FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialisation: Computer Science	The *autumn* semester, *2022* **Open** / Confidential
Author: Liv Selle Underhaug	
Supervisor at UiS: Rui Esteves External supervisor: Dan Edvard Halvorsen	
Thesis title: Using Agile Scrum to Develop a Containerized Asset Management System	
Credits (ECTS): 20	
Keywords: Asset Management System, Agile Scrum, Containerization, .NET, React, Relational Database	Pages: 93 Stavanger, *December 15th 2022*

Approved by the Dean 30 Sep 21
Faculty of Science and Technology

0.1 Abstract

This thesis describes the process of developing a pilot for an asset management system. The purpose of this thesis is to demonstrate how to go from a problem to an implemented solution. The problem is solved by using *Agile Scrum,* as a project management framework. Firstly, I have identified the requirements and features of a potential solution. Secondly, I have created a system architecture specification on how to implement the potential solution. Finally, I have implemented a full stack application which enables the IT department of Bouvet to register (assets) and track assets (loans) in a dynamic way. The solution involves a user interface with a dashboard of the company's assets and active loans. Moreover, functionalities to perform operations involving the assets. The frontend communicates with the backend through a REST API. The backend handles and processes the requests received from the frontend and communicates with the database. This thesis focusses on how to create maintainable code with a seamless handover in mind.

Link to source code and documentation on GitHub:

https://github.com/livun/bouvet-asset-hub

Link to project on GitHub Projects:

https://github.com/users/livun/projects/1

0.2 Acknowledgements

First, I would like to thank my supervisor Rui Esteves, for valuable guidance and expertise throughout the thesis project.

In addition, I would also like to thank Bouvet for giving me the opportunity to develop and write my thesis for them. I especially want to thank my supervisor at Bouvet, Dan Edvard Halvorsen, for valuable advice and support.

Table of Contents

0.3 List of Acronyms and Abbreviations

API	Application Programming Interface
Azure AD	Azure Active Directory
C#	Pronounced “See Sharp”
CSS	Cascading Style Sheets
EF Core	Entity Framework Core
FP	Functional Programming
HTML	HyperText Markup Language
JS	JavaScript
LINQ	Language-Integrated Query (C#)
MSSQL	Microsoft SQL
OOP	Object-Oriented Programming
ORM	Object-Relational Mapper
REST	Representational State Transfer
SPA	Single-Page Application
SQL	Structured Query Language
TS	TypeScript
TSQ	TanStack Query
UI	User Interface

1 Introduction

In this chapter I will begin with introducing the problem I was challenged to solve by Bouvet. I will also introduce the company, relevant work, and the defined goals of the thesis.

1.1 Problem

During the Covid-19 pandemic, Bouvet lent out a significant part of its IT-equipment for home office use. This resulted in a lack of control and overview on where the equipment was. There was no procedure or controls for tracking the equipment that was lent out when the pandemic broke. When the time came for the employees to return to the office, a lot of equipment was missing. As a result of this, the company identified a need for a better system that keeps track of its equipment. The pandemic caused extraordinary circumstances and does not represent the usual way of lending out equipment, but it recognised that the existing system did not work.

The existing system consists of manually updated Excel spreadsheets. When new equipment arrives to the office, they must be added to the spreadsheet. When equipment is lent out to employees, entries are made in the same spreadsheet. This process especially relates to computers, other types of equipment are not being tracked systematically. Bouvet emphasized that this system is inefficient, time consuming, manually intensive and does not include all kind of equipment.

In summary, the problem I am trying to solve, is: “How can Bouvet keep better track of its equipment?”

1.2 About Bouvet

Bouvet is a Norwegian consultant company, established in 2002, that delivers digital solutions to its clients. With more than 1800 employees in 17 offices, in Norway and Sweden, the solutions involve technology, design, communication and project management.

Bouvet Felles assists the Bouvet group with several in-house services. Such as the internal IT department, which is responsible for all things related to the operation and security of Bouvet’s environments. The IT department is primarily the client and user of the solution trying to solve the problem identified in the previous section. Bouvet Felles also has stakes in the solution, thus making them the client as well. In the next paragraph, I will present why they have interests in the solution.

Starting from January 1, 2023, Bouvet will implement a new Enterprise Resource Planning (ERP) system called Xledger. Initially, the system will concern salary and accounting, but it will also be relevant for my solution. In trying to solve the asset tracking problem, Bouvet Felles wants an integration towards Xledger, that notifies it when an asset is connected to an employee. This feature is relevant for cost accounting and saves Bouvet Felles from extra manual work.

1.3 Goal

The goal of this thesis is to develop a solution for the Bouvet's IT department, to keep track of its assets and ease their workload. The solution must reduce the amount of manual work. To prove this solution, a functional pilot should be implemented. This pilot must be designed and implemented to allow future extension of functionalities. The project and the code must be developed using principles that allows an efficient handover to Bouvet.

1.4 Outline of thesis

- Chapter 2 contains the background and theory this thesis is based on.
- Chapter 3 contains the experiment, i.e., how the application was planned, developed, and deployed.
- Chapter 4 contains the results of thesis.
- Chapter 5 concludes the thesis.

2 Background and Theory

In this chapter I will present the background and theory relevant for this bachelor thesis. I start by presenting some technologies and libraries that I found necessary to the development of my application. Then, I elaborate on the theory of Agile Scrum, a framework for project management often used when developing software. I will also go into how it is used in project. Finally, principles of design and development are explained.

2.1 Technology and Libraries

When developing an application, there are several technological decisions to be made. For example: which technology to use when developing the frontend and the backend, how to store the data, how to take care of security, how the user and possible 3rd party systems interfaces the application, and how and where the application is deployed. In this section, I will go through the technology used, in a section 3.2, the justification of these choices will be made.

2.1.1 Frontend

The frontend is built using the JavaScript framework React, together with libraries as Material UI and TanStack Query. I will now elaborate on these libraries.

2.1.1.1 React and TypeScript

React[1] is a free and open-source JavaScript library for building interactive component-based user interfaces (UIs). It is one of the most frequently used web framework and it is created and maintained by Meta, formerly known as Facebook [1]. React is written in JavaScript (JS) but can also be used with TypeScript[2] (TS). TS is a programming language created by Microsoft. TS is built on JS and adds additional syntax to the JavaScript scripting language. TS is a strongly and statically typed programming language. Strongly typed means that the language has strict typing rules and does not allow unrelated types to be converted. Statically typed implies that the language verifies and enforces the constraints of types on values [2]. All this adds type safety to code. TS comes with its own compiler and compiles to JS, thus reducing the chances of errors and bugs at run time, catching them at build time. To generate the boilerplate version of a React TS single-page application (SPA), one can use the

[1] React documentation: https://reactjs.org/docs/getting-started.html
[2] TypeScript documentation: https://www.typescriptlang.org/docs/

Create React App toolchain. A SPA loads a single web document with all necessary assets that are required for the application to run, therefore interactions with the page do not require loading new pages from the server. [3]–[6]

2.1.1.2 MUI – Material UI and MUI X

Material UI[3] is a suite of user interface (UI) tools to help create UIs fast. MUI is an open-source library of free and ready to use React components that implements Google's Material Design. Material Design is a design system. Mui X is a library of advance components with more complex use cases. [7]–[10]

A React component is a reusable piece of code, like a JavaScript function, only it returns HTML. With components, we can split the UI into independent pieces [11].

2.1.1.3 TanStack Query and Axios

TanStack Query[4] is an open-source library that handles state management, caching and data fetching [12]. TanStack Query can be used together with Axios[5], which is an open-source HTTP client library based on promises. It is used for sending asynchronous HTTP request to REST endpoints. [13]

HTTP stands for *Hyper Text Transfer Protocol* and is a client-server protocol for fetching resources. An HTTP request is a message that contains a method (GET, POST, PUT or DELETE), a path, headers, a protocol version and possibly a body with data. An HTTP response contains a status code, which indicates a successful response or not. Furthermore, a status message, protocol version, headers, and possibly a body with the requested data. HTTPS is an encrypted version of HTTP, where the S stand for Secure. [14], [15]

[3] MUI documentation: https://mui.com/
[4] TanStack Query documentation: https://tanstack.com/query/v4/docs/overview
[5] Axios documentation: https://axios-http.com/docs/intro

2.1.2 Backend

The backend is developed using the ASP.NET Core 6 Web API complemented with Entity Framework Core. I will now present these technologies.

2.1.2.1 ASP.NET Core 6 Web API

The ASP.NET Core 6 Web API[6] is a framework for building HTTP services. It is a platform for building RESTful applications on the .Net Framework with the programming language C#. A RESTful API uses HTTP request to access and use data, and follows the architectural constraints of *Representational State Transfer* (REST). REST provides standards between physically separated systems, for example a client and a server. [16]–[19]

2.1.2.2 Entity Framework Core

Entity Framework Core[7] (EF Core) serves as an *Object-Relational Mapper* (ORM) which enables working with a database using .NET objects and *Language-Integrated Query* (LINQ). LINQ is querying capabilities in C#. EF Core simplifies the interaction between the backend application and the database. [20], [21]

2.1.3 Database

Microsoft SQL[8] (MSSQL) is a relational database management system. As a database, the primary functionality of MSSQL is to store and retrieve data [22]. The Structure Query Language (SQL) is the standard language used to manage a relational database and the data stored in them [23]. A relational database is a type of database that is optimised for storing data organised following the relational model. The relational model is developed by Edgar F. Codd and based on the mathematical set theory. The word relation can be used synonymously with table. The columns describe the relation's attributes, and each column can only have values from one type, such as an integer, string, or a date etc. In the relational model, the columns order has no effect on the meaning of the data. The row of data in a table is an instance of a relation. There can only be one value at the intersection of a column and a row, there are no duplicate rows, and a primary key is a value or combination of values that

[6] ASP.NET Core 6 Web API documentation: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-6.0
[7] Entity Framework Core documentation: https://learn.microsoft.com/en-us/ef/
[8] Microsoft SQL documentation: https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16

uniquely identifies each row, and thus make the row unique. As with the columns, the order of row is insignificant. [24]

One of the advantages of using the relational model is the use of normalization. "Normalization is a formal process for deciding which attributes should be grouped together in a relation" [25, p. 184]. A major purpose of normalization is to reduces data redundancy [25].

2.1.4 Azure Active Directory

Azure Active Directory[9] (Azure AD) is a cloud-based identity and access management service. It is used to manage authentication (confirmation of identity) and authorization (permission to access) of users and groups to services and resources. [26]

2.1.5 Containerization

Containerization provides the possibility to run an application on any environment or infrastructure, independent of host operating system, and does not rely on what is currently installed on host. A container contains libraries, frameworks, and dependencies necessary to run the application, as it is a means of packing together software code. [27]

Docker[10] is an open-source technology and it is the default container format, providing an open standard for packing and distributing containerized applications. A Docker Image is a template with instructions for creating a Docker Container at runtime. Containers are runnable instances of an image. [28], [29]

[9] Azure Active Directory documentation: https://learn.microsoft.com/en-us/azure/active-directory/
[10] Docker documentation: https://docs.docker.com/

2.2 Agile Scrum

Agile Scrum is a framework for project management, often used when developing software. I will now present a summary of one configuration of Agile Scrum according to Vanderjack [30], and following with an explanation of how it is used in this project.

2.2.1 Theory

One can identify three phases in Agile Scrum. The first and second phase take place at the start of a new project and can be seen as planning phases. The third phase is an iteration of steps that are repeated throughout the life of the project and called the *agile iteration cycle*. Vanderjack [30] also recognises two distinct roles, specifically the *Product Owner* (PO) and the *Scrum Master* (SM). The PO represents the client who needs a solution, and the SM leads the developing team through the Agile Scrum phases.

At the first phase of Agile Scrum, the PO and SM work closely to identify the problem and needs from a solution. In this phase scope is defined and ideas are brainstormed to solve the problem. The SM and team capture and refines what is requested from the project. These requirements are written in so-called *user stories*. A *user story* is an informal narrative of a feature from the perspective of an end-user.

At the second phase of Agile Scrum, the *backlog* of user stories is processed, and the user stories are decomposed into smaller and more specific user stories, and eventually broken up into concrete tasks. A task provides information on what to create, and sometimes time estimates on how long it will take to finish the specific task.

When the two first phases are completed, the last and iterative phase can begin. At this point the developing of a solution starts. One iteration, also called a sprint, is set to a defined time frame depending on the size and scope of the project. Each sprint consists of these steps:

- At the first step, the sprint is planned, thus choosing and prioritizing tasks from the project backlog into a sprint backlog.
- At the second step, the tasks in the sprint backlog are carried out. Designing, developing, and unit-testing take place.
- At the third step, environment testing happens, thus testing code in combination with other code modules or even the entire system.
- At the fourth step, the sprint deliverables are demonstrated, to the team and occasionally the PO.

- At the fifth step, the sprint tasks are declared done by some predefined rules and code is moved to production. When the code is in production, it is available for people outside the team. If bugs are identified, they will become new tasks and added to the backlog.
- At the sixth and last step the sprint retrospective is carried out. The team evaluates what went well working during the sprint and what areas need to be improved.

Moving on after the iteration steps are completed, the team will start a new iteration with the evaluation from the retrospective. Alternatively, the project is declared finished. [30]

2.2.2 Implementation of Agile Scrum

I used Agile Scrum as an instrument to carry out the project and fulfil the solution to the problem presented in section 1.1. My work was integrated in a project where other people contributed. In the beginning of the project, my technical supervisor Dan Edvard Halvorsen's role was PO, where he facilitated the first meetings with the client, Bouvet's IT department. Throughout the project I took on the various roles identified in Agile Scrum. My role was PO when I communicated with the client without Dan. When I facilitated and planned the project, thus carried out the two first planning phases recognized in previous section, I took the role as SM. When going through the agile iteration cycle, I had the role as developer.

I used GitHub Projects as a tool for planning and tracking the work of the project. Among its features, it has a *Kanban* board. A Kanban board visually shows the status of the work at the various stages of the project execution. GitHub Projects complement the Agile Scrum way of developing, allowing the user to add the features and tasks to the Kanban board backlog stage. More details on how GitHub projects and the last two phases of Agile Scrum were carried out, will be presented in the third chapter, Experiment.

2.3 Design Principles and Patterns

To develop robust, testable, and maintainable software in a constantly changing environment, there exist a set of design principles and patterns, to avoid common problems. In this section, I will first define what is *maintainability* and some metrics that can be used to uphold the maintainability. Secondly, I will introduce two paradigms of programming, which are *Object-Oriented Programming* and *Functional Programming*. Thirdly, I will describe the object-oriented design principles, also known by the acronym SOLID. Lastly, I am going to introduces a few design patterns I used in this project.

2.3.1 Maintainability, Coupling and Cohesion

Maintainability is defined as "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" [31, p. 46].

Low *coupling* and high *cohesion* are two metrics that indicates that software is maintainable. Coupling concerns how modules depend on each other. A *module* is a code section, thus a part of a program that is assigned specified functionality. Low coupling means a low degree of dependency, and high coupling mean high degree of dependency. In code on should strive for low coupling, thus changes to a module will decrease the impact on other modules. Cohesion measures the level of how related the element in a module are. The aim is to achieve high cohesion, which ensures that related code is close to each other. [32], [33]

2.3.2 Object-Oriented Programming and Functional Programming

C# is a type-safe and essentially an object-oriented programming language. Object-Oriented Programming (OOP) is a software modelling paradigm that is based on objects that has properties and methods. The property of an object describes it characteristics, and the methods describe its behaviour. Objects can be used to represent real-world items and concepts, that interacts to solve problems. A class is a template that defines the objects that share the same set of properties and methods. Classes are used to instantiate and create an object. [34], [35]

Functional Programming (FP), as opposed to OOP, is based upon functions, rather than objects. The software is built upon applying and composing functions. Functions promotes immutability, meaning the objects state cannot change after its created. There are several

types of function, some of them are, *pure functions*, where output depends solely on the input and there are no side effects. A side effect means that the function modifies the context outside of itself. There are *higher-order functions* that take other functions as input, and *monads* returns a normal value as a wrapped value. [36]

These two ways of programming does not necessarily exclude one another, they can coexist, and thus also complement each other. The C# language supports functional programming by using for example LINQ technologies, which is a form of functional programming, and libraries like *language-ext* that provides functional programming capabilities to C#. [35], [37], [38]

2.3.3 SOLID

The SOLID design principles of object-oriented software development were first introduced by Robert C. Martin in "Design Principles and Design Patterns" [39]. These principles are used to create simple, understandable, flexible, reliable, and evolvable software [40].

First, S – *The Single Responsibility Principle*, a class should never have more than one reason to change. Thus, only have one purpose and thus one responsibility. Second, *O – The Open Close Principle,* software entities, classes, modules, and function, should be open for extension, but closed for modification. Third, *L – The Liskov Substitution Principle*, "derived classes should be substitutable for their base classes" [39, p. 8], meaning that a functions that use a base class, must be able to use a derived instance of a base class. Fourth, *I – The Interface Segregation Principle*, client interfaces should be specific, so they do not depend on interfaces they do not use. Fifth and last principle, *D – The Dependency Inversion Principle*, dependencies should target interfaces or abstract classes, not a concrete class. [39]–[41]

2.3.4 Design Patterns

"Design patterns is a general reusable solutions that can be applied to commonly occurring problems" [35, p. 18]. The *mediator pattern*, is a behavioural pattern that "defines an object that encapsulates how a set of objects interacts, reducing their dependency on one another" [35, p. 65]. It is used for decoupling objects, as the mediator works as a message broker and the objects only communicate through the mediator object.

In this project I have used the MediatR package, which is a simple mediator implementation in .NET. This implementation supports request messages, which is dispatched to a single

handler, and notification messages which is dispatched to multiple handlers. The handler receives and solves the message. [42]

The *dependency injection* (DI) *pattern* makes sure modules receives their dependencies, without needing to know how to create the objects that they depend on. This removes hard-coded dependencies and leads to a lower degree of coupling in code. DI is accomplished with creating high-level abstractions of an object, as an interface or a base class. The dependency must be registered in a service container, which is usually provided by the framework. Finally, one can inject the service into the class constructor where the object is going to be used. DI helps developers to uphold two of SOLID's principles. The Dependency Inversion Principle, which is about classes being dependent on an abstraction, and the Single Responsibility Principle, which ensures that the object is only responsible for itself. [43], [44]

3 Experiment

This chapter will go into details of how the application was planned, developed, and deployed. First, the business case specification is presented, where the planification process took place. Next, the design and architecture are described together with the justification of choices that were made. Lastly, the implementation of application is presented.

3.1 Business Case Specification

Bouvet's problem in short; keeping track of its assets, was the starting point of creating a business case specification. From this basis, the steps that were made to undertake this were first preparatory work to aid problem specification and then the two first phases from Agile Scrum, identified in section 2.2.2.

3.1.1 Preparatory Work

To prepare for the first phase of Agile Scrum, specifically to identify the problem and the requirements of a solution, I did research on topics like *asset management* and *inventory tracking system*. The reason of doing this was to get an idea of what *keeping track of assets* could entail. I did this to be better prepared for the conversation with the client and to create a questionnaire to serve as a helper for gaining knowledge on the client's essential needs.

I focused on two different asset management systems. The first was Asset Panda[11] and the second, an open-source application called Snipe-It[12]. These two systems gave me input and knowledge on the concept of asset management. I identified relevant keywords from this knowledge, that I could use to continue my work. The keywords were *dashboard*, *tagging*, *check-in*, *check-out*, *categorization*, and *access control*. Based on these keywords I created a questionnaire with a proposed list of features and explanations of them, that could be a part of a solution. The questionnaire was used to start the first conversation with the client.

[11] https://www.assetpanda.com/
[12] https://snipeitapp.com/

3.1.2 The Planning Phases

Based on the questionnaire, I conducted the first meeting with the client. My role at this point were PO. My goal was to identify the problem and requirements. The client elaborated on the challenges of asset management, thus identifying the features needed from the solution. The key requirements were:

- An automized and streamlined registration of new assets.
- An automized and streamlined lending process.
- A dashboard to manage it all.
- Bonus: An integration towards the external system notifying when assets is connected to an employee

As mentioned before, the currently existing solution is to manually add new assets to an Excel spreadsheet, without any other software to automatically track the assets. Based on these requirements I took the role as SM and identified the features needed to meet these requirements and after this was done, the second phase of Agile Scrum took place. I wrote out the user stories, I created sequence diagrams to visualize the features and user stories, and based on these diagrams the backlog on GitHub Projects was filled with tasks. Fig. 6 displays an example of a sequence diagram.

To summarize the scrum process at this stage, I have identified a list of requirements and consequent backlog. Due to the extension of the backlog, I planned to implement a subset that could work as a functional pilot to demo the concept as part of my academical work. However, this remaining backlog will be useful for Bouvet in the future at a project handover.

3.1.3 Features and User stories

In this section I am going to present the features recognized from the requirements mentioned in the previous section. I have decomposed each feature into one or several user stories. In the "/docs" directory at GitHub[13], there are sequence diagrams linked to the features and user stories. A user in this scenario is an administrator and user of the asset management solution I am implementing.

3.1.3.1 Feature: Add new asset

- *As a user, I want to do a barcode scan of the serial number of an asset, identify the serial number, then generate a QR code, print out a label, which I can stick on the asset, to make it easier to manage that asset.*
- *As a user, I want to add assets without serial number, with generating a unique id and QR-code and stick to asset, to make it easier to manage that asset.*

3.1.3.2 Feature: View all assets in the system

- *As a user, I would like to view all assets in a dashboard, to get an overview of my assets.*
- *As a user, I want to see all assets based on category, to easier identify assets in dashboard.*

3.1.3.3 Feature: View a single asset in the system

- *As a user, I would like to see a single asset with all its information, to get more knowledge about the asset.*
- *As a user, I would like to see who has asset if it is being lent out.*

3.1.3.4 Feature: Scan asset and read information

- *As a user, I want to scan the asset's QR-code and view its information, to have easy access to information.*

3.1.3.5 Feature: Update and add information to one or more assets at the same time

- *As a user, I want to change and add information to an asset.*
- *As a user, I want to change status on one or multiple assets at once.*
- *As a user, I want to scan the resource and do simple updates on asset.*

[13] https://github.com/livun/bouvet-asset-hub/tree/dev/docs

3.1.3.6 Feature: Delete a "wrongly added" asset

- *As a user, I want to delete assets created by mistake, to make sure assets table is not populated with unnecessary items.*

3.1.3.7 Feature: Deprecate an asset

- *As a user, I want to change the status of the asset to be unavailable/not usable, but still have the asset in the system. I.e., the status should be changed from something to "discontinued".*

3.1.3.8 Feature: Scan asset and lend out to employee

- *As a user, I want to scan the asset, choose new loan and a time span, and lend out to employee.*
- *As a user, I want to scan the asset, choose "no time limit" and lend out to employee.*
- *As a user, when an asset is lent out, I want the system to send a notification to external system to mark the expense on employee, to reduce manual work.*
- *As a user I want to extend time of loan, to meet employees needs if requested.*

3.1.3.9 Feature: Scan asset and extend the loan

- *As a user, I want to scan the asset and extend the loan linked to that asset.*

3.1.3.10 Feature: Scan asset and hand in loan/asset

- *As a user, I want to scan the asset and remove loan from loans table, so that loans table only have "active loans".*

3.1.3.11 Feature: Add new loan

- *As a user, I want to add a new loan manually from the dashboard.*

3.1.3.12 Feature: Add loan to history

- *As a system, I want the "finished" loan to be removed from "Loans Table" and added to "LoanHistory Table".*

3.1.3.13 Feature: View all loans

- *As a user, I want to view all loans in a table, to get an overview of loans in system.*

3.1.3.14 Feature: View one loan

- *As a user, I want to view a single loan from table, to see information connected to that loan.*

- *As a user, I want to scan an asset and see the active loan connected to asset, to get easy access on information.*

3.1.3.15 Feature: Hand in loan

- *As a user, I want to hand in loan from the dashboard.*
- *As a user, I want to hand in loan from a single loan view.*

3.1.3.16 Feature: Extend loan

- *As a user, I want to extend loan from the dashboard.*
- *As a user, I want to extend in loan from a single loan view.*

3.1.3.17 Feature: View all loans linked to employee

- *As a user, I want to look up employee numbers and get a list of loans connected to this employee, to get an overview and easier manage employees' assets.*

3.1.3.18 Feature: View loan history

- *As a user, I want to see previous loans in a separate table, so that I can removed them from loans table, but I still could track history.*

3.1.3.19 Feature: Add new category

- *As a user, I want to add a new category from the dashboard.*

3.1.3.20 Feature: Send notification

- *As a user, I want there to be an automatic email sent out to the employee when it's time to return the asset.*

3.1.3.21 Feature: Access Control

- *As a system, I want the users to be authenticated and authorized to access data and perform actions.*

3.1.3.22 Feature: Notifying Xledger

- *As a system, I want there to be sent of an API call to the external system, Xledger when an employee is connected to a new asset, so that Xledger can automatically do cost accounting.*

3.2 Design, Architecture, and Infrastructure

Before I started coding and implementing the features and user stories I identified during the business case specification, I had to specify a plan for the system architecture. I will now present this system architecture specification. Furthermore, I will present the justification for the decisions that I made, while also assessing some alternatives.

3.2.1 Architecture

My solution follows a *three-tier architecture*. The first tier is a *presentation tier*, meaning the user interface, where an end-user is in contact with the solution. The second is an *application tier*, this is the centre of the solution, where the data from the other tiers gets processed. The third tier is a *data tier*, also called the database tier, here the data involved in the application gets managed and stored. The application tier is responsible for the communication, the presentation tier cannot communicate with the data tier and the other way around. [45]

Fig. 1 displays the proposed architecture diagram. The presentation tier, also called the frontend, and the application tier, also called the backend, will be deployed in Docker Containers. I plan to use Bouvet's own SQL Server to store data. I will register the frontend- and backend application in Bouvet's Azure AD resource. Moreover, I will connect to Xledger's API to send HTTP requests.

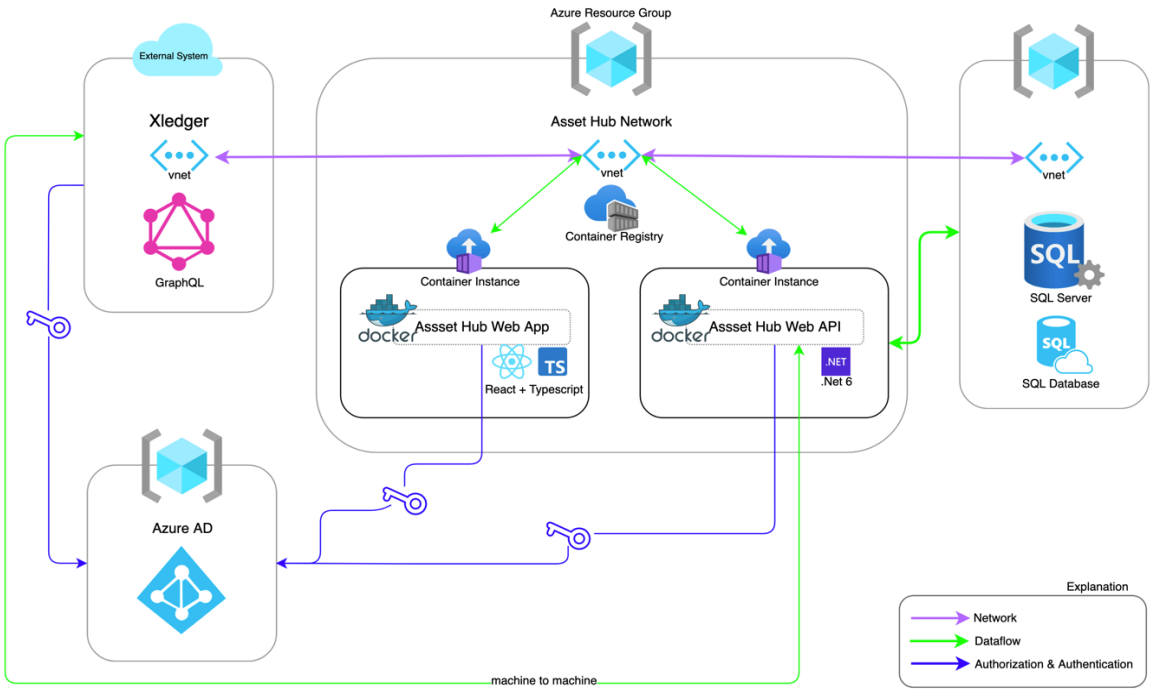

Fig. 1. Architecture diagram.

My presentation tier is a React SPA, the application tier is a ASP.NET Core 6 Web API, and the data tier, the MSSQL relational database. The three-tier architecture in software applications is a well-established architecture that assumes independence of the others in terms of implementation and infrastructure. This is beneficial because a tier can change its implementation or infrastructure without the need to revise the other two. Also, it can be developed simultaneously and scaled without impacting the other tiers. [45]

There are many system architecture alternatives. One example is a two-tier architecture, also known as a client-server architecture. In this example there is no application tier, only a client tier and server tier. Recall these are also known as the user interface, and database, respectively. The advantages of this alternative architecture are that is easier to develop and deploy solutions. Whereas, one of the disadvantages of not having an application tier is that it could be more difficult to implement business logic and security. [46]

I decided to use the three-tier architecture. This is because the amount of logic the solution requires, justifies the advantages of a three-tier architecture. The application tier will provide a more secure solution, as the client will not have direct access to the database. Moreover, regarding security, I can validate incoming data and add controlled access to my API endpoints. The three tier-architecture is also well known, easy to understand and familiar to the company. This allows for an easier posterior integration and modifications by Bouvet. It makes a good candidate for developing a small application, because the time to have a simple application up and running is short.

3.2.2 Presentation Tier

HTML, CSS, and JavaScript are the building blocks for web development. HTML is for providing structure, CSS is for styling, and JavaScript is for enabling interactive webpages. While it is an option to develop in plain JavaScript, there exist a variety frontend framework and libraries that are designed to improve the developer experience. A framework is a library that gives predefined ways of how to build the software. It makes it easier and faster to develop dynamic and interactive UIs. The code is therefore more scalable and maintainable. [47]

3.2.2.1 Why React, TypeScript and MUI?

As mentioned, React ranks very high on the list of popular web libraries and enjoys a large community of supporting developers. React is well documented, and because of its popularity, there are a vast number of third-party libraries that can be used with React, like MUI and TanStack Query etc. React has a JavaScript syntax extension called JSX (TSX when using TypeScript), that allows developers to write HTML directly in React [48]. JSX is one of the reasons that I chose to use React. JSX components makes it easy to create a dynamic UIs, because I can have logic, HTML and CSS all in the same ".tsx" file. Based on the use case, I can make the components as dynamic, or as static as I want. A dynamic component is reusable, whereas a static component is created to have only one job. The dynamic components can easily be reused in several scenarios, which reduces duplicate code.

Another benefit of using React is the MUI library introduced in section 2.1.1.2. MUI provides components that are ready and styled "out-of-the-box". Examples of such components are grids, buttons, forms etc. This makes it easier to create dynamic layouts, as I do not need to use time customizing CSS and HTML. Moreover, the components provide complex functionality. Therefore, as a developer, I can focus on the logic rather than making sure the webpage looks presentable and user friendly.

There are also other frameworks, for instance Angular, which have many of the same functionalities as React. When comparing React and Angular, React is mostly used for building UI-components that display varying data. Angular, on the other hand, is used to build large complex and feature rich enterprise-grade applications [49]. I decided to use React because of the benefits listed above. Moreover, the fact that another framework like Angular, does not provide significant advantages, as they have similar functionalities. In my solution, React would provide me with the necessary tools to build a dynamic UI fast. Angular

strengths were redundant in this matter, as my solution is neither complex nor especially large. At Bouvet, there is a high level of expertise in React. Therefore, using React would also facilities an easier project handover.

I chose to use React with TypeScript (TS), as it adds type-safety. The TS compiler identifies errors at compile time instead of runtime, making it easier for the developer to debug and find the source of problems. Moreover, as TS is strongly and explicitly typed, it is easier for the developer to read and follow the code. Using TS, I was encouraged to avoid all ambiguous types, i.e., *any*-types. When working with an API, it is common to create a so-called *contract* which specifies request and/or response types. Initially the object received from the API is JSON formatted, but with TS I am forced to use the specified types, given by the contract. As explained, it is easier to develop and review the code which has specified types. I used a generator to get these types to match the backend types, I will elaborate on this later in section 3.3.3.4.

3.2.2.2 Why Axios and TanStack Query?

I tried to use JavaScript built-in Fetch API to create generic HTTP requests. When using it, I had to write a lot of boilerplate code to create one generic request. As I am working with an API with many endpoints, I needed an API client implementation that did not require me to write a lot of code on each HTTP request. Therefore, I found Axios, which have the same purpose as Fetch API, but comes with some extra functionality. It comes with automatic JSON conversion and method aliases, making it is easy to create clean and concise HTTP requests. Axios can be used alone, but combined with TanStack Query, it provides many benefits, therefore I chose to use the combination.

TanStack Query (TSQ) caches server data and stores it to a query-key. Therefore, if in the future the same query is made, the cached data can be used instead of fetching it from the server. At some point the cache is deleted. This reduces the time it takes to access data and the amount of requests to the server, hence reducing the network load. TSQ Query makes mutating data using POST, PUT, and DELETE easy, as it provides call-back functions to handle every stage of the mutation cycle. Using TSQ with Axios, makes it possible to write an Axios request in a few lines of code as opposed to the fifteen lines I needed to use with the Fetch API. A large reason for the reduced complexity is the inbuilt error handling TSQ provides. TSQ also comes with built-in booleans like *isLoading*, *isSuccess* and *isError* making it easy to dynamically render the web page.

3.2.3 Application Tier

When choosing a framework for developing the application tier, I first and foremost had to decide what kind of programming language to use. A compiled language or an interpreted language. A compiled language is compiled into machine code before it is executed, whereas an interpreted language is executed line by line without any compilations. As a result, a compiled language is faster than an interpreted language. When developing, errors will prevent a compiled language to compile, whereas debugging will not happen before runtime for an interpreted language. I decided to use a compiled language, because it is faster, and it improves the developing experience. I will get help from the compiler while debugging, thus making it easier to remove logical and semantic errors. [50], [51]

I chose to use C#, which is Microsofts own language. C# is a high-level language, meaning that it is easy to read. As with TypeScript, C# is type-safe and thus come with the same benefits as TypeScript. An alternative to C# would be Java. Java is developed by Sun Microsystems and owned by Oracle. As programming languages, C# and Java are similar, they are both static- and strongly typed object-oriented languages. The decision to use C# came down to what is the preferred platform for the department I worked for at Bouvet. Several of Bouvet's clients request the use of the Microsoft platform. Because C# and the Microsoft platform is widely used in that department, it was the natural decision to build on Microsoft's .Net Framework. Consequently, I used the ASP.NET Core 6 Web API to develop an API.

ASP.NET Core 6 Web API provides simple and light-weight services and comes with several advantages. The endpoints come with automatic serialization of classes to JSON, the policy-based authorization makes it easy to provide access control rules on endpoints and routes can be defined inline [17]. The Web API comes with built-in support for dependency injection, which I presented in section 2.3.4. The framework is open-source, allowing growth in online community, which can provide support and guidance.

3.2.4 Data Tier

During the planning process I identified the data models displayed in the entity-relationship (ER) diagram in Fig. 2. I had to find a way to store the asset entities and track these asset entities while employees were lending them. An asset could be lent to many employees, but only to one at the time. However, an employee could be borrowing many assets at the same time. Therefore, I identified the need for a loan entity which connected this many-to-many relationship. Utilizing an ER-diagram helped me with the process of normalizing the data and identifying which attributes that should be stored together. An ER-diagram can be directly translated into relations [24].

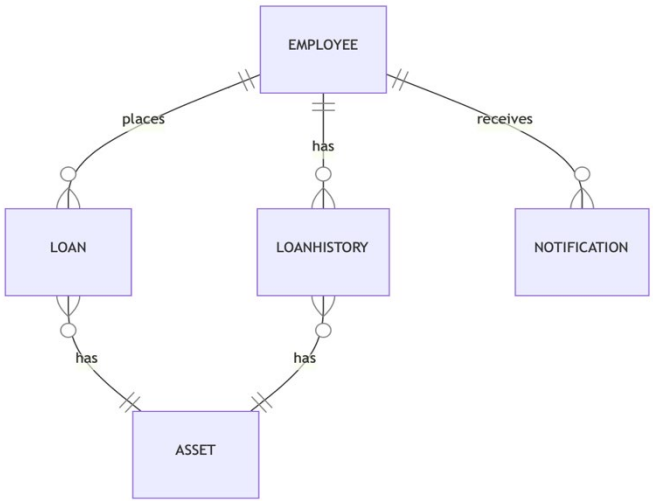

Fig. 2. Entity Relationship diagram.

The typical use case for the relational data model is when; the data is structured, there is anticipated minimal changes to the model, the data is simple and transactional, and the relationships are well defined [52], [53]. Looking at the diagram, my models are simple, static, and highly based on relationships. A loan has a transactional nature, meaning that there are several subprocesses that needs to succeed for the transaction to succeed [54]. In this case for example, creating a loan entity in the database, entail checking availability of asset entity and changing its status, before proceeding to create the loan entity.

While identifying the models, I contemplated the possibility of using a different model than the relational model. For example, the document model. The document model reflects the object format used in application code, and a document database stores data in documents, for example JSON. This allows for dynamic and evolving data without the need to define a model in advance but does not support many-to-many relationships between data types. However, I decided that the relational model was best suited for my solution. [53], [55]

3.2.5 Security

There are two matters to consider when implementing security to my solution. First, the fact that an admin will have to sign-in to the application in order to view information and perform actions. The second matter is handling employees and employee information in regards of lending assets. It would be beneficial for the application to avoid storing sensitive user data, as comes with extra concerns and responsibility.

Bouvet uses Azure AD in their intranet solution, thus has already an existing directory over all employees in the company. Therefore, it would be constructive to use this already existing resource in the company. With Azure AD, I can outsource the sign-in experience and add conditional access policies in connection with the endpoints in my API. I am also able to store minimal information in my solution regarding identifying employees that lends assets. This is useful because storing sensitive person information requires me to consider the *General Data Protection Regulation* (GDPR). The only data my solution needs to store is a key from Azure AD that connects the employee entity to the employee in Bouvet's directory. Hence, my solution does not need to store any personal data.

3.2.6 Deployment

To make the application available to use, I had to choose a deployment method. Among the alternatives I considered, I chose to use containerization, which is often compared to virtualization. Whereas containers take on the host capabilities, a Virtual Machine (VM) includes a complete OS. This is visualised in Fig. 3.

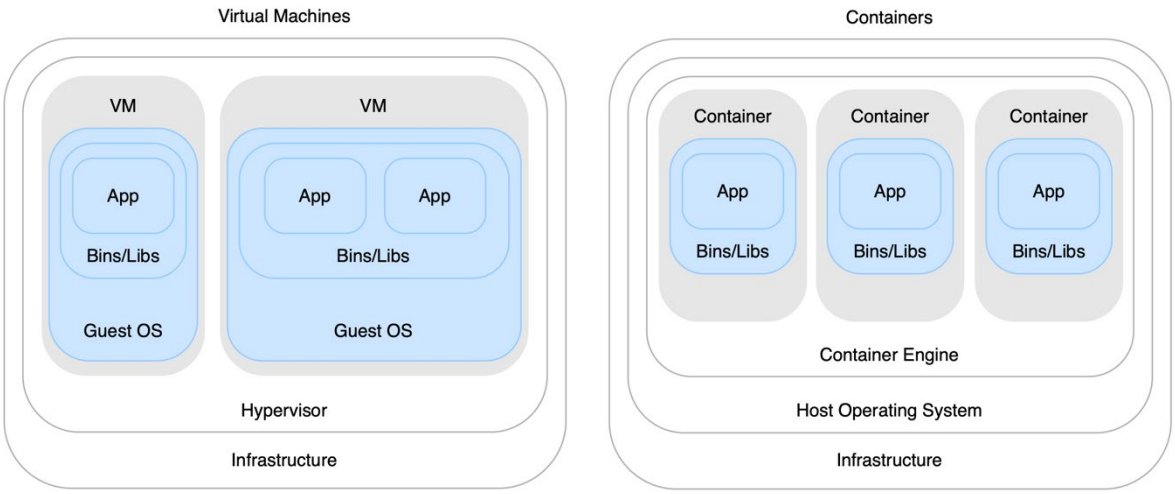

Fig. 3. Virtual Machines vs. Containers. Adapted from [56].

One of the main reasons for choosing containers is its portability. As I mentioned in the section 2.1.5, the container includes all that is needed to run the application. This allows it to run on Linux, Windows, and Mac OS's, on virtual machines, on physical servers, on the cloud etc. The containers are relatively lightweight as compared to virtual machines, in terms of memory utilization and CPU utilization. The reason a container is lightweight is due to their separation from the underlying infrastructure and OS. This makes it possible for multiple containers to run on the same host, but isolated from each other. This isolation prevents other containers to be affected if one container crashes, or application within them fail. However, a container is not isolated from the host OS, so vulnerabilities with the host can impact all containers running on that host. On the other hand, a VM is a standalone system, which makes it resistant to vulnerabilities of a shared host. Furthermore, a container's *lightweightness* provides fast app start up and shut down. This gives the opportunity to quickly release new applications and upgrades, thus speeding up the development process. A VM takes much more time to spin up and it requires more resources. [57]–[60]

A server can run many more containers than VMs, but a VM can also run containers, meaning that they are not mutually exclusive [61]. Choosing deployment methods depends on

the use case. If there are specific hardware requirements or the need for isolating system resources and entire working environments, a VM is needed. Containerization is good for isolating applications, it is easy to manage and fast, and it is therefore a good candidate for developing web application services. [57], [62]

Based on the comparison of containers and VMs, I chose to use containerization as the deployment method for my solution. This allowed me to deploy the frontend in one container and the backend in another. Thus, facilitating an easier handover due to its portability.

Regarding why I chose this technology stack, I would like to emphasise the importance of maintainability in code. My department at Bouvet holds sufficient knowledge and resources to continue this project beyond the completion of my thesis. The usage of a familiar architecture, together with the necessary documentation that I created, and implementation of these well-known technologies, are all crucial to maintainability and longevity of this project.

3.3 Implementation

At this point, the backlog of tasks was filled and the architecture of my solution was planned. Therefore, I could begin the third and iterative phase of Agile Scrum, consequently, start coding and testing. I did not follow the steps of the iteration to the point, hence I did not plan specific sprints or performed sprint retrospectives. I had to find an approach to the agile iteration cycle that was practical in my situation, because I was developing alone, and the timespan of the project was relatively short. First, I decided which features I was going to implement for the pilot, then I planned the course of action. Fig. 4 displays the Kanban board filled with tasks and features. The first column of the board is the backlog. This is captured in the middle of the project.

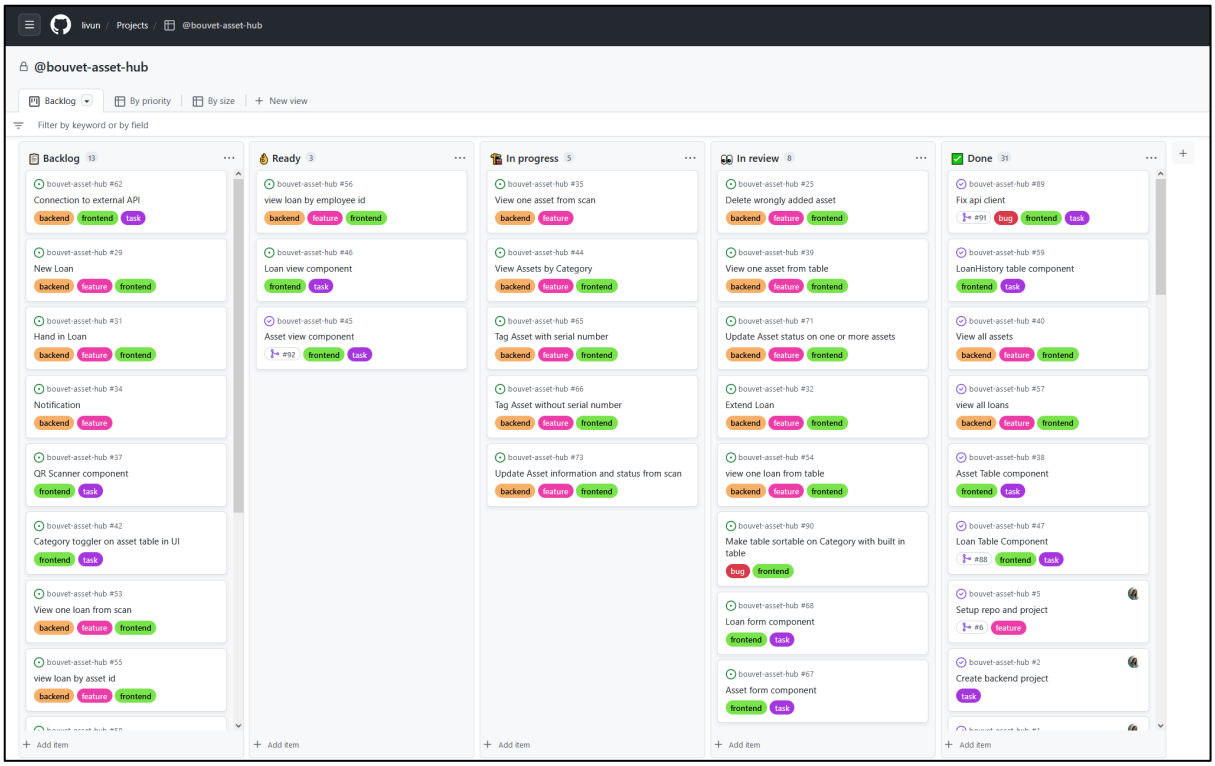

Fig. 4. Kanban board from GitHub projects.

I decided that to pilot the project, I had to focus on the fundamental role of the application. Such as the so-called CRUD (create, read, update, and delete) operations. This would create the foundation for the implementation of these features:

- Feature: Add new asset
  Feature: View all assets in the system
- Feature: View a single asset in the system
- Feature: Scan asset and read information

- Feature: Update and add information to one or more assets at the same time
- Feature: Delete a "wrongly added" asset
- Feature: Deprecate an asset
- Feature: Scan asset and lend out to employee
- Feature: Scan asset and hand in loan/asset
- Feature: Add loan to history
- Feature: View all loans
- Feature: View one loan
- Feature: View all loans linked to employee
- Feature: View loan history

To achieve this, I needed to implement a backend that could handle this functionality, a database to store all the necessary data, and a frontend that could serve as a UI and make it available for the end-user. Hence, I had to get an ASP.NET Core Web API, MSSQL Server, and a React App up and running. With this in place, I could first implement all the functionality to perform CRUD operations in the backend. I used the sequences I created during the planning phase as templates for the implementation. This is described in section 3.3.2. When the backend was tested and worked, I created the UI with a dashboard, where the user could perform the actions specified in the features. This is described in section 3.3.3.

I decided that features that implemented scanning was secondary to the other features, because they could not be implemented until the other features were done. Hence, I decided to add the QR- and barcode scanning and printing of labels towards the end of the project. Furthermore I decided that “Feature: Send notification”, “Feature: Access Control”, and “Feature: Notifying Xledger” was not a priority and necessary for the pilot. I will explain why I made this decision in section 4.2.1.4.

I will now provide a short summary of the features I have implemented. Then I will go through the development of the backend, frontend, and database, while presenting code snippets, functionality, and considerations. All code snippets, tables and figures are retrieved from the provided source code and documentation in GitHub. Thus, my own work if nothing else is stated.

3.3.1 Summary of the Features Implemented

With the development of the frontend, backend, and database storage, I have created a pilot that implements the features mentioned in section 3.3. For an end-user it is now possible to get an overview of every asset, active loan and previous loans stored to the system. There can be added new assets and loan, in a manual and automatic way. Moreover, with a QR-scanner, one can easily identify an asset and its status, and do simple actions. When a loan is handed in, its removed from loans and added to the loan history.

The frontend is responsible for the user experience, making every functionality available to the user. The backend handles the requests from the frontend and act on these requests. This is primarily separated into two main jobs. Either to retrieved data from the database or mutate data in the database. While describing the implementation of the backend in the next section, I am also elaborating on the decisions I made to create a maintainable solution.

3.3.2 Backend Development

In this section I will go through the implementation of the backend. As mentioned in the introduction of this subchapter, the backends role was to perform CRUD operation. The features with the keyword *add*, is connected to *Create* in CRUD. The features with the keyword *view*, is connected to *Read* in CRUD and so forth. Hence, I needed to implement functionality in the backend that had an API, so that the frontend could communicate with- and perform HTTP requests to the backend. In needed functionality that performed the task of retrieving or modifying the requested data. Finally, I needed functionality that stored this data in a database. I will now present the implementation I developed to perform these tasks.

I will first present how I structured that backend and why. Secondly, I will discuss cohesion and coupling, while I give an overview of backend. Thirdly, I will go deeper into the implementation of the backend code, while demonstrating what I did. Finally, I will briefly mention the testing of the backend.

3.3.2.1 Backend Structure

When implementing the backend, I strived to follow relevant design principles, to create a backend with high cohesion and low coupling. As mentioned in section 2.3.1, one should aim to have high cohesion and low coupling to reduce complexity in code and make it more maintainable. In this section I will present what I did to achieve this.

Initially, I contemplated how to structure the backend. In this matter, I had to consider the complexity of the backends role. An alternative was to develop the backend based on the principles of *Domain Driven Design* (DDD). DDD aims to focus on centring the code around a business domain where the software will be used. A domain consists of several subdomains, dividing different parts of business logic. DDD enables high cohesion and low coupling, and is used to solve complexity in the sense of many different data sources and many different business goals [63]. The role of my backend was more or less to query and mutate data. There were no complex business logic and therefore DDD seemed like a redundant choice. Instead, I chose to implement a simpler layered structure, while focusing on achieving high cohesion and low coupling. This is shown in Fig. 5, where the arrows shows the dependencies.

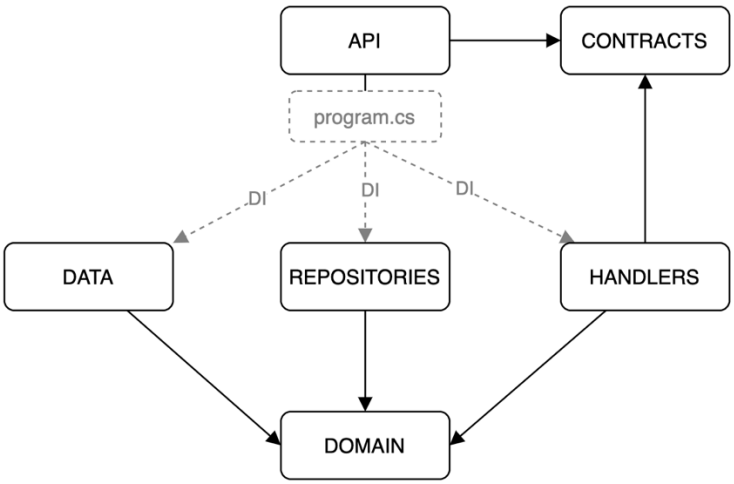

Fig. 5. Backend Assemblies.

3.3.2.2 Low Coupling and High Cohesion

The backend consists of six separate assemblies. An *assembly* is a collection of resources and way to separate code, thus making it easier to work with and reuse [64]. In my solution, each assembly holds parts of code with similar modules. I did this, as an attempt to develop a loosely coupled backend.

The *Contracts* and *Domain* assemblies contain data classes, meaning that they do not hold any behaviour, hence only state. *Contracts* contains the data transfer objects (DTOs), which is an object meant for transferring data between processes, like over the network. *Contracts* also contains query and command objects. Queries asks for data, whereas commands mutate it. These objects are the way the API communicate through the handlers with essentially the database. More on this in section 3.3.2.5. Domain contains the entities, the data models of the relations that is stored in the database. The two class libraries do not have any dependencies, thus promoting low coupling. This is visualised in Fig. 5, where there are no arrows from *Contracts* and *Domain*, only towards.

The *Handlers* assembly contains a collection of all mediator handlers. As introduced earlier, the mediator handlers work as message brokers. They receive a query or a command, processes it and returns what were requested. The mediator pattern is implemented to separate the concerns of the different parts in code, thus preventing tight coupling. As illustrated in Fig. 5, the API is not aware of the Domain and how the data is processed. The API only communicates with rest of the program through the handlers.

The *Data* assembly contains the implementation of the ORM, EF Core, and instantiate the *DbContext*. The *DbContext* class is responsible for the interaction with the database. The *Repositories* assembly contains a collection of repository classes, which contains methods to access the database. Each repository class contains methods to get or mutate a specific entity, thus making these classes highly cohesive because the methods all relate to the same entity object. The API assembly contains controllers which handles incoming HTTP request. As with the repository classes, the controller classes are based on the entity they are handling, thus making these classes cohesive as well.

I use *dependency injection* (DI) to register all the services in the *Handlers*-, *Data*-, and *Repositories* assemblies, thus making sure that the are no hard-coded dependencies when the services are being used.

To summarize this section, I have achieved a low degree of coupling with using DI pattern and the *mediator* pattern. Furthermore, I have enabled high cohesion in the modules where there is more logic, the controller classes, and the repositories classes. Proceeding, I will give an outline on how the assemblies are connected, and afterwards go into more details on each assembly and the modules they hold.

3.3.2.3 Connection Between Assemblies

I will now describe how the assemblies are connected, with use of an example. This is all visualised in Fig. 6.

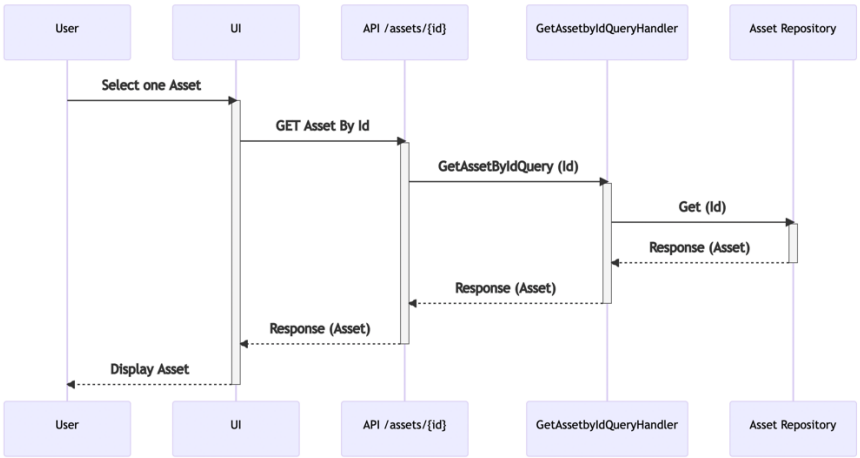

Fig. 6. Example of a sequence diagram.

When the API receives an incoming HTTP request from the client, it processes the route and invokes the correct method. Let us say the route is “api/assets/1”. This route is mapped to the *GetAssetById* method. The controller method sends of a mediator request, in this case the

GetAssetByIdQuery. The *GetAssetByIdQueryHandler* receives the request and processes it. The *AssetRepository* is dependency injected into the handler's constructor, thus the handler can call the relevant *Get* method to query for asset with id 1, in this example. The *DbContext* is dependency injected into the repository's constructor. The repository can therefore retrieve the data from the database. Asset with id 1 is returned, first to the handler, then from the handler to the controller, and finally the client gets its response.

I have now briefly gone through what is happening in the backend. Proceeding, I am going to elaborate on each part of code.

3.3.2.4 API

The API controllers holds all the endpoints defined in the API specification in Table I.

Table I
The implemented REST API

Resource	GET	POST	PUT	DELETE
/assets	Retrieve all assets	Create new asset	Update assets from list of id's	-
/assets/1	Retrieve asset by Id	Error	Update asset by Id	Delete asset by Id
/assets/6B29FC40...	Retrieve asset by Guid	-	-	-
/assets/1/loans	Retrieve all loans for asset by Id	-	-	-
/loans	Retrive all loans	Create new loan	-	-
/loans/1	Retrieve loan by Id	-	Update loan by Id	Delete loan by Id
/employees/1/loans	Retrieve all loans for employee 1	-	-	-
/loanhistory	Retrive all loanhistory	-	-	-
/categories	Retrive all categories	Create new category	-	-
/categories/1	Retrive category by Id	-	Update category by Id	Delete category by Id
/categories/1/assets	Retrieve assets by category	-	-	-

A controller class handles the incoming HTTP request and returns a response to the request. I have implemented my controllers according to the REST design principles, which focus on resources. A *resource* is an object, data or service that can be access by the client [17]. The resource is identified by the *Uniform Resource Identifier* (URI). This is exemplified in Table I. The table describes all the implemented HTTP requests. I will demonstrate an example of a controller class and a method that handles a HTTP GET request in Code 1.

```
1 [ApiController]
2 [Route("api/[controller]")]
3 public class AssetsController : ControllerBase
4 {
5     // Code removed for brevity
6
7     // GET /assets
8     [HttpGet]
9     public async Task<ActionResult<List<AssetResponseDto>>> GetAssetsAsync()
10    {
11    var result = await _mediator.Send(new GetAssetsQuery());
12    return new ActionResultHelper<List<AssetResponseDto>>()
13        .OkOrNotFound(result, "Currently no assets in table!");
14    }
15
16    // Code removed for brevity
17 }
```

Code 1. Example of a controller method (AssetsController.cs).

The route name is defined by the class name, as we can see in line 2 and 3 in the example, the route becomes “api/assets”. In line 8, I define that I am going to handle a HTTP GET request. This request returns an *ActionResult* that contains the HTTP Status code and response object, which in this case is a list of *AssetResponseDto*. In Code 2, I am displaying how I can specify the route when querying based on a parameter. In this example route becomes “api/assets/{id}”.

```
1 [Route("{id}")]
2 [HttpGet]
3 public async Task<ActionResult<AssetResponseDto>> GetAssetbyIdAsync(int id)
4 {
5     var result = await _mediator.Send(new GetAssetByIdQuery(id));
6     return new ActionResultHelper<AssetResponseDto>()
7       .OkOrNotFound(result, "Asset does not exist!");
8 }
```

Code 2. Example of a controller method whith input parameters (AssetsController.cs)

I have created five API controller classes based on the five different resources one can access specified in Table I. The methods in the controller classes are approximately the same. They dispatch a mediator query or command, which is received by corresponding mediator handler. The controller always returns an *ActionResult* which contains a DTO. As mentioned in section 3.2.3, the ASP.NET Web API automatically serialize and deserialize the incoming and outgoing data. Therefore, I only need to define what I want to receive in the input field of my controller method, and what I am returning from the controller method.

When returning an *ActionResult* (line 6-7 in Code 2), I have created a helper method that is used to evaluate which response is returned to the client. Whether to send an *OK* result, with status code 200, or and *NotFound*/*BadRequest* with corresponding status codes 404 or 400. This is decided by functionality of the *Option* type that I am going to present in section 3.3.2.8. The helper methods are displayed in Code 3.

```
1 public class ActionResultHelper<T> : ControllerBase
2 {
3   public ActionResult<T> OkOrNotFound(Option<T> result, string msg)
4   {
5       return result.IsSome ? Ok(result.FirstOrDefault()) : NotFound(msg);
6   }
7   public ActionResult<T> OkOrBadRequest(Option<T> result, string msg)
8   {
9       return result.IsSome ? Ok(result.FirstOrDefault()) : BadRequest(msg);
10  }
11 }
```

Code 3. Helper method (ActionResultHelper.cs).

I extracted this helper method because I found myself repeating the lines 5 and 9, of Code 3, on each method in each controller. Therefore, to avoid duplicate code and make it more readable, I created a method for it. The downside of extracting these methods is that I have created a hard-coded dependency on the *ActionResultHelper* class. Each time I am using the method, I must instantiate it with the keyword *new*, which creates high coupling in my code. This is a good example of a situation where I can use the dependency injection pattern to decrease coupling. To improve this module, I have added this enhancement as a task to the backlog.

The API assembly also contains the “Program.cs” file, which is the entry point of the application and where the application runs from. All services that are used are registered here, among them are the ones that are used with dependency injection, the *DbContext*, the MediatR Service etc.

In “Program.cs” I also had to register the *Cross-Origin Resource Sharing* (CORS) policy. CORS authorizes HTTP request from other origins, than which the application is served from. Usually the same-origin policy, which is a part of browser security, prevent request from different origins. Adding a CORS policy enables us to control explicitly which cross-origin request that are allowed. [65]

This had to be done, because the client, i.e., the frontend, is served from another port than the backend. In Code 4, from line 1 to 10, I create the policy. I have specified which origin, in my case, which port that is allowed to make request to the backend. Finally, in line 12, I used the built-in *UseCors* method to add the policy to the application.

```
1  var MyCorsPolicy = "_myCorsPolicy";
2  builder.Services.AddCors(options =>
3  {
4      options.AddPolicy(name: MyCorsPolicy, policy =>
5      {
6          policy.WithOrigins("http://localhost:3000")
7              .WithHeaders("*")
8              .WithMethods("PUT", "DELETE", "GET", "POST");
9      });
10 });
11 // Code removed for brevity
12 app.UseCors(MyCorsPolicy);
```

Code 4. CORS policy (Program.cs).

3.3.2.5 Contracts

The DTO's of the *Contracts* assembly are data that are being sent to- or retrieved from the frontend. These objects work as contracts with the client, defining which kind of data and how the data looks like. DTO's flattens nested entity objects, thus making them easier to serialize and use in the frontend. Creating DTO's helps with decoupling the API from the assemblies that deals with the database. When using DTO's I can also decide which properties of my entity classes I want to expose.

I have used the *record* type in C#. The record type's primarily function is to store data, they are also immutable, as opposed to classes [66]. This gives me the opportunity to define the DTO's in a compact way using the *positional syntax* to define the DTOs properties [67]. I have displayed this in Code 5.

```
1 public record CreateLoanDto(DateTime IntervalStart, DateTime? IntervalStop,
2                              bool IntervalIsLongterm, int AssignedToValue,
3                              int AssetId, string? BsdReference);
```

Code 5. Example of a DTO record (CreateAssetDto.cs).

The *Contracts* assembly also contains query and command objects that implements the MediatR's *IRequest* interface. As mentioned in section 2.3.4, a request message is dispatched to the handler. The request describes the query or commands behaviour and its return value. In Code 6, I demonstrate an example of a query and in Code 7, an example of a command.

```
1 public record GetAssetByIdQuery(int Id) : IRequest<Option<AssetResponseDto>>;
```

Code 6. Example of a Query (GetAssetByIdQuery).

```
1 public class CreateLoanCommand : IRequest<Option<LoanResponseDto>>
2 {
3     public DateTime IntervalStart { get; set; }
4     public DateTime? IntervalStop { get; set; }
5     public bool IntervalIsLongterm { get; set; }
6     public int AssignedToValue { get; set; }
7     public int AssetId { get; set; }
8     public string BsdReference { get; set; } = "";
9 }
```

Code 7. Example of a Command (CreateLoanCommand.cs).

The *IRequest* interface can be used with both classes and records. I decided to use record when there was none to three parameters. They could be easily instantiated with the positional syntax or be mapped. Code 2, line 5, shows an example of a request getting instantiated and dispatched from the controller. When DTO object received from the client had several parameters like the *CreateLoanDto* of Code 5, I used a mapper to instantiate the *CreateLoanCommand* of Code 7. This is exemplified in line 5 of Code 8. I will explain the use of mapper in section 3.3.2.8.

```
1 // POST /loans
2 [HttpPost]
3 public async Task<ActionResult<LoanResponseDto>> AddLoanAsync(CreateLoanDto dto)
4 {
5   var result = await _mediator.Send(_mapper.Map<CreateLoanDto, CreateLoanCommand>(dto));
6   return new ActionResultHelper<LoanResponseDto>().OkOrBadRequest(result, "Could not add loan!");
7 }
```

Code 8. Example of a POST controller method (LoansController.cs).

The controller method in Code 8 also demonstrated how the received DTO is defined in the input parameter in line 3.

3.3.2.6 Domain

In the *Domain* assembly, I have defined the entity models that my application is built on. These classes describe how the data is stored in the database. Fig. 7 displays the class diagram.

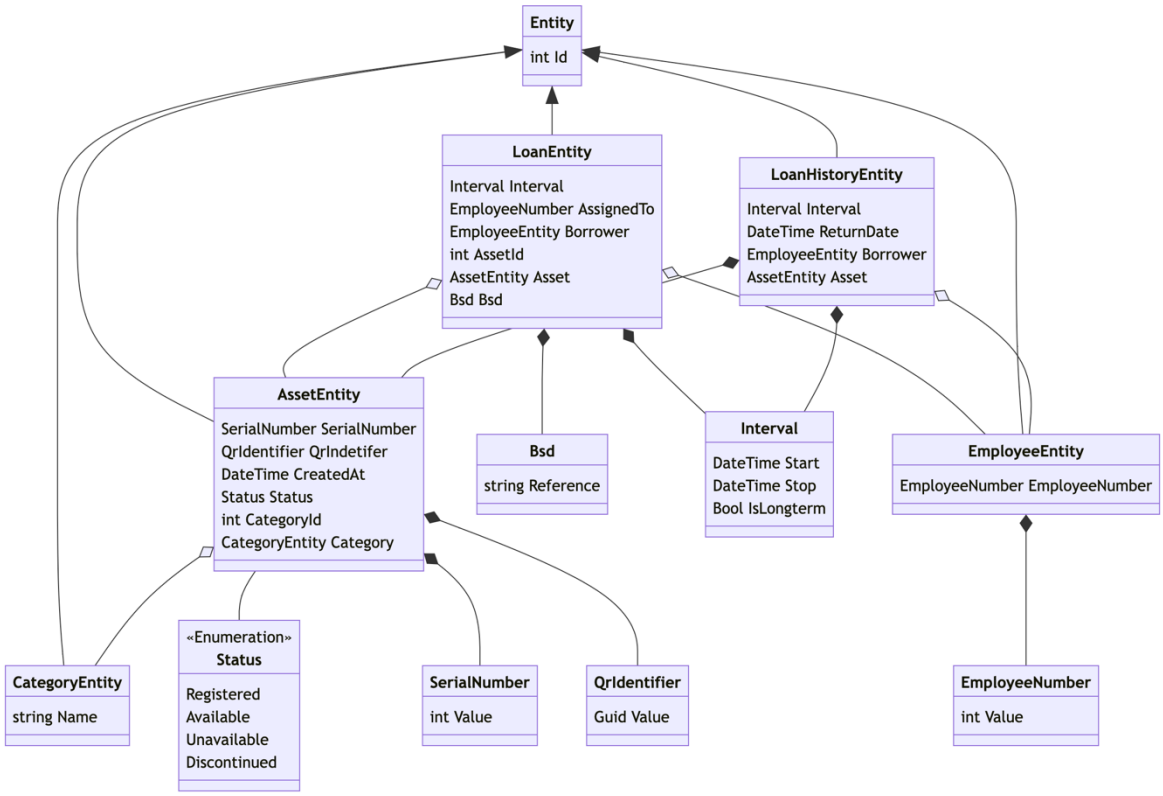

Fig. 7. Class diagram.

All entity classes inherit from the base class *Entity*, this is demonstrated with arrows. The classes with the white diamonds represent an aggregation relationship, meaning that the objects exist independently, but one object contains the other. The classes with black diamond represent a composition relationship, meaning that the life of the object depends on the other object, thus it is owned. [68]

The reason that I have used composition in my entity classes is to have the possibility to reuse classes or change them, especially in the developing stage. Composition allows me to deal with complex objects and make the code more readable. I can also validate each class independently, making sure I do not violate the single responsibility principle. This would have happened if I were to create a validator for the whole class and validated based on different criteria.

The *Notification* entity from Fig. 2 is not part of the initial data model. The notification feature based on this entity was not implemented in the pilot. I will elaborate on this in section 4.2.1.4.

3.3.2.7 Data

In have implemented a model with EF Core. A model consists of a context object and entity classes.

```
1  public class DataContext : DbContext
2  {
3      public DataContext(DbContextOptions<DataContext> options) : base(options)
4      {
5          Database.EnsureCreated();
6      }
7      public DbSet<AssetEntity> Assets => Set<AssetEntity>();
8      public DbSet<LoanEntity> Loans => Set<LoanEntity>();
9      // Code removed for brevity
10     protected override void OnModelCreating(ModelBuilder modelBuilder)
11     {
12         modelBuilder.Entity<AssetEntity>()
13             .Property(s => s.Status)
14             .HasConversion(
15                 v => v.ToString(),
16                 v => (Status)Enum.Parse(typeof(Status), v));
17         // Data seeding happens in this section (code removed for brevity).
18         base.OnModelCreating(modelBuilder);
19     }
20 }
```

Code 9. How the model is created (DataContext.cs).

The *DataContext* implements EF Core's *DbContext* class. The context represents a session with the database and is dependency injected where the database needs to be accessed. In Code 9, line 5, the database is created if it does not already exist. In line 7-8 I define the tables in the database based on the entity classes from the *Domain* assembly. Since I use enumeration types in the *AssetEntity* class, I need to configure them as strings and not integer. This happens from line 12-16. From line 17, I seed the model with data, this is removed from the example. These two last actions are not necessary, but convenient when developing and debugging the application.

To use EF Core with MSSQL Server, I have register and configure the *DataContext* model with the database provider and a connection string. This is displayed in Code 10.

```
1 var connectionString = builder.Configuration.GetConnectionString("DataContext");
2 builder.Services.AddDbContext<DataContext>(options =>
3 {
4     options.UseSqlServer(connectionString);
5 });
```

Code 10. Registration of DataContext (Program.cs).

For demonstration purposes of the pilot, I have stored the connection string in "appsettings.json". This is not how it is done in production, nor necessarily in development because it exposes the secret information. I have used Visual Studios Secrets Manager during development, thus not exposing any app secrets to the public. When moving into production I can use implementations like Docker Secrets or Azure Key Vault, which is tools to store sensitive data.

At this point the context is ready to be used in the application.

3.3.2.8 Handlers

When a controller has dispatched a mediator request message, it is a handler's job to receive and process the request. There must be one handler per command or query. The handlers are approximately similar. They all receive a repository dependency and a mapper dependency in the constructor, following the DI pattern. Code 11 displays an example of a handler.

```
1 public class CreateAssetCommandHandler : IRequestHandler<CreateAssetCommand, Option<AssetResponseDto>>
2 {
3     private readonly IAssetRepository _repository;
4     private readonly IMapper _mapper;
5     public CreateAssetCommandHandler(IAssetRepository repository, IMapper mapper)
6     {
7         _repository = repository;
8         _mapper = mapper;
9     }
10    public async Task<Option<AssetResponseDto>> Handle(CreateAssetCommand command)
11    {
12        var asset = await _repository.Get(AssetPredicates.BySerialNumber(command.SerialNumberValue));
13        if (asset.IsNone || command.SerialNumberValue == "")
14        {
15            var assetEntity = _mapper.Map<CreateAssetCommand, AssetEntity>(command);
16            return _mapper.Map<AssetEntity, AssetResponseDto>
17            (
18                (await _repository.Add(assetEntity)).First()
19            );
20        }
21        return Option<AssetResponseDto>.None;
22    }
23 }
```

Code 11. Example of a Command Handler (CreateAssetCommandHandler.cs).

Depending on the nature of the handler, they contain more or less logic. Common for all handlers that processes commands, are that they map the command object to an entity object, then the relevant repository method is called. The returned entity object is then mapped to a DTO and returned to the controller. A query handler is simpler. It calls the relevant repository method, map the received entity object to a DTO and returns it to the controller.

During this process I use three implementations to make the code more reusable, readable, and compact. I use the *Option* type of *language-ext*, the AutoMapper package and some predefined *predicates delegates*.

The Option type is used each time I return something from the handlers or the repositories, except from the controllers. The Option type is taken from functional programming and encapsulates an *optional* value. Either the type is a *Some* and contains a value, or it is a *None* and empty. I use this to avoid returning nulls in my code. Having to deal with nulls in code increases complexity, is error prone, and makes the code less readable. Using the Option type requires me to think about potential issues in compile time rather that runtime. This makes

debugging easier. When an object is returned from the handler, I wrap the object in the Option type.

The AutoMapper library helps me to automatically map one object to another and omits the parameters that are not common between the objects. It also has implementation for mapping nested objects to flat ones, and the other way around. To use AutoMapper I must register a mapping profile for each object I want to map. This is displayed in Code 12. The nested object must be first, and the *ReverseMap* allows us to map both ways.

```
1 public MapperProfiles()
2 {
3     CreateMap<AssetEntity, CreateAssetCommand>().ReverseMap();
4     // Code removed for brevity
5 }
```

Code 12. Example of a Mapper Profile (MapperProfiles.cs).

In Code 13, I register the AutoMapper services, thus making it available for use via DI. The automatic mapping dramatically reduces lines of code, compared to writing the mappers myself. I considered where to do the mapping between the entity objects and the DTO's, the query objects, and the command objects. One alternative was to do the mapping in the controllers, whereas the other alternative was to do it in the handlers. In decided to do the mapping in the handlers, as I did not want to expose the entity objects to the API assembly. I concluded that it provided me with the opportunity to keep low coupling in code.

```
1 var handlersAssembly = AppDomain.CurrentDomain.Load("Bouvet.AssetHub.Handlers");
2 builder.Services.AddAutoMapper(handlersAssembly);
```

Code 13. Registration of the AutoMapper service (Program.cs).

The *predicate delegates* are methods that check if a parameter meets a set of criteria, which is type of a HOF used in functional programming [69]. The delegate takes one input parameter and returns a Boolean. Code 14 displays an example of a predicate delegate wrapped in an expression. I had to wrap the predicate delegate in an expression, to have the opportunity to set the criteria in which the comparison is made. In Code 14, the delegate parameter is the *AssetEntity*, and the criterion is the input parameter *id*.

```
1 public static Expression<Func<AssetEntity, bool>> ById(int id)
2 {
3     return (a => a.Id == id);
4 }
```

Code 14. Example of a predicate delegate (AssetsPredicates.cs).

These predefined delegates are used with the LINQ queries in the repositories. This allows me to reuse the repositories *Get* method, based on which value I am querying with. I will explain more on this in the next section.

3.3.2.9 Repositories

The repositories assembly consist of services that implement methods to access data in the database. I have created one repository class per entity object I want to access from the client. This is the *AssetEntity, LoanEntity, CategoryEntity,* and *LoanHistoryEntity.* These repository methods are collected in classes so that they can easily be reused. They also prevent handlers from violating the Single Responsibility Principle, as its responsibility is to process the request message received, and not deal the with database.

Each repository implements its own specific interface, and holds the CRUD methods necessary for that entity, thus implementing only what is needed. This enforces the Interface Segregation Principle, making the class only depend on an interface they actually use. The *DataContext* is dependency injected into the repository's constructor, thus the database is available to query on or do other mutations. Code 15 is an example of a *Get* method.

```
1 public async Task<Option<AssetEntity>> Get(Expression<Func<AssetEntity, bool>> predicate)
2 {
3     return await _context.Assets
4         .Include(a => a.Category)
5         .AsQueryable()
6         .Where(predicate)
7         .FirstOrDefaultAsync();
8 }
```

Code 15. Example of a repository method (AssetRepository.cs).

As explained in section 3.3.2.8, I have use predicates to make some repository method reusable. This *Get* method receives a predicate delegate in the input field. I do a LINQ query with this predicate, and return the queried data. As with the handler, the return is wrapped in

the Option type. If the query returns a null, the null gets wrapped in the Option type and becomes a *None*. However, if the query returns an object, it gets wrapped as a *Some*.

Code 16 display how a registered the repositories, making them available to be dependency injected into the handlers.

```
builder.Services.AddScoped<IAssetRepository, AssetRepository>();
builder.Services.AddScoped<ILoanRepository, LoanRepository>();
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ILoanHistoryRepository, LoanHistoryRepository>();
```

Code 16. Registering the repositories for dependency injection (Program.cs).

As mentioned previously, the backends main role, working as an API, is to implement CRUD operations. I have attempted to create backend that carries out this role, while ensuring principles are upheld. Thus, implementing the DI pattern and the mediator pattern, which promotes loosely coupled code. Each module in code in responsible for a single task, enforcing the single responsibility principle, thus tightly connected to high cohesion. The reason for making all these decisions is maintainability. Proceeding from the implementation of the backend, I am going to look briefly on the testing I carried out.

3.3.2.10 Testing

As mentioned in the introduction of this chapter, I intended to develop a backend that could perform CRUD operations. Based on Table I and the multiple sequence diagrams[14], I created all the elements necessary to perform the first round of testing. Testing is a part of the Agile Scum iteration and important part to make sure code is working as it is intended.

I used the SwaggerUI tool which is a web UI that allows one to test each method in my controllers. The tool uses the *OpenAPI specification*, which is a description of the APIs capabilities [70]. ASP.NET Core Web API implements middleware that generates the OpenAPI specification. The SwaggerUI tool allowed me to manually test every endpoint, which can be helpful in a development process and while debugging. Moreover, I implemented integration tests, that automatically tested all my endpoints. While doing changes to the code, I could easily check if any test failed and why. Thus, making it easy to debug and find the source of problems.

This concludes the section about the backend implementation. I will now move on to the implementation of the frontend.

[14] https://github.com/livun/bouvet-asset-hub/tree/dev/docs/sequence-diagrams

3.3.3 Frontend Development

In this section I will go through the implementation of the frontend. As with the backend, I had to understand what the frontends job were in the features I was implementing. It has essentially two jobs. The first is to provide the user with a dashboard, where *assets*, *loans*, *loan history*, and *categories* can be viewed. Secondly, to provide the user with the opportunity to perform actions on these resources, hence adding, editing, and deleting. Thus, the UI needed tables for the dashboards and forms to perform these actions.

Before going into details of how I achieved this, I will first discuss some decisions I made regarding design. Secondly, I am going to explain some core elements of creating a frontend with React and how it was structured. Thirdly, I will elaborate on some of the building blocks of my frontend application, hence the use of TS, the modules necessary to communicate with the backend, and the use of styled MUI components. Finally, I will describe how I used these building blocks to implement the different aspects of the user interface.

3.3.3.1 Design Decisions

When planning how I was going to design and create the user interface, I had to think about the end-users and whom they were. The solution is made primarily for Bouvet's IT department. They are used to work with Excel spreadsheet, various kinds of software and hardware, and possess a high level of technical skills. Given that the intended user is technology proficient, I focused on giving the user the most expressive options. This can be overwhelming for the average user, however this is not prioritized given the nature of the intended user. I will elaborate on this when I go into details about the table component in section 3.3.3.6

In the planning phase of Agile Scrum I created wireframes in *Figma*, which is a tool for design prototyping. A wireframe is a blueprint of the UI, providing a simple visual guide of how the web application can look like and can be connected [71]. Fig. 8 displays the basic layout of the UI in the browser.

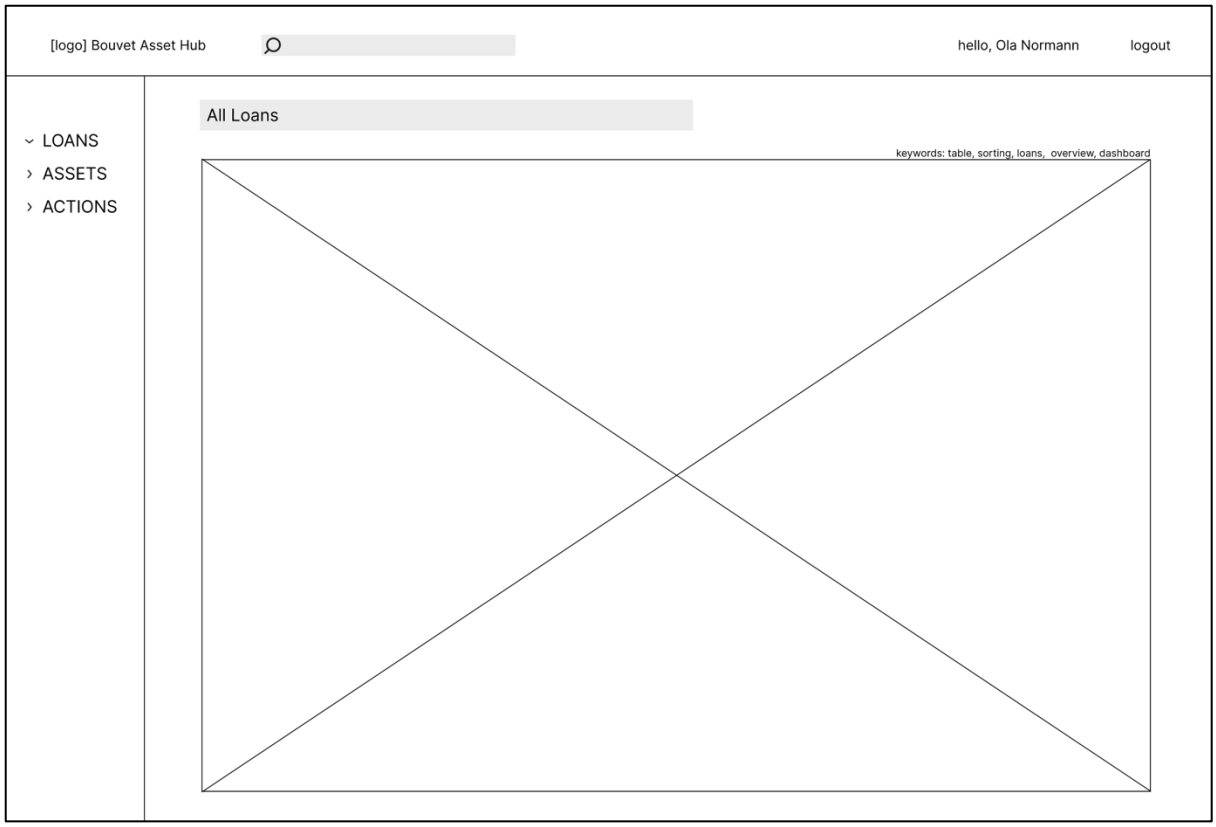

Fig. 8. Wireframe browser view.

I wanted there to be a dashboard were all the data could be displayed. Furthermore, a simple menu to access the different parts of the webpage. When adding the features that entailed scanning, I needed some sort of *Mobile* view, this is displayed in Fig. 9. Eventually this functionality could be its own independent mobile app, which could enhance the user experience, thus was out of the scope of this project. Therefore, I planned that the *Mobile* view would get its own route, where the layout and placement of components was especially planned for use from a smart phone.

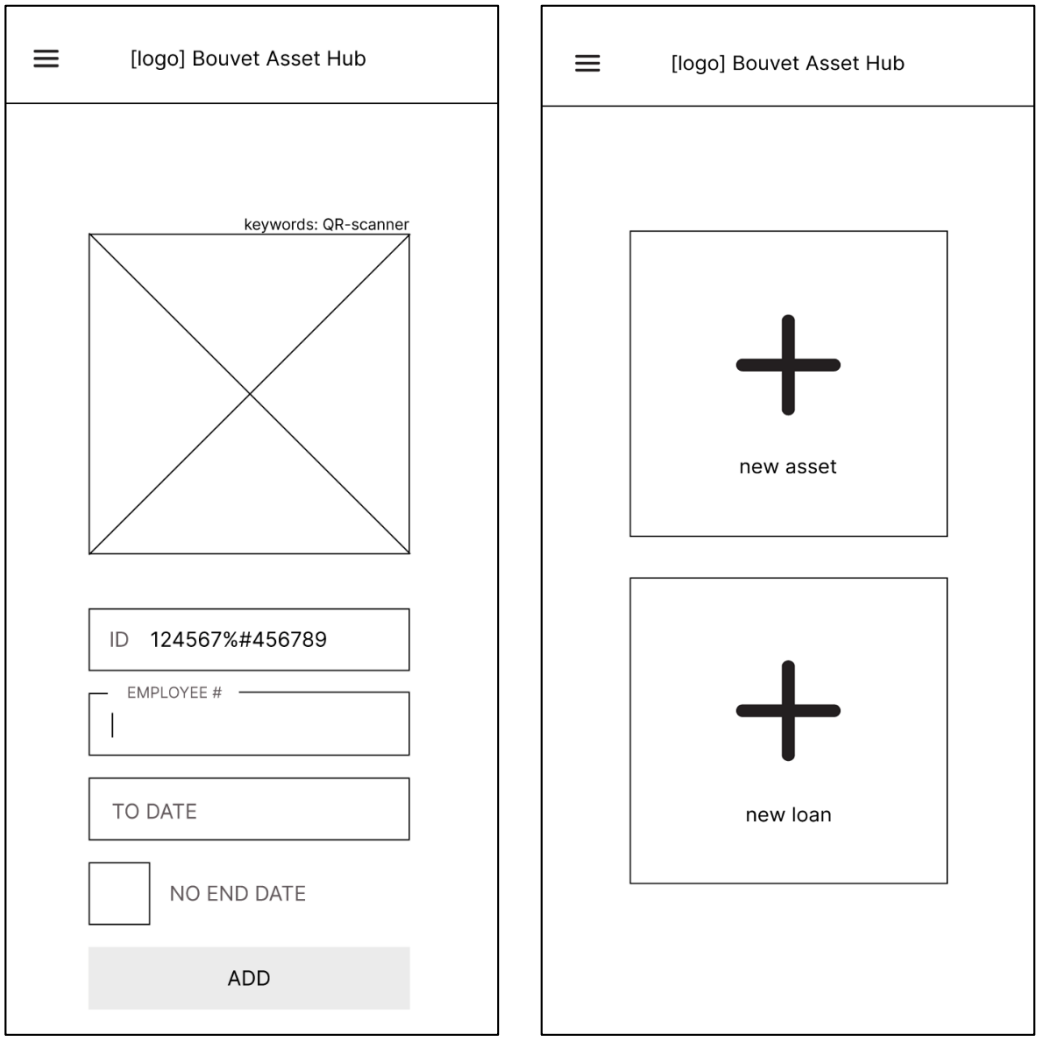

Fig. 9. Wireframes of Mobile view.

Based on these wireframes, I proceeded with the development process.

3.3.3.2 The Core Elements of the Frontend

As I described in section 2.1.1.1, the frontend is developed using the React library. A React app consists of several components, denoted by TSX. Which essentially is a collection of functions. These functional components call each other inside of their body. A component calling another is often indicated by a parent- and its child component. Before going into details on the components that make up my frontend, I want to briefly explain how the React code eventually runs in the browser and how I achieved this using a boilerplate.

In the first step, all the source code which consist of multiple TSX files, is translated by Babel. Babel is JS compiler that translates TSX to a specific version of JavaScript that can reliably run in the browser. The next step in the process is the bundling, where all the newly translated JS files are compressed by Webpack, which is a module bundler, into a few files. Finally, these files are now able to run on a JS engine in the browser. [72], [73]

I generated a boilerplate version of the React frontend with Node Package manager (NPM), The package manager helps me install and configure external JS modules. NPM makes sharing modules of code easy [74]. This boilerplate code comes configured with Babel and Webpack, and the basic structure of a runnable application. The frontend boilerplate holds the "index.tsx" which is the entry point of the application. The entirety of the application content is generated within this file. It is in the "index.tsx" where I get the reference point to the HTML and mount the *App* component. The *App* component is the outmost component, hence it does not reside in another component. *App* holds all the components of the frontend and renders them based on routing.

I have made a distinction between the components that deals with the views and routing, and the component that are rendered inside these views. From now, this is denoted by views (components) and components. Moreover, a view is unique, but a component can be reused. A view is rendered based on routing and nests several components inside it. In the source code, the views are collected in the "views" directory, and the components in the "components" directory. In addition, there are an "api"-, a "config"-, and an "utils"-directory. I made this distinction to make the frontend code clear and organized. I will get back to these directories in a later section.

To add routing, I used the NPM package *react-router-dom.* All packages that my frontend application depends resides in the "package.json" file. Code 17 displays how I wrap the *App* component in the *BrowserRouter* module. This is done, to make routing available in every component inside of *App*.

```
1 root.render(
2     // Code removed for brevity
3     <QueryClientProvider client={queryClient}>
4         <BrowserRouter>
5             <App />
6         </BrowserRouter>
7     </QueryClientProvider>
8 );
```

Code 17. Adding modules to the frontend (index.tsx).

The registration of routes is done in the *App* component. This is displayed in Code 18. Every route is indicated by the keyword *path* in line 2 to 7. The keyword *element* tells which view is being rendered on each path. I have wrapped each view in the *Main* component, which is a

reusable layout component I made to make sure the layout stays the same on each view in the browser.

```
// Code removed for brevity
<Routes>
    <Route path='assets' element={<Main open={open} child={<Assets/>}/>} />
    <Route path='assets/:id' element={<Main open={open} child={<Asset />} />} />
    <Route path='loans' element={<Main open={open} child={<Loans/>}/>} />
    <Route path='loans/:id' element={<Main open={open} child={<Loan />} />} />
    <Route path='loanhistory'element={<Main open={open} child={<LoanHistory/>}/>} />
    <Route path='categories' element={<Main open={open} child={<Categories />} />} />
    <Route path='mobile' element={<Mobile />} />
</Routes>
```

Code 18. Registering routes (App.tsx).

The *Mobile* view does not implement the *Main* component, as its layout is created specifically for use on a mobile.

3.3.3.3 Utilizing TypeScript

In section 3.2.2.1 I wrote about why I chose to use TS. Mainly because it adds type-safety and is explicitly typed. When using TS, one should explicitly add types to each variable in code. But to really utilize TS's strength, and make code readable and predictable, I created my own types and interfaces. Hence, I created interfaces of objects I used throughout the code. Interfaces is also useful when working with *props*.

Props can be any JS value, like objects and functions etc. When creating a functional component, we can define the props as function arguments. Props is how we pass data from a parent component to a child component. The child component thus renders based on this data. [72]

When passing several parameters as props to a component, creating an interface of these values makes the code more readable and neater. Thus, I have defined interfaces for several of the props I am passing throughout my components. This is not done in every component, as I realized while I was writing the frontend that this could be beneficial. I have therefore added this enhancement of frontend code as task to the backlog.

As I also mentioned in section 3.2.2.1, that TS makes writing the client code easier. The DTOs of the backend that defines how the retrieved data looks like, must be matched by types in the frontend. When doing this, I am in control of how the data looks like, thus making it easier to create components that uses these datatypes. I will now go into details on how I created these contract types.

3.3.3.4 Generating Client Types

To interact with the backend, I had to create an API client. This is the code which holds the HTTP requests and types. When interacting with an API with many endpoints it can be useful to generate the API client automatically to make the development process faster and prevent mistakes. One can do this with many open-source API client generators. They use the OpenAPI specification I mentioned in section 3.3.2.10. I attempted to do this myself with Acacode's TypeScript API generator library[15]. The API client that was generated by Acacode was difficult to implement, as the code did not appear and work as needed, especially considering error handling. To make sure that the client worked as needed, I decided to write it myself. More on this in the next section. However, I used Acacode and the OpenAPI specification to generate the client-types. These types work as contracts with the backend, as I introduced in section 3.2.2.1. The "_generated" directory holds "api-types.ts", "build-api.ps1" and "swagger.json". When I run the PowerShell script in the terminal, a file called "api-types.ts" is generated based on the OpenAPI specification in "swagger.json". This was convenient while developing. If I changed my DTOs during the development process, I could generate new and updated types with this script. This did not affect the API client I wrote myself, as long as I did not change the names on the DTO's in the backend.

3.3.3.5 Implementing Axios and TanStack Query

With the *client-types* in place, I implemented an API client with the Axios as I mentioned in section 3.2.2.2. In Code 19, I configured the Axios client with the URL and port of my backend. While developing, this is set to localhost, but would be changed to the URL of the server where the application is hosted in a production environment.

```
1 import axios from "axios";
2 export default axios.create({baseURL: "https://localhost:4000/api"});
```

Code 19. Configuring Axios API client (apiClient.ts).

Furthermore, I implemented four generic Axios HTTP request, one GET, one POST, one PUT, and one DELETE in "api/genericAxios.ts". These were used to perform the CRUD operations necessary to fulfil the features I was implementing. Code 20 displays examples of

[15] Acacode documentation: https://github.com/acacode/swagger-typescript-api

the generic GET and POST. I did not write any error handling in these methods, as TSQ handles this for me. The methods only need to return the data from the response.

```
export async function get<TResponse>(url: string, headers?: any) {
    const response = await apiClient.get<TResponse>(
        url,
        { headers: headers },
    );
    return response.data
}
export async function postItem<TRequest, TResponse>(url: string, body: TRequest, headers: any) {
    const response = await apiClient.post<TResponse>(
        url,
        body,
        { headers: headers },
    );
    return response.data
}
```

Code 20. Generic Axios HTTP requests (genericAxios.ts).

From these generic functions, I created specific functions, with the specific endpoints according to the resources in Table I. To provide a structure similar to the backend, I separated the specific functions based on the resource they were requesting for. Code 21 shows two examples of specific HTTP request.

```
export const getAssetByIdFn = async (id: number) => {
    return await get<AssetResponseDto>(`/assets/${id}`)
};
export const postAssetsFn = async (dto: CreateAssetDto) => {
    return await postItem<CreateAssetDto, AssetResponseDto>(`/assets`, dto, regularHeaders)
};
```

Code 21. Examples of specific Axios HTTP requests (assetApi.ts).

The *getAssetByIdFn* uses the generic *get*-function I made. The *postAssetsFn* uses the generic *postIte*m-function I made. When creating these specific methods, the input and output is based on the generated *client-types*, I introduced in the previous section. Thus, every HTTP request is bound to the contract defined in the backend.

I found my implementation more transparent and easier to work with, as opposed to the generated client I attempted to use first. This is because I separated the requests based on the resource I was working with, and the generic functions could be reused. Moreover, when I implemented it with TSQ it enhanced the developer experience.

While Axios performs the HTTP request, TSQ is used for handling the lifecycle of the request and the data. I have introduced the benefits of using it in section 3.2.2.2, I will now explain how I implemented it.

TSQ is configured in “config/queryClient.ts” and wraps *App* in “index.tsx”, as showed in Code 17, line 3 and 7. This is done to make the module available in every component in the frontend application. The two methods I have used from TSQ is *useQuery* and *useMutation*. The first is for querying and retrieving data and is always used with a HTTP GET. The latter is for mutating data, and is used with an HTTP -POST, -PUT or -DELETE. Code 22 present an example of *useQuery*.

```
1 const { isLoading, isSuccess, isError, error, data} =
2       useQuery<LoanResponseDto[], Error>(["loans"], getLoansFn)
```

Code 22. Example of useQuery (Loan.tsx).

Since using TS, I define the expected object to be queried. This is done in line 2, and is either an array of *LoanResponseDto* or an *Error*. In the input field, I specify the query-key *loans*, and which HTTP request method I will use the retrieve the data. In line 1, I use object *destructuring*, which is a way to unpack an object into individual variables [75]. This is useful because I can decide which properties of the object I want to use and be explicit about it. I often use *destructuring* when I unpack the props received in a component. How I use the variable destructured from *useQuery* will be described in section 3.3.3.7. Code 23 present an example of *useMutation*.

```
1 const addAsset = useMutation(() => postAssetsFn(assetForm), {
2     onError: () => {
3         openAlertBar("Cannot add asset.", false)
4     },
5     onSuccess: (data) => {
6         queryClient.invalidateQueries(["assets"])
7         // Code removed for brevity
8     }
9 });
```

Code 23. Example of useMutation (AddItemsFab.tsx).

The mutation from the example is created to add a new asset, thus the *postAssetsFn* is used. If the mutation fails, I can explicitly decide what happens next. In this example an *AlertBar* component gets rendered to notify the user about the error. However, if mutation is a success, I can use query invalidation to mark data as stale. As I described previously in section 3.2.2.2, TSQ caches the queried data on the query-key. When data is mutated, there is new- or changed data in the backend. I must therefore make sure that the frontend data is aligned. If a query on the invalidated query-key is being rendered, it will be re-fetched in the

background. In Code 23, it was added a new asset, thus the query-key *assets* becomes invalid because the frontend data is out-of-date.

The implementation of Axios and TSQ provides me the opportunity to seamlessly communicate with the backend. The TSQ *useQuery* and *useMutation* is used in every component where data is involved. Before presenting how I used the several building blocks to create the implemented the user interface, I am going to elaborate on the use of MUI components.

3.3.3.6 Utilizing MUI Components

All my views and components use at least one MUI component. As I mentioned in section 3.2.2.1, the components are styled and ready to use. It is important to differentiate the components that I have created, and the MUI components. MUI provides components that already holds a lot of HTML and CSS. This means that how they behave and how they look are already defined. I can tweak these configurations to adapt the components to my need. However, the components I have made, consist of several MUI components, to make up the whole layout and functionality of it. My components also hold data provided from the API, they hold the logic of what happens when forms are filled or buttons are clicked, and the conditional rendering. The MUI components work as building blocks of the layout, design, and functionality of my components. Hence my components work as building blocks of the web page and hold everything together.

The MUI X *DataGrid* component have an important function in my frontend application. When I wanted to create dashboards that displayed all assets, active loans, and previous loans, I needed tables that could hold a large amount of data and still be fast. The *DataGrid* component provides me with this quality, and other functionality relevant to the dashboard.

I have created a component called *DataGridTable*. This component is made reusable, hence I can reuse it in several views: *Assets, Loans, LoanHistory,* and *Categories*. Code 24 displays how I have added and configured the *DataGrid* MUI component in my DataGridTable component. The most important configuration is in line 2 and 4. *Rows* holds the data that is injected to the table. The data that is displayed in these tables are the DTOs retrieved from the API, for example an array of *AssetResponseDto*.

Columns holds the *GridColDef*, which is how the columns are defined in the table. A *GridColDef* is an array that consists of a column definition per column in the table.

```tsx
<DataGrid
    rows={rows}
    rowHeight={40}
    columns={gridColDef}
    pageSize={pageSize}
    onPageSizeChange={(newPageSize) => setPageSize(newPageSize)}
    rowsPerPageOptions={[10, 30, 50, 70, 100]}
    checkboxSelection={pathname === "/assets" ? true : false}
    // Code removed for brevity
/>
```

Code 24. Configuration of DataGrid MUI component (DataGridTable.tsx).

To make DataGridTable reusable, I must dynamically create the column definitions based on the object type that the table is displaying. If the object has dates, I want them to be displayed in a certain format etc. Therefore, I made two files in my utils directory, "columnFormatProviders.ts" and "ColumnFormatter.tsx". The first is a collection of methods that formats each field in an object type. This is exemplified in Code 25, where I have made a formatter that creates a column definition when the type of the object field is a string. The "filter" of line 2, checks if the object is indeed a string, and returns a *Boolean*.

```tsx
function formatString({key, object} : IFormatInput) : IFormatOutput {
    const filter = () => typeof (object) === "string";
    const value = () => {
        const col: GridColDef = {
            field: key,
            headerName: formatHeaderKeys(key),
            headerAlign: "left",
            align: "left",
            flex: 1,
            type: typeof (object)
        }
        return col
    }
    return {filter, value}
}
```

Code 25. Example of a formatter method (columnFormatProvidors.ts).

The second, the *ColumnsFormatter* component uses the object type that is going to be displayed in the table. Then maps through the array of formatter functions on each object field. It returns a column definition for that field if the filter is true. Finally, the component returns an array of all necessary column definitions, a complete *GridColDef*.

There are advantages of dynamically defining how the data in the columns of a *DataGrid* should look like. I do not need to do any changes to the component if a DTO changes in the backend, thus changing the data structure. Moreover, it is reusable, therefore preventing me from writing duplicate code. There are also advantages of using MUI X *DataGrid*, as I mentioned previously. It has some built-in functionalities that among other things provides sorting and filtering on each column. This is quite useful, as I do not need to create these functionalities myself. Some of my planned features was automatically achieved with this functionality. An example is the "Feature: View all loans linked to employee". With filtering, I can achieve this directly in the table without the need for creating a separate view and table. This functionality is displayed in Fig. 11.

The MUI X *DataGrid* is an advanced component which provided me an easy implementation of the dashboard. I have also used several other MUI components implement a coherent user interface. I have used *Box, Grid* and *Stack to* create layout. I have used *Buttons*, *Textfields*, and *Dialogs* to create forms for user input. The *SnackBar, Alert,* and *CircularProgress* components is used for giving feedback to the user. The *Menu* and *SpeedDial* components are used to create the navigation components. All these components are building blocks of my components, as I mentioned previously.

Next, I will elaborate on how I used these building blocks to create the views and components that make up the UI.

3.3.3.7 Views: Assets, Loans, LoanHistory and Categories

These views implement the *DataGridTable* component. The *Asset, Loans* and *LoanHistory* views are implemented similarly, due to the purpose, specifically to provide a dashboard that displays data. With these views I implemented and enabled these features:

- Feature: View all assets in the system
- Feature: Update and add information to one or more assets at the same time
- Feature: View all loans
- Feature: View all loans linked to employee
- Feature: View loan history

An example of a dashboard is shown in Fig. 10.

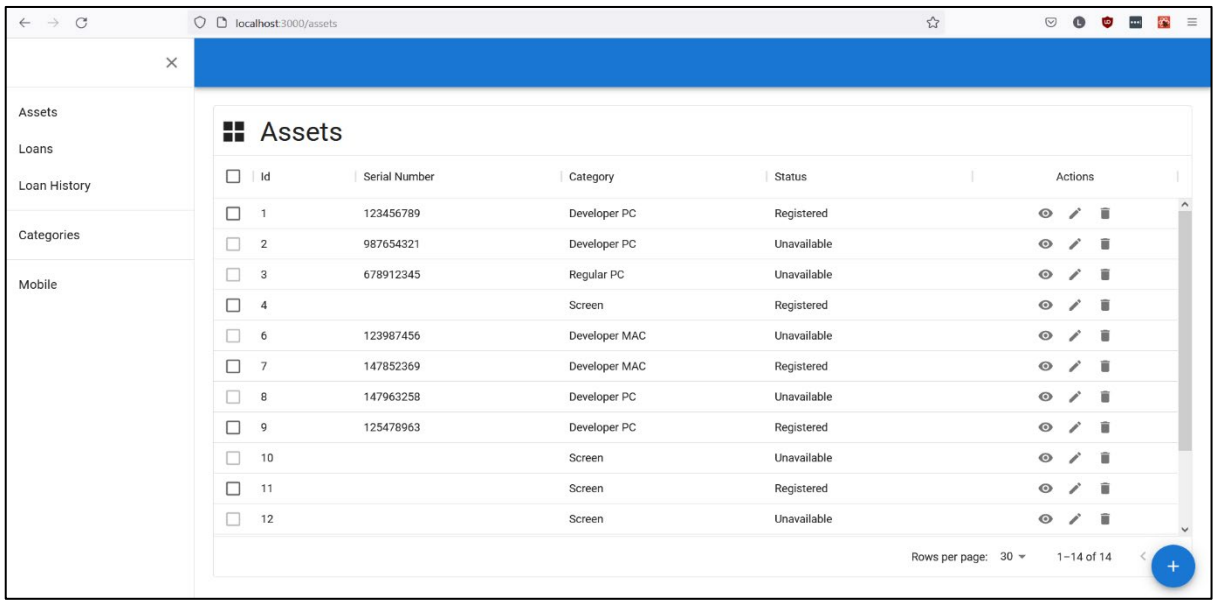

Fig. 10. Example of a dashboard.

Essentially what we see here is a table of all assets, hence an array of the *AssetResponseDto* is visualised. The checkboxes allow you to select all the assets, except from those which have the status *Unavailable*. When a checkbox is active, a form in the top right corner of table is rendered, and you can change the status of several assets at once. This form is not visualised in Fig. 10, as there are no active checkboxes. Each row in the table has buttons that link to actions, eventually these links lead to a view of the specific asset. I will elaborate on this view in the next section. This is similarly implemented in the *Loans* view. Fig. 11 display the *Loans* view with the filtering functionality in action.

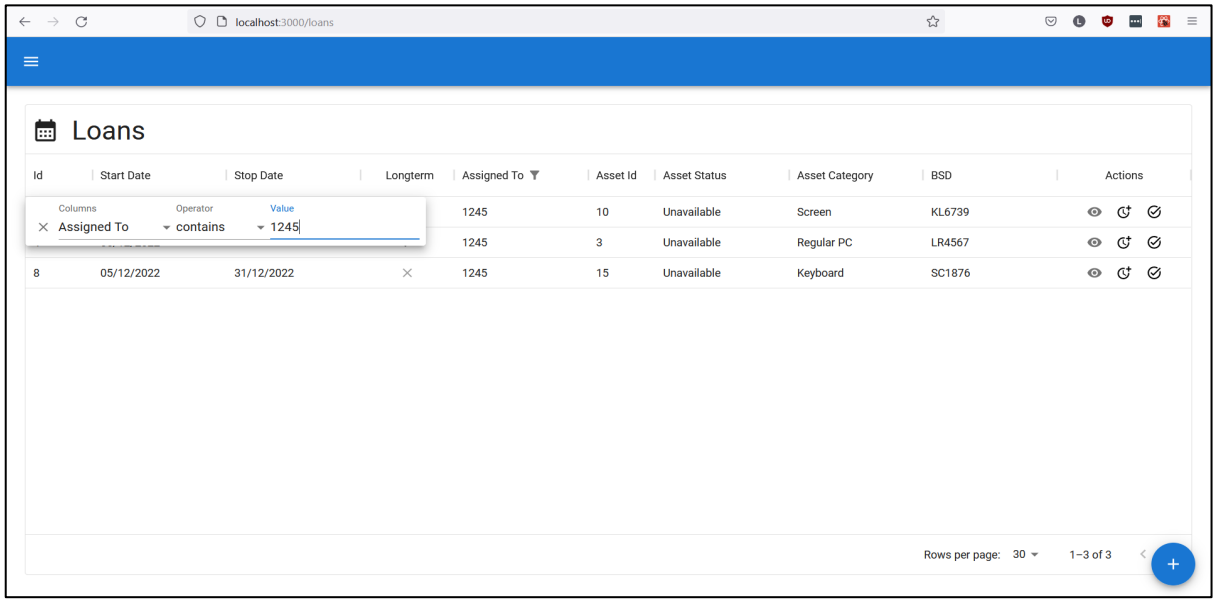

Fig. 11. The Loans view with filtering in action.

Code 26 displays an example of how I created the *Assets* view component. This component gets rendered by routing, as explained in section 3.3.3.2. In line 2 and 3, I use TSQ to retrieve data, as described in section 3.3.3.5. Then I display other components based on conditional rendering. I will now go through what is happening in line 5 to 10 in Code 26. There are several checks based on the Booleans received and *destructured* from the query. First the reusable *CircularLoader* component I have created is displayed while we are waiting for the data to be retrieved. If an error occurs, the *NotFound* component renders. If data is retrieved, thus *isSuccess* is true, the *DataGridTable* is rendered. The data retrieved from the query is passed as a prop to *DataGridTable* which renders the table as described in section 3.3.3.6.

```
1  export default function Assets() {
2      const { isLoading, isSuccess, isError, error, data} =
3            useQuery<AssetResponseDto[], Error>(["assets"], getAssetsFn)
4      return <>
5      { isLoading
6      ? <CircularLoader />
7      :  isError && axios.isAxiosError(error)
8      ? <NotFound message={error?.response?.data} />
9      : isSuccess
10     ? <DataGridTable<AssetResponseDto> rows={data} headerName="Assets" />
11     : <></>
12     }
13     <AddItemsFAB />
14     </>
15 }
```

Code 26. Example of a View component (Assets.txs).

The *AddItemsFAB* component is also rendered in these four views. This is a component that handles adding new items and uses the TSQ *useMutation* function described in section 3.3.3.5. I will elaborate on this component after I have described the other views.

The *Categories* view also uses the *DataGridTable*, but the purpose of the component is different from the other mentioned. This view displays all the categories available with the opportunity to edit and delete these items. It also renders an *DataGridTable* of assets, based on the selected category. Fig. 12 displays how this looks like.

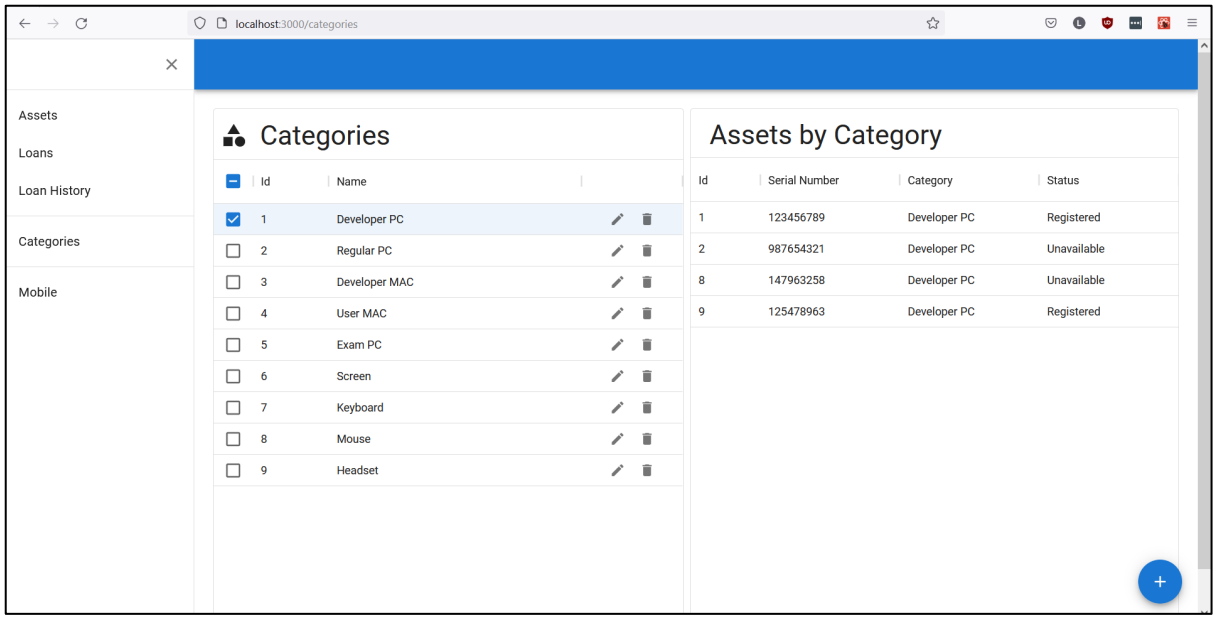

Fig. 12. The Categories view.

3.3.3.8 Views: Asset and Loan

The *Asset* and *Loan* view displays a single asset or a single loan, correspondingly. These views are dynamically routed denoted by the “assets/:id” and “loans/:id” routes, as is displayed in Code 18. In the *Asset* view, one can edit and delete the asset based on its status. In the *Loans* view, one can extend or hand-in a loan. With these views I implemented and enabled these features:

- Feature: View a single asset in the system
- Feature: Update and add information to one or more assets at the same time
- Feature: Delete a "wrongly added" asset
- Feature: Deprecate an asset
- Feature: Add loan to history
- Feature: View one loan
- Feature: Hand in loan
- Feature: Extend loan

Fig. 13 displays an example of the *Asset* view and the *Loans* view.

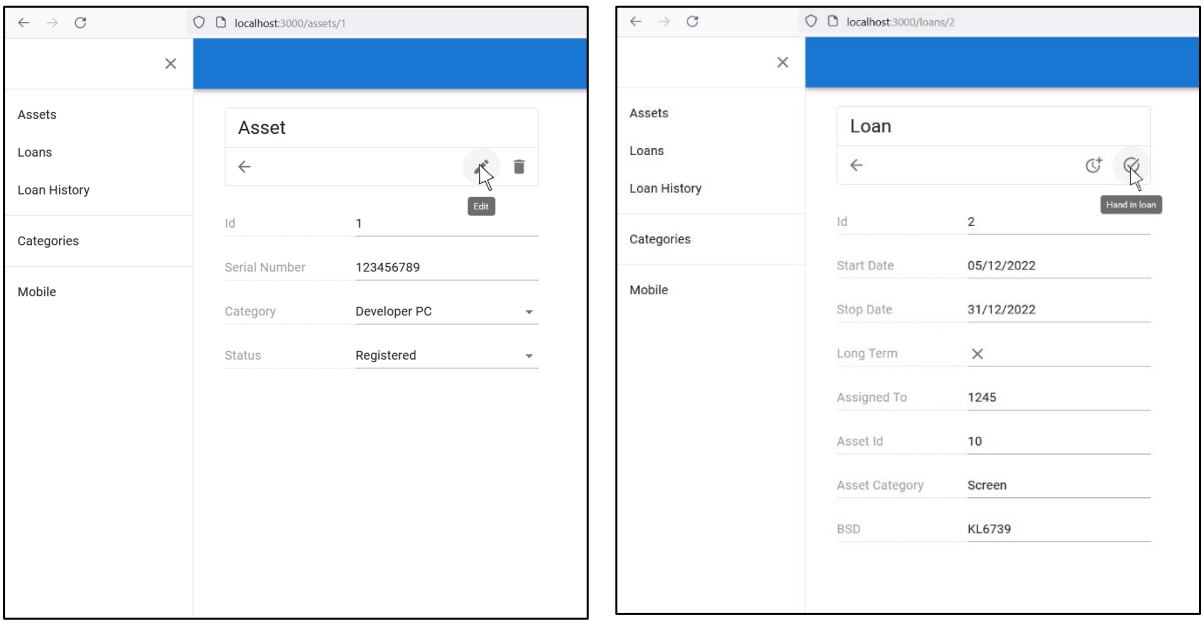

Fig. 13. Illustration of UI. (a) The Asset view. (b) The Loan view.

In Fig. 13(a) the edit button changes the state on the form from being inactive to active. All icon buttons used, make use of a MUI component called *Tooltip*. This provides a helper text when hovering these buttons, allowing us to read which action the icon represent.

The icon buttons, renders a MUI *Dialog* component, which is a window that appears in front of the content, disabling everything else until action is made. I have used dialogs almost every time I ask the user to perform an action. This will be exemplified in section 3.3.3.9.3.

3.3.3.9 Components

I have already mentioned several components I have created, that make up the views of the UI. The *DataGridTable* component, I presented in section 3.3.3.6, serves a substantial role in the UI. The other reusable components I have made are the *CircularLoader*, *NotFound*, *AlertBar*, *and Main*. The three first are components that provides information to the user. The latter is presented in section 3.3.3.2. I will not go into detail on the two first as they are self-explanatory. However, because the *AlertBar* is used extensively throughout the UI, this will be presented.

3.3.3.9.1 AlertBar

The *AlertBar* uses the MUI components *SnackBar* and *Alert*. *SnackBar* is a temporarily popup in the bottom left corner and *Alert* displays a short message and is coloured on severity of message. Used in combination, I have made a *AlertBar* that pops up every time a mutation is successful or not, with a message to the user. To make it reusable I use props to provide severity and a message. If there is an error, it renders red and if it's a success it renders green. Fig. 14 displays an example of the *AlertBar*.

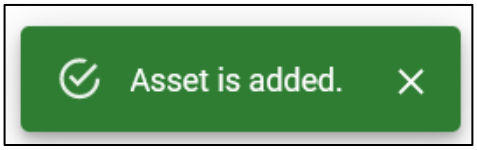

Fig. 14. The AlertBar component displaying a successfull message.

3.3.3.9.2 FullMenu

The *FullMenu* component consist of the MUI components *AppBar* and *Drawer*. The *AppBar* serves as a header and *Drawer* is used for navigation. It can be opened and closed on action from user. I wanted to use a side navigation to makes sure I had enough space for the all the routes. Moreover, enough space if I wanted to add several routes.

3.3.3.9.3 AddItemsFAB

The *AddItemsFAB*, mentioned in section 3.3.3.7, uses the MUI component *SpeedDial*. FAB is abbreviated from Floating Action Button, and that is essentially how it appears, in the lower right corner in the views that also display the dashboard. Add Items describes its role,

specifically to add new items, hence an asset, a loan, or a category. Fig. 15 displays the *AddItemsFAB*.

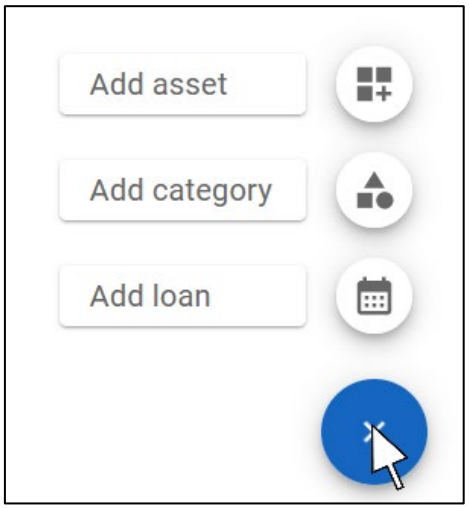

Fig. 15. Illustration of an active FAB.

The action buttons render a form in a dialog, introduced in section 3.3.3.8. The dialog and the FAB floats independently from the rest of the layout. This provides an easy implementation with other components as it does not affect them or the layout. Fig. 16 display a dialog with a form that is used to add new loan.

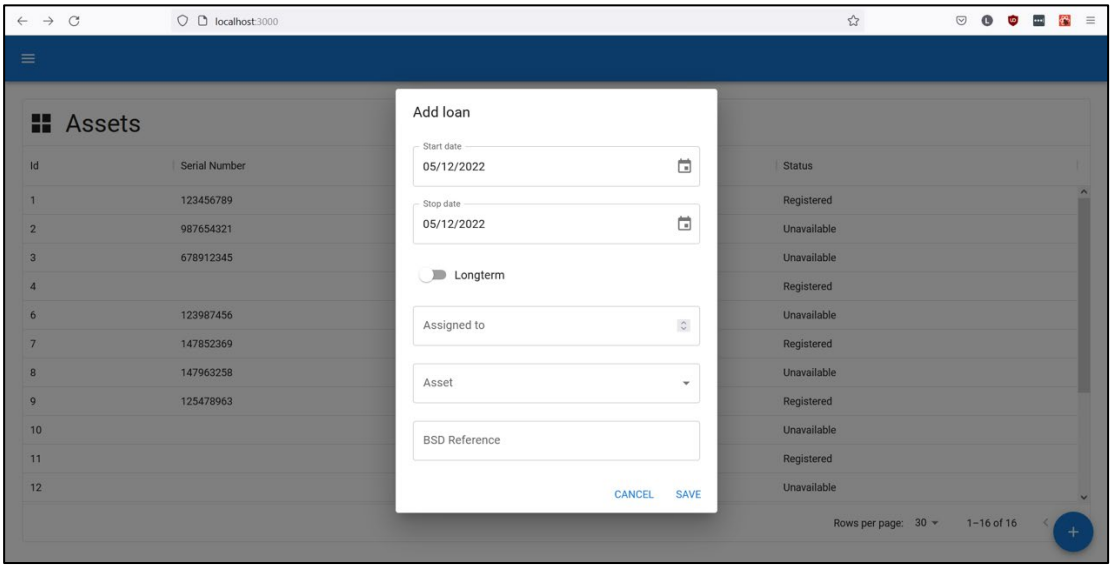

Fig. 16. Illustration of the Add Loan dialog.

With this component I implemented and enabled these features:

- Feature: Add new asset
- Feature: Add new loan
- Feature: Add new category

The solution requirements presented in section 3.1.2, essentially asked for a dynamic way to add new assets and loans. This is described in the features and the user stories that entails scanning. Regardless, I wanted to add a manual way to add new items as well, because the functionality is more or less the same, except from the scanning. I will now continue with the implementation of the *Mobile* view, which provides the dynamic way to add assets and loans.

3.3.3.10 The Mobile View

As presented in section 3.3.3.1, I decided to add a view in my frontend created specifically to be used from a mobile. The *Mobile* view uses the components in the "components/mobile" directory. The features that this view implements are:

- Feature: Add new asset
- Feature: Scan asset and read information
- Feature: Scan asset and lend out to employee
- Feature: Scan asset and extend the loan
- Feature: Scan asset and hand in loan/asset

To dynamically add new assets I wrote out some user stories in section 3.1.3.1, that I want to repeat as they provides elaborate descriptions on how I should implement the adding of new asset.

1. *As a user, I want to do a barcode scan of the serial number of an asset, identify the serial number, then generate a QR code, print out a label, which I can stick on the asset, to make it easier to manage that asset.*
2. *As a user, I want to add assets without serial number, with generating a unique id and QR-code and stick to asset, to make it easier to manage that asset.*

To meet the requirements of the first user story, I needed to implement a barcode-scanner, a function that created QR-codes and an implementation towards a label writer. The second, was implemented in the previous section, except from the label writing. The other features required an implementation of a QR-scanner.

3.3.3.10.1 Barcode Scanner

I used the NPM package *html5-qrcode*[16] together with a React implementation of this module "HTML5QrCodePlugin/Html5QrcodePlugin.jsx"[17]. I placed the module in my "src" directory and created a wrapper component *HTML5QrWrapper* to inject this module into my *NewAssetMobile* component. This module allowed me to scan barcodes and QR-codes, with the mobile camera and from uploading an image. The drawback of using this package, was that scanning from mobile camera took some time, and gave unstable results. The quality and size of the barcode was important. Nevertheless, taking a photo and uploading it worked fine.

[16] html5-qrcode documentation: https://github.com/mebjas/html5-qrcode
[17] Html5-qrcode-React documentation: https://github.com/scanapp-org/html5-qrcode-react

I decided to use this module for the pilot and for demonstration purposes but added an enhancement of this feature as a task to the backlog.

3.3.3.10.2 QR Service and Label Printing

The purpose of generating a QR-code was to print it using a label printer. Therefore, when I add a new asset, a Globally Unique Identifier (GUID) connected to that new asset is created. As it is unique, it is used to generate a QR-code with the NPM package node-qrcode[18]. The GUID gets stored in the database, and when a QR-label is scanned, I can retrieve the asset on the GUID.

In the "utils/Qr-service.tsx" I have started this implementation. However, due to hardware requirements not yet available at development time, I could not complete it. The label printer was not available, hence I could not implement the printing of the generated QR-code, neither the sending of the QR-image to the label printer. These missing functionalities were added as a new feature to the backlog. Nevertheless, I decided that I could implement the remaining functionalities: to store the asset to the database, to create the GUID, and to generate the QR-code.

3.3.3.10.3 QR Scanner

Before starting to implement the barcode scanner, I had already created the *QRScanner* component. This could not read barcodes, hence I had to implement a barcode scanner. The *QRScanner* component is implemented with the NPM packages *jsQR*[19] and React Webcam[20]. Code 27 displays the *scan* function which perform the scanning and reading of the identified QR-code in image. The React *Webcam* activates the camera on a phone, then the video retrieved is drawn in a *HTML Canvas* element. Then the *jsQR* method is called with the image data received from the canvas. This method reads and identifies the QR-code. The *requestAnimationFrame* is a built-in JS function that runs the *scan* function in a loop until it has identified a QR-code. The *scan* method is called when the component is rendered. The component returns the identified QR-code to its parent component.

[18] node-qrcode documentation: https://github.com/soldair/node-qrcode
[19] jsQR documentation: https://github.com/cozmo/jsQR
[20] React Webcam documentation: https://github.com/mozmorris/react-webcam

```
function scan() {
    const video = videoRef.current?.video
    if (video && video.readyState === video.HAVE_ENOUGH_DATA) {
        setIsVideoLoading(false);
        const canvasElement = canvasRef.current;
        const canvas = canvasElement?.getContext("2d");
            if (canvas && canvasElement) {
                canvasElement.height = video.videoHeight;
                canvasElement.width = video.videoWidth;
                canvas.drawImage(video, 0, 0, canvasElement.width,
                                 canvasElement.height);
                const imageData = canvas.getImageData(0, 0, canvasElement.width,
                                                       canvasElement.height);
                var code = jsQR(imageData?.data, imageData?.width,
                                imageData?.height, {inversionAttempts:
                                                    "dontInvert",}
                               );
                if (code && canvas) {
                    // Drawline() removed for brevity, the method draw a square
                    // around the identified QR code
                    handleQrGuid(code.data);
                }
            }
    }
    requestAnimationFrame(scan);
}
```

Code 27. The Scan method of QR-Scanner (QRScanner.tsx).

As opposed to the barcode scanner, the *QRScanner* component worked well at identifying QR-codes, hence I did not want to discard this component before I eventually had a scanner that performed better and did both QR and barcode reading. The drawback of this component is the use of the NPM packages and the built-in JS function *drawImage*. I will elaborate on this in section 4.2.1.1. Consequently, this is also added as a task to the backlog as a possible enhancement.

Having these modules in place, I could continue to implement the *Mobile* view. Fig. 17, display the actions possible to make from the *Mobile* view. Either do a QR- scanning or adding a new asset, when clicking the QR-icon button or clicking the plus-icon button, correspondingly.

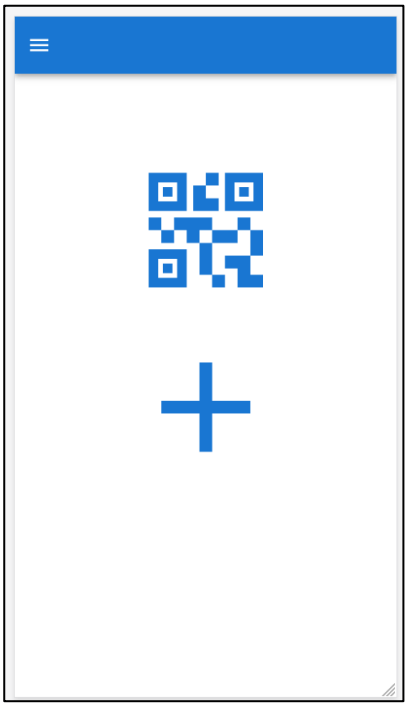

Fig. 17. Illustration of the menu in the Mobile view.

3.3.3.10.4 Add Asset

When choosing to add an asset, the *NewAssetMobile* component is activated. This holds the *HTML5QrWrapper* component, which makes barcode scanning possible. When decoded text, hence a serial number, is identified, it is returned to *NewAssetMobile***.** Furthermore, the component opens a dialog, asking the user to confirm that the serial number identifies is correct. If not, the *HTML5QrWrapper* is activated, and the user can perform a new scanning. If it is correct, a dialog with a form opens and user can set the asset type and save it to the database. As I have elaborated on earlier, *useMutation* handles this process. Fig. 18 demonstrates how to add a new asset from the *Mobile* view.

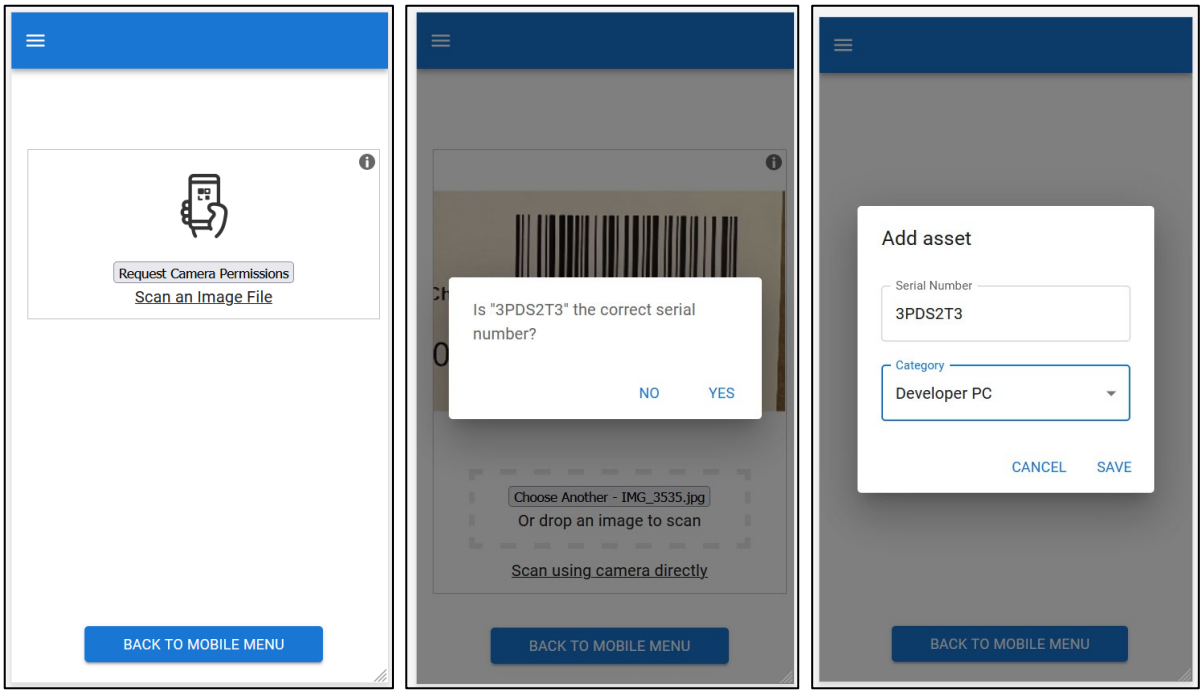

Fig. 18. Illustration of barcode scanner. (a) The scanning window using the HTML5QrWrapper. (b) Confirmation of barcode scan dialog. (c) Add asset form dialog.

3.3.3.10.5 Scanning Asset

When choosing to scan an asset, the *MobileActions* component is activated. This holds the *QRScanner* component, which makes QR scanning possible. When there is an identified GUID retrieved from the *QRScanner*, a *useQuery* is called to retrieve the asset and furthermore another query for the loan on that asset, if there exists one. Code 28 displays the *loanQuery* with the enabled flag activated in line 5. This means that the query is inactive and will not execute before an *asset* is retrieved and a *assetId* exist.

```
1 const loanQuery =
2       useQuery<LoanResponseDto, Error>(["loan", assetId], () =>
3                                        getLoanByAssetIdFn(assetId), {
4         onSuccess: (data: LoanResponseDto) =>s etLoanId(data.id),
5         enabled: !!assetId
6 });
```

Code 28. Example of useQuery with "enabled" activated (MobileActions.tsx).

When *MobileActions* have received the asset and potentially a loan, the user now has several options. If there is no loan and the asset status is set to available, the user can create a new loan with that asset. On the other hand, if the asset status is unavailable and there exist a loan, the user can choose to extend or hand in the loan. Fig. 19 displays these options.

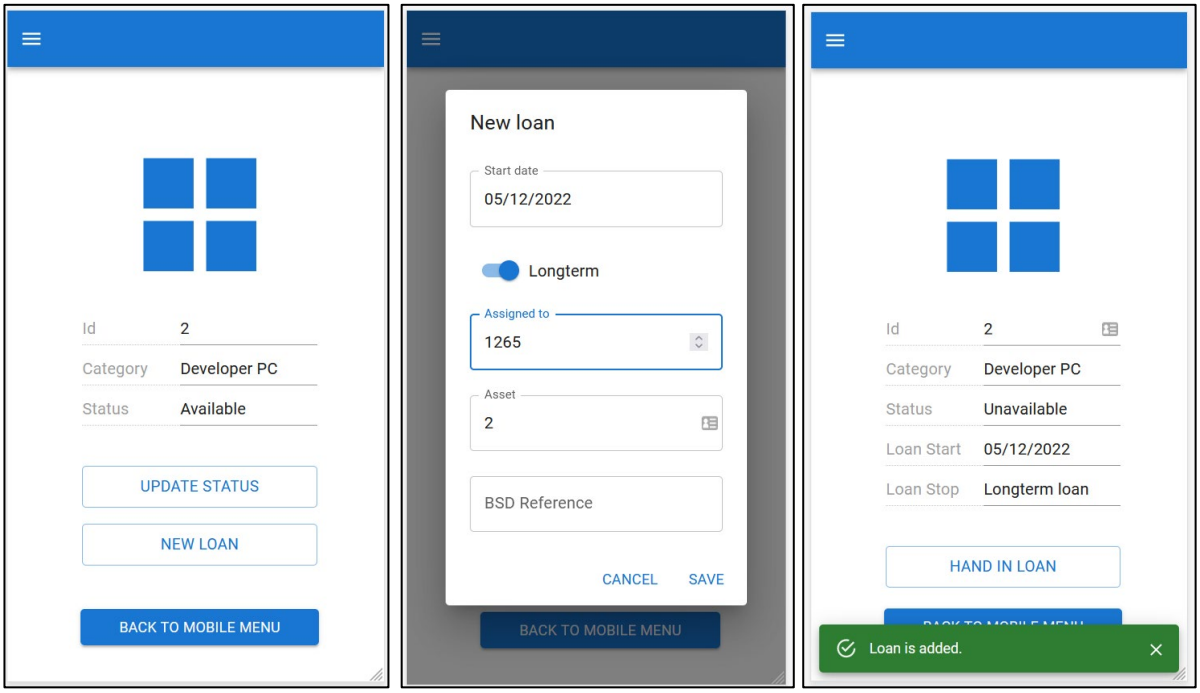

Fig. 19. Illustration of possible actions after a QR scan has identified an asset. (a) Information window of an available asset. (b) New loan dialog. (c) Information window of an unavailable asset.

With the implementation of the *NewAssetMobile* and *MoibleActions* component, I have fulfilled the feasible requirements of the user stories. These implementations come with several limitations. I will address the limitation and shortcomings of my pilot in section 4.2.1. However, before doing so, I will continue with describing the database implementation and the containerization of the solution.

3.3.4 Database Implementation

In section 3.3.2.7, I described how I implemented the data model with the ORM tool EF Core. When using an ORM, it is not necessary to create any data access code except from the LINQ queries in the repositories classes I presented in section 3.3.2.9. The LINQ queries gets translated to SQL by the database provider implemented with the ORM. Fig. 20 displays the data model of the database.

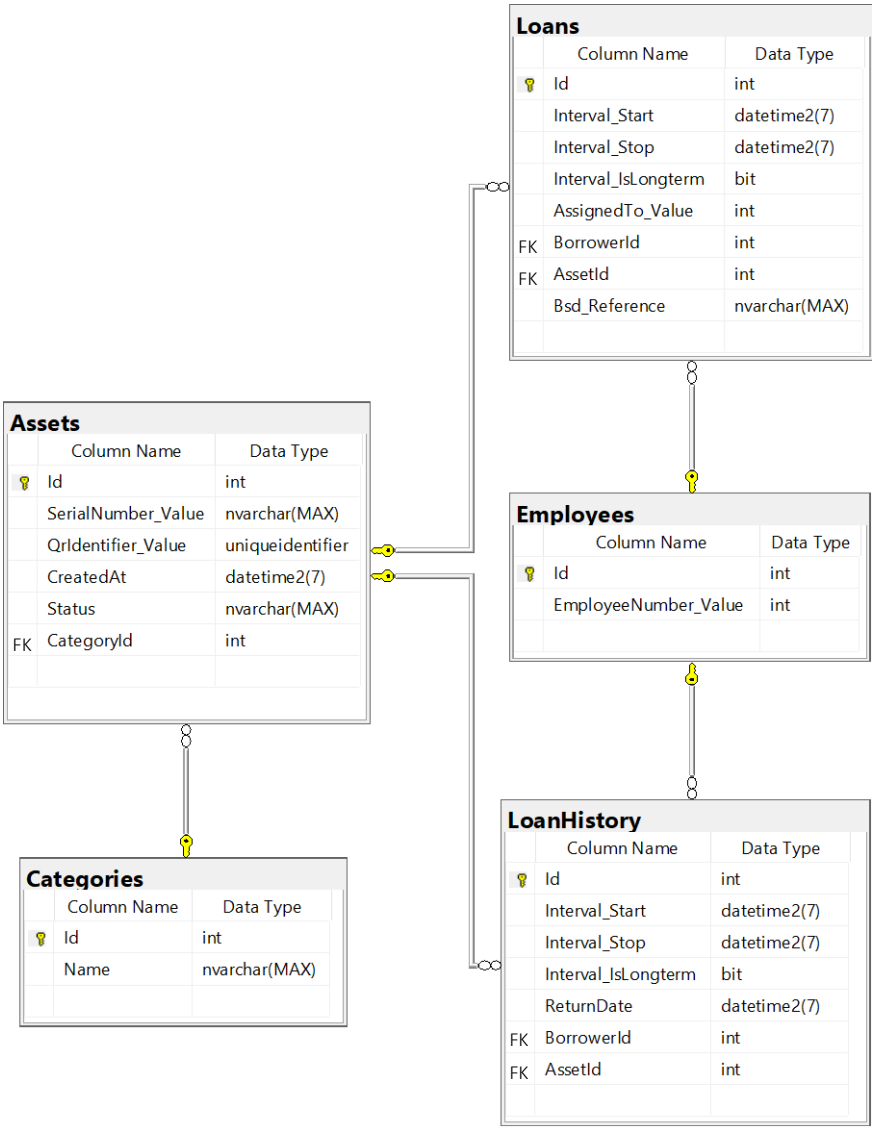

Fig. 20. Data Model diagram.

3.3.5 Containerizing the Solution

I am going to describe how I have arranged for the application to run using Docker Containers. As described in section 2.1.5, a Docker Image is necessary to create a Docker Container. Hence, I needed three images, one for the database, one for the frontend, and one for the backend. While I wanted to use Bouvet's own server to store the data in production, I did not have access to this while developing. Therefore, I decided to use a Docker Container for the MSSQL Server. To do this I used the MSSQL Server image which is available at Microsoft Artifact Registry[21]. The images for the frontend and the backend are built from a *Dockerfile*, which resides in the root folder together with the entry-points of the applications. A Dockerfile is a text document with instructions on how to build the application. The Dockerfile for the backend is generated by Visual Studio IDE, which is a programming tool from Microsoft. The Dockerfile for the frontend is inspired from the Medium article by Mwila [76]. One of the Dockerfile commands is FROM, which specifies the parent image. The following commands are based on this parent image [77]. In my case the parent images hold the frameworks and runtimes needed to build and run the applications I have created. These parent images resides in a registry, which is a repository of images [77]. Microsoft has its own registry, as mentioned previously. There is also a default registry that can be browsed on Docker Hub. Hence, when creating a container instance from an image, we do not need to install any frameworks, libraries, or dependencies. It is all contained in the container.

To create container instances from these images, I used the *Docker Compose* tool. This tool is used for defining and running a multi-container application. I have created a "docker-compose.yml"[22] file, which is a configuration file, were I have defined the three services of the solution.

When defining a service, there are several steps involved. Defining the image, the container should depend on. Defining from where the image should be built if it does not exist, nor is from a registry. Defining the ports (HOST_PORT : CONTAINER_PORT), volumes and environment variables. The Docker Compose tool sets up a network for all the services by default, that means that the services are reachable by all the containers on that network. The containers communicate inside the network on the container-ports. If they are reachable from the outside, the host-port is also defined. [78]

[21] https://mcr.microsoft.com/
[22] https://github.com/livun/bouvet-asset-hub/blob/dev/src/docker-compose.yml

When defining the database service, I have defined which image it depends on, this image already exists in the Microsoft Artifact Registry. When the compose file is eventually executed, it pulls that image from the registry. I also define two environment variables, a password, and the ACCEPT_EULA=Y, which is to confirm acceptance of a licensing agreement. This is necessary to use the MSSQL server image. The last thing I must define for the database, is the *Docker Volume.* This is to persist the data stored in the database if the container is deleted and must be rebuilt. This volume is connected to a physical location on the host.

In the Docker Compose file, I must add the Docker Volume and define it to be an external volume. Before executing the compose, I must create this volume. This is all explained in the "README.md" on GitHub[23].

When defining the frontend service, I define the image the container is going use. Then, I define that the image must be built, and where to find the Dockerfile to build this image. Finally, I define which port the frontend is accessible from.

When defining the backend service, this process is similar to defining the frontend service. The process differs on three aspects. Firstly, there are two ports defined, instead of only one. One for HTTP and one for HTTPS. Secondly, the backend is defined to depend on the database, whereas the frontend does not depend on any other services. The dependency implies that the container instance of the backend must be created after the container instance of the database. This defined order of creation is due to the fact that when the backend is built, it is configured to connect to an existing database. Lastly, there is a certificate defined, as opposed to the frontend where I do not define any certificates. The certificates are necessary for hosting over HTTPS, which is the default for ASP.NET Core. I have used the *dotnet dev-certs* tool for using a self-signed certificate for development and hosting on localhost [79].

When all the services and volumes are defined in the Docker Compose file, I can execute it. To do this I need a software named Docker Desktop, which includes the Docker Engine and the Docker Compose tool. The *Docker Engine* consists of the technology that builds and containerizes the application [80].

[23] https://github.com/livun/bouvet-asset-hub/blob/dev/README.md

To configure the Docker Compose for production and to be hosted on the web, there are a few more steps that needs to be completed. As for the pilot, I decided to keep this process out of the scope of the project, due to a restricted timeframe. Nevertheless, it is now possible to run the application with Docker Compose. It creates three containers, which can run on the local network, communicate, and demonstrate the solution.

3.3.6 Summary of the Pilot

This section concludes the implementation of the pilot. I have thoroughly described how I decided to execute the implementation of the features identified in the Agile Scrum planning phases. I have developed a backend that interacts with a database, and a frontend. This is all described in section 3.3.2, 3.3.3, and 3.3.4. I have also elaborated on the use of Docker as deployment method. This pilot has its limitations and shortcoming, which I will describe in section 4.2.1 of the Results chapter. Moreover, I made use of the backlog while developing and moved the features and tasks along the *Kanban* board to visualise and organise the development process.

In the next chapter I will discuss the results of this project and assess the accomplishment of the proposed goals.

4 Results

In this chapter I will present the results of my thesis. I will first assess the use of Agile Scrum, thus how I planned the solution and facilitated for a handover. Finally, I am going to discuss the outcome of the implementation, hence the pilot and its limitation. I will assess if the initial goal of thesis was achieved.

4.1 Assessing the Use of Agile Scrum and GitHub Projects

I have presented how I have used Agile Scrum throughout the thesis. This project management framework provided me with the necessary instruments to plan and execute the pilot. GitHub Projects provided me with the tools to carry out Agile Scrum, as I used it to organize and visualise the backlog. To create a comprehensive and complete solution, the use of the Agile Scrum methodology was very convenient and helpful. The two planning phases resulted in the requirements, features and user stories presented in section 3.1.3. The most challenging part of planning was deciding which features I should prioritize for the pilot. This was the first step of the agile iteration cycle. I believe that the scope of the implemented features became too large. I think that I could have achieved a more maintainable and deployed solution, but with fewer functionalities if I had been more cautious in this first step. I will continue this assertion when discussing the results of the pilot, after I discuss the handover.

The use of Agile Scrum and GitHub Projects facilitated for an easier handover. The project is thoroughly documented and developed with the handover in mind. GitHub projects contains a backlog filled with feature, tasks, and enhancements. This will be made available for the handover team. The backlog distinguishes on requirements necessary to complete the solution, and the enhancements. The implementation of Azure AD, the integration towards Xledger and the label printing is required. While better scanning functionalities etc., are enhancements. I have also focused on creating maintainable code with the handover in mind, I believe this is achieved specifically in the backend implementation. This deemed to be more challenging in the frontend, due to the limitations of the packages used and the scope of the pilot. Nevertheless, the project is made ready for a handover, where the necessary information will be provided.

4.2 Assessing the Pilot

As mentioned in the previous section, I believe that I could have done a better job at planning which features to implement for the pilot. This is primarily because of how the *Mobile* view was implemented. While some features were not possible to implement because of hardware requirements, the quality and maintainability of the *Mobile* view features are not satisfactory. In the next section, I will discuss the limitations and shortcoming of the implementation. Hence, the Mobile view, frontend components and the lack of validation. I will also elaborate on how the features not implemented limits the pilot. After discussing the limitations, I will conclude with an assessment of the pilot.

4.2.1 Limitations and Shortcomings of Implementation

I am first going to address the use of NPM packages, the limitations of the *Mobile* view and other components. Secondly, I will elaborate on the lack of validation, on the client side and server side. Finally, I will mention how the features that are not implemented limits the pilot.

4.2.1.1 NPM and the Mobile View

Use of the NPM packages in the *Mobile* view comes with several limitations, as I have already mentioned in section 3.3.3.10. Maintainability is also important when developing the frontend, as well as the backend. When using NPM packages, it is important to be aware of the packages' vulnerabilities and overall health. Snyk Advisor[24] is an open-source tool that gives packages a health score from 1 to 100, based on security, popularity, maintenance, and community [81]. The tool is used to check the health score of the implemented packages. The packages used in the *Mobile* view, have mediocre health scores. This is primarily because they are no longer being maintained. The use of these packages affects the maintainability of my frontend application negatively.

Furthermore, the implementations of scanners impose a limitation to the solution. The *QRScanner* component uses the built-in JS function *drawImage*, mentioned in section 3.3.3.10.3. There is not support for this method in Safari on IOS. Regardless, the *QRScanner* is supported in other bowsers on a mobile, i.e., Chrome or Firefox. The *html5-qrcode* package, does is not provide an optimal solution, because of its unstable results. Therefore, I

[24] https://snyk.io/advisor/

have added tasks in my backlog to enhance these features, with a suggestion to create a scanner that handles both QR-codes and barcodes.

4.2.1.2 Frontend Components

Several of the non-reusable frontend components got very big and complex. This affects the code readability and maintenance. Therefore they should have been split into several reusable components. Which prevents me from writing duplicate code and is therefore a more efficient way to develop. When I saw that the components got big and complex, I did not have time to improve them. Therefore, I have added this enhancement of the frontend components as a task to the backlog.

4.2.1.3 Validation

I have not implemented validators in the backend, nor have I implemented user feedback on the forms in the frontend. This was not implemented due to time constraints. When it comes to validation in the backend, this is important for security and data accuracy. Thus, preventing malicious users from submitting invalid data. Moreover, making sure that the retrieved data matches the expected data and conforms to defined business rules. One example from my solution is a validation of the *Interval* class. A validator should have ensured that the stop date was after a start date. As ASP.NET Core Web API has built-in encoding, it automatically validates the incoming data against the excepted data. If it is not a match, the request returns a *Bad Request*. Nevertheless, I have added a task to the backlog to implement business rule validators.

When it comes to frontend validation, this is useful for the user experience, and to reduce the load on the backend. The forms have restrictions added to them, but there is no information feedback to the user, if something is wrong. This is a limitation because it reduces the user experience. Hence this enhancement is added as a task to the backlog.

4.2.1.4 The Features Not Implemented

A limitation regarding security is the fact that I chose to not implement Azure AD in the pilot. Azure Ad provides security to the system, as it authorizes and authenticates the users. As the functionalities of this system is created for the IT department and Bouvet, it should therefore only be accessible for them. Hence an implementation of Azure AD is a requirement. I decided that to pilot the solution, and to showcase the functionalities, it was

not imperative to implement Azure AD. Nevertheless, it must be implemented before it is sat in production.

Another feature I decided to not implement for the pilot, is the integration towards Xledger. This does not necessarily limit the solution, but it should be integrated as it is an important requirement for the scope of a finished solution. This feature lowers the workload on the IT department, and make sure that every aspect of asset tracking is thought of. The reason for not implementing this feature during the pilot, was that not all required information about the Xledger's API, was available to me during the development phase. This made it impossible to fulfil and thus out of the scope of the pilot.

As mentioned in section 3.3.3.10.2, hardware requirements prevented me from finishing the feature that involved sticking a label to a new asset that was added. This feature is very important, as forms the basis for easy identification and handling of a single asset. Without the labels, this becomes very cumbersome.

The last feature I did not implement, was the notification feature, reminding the lender to return its loan. I decided that this feature was out of scope regarding the time frame I had to develop the pilot. This feature also holds an important part of a finished solution as with the Xledger integration. To implement this feature, I would have created a new service in the backend, that essentially had two jobs. First to check the database daily for loans that were reaching its expiration date. Secondly, to send an email to notify the lender when it is time to hand in the asset. This feature remains in the backlog, along with the other features that were not implemented.

When addressing these limitations, I have mentioned several times that the backlog contains these missing features and enhancements as tasks. This is done to provide an easier handover of the project. I have discussed the handover in section 4.1.

4.2.2 Assessment of the Implementation

The goal of the thesis was presented in section 1.3, which initially was to help Bouvet keep track of its assets and ease the workload on the IT department. In section 3.3.1 I have summarized the features implemented in the pilot. The pilot demonstrates a solution that meets the initial goals at some level, but as discussed in the previous sections, also has its shortcomings. An important part of the result is the identification of requirements and features of said solution. While the pilot has not achieved every requirement, this is not its main objective. The pilot demonstrates a large range of functionalities improving and enhancing the asset tracking experience. It reduces the amount of manual work, by implementing an automatic way to register new assets. In the past the registration of new assets has been maintained manually with entries in Excel spreadsheets. The implementations of features involving loans also enhances the asset tracking, as there now exist specific functionality for this. A finalized solution will provide a more efficient way to keep track of the assets in the company.

5 Conclusion

In this thesis I have demonstrated the process of developing a pilot for an asset management system for Bouvet. The results demonstrate the importance of planning, whereas it provided me with a clear overview of what to develop and how to facilitate the handover. The pilot has revealed some limitations which I have elaborated on in section 4.2.1. As expected, the pilot does not include all the implemented functionalities that are required for production. These are described in my backlog as well as the documentation necessary for an easy handover. As long as my backlog is implemented, the implemented solution gives Bouvet a tool for better keeping track of its assets, and thus eases the workload on the IT department.

References

[1] Stack Overflow, 'Developer Survey 2022', Jun. 2022 [Online]. Available: https://survey.stackoverflow.co/2022/. [Accessed: Nov. 12, 2022]

[2] K. Devaki, 'Statically v. dynamically v. strongly v. weakly typed languages', *Educative: Interactive Courses for Software Developers*. [Online]. Available: https://www.educative.io/answers/statically-v-dynamically-v-strongly-v-weakly-typed-languages. [Accessed: Nov. 12, 2022]

[3] MDN Web Docs, 'SPA (Single-page application)', Sep. 21, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA. [Accessed: Oct. 30, 2022]

[4] Meta Platforms Inc., 'Glossary of React Terms', *React*. [Online]. Available: https://reactjs.org/docs/glossary.html. [Accessed: Oct. 30, 2022]

[5] Meta Platforms Inc., 'Static Type Checking', *React*. [Online]. Available: https://reactjs.org/docs/static-type-checking.html. [Accessed: Oct. 30, 2022]

[6] Microsoft, 'What is TypeScript?' [Online]. Available: https://www.typescriptlang.org/. [Accessed: Oct. 30, 2022]

[7] Google, 'Introduction', *Material Design*. [Online]. Available: https://m2.material.io/design/introduction. [Accessed: Nov. 12, 2022]

[8] MUI, 'Material UI - Overview'. [Online]. Available: https://mui.com/material-ui/getting-started/overview/. [Accessed: Nov. 12, 2022]

[9] MUI, 'Move faster with intuitive React UI tools'. [Online]. Available: https://mui.com/. [Accessed: Nov. 12, 2022]

[10] MUI, 'MUI X - Overview'. [Online]. Available: https://mui.com/x/introduction/. [Accessed: Nov. 12, 2022]

[11] Meta Platforms Inc., 'Components and Props', *React*. [Online]. Available: https://reactjs.org/docs/components-and-props.html. [Accessed: Nov. 12, 2022]

[12] T. Linsley, 'Overview, TanStack Query Docs', *TanStack*. [Online]. Available: https://tanstack.com/query/v4/docs/overview. [Accessed: Nov. 15, 2022]

[13] Axios, 'Getting Started'. [Online]. Available: https://axios-http.com/docs/intro. [Accessed: Nov. 15, 2022]

[14] MDN Web Docs, 'An overview of HTTP'. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [Accessed: Dec. 04, 2022]

[15] MDN Web Docs, 'HTTPS'. [Online]. Available: https://developer.mozilla.org. [Accessed: Dec. 04, 2022]

[16] S. Chauhan, 'Difference between ASP.NET MVC and ASP.NET Web API', *DotNetTricks*, Sep. 04, 2022. [Online]. Available:

https://www.dotnettricks.com/learn/webapi/difference-between-aspnet-mvc-and-aspnet-web-api. [Accessed: Oct. 30, 2022]

[17] Microsoft, 'ASP.NET Web APIs', *Microsoft*, Oct. 26, 2022. [Online]. Available: https://dotnet.microsoft.com/en-us/apps/aspnet/apis. [Accessed: Oct. 30, 2022]

[18] Red Hat, Inc., 'What is a REST API?', May 08, 2020. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Accessed: Oct. 30, 2022]

[19] Codecademy, 'What is REST?' [Online]. Available: https://www.codecademy.com/article/what-is-rest. [Accessed: Dec. 06, 2022]

[20] Microsoft, 'Language-Integrated Query (LINQ) (C#)', Oct. 13, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. [Accessed: Oct. 30, 2022]

[21] Microsoft, 'Entity Framework Core', May 25, 2021. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/. [Accessed: Oct. 30, 2022]

[22] Microsoft, 'Databases', Nov. 19, 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/databases/databases. [Accessed: Dec. 14, 2022]

[23] P. Loshin and J. Sirkin, 'Structured Query Language (SQL)', *TechTarget*, Feb. 2022. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/SQL. [Accessed: Dec. 13, 2022]

[24] J. L. Harrington, Ed., *Relational Database Design Clearly Explained*, Second Edition. San Diego: Academic Press, 2002 [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558608207500003

[25] V. Vidhya, *Database management systems*. Oxford, England: Alpha Science International Ltd., 2016.

[26] Microsoft, 'What is Azure Active Directory?', Sep. 15, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis. [Accessed: Oct. 22, 2022]

[27] Red Hat, Inc., 'What is containerization?', Apr. 08, 2021. [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization. [Accessed: Oct. 23, 2022]

[28] Docker Inc., 'Docker overview', *Docker Documentation*, Oct. 28, 2022. [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed: Oct. 30, 2022]

[29] Microsoft, 'Kubernetes vs Docker', *Azure Microsoft*. [Online]. Available: https://azure.microsoft.com/en-us/solutions/kubernetes-vs-docker/. [Accessed: Oct. 30, 2022]

[30] B. Vanderjack, *The Agile Edge: Managing Projects Effectively Using Agile Scrum*, First Edition. New York: Business Expert Press, 2015.

[31] ‘IEEE Standard Glossary of Software Engineering Terminology’, *IEEE Std 61012-1990*, pp. 1–84, 1990, doi: 10.1109/IEEESTD.1990.101064.

[32] I. Montiel, ‘Low Coupling, High Cohesion’, *Clarity Hub*, Sep. 17, 2018. [Online]. Available: https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6. [Accessed: Nov. 25, 2022]

[33] G. Wright, ‘Module’, *TechTarget*. [Online]. Available: https://www.techtarget.com/whatis/definition/module. [Accessed: Nov. 26, 2022]

[34] C. Giridhar, *Learning Python Design Patterns - Second Edition: Leverage the Power of Python Design Patterns to Solve Real-World Problems in Software Architecture and Design*. Birmingham: Packt Publishing Ltd., 2016.

[35] K. Singh, A. Ianculescu, and L.-P. Torje, *Design Patterns and Best Practices in Java: A Comprehensive Guide to Building Smart and Reusable Code in Java*. Birmingham: Packt Publishing Ltd., 2018.

[36] S. Wlaschin, *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Sebastopol: The Pragmatic Programmers, LLC, 2018.

[37] P. Louth, ‘C# Functional Programming Language Extensions’. Oct. 30, 2022 [Online]. Available: https://github.com/louthy/language-ext. [Accessed: Oct. 30, 2022]

[38] Microsoft, ‘Functional programming vs. imperative programming - LINQ to XML’, Sep. 15, 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming. [Accessed: Oct. 30, 2022]

[39] R. C. Martin, ‘Design Principles and Design Patterns’, *www.objectmentor.com*, p. 34, Jan. 2000.

[40] B. Gantulga, N. Munkhtsetseg, D. Garmaa, and S. Batbayar, ‘Using Five Principles of Object-Oriented Design in the Transmission Network Management Information’, in *Advances in Intelligent Information Hiding and Multimedia Signal Processing*, Singapore, 2020, pp. 325–333.

[41] H. Singh and S. I. Hassan, ‘Effect of solid design principles on quality of software: An empirical assessment’, *Int. J. Sci. Eng. Res.*, vol. 6, no. 4, 2015.

[42] A. Patel, ‘Use MediatR in ASP.NET or ASP.NET Core’, *.NET Hub*, Aug. 03, 2021. [Online]. Available: https://medium.com/dotnet-hub/use-mediatr-in-asp-net-or-asp-net-core-cqrs-and-mediator-in-dotnet-how-to-use-mediatr-cqrs-aspnetcore-5076e2f2880c. [Accessed: Nov. 26, 2022]

[43] Microsoft, ‘Dependency injection in .NET’, Nov. 18, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. [Accessed: Nov. 26, 2022]

[44] T. Janssen, ‘Design Patterns Explained – Dependency Injection with Code Examples’, *Stackify*, Jun. 19, 2018. [Online]. Available: https://stackify.com/dependency-injection/. [Accessed: Nov. 26, 2022]

[45] IBM Cloud Education, ‘What is Three-Tier Architecture’, *IBM*, Sep. 16, 2021. [Online]. Available: https://www.ibm.com/cloud/learn/three-tier-architecture. [Accessed: Nov. 03, 2022]

[46] P. Ndemo, ‘2 and 3 Tier Architecture’, *Medium*, Jun. 16, 2020. [Online]. Available: https://medium.com/@paulndemo/2-and-3-tier-architecture-4a473e5ced3d. [Accessed: Nov. 15, 2022]

[47] MDN Web Docs, ‘Introduction to client-side frameworks’, Oct. 28, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction. [Accessed: Nov. 15, 2022]

[48] Meta Platforms Inc., ‘Introducing JSX’, *React*. [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed: Nov. 15, 2022]

[49] H. Dhaduk, ‘Angular vs React: Which JS Framework your Project Requires?’, *Simform - Product Engineering Company*, Feb. 16, 2021. [Online]. Available: https://www.simform.com/blog/angular-vs-react/. [Accessed: Dec. 06, 2022]

[50] FreeCodeCamp, ‘Interpreted vs Compiled Programming Languages: What’s the Difference?’, Jan. 10, 2020. [Online]. Available: https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/. [Accessed: Dec. 09, 2022]

[51] GeeksforGeeks, ‘Difference between Compiled and Interpreted Language’, Mar. 26, 2020. [Online]. Available: https://www.geeksforgeeks.org/difference-between-compiled-and-interpreted-language/. [Accessed: Dec. 09, 2022]

[52] Ambitious Systems, ‘Relationships decide the best data model fit’, Feb. 16, 2021. [Online]. Available: https://ambitious.systems/relationships-influence-data-model-fit. [Accessed: Nov. 16, 2022]

[53] P. Xavier, ‘Data Models Revealed’, *Medium*, Jun. 11, 2022. [Online]. Available: https://towardsdatascience.com/data-models-revealed-how-to-choose-the-right-model-647f19469b89. [Accessed: Nov. 16, 2022]

[54] R. Grischenko, ‘Data Modeling Techniques Explained’, *Mighty Digital*, Oct. 15, 2021. [Online]. Available: https://www.mighty.digital/blog/data-modeling-techniques-explained. [Accessed: Nov. 16, 2022]

[55] Integrant, ‘When to Use SQL vs. NoSQL’. [Online]. Available: https://techblog.integrant.com/when-to-use-sql-vs-nosql. [Accessed: Nov. 16, 2022]

[56] K. Clark and C. Jackson, ‘The true benefits of moving to containers’, *IBM Developer*, Sep. 10, 2020. [Online]. Available: https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1/. [Accessed: Nov. 17, 2022]

[57] I. Buchanan, ‘Containers vs Virtual Machines’, *Atlassian*. [Online]. Available: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms. [Accessed: Nov. 17, 2022]

[58] Aqua Security, ‘Container Advantages: 7 Reasons to Adopt a Containerized Architecture’, *Cloud Native Wiki by Aqua*. [Online]. Available: https://www.aquasec.com/cloud-native-academy/docker-container/container-advantages/. [Accessed: Nov. 17, 2022]

[59] IBM Cloud Education, ‘Containerization’, *IBM*, May 15, 2019. [Online]. Available: https://www.ibm.com/in-en/cloud/learn/containerization. [Accessed: Nov. 17, 2022]

[60] Future Techno India, ‘Deployments on Containers’, *Medium*, Aug. 09, 2021. [Online]. Available: https://futuretechnoindia.medium.com/deployments-on-containers-aa2353b15cff. [Accessed: Nov. 17, 2022]

[61] Aqua Security, ‘Docker Containers vs. Virtual Machines’, *Cloud Native Wiki by Aqua*. [Online]. Available: https://www.aquasec.com/cloud-native-academy/docker-container/docker-containers-vs-virtual-machines/. [Accessed: Nov. 17, 2022]

[62] N. Janetakis, ‘Virtual Machines vs Docker Containers: a Comparison’, *CircleCi*, May 16, 2018. [Online]. Available: https://circleci.com/blog/virtual-machines-vs-docker-containers-a-comparison/. [Accessed: Nov. 17, 2022]

[63] S. Miteva, ‘The Concept of Domain-Driven Design Explained’, *Microtica*, Jul. 03, 2020. [Online]. Available: https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f. [Accessed: Nov. 25, 2022]

[64] Microsoft, ‘Assemblies in .NET’, Sep. 08, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/assembly/. [Accessed: Nov. 26, 2022]

[65] R. Anderson and K. Larkin, ‘Enable Cross-Origin Requests (CORS) in ASP.NET Core’, *Microsoft*, Jun. 03, 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/cors. [Accessed: Nov. 27, 2022]

[66] J. Miskovic, ‘Class vs Record: Difference between class and record type in C#’, *josipmisko.com*, Oct. 20, 2022. [Online]. Available: https://josipmisko.com/posts/c-sharp-class-vs-record. [Accessed: Nov. 27, 2022]

[67] Microsoft, ‘Records (C# reference)’, Oct. 07, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record. [Accessed: Nov. 27, 2022]

[68] J. Trivedi, ‘Association, Aggregation and Composition’, *C# Corner*, Jan. 12, 2022. [Online]. Available: https://www.c-sharpcorner.com/uploadfile/ff2f08/association-aggregation-and-composition/. [Accessed: Nov. 27, 2022]

[69] A. Saini, ‘C# | Predicate Delegate’, *GeeksforGeeks*, Apr. 04, 2019. [Online]. Available: https://www.geeksforgeeks.org/c-sharp-predicate-delegate/. [Accessed: Nov. 28, 2022]

[70] Microsoft, ‘ASP.NET Core web API documentation with Swagger / OpenAPI’, Nov. 10, 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger. [Accessed: Nov. 28, 2022]

[71] T. Gemayel, ‘How to wireframe’, *Figma*, Aug. 12, 2019. [Online]. Available: https://www.figma.com/blog/how-to-wireframe/. [Accessed: Nov. 29, 2022]

[72] ‘Glossary of React Terms – React’. [Online]. Available: https://reactjs.org/docs/glossary.html. [Accessed: Nov. 29, 2022]

[73] M. Abdelmogoud, ‘How the browsers understand JavaScript’, *Medium*, Sep. 27, 2019. [Online]. Available: https://medium.com/@mustafa.abdelmogoud/how-the-browsers-understand-javascript-d9699dced89b. [Accessed: Nov. 29, 2022]

[74] npm, Inc., ‘About npm’, *npm*. [Online]. Available: https://www.npmjs.com/about. [Accessed: Nov. 29, 2022]

[75] MDN Web Docs, ‘Destructuring assignment’, Oct. 31, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment. [Accessed: Nov. 30, 2022]

[76] L. Mwila, ‘Build a Multi-Container Docker Application with Docker Compose with a React, Node, and Postgres App’, *Medium*, Jul. 30, 2020. [Online]. Available: https://levelup.gitconnected.com/build-a-multi-container-application-with-docker-compose-460f6199ef3c. [Accessed: Dec. 04, 2022]

[77] Docker Inc., ‘Glossary’, *Docker Documentation*, Dec. 01, 2022. [Online]. Available: https://docs.docker.com/glossary/. [Accessed: Dec. 04, 2022]

[78] Docker Inc., ‘Networking in Compose’, *Docker Documentation*, Dec. 01, 2022. [Online]. Available: https://docs.docker.com/compose/networking/. [Accessed: Dec. 04, 2022]

[79] Microsoft, ‘Hosting ASP.NET Core image in container using docker compose with HTTPS’, Nov. 27, 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/docker-compose-https. [Accessed: Dec. 04, 2022]

[80] Docker Inc., ‘Docker Engine overview’, *Docker Documentation*, Dec. 01, 2022. [Online]. Available: https://docs.docker.com/engine/. [Accessed: Dec. 04, 2022]

[81] Snyk Ltd., ‘Snyk Open Source Advisor’, *Snyk Advisor*. [Online]. Available: https://snyk.io/advisor. [Accessed: Dec. 03, 2022]