

Casper Fabian Gulbrandsen
Sander Låstad Olsen
Kristian Jegerud
Marthin Gunerius Klækken

Inventory Management System for Cryopreserved Biological Material

Bachelor's project in Department of Computer Science

Supervisor: Tom Røise

May 2021

Casper Fabian Gulbrandsen
Sander Låstad Olsen
Kristian Jegerud
Marthin Gunerius Klækken

Inventory Management System for Cryopreserved Biological Material

Bachelor's project in Department of Computer Science
Supervisor: Tom Røise
May 2021

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



Abstract

Cryogenetics is a growing company in storing and the preservation of aquatic genes, and they wanted a solution that could store their data in a database with an associated web application. The solution was a website where you can see an overview of all the containers and a map over the content in each of them. You can also see which excel-spreadsheets are saved as a backup in the database, and which are ready for approval. In essence, they wanted an inventory management system.

Our solution contained a back end API written in .NET 5. The web application was customized for both PC, tablets and mobile phones. Cryogenetics requested a product that could easily be developed further, and we therefore facilitated for this. We had focus on working professional during the project with tools like GitHub, Scrumban and Confluence.

Sammendrag

Cryogenetics er en bedrift i vekst innen lagring og bevaring av akvatiske gener, og de fremmet et ønske om en løsning som kunne lagre dataen deres i en database med en tilhørende webapplikasjon. Løsningen var en nettside hvor man kan se en oversikt over alle beholderne og et kart over innholdet i hver av dem. Man kan også se hvilke excel-regneark som ligger lagret som en backup i databasen og hvilke som ligger klar for godkjenning. Kort fortalt, de ønsket et lagerstyringssystem.

Løsningen bestod også av et backend API skrevet i .NET 5. Webapplikasjonen er tilpasset både for PC, nettbrett og mobiltelefoner. Cryogenetics ønsket et produkt som lett vil kunne videreutvikles, og det er i vårt prosjekt derfor lagt godt til rette for dette. Vi har hatt fokus på å arbeide profesjonelt med arbeidsverktøy som GitHub, Scrumban og Confluence.

Preface

This bachelor thesis is written by Sander Låstad Olsen, Marthin Gunerius Klækken, Casper Fabian Gulbrandsen and Kristian Jegerud at the Department of Computer Science at NTNU in Gjøvik.

We would like to acknowledge those that helped us along the way in the process of developing the product and writing this thesis. A special thank you to Steffen Wolla and Cryogenetics AS for giving us the task and helping us with answering any questions we would have. We would also like to thank our advisor Tom Røise for his help in writing this report. Finally we would like to thank our friends and family for the support provided to us.

Contents

Abstract	iii
Sammendrag	v
Preface	vii
Contents	ix
Figures	xi
Code Listings	xiii
Acronyms	xv
Glossary	xvii
1 Introduction	1
1.1 Background	1
1.2 Project Group	4
1.3 The Report	5
2 Development Process	7
2.1 Development Methods	7
2.2 Scrumban	8
2.3 Execution	9
2.4 Summary of Work and Meetings	11
3 Requirement Specification	15
3.1 Use Cases	15
3.2 High-level Description	16
3.3 Expanded Description	17
3.4 Operational Requirements	18
3.5 Security Requirements	19
3.6 High Level Misuse Cases	20
4 Technologies	23
4.1 Cloud	23
4.2 React	25
4.3 TypeScript	27
4.4 Docker	28
4.5 Git	28
4.6 CSS Grid	28
4.7 RESTful API	29
4.8 .NET 5 & C#	29
4.9 Technical Memo	31

5	Design	33
5.1	Structure	33
5.2	Front End Structure	34
5.3	Database design	37
6	Implementation	41
6.1	Front End Web Interface	41
6.2	Routing	43
6.3	The Navigation Bar	47
6.4	The Storage Tanks	49
6.5	User profile page	60
6.6	The Authentication Process	61
6.7	Back end	66
6.8	Back End Security	69
6.9	Production setting	72
7	User Interface	73
7.1	Fluent UI	73
7.2	Layout	73
7.3	UI Examples	75
7.4	Responsive User Interface	79
7.5	User Interface Iterations	81
7.6	Web Content Accessibility Guidelines	82
8	Development Environment	85
8.1	Front End	85
8.2	Back End	88
9	Quality Assurance	89
9.1	Front end	89
9.2	Back end	93
9.3	Testing	94
9.4	Automated Test	95
10	Conclusion	97
	Bibliography	103
A	Scrumban tasks	105
B	Project Agreement	109
C	Project Plan	115
D	GUI-workshop	129
E	Rough initial recurring price estimate	135
F	Initial SQL draft	139
G	Swagger UI, and authorization configuration	143
H	Managed identity	147
I	Toggl Track Summary Report	151

Figures

2.1	Scrumban Board 3/5-21	9
2.2	Quick overview of the development process.	11
3.1	Use case diagram (created in Draw.io)	15
3.2	Primary misuse cases (created in Draw.io)	19
5.1	Current overall architecture	33
5.2	Our Front End Folder Structure (created in draw.io)	36
5.3	Example schema for instances of fish, tank and client documents . .	39
6.1	Front end basic layout and areas	42
6.2	File structure - front end (made in draw.io)	43
6.3	Front End Routing Structure	44
6.4	500 Liter Tank	53
6.5	Tank table	56
6.6	Application registration on portal.azure.com	62
6.7	A user tries to log in with an email that is not listed in the app's Azure AD	63
6.8	Screenshot from Microsoft docs. [11]	66
7.1	Front end layout (Laptop screen)	74
7.2	Color palette	75
7.3	Dashboard landing page	75
7.4	Dropdown	76
7.5	User profile page	76
7.6	Filtering the list by the date uploaded	77
7.7	47 Liter tank	78
7.8	500 Liter tank	78
7.9	Front end layout (Tablet screen)	79
7.10	Account page before and after clicking on the hamburger menu in mobile view	80
7.11	Mobile layout	81
7.12	Mobile layout	82
7.13	Mobile layout	82
7.14	Mobile layout	83

8.1	Git branch structure example	87
8.2	Endpoint to get all fish in a specific tank with parameters	88
9.1	Testing the responsive design on an iPhone display	94
D.1	Wireframe designs	131
D.2	Design iterations of the approve files list	132
D.3	From suggestion to implementation	133
E.1	Rough initial recurring price estimate	137
F.1	Rough initial EER diagram design	141
G.1	Swagger UI showing some endpoints, and link to specification . . .	145
G.2	Authentication on Swagger development site as configured through the Azure Portal.	146
H.1	Managed identity enabled on App Service	149
H.2	Access policy of keyvault with managed identity	150

Code Listings

6.1	Using Fluent UI components	42
6.2	Main Routing	45
6.3	Using Protected Routes	45
6.4	Protected Routes Component	46
6.5	Side Menu Code Example	47
6.6	Adding properties to custom components	48
6.7	Infinite Scroll	50
6.8	Fetching all tanks	51
6.9	Redirecting to new page	52
6.10	Display a tile in 500L tank	53
6.11	Fetching map data from API	54
6.12	Handling of tile click	55
6.13	Constructing fish location in tank	57
6.14	Function used to fetch from API	58
6.15	Using the FetchFromAPI function	60
6.16	Checking and retrieving access token for Microsoft Graph API	60
6.17	fetch a user's job title from Microsoft Graph API	61
6.18	Get access token code	64
6.19	Instantiating authorization system with graph api and test scopes.	65
6.20	Example class deriving DynamicObject	68
6.21	Fetching parameterized data using raw Structured Query Language (SQL) - Pseudocode	69
6.22	Fetching parameterized data using Language integrated query (LINQ) to SQL - Pseudocode	69
6.23	Pseudocode JObject for safe SQL generation	69
6.24	Enabling CORS in Azure APIM	72
8.1	Dockerfile	85
8.2	docker-compose.yml	86
9.1	Before refactoring	90
9.2	After refactoring	91
9.3	Before Prettier is run	92
9.4	After Prettier is run	93

Acronyms

- CORS** Cross-Origin Resource Sharing. 71, 72
- CSP** Cloud Service Provider. 24, 25
- CSS** Cascading Style Sheets. 79
- DoS** Denial of Service. 34, 71
- LINQ** Language integrated query. xiii, 69, 70
- MVC** Model-View-Controller. 35
- MVVM** Model-View-ViewModel. 34
- NPM** Node Package Manager. 71, 92
- NTNU** Norwegian University of Science and Technology. 1
- PaaS** Platform as a Service. 23
- RDBMS** Relational Database Management System. 37, 38
- SaaS** Software as a Service. 23
- SPA** Single page application. 33
- SQL** Structured Query Language. xiii, 69
- UI** User Interface. 34
- VM** Virtual Machines. 28

Glossary

Azure AD Azure Active Directory, identity management service. Access control for users and objects.. 25, 70, 71

code coverage a measurement of how much of the code are executed while the automated tests are running. 96

component based development an approach to software development that focuses on the design and development of reusable components.. 4

front end In development refers to the practice of converting data to a graphical interface for example through the use of HTML and CSS. 8, 27, 34

kanban A workflow management method to help visualize your work and maximize efficiency.. 8

media queries A popular technique introduced in CSS3 to deliver different style sheet to different devices or screen sizes. It works as a condition that has to be met for the CSS to be read by the browser like for example the screen width or the screen format. 79

milt is the semen from fish, molluscs and certain other aquatic animals that reproduces by spraying this fluid, which contains the sperm cells, on eggs. It may also refer to the testicles or the sperm sacs that contains the semen. 2

react hook hooks allows you to use state and other React features without writing a class. 47

refactoring a technique for gradually improving the quality of program code. The purpose is to restructure code to make it easier to read, and easier to maintain and further develop.. 90

rem A relative unit of measurement in CSS. Rem are defined by the font size of the root element, with 1rem being equal to the base font size of 16 pixels.. 79

scrum is an agile development methodology used in the development of Software based on an iterative and incremental processes. Scrum is adaptable, fast, flexible and effective agile framework that is designed to deliver value to the customer throughout the development of the project.. 8

suspicious constructs constructs that may not represent what the programmer intended to do. E.g. expressions without side effects used in a context where side effects are expected.. 92

test driven development a software development methodology which consists of short iterations where new tests covering the desired improvements or new functionality are written first, then the production code needed to make the tests pass flawlessly is implemented.. 96

upsert Either updates or sets if the document already exists, in Cosmos DB in an atomic fashion.. 37

web components A set of web APIs that allow you to create new custom, reusable and HTML tags to use in web pages.. 26

Web Content Accessibility Guidelines A number of guidelines for how web pages and user interfaces are designed ensuring that information on the Internet is easily accessible to everyone, regardless of functional ability. 82

workfolder A special folder in Cryogenetic's SharePoint, containing finished worksheets ready for approval.. 17

worksheet Worksheet is an excel document that records work in progress or work that is done. xviii

Chapter 1

Introduction

This thesis is written by four students for their bachelor's degree in Computer Science at the Norwegian University of Science and Technology (NTNU) during the spring semester of 2021.

1.1 Background

Cryogenetics is a Norwegian company that specialize in cryopreservation of milt from several different aquatic and marine species as well as provision of technologies and services for improved reproduction.

The company has locations worldwide with headquarters based in Hamar, Norway. The technology makes it possible to freeze sperm (otherwise known as milt) from male fish, to then later thaw to use in reproduction when needed, allowing for more efficient food production with less use of medicines.

Conservation and Cryopreservation

Cryopreservation is the process of freezing biological material at extreme temperatures, causing all biological activity to stop, including reactions that would otherwise lead to cell death.¹ Cryogenetics have developed protocols for many different species to provide the best preservation of the desired genes. They also provides qualified knowledge to help clients get the most out of their fish and valuable genetic material.

Their workflow for storing their data is today based on data handled in Excel spreadsheets stored in a work folder in SharePoint². When these spreadsheets are finished and approved by an employee they are then stored in a different folder containing other approved documents.

¹<https://www.cryogenetics.com/products-and-services/>

²<https://www.microsoft.com/nb-no/microsoft-365/sharepoint/collaboration>

Task

As Cryogenetics continues to grow as a world-leading company in cryopreservation of milt, their need of effective overview of documents as well as to automate certain activities quickly becomes a necessity.

The process of having to look up all the documents in the SharePoint work folder, to then review and approve the documents by manually moving them to the database demands a lot of time.

Therefore, our task is to create a web user interface for Cryogenetics' staff which aim to reduce the time and labor needed to review, approve and move the documents to the work folder. We will not reinvent the wheel and replace Excel with a separate service, but rather create a service that automates the process of reviewing and approving the excel documents, as requested by Cryogenetics.

Project Description

We will deliver a robust web user interface for Cryogenetics' staff. Even though it is primarily made for PC usage, the interface is made with careful breakpoints in order to scale well with other common devices, such as tablets and phones.

The web user interface will have the following functionality:

- The user will be able to log in with their Microsoft account.
 - The staff will already have their own Microsoft accounts which they use for the company's SharePoint work folder. The staff will therefore not need to create a brand new account just for this web application.
- A dashboard which shows relevant information that may be helpful such as:
 - How many files are awaiting approval
 - How many files are currently stored as backups
 - How many fish are currently stored in tanks
 - How many tanks are currently operational
- A list of files waiting to be approved with:
 - Approve functionality for each file
 - The ability to filter the files based on file types and dates
- A list of files stored as backups with:
 - Download functionality for each file
 - The ability to filter the files based on file types and dates
- A list of all tanks where:
 - The user should be able to filter the files based on tank sizes (11, 47 and 500 Liter capacities)
 - The user should be able to register a new tank
 - Each tank consists of cylinders where the milt is stored, and the staff will be able to click on each tank in order to get a good visual overview

of each cylinder.

- A user profile page that retrieves relevant information from the user's Microsoft account.
- A search bar which allows the staff to look up specific details in the database.

The project will also consist of an Azure database that will store the approved excel documents. This means that once the admin approves the file, it will be transferred to this database. We convert the excel documents that are transferred to the database into JSON-format. This allows us to use SQL queries for when the staff uses the search bar on the front end to look up specific details.

Delimitation

The product we will create for Cryogenetics must be able to be used as a tool in everyday work, and we will therefore focus on creating a robust service that has functioning features, and that are easy to maintain and develop further. Cryogenetics are closely integrated with the existing Microsoft products, and we therefore want to take advantage of the fact that all Cryogenetics employees have a Microsoft account. This means that we do not have to take responsibility of creating new users and storing account information securely.

The report will focus mainly on the development process, choice of technologies and the reasoning behind our choices. We will also discuss our development process, what we learned from this project, and what we would have done differently.

Target Group

When we develop the services, we will always have the staff in focus. They are the ones who will use the product, and it is therefore important that we make the product for them.

The target group of our report is parties involved in the project, such as our client, our supervisor and our department at NTNU. The report will also be relevant for developers who may further develop the product, our fellow students and future employers.

Why We Chose This Task

When this group came together we all wanted to choose a task where the focus was on developing a product, rather than a research based and more theoretical task. All four members in our group had preferences on technologies we wanted to use and wanted to learn, so this was something we had in mind when we decided which task to choose.

We had an early meeting with Cryogenetics represented by Steffen Wolla. Here we clarified ambiguities in the task, and we as a group gave some input and Cryo-

genetics was able to present several ideas. After this meeting we as a group got a very good impression of the the task, the company and the contact person.

Since the task was to create a product that Cryogenetics wanted to use in their workflow, combined with great freedom when choosing technologies and services; we decided to put this task at the top of the list of desired tasks.

1.2 Project Group

All four members on the group study Bachelor in Computer Science at NTNU Gjøvik, and with the exception of 5th semester electives we all have the same foundation. In addition to the standard topics in the course, Sander and Marthin have taken the IMT3501 Software Security course. This was useful when creating a secure product that contains confidential information. Kristian and Casper have both taken the courses IDG1362 Introduction to user-centered design and PROG2053 WWW-technologies. This was useful when creating a web-service with good usability that follows given standards for, among other things, WCAG 2.0 (See section 7.6).

The WWW-technologies course has also given us particularly good experience in developing an organized and efficient web-service, which means that we have experience of what to do and what not to do. Casper has also taken the course IMT3281 Application Development, which will be a benefit as he has experience with development of complex applications.

Throughout the course the group have gained competence in fields such as database development, software security, algorithmic methods (fast and efficient coding), Software Engineering (planning, implementations, development methods, etc...), and web development (HTML5, CSS, JavaScript, component based development).

We therefore believed that we had a good basis to complete this project in a satisfactory way, and to develop a robust product for Cryogenetics.

Roles

During the planning phase we decided roles and responsibilities for the project. It was not intended that these should be binding, but the roles fell so naturally that we stayed with them throughout the project period. Many of the tasks on the website are developed through collaboration and pair programming between two group members.

The roles were as follows:

- **Sander:** Project manager and back end developer. Responsible for managing the group, ensuring communication with the product owner, and responsible for organizing and convening meetings.
- **Marthin:** Project Security Manager and front end developer. Responsibility to ensure that the development follows approved and secure development standards. He also assisted on the back end when he had time to spare.

- **Casper:** Project Development Manager, and front end / UX developer. Responsible for setting up workable development environments, and making sure the team follows the chosen development model and is using the same code standard. Also responsible for maintaining the GitHub repository and making sure the team follows agreed upon Git standards.
- **Kristian:** Project documentation manager and front end developer. Responsible for making sure that as much as possible of the development and administration is documented and made easily available to the team. Also responsible for taking notes during meetings and ensuring that the team members log their hours accurately.

1.3 The Report

The source code is attached as a ZIP-file in the submission of the thesis. The report has the following structure:

1. **Introduction.** Briefly about the client, the project assignment, clarifications, target group and the project group.
2. **Development Model.** Choice of development model and why we chose as we did. How we worked and briefly about the progress.
3. **Requirement Analysis.** Use case diagrams. The operational and security requirements we concluded with.
4. **Technologies.** Review of the technologies we have used in the project and why we have used them.
5. **Design.** How we have structured our application, which pattern we took inspiration from and design of database.
6. **Implementation.** How we implemented our application. Deeper dive into the code, with some examples of how we used the technologies we chose to use.
7. **User Interface.** The result of our website, with examples of final UI design. Some UI tests regarding WCAG.
8. **Development Environment.** Which tools we have used during the development.
9. **Quality Assurance.** How we made sure that we produced high quality code.
10. **Conclusion.** What we have learned from the project, what we did right and what we could have done different.

Chapter 2

Development Process

In this chapter will go into more detail on the argumentation of our development methods, and how we actually carried out the project work. We will describe which tools out of the toolbox we decided to use, and why we thought these tools would help us achieve the best systematic process from a mere idea, to an actual working product.

2.1 Development Methods

In short, a development method is a structured set of activities required to develop a software system [1].

There are many ways to organize the process of developing a software product. However, the common factor that all of these methods are designed to describe, is to help the team to determine a set of norms that says "this is how we are going to work, and this how we are going to communicate with each other".¹

Even though these methods are designed with many common factors, they are still meant to be applied to different types of projects. We discussed early on the various ways we wanted to work on this development project, and we all agreed that an agile development process was the best suited method.

So why did we decide this?

- Cryogenetics did not have a hard-determined set of requirements, but rather wanted to be actively involved in the development process.
 - If we were going to use the waterfall method, it would usually be a good idea for the customer to have a pre-determined set of requirements and to not interfere much with the work process after the requirements have been documented. In essence, it is not a flexible development process.
 - Agile methods creates room for change during the development process, and sudden features and changes is something Cryogenetics told

¹<https://www.agilealliance.org/what-is-scrumban/>

us early on that could occur. It is a flexible development process.

- Our roles were flexible
 - Aside from the fact that we had pre-determined who works on the back end and who works on the front-end, there was still a lot of flexibility in terms of who can do what. Therefore, a kanban [2] board is something that we knew early on would fit us well in the project development.
- The project had a relatively short timeline.
 - When the requirements are vague and the timeline is limited it is highly important for us to be flexible in terms of what software components are to be prioritized.
Therefore, being able to have a continuous workflow with the simple use of a scrumban board, where the components of higher priorities are developed first was very important for us. We could then let components of lower priorities wait until we have time to spare.

Due to the flexibility required, an agile method was required. We landed on the combined method of using both scrum and kanban, a method which quite remarkably is called Scrumban.

2.2 Scrumban

Scrum and Kanban are what's usually considered the fundamental flavors of Agile. Scrumban uses short iterations which should not exceed two weeks ². After the iteration is finished, the team reflect over the work that has been done, and along with the product owner plan for the next iteration. Our iterations fluctuated in length and sometimes were longer than recommended which is why it is important to add that we did not follow the text-book definition of Scrumban. How we used Scrumban began with the Scrumban Board.

Scrumban Board

Because we used GitHub as our source control manager, we also decided to use their integrated project boards [3] for our work management tool (See figure 2.1), and all of our stories(tasks) were added into our to do column.

Tasks are by default added to the *To Do* column, and when a group member starts a task, it is transferred to the *In progress* column, and the task would be assigned to that group member. Once the task has been completed and the component is working, it is transferred to the *Done* column. We think this gave us a good overview over the front end work flow. It made it easy to see which tasks remained and what the other group members worked on. A selection of tasks from the Scrumban board can be found in appendix appendix A.

²<https://ora.pm/blog/scrum-vs-kanban-vs-scrumban/>

Planning in Scrumban is based on the demand for it. This means that when the to do column was empty, a planning meeting was held to add new issues, and to also discuss which we should prioritize.

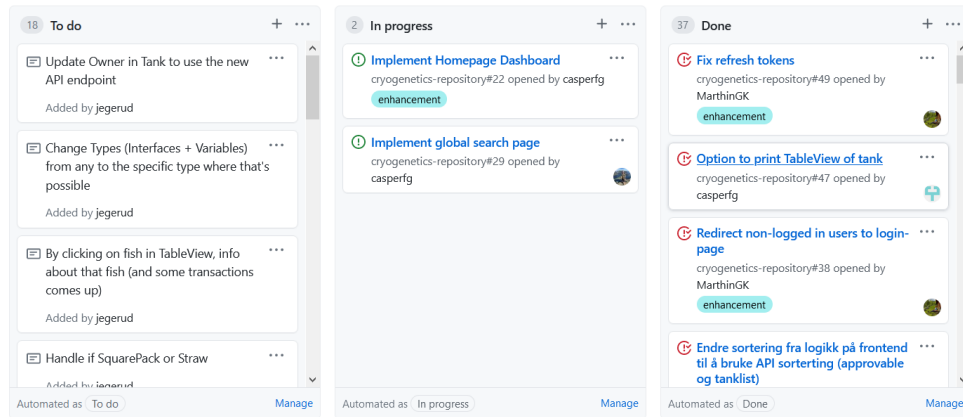


Figure 2.1: Scrumban Board 3/5-21

2.3 Execution

Because of having three developers work on the front end whilst mainly one developer work on the back end, it was only the front end group who took the Scrumban methodology into use.

On the back end, cards and to do lists was not in use. In retrospect, this was a mistake. A continuous backup system was used for the source code. Continuous integration was provided from the main branch to an Azure App Service, so that when code was pushed, it was compiled and set into production. This had the effect that only code that was secure could be pushed to the repository, which meant fewer and larger commits. Instead, a separate production branch should have been set up, where the continuous integration was set up. Or committing to a development branch, and only merging production ready code to master.

Meetings with Product Owner

The representative from Cryogenetics was Production Manager and Business Developer Steffen Wolla. We started early on in January to have meetings with the product owner, but the agreement of weekly meetings did not start until the end of the preliminary project (1. February). Here, we agreed to have a weekly meeting every Monday at 12PM, where we updated him on the progress we had made since the last meeting. Here, Wolla also supplied more ideas that we could implement, or improve on what we already had implemented.

Because of the Coronavirus pandemic, all but one of the meetings were digital via Microsoft Teams. We think meetings through Teams have worked well,

although there is little doubt that we would have preferred more physical meetings than we have had. This is because meeting physically makes it easier to have a natural conversation around the product, and we could also have run user tests together with Cryogenetics. Despite of this the meetings were vital for us to understand exactly how Cryogenetics worked on a day to day basis and also to brainstorm further ideas along with the product owner.

We had one physical meeting in January at their offices, where Wolla gave us a tour of their building and a closer look at how they worked. We got to see their cryopreservation tank storage and also see how they would organize the contents within the tanks. This gave us an insight into their everyday work and the tools they use, which was beneficial when developing the services later on. After the tour we sat down all together to discuss various use cases for the project and also give Cryogenetics a presentation of some design ideas we had prepared before the meeting.

Front End Meetings

On the front-end we had daily morning meetings every weekday, where we discussed what we had worked on since the day before, and what we were going to work on that day. Here we also discussed the status of the Scrumban board and what we needed to prioritize going forward. This kept us on the same page and helped us preserve a good communication together throughout the development process. Another effect of these meetings is that we became considerably better at allocating the necessary work components to the correct developers, as we all had different experiences in different fields. Because of the Coronavirus pandemic, the vast majority of these meetings were held on Microsoft Teams.

Group Meetings

We had group meetings every Thursday where the back end developer would also join to update the front end on his work, and vice versa. Here, we analyzed the changes that had been made since the last meeting and how the new back end and front end changes could be integrated together in order to create a working component.

We tried to keep these meetings physical whenever we could, but variation in the recommendations around the pandemic made it difficult to make this a regular routine. We felt that physical meetings provided a better atmosphere of dialogue, and that it generally builds more trust and stronger bonds between the developers. So this was something we agreed early on was important as a group to try to achieve.

Meetings with the Supervisor

We were quite unlucky when it came to supervisors. Because of different complications with the first two supervisors we had, we did not have a proper guidance

before we got our third supervisor. At this point, we were already well into the month of April, but Tom Røise stepped in as our third supervisor in time to guide us through our thesis writing.

We had our first meeting with Tom Røise on 30. April. Here he gave us feedback on what we had written so far in our report, and we agreed to meet again over Microsoft Teams every Friday until the project's deadline. The main focus of these meetings in general was to give us a feedback on what we had written since the last meeting, as well as to discuss the future structure of the report itself.

2.4 Summary of Work and Meetings

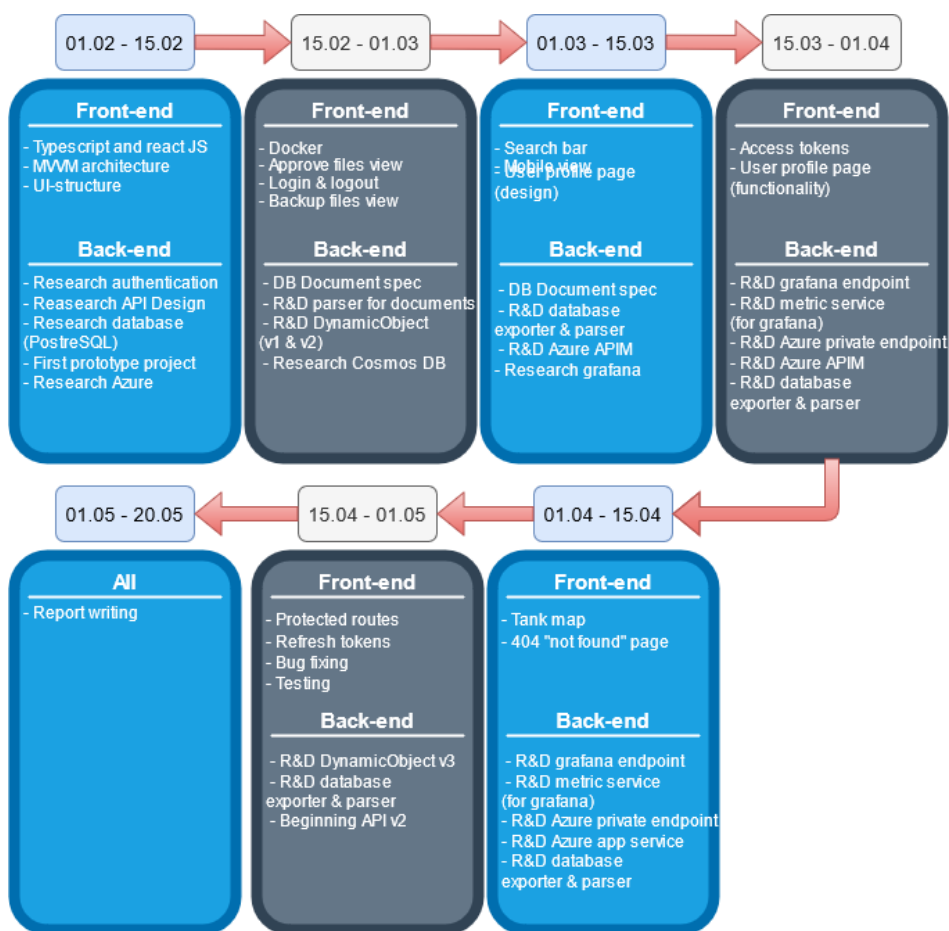


Figure 2.2: Quick overview of the development process.

We mentioned above (in section 2.2) that we chose the flexible management methodology of Scrumban, and we agree that this has been the right choice for this project. We followed our determined version of Scrumban throughout the entire project, and Scrumban have shown to be an immense help throughout the

entire development process. This is especially thanks to the to do list that was continuously updated to supply us with prioritized tasks, and of course the Scrumboard in general which helped us all stay updated on what components were being worked on.

As we also mentioned in section 2.2, we replaced our sprints with a continued workflow, so instead of a summary of weekly sprints we have a brief summary every 15 days.

1. February — 15. February

Our preliminary project was now delivered and it was time to focus on the main project. Repositories were set up on GitHub, (read more about our use of Git in section 4.5). The repositories were set up with a foundational structure within MVVM (section 5.2), and the use of React (section 4.2) and TypeScript (section 4.2) as the main framework and programming language were chosen for the front end department.

15. February — 1. March

Docker (section 4.4) is setup on the front end. Azure authentication for Microsoft login/logout functionality is implemented. The web user interface (chapter 7) have also seen great improvements by the implementation of the login page and file listing views.

1. March — 15. March

On the front end we have implemented a web user interface view that is also compatible with both tables and mobile devices. The search bar is now working to find fish throughout the entire database as the product owner first and foremost wanted, but global search is also requested. The user profile page have also been implemented in terms of design, but fetching data from the user's Microsoft account is yet to be implemented.

15. March — 1. April

Functionality for making GET-requests against protected Microsoft account data and display the data on the user profile page was implemented. This was done with Microsoft Graph access tokens.

1. April — 15. April

The tank map is now a working component. Azure authentication now uses redirect instead of a popup.

15. April — 1. May

Protected routes [4] was implemented. "404 not found page" was made, in case a user tries to navigate to a file path in the URL which do not exists. Refresh tokens [5] also became a working functionality. Bug-fixing, refactoring and redesign became a bigger focus on the front end, and the main functionality of the web user

interface is finished. Report writing now became the bigger focus.

1. May — 20. May

Report writing is the main focus. We also had a presentation of the product to Cryogenetics, as well as some user tests on the web user interface. Read more about the user tests in chapter 9.

Chapter 3

Requirement Specification

Requirement Specification is where we specify all requirements that are to be imposed on the design and verification of the system. This describes what the software will do and how it will be expected to perform, as well as what it should do and shouldn't do.

3.1 Use Cases

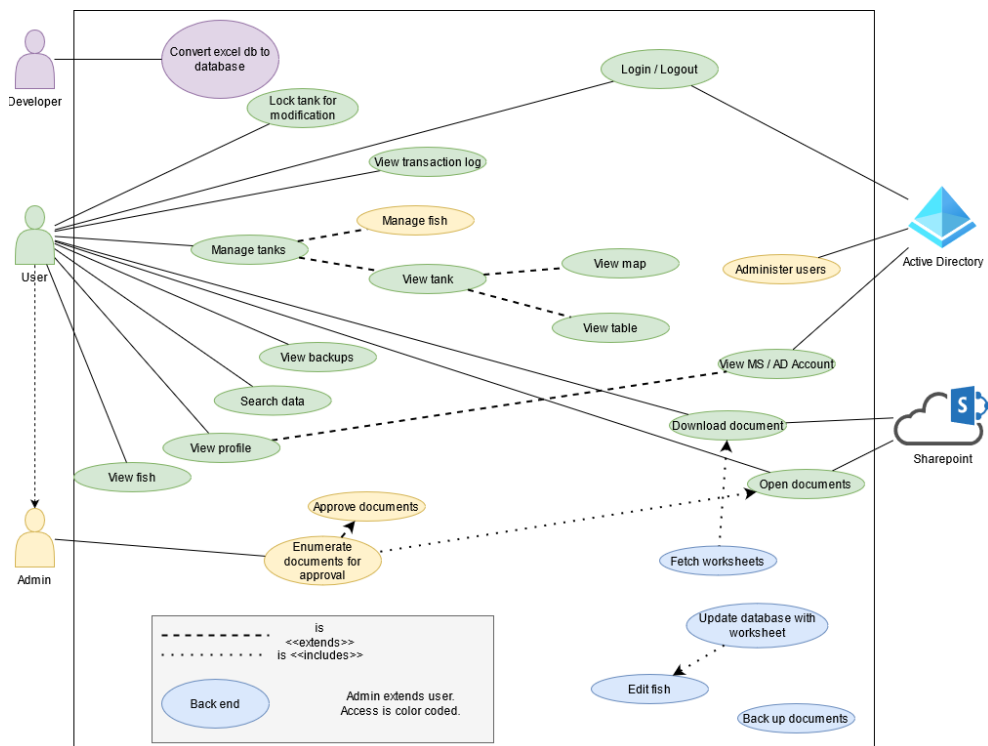


Figure 3.1: Use case diagram (created in Draw.io)

Being able to locate and view data quickly, along with automating adding the worksheets to a robust database were the most important use cases. It is how we chose the use cases to use for our high-level descriptions, and our expanded use cases. See section 3.2 and section 3.3.

3.2 High-level Description

Use Case:	View fish
Actor:	User & Admin
Purpose:	View data from a fish-centric perspective
Description:	A user should be able to view all data regarding a specific fish individual, including all locations the individual is stored in.
Use Case:	Search data
Actor:	User & Admin
Purpose:	Find a specific resource that is stored in the database
Description:	A user can search all data stored using a search bar, including all tanks and fishes, from a single location. The user can also use basic filtering, such as searching for a specific fish species.
Use Case:	View tank
Actor:	User & Admin
Purpose:	To see information about what is stored in the tank
Description:	The user can find and click on the unique tank based on tank ID in the provided list of tanks. The user will then get an overview over the different fish that is stored in the tank that was selected. This includes an overview over fish type and species, as well as the name of the client that is the owner of that fish. It should also provide an overview of where in the tank the fish is stored.
Use Case:	Update database with worksheet
Actor:	Back end
Purpose:	Use the worksheets to update the database.
Description:	Uses approved worksheets containing updates to the data, and adds them to a transaction log. The transaction log is used to update the database.
Use Case:	Create new tank
Actor:	Admin
Purpose:	To register a new fish tank in the database
Description:	From the tanks-route, an admin can click on the "New tank" button. Here, a popup will appear which prompts the admin to fill in the tank credentials, such as the tank serial number, the tank location and the tank type.

3.3 Expanded Description

Use Case:	Approve documents
Actor:	Admin
Purpose:	Queue ready documents to update database
Description:	A worker queues a worksheet in a special folder on SharePoint. An administrator can the read and approve the document when ready. Approving the document should queue it for modifying the database.
Pre-condition 1:	The worker must have internet connection
Pre-condition 2:	A valid document must exist in the SharePoint workfolder.
Post-condition:	The document is converted into JSON-format and is loaded into the database
Detailed course of action:	<ol style="list-style-type: none"> 1. Worker navigates to the approve-files list. 2. Worker locates the file they wishes to approve. 3. Worker clicks on the "approve"-button besides the file. 4. The file is copied into the backup-files list. 5. The file is converted from Excel into structured JSON-format. 6. The structured JSON document is then used by the back end to update the database with it's content. 7. The file is removed from the SharePoint workfolder. 8. The file is removed from the approve-files list.
Alternative scenarios:	<p><i>Variants: Error situations:</i></p> <ol style="list-style-type: none"> 1. The website fails to establish connection to the SharePoint workfolder. 2. There are no files in the approve-files list. 3. The file can't be loaded into the backup-files list. 4. Invalid document type (f.ex. .docx) prevents the file from being to converted into JSON.

Use Case:	View tank map
Actor:	User & Admin
Purpose:	Give quick overview over tank, and all stored fish.
Description:	The user can find and click on the unique tank based on the tank ID in a provided list of tanks. The user will then get an overview over the different fish that is stored in the various compartments of the tank that was selected. This includes an overview over fish type and species, as well as the name of the client that is the owner of that fish. It should also provide an overview of where in the tank the fish is stored.
Pre-condition 1:	The user must be on the list of tanks route.
Pre-condition 2:	The tank type must be in a list of pre-approved types.
Post-condition:	The map of a selected tank is shown.
Detailed course of action:	<ol style="list-style-type: none"> 1. Worker locates a tank from the tank list. 2. Worker clicks on the tank. 3. Data about the tank is fetched from back end API. 4. Worker is navigated to the tank map route. 5. Worker sees a map of the tank.
Alternative scenarios:	<p><i>Variants: Error situations:</i></p> <ol style="list-style-type: none"> 1. The API fails to load the tank map. 2. Internet connection is lost.

3.4 Operational Requirements

Throughout the project period we have through internal discussion and frequent dialogue with the product owner, come up with operational requirements that should ensure that the system scales well and remains available and operational.

- The database must be able to store at least 10 GB of data (2 orders of magnitude more than current storage).
- The database must not have data loss. (< 12 nines)
 - The capacity of their current database is in the order of magnitude of about 50MB, but since Cryogenetics is a company that is currently experiencing fast growth it is important to have a database that scales well in the future.
- The user interface language must be in English, and optionally in Norwegian.
- Constant uptime is important, but not exceedingly so, due to work still being performed in excel spreadsheets. An uptime of no less than 99.9% (3 nines) is required.

3.5 Security Requirements

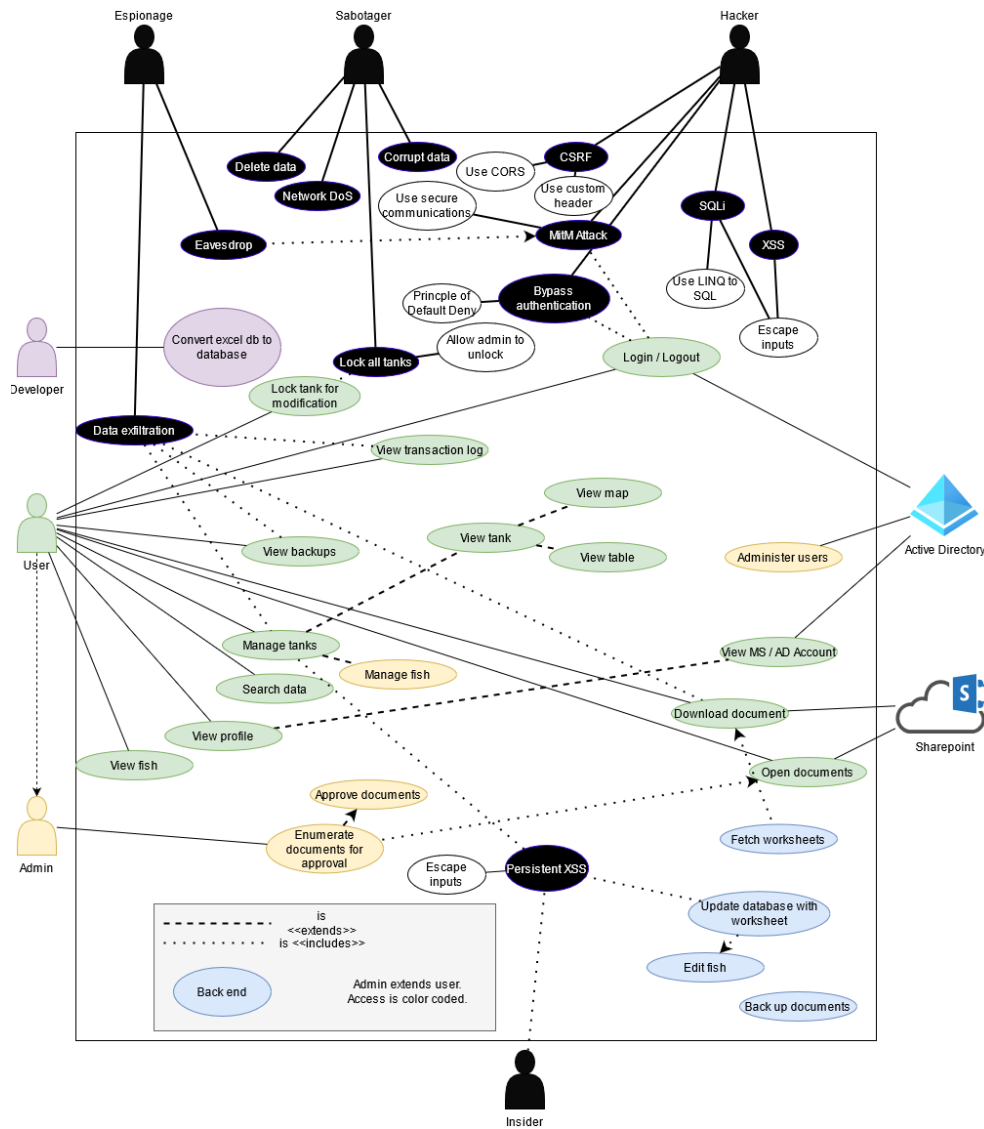


Figure 3.2: Primary misuse cases (created in Draw.io)

Cryogenetics explicitly expressed that the security and confidentiality of all production and customer data was of vital importance. No more specific security requirements were made.

This meant that all the Excel spreadsheets that are stored in the SharePoint work folder must never be disclosed or made available to anyone except for Cryogenetics' staff.

Apart from this there has not been a lot of input from Cryogenetics regarding the security, so we had some internal group discussions and individual reflections

on the listed requirements that we believed to be important for this system:

- Access tokens must have a limited life time. Limiting the lifetime of the access token limits the amount of time a potential attacker can use a stolen token.
 - This does of course quickly create another problem by rendering a protected resource inaccessible for the user after the access token has expired. To solve this problem, OAuth 2.0 introduced an artifact called a refresh token. We implemented the refresh token into our system, which allows an application to obtain a new access token without prompting the user.
- Only emails listed in Cryogenetics' SharePoint Active Directory should be listed on the website's Azure application.
 - If a user is logged in with an email that is not listed in Cryogenetics' SharePoint Active Directory, it means that the user will not be able to see any of the files in that directory. This is another reason why we chose to use Azure Authentication as the login method for our system.
- Worksheets and customer information must never be displayed directly in the web page, but must be loaded in from SharePoint.
 - Worksheets and customer information can only be accessed in Cryogenetics' SharePoint Active Directory. This directory will of course be accessible through our web page, but the directory information itself will never be directly displayed on our web page.
- HTTPS communication encryption through Transport Layer Protocol (TLS) is to be used.
 - This is not something we were able to implement before the project deadline.

Explanations for these security requirements are addressed throughout the report, but the topics regarding application security and authentication can especially be found in section 6.6.

3.6 High Level Misuse Cases

For the misusecase diagram, see figure 3.2.

Misuse Case:	Network DoS
Actor:	Saboteur
Purpose:	Sabotage use of website and database
Attack:	There are two main variants: <ul style="list-style-type: none"> 1. Low level DoS (OSI 3 and down) Flood the server with more traffic than it can handle, preventing valid requests from going through. 2. Mid/High level DoS (OSI 4 and up) Send specially crafted requests, that exploit a weakness in the application, slowing down or preventing service.
Mitigation:	<ul style="list-style-type: none"> 1. Low level DoS (OSI 3 and down) Use a DoS mitigation service, such as Cloudflare, Azure Front Door (which has Azure DDoS Protection Basic), Azure DDoS Protection Standard, or other. 2. Mid/High level DoS (OSI 4 and up) <ul style="list-style-type: none"> a. Avoid unauthenticated access, and rate-limit requests (especially computationally expensive ones). b. Log utilization of endpoints per client, monitor for suspicious use and revoke access tokens. (Although for this project in specific, we would want this automated, as no dedicated IT department exists in Cryogenetics) c. Take care when allocating resources that may persist beyond a request (such as caches).

Misuse Case:	Data deletion
Actor:	Insider & Saboteur
Purpose:	Deny access to critical resources
Attack:	Delete critical resource(s) denying access to them, potentially permanently.
Mitigation:	Use append-only storage, or soft-delete. Soft-delete requires a set waiting period before resources are actually deleted, allowing resources to be restored.

Misuse Case:	Corrupt data
Actor:	Insider & Saboteur
Purpose:	Deny access to critical resources
Attack:	Corrupt critical resource(s) denying access to them, potentially permanently. Corrupted data is different from delete, in the sense that the invalid data may propagate throughout the system. It may also be more difficult to detect, as there may be no easy way to detect if the data is valid or not.
Mitigation:	<ul style="list-style-type: none">• High level: Use a transaction system with a transaction log as the only way of modifying the database. In the case someone does input invalid/corrupt data, it is always possible to go back and remove bad transactions, then to rebuild the database. This however does not mitigate the issue of an employee intentionally inputting invalid/bad data instead of the business data they were supposed to. This could be as simple as a human error, writing down the wrong tank number or fish id for example.• Low level: Store data in a system where multiple redundant copies are kept, and that has integrity checking and prevents data degradation.

Chapter 4

Technologies

In this chapter we will discuss what kind of tools we decided to utilize in order to create our application, as well as why we chose to use those tools as opposed to other possible tools.

4.1 Cloud

Why Cloud?

We decided to use Microsoft Azure¹ as our main platform of choice, instead of hosting locally at Cryogenetics, or hosting co-location in a data centre. Cryogenetics as a company is not specifically focused on IT, and does not have a significant IT department, opting to instead outsource most of it's operations to 3rd parties. Hosting locally, or co-location may require significantly more maintenance than using Platform as a Service (PaaS) or Software as a Service (SaaS), as more things need to be managed by you². Platforms such as Azure are generally more expensive than either local hosting, or co-location; assuming you have a IT department capable of maintaining the software and hardware stacks, or you need significant raw compute power. This project has neither.

Additionally, public cloud providers provide significantly more flexibility in terms of compute. And are a great fit for tasks with spiky loads. As such, this project was an excellent fit for using a cloud service provider with SaaS and PaaS options.

Vendor Lock-In Concerns

Using PaaS presents a concern with vendor lock in. That is, code being tied in with a specific vendors systems. If the vendor for some reason becomes unsuitable

¹<https://azure.microsoft.com/nb-no/>

²<https://docs.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility#division-of-responsibility>

to use (no longer available, increases prices, etc.), porting code to a new vendor's system would be a significant hurdle.

Specifically with regards to Microsoft under previous management, their past concerning strategies like "*Embrace, extend, extinguish*"^[6] culminating in an anti-trust case with the U.S government³ is not to be ignored.

Under recent management Microsoft has open sourced some of their code portfolio⁴, and licensed them under permissive licenses. This includes the native SQL adapter used for Cosmos DB (used by the project, under a MIT license⁵), and Kudu (used for deployment of the back end, under an Apache-2.0 license⁶).

In a worst case scenario, due to the permissive licenses, modifying the adapter API's, and emulating past services would be possible. Although it would be preferable to build our systems with an additional layer of abstraction, creating a simpler and unified API surface of what we use, and have it utilize our Cloud Service Provider (CSP)'s API's.

Security Concerns

With regards to security, both public cloud platforms and co-location generally provide physical security. With local-hosting, this is something Cryogenetics would have to manage themselves.

Public cloud providers do come with some specific security concerns, especially around insider threats, and leaking information on shared hardware.

Spectre and to a lesser degree Meltdown, pose a threat when software from multiple tenants run on the same hardware. These threats are mitigated by either using dedicated managed hardware (hardware which serves a single tenant only), or running on processors with architectural mitigations in place, such as AMD Epyc CPUs with Secure Encrypted Virtualization⁷ [7]. Noting that the SEV ability to prevent the hypervisor from snooping guest memory is not a significant factor in our threat assessment, as we already assume trust of Azure by using Azure Cosmos to store our data. But it does prevent guests from other tenants from reading sensitive data. Although that does not mean it is foolproof.

Another mitigation would be to use something like Open Enclave⁸ and run in a Trusted Execution Engine, using f.ex Intel's Software Guard Extensions. It would likely be significantly more difficult to implement and maintain, and requires processors to have a TEE. AMD's Secure Memory Encryption also does require specific processors, but does not require changes in how code is written and run. It would also probably be quite interesting to try one day.

³https://en.wikipedia.org/wiki/United_States_v._Microsoft_Corp.

⁴<https://opensource.microsoft.com/projects/>

⁵<https://github.com/Azure/azure-cosmos-dotnet-v3>

⁶<https://github.com/projectkudu/kudu>

⁷<https://www.amd.com/system/files/documents/cloud-security-epyc-hardware-memory-encryption.pdf>

⁸<https://github.com/openenclave/openenclave>

We concluded the security advantages from using SaaS and PaaS on a trusted cloud provider far outweigh the downsides for this specific deployment.

Which CSP

We then identified the most prominent players in the field, which at the time of writing was AWS (Amazon Web Services⁹), Azure (Microsoft Azure¹⁰), and GCP (Google Cloud Platform¹¹). Part of our team had previous experience using Azure, and SharePoint¹² and Azure AD (Azure Active Directory)¹³.

Cryogenetics were already using SharePoint and Active Directory, so some amount of the application would likely need to be integrated with Azure regardless of the choice of CSP. The pricing for the modules we thought we would need were not too far off what competitors offered, based on a rough estimate of about 5000 nok/month in recurring costs (see figure E.1, and about 4000 NOK as a one time license payment for IronXL (the excel parser used).

Azure also has (but is not the only one with) excellent support for C# (as both are maintained by Microsoft), which we wanted to use for the language on our back end.

The factors above: previous experience, SharePoint and active directory already being used, and excellent support of C# within an acceptable budget made us land on Azure to host our project.

4.2 React

Choice of JavaScript Framework

JavaScript, HTML and CSS are the three fundamental technologies in modern web development. HTML is responsible for the structure and semantics for the website, while CSS is handling the setup, colors and other styles. If we want a website with more advanced features and interactions we need to use JavaScript. In recent years, WebAssembly¹⁴ has come to the forefront as a popular addition to JavaScript. WebAssembly is ideal for heavy duty tasks such as game development, AR/VR live applications, platform emulation and image recognition. Since there were no features in our web application that required the performance of WebAssembly, and that none of us have any experience with this before, this became something we chose not to use in the project.

Developing complex websites with standard JavaScript (popularly called VanillaJS¹⁵) is possible, and was for a long period the most common way of developing

⁹<https://aws.amazon.com/>

¹⁰<https://azure.microsoft.com/nb-no/>

¹¹<https://cloud.google.com/>

¹²<https://www.microsoft.com/nb-no/microsoft-365/sharepoint/collaboration>

¹³<https://azure.microsoft.com/nb-no/services/active-directory/>

¹⁴<https://developer.mozilla.org/en-US/docs/WebAssembly>

¹⁵<http://vanilla-js.com/>

websites. However, since 2010 the use of JavaScript frameworks has exploded. The number of frameworks are huge, but they all have in common that they simplify the JavaScript code at the same time as they bring new and powerful functions for the developer to use. No one agrees on which framework is the best to use, so choosing the optimal framework can be hard.

When we did the research in connection with the choice of framework for our website, we emphasized using a framework that was easy to learn, well documented and a framework we really wanted to learn. From before, two of us had experience with LitElement¹⁶, since this was used in the WWW-Technologies course in our 5th semester. We found this framework inconvenient and inefficient to use, and wanted therefore to try another and more popular framework. Our choice was therefore between the three of the most known and used frameworks Vue.js¹⁷, Angular¹⁸ and React¹⁹. At StackOverflow, in 2020 they had a survey among their readers that showed that React is the second most used framework behind jQuery, whereas Angular is number 3 and Vue.js is number 7. When it comes to demanded skills, Geek4Geeks did a study²⁰ on which frameworks are most in demand in working life, and concluded that React is the most demanded skill. This may also indicate that developers don't want to work with React, and that the jobs demanding Angular and Vue.js skills is taken immediately, but since the StackOverflow survey shows high popularity for React we conclude that this is not the case.

The choice ultimately fell on React as they have a large user base that provides many solutions online to problems we ended up in, good documentation that makes it easy to learn, and good support for extensions that we can use in our web service.

About React

React was created by Jordan Walke, a software engineer at Facebook in 2011, and is today maintained as a Open Source²¹ by Facebook and a community of different developers and companies. The framework makes it easy to create web components, which can include and use JavaScript functions and variables. This allows you to call on the components in the same way as you uses the familiar html tags `<div>`, `<p>` and `<h1>`, which easily can and should be reused.

The most important core functions of React for us are:

- Easy to create components, making it easy for us to scale our website
- Passing JavaScript expressions as properties between components, which is important for us when using the Model-View-ViewModel pattern.

¹⁶<https://lit-element.polymer-project.org/guide>

¹⁷<https://vuejs.org/>

¹⁸<https://angular.io/>

¹⁹<https://reactjs.org/>

²⁰<https://www.geeksforgeeks.org/angular-vs-reactjs-which-one-is-most-in-demand-frontend-development-framework-in-2019/>

²¹<https://opensource.org/about>

- Single-Way data flow, keeping our architecture simple
- Powerful State handling, keeping data away from cookies and therefore making our website faster.
- Large number of libraries that can be used
- Easy integration with TypeScript, as we decided to use TypeScript with our website.

React is covered by the MIT License²². More on how we have used React in chapter 6.

4.3 TypeScript

About TypeScript

TypeScript is an open-source programming language designed and released by Microsoft in 2012. It is a superset of JavaScript and provides optional static typing, classes, and interfaces. Since Typescript cannot be run or understood in any browser, it is transcompiled²³ to JavaScript. It can be used as a language on both client-side and server-side when developing applications.

TypeScript has over the past few years become very popular among front end developers. That is because it:

- Provides optional static typing
- Makes it easy to spot bugs early as the compiler will complain if something does not match the typing
- Provides increased predictability and readability
- Provides fast refactoring

Why TypeScript?

Whereas JavaScript is a dynamically typed language, where types are checked and datatype errors are spotted only at the runtime, TypeScript introduces optional strong static typing. This means that once a variable is declared, it doesn't change its type and can take only certain values. Checking the type at runtime may not be a disadvantage, because it offers more flexibility and enabling program component changes dynamically. We considered our program quite large, and we therefore wanted more control over the debugging. TypeScript's static typing leads to the compiler alerting developers to type-related mistakes, preventing bugs from hitting the production phase. Using TypeScript in projects also speeds up refactoring and debugging, and because of the strong static typing it gives the code more structure and makes it more readable. These are qualities we consider important in our relatively large project. TypeScript also works well with React, so using it in our project actually became a no-brainer.

²²https://en.wikipedia.org/wiki/MIT_License

²³<https://www.codemotion.com/magazine/Glossary/source-to-source-compiler/>

4.4 Docker

Docker is an open source containerization platform, which allows developers to package applications into containers²⁴. A container is a standardized executable component that combines application source code with OS libraries and dependencies, which makes the source code possible to run in any environment. In short, Docker is a tool that enables developers to build, deploy, run, update and stop containers using simple commands. For us, using Docker is very important. That is because we use different operating systems (Windows and Linux), and using Docker to containerize a website gives us more predictability as Docker makes our website run the same on all operating systems.

The containers used by Docker offers similar benefits to Virtual Machines (VM)s, which includes e.g. isolation of application (although not to the same degree) and cost-effective scalability. But it also have some advantages over VM's:

- Containers are far more lightweight than VMs, as they only include OS processes and dependencies what is needed to run the code.
- Containers have a greater resource efficiency than VMs, which means that we could run our website much easier on our hardware.
- Containers are, compared to VMs, faster and easier to deploy and restart, which means that we could save some time when developing our website.

More on how we used Docker when developing our website in section 8.1.

4.5 Git

One of the most used version control system in the world today is Git²⁵, originally developed in 2005 by the famous creator of Linux, Linus Torvalds²⁶. All members of this bachelor thesis had an experience in using Git from previous projects, so we did not even entertain the idea of using another system since we all were very familiar with the development process.

One of the biggest advantages of Git, which we used extensively in this project, is the branching capabilities it provides. The front end project was from the start divided into two branches stored in a repository on GitHub²⁷ and several locally stored branches. You can read more about how we used Git and Git branches in section 8.1.

4.6 CSS Grid

While HTML is used to define the structure and semantics of the website's content, CSS on the other hand is used to style and lay it out in whatever way the

²⁴<https://www.ibm.com/cloud/learn/docker>

²⁵<https://www.trustradius.com/version-control>

²⁶<https://www.atlassian.com/git/tutorials/what-is-git>

²⁷<https://github.com/>

developer wants²⁸. One can use CSS to alter the font, color, size, margins, add animations and much more that makes the website more aesthetically pleasing for the user. CSS can also be used to control the web page's layout, where some popular methods are using tables, the box model, and CSS flex box. In recent years, CSS Grid²⁹ has become more and more popular, a method that two of the group's members have been introduced to in the course PROG2053 WWW-Technologies. CSS Grid divides a page into major regions or defines the relationship between HTML classes in terms of size, position, and layer.

We chose to use CSS grid as we thought it was a more clear tool to use than for example Bootstrap. We considered CSS Grid as a more suitable tool to use to create responsive websites that also work on tablets and phones. Another important factor for us when we decided to use CSS Grid was the fact that we wanted to improve our abilities, as CSS Grid is considered by many to be the future of web development³⁰.

4.7 RESTful API

For our API technology, we had two major contenders: RESTful and GraphQL.

REST (Representational state transfer) is an API pattern that makes CRUD (Create, Read, Update, Delete) operations very easy. It is simple to implement, and uses the GET (Read), PUT (Update), POST (Create) and DELETE (Delete) http methods representing the CRUD operations.

GraphQL is a more powerful API query language than REST, but with added power also comes greater complexity to implement.

One main differentiator between REST and GraphQL, is that GraphQL allows you to pick specific fields to fetch, whilst RESTful API's do not have to allow this. Although REST API's can be expanded to allow this, by for example following the OData standard³¹.

We elected to use a simple RESTful api, as our documents are not so large that being able to select specific fields are important. And the added complexity of implementing GraphQL is something we decided was not worth it. We also added ordering, and planned to add filtering (based loosely on OData), as both add value to the the API consumers (especially the front end).

4.8 .NET 5 & C#

.NET 5

.NET 5 had excellent support for creating web-API's through ASPNET Core, as well as being fully supported by most major cloud providers. It is portable and

²⁸<https://developer.mozilla.org/en-US/docs/Learn/CSS>

²⁹https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout

³⁰<https://www.zionandzion.com/css-grid-the-future-of-web-development/>

³¹<https://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.html>

cross-platform, and is fast enough for the requirements of this project. We also had some prior experience using .NET 5, and earlier versions of the framework.

Please note that .NET 5 is the successor to .NET Core 3.1, not .NET Framework. See ³²

Language Features

C# was chosen primarily due to past experience with the language, as well as a good ecosystem for developing websites and web-API's (ASP.NET Core). C# is a high-level, statically typed and jit'ed language, compiled to CIL (Common Intermediate Language).

C# is cross platform, allowing it to run on newer ARM server processors such as Ampere Altra³³, and AWS's Graviton³⁴. Although the performance requirements of this specific project does not require the capability to run on high-performance ARM processors³⁵³⁶, this does not mean that the usually higher efficiency of these cpu's can reduce cost. But as the overall recurring costs for hosting this project are estimated to be reasonably low, this is not a significant factor in this case.

C# also features automatic memory management (specifically, garbage collection) preventing a plethora of potential issues, like accessing reallocated memory, and buffer overflows. Starting from C# 8, there is also support for what is known as "*nullable reference types*", which essentially means that the compiler enforces initialization of values. As a consequence, null checks are not needed, as reference types cannot be null, unless specifically tagged as a nullable value (in which case the program author is responsible for checking if a non-null value has been assigned).

³²<https://docs.microsoft.com/en-us/dotnet/core/dotnet-five>

³³<https://amperecomputing.com/altra/>

³⁴<https://aws.amazon.com/ec2/graviton/>

³⁵<https://www.servethehome.com/ampere-altra-wiwynn-mt-jade-server-review-the-most-significant-arm-server/>

³⁶<https://www.phoronix.com/scan.php?page=article&item=ampere-altra-q80&num=1>

4.9 Technical Memo

Factors	Should we develop a separate iPad application or focus more on in making the website more compatible with tablets and mobile devices?
Discussion	<p>Starting Point:</p> <ul style="list-style-type: none"> - None of us were familiar with Swift or had any experience in developing for iPad and iOS. - We would have to spend a lot of time learning a new platform - None had prior access to the right tool to develop for iOS (Mac). - At least 2 of the developers had prior experience in developing responsive websites <p>Design:</p> <ul style="list-style-type: none"> - Since we decided beforehand in following the Microsoft Fluent UI guidelines, combining that with Apple's guidelines in app development would create unnecessary more work in the design phase. <p>Performance:</p> <ul style="list-style-type: none"> - A native iPad application would perform better than a website <p>Other:</p> <ul style="list-style-type: none"> - Another expenditure from Cryogenetics to pay for a developer account at the App Store - It would be less complex to keep everything in one web application - We estimated we would not be able to create a well functioning website if our resources were split.
Solution	<ul style="list-style-type: none"> - Based on the discussion above, the group decided to not develop a separate application for iPad, but instead focus more on creating the website more compatible with mobile devices. - Cryogenetics agreed with this as they would mostly be using laptops to navigate our system.

Chapter 5

Design

In this chapter we will take a closer look at how we have structured the project. This means that we explain the choice of patterns on the front end, how we have chosen to design the database, and what adjustments have been made for our project.

5.1 Structure

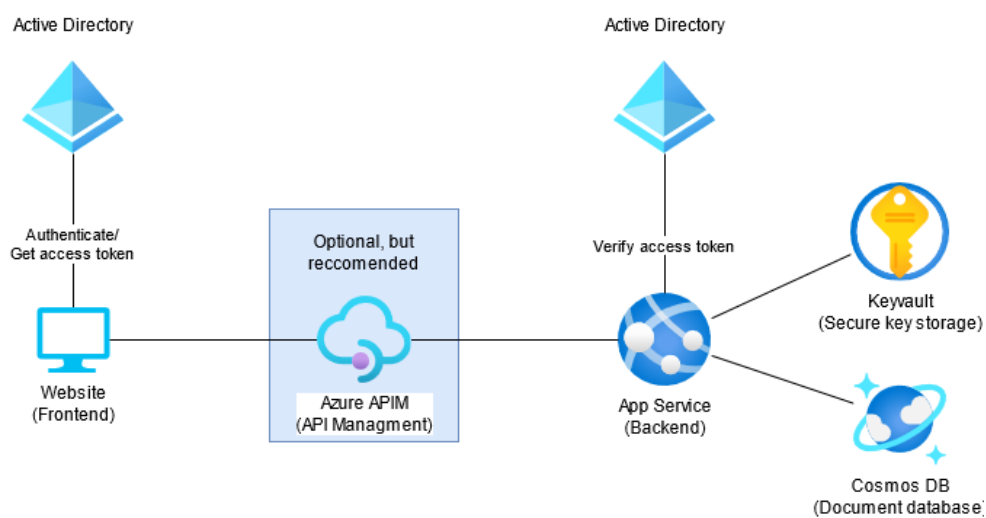


Figure 5.1: Current overall architecture

The current architecture consists of two main modules; a front end (a static Single page application (SPA) webpage), and a back end (hosted as an Azure app service).

They communicate with each other through a simple REST API, where the front end handles displaying data and user interaction. The back end handles abstracting away the complexities of the back end database, and presenting the

data as a simple REST API.

The back end uses the Microsoft Identity platform to validate the access tokens provided. Cosmos DB provides data storage in the form of a document database, and the key-vault provides safe storage of sensitive keys (specifically, the cosmos db connection string, and Azure app registration secrets).

Azure APIM or a similar service such as Azure Front Door is recommended as an endpoint for the back end API, to handle potential Denial of Service (DoS) attacks. The front end SPA webpage can be hosted on any service capable of static file serving.

5.2 Front End Structure

Model-View-ViewModel

At the beginning of the project, we decided to use a Model-View-ViewModel[8] for our web application. This software design pattern was chosen because Model-View-ViewModel (MVVM):

- Makes it flexible to work on both User Interface (UI) design and front end logic near-simultaneously, which was crucial for us as we were often several who worked on the modules of the code.
- Makes it easy to write and test both units and modules. (See section 9.4)
- Provides an easy organization of our components, because we expected to end up with many components.
- Allows you to refactor parts of the code without affecting other parts of the code (See section 9.1)

Figure 5.2 shows how we chose to structure our front end application directory. In the root folder we find in addition to the `src/` folder, all the configuration files. These are configuration files for React, Docker, ESLint and Git. We also find the `package.json` and `package-lock.json`, which contains information about the NPM packages we have used. We set up the project after the MVVM pattern as we have structured as follows:

- **Model** contains all the model files for the web page. Each model file stores all the data for its modules.
- **Pages** contains the pages files for the web page, where each page file works as an entry for the module.
- **View** contains all view files, style sheets and view components.
- **ViewModel** contains all the ViewModel files. These files contains the logic, the communication with the database and provides communication between View and Model.
- **Interfaces** contains the files with the interfaces of the various modules. Each file contains the interfaces of its module, for example: `TankListInterfaces.tsx` contains all the interfaces of the TankList module.

We have tried as long as possible to follow the MVVM the pattern, by storing

data in the model, all functions (both communication to API and View logic) in the ViewModel and keeping the View as free as possible for anything other than presentation logic. But there are several places in our code we did not find this natural. An example of this is for the map of the tanks, where there is a map for 11 Liters, a map for 47 Liters and one for 500 Liters. As described in section 6.4 the three tanks are all using the same Model, View and ViewModel file, which the only difference between the them is the map component itself. The 500 Liter tank differs quite strongly from the other two tanks in terms of how we have built up the component, which means that there are some functionalities needed to show this tank that is not needed for the other two tanks. We have therefore chosen to put the functions and data that explicitly belong to 500 Liter tanks in the component file.

Another place we have done this is in the list of tanks. As explained more in detail in section 6.4, the Tanklist module is built up of `TankListView.tsx` working as the container for the page. This uses the `TankView.tsx` component to render each Tank with its properties as card component¹. As described in section 3.1 it should be possible to edit some of the tank properties. We tried to place the logic behind these functionalities in the ViewModel, but we found this inconvenient as it included unnecessary use of additional arrays and functions. We therefore chose to place the logic in `TankView.tsx`.

Usually when using MVVM in a project, the View is the entry point in the application, as opposed to MVC where it is the controller that is the entry point². When we started developing our website, we were not aware that it was common in an MVVM pattern to use View as an entry point to the application. We therefore started developing with ViewModel as the entry point that initialized the Model and sent the data to View. We think this worked well, and chose to use the same structure of the modules further.

When we became aware that it is common MVVM to use View as an entry point, we thought it was too late to rewrite our code so that the View became the entry point. Such a rewrite would take a lot of time, and be risky considering that there could be some functionalities that would not work. We thought it was not worth spending a lot of time on fixing a system that already worked well just to be able to say afterwards that we followed the pattern to the letter. It can therefore be said that our website also has similarities with a Model-View-Controller (MVC) pattern, as it is the Controller that is the entry point in MVC.

Modules

Our application mainly consists of 8 modules, where 5 of the modules at least each have their own Model / Provider, a ViewModel and a View. The eight modules are:

- Authentication and login page. This is the module that differs greatly from

¹<https://www.nngroup.com/articles/cards-component/>

²<https://www.guru99.com/mvc-vs-mvvm.html>

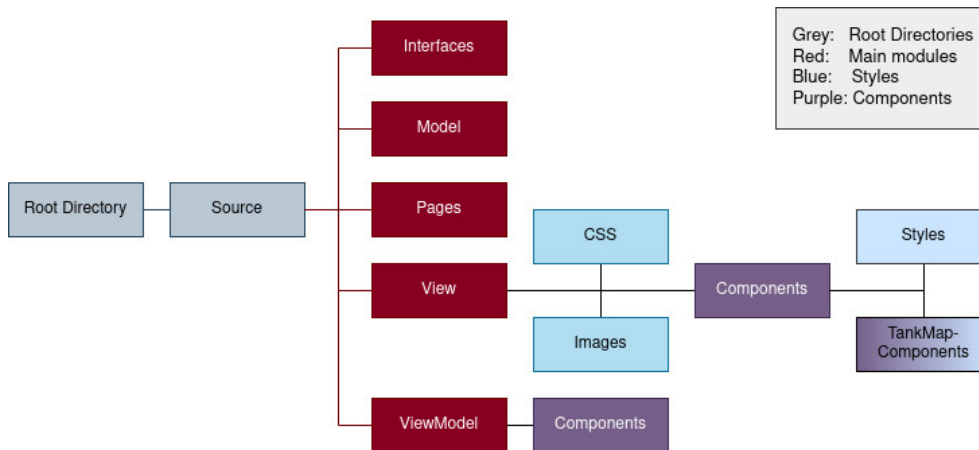


Figure 5.2: Our Front End Folder Structure (created in draw.io)

the others in terms of structure, which is basically due to a lot of communication and authentication with Microsoft.

- Account page. Does not have a ViewModel, but does have a model and a view.
- Backups. Should fetch all the backups from the database, display them as a list for the user and give them the opportunity to download them as an excel document. Everything is implemented on the front end, but lacks endpoints in the back end.
- Pending files. Fetches files that are ready for approval in the SharePoint folder and displays them as a list for the user. Should give the user the opportunity to approve these, but it lacks an endpoint in the back end for this.
- The global search. Should give the user the opportunity to search the entire database, and then show the result of this search. Roughly finished in front end, but missing endpoints in the back end to do a search.
- The homepage. Its the homepage for our website, and thus the page you get to after logging in. Works as a simple dashboard, with some key figures around files, tanks and clients, as well as the clock for the various places Cryogenetics has offices.
- List of tanks. Fetches all the tanks in the database and displays them as a list. For each tank it is possible to change the location, availability and owner properties.
- The map / list of the contents in a tank. Fetches all the fishes in the chosen tank, and presents it as a map of the contents of each tank, as well as what is in the cylinder / slot that the user has clicked on. The content can also be presented as a table. Consists of multiple components for each type of tank, with both TypeScript and CSS files.

5.3 Database design

Initially we had thought to use a SQL RDBMS for our backend database, which we created a rough draft schema for, see F.1. Although after further communication with Cryogenetics, it became clear they wanted further flexibility that originally anticipated.

The data from existing spreadsheets did have a loose schema attached to it, but finding primary keys, and what were valid values and datatypes proved to be challenging.

To allow for the flexibility required, using SQL, we saw two main options:

1. Dynamically add columns to existing tables as they were required, potentially leaving the table sparse. Another augmentation to this would be to have a main data table with existing data, and create a new table for dynamic data. The new data table could have its columns altered as needed. Then a simple join would be performed on the item id.
2. Create a new "additional data" table, where any additional data is stored as "id, key, value". Essentially a key value pair table.

Neither option seemed preferable, so instead we went for a native flexible document database. Using a document database does have inherent issues with our requirements as well, but they can be mitigated.

Issues:

1. *Race conditions*: Documents are atomic structures, which means that documents will never partially be updated. But an issue presents itself if a document is read by two parties, A and B. A and B's documents are then modified in disparate ways, and they are both then pushed to the database. As upserting a document overwrites the entire previous document, either the modifications made by A or B is lost.
2. *Consistency level*: The issue with race conditions are exacerbated by using consistence levels other than "strong"³ (which is not the default consistency level as of writing). The "strong" consistency level guarantees linearity, that is: any read is guaranteed to read the latest commit-ed write. Other consistency levels do not necessarily guarantee this (at least not across multiple regions).

The first mentioned above can be mitigated, by using a transaction log. Instead of updating the database directly, create a transaction log of all updates. Then have a single instance go through transactions and mutate the database accordingly.

This does solve the issue of concurrent changes being lost. It however does create a new issue with regards to performance, but as per the operational requirements, this was not considered to be an issue.

With the added flexibility required, we switched from a Relational Database Management System (RDBMS), to a document database. Specifically Azure Cos-

³<https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels#strong-consistency>

mos DB. The call was made that creating a transaction system (which would have been necessary with RDBMS anyways) was less work than adopting a RDBMS to work with semi-structured data.

For an example scheme of a specific instance of data, see figure 5.3. The scheme is for a specific document of a fish, tank and client. The database contains many of these documents. As the data is semi-structured, the documents do not necessarily have the same properties, but the general structure is currently the same across all documents of a type.

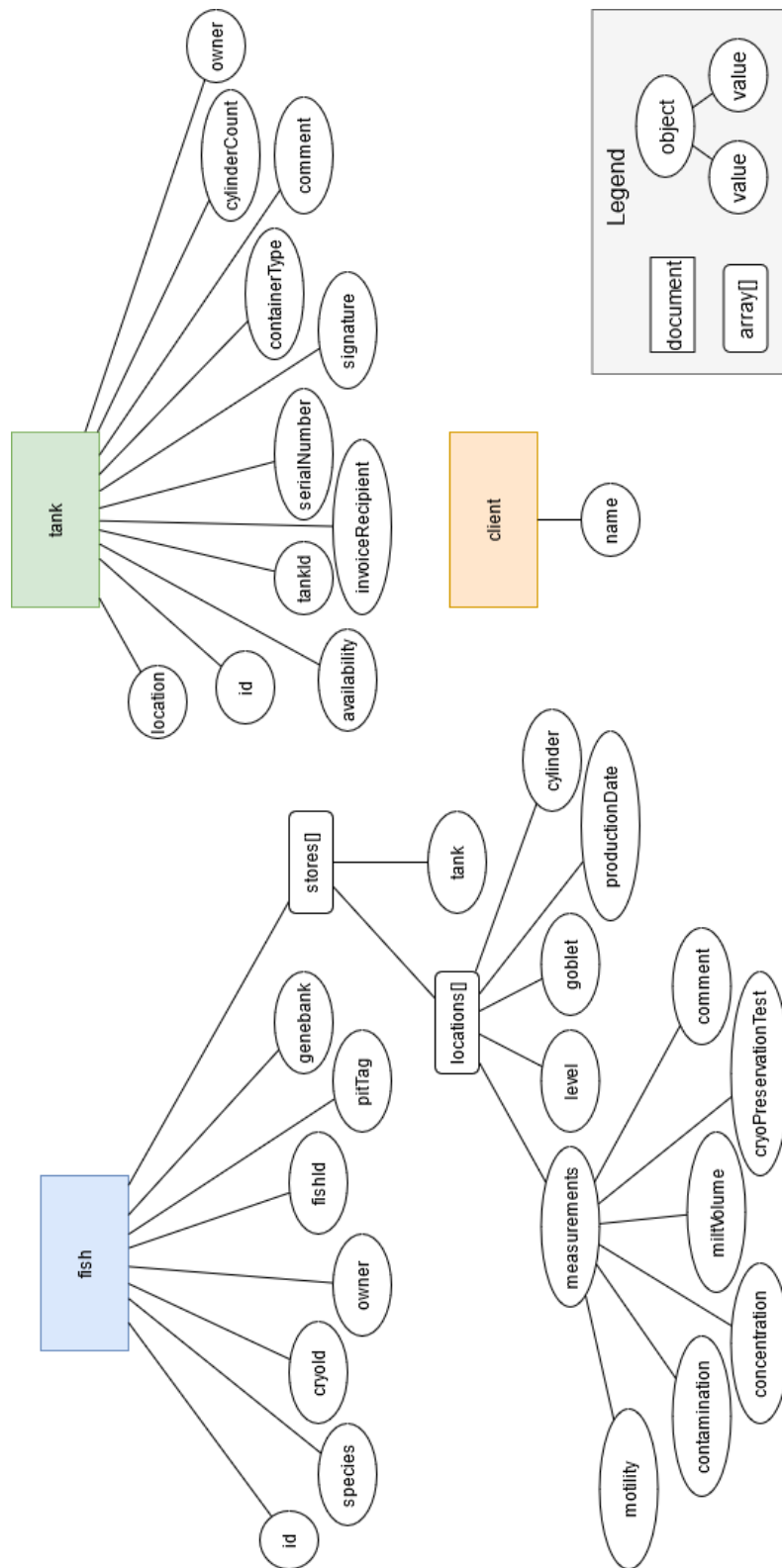


Figure 5.3: Example schema for instances of fish, tank and client documents

Chapter 6

Implementation

Our web application consists of many modules and components, so we therefore choose not to go into detail on all of them. We have chosen to take a closer look at the most essential modules such as routing, navigation bar, map of the content in a tank, login and authentication. We will also take a closer look on implementation of the back end, and the security concerns round this.

6.1 Front End Web Interface

Of the products we have developed during the project period, it's the website that the employees at Cryogenetics see and use directly. This was made with the framework React, which we justified in section 4.2. Our website is inspired by Microsoft's Office Online programs, where *Word Online* is the website we have taken most inspiration from. In any case, we have tried to keep the website as simple as possible, and it then consists mainly of the modules:

- **Navigation bar** (see section 6.3) - No matter where the application is, you will always find it at the top of the screen. Contains functionalities for navigating around our website.
- **Content area** - The area of the screen that displays the content the user wants to be displayed.

On larger screens, there will be margins on each side of the content area. Since we have created a responsive web page (see section 7.4), these will be adjusted automatically based on the screen size, and thus ensure that the web page layout is intact. See figure 6.1 for an overview over layout.

Components

You can easily create a working and good website by just using React, but by using a UX Framework you can save a lot of time while getting a better result. A UX framework is a framework that offers stylesheets and control components,

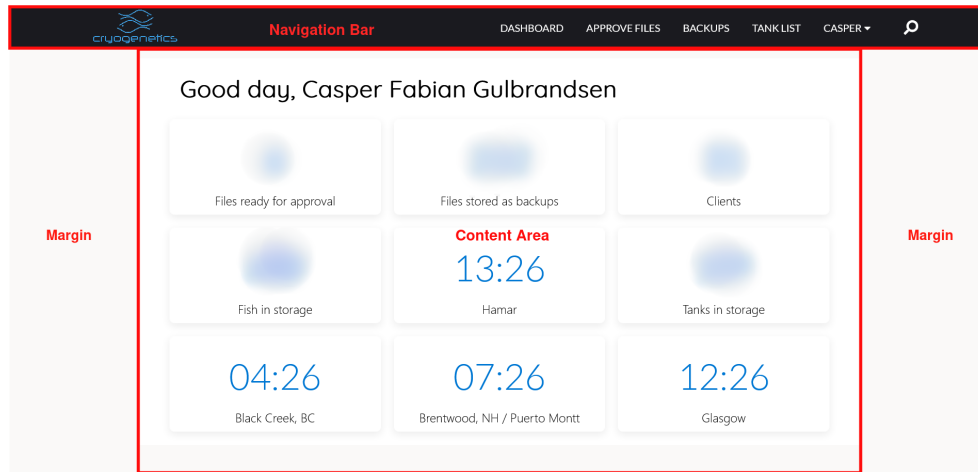


Figure 6.1: Front end basic layout and areas

which can easily be imported into the project and used on an equal footing with our own components.

As described in section 7.1, we chose to use the UX framework Fluent UI[9] when we developed our website. Fluent UI made it easier for us to design our program according to Microsoft's recommendations. This is because in Fluent UI there are different styles that we can use, such as standardized colors, icons and typography. Fluent UI also has different controller components we used, where we used basic inputs as buttons, links and dropdowns. We also used progress indicators and spinners for when content is loading, dialog for getting confirmation from user and menu items like the pivot controls and overflow set. In code listing 6.1 we see how we used the pivot control component from Fluent UI. By importing them to the current file, we used them in the same way as the regular HTML tags in addition to sending the correct properties to the component.

```

1 import { IPivotStyles, PivotItem, Pivot }
2 from "office-ui-fabric-react/lib/Pivot";
3 ...
4 <Pivot
5   aria-label="Pivot Controls"
6   selectedKey={props.selectedTab}
7   onClick={props.handleClick}
8   headersOnly={true}
9   getTabId={getTabId}
10  styles={pivotStyles}
11 >
12   <PivotItem headerText="All" itemKey="all" />
13   <PivotItem headerText="My Recent" itemKey="myrecent" />
14 </Pivot>

```

Code listing 6.1: Using Fluent UI components**Structure**

You can from figure 6.2 see how our front end application is structured.

- *index.html* - the page template. Created automatically by React when setting up the project, and contains imports of stylesheets and a div which works as the entry point for the *index.js*.
- *index.js* - the JavaScript entry point. Created automatically by React when setting up the project. Gets the root div from *index.html* and renders *App.tsx*.
- *App.tsx* - The component that controls our entire application. Here is the router and protected routes (section 6.2), and authentication (section 6.6). Ensures that navigation bar is rendered everywhere in the application.
- *AuthProvider.tsx* - Responsible for authenticating the user trying to log in, and making sure to properly log out. Also responsible for fetching new OAuth and UserOAuth keys when these expire (see section 6.6).
- *Main modules* - the blue rectangles in figure 6.2. The modules responsible for providing the content on the website (what each module is responsible for is described in section 5.2).

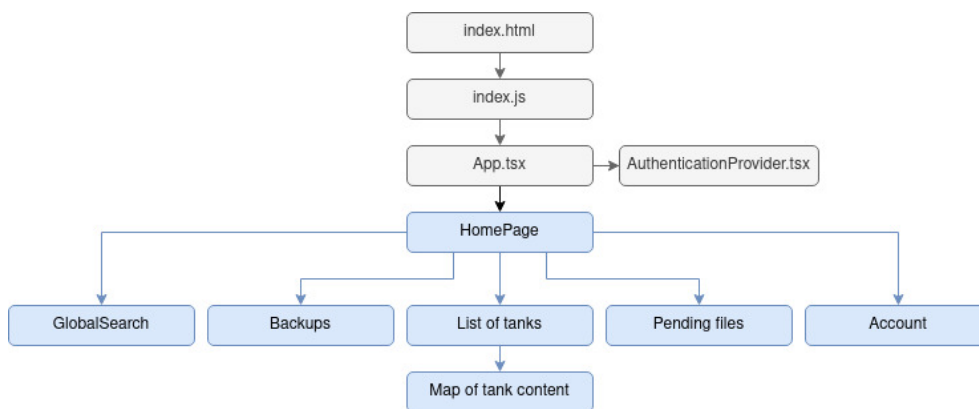


Figure 6.2: File structure - front end (made in draw.io)

6.2 Routing

Routing is the capacity to show different pages to the user. To move between different parts of the website with the URL bar or by clicking on an element¹, like the ones we have added in the navigation bar. Rather infamously, React comes

¹<https://www.freecodecamp.org/news/a-complete-beginners-guide-to-react-router-include-router-hooks/#what-is-routing>

without routing pre-installed, so we decided to use a library called `react-router`, which is currently the most popular routing solution in React development².

React router is a tool that allows you to handle routes in a web app, using dynamic routing³ which means that routing takes place as the app is rendering. Dynamic routing is the way to trigger the contents of the application to render on screen. The opposite would be static routing which is when you declare routes as part of the app's initialization before any rendering takes place.

We researched alternatives to `react-router` as well, like for example `Wouter`⁴ which is a more minimal version of `react-router`. In the end we decided to use the most popular one because of increased access to documentation.

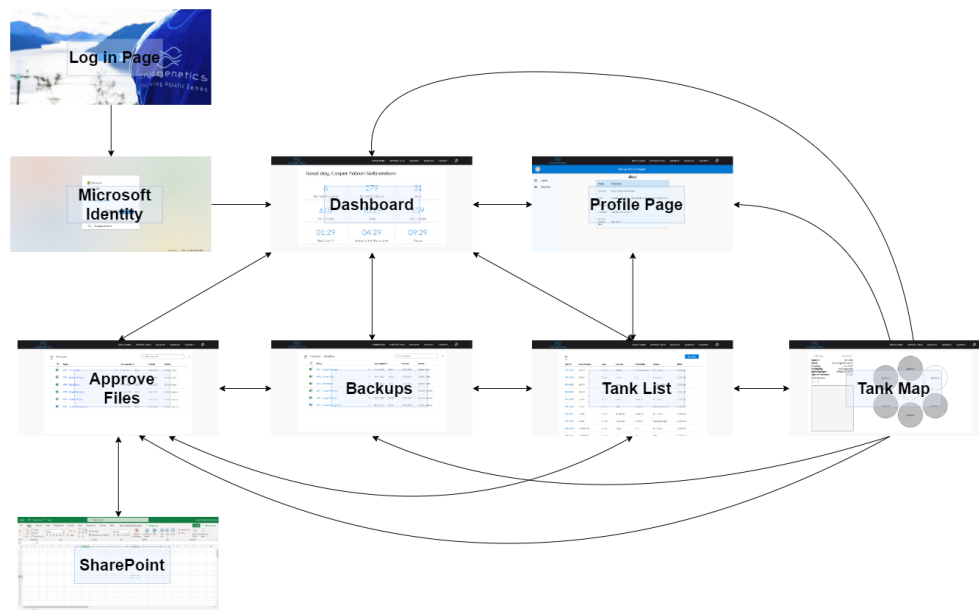


Figure 6.3: Front End Routing Structure

Routing Structure

Figure 6.3 displays in a flowchart how the web page routing structure is designed. With this design in mind, we wanted to provide a structure that had little to no dead ends in the sense that the user should not have to go back a page to navigate to another. Every main page can be accessed from each other, except for the individual tanks, which can only be accessed from the tank list.

²<https://reactrouter.com/>

³<https://www.educative.io/edpresso/what-is-a-react-router>

⁴<https://blog.bitsrc.io/wouter-a-minimalist-alternative-to-react-router-2756690c2b77>

Route vs Protected Routes

The user can switch between the various routes (e.g. /home or /approvable) by using the navigation bar (see section 6.3) or by changing the URL in the browser. The routing of our application is twofold. The first part deals with which URL the user has entered, where it either sends the user to the login page, to a 404 page, or to the ContentContainer. This router part is shown in code listing 6.2, where we can see how it handles the different cases.

- If the path after the URL prefix is '/', the code is rendering the LoginContainer.
- If the path matches one of the keywords on line 4 and 5 in code listing 6.2, the ContentContainer with the protected routing is rendered.
- If the path does not match any of the keywords, the component NotFound will be rendered.

```

1 <Switch>
2   <Route exact path="/" component={LoginContainer} />
3   <Route path=
4     "/(home|approvable|backups|file|account|tankmap|tanks
5     |account/about|account/business)(/?.*)"
6     component={ContentContainer}
7   />
8   <Route path="/notfound" component={NotFound}/>
9   <Route component={NotFoundHandler} />
10 </Switch>

```

Code listing 6.2: Main Routing

If the user has a valid URL, part two of the routing is rendered. From code listing 6.3 we can see an excerpt of the code, where we can see that it uses ProtectedContainerRoute components rather than the Route.

```

1 <Switch>
2   <ProtectedContainerRoute exact path="/home" component={() =>
3     <Home name={currentUser
4       ? currentUser.name
5       : "USERNAME"}
6     />
7   }/>
8   ...
9   <ProtectedContainerRoute
10    exact path="/backups" component={Backups}
11  />
12  ...
13  <Route component={NotFoundHandler} />
14 </Switch>

```

Code listing 6.3: Using Protected Routes

ProtectedContainerRoute is a self-written component that will take the user back to the login page if they are not authorized. If the user is logged in, it will render the correct component. From code listing 6.4 we can see this component. For the function to work as intended, it needs to import Route and Redirect from React-Router-Dom, and AzureAuthenticationContext from our AuthenticationProvider. As we can see from code listing 6.3, the component takes three parameters:

- *component* is the component that will be rendered if the user is authorized.
- *exact* is a boolean that says if the path should match the URL exactly.
- *path* the string that stores the path used by Route, in the same way as described in code listing 6.2.

Our component returns a Route, just as we did on part 1 of the routing (See code listing 6.3. On line 11 we say (...rest), which means that we send all properties except the Component. In our case that means exact and path. What sets this route apart from previous example, is that we use render instead of Component. Using render allows us to use convenient inline rendering and wrapping without explaining the undesired remounting before using it. This means that we can check against AzureAuthenticationContext whether the user is actually logged in, and therefore choose what to render based on this. If the user is logged in, the component sent to the function is rendered with its properties. If the user is not logged in, it is redirected back to the login page on URL '/'.

```

1 import { Route, Redirect } from "react-router-dom";
2 import {
3   AzureAuthenticationContext
4 } from "../Model/AuthenticationProvider";
5
6 export const ProtectedContainerRoute =
7   ({ component: Component, ...rest }:
8     { component: any, exact: boolean, path: string }) => {
9   return (
10     <Route
11       {...rest}
12       render={props => {
13         if (AzureAuthenticationContext.Singleton.
14           isLoggedIn()) {
15           return <Component {...props} />;
16         } else {
17           return (
18             <Redirect
19               to={{

```

```
20         pathname: "/",
21         state: {
22             from: props.location
23         }
24     }}
25 />
```

Code listing 6.4: Protected Routes Component

6.3 The Navigation Bar

A staple in modern user interfaces, the navigation bar was the very first component we created for the web application. It was intended to be the main form of navigation on the website since it is ever present at the top no matter which page the user is currently on. In section 7.2 we talk more about how it affects the user interface and navigation, while here we will take a deeper dive in how it was implemented.

The Navbar component's HTML is fairly straightforward with an encasing `<nav>` tag surrounding the entire component, with five `<Link>`-tags which makes up the links that the user can click on to navigate the website. One of the `<Link>`-tags are represented by the company's logo and will always take the user back to the front page.

The other 4 links are all defined by their `className` `nav-item` which is the way you organize the various HTML-tags so you can define a common style in the style sheet later with CSS. `nav-item` routes the user to the various pages (see section 6.2). Next to the 4 links we implemented the navigation bar to dynamically display the current user's name. This looks similar to the other links, but it acts differently when the user interacts with it. Clicking on the username, the page will render another component which is the Dropdown. This contains a link for the user to log out, and also a route to the user's account page. Using a drop down menu for this was decided because adding more than 5 links on the navigation bar would make it look very crowded, without sacrificing visibility by making the font-size smaller. The drop down menu also made it easy to add more links should it be necessary.

The graceful part of the navigation bar appears when the user access the website on a mobile device or resizes the window. It will automatically move the links to a side menu which is accessed by pressing the hamburger menu button that appears in their stead (see figure 7.9 and figure 7.11). This was implemented with CSS media queries and the react hook `useState`.

```
1 const [isSideMenu, setSideMenu] = useState(false);
2 const handleMenuClick = () => setSideMenu(!isSideMenu);
3 ...
4 <div
```

```

5     className="menu-icon"
6     onClick={() => {
7         handleMenuClick();
8         setDropDown(false);
9         closeOverlay();
10        }}
11 >
12     ...
13 <ul className={isSideMenu ? "nav-menu active" : "nav-menu"}>
14     ...
15 </ul>

```

Code listing 6.5: Side Menu Code Example

The way we implemented the side menu was by using the boolean const we called `isSideMenu`, which decides what the class name of the unordered list tag `` will be, as seen on line 13 in code listing 6.5. By having different class names in different boolean states, allowed us to use different styles in the CSS while it being true as opposed to it being false. The way the state changes is by clicking on the `div menu-icon`, which is only rendered when the window reaches below 1367 pixels as per the media queries in the CSS style sheet, thus changing the view to the so called tablet version.

We also added a check for each time something is clicked if the side menu or the drop down menu is open or not, adding some quality of life effects to using the navigation bar. Clicking a link in the side menu for instance will route you to your desired page and also close the side menu for you, so the user does not have to add more clicks to get where they want. An issue arose for this check with the custom `DropDown` component we made, in that it did not initially have a native `"onClick"` property. This was solved by adding a custom property to the component (see code listing 6.6 on line 1 and line 9) so the `<dropDown>` component would share the same properties as a `<button>` tag.

```

1 function Dropdown({ ...buttonProperties }) {
2     ...
3     return (
4         <>
5             <ul
6                 onClick={handleClick}
7                 className={isDropDownMenu ?
8                     "dropdown-menu clicked" : "dropdown-menu"}
9                 {...buttonProperties}
10            >
11                ...
12            </ul>
13        );
14    };
15 }

```

```
14 // (Navbar.tsx) //
15 {isDropdown && (
16     <Dropdown
17         onClick={() => {
18             dropDownClick();
19             sideMenuCheck();
20         }}
21     />
22 )}
23
```

Code listing 6.6: Adding properties to custom components

6.4 The Storage Tanks

Cryogenetics store the aquatic milt in large storage tanks. They wanted us to develop a way for them to view the contents of their tanks in a quick and easy way. Since they already used a map of the tanks for their work, we decided with them to replicate this process on the website with a similar interactive map and later a simple table view of the tank's contents.

The list of tanks, the list of backups and the list of the pending documents are very similar, where it is mostly just the actual information card and somewhat different functionalities over the actual list that separates them. We will therefore only go into one of them, as it would be very similar. We have chosen to take a closer look at TankList, as we think this makes sense because we also go further into detail on TankMap and TankTable. Another important reason is because it has some functionalities that are not found in the two other lists.

Tanks List

Each time the list of tanks is loaded, the LoadData function in TankListViewModel fetches the first 20 tanks from back end. As we can see in code listing 6.7, the map is wrapped inside a InfiniteScroll component. Infinite scroll is a technique where more content automatically loads as the user scrolls down the page, which eliminates the user's need to click to the next page. As there exists multiple third party packages which works with React, and we do not see the need to spend time needlessly developing our own version, we decided to use a component called *React Infinite Scroll Component*⁵. Worth noting in this snippet is the line 3, where we send our LoadData as a property, which means that every time we reach the bottom of the page, LoadData fetches another 20 tanks and appends them to the array of tanks. On line 12 to 14 we can see how the code maps through the array of tanks, and send its data to the TankView component. This component then creates a card of the tank data and displays this on the screen.

⁵<https://www.npmjs.com/package/react-infinite-scroll-component>

```
1 <InfiniteScroll
2   dataLength={tanks.length}
3   next={props.LoadData}
4   hasMore={true}
5   loader={<ProgressIndicator .../>}
6   endMessage={
7     <p style={{ textAlign: "center" }}>
8       <b>All tanks loaded</b>
9     </p>
10    }
11  >
12  {tanks.map((item: any, i: number) => (
13    <TankView .../>
14  ))}
15 </InfiniteScroll>
```

Code listing 6.7: Inifite Scroll

An excerpt of the `LoadData` function is shown in code listing 6.8. The function is asynchronous (see section 6.4 for further explanation on async functions), and as fetching from the API is somewhat slow, we must therefore wait for the function to finish (which we do using the `await` operator). The function contains a nested function `getData`, which is called on line 30. As the `getData` function is also asynchronous, the main function waits for `getData` to finish before it moves on.

Inside `getData` the first thing that is done, is to construct the current URL. This is done because we used a paging system in the back end, and we therefore needed to update our endpoint URL for each fetch to use the next page token. We also specified in this URL that we want to fetch 20 tanks at the time, sorted by Tank ID ascending after request by Cryogenetics. Furthermore, the tanks from the back end are fetched with our API function (see section 6.4), which is stored in the `const ids`. We then map through this array, to distinguish the objects that contain invalid data that would not be able to be displayed in the list. From the fetch is the next page token included, and this is stored in state and ready to be used next time `LoadData` is called.

When we fetch the tanks from back end, we get properties of that tank including the id of the client who owns that tank. For the employees at Cryogenetics, it is not very informative to see what ID the clients have in the back end, so they wanted the name of the client to be displayed. This is done by mapping through the array `ids`, where the raw tank data is stored. The map is made asynchronous, as we need to fetch data for each loop. For each loop we also need to construct a URL, as this contains the client id from the first fetch. As we can see on line 17 and 18, we take into account that some of the data does not have a valid client id (e.g. zero) and we therefore replaces this with the id for 'no owner'. After the URL is

constructed, we fetch the client's name from the API. As the result from this map function is stored in an array named data, we can return the data as an object for each loop which is then stored in the array. This means that we can format the data from back end at the front end as we wish, e.g. naming the properties as we want.

After the getData function is finished, the data is concatenated to the list of tanks. tanksLoading is also set to false, which signals to the View that the loading of new data is complete and ready to be displayed.

```
1 const LoadData = async () => {
2   async function getData() {
3     const endpoint = '?PageToken=' + nextPageToken
4       + '&MaxItems=' + Number(maxItems)
5       + '&Sort=ASC%28Properties.Tank%20ID%29';
6     const ids: any =
7       await FetchFromAPI('tanks', 'GET', endpoint);
8     ids.items.map((item: any, i: number) => {
9       if (!ids.items[i].Properties) {
10        ids.items.splice(i, i);
11      }
12    })
13    setNextPageToken(ids.nextPageToken);
14    const data = Promise.all(
15      ids.items.map(async (i: any) => {
16        const clientsEndpoint = '/'
17          + (i.Properties['Owner'] !== 0
18            ? .Properties['Owner']
19            : 'eefd4dee-db90-4ac8-a08c-d6f7ed76e7fc');
20        const owner: any =
21          await FetchFromAPI('clients',
22            'GET',
23            clientsEndpoint);
24        return {
25          "tid": i.Properties['Tank ID'],
26          ...
27          "owner": owner.Properties.Client,
28          ...
29        }
30      })))
31    return data
32  }
33  getData()
34  .then((data: any) => {
35    setTanks([...tanks.concat(data)]);
36    setTanksLoading(false);
```

```

37 |     })
38 | }

```

Code listing 6.8: Fetching all tanks

When a user clicks on a tank in the list, it is forwarded to a new page where you can see the contents of this particular tank (see section 7.3). The way this is done is by using the react hook `useHistory` from `react-router-dom`. This gives us access to the history stack⁶, which we use to navigate on our page. We can therefore use our history object and its function `push()`, to route to a new specific URL. This contains a parameter `tid`, which is the id for the selected tank in the back end. Code listing 6.9 shows how this is done, where `id` is the id of the tank.

Code listing 6.9: Redirecting to new page

```

history.push("/tankmap?tid=" + id);

```

The page for a particular tank consists of three components. One component that shows the properties of that particular tank, one component that shows the content of the cylinder/spot clicked on by the user in the map, and one component that shows the content in the tank.

The contents of the tank can be seen in two different ways, a view where the content is:

- represented as a map that mimics the tanks (see section 6.4)
- represented as a vertically scrolling table (see figure 6.5)

Tank Map

Once the page of a given tank is loaded, the first thing that happens is that the properties of this tank is fetched from the API using our function described in section 6.4. These are the same properties that the user can see in the list of tanks. When these properties are fetched, the loading boolean is set to false and the data in this component is displayed. Then, all the fish in the selected tank are fetched from the API. We have three different types of tanks, but the 47 Liters and 11 Liters are pretty much the same, which means that we can act on whether the tank is a 500 Liter or not. The implementation of the three tanks are relatively similar and we therefore choose not to go into each of them carefully. Since the 500 Liter tank is the most advanced, we have chosen to take a closer look at this.

The map has a darker color in the spot if there are fish in it (see code listing 6.10). When the user click on a spot in the map, information about which fish and some of its properties should be shown on the screen. We therefore came to the conclusion that there were two ways this could be done:

1. Construct four three-dimensional arrays that is directly sent to each section in the map component. X would then be the level, Y would then be the spot in section and Z would have been any more instances in that spot. When

⁶<https://reactrouter.com/web/api/history>

clicking on a spot, the information from these arrays would be displayed. This is considered to be the most effective.

2. Load all the fish data in the tank, and use this to construct a boolean array which contains the status of whether each spot has content or not. When clicking on a spot the original array will be copied before it is filtered, so that only the fish located in the given spot are left. This array is then displayed. This is considered to be the easiest to implement.

We chose to go with option number 2, as we did not have unlimited time, and we reckoned that the time saved could be used for better purposes.

As option 2 describes, the first thing to do was to fetch all the fish in the data and store it in the state (see line 1 to 3 in code listing 6.10). Further we need to construct a one-dimensional boolean array, where we use a 4 digit index system to say where the tank has content or not. The first digit in the index represents the level (1 or 2), the second digit represents the section (1 to 5 where 5 is center) and the two last digits is the spot in the section (1 to 29). When we map through the array of fishes (line 9), we take its level, section and spot and add these together as a string and then convert this string to a number. This number is the index in the array for this fish, and on line 12 to 14 we set the position in the location array to true.

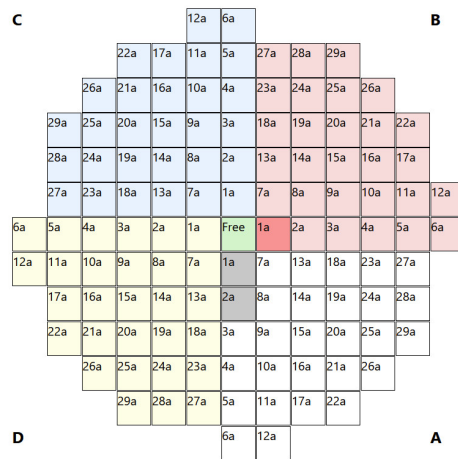


Figure 6.4: 500 Liter Tank

```

1 const endpointFish = '/' + tid + '/fish?MaxItems=100';
2 const fishRes: any = await FetchFromAPI('tanks',
3                                     'GET',
4                                     endpointFish);
5 fishRes.items.map((item: any) => {
6   if (type !== '500 Liter') {
7     ...
8   } else {
9     item.Properties.stores[0].Locations.map((i: any) => {
10      objarray.push({...});
11      const level = (i.Level === 'a' ? 1 : 2);
12      loc[Number(String(level) +
13              String(i.Color.charCodeAt(0) - 64) +
14              String(i.Box))] = true;
15    })

```

```

16     }
17   })
18   setLocationInfo(loc)
19   return objarray;

```

Code listing 6.10: Display a tile in 500L tank

When the whole array is mapped through, the location array is sent to the component that displays a section on the map. This component is called four times each with its own CSS class, which ensures that the layout is as it is. A small excerpt from this component is shown in code listing 6.11. Here we can see that we map 29 times, and for each iteration we calculate the index for this tile (a spot on the 500 Liter tank). We then return the HTML for this tile, where the style property decides whether the tile has content or not. The constructed location array is then used to check if there is content in this tile. If it is, then we use the function `getActiveButtonStyle()` to get the right color for our tile. If it is empty, we use the `getInactiveButtonStyle()` function to get color for our tile.

```

1  [...Array(n)].map((item, i) => {
2    const tileID = Number(String(props.level === 'a' ? 1 : 2)
3      + String(i + 1)
4      + String(props.section));
5    return (
6      <div className={'tile500 a' + Number(i+1)}
7        style={props.locationInfo[tileID]
8          ? getActiveButtonStyle(props.color)
9          : getInactiveButtonStyle(props.color)}
10       key={i}
11       onClick={()
12         => props.handleTileClick(i+1,
13           props.level,
14           props.section)}
15     >
16       {i+1}{props.level}
17     </div>
18   )}
19 )}

```

Code listing 6.11: Fetching map data from API

When a tile is clicked on, we use a self-written function to filter out the data that should be displayed. As we can see from code listing 6.12, the function takes the area, level and section of the tile that is clicked. The first thing that is done, is to set the selected tab in the information box to 0. This means that every time a new tile is clicked, page 1 in the information box will always be displayed. Then we convert the section from number to characters because the section is stored as a character in the back end. We can now map through and filter the array

with tank content, and remove every fish item from the array that does not match with the selected tile. After this is done, the filtered array is displayed by using `setSectionProperties`.

```
1 const handleClick = (area: number,  
2     level: string,  
3     section: number): void => {  
4     setSelectedTab(0);  
5     let charsection = '';  
6     switch (section) {  
7         case 1: charsection = 'A'; break;  
8         ...  
9     }  
10    const list: any = locations.filter((item: {  
11        color: string | string[],  
12        box: string | string[],  
13        level: string | string[] }) =>  
14        item.color.includes(charsection) &&  
15        item.box.includes(area.toString()) &&  
16        item.level.includes(level)  
17    ).map((filteredItem: any) => ( filteredItem ));  
18    const obj = [...list];  
19    setSectionProperties([...obj]);  
20 }
```

Code listing 6.12: Handling of tile click

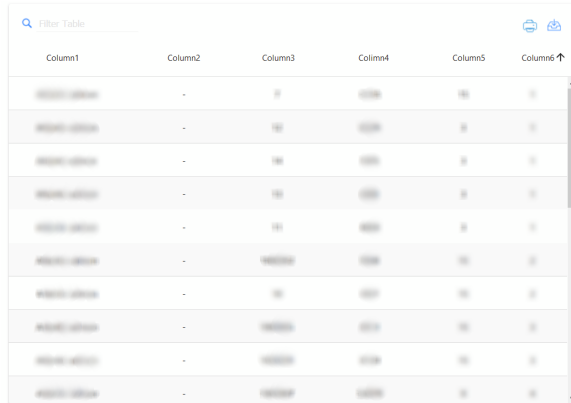
Tank Table

In addition to the map, Cryogenetics also expressed late in the development process an interest in also including a simple table of the tank's content. As mentioned it was towards the end of the development so us making our own table component could prove difficult in regards to time restrictions. We decided it was possible to implement after finding a ready made library called `react-data-table-component`⁷, which fit perfectly to Cryogenetics' wishes and proved easy to use.

There are multiple React table libraries available, but we discovered that many of them required a lot of customization. They were also missing some key features like for example built in sorting and filtering. The library we chose was extremely easy to use along with our API since it only has to read JSON formatted data. All we had to do was attach the correct JSON identifier to the correct column in the table. An example of how this component turned out is displayed in figure 6.5. The data itself has been blurred due to our NDA, but the figure gives an idea of how it turned out.

⁷<https://www.npmjs.com/package/react-data-table-component>

Cryogenetics also asked for a way for them to quickly print out the table itself, which might be handy if the employees have to bring the content with them in a place where it is inconvenient to bring a PC or tablet. This was a major reason we chose the react-data-table-component, because there exists an extension library for it called react-data-table-component-extensions⁸. This allowed us to easily add a print functionality, which used the browsers internal print function, and formatted the table into a readable document. As a bonus, the extension library included a way for the user to save the table as an excel document or CSV file. It also has a text field for filtering the table.



Column1	Column2	Column3	Column4	Column5	Column6 ↑
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6

Figure 6.5: Tank table

Location Designation

For a 500 Liter tank, Cryogenetics has a standardized designation of how the fish are positioned. If a fish is located in spots 1, 2, 3 and 4 at level a in section A, this fish will have a location designation A1a - A4a. Cryogenetics wanted to have this designation in the table view, as they are familiar with this. Each 500 Liter tank has 258 spots, and by showing one row in the table for each spot, the table would become very large and not very useful. As the fish often is located over multiple spots, the designation used by Cryogenetics will therefore reduce the number of rows in the table.

When fetching the tank data from the back end, we get a tank centric array in return. This means that the array says which fish is located in each spot, and not in which spots each fish are. We therefore created a function that takes this array and converts into a fish centric view, with a location property that uses the Cryogenetics designation. We would here like to clarify that if a fish of same species has locations that is not coherent, this species will appear in several rows in the table.

An excerpt of the function that does this operation is shown in code listing 6.13, and takes the array of the tank content as a parameter. The first thing that is done is to sort the array first by the property box, then by color and then by level. This is done so that we can afterwards split the arrays up, first by level and then by section (property color). The array is split up by level by using the JavaScript array function `filter()`⁹, and put in separate arrays. When

⁸<https://www.npmjs.com/package/react-data-table-component-extensions>

⁹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Object

splitting the array by section, we send each of the arrays to a self-written function `splitByColor`. This function maps through each fish in the received array, and pushes the fish to the correct index in a two-dimensional array based on its section. Since the section property `color` is a capital letter, we use the unicode of this letter to get its index (A is 0, B is 1 etc.). The two two-dimensional arrays (one for each level) is then returned and stored in a temporary three-dimensional array `newArray`. This array is then, on line 26, flattened to a two-dimensional array using the array function `.flat()`. We have now an array where each row contains one specific species. We can use this to construct the Cryogenetics designation by using the first element (from) and last element (to) in each row. On line 30 to 44 we map through this array row by row. For each row we construct the location from and location to, before we take all the properties of the fish, as well as adds the final designation as the property `location`. This new fish object is then pushed to the array `returnedArray`, which is returned when each row is mapped through.

```

1 const sortAndFilter = async (array: any) => {
2   const returnedArray: any[] = [];
3   const splitByColor = (arr: any[]) => {
4     const newArr: any = [[], []];
5     arr.map((item: any) => {
6       newArr[item.color.charCodeAt(0) - 65].push(item);
7     })
8     return newArr
9   }
10
11   array.sort((a: any, b: any) => a.Box > b.Box);
12   array.sort((a: any, b: any) => a.color > b.color);
13   array.sort((a: any, b: any) => a.level > b.level);
14   const newArray: any = [[[]], [], [], []];
15
16   newArray[0] = array.filter((item: any) =>
17     item.level.toString().toLowerCase() === 'a'
18     ).map((filteredItem: any) => ( filteredItem ));
19   newArray[1] = array.filter((item: any) =>
20     item.level.toString().toLowerCase() === 'b'
21     ).map((filteredItem: any) => ( filteredItem ));
22   newArray[0] = splitByColor([...newArray[0]]);
23   newArray[1] = splitByColor([...newArray[1]]);
24
25   const filteredArray: any =
26     newArray.flat().filter((element: any) => {
27     return element.length !== 0;

```

ts/Array/filter

```

28     });
29
30     filteredArray.map((item: any, i: number) => {
31         const locationFrom: string =
32             item[0].color.toString()
33             + item[0].box.toString()
34             + item[0].level.toString();
35         const locationTo: string =
36             item[item.length - 1].color.toString()
37             + item[item.length - 1].box.toString()
38             + item[item.length - 1].level.toString();
39         const obj = {
40             iid: item[0].iid,
41             ...
42             location: locationFrom + " - " + locationTo,
43         }
44         returnedArray.push(obj);
45     })
46     return returnedArray;
47 }

```

Code listing 6.13: Constructing fish location in tank

Fetching from API

We decided early in the project period that we wanted to create a standardized function that we could use for both GET requests from the API, while also for PUT, POST and DELETE to the back end API. We wanted to create this as a function we could import in the ViewModel files, and it was therefore placed in the components folder inside the ViewModel folder. In code listing 6.14 we can see the function which was made.

For the function to work, we needed to do two imports. The first import was the url prefix from App.tsx, which is the first part of the url where the API is running on. The second import we had to do was the AzureAuthenticationContext from the AuthContextProvider. This is used for getting the OAuth key from Azure, which is discussed more in section 6.6.

```

1 import { urlPrefix } from '../App';
2 import
3     AzureAuthenticationContext
4 from "../Model/AuthenticationProvider";
5
6 export async function FetchFromAPI(
7     category: string,
8     method: string,

```

```
9     epoint: string,  
10     obj?: any)  
11 {  
12     const url = urlPrefix + category + epoint;  
13     const ids = await (  
14         await AzureAuthenticationContext.Singleton.  
15             fetchAuthorized(url, {  
16                 method: method,  
17                 mode: 'cors',  
18                 headers: {  
19                     'Content-Type': 'application/json',  
20                     'Origin': 'http://localhost:3000'  
21                 },  
22                 body: JSON.stringify(obj),  
23             })  
24     )  
25     return (method !== 'DELETE' ? ids.json() : ids);  
26 }
```

Code listing 6.14: Function used to fetch from API

As we can see from line 6 in the code listing, our API function is asynchronous¹⁰. This means that it does not block script execution, and things like UI updates can happen in the background. If we call our function as shown in code listing 6.15 we can put `await` in front of our function call, allowing us to wait asynchronously for the function to finish executing. For longer executing functions, this lets us keep the site responsive, as we do not block the primary thread for an extended period of time.

The required parameters that we must send to the function each time it is run are:

- Category - Which endpoint category it should be fetched from. As in code listing 6.15 we are fetching from tanks, where we are getting all the fish in one tank.
- Method - HTTP method (GET, POST, PUT or DELETE).
- Epoint - The final endpoint, where it can be specified whether to sort, retrieve something with a given ID and which page you are on. In code listing 6.15 the endpoint is defined as `endpointFish`, where we get the fish from tank with the id `tid`. We also specify that we want a maximum of 100 items from the request.

As we can see from line 10 in the code listing, we can also send a JSON object to the function. The question mark after `obj` indicates that this is an optional parameter, because it is only used to POST and PUT, and it therefore doesn't make any sense

¹⁰https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/await_sync_function

to send an empty JSON object every time we are doing a GET request.

Code listing 6.15: Using the FetchFromAPI function

```
const endpointFish = '/' + tid + '/fish?MaxItems=100';
const fishRes: any = await FetchFromAPI('tanks', 'GET', endpointFish);
```

Further in the function is the urlprefix, category, and endpoint put together to the final url, which is then sent to the `fetchAuthorized` function from `AuthContextProvider`.

At the end of the function the data retrieved from the fetch is returned. If the method is not DELETE, the data from the fetch is a JSON object and therefore needs to be parsed before it is returned. If the method is DELETE the data from the fetch can be returned as it is.

6.5 User profile page

The user profile page (also called account page) is a page that Cryogenetics wanted us to develop as a way to get a quick and simple overview over the user's Azure account information. It is not intended as a main component of the site but is rather meant to be a shortcut to user information, so that the user does not have to navigate themselves to Azure to get an overview of their account.

Fetching from Microsoft Graph API

Microsoft Graph API is a RESTful web API that enables us to access Microsoft Cloud service resources. [10]

```
1 const retrieveValue = async (Url: string): Promise<any> => {
2
3     const result = await AzureAuthenticationContext
4         .Singleton.fetchAuthorized(Url, {
5         method: 'GET'
6     });
7
8     if (result.status == 401) {
9         const testAccounts = testPublicApp.getAllAccounts();
10
11         testPublicApp.setActiveAccount(testAccounts[0]);
12
13         const request = { scopes: ["user.read"] };
14         testPublicApp.acquireTokenSilent(request).then(x => {
15             console.log("GOT REFRESH TOKEN: ", x);
16         });
17     }
18     else {
```



```
19     const data = await result.json();
20
21     return data;
22 }
23 }
```

Code listing 6.16: Checking and retrieving access token for Microsoft Graph API

The above code checks the status code of `result` and will in this case either return a 401 (forbidden) or a 200 (OK) status code. If the 401 status code is returned then we retrieve a new access code based on the request, which in this case carries the scope "user.read". The Microsoft Graph permission reference "user.read" is what grants permission to read the profile of the signed-in user.

```
1 const userUrl = 'https://graph.microsoft.com/beta/me'
2
3 const getJobTitle = async (): Promise<any> => {
4     const fetchDetails = (await retrieveValue(userUrl)).jobTitle;
5     return fetchDetails;
6 };
```

Code listing 6.17: fetch a user's job title from Microsoft Graph API

The above code displays a simple example of how we fetch the user's job title with the endpoint `userUrl` as the parameter to the `retrieveValue` function from the code listing 6.16.

6.6 The Authentication Process

Azure Portal Application Setup

We chose to use Azure Active Directory Authentication (Azure AD), which is Microsoft's cloud-based identity and access management service. This begins with the registration of our Single Page Application (SPA) on Azure Portal. Azure AD provides a very simple user access security, as upon the application registration there is an option to choose who can use the application and access the API:

Microsoft Azure

Search resources, services, and docs (G+)

Home > Standardmappe >

Register an application

*** Name**
The user-facing display name for this application (this can be changed later).

Cryogenetics API ✓

Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Standardmappe only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web e.g. https://example.com/auth

Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from [Enterprise applications](#).

By proceeding, you agree to the [Microsoft Platform Policies](#)

Register

Figure 6.6: Application registration on portal.azure.com

On the option of "Who can use this application or access this API?" we selected "Accounts in this organizational directory only (Standardmappe only - Single tenant)". This means that in order for a user to gain access to this API, the user's email must be listed in our Azure Active Directory (Standardmappe).

If a user tries to login to our application with an unlisted email, they will be met by this:

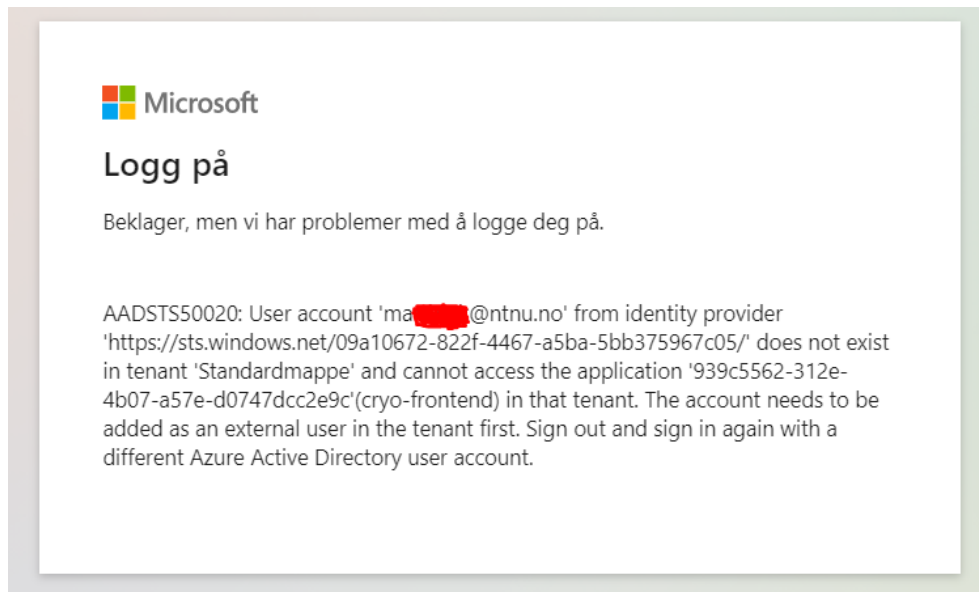


Figure 6.7: A user tries to log in with an email that is not listed in the app's Azure AD

This naturally solves an otherwise great barrier in the quest of keeping the application accessible only for those users that are meant to have access to it.

Azure AD also offer simple built-in roles that restrict or allow users to different types of permissions. Cryogenetics did for example have a wish of allowing only a couple of staff members to have read-permissions of the application's user-profiles. This was easily implemented by grouping the staff into different roles, which we called "admins" and "users".

After we clicked "Registrer", we were assigned an Application ID (also called Client ID). This value is what uniquely identifies our application in the Microsoft identity platform, which we'll talk about in the next subsection.

Microsoft Identity Platform

We registered our application on Azure portal, but it is not enough to just create an application on a website. We also need to do quite a bit of coding, and this is where Microsoft Identity Provider (MIP) comes into great assistance.

To put it in simple terms, MIP is what essentially connects the user to our registered application, and it is what helps us build our application (code-wise). It is what allow users to be able to sign in to our application using their Microsoft identities, by providing authentication and authorized access to our API or Microsoft Graph.

There are several components that are offered by the Microsoft identity platform. This includes, but are not limited to:

- OAuth 2.0 and OpenID Connect standard-compliant authentication service,

enabling us to authenticate Microsoft accounts.

- Open-source Microsoft Authentication Libraries (MSAL). These libraries provide features such as acquiring access tokens, getting and setting user accounts, and more helpful resources that are important for us as developers to create this application.

(Read more about OAuth and MSAL in section 6.6)

Access and Refresh Tokens

Access tokens are a fundamental part of the authentication process as the access token is the key that is used by the APIs to perform authentication and authorization. It is also what allows the users of our application to securely call protected resources.

- We use access tokens in the format of JSON Web Tokens (JWTs), which we acquire from the Microsoft identity platform.
- We use two different types of access tokens. One for the Microsoft Graph API, and one for our own API.

Upon login, the access and refresh tokens are set by acknowledging the application ID, as well as the OAuth 2.0 authorization code, which is a code obtained by the Microsoft identity platform upon user authentication.

When an access token expires, a new access token must be acquired by using a refresh token (which have a longer lifetime than the access token), and reauthenticate the user without interactive prompting. This keeps the user from having to manually log in every day.

Handling Endpoints and Access Tokens

If the access token expires, the new access token is fetched by the following code:

```
1 public async getAccessToken(endpoint: string): Promise<string> {
2     const hostname = new URL(endpoint).hostname;
3
4     if (AzureAuthenticationContext.Tokens == null)
5         AzureAuthenticationContext.Tokens = new Map<string,
6             AuthenticationResult>();
7
8     if (AzureAuthenticationContext.Tokens.get(hostname) != null) {
9         const now = new Date(Date.now() + 60);
10        const expiry = AzureAuthenticationContext.Tokens
11            .get(hostname).expiresOn;
12
13        if (now >= expiry) {
14            await this.refreshAccessToken(endpoint);
15        }
16    }
```

```

16     }
17     else {
18         const accounts = this.myMSALObj.getAllAccounts();
19
20         if (accounts.length > 0) {
21             this.myMSALObj.setActiveAccount(accounts[0]);
22             await this.refreshAccessToken(endpoint);
23         }
24         else {
25             handleUrl();
26             // User is not logged in, need to log them in.
27         }
28     }
29
30     return "Bearer " + AzureAuthenticationContext.Tokens
31         .get(hostname).accessToken;
32 }

```

Code listing 6.18: Get access token code

Due to us having to handle multiple access tokens, we created a custom system to manage them for us. This system allows us to call `AzureAuthenticationContext.Singleton.fetchAuthorized(...)`. `fetchAuthorized` is essentially a wrapper around the built in `fetch API`¹¹ that handles setting and refreshing access tokens.

Internally two maps for the different endpoints are maintained. One for scopes, and one for tokens. The function `getAccessToken` is responsible for handling which tokens correspond to which endpoint, and to refresh them if necessary. See code listing 6.18.

The reason it is necessary to have multiple access tokens (one for each endpoint), is that the Microsoft Identity Platform disallows multiple audiences per token for security reasons (as otherwise the back end could get access to read user data through scopes not intended for it). In our application, our back end would be one audience, and the Microsoft Graph API would be another.

Code listing 6.19: Instantiating authorization system with graph api and test scopes.

```

const endpointScopeMap: Map<string, string[]> = new Map<string, string[]>();
endpointScopeMap.set((new URL("https://<URL_of_backend_service>")).hostname, [
    ↪ api://fd197b0d-c07c-41c6-9f97-0bcf9c37954a/test.test"]);
endpointScopeMap.set((new URL("https://graph.microsoft.com/beta/me")).hostname,
    ↪ ["user.read"]);

const authenticationModule: AzureAuthenticationContext = new
    ↪ AzureAuthenticationContext(endpointScopeMap);
AzureAuthenticationContext.Singleton = authenticationModule;

```

¹¹https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

At a high level, the authentication flow from the front end to the Microsoft Identity service looks like this (from the microsoft documentation, see [11]):

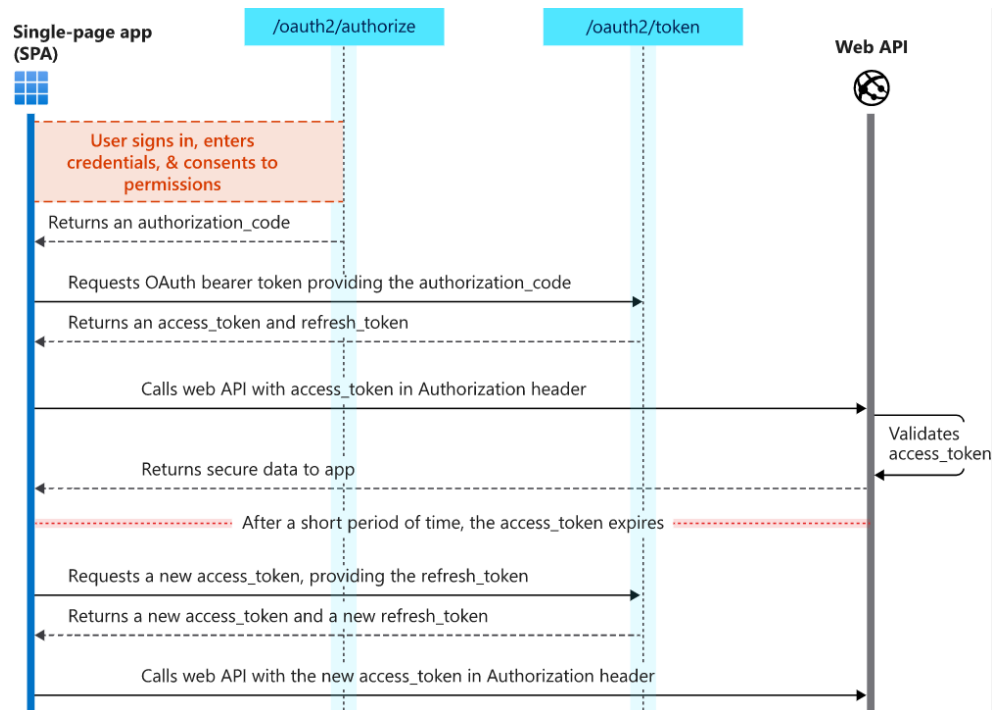


Figure 6.8: Screenshot from Microsoft docs. [11]

6.7 Back end

Paging

To prevent unreasonably sized requests, a paging system was implemented. The way the paging system works is by providing consumers of the API with two tokens, *Next Page Token* and *Previous Page Token*. They can then provide the token to endpoints implementing the system, which will return them a new page of results. The token should be treated as an opaque string as the consumer (they should not attempt to parse, or modify it. As the format may change without warning).

Cosmos DB does implement a paging system for querying the database in the Cosmos DB .NET SDK, but only offers continuations of existing queries in the forward direction.

The paging system works by first ordering all results by `_ts` and then by `id`. `_ts` is the timestamp the document was last modified specified in unix time in seconds. As `_ts` only has a precision of whole seconds, we also order by `id` which is unique per object.

Using `id` is the only field required to get a well defined output, as it is unique. But `_ts` was included for the convenience of always having results ordered by

time.

Continuation tokens are stateless on the server side, and work by encoding the `id`, and `_ts` properties of an object, and if the query is going backwards (previous page token). The next page token is encoded from the last object returned, and the previous page token is encoded from the first object returned.

$$\begin{aligned} \text{Predicate} \Rightarrow \\ _ts \geq \text{token}._ts \wedge (_ts \neq \text{token}._ts \vee (id \geq \text{token}.id \wedge (id \neq \text{token}.id \vee \perp))) \end{aligned} \quad (6.1)$$

Shortened to:

$$\text{Predicate} \Rightarrow _ts \geq \text{token}._ts \wedge (_ts \neq \text{token}._ts \vee id > \text{token}.id) \quad (6.2)$$

The query is generated by appending the predicate in equation (6.2), and then ordering by `_ts` and `id` as discussed above.

$$\begin{aligned} \text{Query}(x, \text{inner}) \Rightarrow x \geq \text{token}.x \wedge (x \neq \text{token}.x \vee \text{inner}) \\ \text{Query}_p \Rightarrow \text{Query}(P_n, \text{Query}(P_{n-1}, \text{Query}(\dots, \text{Query}(P_0, \perp)))) \end{aligned} \quad (6.3)$$

The predicate in equation (6.2) was created by following the scheme in equation (6.3), and passing in `_ts` and `id` as parameters. `Query(x, inner)` filters for a specific property, where `x` is the property and `inner` is an inner condition (either more `Query` calls, or false if it's the last property in the chain). `x` represents the property of a document in the database, whilst `token.x` represents the stored value of the property `x` in the continuation token.

$$\text{Query}_{backward}(x, \text{inner}) \Rightarrow x \leq \text{token}.x \wedge (x \neq \text{token}.x \vee \text{inner}) \quad (6.4)$$

For the previous page token, we simply change the greater or equal than to a less than or equal, see equation (6.4).

We did a prototype implementation of allowing arbitrary ordering, which in itself works, but is missing some details to function with the page tokens. Specifically, the continuation tokens need to include the values of the fields which are ordered by, and the predicate generation needs to also query by the additional properties following equation (6.3).

Dynamic Object

To facilitate working with semi-structured objects, a special *Dynamic Object* base class was made. This is to solve the issues arising from not knowing up-front the exact structure of the data we are working with.

Our custom class implements the *IDictionary* interface, which is specially treated in the *Newtonsoft.JSON* library¹².

If a class implements the *IDictionary* interface, the classes properties will not be directly serialized. Instead the contents of the dictionary will be serialized recursively.

We have an instance of the *JObject* class as a member. In our implementation of *IDictionary* we directly pass all parameters to our instance of the *JObject*.

We cannot inherit from *JObject* directly, as it has special treatment in *Newtonsoft.JSON* due to it being a descendant of *JToken* which is used for serialization. If we do, we will be unable to deserialize to our class. This is why we keep the *JObject* as a member, and do not inherit from it.

We then created five special methods for fetching data from the internal *JObject*.

- *Get* Attempts to fetch the value of a given key and type. If the key does not exist or a cast to the type fails (wrong type), null is returned.
- *Require* Fetches a value of a given key with a given type. This method will throw an exception if either no value is found, or the value is of the wrong type.
- *GetArray* Equal to *Get*, but is used for arrays.
- *RequireArray* Equal to *Require*, but is used for arrays.
- *Set* Sets a given key to a specified value.

Code listing 6.20: Example class deriving *DynamicObject*

```
public class DynamicUniqueObject : DynamicObject
{
    public DynamicUniqueObject()
    {
        Id = Guid.NewGuid();
    }

    public DynamicUniqueObject(JObject other) : base(other)
    {
        Id = Guid.NewGuid();
    }

    public static readonly string IdProperty = "id";
    public Guid Id
    {
        get => Require<Guid>(IdProperty);
        set => Set(IdProperty, value);
    }
}
```

An example of a class deriving from *DynamicObject* would be *DynamicUniqueObject*, which is the base class for all objects requiring an unique id. The constructor taking a *JObject* parameter can be used to set the root *JObject* of the class. It is

¹²<https://github.com/JamesNK/Newtonsoft.Json/blob/f7e7bd05d9280f17993500085202ff4ea150564a/Src/Newtonsoft.Json/Serialization/JsonDictionaryContract.cs#L121>

also possible to nest objects inside each other, which is what is done for the *DynamicFish* object, which contains *DynamicLocation* data according to figure 5.3.

6.8 Back End Security

We will be referring to vulnerability names from OWASP's top 10 from 2017, see source [12].

A1:2017-Injection

We base this paragraph with trust in the Azure Cosmos DB C# adapter v3.17.0¹³.

To mitigate the risk of SQL injection, we use LINQ (Language-Integrated Query) to SQL generation where possible. LINQ is an extension library found in the core .NET libraries, that opens up a set of extension method to be used on enumerable objects (lists, arrays, etc.). These methods reproduce much of what you would expect from SQL, but in a manner which is checked by the compiler.

Code listing 6.21: Fetching parameterized data using raw SQL - Pseudocode

```
int value = 13;
var queryDefinition = new QueryDefiniton("SELECT_f.foo_FROM_f_WHERE_f.foo_>_@value");
queryDefinition.WithParameter("@value", value);
var result = database.query...(queryDefinition);
```

Code listing 6.22: Fetching parameterized data using LINQ to SQL - Pseudocode

```
int value = 13;
var result = database.GetItemLinqQueryable<int>().Where(x => x > value)...query();
```

Using LINQ to SQL allows us to disuse plain text SQL for most queries (with one exception in our codebase). Abstracting away the plaintext SQL also does mean that porting from a SQL back end to another type of database should be easier, being less SQL to translate. However, there is a downside. Dynamic properties (properties we do not know of ahead of time) are more difficult to get working with LINQ to SQL. We attempted to create a dummy class, one that we could use the lookup operator on in C# (which is supported by LINQ to SQL), and to chain those together to allow unknown properties to be accessed (code listing 6.23). This failed due to the LINQ to SQL translating layer not understanding how to cast from the type of object you wanted, to the "Dummy" type.

In retrospect, this might be solvable by using "JObject"¹⁴ as the target type.

Code listing 6.23: Pseudocode JObject for safe SQL generation

```
using Newtonsoft.Json.Linq; // For JObject

var result = database.GetItemLinqQueryable<JObject>()
    .Where(x => x["dynamicProperty"] > value)...query();
```

¹³<https://github.com/Azure/azure-cosmos-dotnet-v3/tree/releases/3.17.0>

¹⁴https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_Linq_JObject.htm

The one notable exception to use LINQ to SQL in our back end codebase is in the paging system. Due to the issue described above with dynamic properties, especially with regard to sorting, raw SQL is used.

As raw SQL is used, it's considered a critical section. We constructed two classes, "OrderBy" and "EscapedOrderBy". The "OrderBy" class takes in the raw user-provided values. The unescaped and insecure "OrderBy" value can then be converted to a secure and escaped "EscapedOrderBy" class by simply passing it in the constructor. All user provided values are escaped in the constructor of "EscapedOrderBy", which then can be added to a query to allow user defined arbitrary sorting.

How the escape mechanism works, is that every character in the raw and insecure user provided input is compared to a vocabulary of known safe characters:

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890._

If the character is in the vocabulary, it is appended to the end of a known safe escaped string. If it does not match, an exception is thrown. You can only add a set of "EscapedOrderBy" to a query, not a set of "OrderBy", and the only way to create a "EscapedOrderBy" is to go through the constructor, which takes an unescaped "OrderBy".

OWASP A1:2017-Injection specifies the following mitigations against injection attacks¹⁵:

1. The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).

Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().

This is followed where possible by using LINQ to SQL. The note is not currently relevant, as no stored procedures are used.

1. Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
2. For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.

Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.

A max length for property names should be set, but currently is not.

A2:2017-Broken Authentication

We base this paragraph with trust in the Azure AD (Active Directory) C# adapter.

¹⁵https://owasp.org/www-project-top-ten/2017/A1_2017-Injection

For authentication we use Azure AD, which we trust to follow the best procedures regarding authentication. We therefore do not handle passwords in our application directly.

A3:2017-Sensitive Data Exposure

We base this paragraph with trust in the Azure AD (Active Directory) C# adapter.

Data stored in the back end database, Azure Cosmos DB is encrypted at rest by default. We use encryption keys managed by Azure instead of customer managed keys (managed by Cryogenetics), due to the ease of maintenance.

For front end to back end communication, the plan was to host the front end as a static website, with HSTS¹⁶ enabled, and CSP (CSP in this section refers to Content Security Policy¹⁷, not Cloud Service Provider) set to only allow first party resources and scripts.

Communication with the back end API is required to use HTTPS. Allowed TLS versions follow that of the host operating system, where 1.2/1.3 or greater is preferred.

Using CSP is also a mitigation against supply chain security issues with respect to Node Package Manager (NPM)¹⁸, react-scripts and it's many transient packages (checked¹⁹ 2021-05-06: 1469!) from disparate authors. (Please also note dependency confusion, although not relevant to our codebase due to a lack of internal packages²⁰). Setting CSP to run with only origin + whitelisted api's will prevent generic information leakage (as they can't easily send web requests to C&C sites). It however, does not prevent against DoS attacks. Nor does it protect the developers machines from being infected, and leak information. It is therefore not enough on it's own to prevent supply chain attacks, but does limit the damage from certain types of attacks.

Cross-Origin Resource Sharing (CORS) is enabled on the back end API server, to allow the frontend to connect.

A5:2017-Broken Access Control

We base this paragraph with trust in the Azure AD (Active Directory) C# adapter.

Access control on the back end is handled in a deny by default fashion, with the Asp.Net Core "*RequiredScope*" attribute selectively exposing scopes (although roles would likely be a better fit, due to individuals at Cryogenetics having distinct roles). Using authorization attributes is the natively supported way to do authorization in ASPNET Core²¹.

¹⁶<https://developer.mozilla.org/en-US/docs/Glossary/HSTS>

¹⁷<https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>

¹⁸https://link.springer.com/chapter/10.1007/978-3-030-52683-2_2

¹⁹Checked with (just dependencies, not development dependencies nor peer dependencies): <https://npmgraph.js.org/?q=react-scripts>

²⁰<https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610>

²¹<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies?view=aspnetcore-5.0>

6.9 Production setting

Azure APIM CORS

We had some issues configuring CORS with Azure APIM for testing, so please note that methods and headers must be provided. You can simply use "*" (star) to allow all methods and headers for testing. Otherwise CORS may not work. Although to minimize attack surface area when running in production, methods and headers need to be explicitly designated.

Code listing 6.24: Enabling CORS in Azure APIM

```
<cors>
  <allowed-origins>
    <origin>http://localhost:3000/</origin>
  </allowed-origins>
  <allowed-methods>
    <method>*</method>
  </allowed-methods>
  <allowed-headers>
    <header>*</header>
  </allowed-headers>
</cors>
```

Back End Configuration

Configuration is done using the `Microsoft.Extensions.Configuration` library. The three main configuration providers used are *file* (`appsettings.json`), environment variables ("*configuration*" under app service in the Azure portal), and key-value store.

Default configuration is stored in the `appsettings.json` file in the repository, which is cloned to instances of the back end. Environment variables are used for configuration options which do not require special treatment, like the configuration option for enabling swagger `useSwaggerInProd` (see section 8.2).

Connection strings are stored in a `key-vault`²², which is specifically made to store sensitive information in a secure way (such as connection strings).

For connecting to the key vault, the app services are configured with managed identity. This essentially allows us to treat the app service as a user in our active directory, and to give it permissions like access to the key vault. No passwords or secret keys are required by us. This does of course require both resources to be hosted on Azure. We then simply set which key vault we want to use in our `appsettings.json` file.

²²<https://azure.microsoft.com/en-us/services/key-vault/>

Chapter 7

User Interface

The employees at Cryogenetics are the only ones who will have access to the website. This means that we placed the usability and functionalities accordingly. Our goal with the web application was that employees should easily recognize similarities from other Office applications, and thus make it easy to understand where to find the information or functionality they are looking for.

7.1 Fluent UI

It was a natural choice for us to decide to use Microsoft's Fluent UI[9] since we wanted the web application to feel familiar for the Cryogenetics employees. It is a collection of UX frameworks for creating cross-platform applications that can share code, design, and behavior. Fluent UI has GitHub libraries for React (web), Android, iOS and macOS, which provides the basis for Microsoft 365 apps and services. Fluent UI also has a React Native library which makes it easy to create cross platform mobile application in JavaScript.

We decided to use Fluent UI because it would save us some time, since we would not have to design all the components ourselves. At the same time it would lead to a more consistent design. We also thought that it made sense to use Fluent UI since the employees at Cryogenetics are used to Excel and SharePoint from before, and that a continuation of this design would fit in well with Cryogenetics. We therefore decided to find inspiration in Microsoft's products, where especially Word Online was a web application we found a lot of inspiration in.

7.2 Layout

The main elements in our website as shown in figure 7.1 is the navigation bar at the top and the content area below this.

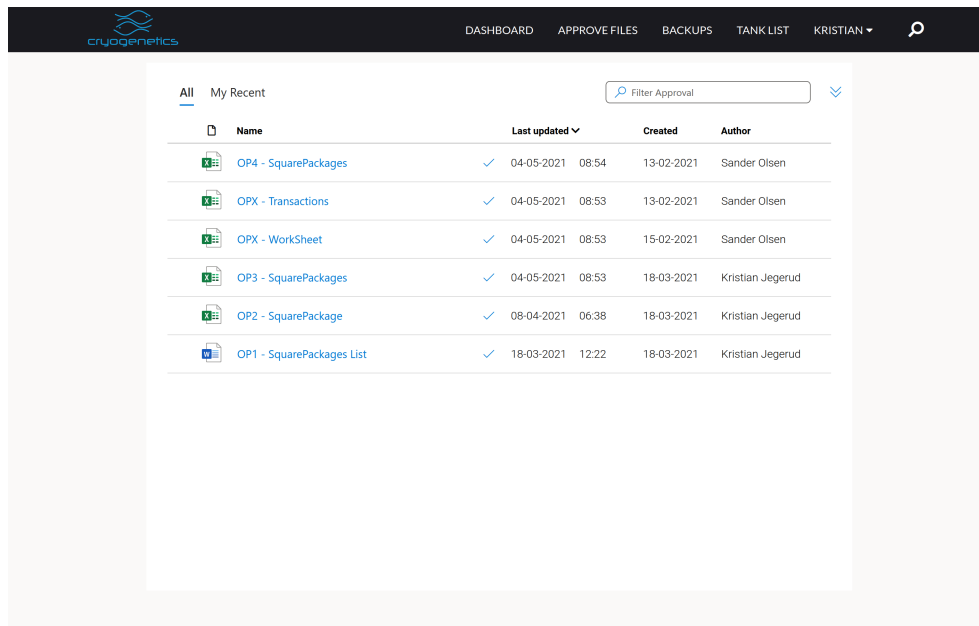


Figure 7.1: Front end layout (Laptop screen)

Navigation Bar

The navigation bar is the black, horizontal bar at the top of our website. It contains links to the various pages of our application, and in addition to a drop down menu located where the name of the employee is. The drop down menu contains two menu options; one to log out of the system and one to take you to the account page. You can read more on how it was implemented in section 6.3.

Content Area

The content area is the space on the page that changes based on where you are on the website. If the user were to view this on a laptop or a desktop computer, it would make sure the margins around the content area stays consistent across the different pages on the website. It keeps the content centered so it is more comfortable to view and navigate on larger screens.

Color Palette

We wanted the color palette to reflect the aquatic work Cryogenetics does and originally tried to replicate their own websites' colors (see appendix D, figure D.2a for the first design's colors). The palette we chose evolved over the course of the different design iterations and we landed on a palette that is both easy to read and fits well with the aquatic theme that we wanted. We established a color palette (figure 7.2) that would be used across the web application.

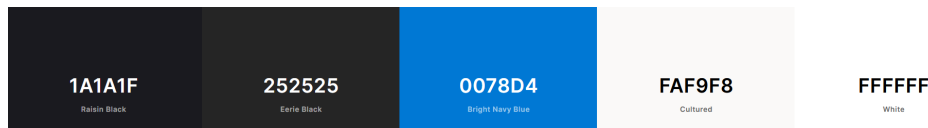


Figure 7.2: Color palette

The "Bright Navy Blue" color being the accent color used in hover effects, dashboard numbers and the logo itself. While the two darker colors reflect the look of the navigation bar and the side menu, with the intention of creating a small contrast between the two to somewhat separate them.

Finally the "Cultured" color is also used to create a contrast between the site's white background and the content area. This color was chosen because it features in the Fluent UI guidelines and will be familiar to Microsoft products users.

7.3 UI Examples

Dashboard

After logging in to the web application the user is welcomed by a message which depending on the time of day is different. Below the welcome message we have implemented a quick overview of the different information that could be useful for the user. Here the user can quickly identify how many files are ready for approval, the current number of clients with stored fish milt, and also a view of the different time zones Cryogenetics' offices are located in.

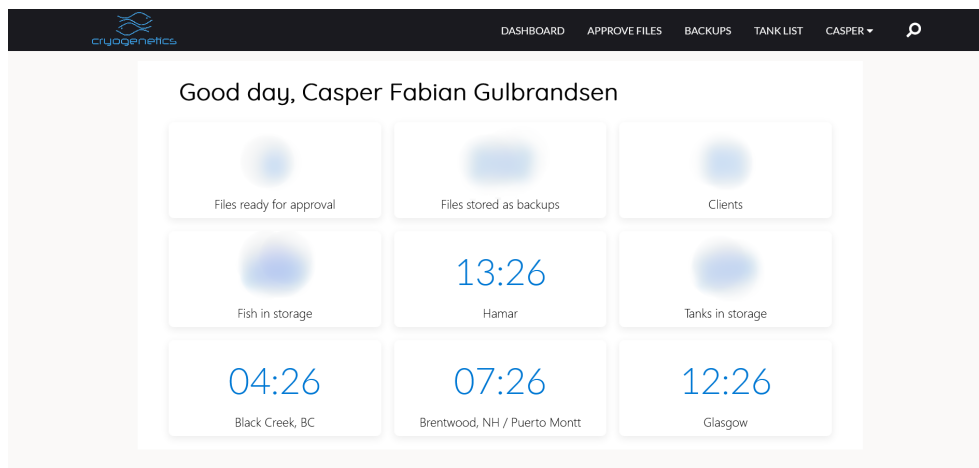


Figure 7.3: Dashboard landing page

User profile

The user profile page (or account page), can be accessed through the dropdown menu on the navigation bar, which appears if the user clicks on their name, which is illustrated in figure 7.4.

The user profile page is made with an intent to have as much "breathing" space as possible, meaning that it should not overwhelm the user with information. All we have is a centered table which fetches and displays resources from a user's Microsoft identity. There are two categories on the left-side menu-bar which each displays a table of information relating to their areas. The "About" category displays private user information like name and account creation date, while the "Business" category displays more work-related information such as company name, department name, job title, business phone etc.

There is also a horizontal, blue bar which displays the first name as well as the avatar of the user.

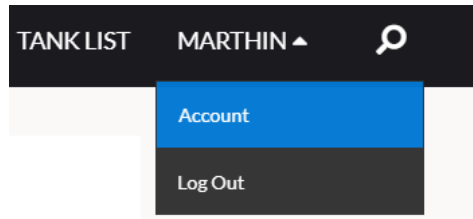


Figure 7.4: Dropdown

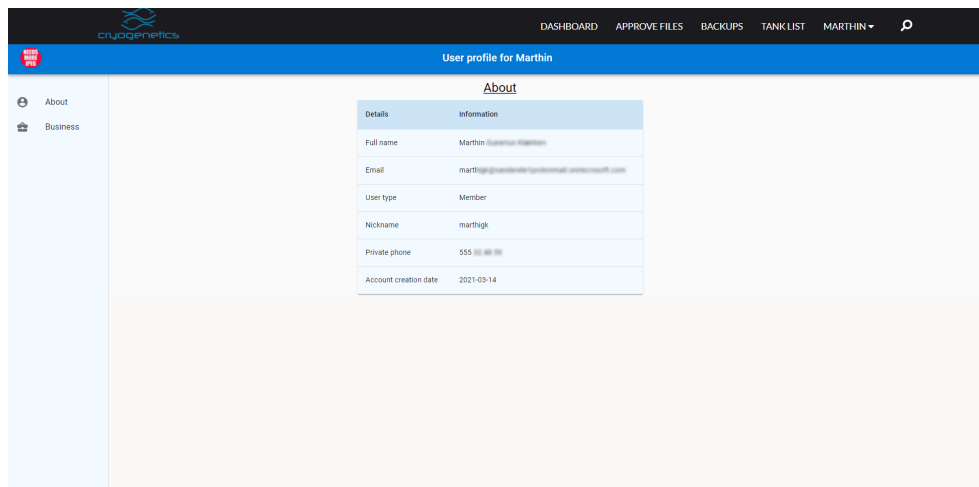


Figure 7.5: User profile page

Read more about the user profile page for mobile view in section 7.4

A List of Files

As mentioned in section 6.4, the different list pages are similar in user interface, with somewhat different functionalities between them. What they have in com-

mon is the same list design with the file icon, the name, the time it was created and updated, and also the author of the file. Cryogenetics would in most cases only upload excel documents, but the list component is also able to display any other Microsoft document icon as well if that should be necessary.

The user is able to filter the list by typing characters in the filter bar at the top, or they can click the two arrows next to it to open the filter menu where they can filter by file type or set a start and end date (figure 7.6). It is also possible to sort the list either alphabetically ascending or descending by clicking on the column titles.

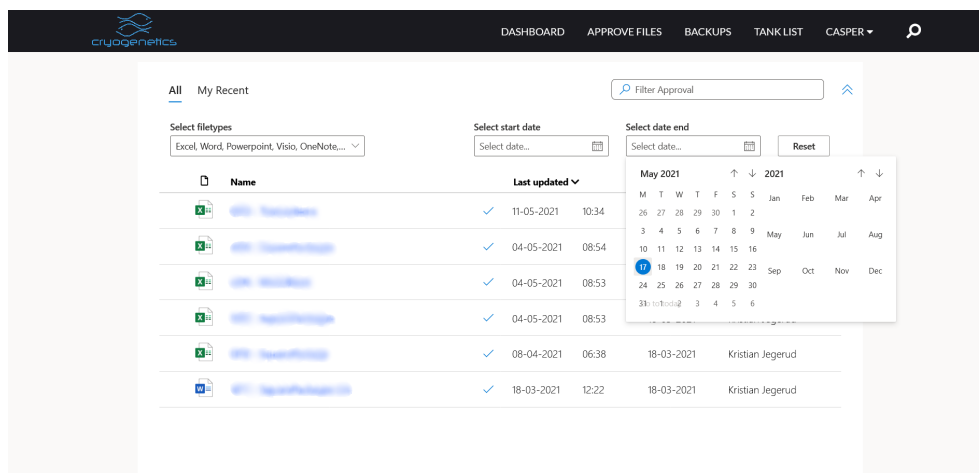


Figure 7.6: Filtering the list by the date uploaded

Global Search Bar

Cryogenetics asked for a feature that would allow them to search for individual species and receive the corresponding information belonging to that species. This feature was requested somewhat late in the development cycle and was therefore not prioritized over other features that needed to be finished. Despite this, the front end team was just about able to make some preparations for future search functionality. This meant adding the actual user interface of a search bar, but without any actual search capabilities in the database.

The search bar is accessed by pressing the search button in the navigation bar, and the search menu will drop down from the top of the screen as an overlay. With a content area which will display the search results. For instance, if a user would search for a specific species, all the tanks containing that specific species would be displayed as a list. Each element in that list would work as a routing link to the tank map.

The Tanks

The user is able to view each individual tank, and depending on the tank's volume is shown a different map of the contents. This is because the different tank volumes have a distinct structure inside. By clicking on each of the sections, the user will be able to see different data about the contents of the exact location within the tank. This will be displayed in the information area on the left hand side as shown in figure 7.7 and figure 7.8. If the user does not need to see the map at that moment, and would prefer to see the contents in a table format. They are then able to click the "view table" link and the map will be swapped out by a table which is also possible to print out, as mentioned in section 6.4.

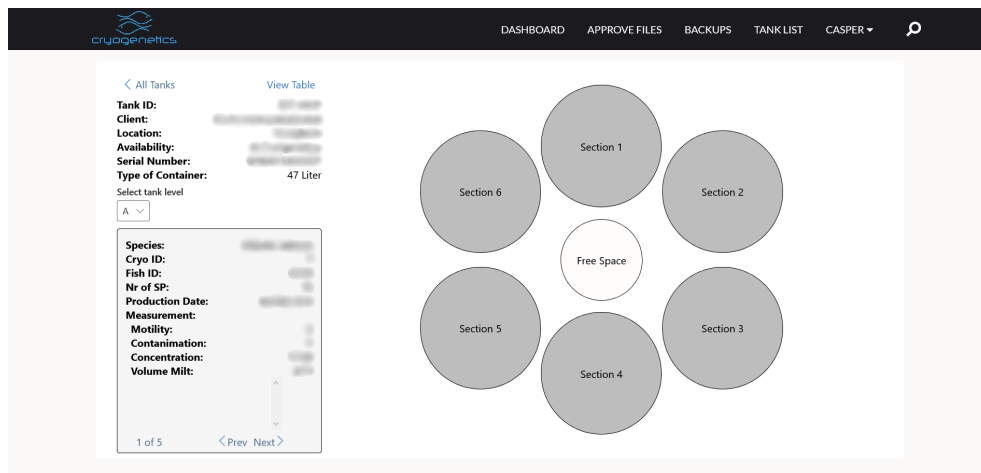


Figure 7.7: 47 Liter tank

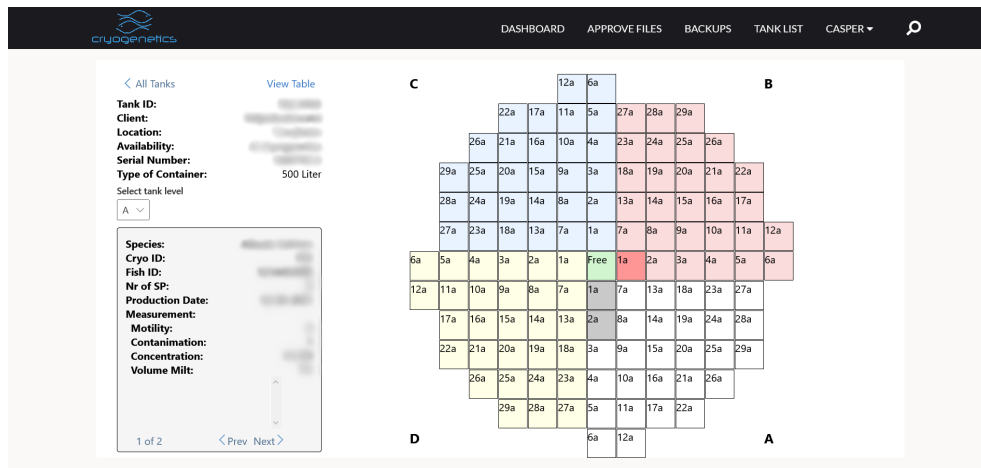


Figure 7.8: 500 Liter tank

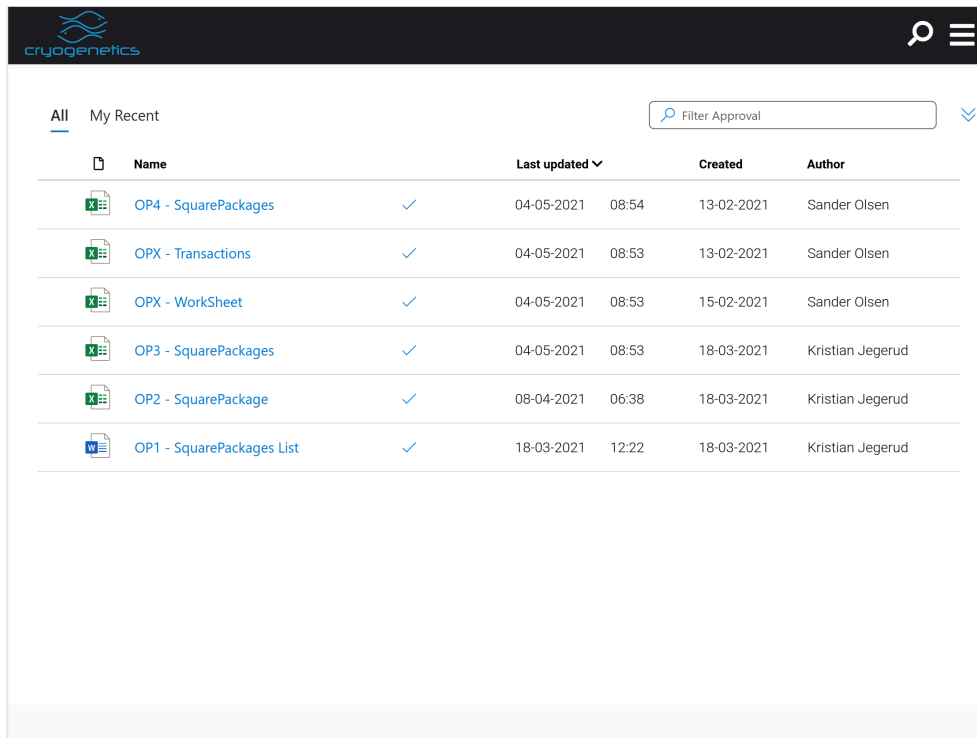
7.4 Responsive User Interface

Designing a web application with a responsive user interface in mind is to create content that adjusts to fit as the window size changes¹. Therefore it is important to keep in mind relative Cascading Style Sheets (CSS) units like rem and to apply media queries so the design automatically adapts to the browser window and to ensure consistency across multiple devices with various display sizes and formats.

Larger Screens

The employees at Cryogenetics work mostly from the office where they have external monitors. We assumed that they have 27 inches 16:9 ratio monitors, and our website is therefore adapted to look good on this screen size. The most important adjustment was to make the margins bigger, where we increased them from 12.5rem to 22rem. We believed this was the best solution as the content would be displayed in the center of the screen, so the user does not have to strain their eyes to look at the content.

Tablet Devices



	Name		Last updated		Created	Author
	OP4 - SquarePackages	✓	04-05-2021	08:54	13-02-2021	Sander Olsen
	OPX - Transactions	✓	04-05-2021	08:53	13-02-2021	Sander Olsen
	OPX - WorkSheet	✓	04-05-2021	08:53	15-02-2021	Sander Olsen
	OP3 - SquarePackages	✓	04-05-2021	08:53	18-03-2021	Kristian Jegerud
	OP2 - SquarePackage	✓	08-04-2021	06:38	18-03-2021	Kristian Jegerud
	OP1 - SquarePackages List	✓	18-03-2021	12:22	18-03-2021	Kristian Jegerud

Figure 7.9: Front end layout (Tablet screen)

¹<https://www.interaction-design.org/literature/topics/responsive-design>

When we adapted the website for tablets, there were two important adjustments we made. The first change was to resize the margins on each side of the content. As we can see in figure 7.9 we removed them completely so that the content extends all the way out to the edge. The second adjustment was to introduce the side bar.

Side Bar

On smaller screens such as tablets and mobile screens, the links in the navigation bar were not adequate. We therefore had to make changes here, and then followed a small guide from Justinmind² for this. Here, among other things, there was a tip about putting the links in a hamburger menu on the page, which for example is done by GitHub on their page. We therefore chose to do this, where the 5 links are placed in a side menu. The side menu appears from the right side of the screen by clicking on the hamburger menu at the top right. The Hamburger menu and side menu are default on screens smaller than 1367 pixels, which in practice means tablets and smaller devices.

The same has been done on the user profile page, as we also implemented a different side bar specifically for this page. Upon the breakpoint, this side bar is transformed into a hamburger menu, and will appear again by clicking on the hamburger icon on the left. This side bar will then appear with a z-index that makes it appear above everything else on the page. The menu can then be closed again by either clicking on one of the categories, clicking on the "X" in the top left corner, or just by clicking outside of the side bar area.

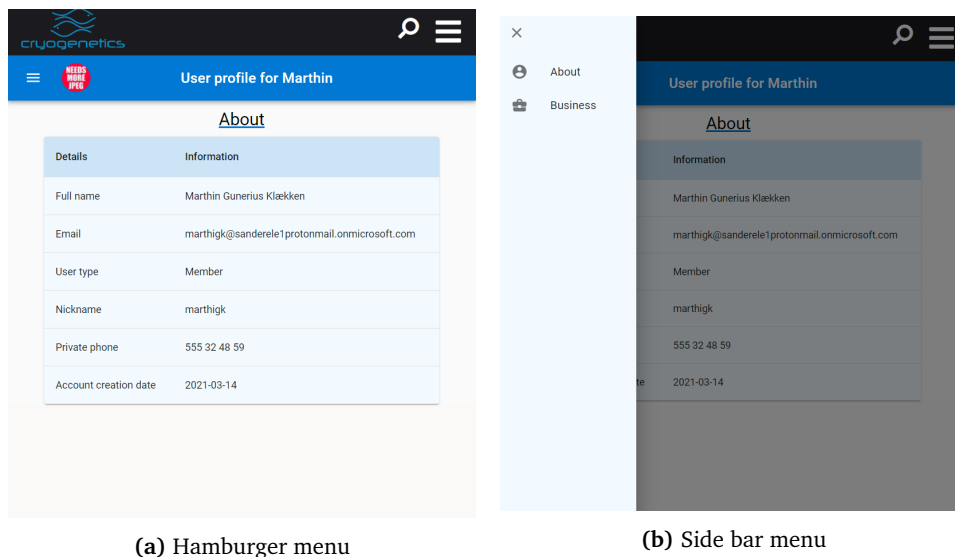


Figure 7.10: Account page before and after clicking on the hamburger menu in mobile view

²<https://www.justinmind.com/blog/hamburger-menu/>

Mobile Devices

It is important for Cryogenetics that they have access to the website when they are out of the office. Sometimes the only devices they have access to might be their mobile phones, and our website should therefore be possible to use on touch screens in mobile phone sizes.

In order to adapt the design to best fit mobile screens, we first had to think different about the design. Because mobile phones are usually held in a vertical direction, the screen therefore becomes quite narrow. As shown in figure 7.11, we have chosen to put elements such as search bar, pivot tabs³ and the list of files under each other. The same is done for properties for the file, such as file name, author, last edited and edit time.

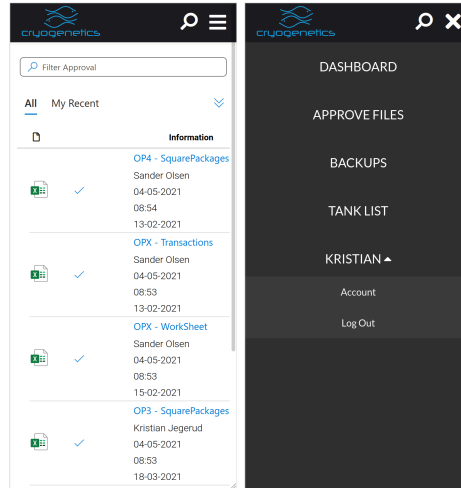


Figure 7.11: Mobile layout

As for the links in the navigation bar, we have done the same for the mobile phones as we did for the tablets (see section 7.4), which is to enter these as menu options in a sidebar that appears when you press the hamburger icon at the top right.

7.5 User Interface Iterations

To find the best possible UX, the design needs to go through various iterations before a final design can be decided upon. How many iterations varies from project to project, but the purpose of creating multiple iterations is that the next iteration is an improvement of the last. This is based on a review process along with the user and the designer. The designs were created with Figma⁴ so that we easily could display the design of the components we later were to recreate with CSS.

In the physical meeting at Cryogenetics' offices, we showed the product owner our first iteration and together we came up with various improvements, both in terms of it looking better and making it easier to navigate according to Cryogenetics. You can view the iteration process in appendix D.

³<https://developer.microsoft.com/en-us/fluentui#/controls/web/pivot>

⁴<https://www.figma.com/>

7.6 Web Content Accessibility Guidelines

Web Content Accessibility Guidelines 2.0⁵ is a set of wide range of recommendations, with the purpose of making Web content more accessible. Websites may be difficult for people with disabilities like low vision, deafness, limited movement and photo sensitivity to perceive and use. By following these guidelines we would make our website easier to use for these. WCAG has three levels which tells how much we fulfilled the requirements[13]:

- Level A: The easiest requirement to meet with only 25 criterias. At this level we are for example not allowed to identify something only by color, like "press the red button to go back".
- Level AA: Requires a bit more commitment as there are 38 criterias. At this level we need to make sure that all the text meets color contrast requirements, and it is also based on the text size as well.
- Level AAA: The level that is hardest to accommodate, with 48 criterias. At this level the requirements are strict with the color contrast requirement for the text, where you for example can only use very dark colors on a very light background and vice versa.

We have in this section chosen to show three of the contrast checks, where we used the online tool WCAG Contrast Checker⁶.

Homepage

The first test was done on the homepage, where we tested if the contrast between the white background and the blue numbers showing e.g. number of files to be approved was large enough. As we can see from figure 7.12 this component passes the test for AAA requirements, which is not very surprising as it is a relatively dark blue text on a white background.



Figure 7.12: Mobile layout

Navigation bar

The second test was done on the navigation bar (see section 7.2). Here we tested whether the white text on the black background met the requirements of WCAG 2.0. Although we have black and white elements, it is not certain that these always meet the criteria, as white text on a black background can often

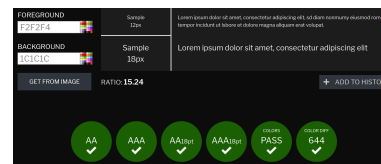


Figure 7.13: Mobile layout

⁵<https://www.w3.org/TR/WCAG20/>

⁶<https://contrastchecker.com/>

be problematic to see. Therefore, it is important that the contrast between the text and the background is high enough for our navigation bar to meet the criteria. As we can see from figure 7.13 this component also passes the requirements for AAA. For the navigation bar we want to point out that logos are not covered by the WCAG requirements, and we have therefore not done a test for the Cryogenetics logo.

Links

The third and last test we have done, was on the links on each file and tank (see figure 7.1). Here we tested whether the blue color of the links and the white background have a large enough contrast. Blue text on a white background and vice versa often cause problems in connection with the AAA requirements⁷. As we can see from figure 7.14 our test fails on the AAA requirements for text size under 18pt. The font size on our links are 1 rem which is the same as 12 pt on our site, which means that this test only meet the AA requirements.

We reckon that our website mostly meets the requirements for AA and not AAA, as this blue color together with white and black is repeated in most places on our website. In addition, the text size is mostly below 18pt on our page, so we won't meet the requirements due to our text being too small.



Figure 7.14: Mobile layout

⁷<https://uxmovement.com/buttons/the-myths-of-color-contrast-accessibility/>

Chapter 8

Development Environment

In this chapter we will take a closer look at our development environments. We wanted to learn how a professional development team worked on a project, and emulate the workflow with efficient and professional development environments and tools. A few of these programs were unfamiliar to us so we learned a lot in using them over the course of this project.

8.1 Front End

Docker

Docker is, as described in section 4.4, a tool that allows developers to package applications into containers, which makes the application run in every environment.

To use Docker in a project, you must create a dockerfile. A dockerfile is a text file with a set of instructions Docker uses when it builds images automatically. Code listing 8.1 shows our dockerfile. It starts with the instructions of pulling the official base image for React, before it sets the working directory to be /app. It then adds the /app/node_modules/.bin to path, before it installs the dependencies. Because we are using TypeScript, that means we also need to install the NPM package for TypeScript and @types/React. After these are installed, it copies the application from our host to the image, before it defines what commands Docker should do when docker run is done.

Code listing 8.1: Dockerfile

```
FROM node:13.12.0-alpine

WORKDIR /app

ENV PATH /app/node_modules/.bin:$PATH

COPY package.json ./
COPY package-lock.json ./

RUN npm install --silent
RUN npm install -g typescript
```

```
RUN npm install --save-dev @types/react -g --silent
COPY . ./
CMD ["npm", "start"]
```

If we were to use only a dockerfile we would have to run it with a long and complicated command in the terminal or in the command prompt. For those who want to run both the server and the client side, Docker has a feature called Docker Compose¹. That is a tool for defining and running multi-container applications in Docker, and with a single command creating and starting all these services. Compose consists basically of three steps:

1. Defining the applications environment in the Dockerfile
2. Defining the the service the application consists of in the `docker-compose.yml`, which make them run together in a isolated environment
3. Running the service using the command `docker-compose up`

code listing 8.2 shows our compose file. It gives name to the container which we have called `cryorepo` as well as defining which Dockerfile and volumes to use. It also set which port Docker should use and it enables hot reloading by setting `CHOKIDAR_USEPOLLING` to `true`. That means that the server will detect if we make a change in our code, and then reload the website.

Code listing 8.2: `docker-compose.yml`

```
version: '3.7'
services:
  sample:
    container_name: cryorepo
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - './app'
      - '/app/node_modules'
    ports:
      - 3000:3000
    environment:
      - CHOKIDAR_USEPOLLING=true
```

With this setup it was easy for us to start the website running on `localhost:3000` with a single and easy-to-remember command:

```
docker-compose up -d
```

For us on the front end team we only had the client side to run, so it was not really necessary to use to Compose for running our website. However, it would make it easier for us every time we run services, as we would not have to remember the command and possibly have to paste it from a note every time.

¹<https://docs.docker.com/compose/>

Git

As mentioned in section 4.5, Git was the chosen version control system because of our common experience with it. We all had used both GitHub and Git in several other projects prior to this one, and saw no reason to spend time needlessly to learn and use one we were not as familiar with. During development we loosely based our branching structure from a blog post called "A Successful Git Branching Model" by Vincent Driessen². The model consists of several branches, namely:

- The main-branch
- The develop-branch
- The feature-branches

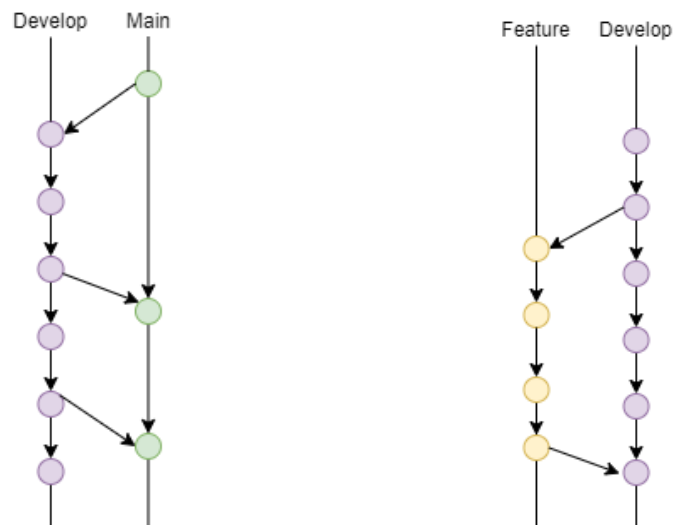


Figure 8.1: Git branch structure example

We considered `main` to be the branch where the source code should always be of a high quality state.

Next to the `main` branch is the `develop` branch, which contained source code that contained the latest features and changes ready for the next merge with `main`. When the code in the `develop` branch were stable enough, all changes would be merged into `main`. This process was handled during morning meetings when we felt enough implementations had been reached and the code was of a satisfying quality.

The act of merging itself was handled using GitHub's pull request feature where the developers could get an extensive overview of the differences between the branches and also if there were any unnoticed conflicts. When the team was satisfied, the merge would be conducted. We had no pre-determined style in our commit messages other than a short description in what changes was made.

²<https://nvie.com/posts/a-successful-git-branching-model/>

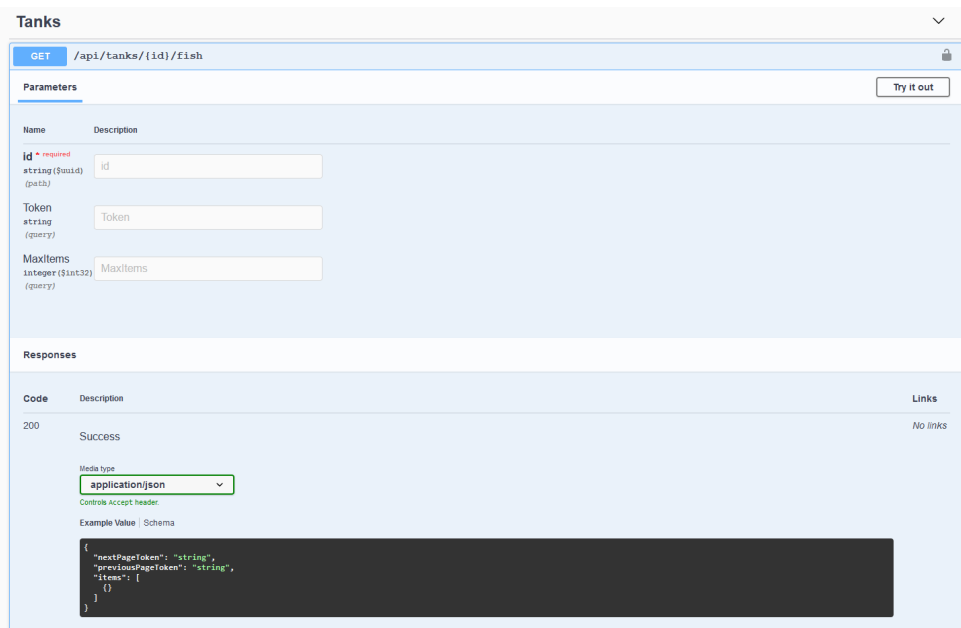


Figure 8.2: Endpoint to get all fish in a specific tank with parameters

Lastly, the feature branches differs from the other two branches in that they were not stored in GitHub, but instead exists locally on the respective developer's computer. These branches were used to develop new features without having to worry about breaking the stability of the program itself. The feature branch exist only as long as the feature is being developed, but will in time be merged into develop. After the merge process, the branch is deleted and the developer creates a new feature-branch when it was time to start development of a new feature.

8.2 Back End

Swagger

A separate API server was set up as a App Service on Azure, with Swagger UI exposed. This allows consumers of the API to test out the API with different parameters, and to experiment with how it functions. It also shows how to handle authorization tokens. See figure 8.2. The website had access control enabled through Azure App Service's built in access control mechanism, configured to only allow users within our test tenant to view the page (which would be transferred to Cryogenetics). See appendix G

Chapter 9

Quality Assurance

We have throughout the project focused on good code quality where we want the code to be easy to read and as efficient as possible. During the project period, we came across a number of techniques and methods that we implemented in the work. Since we cooperated with a company, we therefore wanted to work as professionally as possible and deliver good code.

9.1 Front end

Code Reviews

One of the techniques we used to write high quality code on front end was by doing a code review with at least one other team member, preferably more. We had no fixed day of the week or month, but we had reviews when we felt it was necessary. The way this was done was that the person who wrote the code showed his work and explained what the code does, so that the other group members understood what the code does and what the point is. This provided a basis for the other team members to understand the code, so that they could provide input and ideas on what could have been done different and maybe more efficient. If the other group members did not have any feedback, the code was pushed to the develop branch and basically seen as finished. An important principle for us when we worked on the code was that we had low threshold to change and give feedback on other's code, also after code reviews.

Code Review 25. march

In this review Kristian, who originally wrote the code behind the view to the map of the tanks, felt that there was too much duplication of code and that this could clearly and should have been done better. He and Casper decided therefore to do a code review, to see if it was possible to do this in a better way. Here they

found that the JavaScript method `Map`¹ was ideal for this problem, and therefore halved the number of lines of code. In code listing 9.1 and 9.2 we can see excerpts of the code showing how the map of a large tank for non-mobile screens was implemented. After the refactoring this bit of function was reduced from 48 lines of code to 24 lines of code.

```
1 export const TankmapTable500: React.FunctionComponent<Props> =
2   props => {
3     if (!props.mobile) {
4       return (
5         <div className="table-500-container">
6           <div className="sectionA">
7             <TankmapSection500
8               section={1}
9               level={props.level}
10              color={"white"}
11              handleClick={props.handleClick}
12            />
13          </div>
14          <div className="sectionB">
15            <TankmapSection500
16              section={2}
17              level={props.level}
18              color={"red"}
19              handleClick={props.handleClick}
20            />
21          </div>
22          <div className="sectionC">
23            <TankmapSection500
24              section={3}
25              level={props.level}
26              color={"blue"}
27              handleClick={props.handleClick}
28            />
29          </div>
30          <div className="sectionD">
31            <TankmapSection500
32              section={4}
33              level={props.level}
34              color={"yellow"}
35              handleClick={props.handleClick}
36            />

```

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

```

37         </div>
38         <div className="free">
39             <TankmapSection500
40                 section={5}
41                 level={props.level}
42                 color={"green"}
43                 handleClick={props.handleClick}
44             />
45         </div>
46     </div>
47     )
48 } else {

```

Code listing 9.1: Before refactoring

```

1 export const TankmapTable500: React.FunctionComponent<ITankMapTable>
2 = props => {
3     const colorsList = ['white', 'red', 'blue', 'yellow', 'green'];
4     const classList =
5         ['sectionA', 'sectionB', 'sectionC', 'sectionD', 'free'];
6     const selectedList = ['A', 'B', 'C', 'D', 'E'];
7
8     if (!props.mobile) {
9         return (
10            <div className="table-500-container">
11                [...Array(5)].map((item, i) =>
12                    <div className={classList[i]}>
13                        <TankmapSection500
14                            section={i+1}
15                            level={props.level}
16                            locationInfo={props.locationInfo}
17                            color={colorsList[i]}
18                            handleClick={props.handleClick}
19                        />
20                    </div>
21                )}
22            </div>
23        )
24    } else {

```

Code listing 9.2: After refactoring

Static Code Analysis

ESLint

Lint², also called linter, is a tool used to point out errors in the code, potential errors, stylistic errors and suspicious constructs in the code. When writing code in dynamically typed languages such as Python and JavaScript you should use tools like Linter ([14]). This is because compilers for such languages do not run with strict rules, tools like Lint can help find potential bugs in the code. The tool used specifically for JavaScript is ESLint³, and is installed using NPM. By using Linter together with TypeScript, we got rid of quite a few bugs and potential errors in the code. This is because we received warnings and small tips for improvements in syntax from both Linter and the TypeScript compiler. For example, this led to us being consistent in the use of `const` instead of `let` and `var` where it can and should be used.

We started using ESLint late in the project, as we were not aware that such tools existed and because we trusted that the IDEs we used would notify us of the most important errors. This meant that when we first started to use it, we received a bunch of error messages and warnings. This led to a lot of time being spent on correcting these errors. It is very likely that using ESLint from the beginning would have saved us some time when debugging the code.

Prettier

When several group members write code, it is almost impossible to get the same layout over the entire code. Therefore, we decided to use tools that format the code so that it is clear and similar. The choice therefore fell on the extension Prettier⁴, which is a tool that comes as an extension to both VS Code and NPM. Prettier ensures a consistent code style throughout the code base. It disregards original styling by deformatting the code, before rewriting it with its own rules by for example wrapping the code if it reaches a certain number of characters on each line.

```
1 useEffect(() => {
2     const fetchData = async () => {
3         Promise.all([getNrOfApprovableFiles(), getNrOfBackups(),
4 getNrOfClients(), getNrOfFishes(), getNrOfTanks(), getPlaceholder()
5     ]);
6     }
7     setGreeting(getTimeGreeting() + ', ' + props.name);
8     fetchData().then(() => { setLoading(false); })
9 }, [cookie])
```

listings/prettierCodeBefore.tsx

²[https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software))

³<https://eslint.org/>

⁴<https://prettier.io/docs/en/index.html>


```
1 useEffect(() => {
2     const fetchData = async () => {
3         Promise.all([
4             getNrOfApprovableFiles(),
5             getNrOfBackups(),
6             getNrOfClients(),
7             getNrOfFishes(),
8             getNrOfTanks(),
9             getPlaceholder()
10        ]);
11    }
12    setGreeting(getTimeGreeting() + ', ' + props.name);
13    fetchData()
14    .then(() => {
15        setLoading(false);
16    })
17}, [cookie])
```

listings/prettierCodeAfter.tsx

As we can see from code listing 9.3, this code is wrapping the document and is quite hard to read. In code listing 9.4 we have used the Prettier plugin, and even though the code became 9 lines longer we thought this was the best solution for our code as we found it easier to read.

9.2 Back end

For the back end which used a statically typed language, the compiler checks that type mismatches are not made. Additionally, the feature of C# 8, nullable reference types (see section 4.8) prevents certain types of errors.

Code quality regarding security aspects (specifically authorization/authentication) were kept strict (through reviewing the code, and following the documentation), due to the confidential nature of the data being worked with.

Quickly iterating prototypes allowed pitfalls of the technologies used to be learned, before significant amounts of code needed to be replaced. An example was how indexes in Cosmos DB works very differently to those of most RDBMS's, in that it requires indexes to be built for any query with consecutive orders.

Outside of that, the implementation were intended as a prototype, later to be replaced with a production quality version. This did not happen due to a lack of time.

9.3 Testing

Manual testing

For the front end team Firefox Developer Tools was used to manually examine, edit, and debug the website⁵. The Page Inspector tool proved useful, and helped in identifying bugs and issues with especially the CSS Grid elements on the page. It made it possible to highlight the components and give a visual feedback on the size and position of the grid.

We also used the Responsive Design Mode tool which made it possible for us to test the website on other screen sizes and formats with our own laptop (see figure 9.1).

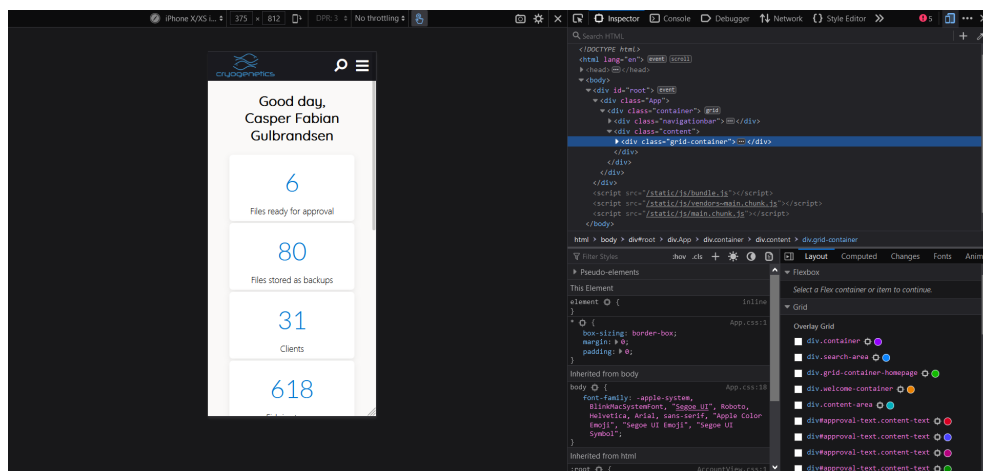


Figure 9.1: Testing the responsive design on an iPhone display

For the back end, manual testing of the RESTful API was performed by using the Swagger UI (see section 8.2). Input values were manually entered, and results were verified to be as expected.

User Testing

Our goal was to have multiple user tests together with the product owner. Because of Covid 19 this proved difficult to achieve, due to rules such as requirement for a home office and restrictions on the number of visitors. This meant that we had to re-arrange our plans and therefore think again.

Since we did not meet the product owner physically for user tests, we therefore had to test the working functionalities ourselves. Where possible, we therefore chose group members to test functionality that the others in the group created. We thought this gave a slightly more realistic picture of the testing, as those who tested did not know about any risks of bugs.

⁵<https://developer.mozilla.org/en-US/docs/Tools>

There are some functionality on the front end that does not have an endpoint to the back end, which means that there is functionality that we have not tested in use. The best example of this is the global search function, where what is missing is an endpoint in the API to filter the search results. This is something we have not tested in use, but we have tested that the results from the search will be presented in a clear way. This was done by generating a mock result using existing endpoints from clients, tanks and fishes, retrieving a random number from each of the endpoints and then shuffling the results in a random order. This mock result was then displayed as it was intended the search results should be displayed.

Mock RESTful API

Throughout the development process, the front end was often further forward than the back end in the development process. This meant that the front end team lacked endpoints on the API to test how fetching from current data from the API and processing of this data worked. Therefore, in order to test certain features in the website, a mock API was created by using Mocki⁶. Mock APIs imitates a real API server by providing realistic API responses to requests, which can hosted both locally or on the public Internet.

Mocki offers a free API editor, where it is possible to create GET endpoints. We used this to create an API which we could test our website on while waiting for our actual API. We first and foremost tested whether the data was presented correctly. When a mock API endpoint in Mocki was created, we were provided with a link we could fetch from. When fetching from this url a JSON object with the mock data was then returned, which we displayed as test data on our website.

We chose not to create a mock API where it was possible to POST or PUT data, as we did not want to pay for such a solution. We also did not consider it as necessary, because at that time the API endpoints for our website was closer to being able to be used for testing.

9.4 Automated Test

Front End

Throughout the project, we have tested our website manually, as described in section 9.3. It worked well, but there is no doubt that it took some time. We therefore had a desire to write automated tests for the most basic functionalities on our website. This was unfortunately deprioritized during the project period. The main reason was the lack of knowledge about automated testing among the group members, which made the threshold for making the tests too high. Therefore, tasks that included development and report writing became easier to prioritize over testing, especially when the manual tests worked as well as they did. In retrospect, we see that this is something we should put more time into, especially in the research

⁶<https://mocki.io/#features>

phase. This would have meant that we had entered the development phase with knowledge of automated tests, so that the code we wrote along the way would have been easier to write tests for.

We made a small attempt to write some UI and device tests towards the end of the project period. We did some more research for testing utilities, and found out that testing our application using Jest and Enzyme⁷ would suit the project best. Jest is a JavaScript library for creating, running, and structuring tests, whereas Enzyme is a tool for making it easier to assert, manipulate and traverse the output of our React Components. We succeeded creating some simple tests for our website, but this was quickly considered unnecessary as these tests would have a very small code coverage.

We also found out during this experiment that we lacked too much basic knowledge about testing, and that this was reflected in how we wrote the code. Our code became very difficult to test, because our components were often a little too large and too complex. The components depended too much on each other. In addition, much of our code also depended on communication with the API, which made it even more difficult because our limited knowledge of testing.

If we were to do the project again with the desired knowledge of testing, we wouldn't go for a test driven development, but we would definitely write tests along the way to make sure that our code would do the same even if modifications and refactoring had been done.

Back End

Initially, property based testing using FsCheck⁸ (Based on Haskell's QuickCheck⁹) was attempted, and was intended to be a large part of building a well developed back end. It however proved difficult to use for complex classes, and combined with frequent rewrites, did not see much use in practice. Due to frequent rewrites, basic unit tests were not utilized as well, being intended to be used for the final implementation. Preferably, unit tests would have been performed before allowing continuous integration to push code to the App Service.

⁷<https://javascript.plainenglish.io/are-you-not-testing-your-react-app-instantly-test-with-jest-enzyme-a-reactjs-2020-tutorial-e9ce0182d66d>

⁸<https://github.com/fscheck/FsCheck>

⁹<https://hackage.haskell.org/package/QuickCheck>

Chapter 10

Conclusion

Cryogenetics wanted an inventory management system with an associated database that would augment their current system of keeping inventory. We believe we have at least partially achieved this by building the foundations of a better system. This makes it easier for them to find what data they are looking for and to keep track of their inventory, and is more robust and automated. Although the project as a whole is not finished, nor featured complete to the degree we would have wanted (specifically with regards to back ups, automating ingestion of worksheets, and the transaction system).

With this system, Cryogenetics would be able to save time and resources they can allocate elsewhere as well as being able to scale their inventory and log it more efficiently than before. Because of reasons like overestimating our time combined with underestimating the scale of some use cases, we were not able to implement everything we initially wanted before the deadline of this report. We did however, lay the ground work for this to be implemented in the future, which Cryogenetics offered us to do as a summer job.

Learning Experience

The last few months has been extremely educational for us, and has given us valuable experience in the software development process. We learned a lot about working with a company and a product owner in both understanding their requirements for the project, and for us to accurately communicate our suggestions in how we would solve it.

We also learned a lot about working together as a team with solving problems together, planning meetings, the importance of tracking progress and detecting deviations early, and finding an agreement despite our different opinions on certain topics. This also means the importance of allocating resources better, so that we do not have a skewed distribution of labor in development. Combined with poor communication between front end and back end at certain times, this led to us not discovering that we would not be finished with everything we wanted.

We have also experienced that it is important to have openness in the group

and each group member shows self-insight. One of the reasons the back end development delayed is that it was not expressed earlier that the workload became too large. We would also like to add, as mentioned earlier, that this could and should have been avoided through closer cooperation between the back end and the front end team.

We also feel that we have become much better at the technologies we have used. The three members on the front end have built on the foundation from previous topics and developed further to have become more skilled in React and TypeScript. New skills have also been acquired from the world of web technologies, where the skills to use various UX frameworks and other 3rd party components have been improved.

On the back end, we improved our abilities with API design (especially RESTful API's). We also gained useful experience in creating a paging system, and allowing sorting in an arbitrary manner (which needs indexes handled), and exploring the different standards, such as OData and OAuth.

We all agreed that the project made us better developers, gave us much needed experience and opportunity to improve our understanding of both familiar and unfamiliar technologies and procedures. At the same time we now have had time to reflect over what we were not so happy with, and could have improved if we were to redo the project from the start.

What Could We Have Done Better/Differently?

In retrospect, there is little doubt that the group should have had much better collaboration between back end team and front end team throughout the project period. We should have included back end team in the morning meetings. At the same time we should have made a greater attempt to meet physically, even though it has been a pandemic period. This would have meant that we would more likely have discovered the overload of work on the back end, and we could therefore have provided more resources there and thus also finished with the core functionalities.

Regarding the development model, we should have used more columns in the Scrumban Board including the three we had to give us an even better overlook of the process. For example a review column with issues that had been implemented, but were primed for a review session within the group.

We also should have had more formal meetings physically with a prepared agenda. The meetings we did have physically as a group together, turned out to be more of a work session. These meetings should have focused more on sharing knowledge, in case one of the group members should for any reason be prevented to continue his work. This is something we should been much better at.

We see in retrospect that there are two major potentials for technical improvements. The first and probably the most important point is the testing of the application. As mentioned in section 9.4 we have no automated tests on the front nor back end, which has led to all the testing being performed manually. This has in

all probability led to a loss of time, and even if writing tests takes time, we would have saved much time from these tests that it could have paid off. The tests would also mean that bugs and errors had been detected earlier and it would have been easier to correct them. There is little doubt that if we could go back in time, we would have written tests for our application in parallel with the development. This would further lead to our application consisting of smaller and more components, as these are easier to write tests for than the larger components the application consists of now.

The second technical potential would be to improve the lack of comments in our source code, as it can be difficult for new developers to understand. It quickly became a terrible habit to not comment our code, and made the task too demanding to document every function at the end of the project.

Bad habits like these during development has shown us throughout this project exactly why they are called **bad** habits, and we had to deal with the problems that became apparent because of it.

Further Work

The system we developed turned out reasonably large and unwieldy, as such documentation is imperative for new developers to understand the system. If this project were to be handed off to new developers in it's current state, it would pose a challenge for the new developers to understand the system as a whole. Focusing on documenting the system, both in code, but also as a whole should have received more attention from the start.

Cryogenetics' further work will largely be about completing the core functionalities of the back end. Of these functionalities, it is to complete parsing of excel documents, the transaction system and functionality for approving excel documents, and creating endpoints for searching in back end that must be completed. These are the functionalities that we expect to be in place before Cryogenetics' employees can start using the system in practice.

On the front end, there is a lot of functionality for the user that as of today is missing, but as the code is structured, these can easily be added as soon as there is functionality in the back end. During the project period, the focus has been on the fact that the functionalities that have been implemented are of good quality and thus can be used in everyday work. We therefore assume that further work on the front end will therefore largely be about new functions desired by Cryogenetics.

Evaluation of the Group's Work

The collaboration in the whole group has been a bit varied, with a lot of good things and some things that should have been done differently. The front end team had a morning meeting every day of the week, where we briefly went through the day's plan, what was done the last 24 hours and questions were discussed. This led to us having good control over the progress in the front end, what the others did and it became easy to solve the problems that arose together.

On the front end team, the collaboration between the three members has worked very well. Due to distances between some of the group members and the fact that the corona pandemic has put an end to several physical meetings between the members, we have cooperated well through Teams. It was also helpful that two of the members met physically and worked together as a pair.

A requirements specification was never established between us as a group and Cryogenetics. In the first weeks, a draft requirements specification from Cryogenetics was provided, which contained some information about the data. This did not contain any specifications for database requirements, and also no requests for front end features. Further, this led to many new functionalities being desired by Cryogenetics throughout the project, where some of the wishes also changed over time. Converting their data to a standardized format proved difficult due to this as well.

This led us to a dilemma:

- We could have adhered to the original agreement where we had established an early requirements specification which we developed according to. Here we would with great certainty have finished what was decided, but the fully developed product would not quite fit the customer as they would have found more desired features after the deadline of the requirements specification.
- We could agree to a very smooth development process, where many wishes came along during the development. This would lead to less overview of the project and an increased probability that we would not be finished, but the product we developed would be more in line with the customer's final wishes.

We elected the second alternative, to give the customer the features they requested. This meant that we accepted a floating requirements specification, and this was also one of the reasons why we chose a flexible development model. In our opinion, the original draft requirements specification was full of shortcomings and Cryogenetics had some, but few desired functionalities early in the project which made it difficult for us to create a long term specification. There were few requirements for the front end in the original specification. It has therefore been a process where the product owner continuously supplies feature requests, which we fulfill and show to the product owner for further input. From the point of view of development, this worked very well, and the indications from the product owner indicates that they have also found this to be a good solution.

From the project's point of view as a whole, this led to us not completing all use cases. This is something we are not very happy with, as we would like to see that we were completely finished with the intended functionalities.

Organization and distribution of Work

We decided early in the project period that three group members would immerse themselves in front end development, and then work on this in the project. The

last group member expressed that he could take responsibility for the back end alone, as he had great knowledge of this from earlier projects. We thought then this seemed like a good plan. As said earlier in this chapter we found out too late that the back end was much further behind the schedule than we wanted, and that it was therefore too late to reallocate resources so that we would finish the project.

If we did the project again, we definitely would have distributed the resources in the group differently than we did. We would then have had two main front end developers who had worked closely together and made sure that the front end kept to the schedule. We would have had one main back end developer who had been in charge of database and API, who had made sure that the back end kept the schedule. The last group member would have been a flexible developer, with knowledge on both front end and back end technologies. This member would probably have worked mostly on the back end with creating endpoints and other easier tasks that the back end team needed to complete.

We believe this would have meant that the back end would have made enough progress to have finished the core functionalities. This could have led to us delivering a product that may have worked in practice.

Final Thoughts

Throughout the project we feel that we have learned a lot, both in terms of skills in the chosen technologies and not least of all how we work best as a team

When we look back on our project now it is clear that the front end is further ahead than the back end. We believe that by distributing the resources evenly between back end and front end, we would have come further overall. This could have led to us being able to supply a working product to Cryogenetics before the release of this report.

We believe that Cryogenetics has received a solid foundation, for the system being able to be used in practice.

Finally, we would like to thank Cryogenetics and especially Steffen, for what we perceived as a very good collaboration, and the experience we have received from the task they gave us.

Bibliography

- [1] I. Sommerville, *Software Engineering*, 10th. Pearson, 2018.
- [2] *What Is Kanban? Explained in 10 Minutes | Kanbanize*, [Online; accessed 3. May 2021], May 2021. [Online]. Available: <https://kanbanize.com/kanban-resources/getting-started/what-is-kanban>.
- [3] *What Are Project Boards?* [Online; accessed 5. May 2021], 2021. [Online]. Available: <https://docs.github.com/en/github/managing-your-work-on-github/about-project-boards>.
- [4] *What Is Protected Routes?* [Online; accessed 5. May 2021], Jan. 2021. [Online]. Available: <https://dev.to/mychal/protected-routes-with-react-function-components-dh>.
- [5] *What Are Refresh Tokens?* [Online; accessed 5. May 2021], Feb. 2020. [Online]. Available: <https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/>.
- [6] *Microsoft Engaged In A Predatory Campaign To Crush The Browser Threat To Its Operating System Monopoly*, [Online; accessed 5. May 2021], Jun. 2006. [Online]. Available: <https://www.justice.gov/sites/default/files/atr/legacy/2006/06/01/V-A.pdf>.
- [7] *Enhance your Cloud Security with AMD EPYC™ Hardware Memory Encryption*, [Online; accessed 5. May 2021], Apr. 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.
- [8] *Model-View-ViewModel (MVVM) Explained*, Online; accessed 10. May 2021, Apr. 2014. [Online]. Available: <https://www.wintellect.com/model-view-viewmodel-mvvm-explained/>.
- [9] *FluentUI*, Online; accessed 12. May 2021, May 2021. [Online]. Available: <https://developer.microsoft.com/en-us/fluentui#/>.
- [10] *What is the Microsoft Graph API?* Online; last accessed 19. May 2021, Apr. 2021. [Online]. Available: <https://docs.microsoft.com/en-us/graph/overview>.
- [11] *Authentication Flow*, Online; accessed 13. May 2021, Mar. 2021. [Online]. Available: <https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow#protocol-diagram>.

- [12] *OWASP Top 10 -2017*, [Online; accessed 5. May 2021], 2017. [Online]. Available: [https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%5C%20Top%5C%2010-2017%5C%20\(en\).pdf](https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%5C%20Top%5C%2010-2017%5C%20(en).pdf).
- [13] *WCAG Levels: level A, AA and AAA compliance*, [Online; accessed 5. May 2021], 2021. [Online]. Available: <https://myaccessible.website/blog/wcaglevels/wcag-levels-a-aa-aaa-difference>.
- [14] *What is a linter and why your should use it*, Online; last accessed 18. May 2021, Jul. 2020. [Online]. Available: <https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it>.

Appendix A

Scrumban tasks

Table A.1: Front End Scrumban Done

Fix refresh tokens	Implement ContextAPI to each View
Option to print TableView of tank	Configure log out
Redirect non-logged in users to login-page	Implement ContextAPI
Endre sortering fra logikk på frontend til å bruke API sortering (approvable og tanklist)	Implement Navigation Bar
Implement location from - location to in table view	List over files to be approved
Add table-view to tanks	Fix dropdown bug
I hovedkomponentene (TankMap, Backups, osv.);' sørge for at all logikk ligger i ViewModel, data lagrer i Provider/Model og view ligger i View	Implement authentication security
Sette opp API bibliotek	Fix double scrollbar (in Home.tsx)
Refactor whole TankList system, putting all logic in VM, data in Provider and just View in the View-files	Mobile view for approvable
Rewrite CSS to use rem instead of pixels	Logic/functionality in FilesList
Fix layout on iPad view	Backup files
Convert TankLists to continous scrollable list (infinite scrolling or virtual list)	Search, Sort and Filter in Backupfiles
Update TankList with features to change the tank properties	Add scaled view for phones
Feature to delete tanks	Fix memory leaks and redirects
Put all interfaces in one file	Robust and rich in functionality search filter
Implement Account view	Implement Homepage Dashboard
TankMap	Mobile view for backups
Change login style to redirect	Remove double scrollbars from page

Table A.2: Front End Scrumban Todo

Update Owner in Tank to use the new API endpoint	Kategorisere backups filer, pivottabs med production og repacking
Change Types (Interfaces + Variables) from any to the specific type where that's possible	AdminPage, user rights
By clicking on fish in TableView, info about that fish (and some transactions comes up)	Sort backups default by filename
Handle if SquarePack or Straw	Possible to remove content in tank if client is changed
Endpoint from API which says if the 47L tank has 6 or 10 cylinders	Feature to share (mail) approvable files and backups
Fix layout on bigger displays than 1080p	Testing av komponenter og logikk i ViewModel
FileSorting in Approvable to use more async functions	Transaction log page
Find a better solution for FileSorting in approvable, where it is NOT using two arrays of content (ViewFiles and Files)	Implement global search page
Confirmation Dialog to be styled as FluentUI	Fix layout on mobile view (using Firefox responsive design mode)

Appendix B

Project Agreement

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

Cryogenetics AS

(oppdragsgiver), og

Sander Låstad Olsen, Marthinus Bunesius
Klækken, Casper Fabian Bulbrandtzen,
Kristian Jegerud (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 01.21 til 07.21.

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
 - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon, reiser og nødvendig overnatting på steder langt fra NTNU i Gjøvik. Studentene dekker utgifter for ferdigstilling av prosjektmateriell.
 - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.
3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle beståtte bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv NTNU Open.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.
6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.
7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.
8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggrupeleder som godkjenner avtalen.
9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.
10. Når NTNU også opptretr som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.
11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:


NTNUs veileder (navn): _____

Oppdragsgivers kontaktperson (navn): Steffen Wolla

Student(er) (signatur): Casper F. Gulbrandsen dato 18.01.21

Martin G. Flokken dato 18.01.21

Kristian Jegøvd dato 18.01.21

SANDR L. OLSEN  dato 18.01.21

Oppdragsgiver (signatur): Steffen Wolla dato 26.01.21

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.

Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.

Plass for evt sign:

Instituttleder/faggruppeleder (signatur): _____ dato _____

Appendix C

Project Plan

Bachelor - Forprosjektrapport

Marthin G. Klækken,
Casper Fabian Gulbrandsen,
Kristian Jegerud,
Sander L. Olsen

January 2021

1 Background and Goals

1.1 Background

Our bachelor project has been assigned to Cryogenetics, a company specializing in cryopreservation of aqueous genetic material.

The goal has been made to automate process logging systems, which currently is manually done using large excel spreadsheets. The current workflow should be maintained, which means that excel files will be continued to be used to input data.

1.2 Project goals

Table 1: High-level project goals

#	High-level goal
HG001	Automate the current manual processing of excel sheets.
HG002	Provide a user-friendly view of the stored data.
HG003	The system should be easily maintainable. Cryogenetics is not a software company, and does not have a large IT staff. The system will likely be maintained by a 3rd party, and as such, good documentation of the systems is important.

HG004	Ensure appropriate confidentiality, integrity and availability. The data is considered confidential. The data is business critical, and is used for production, integrity is also critical. Certain steps of the production process is held to tight deadlines, as such availability is also important. However as the data is input with excel spreadsheets, short outages may not be disastrous. Still, availability is likely important. Further consultations with the company is required to decide a MTD and RTO ¹ .
HG005	Ensure non-reputability.
HG006	Provide a flexible, fish-centric ² view of the data.

Learning goals

During this project we want to learn more about and become better at working in groups. We will therefore focus on:

- Organizing and conducting digital meetings due to the corona virus situation.
- Using a formal management technique, such as Scrum-ban to manage a larger scale project.
- Using React to create a responsive and maintainable web application.
- Use CosmosDB to create a flexible database for storing the excel documents to store in a secure place.

1.3 Frameworks

- The web interface should work on all browsers supporting HTML5 and Javascript (ES5)
- The web interface should be scalable so that the website looks good for handheld devices
- Servers will be set up in Azure so that it can be delivered as a finished product to Cryogenetics

¹Maximum tolerable downtime, recovery time objective

²Instead of the data being viewed for each slot in the tanks, instead view the data as per-fish, with the tank location as a column. Essentially a transpose of the view

2 Scope

2.1 Subject field

Cryogenetics is a fast growing company and it is clear that their system using only Sharepoint and Excel documents is not a sustainable solution. In short, this process involves the manual creation of excel documents from predefined templates, where these are stored in folders in a common Sharepoint solution. These documents are manually updated and reviewed, where approved excel documents are manually placed in a folder of approved documents.

Our task will then be to automate parts of this process, make their data more secure and store this data in a safer place than in these sharepoint folders. It should still be possible for Cryogenetics to continue to use Excel and Sharepoint in their work process as this has great benefits for proper use. Therefore, we will design a database that makes it possible to upload excel documents with different structure and create an administrator page and approval system for these in the form of a website.

Our task is therefore not to manage the customer data. Nor should we create a user interface for entering data into the database, as the data should be parsed from excel documents allowing them to retain their current workflow.

The thesis will therefore cover many different technologies within our field:

- Responsive web applications using HTML, CSS and React
- RESTful API developed in .NET
- Design of database and data modelling
- iPad application developed using React Native
- Setting up servers in Azure
- Internationalization of both Web and iPad application
- Facilitate for people with disabilities

3 Organization

3.1 Organization chart

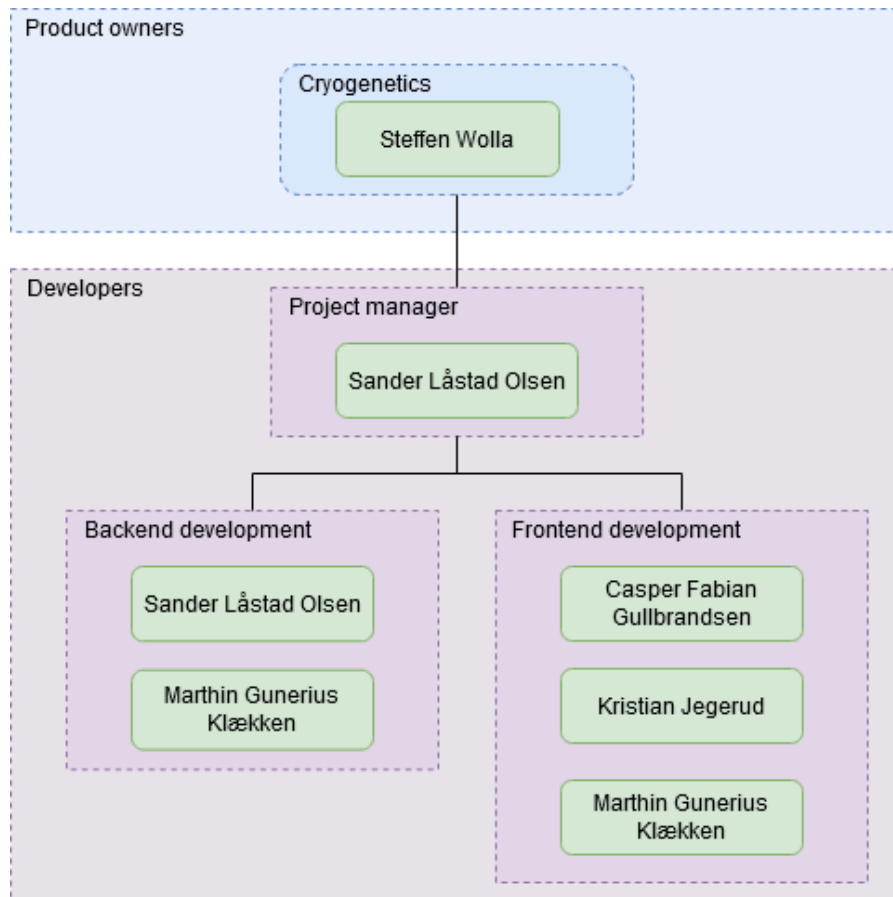


Figure 1:

3.2 Roles and responsibility

Product Owner: Steffen Wolla

Steffen Wolla represents Cryogenetics' interests and will act as the product owner of the project. His responsibilities in the development are to have a clear vision of what Cryogenetics wishes to be developed and convey this vision to us. He will be available to answer any questions about Cryogenetics' production routines and be present at our weekly meetings.

Project Manager: Sander L. Olsen

Sander L. Olsen will act as the Project Manager. He will be the link between the product owner (Steffen Wolla) and us. His responsibilities as the project manager will be to organize meetings on behalf of the team with Cryogenetics and/or the bachelor thesis advisor.

Project Security Manager: Marthin G. Klækken

Marthin G. Klækken will act as the Project Security Manager. His responsibilities will be to ensure that the development follows approved and secure development standards. He will implement security policies, regulations and rules to make definite sure that the team develops a product that satisfies the security standards Cryogenetics expects.

Project Development Manager: Casper F. Gulbrandsen

Casper F. Gulbrandsen is the development manager and will be responsible for setting up workable development environments. He will also ensure that the team follows the chosen development model and are using the same code standard. Gulbrandsen will maintain the GitHub repository and make sure that the team follows agreed upon Git standards.

Project Documentation Manager: Kristian Jegerud

Kristian Jegerud will be the Documentation Manager of the project. He will make sure that as much as possible of the development and administration is documented and made easily available to the team. He will take notes during meetings and ensure that team members log their hours accurately.

Additional Roles:

In addition to the above roles every team member will serve as developers, with Sander L. Olsen as the main back-end and database developer, with Marthin G. Klækken as additional assistance. Marthin G. Klækken, Casper F. Gulbrandsen and Kristian Jegerud as the main web-application developers and front-end work as shown in figure 1. Casper F. Gulbrandsen and Kristian Jegerud will also oversee the design of the website.

4 Organization of quality assurance

4.1 Documentation, standard use and source code

We want to deliver a product to Cryogenetics that is stable and solidly made, and it is therefore important that we take the time to establish good routines. This then means that we must document the project well, ensure well-written code and do what we can to secure our code in the best possible way.

Storing of documents

- All source code is stored in GitHub
- Report is written in LaTeX with Overleaf
- The LaTeX report is synchronized with GitHub as a backup
- Meeting notes and other important documents is uploaded to our Teams channel

Internationalization

The client has expressed a desire for internationalization of the service, as they have laboratories in the USA and Chile, among other places. We will therefore follow the standards for internationalization when we develop the service.

4.2 Development routines

We have agreed to use Scrumban as our development routine. Scrumban is a flexible agile method for continuous development, offering a smooth workflow which fits well for a smaller group who have different components of a larger system to create.

Combining a kanban board with the scrum development method gives us a prioritization-on-demand, which provide us with the more important components to work on next. Scrumban also offers us the prescriptive agile nature of scrum while also being able to use the process improvement of kanban to allow the team to continually improve the work process of various components. This is important for us as we have been given several different components of various priorities to develop.

We are also divided into two different small groups developing different components of the same system. We feel like this makes simple team collaboration a better alternative than having a scrum master for each team.

We are having weekly meetings with the product owner to give updates, receive updates, and to let each other know about further developments and planning. We also have frequent internal team meetings both physical and on Microsoft Teams, while also actively asking the advisor Aland for developing/architecture advice.

4.3 Tools

Name	Type	Area of use
Overleaf	Online LaTeX editor and compiler	Project report
GitHub	Provider of Internet hosting for software development and version control using Git	Version control
Visual Studio	Official IDE for development of .NET applications	Back-end development
Visual Studio Code	Free source-code editor	Front-end and back-end development
Docker	Software containerization platform	Containerization of application
Microsoft Azure, various IaaS, PaaS & SaaS	Cloud provider	Hosting of back-end servers and services
Toggl Track	Online time tracking and reporting services	Time registration
GitHub Project Board	Online Kanban tool for organizing and prioritizing the work	Project management
Expo	Platform for making universal native apps for Android, iOS and Web	iPad application
Microsoft Teams	Communication platform	Workspace chat, video-conferencing, files and notes storage, and task-management

4.4 Risk analysis

Nr	Event	Probability	Consequence	Action
1	Inadequate documentation	Likely	Problematic	Yes
2	Project not finished in time	Likely	Critical	Yes
3	Development costs go over budget	Unlikely	Problematic	Yes
4	Web service is not well enough developed and will not be used	Unlikely	Critical	Yes
5	Source code leakage	Unlikely	Unproblematic	No
6	Important persons in team becomes ill or absent for other reasons	Unlikely	Problematic	Yes
7	Loss of source code	Unlikely	Critical	Yes
8	Cryogenetics ends project	Unlikely	Critical	No
9	Issues arising communicating with the Sharepoint service, or the Sharepoint service going down	Unlikely	Problematic	No (out of scope)
10	Miscommunication / underspecification, end up building something the customer does not want.	Likely	Critical	Yes
11	Feature-creep, new features and changes keep being added, preventing progress on the project.	Likely	Critical	Yes

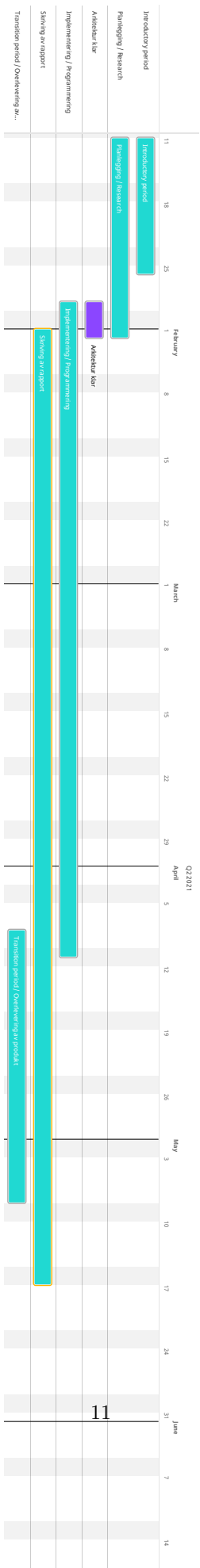
4.5 Risk Management

Nr	Solution
1	If we see that we are on the verge of ending up with inadequate documentation, we have to make a quick assessment of whether the documentation should be prioritized before other and less important parts of the project. This is because we will hand over a product that we will not continue to operate, and the company must then have good documentation of the product that they can use further.
2	If we see that we will not finish the entire project within the time frames we have decided, we will have to talk with client and find a solution. This will then involve a solution to remove parts of what is left of the project and give the client a way forward after our project is finished.
3	Go over the costs that have been set up and look at solutions where we can cut down on the budget. This might be more flexible uptime for servers or less space on the servers. Another relevant solution would be to discuss the budget with the client, and find out whether it would be relevant to expand the budget.
4	Do a thorough job in the research phase while at the same time constantly work closely with the client so that the quality of the product meets the requirements. Beyond that, it is our responsibility to do good enough research and learn the technology well that the product is as good as possible.
6	Since we do not have the opportunity to bring in a replacement, we must therefore exchange expertise and share information with each other so that if someone should go missing, someone else has the necessary expertise needed.
7	The source code is stored in Github, a well-known version-control system owned by Microsoft, which makes us consider it a safe place to store the source code. We will also make sure to have important parts of the code stored as backups, preferably on physical hard drives at home.
10	Have frequent meetings with the customer, and use a agile development methodology, allowing for changes to be made underways to suit the customers needs.
11	Prioritize what the customer wants, and make sure the client is aware of what is possible, and what is not. Freeze the architecture

5 Plan for implementation

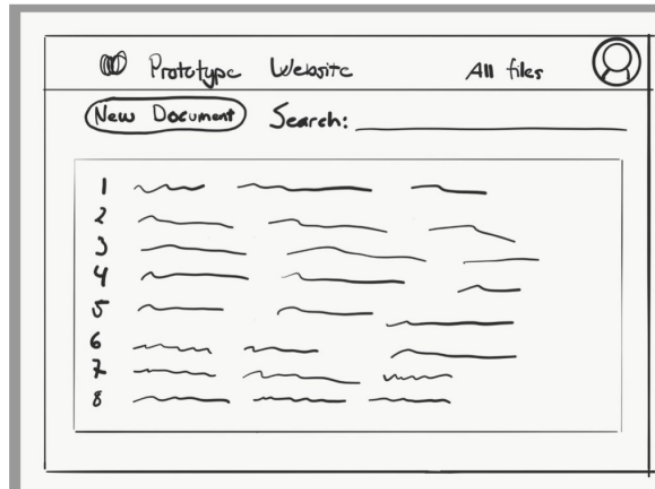
5.1 Gantt diagram

Se figure 5.1.

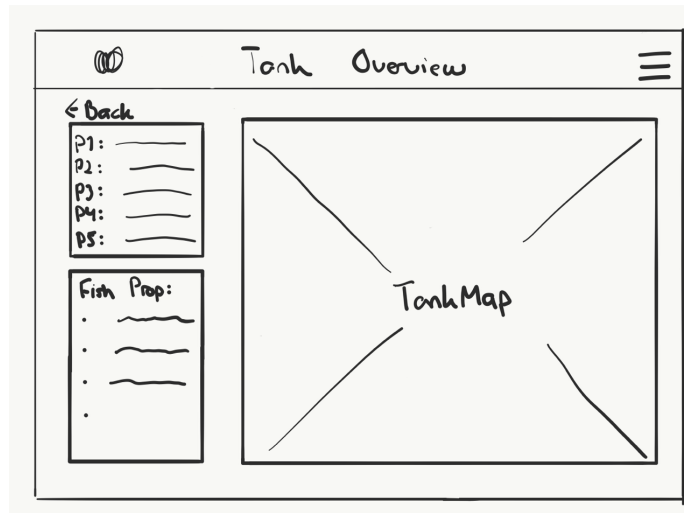


Appendix D

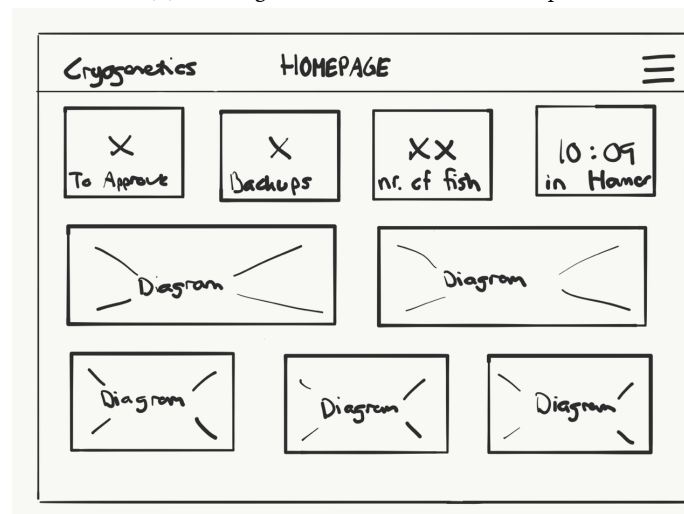
GUI-workshop



(a) UI design wire frame of the list of files

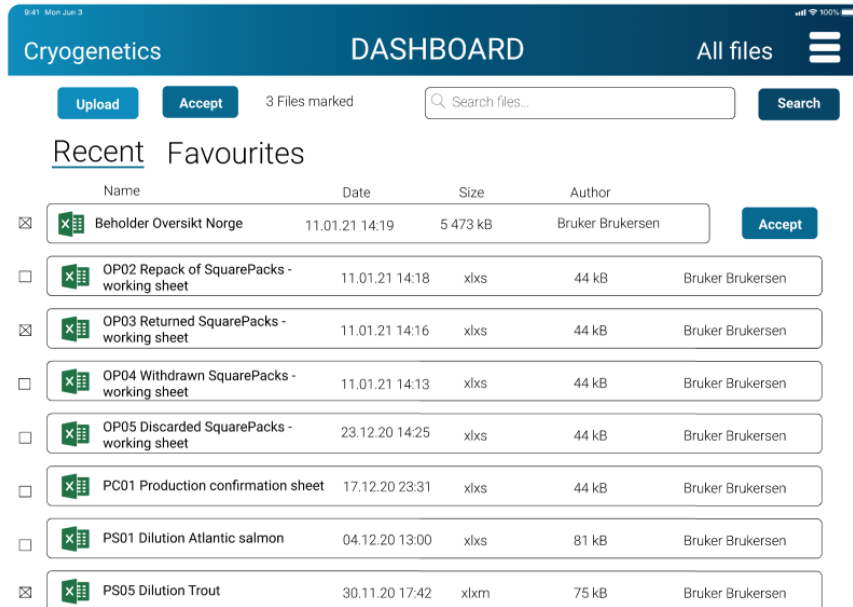


(b) UI design wire frame of the tank map

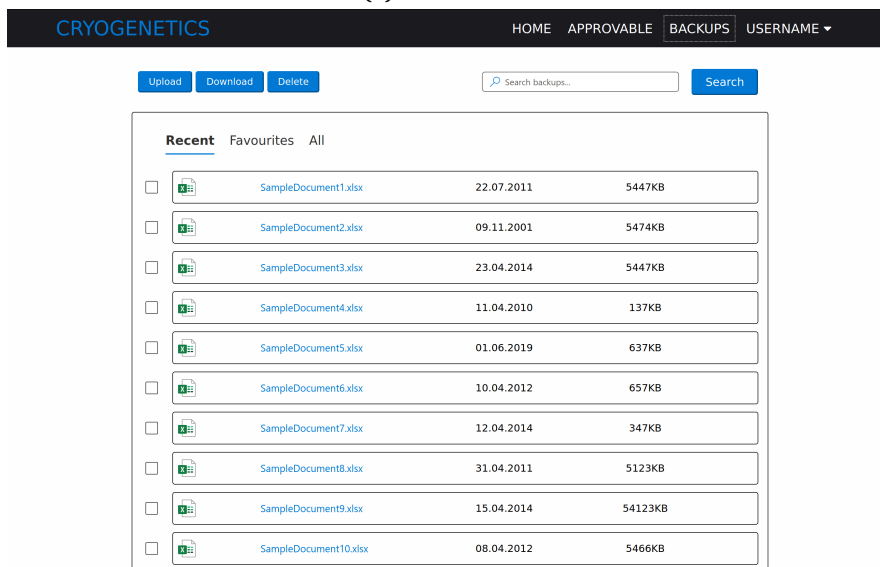


(c) UI design wire frame of the dashboard/homepage

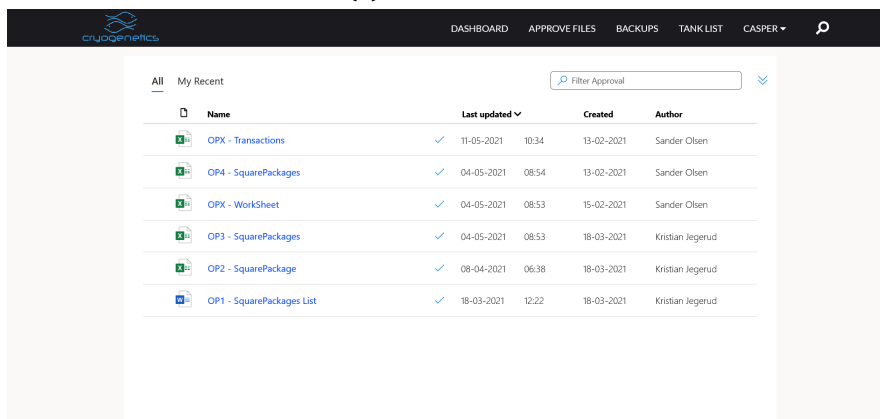
Figure D.1: Wireframe designs



(a) First iteration

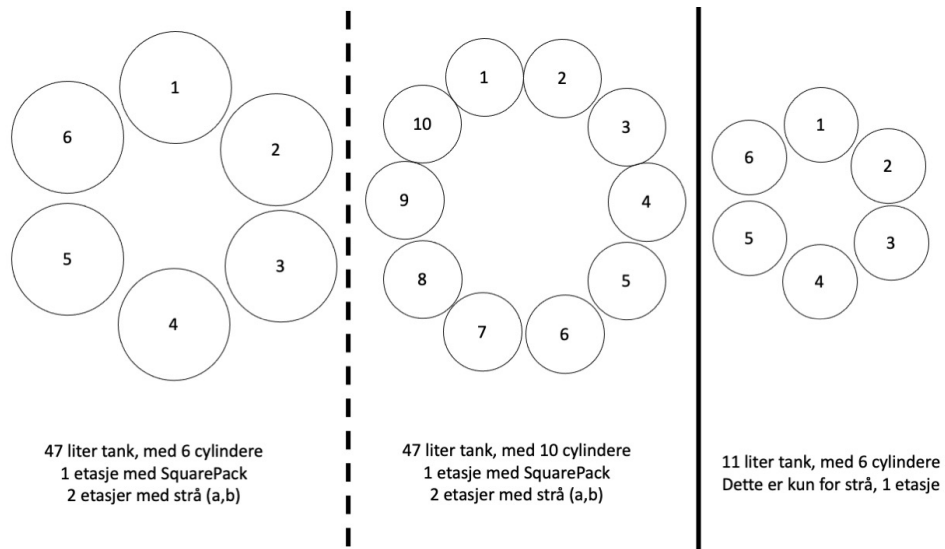


(b) Second iteration

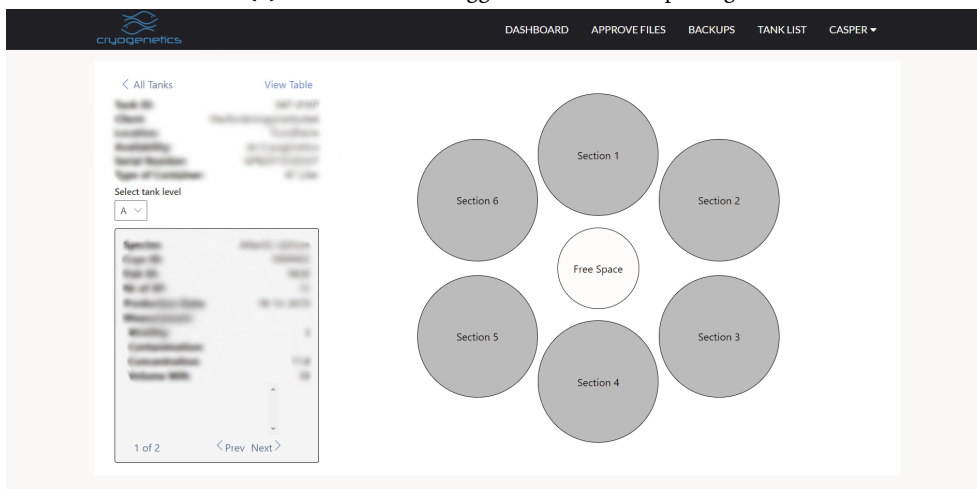


(c) The final design

Figure D.2: Design iterations of the approve files list



(a) Product Owner suggestion of tank map design



(b) Implementation of PO's suggestion

Figure D.3: From suggestion to implementation

Appendix E

Rough initial recurring price estimate

Microsoft Azure Estimate

Overslag, kan effektiviseres.

Service type	Custom name	Region	Description	Estimated monthly cost	Estimated upfront cost
App Service		West Europe	Premium V3 nivå; 1 P2V3 (4 kjerne(r), 16 GB RAM, 250 GB lagring); 3 år reservert; Linux OPERATIVSYSTEM	kr948.02	kr0.00
Azure Cosmos DB			Autoskaler klargjort gjennomstrømming, Skrivning til enkelt område (enkelmaster) - West Europe; 4 x 1 000 RU/s x 730 Timer x 20 % gjennomsnittlig bruk; Tilgjengelighetssoner aktivert; 10 GB lagring	kr731.08	kr0.00
Storage Accounts		West Europe	Block Blob Storage, Generell bruk V2, GZRS redundans, Varm tilgangsnivå, 200 GBKapasitet - Betal for forbruk, 1 000 000 Skrivehandlinger, 100 000 liste- og beholderoppretting, 10 000 000 lesehandlinger, 100 000 arkiv for høyt prioriterte lesehandlinger, 1 000 andre operasjoner, 1 000 GB Datahentning, 1 000 GB arkiv for høyt prioritert datahentning, 1 000 GB dataskrivning, 1000 GB dataoverføring med georeplikering	kr214.42	kr0.00
Azure Kubernetes Service (AKS)		West Europe	2 D2 v3 (2 vCPU-er, 8 GB RAM) noder x 730 Timer; Betal for forbruk; 1 administrerte OS-disker - E2, 1 klynger	kr2,018.79	kr0.00
Service Bus		West Europe	nivå Standard: 20, 1 lytter(e) via hybridtilkobling + 0 overforbruk per GB, 720 relétime(r), 10 000 relémelding(er)	kr292.06	kr0.00
Bandwidth			Internett - utgående, 50 GB utgående dataoverføring fra West Europe rutet via Microsoft Global Network	kr31.95	kr0.00
Key Vault		West Europe	Hvelv: 500 000 operasjoner, 10 000 avanserte operasjoner, 4 fornyelser, 4 beskyttede nøkler, 4 avansert beskyttede nøkler; administrerte HSM-utvalg: 0 standard B1 HSM-utvalg x 730 Timer	kr305.50	kr0.00
Container Registry		West Europe	Standard nivå, 2 enheter x 30 dager, 10 GB båndbredde, 0 GB Ekstra lagring	kr375.04	kr0.00
Support			Support	kr811.41	kr0.00
			Licensing Program	Microsoft Online Services Agreement	
			Total	kr5,728.27	kr0.00

Disclaimer

All prices shown are in Norwegian Krone (kr). This is a summary estimate, not a quote. For up to date pricing information please visit <https://azure.microsoft.com/pricing/calculator/>
This estimate was created at 1/18/2021 12:37:59 PM UTC.

Figure E.1: Rough initial recurring price estimate

Appendix F

Initial SQL draft

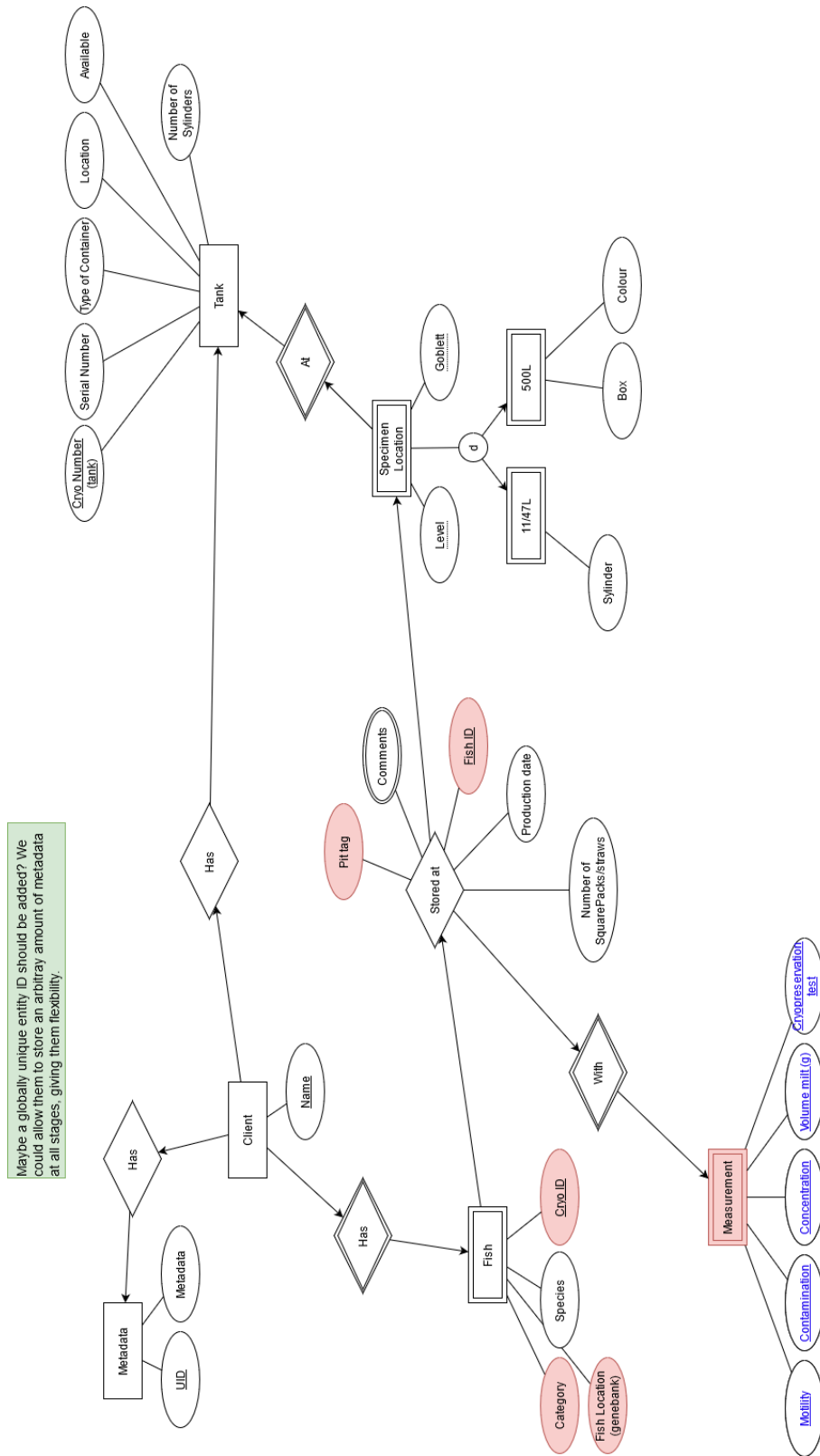


Figure F.1: Rough initial EER diagram design

Appendix G

Swagger UI, and authorization configuration

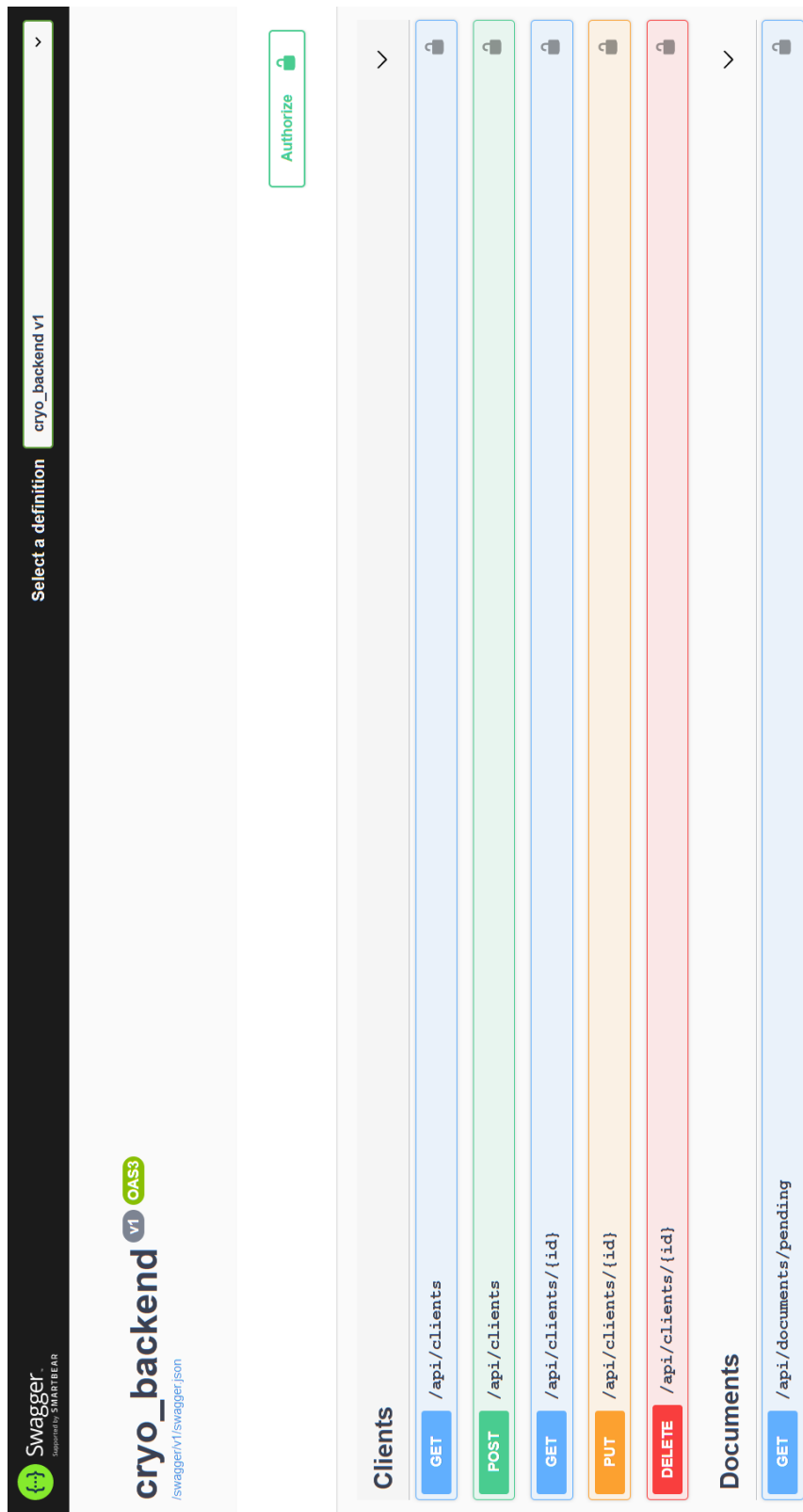


Figure G.1: Swagger UI showing some endpoints, and link to specification

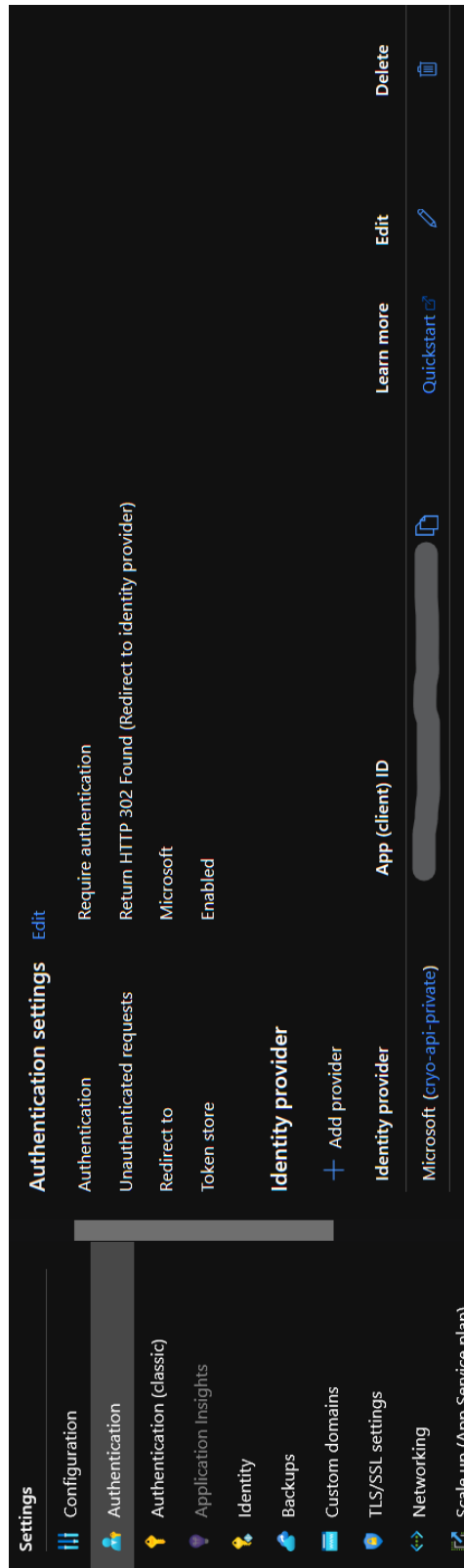


Figure G.2: Authentication on Swagger development site as configured through the Azure Portal.

Appendix H

Managed identity

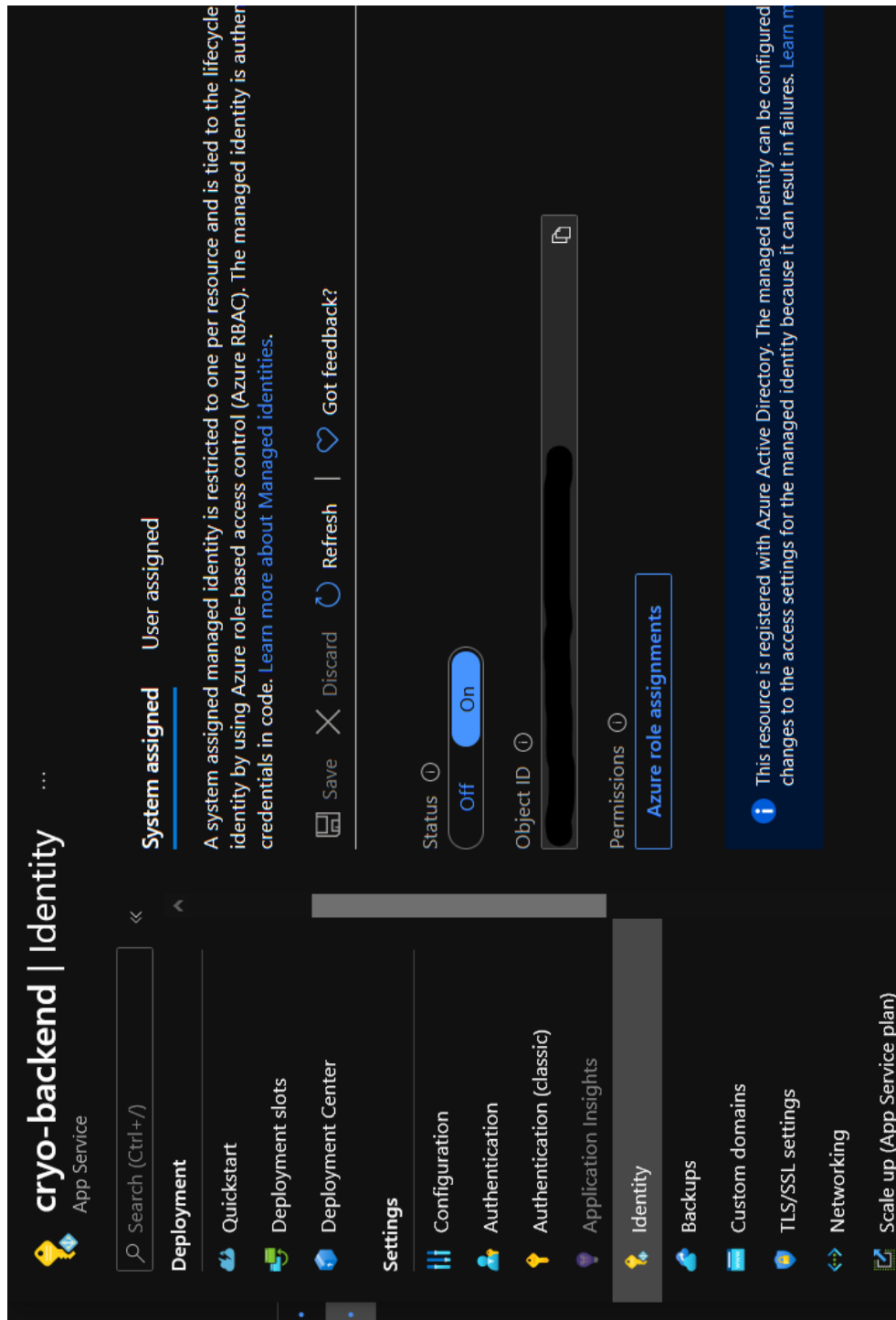


Figure H.1: Managed identity enabled on App Service

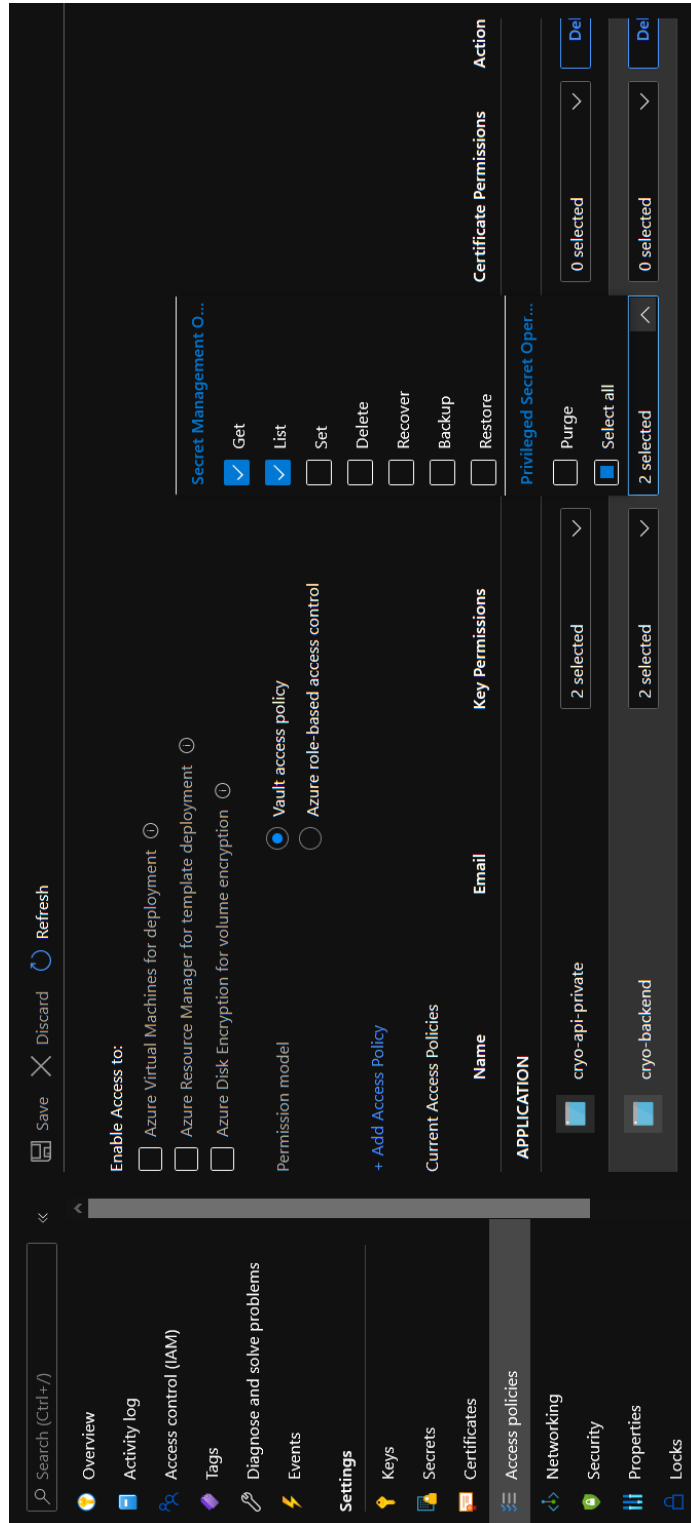


Figure H.2: Access policy of keyvault with managed identity

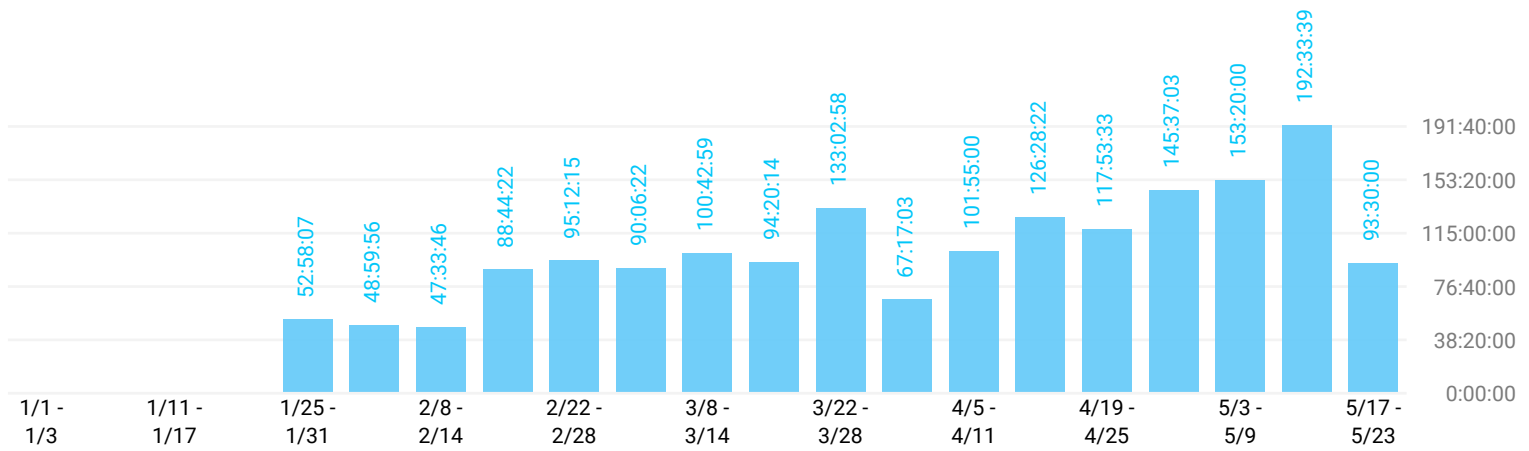
Appendix I

Toggl Track Summary Report

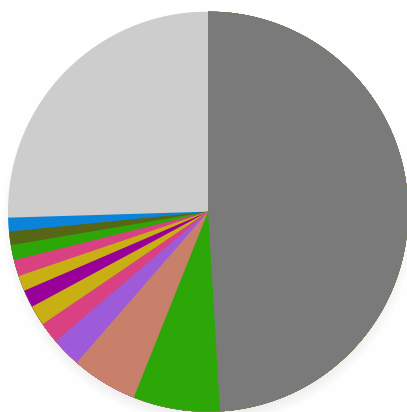
Summary Report

01/01/2021 – 05/20/2021

TOTAL HOURS: 1750:15:39





USER	DURATION
SA Sanderlo	486:02:17
MA Marthigk	431:50:00
KJ Kristian Jegerud	420:15:00
CA Casperfg	412:08:22



TIME ENTRY	DURATION
Without description	857:39:06
Thesis writing	123:00:00
Account page development	93:20:00
Tankmap	43:25:00
Review	29:35:00
Refresh token development	27:30:00
Morgenmøte	24:09:12
Material-UI/Account View research	24:00:00
Tanklist	21:25:00
Login page development	21:00:00
Protected Routes development	21:00:00
Implementation	19:30:00
Other time entries	444:42:21

USER - TIME ENTRY

DURATION

USER - TIME ENTRY	DURATION
 Casperfg	412:08:22
Design Pattern	1:23:16
Design prototype	3:35:00
Docker setup	1:00:00
Fiks av headerbar	0:00:16
Gruppemøte	3:44:00
Jira setup	1:40:00
Mandagsmøte	0:46:00
Morgenmøte	13:54:12
MVVM	1:17:00
Møte med Cryogenetics	5:33:00
Prototyping	5:12:24
Without description	374:03:14
 Kristian Jegerud	420:15:00
Abstract	1:30:00
Administrativt møte	0:15:00
API Meeting	2:00:00

USER - TIME ENTRY
DURATION

USER - TIME ENTRY	DURATION
API og Frontend	10:55:00
Backup view	2:05:00
Bugfixing	13:00:00
Bugfixing, Authentication	0:45:00
Bugfixing, Docker	1:35:00
Bugfixing, Search	1:15:00
Checkboxes on file list	2:40:00
Code Quality	5:00:00
Conclussion	11:00:00
ContextAPI	2:30:00
Design	6:05:00
Design enhancements	1:00:00
Design fixes	1:30:00
Development Environment	4:45:00
Dialogs on files	1:30:00
Docker problemløsning	0:15:00
Etterarbeid av møte	0:30:00

USER - TIME ENTRY
DURATION

USER - TIME ENTRY	DURATION
FilesListView	6:15:00
FilesListView, not finished	3:45:00
Filter, minDate og maxDate	3:00:00
Fixed commented by others	1:45:00
Forberedelse til møte	1:00:00
Gjennomgang av rapport	1:30:00
GlobalSearch	14:45:00
Good practices and tips for React/Typescript	1:30:00
Gruppemøte	3:25:00
Homepage	1:15:00
Håndtering av OAuth key (cookies)	0:30:00
Implementation	19:30:00
Improving Code Quality	5:30:00
Introduction	6:10:00
Mandagsmøte	1:00:00
Morgenmøte	10:15:00
MVVM in React	4:15:00

USER - TIME ENTRY
DURATION

USER - TIME ENTRY	DURATION
Møte med Cryogenetics	2:05:00
Møte med veileder	3:45:00
New design for FilesList	4:15:00
Physical meeting with Cryogenetics	3:00:00
Preparations	1:00:00
Presentation API	0:30:00
Propertiesbar & Backup view	1:55:00
React & FluentUI	12:15:00
React and CSS	1:00:00
React, FluentUI	5:15:00
Refactoring	16:45:00
Refactoring in FileSorting	1:30:00
Repository and Kanban organization	3:15:00
Requirements Analysis	0:35:00
Research API Fetching	1:00:00
Research ContextAPI	3:00:00
Research for thesisreport	4:45:00


USER - TIME ENTRY

DURATION

USER - TIME ENTRY	DURATION
Research Implemented MVVM	3:45:00
Research Mobx architecture	10:00:00
Research Redux & ContextAPI	2:50:00
Research, Spreadsheets in React/TS	11:45:00
Research/Implementation MVVM	6:30:00
Review	29:35:00
Search and Filter in backups	2:00:00
Search filter	6:20:00
Sorting & searching	3:00:00
Sorting of files	6:15:00
Tanklist	21:25:00
Tankmap	43:25:00
Tanktable	4:00:00
Technologies	5:00:00
Temp Dashboard	1:50:00
Testing	11:00:00
Torsdagsmøte	0:15:00

USER - TIME ENTRY
DURATION

User Interface	10:40:00
Various	1:00:00
Weekly meeting with Cryogenetics	1:00:00
Working on project plan	5:45:00
Workshop med Cryogenetics	0:45:00
Writing Scope, Risk analysis	3:00:00
Without description	7:55:00

 Marthigk	431:50:00
401 error research on account page	6:00:00
Account information view development	5:00:00
Account page development	93:20:00
Azure authentication research	12:00:00
Backend research	2:00:00
Cookies/storage research	8:00:00
discussion about UI setup	4:00:00
Forprosjektrapport-skriving	4:00:00
Login page developement and merging	4:00:00


USER - TIME ENTRY

DURATION

USER - TIME ENTRY	DURATION
Login page development	21:00:00
Login page redirect development	16:00:00
Login page redirect research	4:00:00
Login page research	2:00:00
Logout functionality development	11:00:00
Material-UI/Account View research	24:00:00
Microsoft Graph API useEffect research	8:00:00
Protected Routes development	21:00:00
Protected Routes research	6:00:00
React research	1:00:00
Refresh token development	27:30:00
Refresh token research	9:00:00
Researched storage/state/useeffect	7:00:00
Researching conversion of excel to Json	2:00:00
Researching react-redux	1:00:00
Thesis writing	123:00:00
UseEffect/LocalStorage research	10:00:00

USER - TIME ENTRY

DURATION

 Sanderlo	486:02:17
Backend	8:00:00
Project plan	2:21:25
Without description	475:40:52

