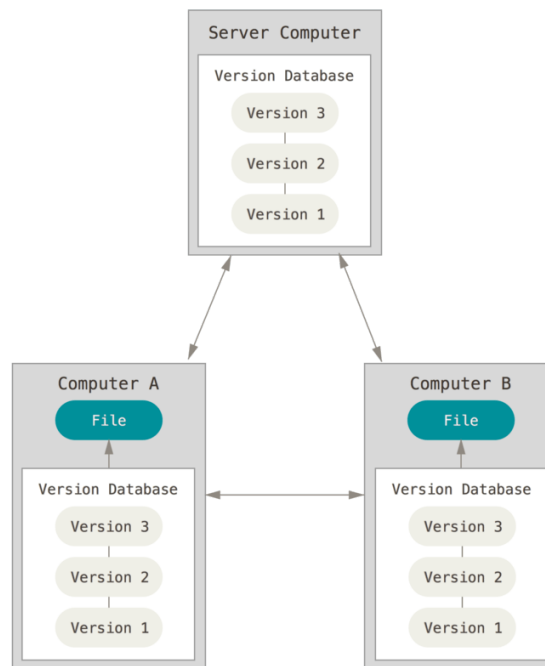


Figure 10: A distributed version control system.⁸

A core difference between Git and most other version control systems is that rather than storing changes as a list of file-based changes, Git stores a snapshot of the entire project file system with each version. This makes changing between versions simpler and more reliable, and enables a variety of options. Git uses checksums at every saved version (commit), which means there is also a high level of data integrity and safety.[34]

To use Git with multiple clients, you need somewhere to host the main repository. This is usually done with a Git hosting service, which hosts the code, and usually also offers a variety of tools. This can for example be viewing, testing, documenting, deploying or distributing the code, as well as additional features such as issues boards and project statistics. Examples of common hosting services for Git are GitHub, GitLab and AWS CodeCommit.[35–37]

⁸<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

3 Method & Technology

3.1 Technology stack

As the client already had an established Stack for developing Web applications, it was strongly advised that the team use at least the core elements of this stack. Not only would this make the end product more familiar and easier for them to integrate, but it would also make it more straight forward for them to guide and help the project along, as well as giving the team more concrete examples from the client's code base to work from.

The core of this stack was Vue2 with Vuetify, Nuxt and Apollo Client for Frontend with TypeORM, GraphQL and Apollo Server for Backend. Both Backend and Frontend were coded in TypeScript. The Backend is connected up to MySQL relational databases, as well as InfluxDB time series databases. The team used this stack as the core for the project, but used many other technologies of their own choosing, based on needs and preferences, especially in Frontend. For an illustration of the usage of this stack, see figure 11.

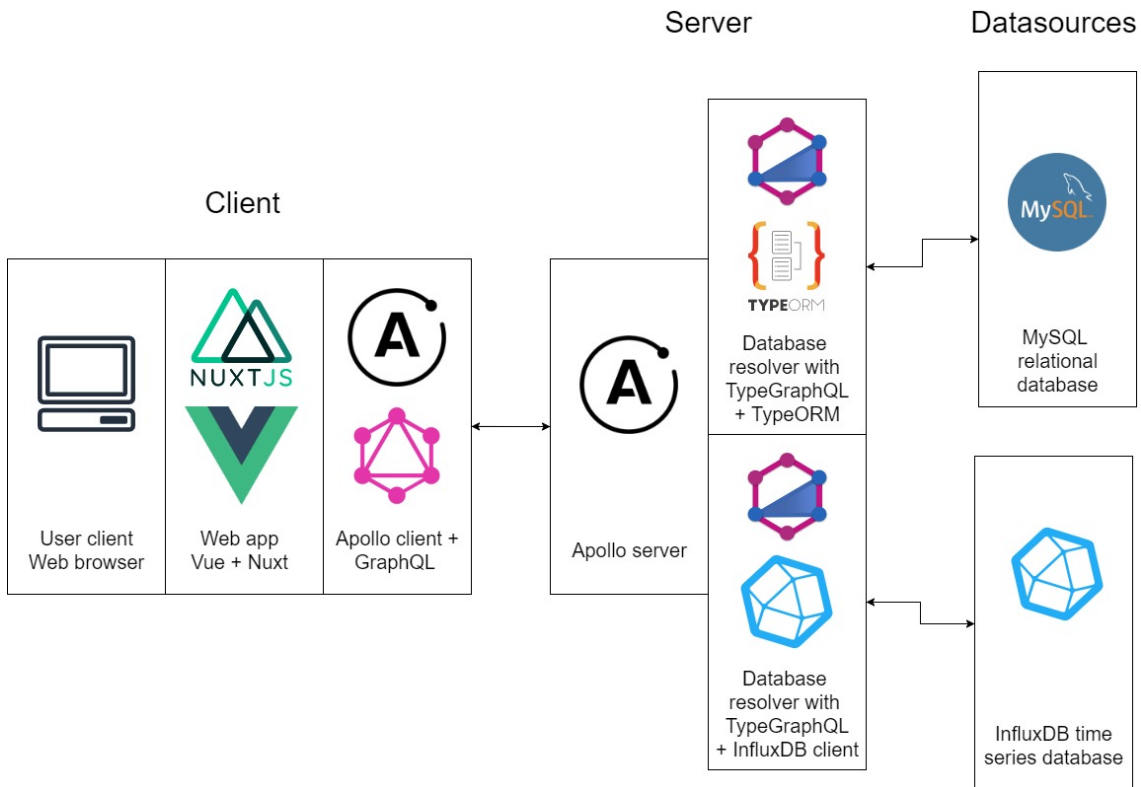


Figure 11: Project architecture of the core technology stack.

3.1.1 Vue

The Frontend Framework for this project was Vue, more specifically Vue2. Using Vue2 was suggested by the client because it was the framework Eurora was created in. Not only would it be easier for them to show examples and help guide the team when using Vue2, but it would also make adopting the project for their own use easier after the bachelor project. In addition to this, it was also the Frontend framework that the team had the most experience with. The team had minimal experience with React, and no experience at all with Angular. This, combined with the steeper learning curves of React and Angular, made Vue an obvious choice. The flat learning curve of Vue was important, given the limited time to learn the other technologies involved in the project.

Vue can start out as a small project without complex systems, and then later can be scaled up to an enterprise project. The limited scope of the project made this an important aspect. Vue has a large community of developers, and as such has many helpful community-made libraries. Aside from the features, Vue also has a slightly higher performance in most categories in comparison to React and Angular,[38] while taking up less space.[39]

3.1.2 Nuxt

Nuxt was used alongside Vue, and brought a lot of helpful features. The main appeal of using Nuxt is how it helps to create a server rendered application without having to write code from scratch for managing things like asynchronous data, various middleware and routing.[40, 41] The application needs to be able to fetch real-time data without having to reload. Among other things, this is facilitated by Nuxt, which made it a good fit for this project.

3.1.3 Vuetify

To help the team follow the Material Design philosophy, the best way was to use a Library or Framework that has this functionality built in to save time. To do this Vuetify was used. Vuetify is a UI Framework built on top of Vue. It has hundreds of pre-made components that are easy to implement to any Vue project.[42] Vuetify was used for almost all components, with the only significant exception being the charts.

It is also one of the most popular JavaScript Frameworks in the world, and is one of few options that offer long term support as well as business and enterprise support.[42] This is important for the project since it is meant to be used as a long term solution for the client. Lastly, Vuetify also support accessibility and Section 508 support,[42, 43] which means it also helped the team follow the principles of Universal Design during development.

3.1.4 ChartJS

In order to visualize the sensor data, a charting tool that was fast, dynamic and customizable was needed. Several alternatives were looked at, for example ChartJS, Vue.plotly and the Vuetify component Sparkline. Sparkline was not as customizable as the team needed, and plotly seemed to focus more on customizing the data while displaying it. The team also had some familiarity with ChartJS and its Vue implementation VueChartJS, which made it the most convenient choice. ChartJS is a powerful charting Library that has a lot of flexibility when it comes to visualization. ChartJS follows Vue's component principles, which makes it easy to use in a component.[44, 45]

3.1.5 Pinia

In order to save and access certain data like user information and settings in Vue, there was need for a state management system. A state management system is a tool used for Frameworks like Vue to access data from different places, like different components. This is useful since the alternative is excessive use of props, and there are also many problems that can not be solved with props at all.[46] Vue has a simple state management system built in, but additional functionality was needed.

At first, Vuex was considered as the choice for state management system, but then Pinia was discovered. Pinia is the successor to Vuex, is easier to implement, and will likely be more relevant in the future.[46, 47] Because of this, the team chose to use Pinia instead. Three different state management stores were made; one for user data, one for application state, and one for settings. To keep the store states even after reloading the page or exiting the web browser, persistence functionality was needed. For this, the team used a plugin for Pinia called 'pinia-plugin-persist'.

3.1.6 TypeORM

Two possible alternatives for ORMs were TypeORM and Prisma. TypeORM is a more traditional ORM which maps tables to model classes. These models are used to make interfaces for CRUD requests to an application at runtime. In comparison, Prisma is a new kind of ORM that mitigates many issues that occur during the usage of traditional ORMs, such as a bloated model instances that can happen in all project sizes. It reduces the problems of mixing business with storage logic, and uses a schema to define application models in a declarative way.[48] TypeORM and Prisma operate on different levels of abstraction. The students had experience with raw SQL queries from previous projects, and were familiar with the mirrored SQL of TypeORM, rather than Prisma's higher level of abstraction.

A few other key factors to be considered was the filtering, type safety, data modeling, as well as the the learning curve. Filtering with TypeORM generally works similarly to SQL operators, an example of this being the `find()` method. Prisma, on the other hand, provides a generic set of operators. This means that TypeORM makes use of filters based on the `ILike` operator for filtering strings, while Prisma provides developers with more specific operators such as `contains`, `startsWith` and `endsWith`. Prisma models are defined in its own Prisma schema while TypeORM uses classes and TypeScript decorators for model definition. Since the project was to be written in TypeScript, and the size of the project was not that large, TypeORM has an edge in terms of data modeling. TypeORM embraces TypeScript, which lets developers have a certain level of type safety for their database queries.[49] The team's existing familiarity with SQL and similar API model systems thus made it an acceptable solution.

3.1.7 InfluxDB

A time series database was required for the sensor data. This is because they are optimized for big and fast moving workloads. In a nominal relational databases such as MySQL, the overhead caused by such workloads would cause slow loading times, slow lookup and bad performance overall.

When looking at time series databases, the most popular were InfluxDB, Prometheus and TimescaleDB, with InfluxDB being the most popular by a wide margin.[50] Because of the large amount of sensor data that the application would potentially handle, performance was an important consideration here. InfluxDB has been found to have the fastest average query time among its contemporaries.[51, 52]

Since time was limited, the learning curve was another important factor. Looking through documentation of the mentioned time series databases, InfluxDB was the most thoroughly documented, and seemed to be the easiest time series database to quickly set up and get working. All this considered, having to use InfluxDB was not a problem, and likely the option the team would have gone with regardless.

To host the InfluxDB database instances used during development, Docker was used. Local instances of InfluxDB were created using a Docker image provided by Enoco, and test data was injected in the form of CSV files with anonymized data from Enoco's databases. For testing purposes, the date values of this data was shifted one year ahead in time. This was done because the data in the local databases would not be automatically updated, which would immediately become problematic while testing, since there would be no data for the most recent hours, days or even weeks. Another advantage with containerized local instances was that the databases were available offline, enabling work for example when travelling.

3.1.8 GraphQL

GraphQL is a query language for APIs, as well as a runtime on the server that you can use to query a variety of different data sources, not just databases. GraphQL queries are usually served over HTTP, which makes them ubiquitous and easy to work with.[53] Using GraphQL in combination with TypeORM, an API for the InfluxDB and MySQL databases was made. This included types and entities, as well as resolvers with operations. Since the API is written in TypeScript, a Library called `TypeGraphQL` was used to make typings for the entities and resolvers.[54]

3.1.9 Apollo Server and Client

In order to communicate between the Backend and Frontend, there was need for a GraphQL server and client. Apollo Server and Apollo Client are a popular choice, with good documentation and utilities, as well as being designed to work together.[55]

Apollo Server is an open-source GraphQL server, and is the GraphQL communication endpoint on the Backend. Apollo Server is straight forward to set up, universally compatible, and supports incremental adoption, which would allow the client to add features as they are needed later on.[55] This also made it easier to learn and use for the students, helped by having a large following which means there is also easily available documentation and examples.

Apollo Client is a comprehensive Library that enables the user to manage both local and remote data with GraphQL. Much like Apollo Server, Apollo Client is easy to use, incrementally adoptable and universally compatible.[56] In this project, Apollo Client was primarily used to make and send queries and mutations to the Backend. A client is made with a HTTP link of the Backend and the access token, and this client is then used to send operations with schema documents generated by GraphQL CodeGen. The Vue Apollo library was used for Vue support.

3.1.10 Amazon Web Services

Amazon Web Services (AWS) is a comprehensive cloud platform, with a huge variety of services.[57] For this project, the two services used was hosting a relational database, and a Git repository. Both of these were provided with limited accessibility from the client.

A distributed version control system was necessary. Here, AWS CodeCommit was used. The main reason AWS was used is because the project is for a private company, and the product source code is owned by them. If the students were to create something open source, the natural choice would have been GitHub, which they have much more prior experience with, and is easier and more accessible to set up. Setup with the AWS Command Line Interface was required to get access to the repositories, and the browser repositories were only accessible through specific links, but beyond that, it in practice worked like any other Git repository.

In addition, the team was given access to an Amazon RDS MySQL database. A local MySQL database instance was later set up as a Docker container, but since the team was already using the AWS database, this was not used much.

3.1.11 Test data

The students were given access to a user on Aurora with three sites, or buildings, connected to it. These sites each had one or more sensors attached, which gave live updated data. This data was from real sensors at real sites, but was anonymized and given generic names so the team could use them without breaching customer privacy. Data from all these sites and sensors was extracted as a .csv file. This data file contained every data point several years back in time, with time ranges and intervals varying from sensor to sensor. The dates were shifted a year ahead, so that there would be continuous datapoints after the time when the data was first extracted.

The various information that the team had access to on Aurora about the test user, as well as each site and sensor, was used to fill in the relational database with test data. You can see the relations between the different tables in figure 12.

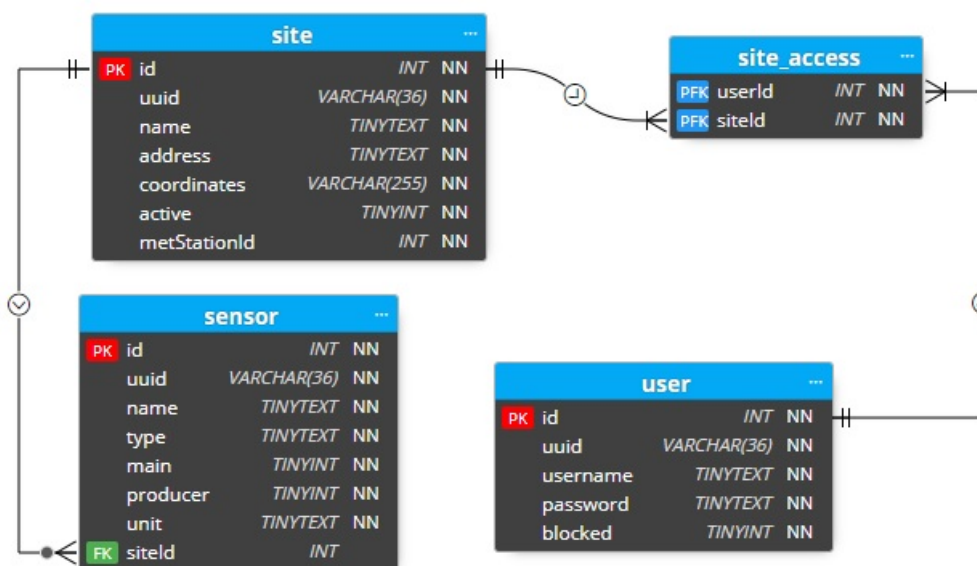


Figure 12: Table relations model for the database storing user data.

The data is polled from each sensor with several fields for each record. For every datapoint, start time, stop time, record time, field type, measurement source, sensor uuid, integrated (whether sensor data is aggregated or not) and measurement value is stored. Among these, only record time, measurement source, sensor uuid, field type and measurement value is sent to Frontend via the API. The sensor uuid is used to connect this data to the correct sensors from the user data database.

3.2 Method & process

The project was divided into three main parts; process, development and documentation. This is illustrated in the second version of the team's Gantt diagram, see appendix A.2. This project plan was followed relatively closely, but with some adjustments made along the way.

3.2.1 Development process

January The project started on the 10. of January with a start-up lecture and a group meeting where the team started planning the work process. During the course of January a pre-project plan was made and a start-up meeting was held with the client and the project supervisor. A first draft progress plan was made, as well as a first draft Wireframe. A second meeting was held with the client, where the team was introduced to the technology Stack the client use, and finished the project contract. The group also started reading up on the relevant technology.

February In February, as well as much of March, the main focus was the parallel course, Systems Engineering, which took up the majority of the team's time. During the course of February, the team worked on getting familiar with the technology that they were going to use. This was generally done by setting up practice projects with for example TypeORM, GraphQL and Nuxt.

March In the start of March, the team had a workshop where one of contacts from the client went over how they implemented the API for the relational database. Based on this, the team proceeded to make a first draft API based on what tables and functionality would be relevant for the project. Work was continued on this throughout March, as well as setting up and making an early draft for the Frontend Vue/Nuxt application.

Towards the end of March the team had another workshop. The main focus of this workshop was getting to know InfluxDB, and how this was integrated with the rest of the technology. The team was also shown some examples of how they could use ApolloClient and GraphQL CodeGen to do queries from Frontend. In late March the team also had to make a poster and hold a presentation about the project, so some time was spent on this as well.

April During April, the system development got started in full. API functionality for Influx was implemented, and a MVP of the Backend was finished. With this, development of the Frontend also accelerated. The majority of core functionality was implemented, such as logging in, querying the relational database for user data and the time series database for sensor data, sorting and filtering sensors based on site and type, presenting selected sensor data in a presentation overlay and visualizing the sensor data with ChartJS.

A Latex project was set up for the main report, and the team started writing notes and bullet points on theory, technology and method. The group also spent some time reading previous bachelor theses for inspiration on report structure and contents. A lot of the current code was also documented with JSDoc, both in Frontend and Backend.

May At the start of may the full focus of the students was turned to writing the bachelor thesis. A main report template had been created during April that was expanded upon, and after a meeting with the bachelor advisor revised parts already written. The thesis was nearing completion a week before the due date and allowed the students to read and assess the text for small mistakes and alterations that had to be done before the final due date. At the end the thesis, along with attachments, was delivered before the due date and the students set up a final presentation for the client, the advisor and the external sensor.

3.2.2 Development method

An iterative development process was used in this project. At the start of the project, the group considered implementing Scrum or Kanban after the end of the Systems Engineering course, but it was decided that this would be of little value with so few project members, little time and experience to organize this, and no dedicated Scrum master. There were, however, used some aspects of Agile development, such as working in iterations, and having some part of the system working after each iteration, so that this could be presented and discussed with the client. This also helped attain a smoother and more educational development process.

At the start of the project, the product had very few clear functional or non-functional requirements, with most of the details being left up to the team. This, combined with the fact that there was some uncertainty around some of the technology used, lead to the team focusing on learning the technology first. While learning the relevant technology and setting up various test projects, viable solutions for the finished product were found and implemented iteratively.

The vast majority of the project work was done online. This was because of several reasons, for example the fact that the team members were not very familiar with each other, that they all primarily used desktop computers for work and had gotten used to working from home during covid 19, and the fact that the client offices were over an hour of travel away. In addition to the Git repositories provided by the client, the team also used two other platforms for cooperation. Teams was used as store process documents, as well as holding digital meetings, while a Discord server was used for daily communication, short notes and voice chat.

3.2.3 Division of Labor and Roles

The three students on team were not familiar with each other beyond a surface level before the start of the project, partially because of the restricted course of study under covid19. This meant that some time was spent getting familiar with each other at the start of the project. This influenced the division of roles, and made it more natural to not select clear roles. Rather than this, temporary roles such meeting leader, meeting secretary and tester were assigned or claimed as needed.

Throughout the project, the group has aimed for an even distribution of working tasks with the option to work extra on their own initiative. When working on tasks, the team communicated in a voice call through discord to keep updated on the progression. In the beginning of the project, the team spent their time on researching the different technologies as well as how to set up the backend. After the MySQL database was up and in working order, the team divided into groups with one Frontend developer and two Backend developers. In backend, the focus was divided into MySQL and InfluxDB. When the MySQL section of the backend was finished, the developer switched to frontend to help with development. The work done on the frontend was a mix of individual coding on different parts of the code, and pair programming with screen sharing. After the time series database was working, the developer started working mainly on the bachelor report, following the Gantt progression plan. Towards the end of the development when most of the code was done, the team members started writing on the report.

4 Results

In this section, the results of the project will be described. The main focus of this project was software development and the design of the application, so that is where most of the focus will be put. In addition to this, there are also the administrative results of the project process.

4.1 Software development results

In the beginning of the project, a couple of functional requirements were set. However, most of the requirements were added during the development process through meetings with the client. Because the task was to create an MVP for the client, the focus has been on implementing core features rather than advanced functionality, as well as writing reusable code for further development.

4.1.1 Functional requirements

In table 1, you can see the functional requirements of the project. These functional requirements were made by the team in discussion with the client. Because the client did not have many clear requirements before the first meeting, most of them were made during the discussion as well as throughout the development. As the table shows, most of the requirements were fulfilled, however there are some requirements that the team feels should have been implemented in a different or better way.

Functional requirement	Achieved	Comment
Log in function	✓	User data stored in LocalStorage
Log out	✓	
List all available sensors	✓	
Add sensors to presentation	✓	
Start presentation with selected sensors	✓	
Change presentation slide change timer	✓	
Present relevant data from selected sensors in presentation mode	✓	
Customizable charts and data for presentation	✗	No time to find and implement solutions
Auto-update graphs during presentation	✓	Occasional render problems updating charts
Filter by building	✓	
Filter by sensor type	✓	
Customizable color themes	✓	
CO ₂ equivalents for graph	✓	Very basic with dummy data, by recommendation from the client
Electricity cost graph	✓	Very basic with dummy data, by recommendation from the client

Table 1: The functional requirements of the system, and which requirements were achieved or not.

4.1.2 Non-functional requirements

The project also had some non-functional requirements, as seen in the vision document (appendix D). These were:

Useability The system has been designed to follow the core criteria of WCAG 1.2.[58] This was mainly done by the extensive use of Vuetify, where all components support functionalities required by WCAG.[59] The application has been designed to be simple, intuitive and easy to use, even for users with low IT competence, or with low familiarity with the connected sensor systems. Vue/Nuxt apps have wide browser support, and the application has been tested to run on Google Chrome, Mozilla Firefox, Safari and Microsoft Edge.

Documentation, maintenance & further development The code base has been thoroughly documented, with descriptions in the form of JSDoc for all relevant methods, functions and utilities, both in Frontend and Backend. See figure 13 for an example of how JSDoc is used in the code. In addition to this, a system documentation document has been written with an overview of the project, as well as a guide for setup and testing. The system must be available and ready for further development by the client after the end of the project. This is achieved by having the entire project hosted on their private repositories, so that it is easily accessible. In addition to being thoroughly documented, the system is made as loosely coupled and functional as possible. This should make maintenance and further development easier.

```
/**
 * Custom fuction for creating a signed access token for a user
 *
 * @param user the {@link User} object to get a token
 * @returns a signed access token made from the user payload,
 * the access token secret and token parameters (here: expiration time)
 */
export const createAccessToken = (user: User): string => {
  const payload: AccessTokenPayload = {
    |   userId: user.id,
    |
  }

  return sign(payload, process.env.ACCESS_TOKEN_SECRET!, {
    |   expiresIn: '365d',
    | })
}
}
```

```
(alias) createAccessToken(user: User): string
import createAccessToken

Custom fuction for creating a signed access token for a user
@param user — the User object to get a token
@returns
a signed access token made from the user payload, the access token secret and
token parameters (here: expiration time)
createAccessToken(user),
```

Figure 13: Top: the implementation of a function called createAccessToken with the accompanying JSDoc. Bottom: a usage of the same function with its description showing

Privacy & data security The system is set up for protection with a login functionality enabled by the use of an access token. However, this safety measure is not fully implemented. While the login functionality is relatively complete, currently the access token and other user data is just saved in a store in LocalStorage, which is a security risk. In addition to this, GraphQL/TypeGraphQL's authorization functionality is not fully utilized in the Backend. While all queries and mutations except the login operation require authorization with a valid access token, there is currently no differentiation between distinct tokens. This means that as long as a user have a valid access token, they have access to every single mutation and query. I addition to this, they can access every site and sensor, as well as other users, regardless of if those sites and sensors are connected to the user. One security measure that was implemented, is a simple method that checks the queries to the time series database for regex injections, and removes unwanted characters.

4.1.3 API

At the beginning of the project there was some uncertainty about whether the team were actually supposed to develop an API or any Backend components at all. The alternative was just using a copy of the client's existing API. In the end, the team decided to build a standalone Backend. Part of the reason for this was that the existing API was very comprehensive, with a lot of functionality that would not be relevant to this project. Another important reason was for the team wanted to get a better understanding of the full system. Thirdly was the fact that the Frontend was planned as a relatively simple application, so the team expected to have time to work on the Backend.

This resulted in making a simple API with an Apollo server implementation serving resolver methods used to make queries and mutations, both to a MySQL relational database used for user meta data, as well as the InfluxDB time series database that holds the actual sensor data. The relations model for the user meta data can be seen in figure 12. Test instances of both these databases were set up, filled with test data, and connected to the API so that they can be accessed in the web application.

4.1.4 Web application

As mentioned in greater detail in chapter 3.1, a single page web application was developed using Vue2 and Nuxt2. The app is designed with a standard Nuxt project structure, using a setup of Vue files like the following: Layouts → Pages → Components. See figure 14 for the structure of this project. The application uses Apollo Client and GraphQL CodeGen for making queries to Backend, ChartJS for making charts, Pinia for state management and Vuetify for design. The design of the application will be presented in greater detail in chapter 4.2.1.

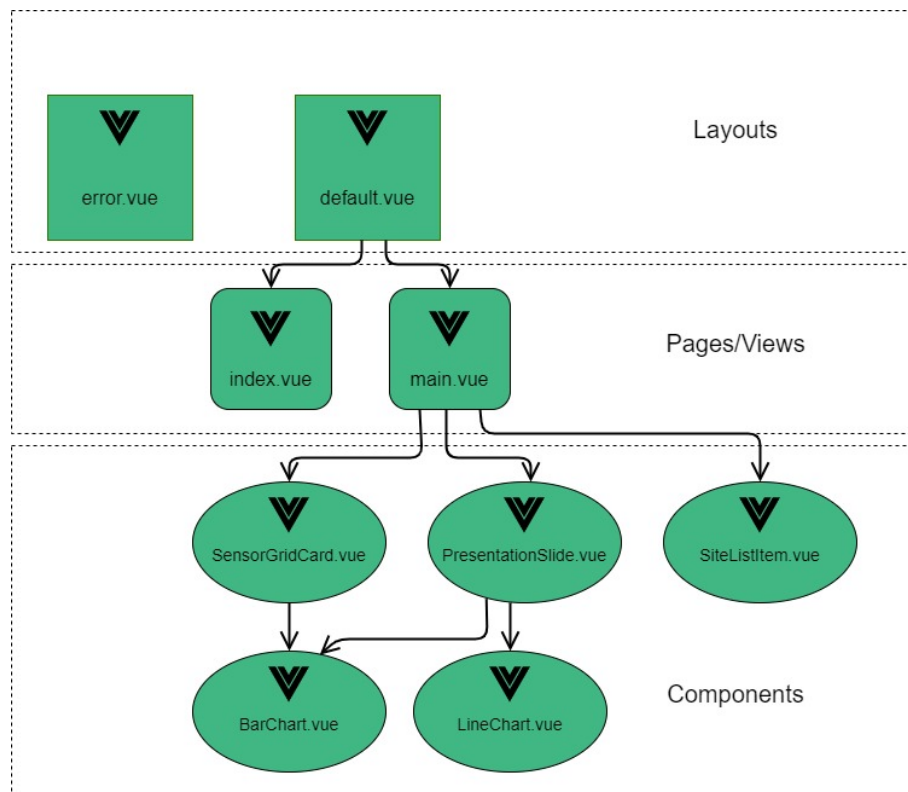


Figure 14: The web application file structure.

4.2 Scientific results

4.2.1 Application design

The design of the Web application has been through several iterations of work, where each iteration has resulted in a new version. This was all based on a Wireframe model that was created at the start of the project, and one can see that the final product is quite similar to the initial Wireframe in structure. The Wireframe can be found in appendix C. The final application is a very simple app, with only two views; a simple login page and the main page. See figure 15 for the login page. The main page have a regular mode and a presentation mode. See figure 16 for the final layout of the main view. Older iterations of the main view can be found in appendix B.1 and B.2.

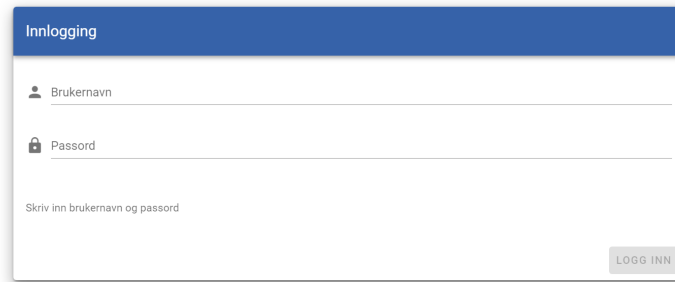


Figure 15: Login page.

The application has a left hand menu where you can filter the displayed sensors based on site or sensor type. This is also where you find the button to start the presentation mode, as well as a slider that adjusts the presentation interval. This menu can be opened and closed from the top bar. See left side of figure 16. On the right side is a settings menu, where currently the only two functionalities are selecting a color theme and logging out. See figure 17. The user can pick one of several predetermined color options, as well as be able to pick their own custom color from either a visual color picker, or by input of a hex code or rgb values. This color changes the primary color, which is used on the graphs, buttons and other parts of the application.

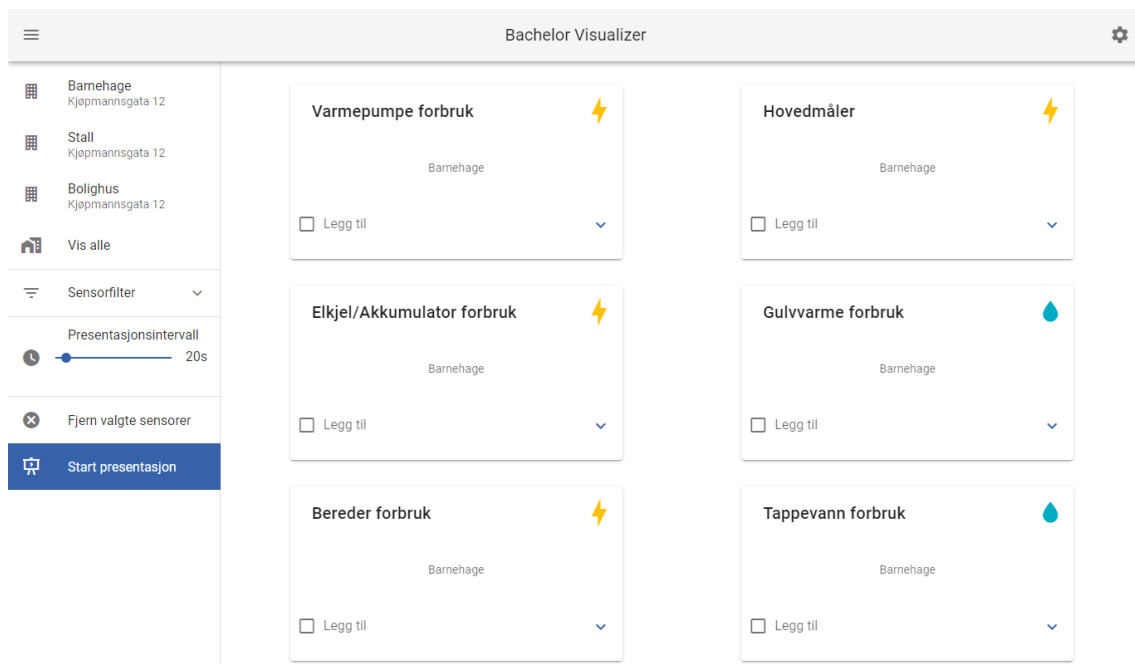


Figure 16: Final iteration of the main view of the application.

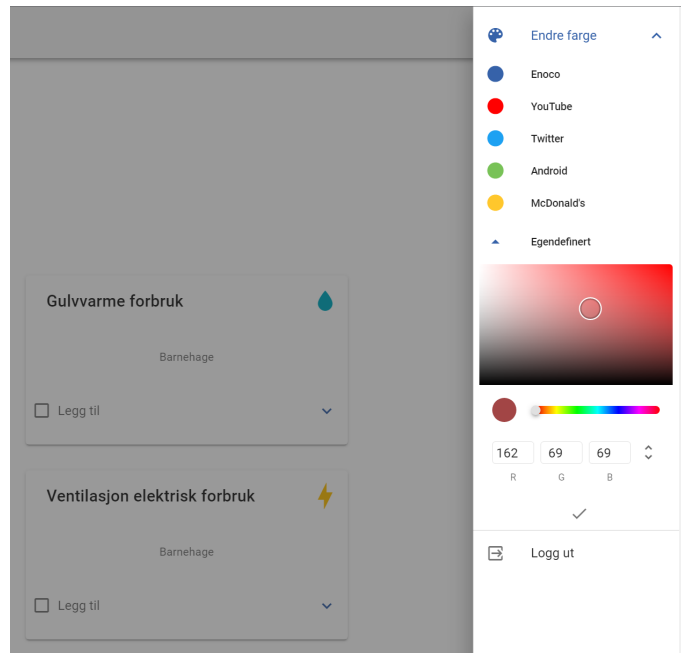


Figure 17: The settings menu on the right side of the main page.

At the center of the application is a grid view of all the sensors based on the current filtering. A drop down can be opened on each sensor item to view a graph of energy usage for the last 24 hours. There is also a checkbox that adds the sensor to the list of sensors to be displayed in presentation mode. When you start the presentation mode, an overlay shows up which rotates through the selected sensors. In each presentation slide, four charts display various data from each sensor. See figure 18. Currently, chart one displays electric prices along with energy usage for the last 180 days. A second chart displays the same, except it has the calculated energy costs per week instead of prices. A third chart displays the energy usage per day for the last week. The final chart displays the accumulated energy usage along with the accumulated CO₂ emission equivalents from the energy usage. Both the energy prices as well as the CO₂ equivalent constants are currently just hard-coded constants rather than being fetched from a database.

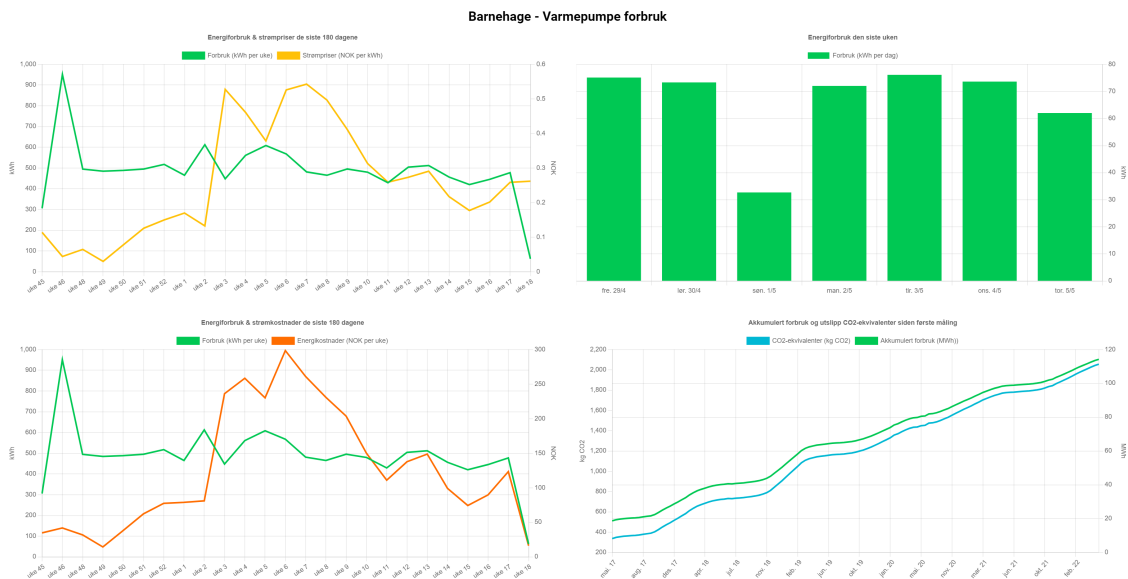


Figure 18: A single slide in the presentation view.

4.2.2 User testing

No user testing was done for this project. This was primarily due to a lack of time, but also because setting up the system requires extensive setup and access to multiple databases. This meant that asking remote users to test the system was very impractical, if not impossible. Early on in the project, it was suggested by the client that they would put the team in contact with some of their customers to provide feedback and user tests, but this was not done.

4.3 Administrative results

4.3.1 Working methodology

As mentioned in chapter 3.2.1, a relatively loose iterative development model was used. This meant the team was able to be very flexible, both in terms of what was worked on, as well as working hours. At the beginning of the project, when working on the process documents and planning, the team worked together in person or on voice chat, since this was work that required a lot of discussion. During the development phase and the start of the documentation phase, more work was done on an individual level, while still with having regular discussions and staying updated. Towards the end of the documentation phase, the work went back to being almost entirely done cooperatively.

4.3.2 Progress plan

A progress plan in the form of a Gantt chart, with the various tasks that needed to be completed. These tasks were, as previously mentioned, split into process work, development and documentation. Two versions were made, one at the beginning, and one in the middle of the project. In appendix A.1 one can see the initial progress plan made by the team in January. A second, updated progress plan was made in March, which can be found in appendix A.2. This second plan ended up being a relatively good representation of the actual progress.

The main difference between the initial plan and the updated plan was that a lot of tasks were compressed towards the last two months. A big part of the reason for this was the Systems Engineering course, which took up a significant amount of time and focus during February and March. Another reason was that certain requirements and technologies remained unclear much longer than anticipated. One example here is the InfluxDB database, which the team was unclear on how to implement and use until late March. The team also initially planned to work on the main report concurrently with the development. The thought behind this was to be able to research relevant theory and technology while learning the stack, and then write theory while this was fresh in mind, but the team was not able to find time to work on the report during this period.

4.3.3 Time sheets

The team members kept track of work hours in an Excel spreadsheet kept in Teams. This kept track of the hours for each member, as well as the total count. These time sheets were also useful for other purposes, such as remembering and keeping track of what exactly were being worked on at what time when writing the main report, or when writing monthly status reports to the team's supervisor.

5 Discussions

In this chapter the results presented in the previous chapter will be discussed. As with the previous chapter, the main focus will be on the software development and the design of the application.

5.1 Software development results

5.1.1 Functional requirements

As seen in table 1, most of the functional requirements set were completed. Many of these were straightforward utilities such as listing and filtering the available sensors, adding selected sensors to the presentation list, or changing the the presentation time intervals. These requirements were implemented without much issues. The functionality of logging in and logging out was also implemented, but it is worth noting here that currently, the user data, including the access token, is saved in a state management store that uses LocalStorage for persistence. For security reasons, this is not a good final solution, but it was deemed acceptable as a temporary solution for an MVP.

Keeping the sensor data updated while the presentation is running is a requirement that was technically achieved, but the team is not satisfied with this implementation. Currently, an interval is created in the PresentationSlide component for each sensor. On a set timer, this interval queries for updated sensor data to use in the presentation charts. Since the data only has a resolution of one hour at the smallest, this interval was set at 15 minutes early on, since there was no need to update the chart data often when the sensor data in the database had not changed. With the timer set to this long, the team failed to notice that there were some issues when the interval updated the chart data during the presentation. This would sometimes lead to empty data arrays being passed to the chart, or the charts failing to load entirely. This was not discovered until after the team had more or less stopped development, so there was no time for extensive troubleshooting or implementing a different solution. Part of the problem might be that when the asynchronous operation of querying for new data happens right before a slide change, the process could end up not being finished until the new slide component is loaded, resulting in empty data arrays.

Another potential source of the problems could be ChartJS/VueChartJS. The team had some problems rendering multiple charts on the page at the same time. This was the case both when opening multiple charts in the grid in the main view, as well as with the four charts in the presentation slides. It is suspected that there are some issues with rendering multiple reactive charts at the same time, but no concrete evidence or sources of error were found when these bugs occurred. This, in addition to how the ChartJS implementation got a little messy and overly complicated, has lead the team to think that a different charting tool could have been a better fit for the application.

In addition to using graphs for showing energy usage, the client suggested implementing a way to visualize other data as well. Data for visualizing electricity prices, as well as calculating electricity costs and CO₂ equivalent emissions, was hard coded in as constants. This implementation is crude, but works as a temporary solution for an MVP. In the final product, these would be fetched from external data sources, to get dynamic and up to date values. Another issue with the electricity prices and costs is that not all sensors are for electric energy flows, but also for example waterborne heat. For these sensors, showing electricity prices and calculated costs make little sense, but this was not taken into consideration by either the team or the client until too late in the development process.

This ties into the only functional requirement that was not achieved at all, namely customizable options for the presentation slides. This includes both having dynamic presentation slides depending on sensor type, for example showing different charts for electricity sensors and waterborne sensors, as well as some level of user customization. This requirement was a relatively central part of the planned application, but the team had to focus on creating a working and somewhat polished product rather than adding more functionality.

As mentioned in chapter 4.1.1, very few clear requirements were given at the start of the project, mostly just suggestions related to the technology stack. This gave the team a lot freedom to be creative, but as a result, also made planning and progression more challenging.

5.1.2 Non-functional requirements

As mentioned in chapter 4.1.2, the system was designed in accord with WCAG 1.2. However, this was not something the team had to work actively towards, as much of this functionality was provided out of the box with Vue and Vuetify. The team is for the most part satisfied with how intuitive and responsive the application is. There are, however, some small improvements and additions that would be advised before shipping it as a commercial product, for example a high contrast or color blind mode.

While the system is made as loosely coupled as possible, with functional programming in mind, there are certainly some improvements that could be made here. One clear example here is the method used for querying for sensor data used in the presentation. This method has a lot of different functionality that could potential be split into several, more readable components. The team has taken care to add extra documentation to bigger, more complicated pieces of code, to ensure it is understandable for future developers.

Currently, the system has very bare bones security measures, so these would have to be heavily modified and expanded upon before commercial use. However, this was always going to be the case, because it was not the focus of the project. In addition, the team was encouraged by the client to not spend a lot of time implementing authorization and authentication. Even if this was a bigger focus and the team members had more experience with data security, these systems would likely have needed to be thoroughly reviewed and modified before commercial use regardless.

5.1.3 API

As mentioned in chapter 4.1.3, the team chose to make a simplified API from scratch, tailored to the project's needs. Although the team struggled a bit while setting up and learning how to use the time series database with the rest of the system, this as a whole was definitively an educational experience. The team members previously had little to no experience working on Backend systems with JavaScript or TypeScript, nor with modern tools such as GraphQL and TypeORM, so a lot of valuable experience was gained during the process. Setting up the Backend also gave a better understanding of the system as whole, which made implementing communication to the Backend from the Frontend easier and more intuitive.

On the other hand, the team did end up spending quite a lot of time learning and setting up the Backend. It is likely that the Frontend part of the system, which was the main focus, would have been more refined and gotten more complete features if minimal time had been spent working on the API. Despite this, the team is happy with the decision to develop their own API, as this allowed them to learn a lot of new technologies and Frameworks, as well as getting a more complete understanding of the system.

5.2 Scientific results

5.2.1 Application design

When choosing the design of the app, the plan was to draw inspiration from Eurora, but in some ways make simplified version of it. One of the problems the team found with Eurora was that it has a lot of pages for sites and sensors with similar or unclear functionality, which can make the application somewhat confusing to navigate. This was part of the reason why the teams focused heavily on making their product as simple as possible. The main page has an overview of all sensors that the current user has access to. These sensors are represented with the card elements as shown on the left in figure 19. As of now, the cards can be opened to display energy usage for the past 24 hours. However, this can easily be changed in the code to give a different time frame by changing one line of code. This could have been a customizable option within the app instead, but because of a lack of time, as well as focus to keep things simple, this was not implemented.

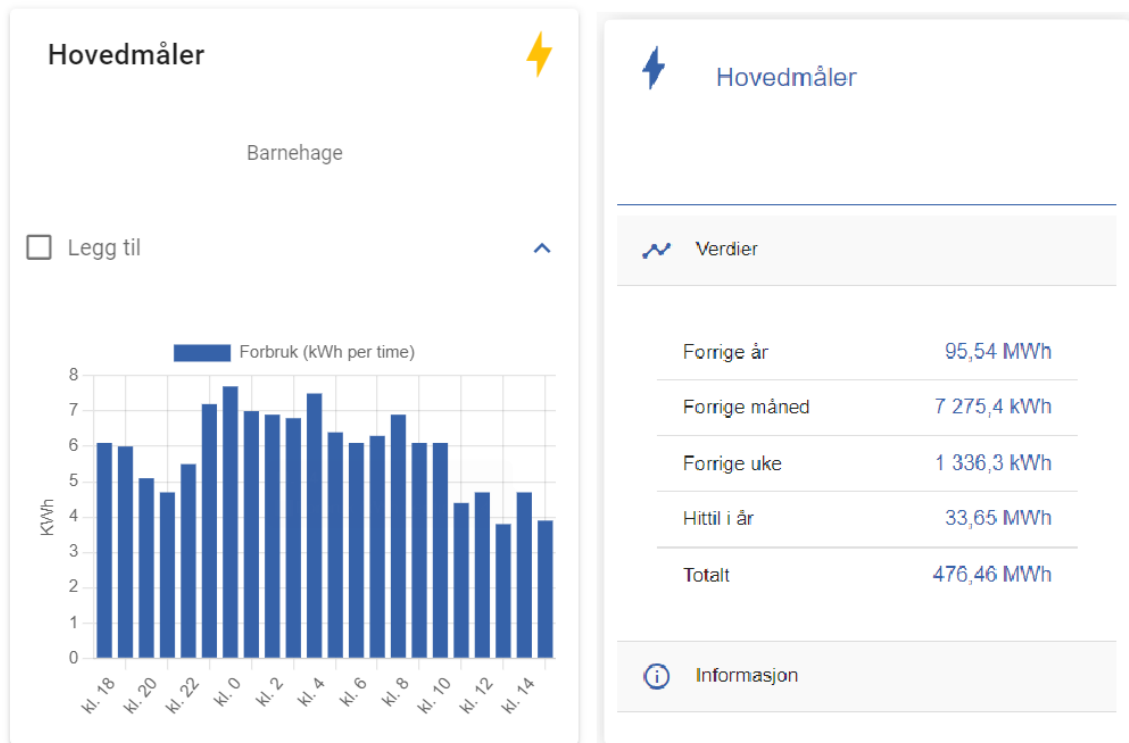


Figure 19: Left: opened sensor card from the main view of the application. Right: sensor card from Eurora.

As mentioned in chapter 4.1.2, a big priority of the system was to keep the application intuitive and accessible, even for users that are either not that familiar with the systems the application manages, or with IT in general. One of the core ways the team tried to achieve this was by limiting excessive options, and keeping the layout as simple as possible. With the current structure of just the main page and a login page, it is very easy to navigate, and all options are always very few clicks away.

This was a philosophy that was core to the project; to keep things simple. This was applied both to functionality and design. Currently, the app is mostly white with a selectable primary color, see figures 16 and 17. Various tests were done adding colors or color accents to different elements, but in the end the team preferred the simple, clean white design. For an example of these tests, see appendix B.2.

Overall, the team was quite satisfied with the general appearance of the application. The team used Vuetify components for every visual aspect of the application except the charts, which lead to a clean, consistent look. Vuetify components such as the top bar or set side bars are also easy to work with, and new buttons or utilities can easily be added at a later date.

5.2.2 Testing

As mentioned in chapter 4.2.2, it was planned to have one or more meetings with the client's customers, which are the would-be users of the application. This would either have been early meetings, possibly with a Wireframe to inquire about wishes for functional requirements, or later on to get feedback on a working MVP. The team decided to wait until they had a concrete product to show for these meetings, but due to various reasons, this was ready later than expected. As a result, a judgement call was made by the client that these meetings should be scrapped, and instead they would take on the role as customers. During a meeting with the client, the team simulated a user test with the client to get some input. While this lead to some feedback and suggestions for functionality, it did not lead to any sort of extensive user tests. This was mainly because of a lack of time.

In addition to lack of time, if user tests were to be conducted externally, the machine performing the user test having to set up the entire system environment. This included setting up a Docker container for the databases, fill them with correct test data, as well as install and run both the Frontend and Backend applications. Because of this, any wider user system testing was not an option.

When it comes to the Backend in particular, testing was not highly prioritized. This was because the Backend developed by the students was not the focus of the assignment and was only meant for development and learning purposes. Since this code in all likelihood would not be used after the end of the project, spending too much time and resources testing and optimizing it was judged to be a waste of time.

5.3 Administrative results

5.3.1 Working methodology

As mentioned in chapters 3.2.2 and 3.2.3, the students were put together on the team at random and were not very familiar with each other before the start of the project. This influenced the working methods and the overall process, especially since none of the team members are naturally inclined to take on initiative or a leader role. This meant that the structure suffered a bit, especially in the beginning of the project. However, this never turned into a major issue, as the team was able to take initiative and work together as needed. Working in a team of three also provided a lot of benefits. The team was able to add different perspectives to problems, while the team size was not so big as to make these different perspective cluttered and unhelpful.

When starting on the main report, a lot of work was done on an individual basis, while other members were working on different things. This lead to a lack of structure and consistency between different sections. This was because different team members worked on different sections independently of each others. When the full focus was shifted to the report, the team was able to discuss and plan more of the report, and a more coherent structure was formed. In light of this, it would have been beneficial to more thoroughly discuss and plan the report out before starting the writing.

5.3.2 Progress

As previously mentioned, the progress of the project was slower than the team would have liked. This was partially because the team members were unfamiliar with each other, but the biggest reason was the Systems Engineering course. This course was much more comprehensive and laborious than expected, with various different submission, project work and an exam that all took a considerable amount of time. This meant that the team was not able to fully focus on the Bachelor project until the end of March. While the Systems Engineering course was taken into account when making the first progress plan, the impact it would have on the project work was definitively underestimated. In addition to this, some of the technology and functional requirements were unclear for much longer than it should have been, notably the usage of the time series database. As the core of the application revolved along displaying the sensor data from the time series database, a lot of development and testing was slowed down or delayed until this was implemented.

6 Conclusions & further development

The problem to be solved in this project was formulated as following:

How can the client's customers display the energy usage and other related data from their sites in a presentable way, on big screens in offices or lobbies, with a web application, and how can this application be made accessible, intuitive and user-friendly.

This report has focused on presenting and discussing the practical solutions and choices that were made when solving this problem.

6.1 Conclusions

Even though there are certain parts of the technical implementation of the application that the team is not satisfied with, and some additional functionality that ideally should be added, the team is quite happy with the finished result. In regards to the formulated problem, the team is confident that the broad goals were largely achieved. The application has a clean design, with a simple and intuitive interface, while being able to present relevant data in a visually interesting and readable manner. The product is clearly distinct in usage from Eurora, most obviously with the presentation mode, but also the fact that all the sites and sensors are easily accessible in one page.

When it comes to the development process, there are some things that could have been handled differently. The main source for improvement here is clarifying goals and requirements, as well as planning ahead. This of course includes the team, that should have made more concrete goals and plans early on in the project, which would have helped accelerating the process, but also from the client, as well as NTNU regarding the project process. The requirements and wishes from the client were very loosely defined, which made planning and making goals more difficult. This caused the students to spend longer before gaining sufficient knowledge on their technology Stack. At the end of the project however, the team members were left with a lot of valuable experiences, not only regarding all the new technologies they learned, but also lessons about how to structure and plan out a larger project.

6.2 Further work

The product made during this project serves the purpose as an MVP for the client, and as such is not expected to be ready for commercial use. The biggest missing requirement here is a more complete system for for authorization and authentication, to maintain data security and customer privacy.

A missing feature that the team would have liked to do more with is customization of the presentation. Currently, the presentation mode only has a preset of charts, but it was always the team's intentions to have some light level of customization here. This is therefore high on the list of functionality to be added.

When it comes to functionality that is implemented, but should be improved, the team would have liked to change how the sensor data is updated during the presentation mode. As discussed in chapter 5.1.1, the current solution here does not work perfectly, so a different solution should be found.

References

- (1) Enoco, *Om oss*, <https://www.enoco.no/om-oss/>.
- (2) Enoco, *Eurora*, <https://www.enoco.no/eurora/>.
- (3) M. Dancuk, *What Is A Database?*, <https://phoenixnap.com/kb/what-is-a-database>, 2021.
- (4) Oracle, *What Is a Database?*, <https://www.oracle.com/database/what-is-database/>.
- (5) Oracle, *What Is a Relational Database (RDBMS)?*, <https://www.oracle.com/database/what-is-a-relational-database/>.
- (6) M. Heiderich, E. A. V. Nava, G. Heyes and D. Lindsay, in *Web Application Obfuscation*, Syngress, 2011, ch. 7, pp. 177–197.
- (7) Amazon Web Services, *What is a Relational Database?*, <https://aws.amazon.com/relational-database/>.
- (8) R. Peterson, *Relational Data Model in DBMS*, <https://www.guru99.com/relational-data-model-dbms.html>, 2022.
- (9) A. Kulkarni and R. Booz, *What is time-series data and why do I need a time-series database?*, <https://www.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/>, 2020.
- (10) InfluxData, *Time Series Database (TSDB) Guide*, <https://www.influxdata.com/time-series-database/>.
- (11) M. DeSea, *Intro to Time Series Databases & Data*, <https://www.youtube.com/watch?v=OoCsY8odmpM>, 2017.
- (12) P. Dybka, *ORMs Under the Hood*, <https://vertabelo.com/blog/orms-under-the-hood/>, 2015.
- (13) M. Liang, *Understanding Object-Relational Mapping*, <https://www.altexsoft.com/blog/object-relational-mapping/>, 2021.
- (14) A. Spittel, *What is a Web Framework, and Why Should I use one?*, <https://welearncode.com/what-are-frontend-frameworks/>, 2018.
- (15) Microsoft, *The MVVM Pattern*, [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)), 2012.
- (16) Techopedia, *What is Data Binding?*, <https://www.techopedia.com/definition/15652/data-binding>, 2012.
- (17) Vue, *Comparison with Other Frameworks*, <https://v2.vuejs.org/v2/guide/comparison.html>.
- (18) S. Martin, *Angular vs React vs Vue.js: Which is the Best Choice for 2022?*, <https://javascript.plainenglish.io/angular-vs-react-vs-vue-js-which-is-the-best-choice-for-2022-5ef83f2257ab>, 2022.
- (19) Angular, *What is Angular?*, <https://angular.io/guide/what-is-angular>.
- (20) React, *Getting Started*, <https://reactjs.org/docs/getting-started.html>.
- (21) Vue, *Introduction*, <https://v2.vuejs.org/v2/guide/>.
- (22) S. Grinaker, *Front-end frameworks: What is important right now?*, <https://enonic.com/blog/front-end-frameworks-what-is-important>, 2019.
- (23) S. Hogg, *Software Containers: Used More Frequently than Most Realize*, <https://www.networkworld.com/article/2226996/software-containers-used-more-frequently-than-most-realize.html>, 2014.
- (24) IBM Cloud Education, *What is Docker?*, <https://www.ibm.com/cloud/learn/docker>, 2021.
- (25) Docker, *Docker overview*, <https://docs.docker.com/get-started/overview/>.
- (26) Interaction Design Foundation, *What is Color Theory?*, <https://www.interaction-design.org/literature/topics/color-theory>.
- (27) Google, *Material Design Guidelines*, <https://material.io/design/guidelines-overview>.
- (28) Centre for Excellence in Universal Design, *Definition and overview*, <https://universaldesign.ie/what-is-universal-design/definition-and-overview/>, 2020.
- (29) Farm Credit Administration IT Section, *Essential Practices for Information Technology Based on Industry Standards and FFIEC Examination Guidance*, tech. rep., 2007.

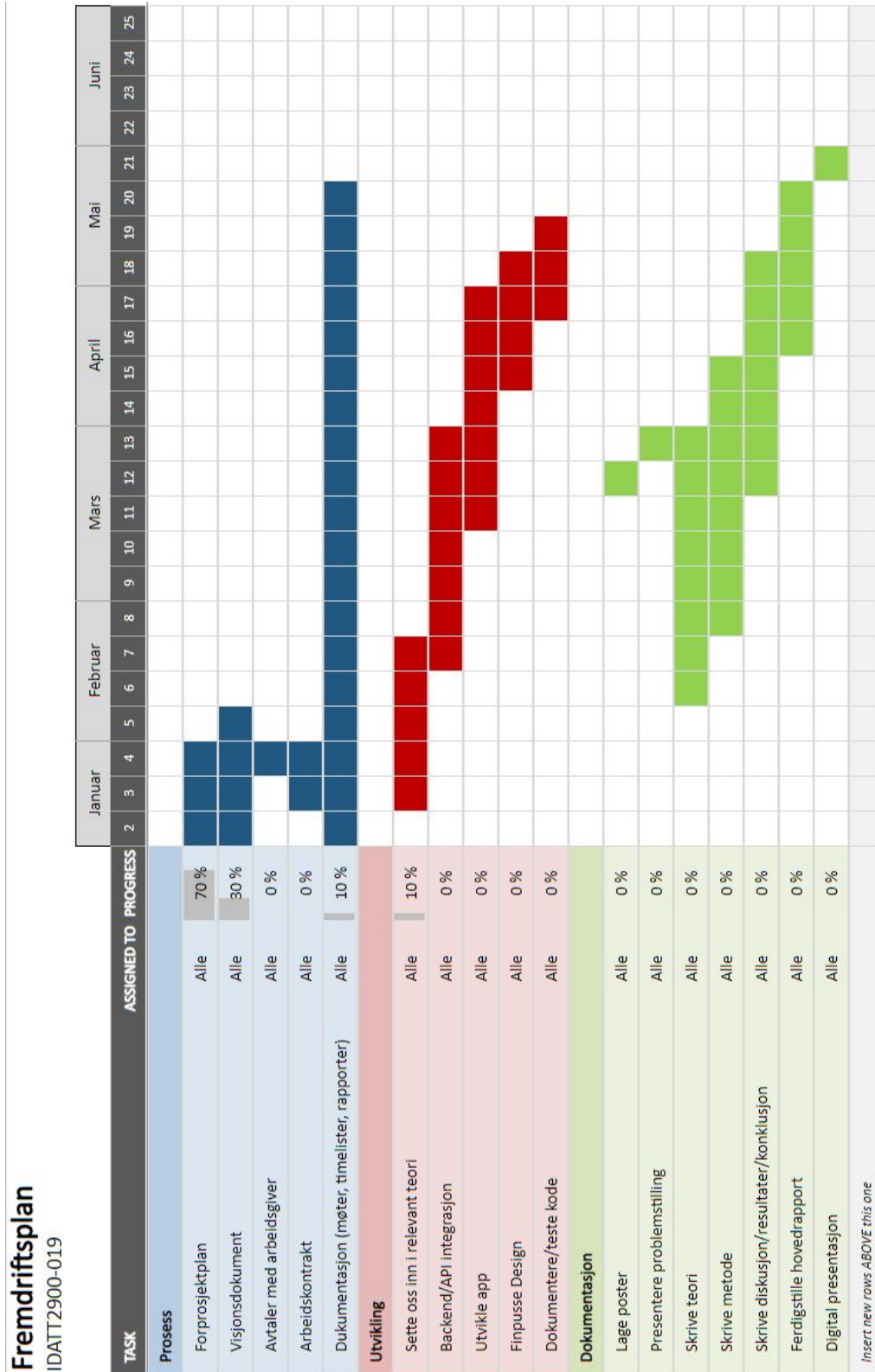
- (30) B. Shiklo, *8 Software Development Models Organized in Charts and Explained*, <https://www.sensoft.com/blog/software-development-models>, 2019.
- (31) S. Chacon and B. Straub, in *Pro Git*, 2009, pp. 8–11.
- (32) RhodeCode, *Version Control Systems Popularity in 2016*, <https://rhodecode.com/insights/version-control-systems-2016>, 2016.
- (33) Git, *Staging Area*, <https://git-scm.com/about/staging-area>.
- (34) S. Chacon and B. Straub, in *Pro Git*, 2009, pp. 12–16.
- (35) GitHub, *Features*, <https://github.com/features>.
- (36) GitLab, *Features*, <https://about.gitlab.com/features/>.
- (37) Amazon Web Services, *AWS CodeCommit Features*, <https://aws.amazon.com/codecommit/features/>.
- (38) S. Krause, *Results for js web frameworks benchmark – round 8*, <https://stefankrause.net/js-frameworks-benchmark8/table.html>, 2018.
- (39) H. Shah, *React vs Vue – The CTOs guide to choosing the right framework*, <https://www.simform.com/blog/react-vs-vue/>, 2022.
- (40) Nuxt, *What is Nuxt?*, <https://v3.nuxtjs.org/guide/concepts/introduction/>, 2022.
- (41) O. Omole, *Nuxt.js: a Minimalist Framework for Creating Universal Vue.js Apps*, <https://www.sitepoint.com/nuxt-js-universal-vue-js/>, 2019.
- (42) Vuetify, *Why you should be using Vuetify*, <https://vuetifyjs.com/en/introduction/why-vuetify/>, 2022.
- (43) United States Environmental Protection Agency, *What is Section 508?*, <https://www.epa.gov/accessibility/what-section-508>.
- (44) ChartJS, *Getting Started*, <https://www.chartjs.org/docs/latest/getting-started/>, 2022.
- (45) VueChartJS, *Getting Started*, <https://vue-chartjs.org/guide/>, 2022.
- (46) Vuex, *What is Vuex?*, <https://vuex.vuejs.org/>.
- (47) Pinia, *Introduction*, <https://pinia.vuejs.org/introduction.html>.
- (48) Prisma, *Prisma vs TypeORM*, <https://www.prisma.io/docs/concepts/more/comparisons/prisma-and-typeorm>.
- (49) TypeORM, *Getting Started*, <https://typeorm.io/>.
- (50) DB-Engines, *DB-Engines ranking of time Series DBMS*, <https://db-engines.com/en/ranking/time+series+dbms>, 2022.
- (51) P. Grzesik and D. Mrozek, *Computational Science - ICCS 2020*, 2020, **12141**, 371–383.
- (52) InfluxData, *Compare InfluxDB to Other Time Series Databases*, <https://www.influxdata.com/products/compare/>.
- (53) K. Stemmler, *What is GraphQL? GraphQL introduction*, <https://www.apollographql.com/blog/graphql/basics/what-is-graphql-introduction/>, 2022.
- (54) TypeGraphQL, *Introduction*, <https://typegraphql.com/docs/introduction.html>.
- (55) Apollo GraphQL, *Introduction to Apollo Server*, <https://www.apollographql.com/docs/apollo-server/>.
- (56) Apollo GraphQL, *Introduction to Apollo Client*, <https://www.apollographql.com/docs/react/>.
- (57) Amazon Web Services, *Cloud computing with AWS*, <https://aws.amazon.com/what-is-aws/>.
- (58) World Wide Web Consortium (W3C), *Web Content Accessibility Guidelines (WCAG) 2.1*, <https://www.w3.org/TR/WCAG21/>, 2018.
- (59) Vuetify, *Accessibility (a11y)*, <https://vuetifyjs.com/en/features/accessibility/>, 2022.

Appendix

A Gantt diagram

A.1 Version 1

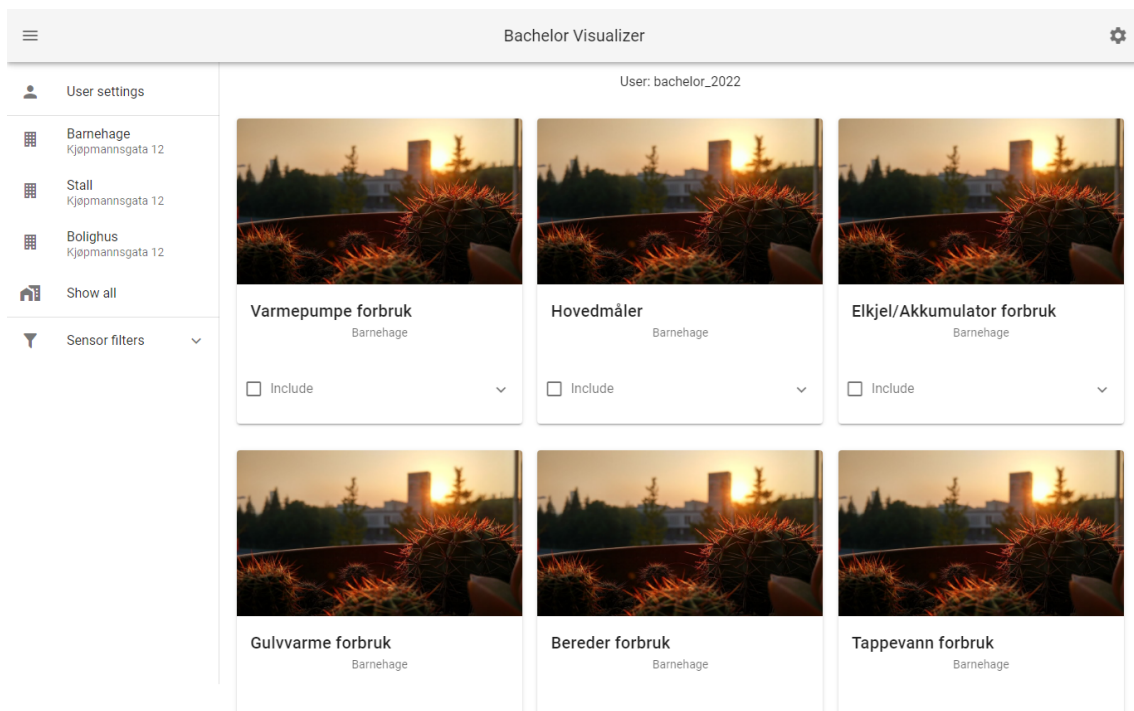
Version number one of the Gantt progress plan. In Norwegian since most of the process documentation was written in Norwegian.



B Main view versions

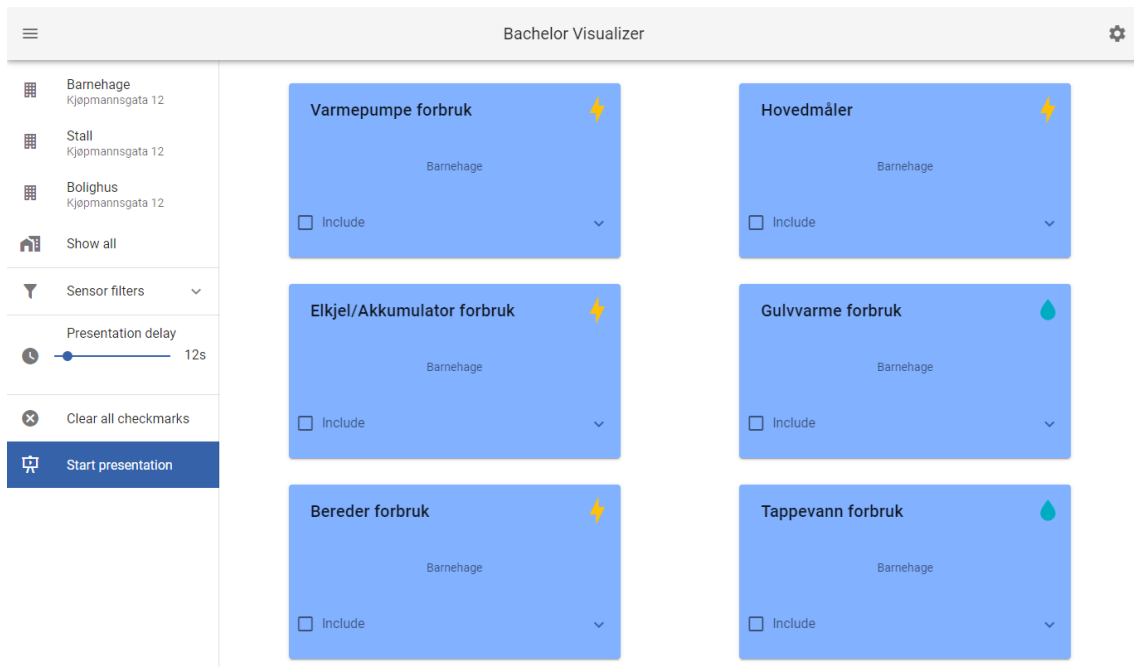
B.1 Version 1

Initial version of the main view, with placeholder content on the sensor cards.



B.2 Version 2

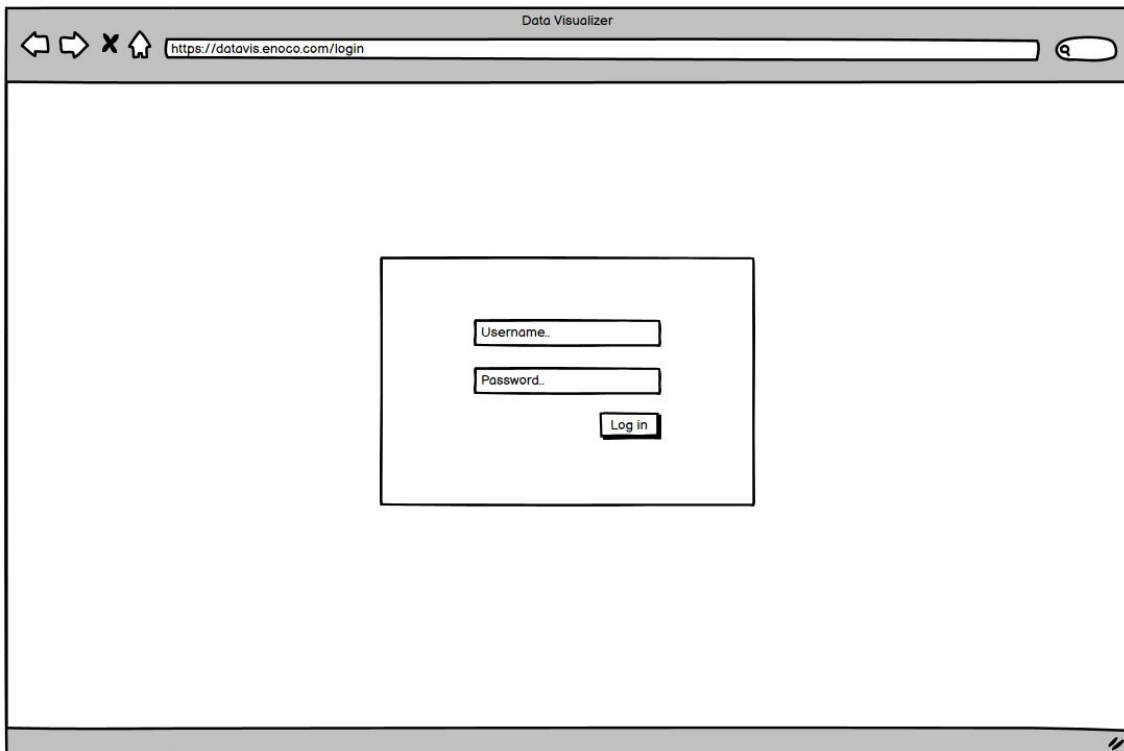
Later version of the main view, similar to the final product, but testing using accented colors as sensor card background.



C Wireframe

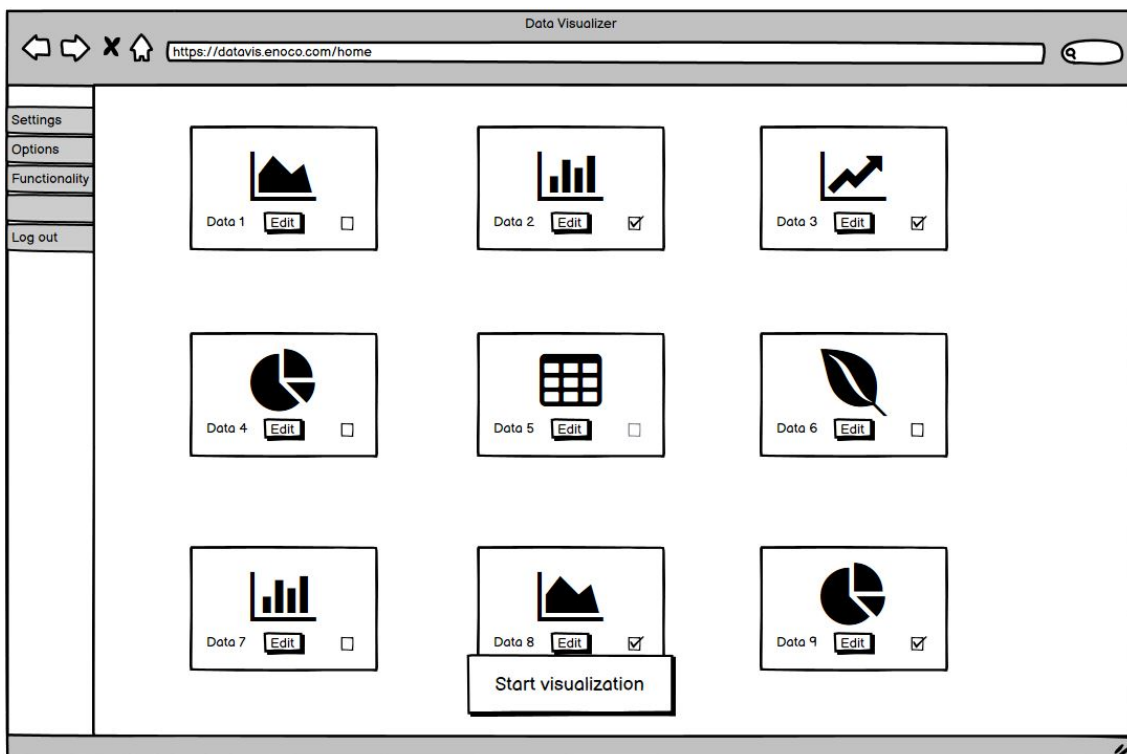
C.1 Login view

The login page in the initial Wireframe model.



C.2 Main view

The main page in the initial wireframe model.



C.3 Presentation view

The presentation view in the initial Wireframe model.



D External process documentation

This is a list of process documents that are not included in the main report, but were included in the project submission.

- Vision document (Visjonsdokument.pdf)
- Requirements documentation (Kravdokumentasjon.pdf)
- System documentation (Systemdokumentasjon.pdf)
- Process documentation (Prosjekthandbok.pdf)
- Project contract (Standardavtale.pdf)
- Frontend repository (/dashboard_web)
- Backend repository (/dashboard_api)
- Sensor data (sensor_data.csv)

