



OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Project Phoenix

Bachelor Thesis 2020

Department of Computer Science

Earl John Torculas Laguardia

s315611

Information technology

Tuan Minh Nguyen

s325862

Computer engineer



Institutt for Informasjonsteknologi

Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo

Besøksadresse: Holbergs plass, Oslo

PROSJEKT NR.

2

TILGJENGELIGHET

Åpen

Telefon: 22 45 32 00

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL Ambita – Project Phoenix	DATO 05/25/2020
	ANTALL SIDER / BILAG 107
PROSJEKTDeltakere Earl John Torculas Laguardia Tuan Minh Nguyen	INTERN VEILEDER Sidney Pontes-Filho
OPPDRAAGSGIVER Ambita AS	KONTAKTPERSON Dennis Hernando Knudsen

SAMMENDRAG

Teknologi vokser raskt, og verktøy som brukes til programmering må oppdateres eller erstattes. Utdaterte nettbutikker i dag pleier derfor å flytte over til et mer moderne rammeverk.

Vi skal jobbe med å flytte produktdefineringen fra dagens infoland over til et mer moderne rammeverk, og en plattform som er mer rustet for fremtiden. Det finnes allerede en superrask support-applikasjon der den hører naturlig hjemme. Gruppen skal dermed bli presentert med en liste over allerede vedtatte features, som vi skal implementere over til det nye support-systemet.

Dette er en gammel applikasjon, brukerne av denne applikasjonen er sulteforet på enklere flyt, og smartere måter å få nye produkter på luften.

3 STIKKORD

Webapplikasjon

Wizard

Overføring

Project Phoenix

Abstract

Technology is continuously developing. Both software and hardware need to be updated regularly to keep up with this rapid growth. Today there are multiple programming languages and technologies that are no longer supported or are no longer being updated. This removes the ability to update some websites or other software that relied on them.

The main objective of this project is to move PKAdmin which is a part of Ambita's current Infoland web application to a more futureproof platform. Design and implementation were done by following user stories and requests from the product owner. The outcome of this project was a newly designed modal that could edit and manage products listed in Infoland's web application.

Acknowledgement

We would like to thank our supervisor Sidney Pontes-Filho for helping us write this bachelor thesis. The advice he gave us had a huge positive impact on this report. His support and guidance throughout this semester are something our group is very grateful for.

We would also like to thank everyone at Ambita AS, especially Dennis Hernando Knudsen for sticking with us until the very end. Their advice and support helped us a lot when working on the features for this project.

Summary

Technology is always changing. Outdated online web applications tend to move over to a new framework that is more suited for the future. Infoland is an old application, and its users wants a faster and smoother experience when managing its products. In this project, we worked on moving the PKAdmin of Ambita's Infoland over to a more robust and modern framework.

This report goes through the development process of Project Phoenix. It covers the technical theories, tools and technologies we used along the way, as well as the different method we used such as scrum.

We designed and implemented the features of Infoland according to the user stories provided to us by the product owner. We ended up with a very reliable, accessible and optimized modal that is also easy to use.

Content

Abstract	3
Acknowledgement	4
Summary	5
List of figures	10
List of Tables.....	13
1 Introduction.....	14
1.1 Background	14
1.2 About Ambita AS	14
1.3 The Project	15
2 Technical Theory	16
2.1 Programming Languages	16
2.1.1 HTML	16
2.1.2 CSS.....	16
2.1.3 SCSS	16
2.1.4 JavaScript.....	17
2.1.5 TypeScript.....	17
2.1.6 Riot.js.....	17
2.1.7 Java.....	17
2.1.8 Play 2	18
2.2 Database Technology	18
2.2.1 PostgreSQL	18
2.2.2 Amazon Web Services	18
2.2.3 AWS Aurora	19
2.3 Integrated Development Environment	19
2.3.1 IntelliJ IDEA.....	19
2.3.2 Visual Studio Code	20
2.4 Version Control.....	20
2.4.1 Git.....	20
2.4.2 Bitbucket	21
2.4.3 Jira Software.....	22
2.4.4 Confluence	23
2.5 Scrum	23
2.5.1 Sprint	23
2.5.2 Sprint Planning	23

2.5.3 Daily Scrum.....	24
2.5.4 Sprint Review.....	24
3 Objectives & Requirements.....	24
3.1 Objectives.....	24
3.2 Rules.....	25
3.3 Requirements.....	26
3.4 User Stories.....	26
3.4.1 The modal.....	26
3.4.2 The wizard.....	28
3.4.3 The identity.....	29
3.4.4 The description.....	30
3.4.5 The delivery.....	30
3.4.6 The relations.....	31
3.4.7 The ordering.....	32
3.4.8 The executives.....	33
3.4.9 The price.....	34
3.4.10 The parameters.....	35
3.4.11 The authorization.....	36
3.4.12 The approval.....	37
4 Method & Material.....	39
4.1 Project Group.....	39
4.2 Project Organization.....	39
4.2.1 Task giver.....	39
4.2.2 Project advisor.....	39
4.3 Development tools.....	39
4.3.1 Atlassian.....	39
4.3.2 Git.....	40
4.3.3 Bitbucket.....	40
4.3.4 Jira Software.....	41
4.3.5 IntelliJ IDEA.....	41
4.3.6 Visual Studio Code.....	41
4.3.7 Slack.....	41
4.3.8 Google Docs.....	42
4.3.9 Daily documentation.....	42
4.4 Implementation.....	42
4.4.1 Implementation requirements.....	42

4.4.2 Design structure	42
4.4.3 Used technologies	47
4.4.4 Used programming languages	47
5 Results	48
5.1 Functionality & Design	48
5.1.1 The Modal	48
5.1.2 The Wizard	49
5.2 Frontend Development	50
5.2.1 The modal.....	50
5.2.2 The wizard	51
5.2.3 Navigation bar	54
5.2.4 The identity	55
5.2.5 The description.....	58
5.2.6 The delivery	60
5.2.7 The relations.....	62
5.2.8 The ordering.....	65
5.2.9 The executives.....	67
5.2.10 The price.....	68
5.2.11 The parameters	69
5.2.12 The authorizations.....	70
5.2.13 The approval.....	70
5.3 Backend Development	71
5.3.1 API	71
5.3.2 API testing	80
5.3.3 Identity functionality	88
5.3.4 Description functionality	92
5.3.5 Delivery functionality.....	94
5.3.6 Relations functionality.....	94
5.3.7 Ordering functionality	99
5.3.8 Price functionality.....	99
5.3.9 Parameters functionality	99
5.3.10 Authorizations functionality	99
5.3.11 Approval functionality	99
5.4 Final results	100
6 Discussion	101
6.1 Design Evaluation	101

6.2 Development process.....	102
7 Conclusion	104
8 References.....	105

List of figures

Figure 1 The Modal description.....	26
Figure 2 New Product button	27
Figure 3 Edit product button	27
Figure 4 Blank modal	28
Figure 5 The Wizard description.....	28
Figure 6 The Identity description	29
Figure 7 Presentation of the identity feature.....	29
Figure 8 The description.....	30
Figure 9 Presentation of the description feature	30
Figure 10 The delivery description	31
Figure 11 Representation of the delivery feature	31
Figure 12 The relations description.....	32
Figure 13 Representation of the relations feature	32
Figure 14 The ordering description	33
Figure 15 Representation of the ordering feature	33
Figure 16 The executives description.....	34
Figure 17 Representation of the executives feature	34
Figure 18 The price description.....	35
Figure 19 Representation of the price feature.....	35
Figure 20 The parameters description	36
Figure 21 Representation of the parameters feature	36
Figure 22 The authorizations description.....	37
Figure 23 Representations of the authorizations feature	37
Figure 24 The approval description.....	38
Figure 25 Picture of highlights.....	40
Figure 26 Picture of our project folder.....	43
Figure 27 Picture of files in each folder.....	44
Figure 28 View.tag code structure	45
Figure 29 styles.scss code structure	45
Figure 30 controllers.ts code structure	46
Figure 31 Laptop spec	47
Figure 32 Modal feature.....	48
Figure 33 The identity feature.....	49
Figure 34 view.tag of edit-products modal	50
Figure 35 The modal scss	51
Figure 36 styles.scss of navigation bar in modal	52
Figure 37 styles.scss of the wizard	53
Figure 38 Navigation bar	54
Figure 39 view.tag of the navigation bar.....	55
Figure 40 styles.scss of the navigation bar	55
Figure 41 Final result of the wizard feature	56
Figure 42 Final code for the identity	57
Figure 43 styles.scss of the identity feature	58
Figure 44 Final result of the description feature	59
Figure 45 Final code for the description feature	59

Figure 46 styles.scss for the description feature	60
Figure 47 Final result of the delivery feature	61
Figure 48 Final code for the delivery feature	61
Figure 49 Example code 1	62
Figure 50 styles.scss of the delivery	62
Figure 51 Final result of the relations feature	63
Figure 52 Final code of the relations feature part 1	63
Figure 53 Final code of the relations feature part 2	64
Figure 54 styles.scss of the relations part 1	64
Figure 55 styles.scss of the relations part 2	64
Figure 56 styles.scss of the relations part 3	65
Figure 57 Final result of the ordering wizard	66
Figure 58 Final code of the delivery feature	66
Figure 59 Example code 2	67
Figure 60 styles.scss of the delivery	67
Figure 61 Final result of the executives feautre	67
Figure 62 Final code of the executive feautre	68
Figure 63 Final result of the parameters feature	69
Figure 64 Final code of the parameters feature	69
Figure 65 styles.scss of the parameter	70
Figure 66 Figure of how API works	71
Figure 67 Figure of /api/ProductType.java	72
Figure 68 Figure of product_type database table	73
Figure 69 api/ProductTypes.java	73
Figure 70 LogisticsProductTypeMapper.java	74
Figure 71 ProductService.java	75
Figure 72 LogisticsProductTypeController.java	76
Figure 73 models/ProductType.java part 1	77
Figure 74 models/ProductType.java part 2	78
Figure 75 /mapping/MetaMapper.java	79
Figure 76 Logistics.routes part 1	79
Figure 77 Logistics.routes part2	80
Figure 78 Product-data.yaml	81
Figure 79 1.sql part 1	81
Figure 80 1.sql part 2	81
Figure 81 Util.java	82
Figure 82 LogisticsProductTypeServiceTest.java part 1	83
Figure 83 LogisticsProductTypeServiceTest.java part 2	84
Figure 84 LogisticsProductTypeControllerTest.java	85
Figure 85 url/productCatalog/types.ts	86
Figure 86 api/productCatalog/logistics/Types.ts	87
Figure 87 app/dataservices/types.ts	88
Figure 88 The identity controller.ts code	89
Figure 89 init function in the identity controller class	89
Figure 90 Example code 3	90
Figure 91 checkbox states	90
Figure 92 The tag.hasTradeType function	91

Figure 93 Ccalling hasTradeType(tradeType) in view.tag	91
Figure 94 Checked and unchecked checkboxes	91
Figure 95 The description controller.ts code	92
Figure 96 init function in the description controller class	93
Figure 97 Mounting tag	93
Figure 98 Unmounting the tags	93
Figure 99 WizardContentRelationsTag	94
Figure 100 Presentation of the relation boxes	95
Figure 101 init function of the relations class part 1	95
Figure 102 init function of the relations class part 1	95
Figure 103 Value check	96
Figure 104 The relations: add function	96
Figure 105 The relations: remove function	97
Figure 106 Google chrome console results	97
Figure 107 The relations: counter function	98
Figure 108 The relations: check all and remove function	98

List of Tables

Table 1 Different HTTP methods	77
Table 2 Final results of features	100

1 Introduction

1.1 Background

Technology is growing fast and tools used for programming need to be updated or replaced. Outdated online web applications tend to move over to a more modern framework. Our group will work on moving the Productcatalog Admin (PKAdmin) of Ambita's Infoland over to a more robust and modern framework, and a platform that is more suited for the future. A fast support-application already exists where this new application can be implemented.

PKAdmin is an old application, and its users want a faster and smoother experience when managing its products. It uses JavaServer Faces (JSF) and Enterprise JavaBeans (EJB), which is no longer supported or updated. We will design and implement the features according to the user stories. The product owner has already created a list of features that is to be implemented over to the new support-application.

1.2 About Ambita AS

Previously named "Norsk Eiendomsinformasjon", Ambita is a technology company that focuses on delivering digitalized property market. The company has a firm vision of creating a more effective and open real estate market in Norway, hence the name "Ambita", which means ambition.

Thousands of customers use their services to view and receive digitalized information about a property or a construction project. With an optimized and user-friendly web application, they provide solutions and detailed information to their customers. The company ensures a good connection with their users, while also maintaining a tight management to secure an easier and more effective way of digitalizing prop tech. Property technology is the use of information technology in property market. It is used to help consumers and businesses to buy, sell and manage properties in a more effective way. (Tomagruppen, 2020)

1.3 The Project

We named this project “Project Phoenix”, named after the bird of resurrection, will focus on redesigning and implementing the old PKAdmin application over to a more modern system. We will use scrum as our development framework, and switch from JavaServer Faces (JSF) and Enterprise JavaBeans(EJB) over to Riot.js and Play 2 Framework.

The product owner has created a list of features from PKAdmin that is to be implemented over to the new version of Infoland. These features are called “user stories” and show a detailed image and description of how the feature are going to be implemented. Most of these features already exists, and just needs to be reimplemented by using Riot.js and Play 2.

Our group will be working alongside their development team, but not on the same project. We will mainly be focusing on the user stories while maintaining a tight communication during the development process. To ensure a good communication between both parties, our group will join their daily scrum and sprint meetings. We will keep each other updated during the development process and stay in contact by using development tools such as “Slack”.

We will be using multiple development tools and services used by Ambita to get a better understanding on how large companies operate. We will also familiarize ourselves with these tools to ensure an effective and efficient development process.

2 Technical Theory

2.1 Programming Languages

2.1.1 HTML

Hypertext Markup Language (HTML) is the main skeleton of a website (w3, 2020). It allows developers to display forms, modals, inputs, buttons, images and other elements. The skeleton is structured to have a “head” and a “body”, both having their own respective roles on the website. The head often includes the title and link to a stylesheet, as well as the background or a navigation menu depending on the website. The body has all the functionalities and elements. These are implemented through the use of the tags syntax. Each element can be customized to have their own style, identity and class, which is inserted inside the tags. The most common way to style the elements is using Cascading Style Sheets (CSS).

2.1.2 CSS

If HTML is the skeleton, then Cascading Style Sheets (CSS) is the skin of the skeleton. It is used to style and animate the elements in an HTML file (w3, 2020). This includes styling its size, position, color, text font, image and even animation. They are implemented through connecting the tags inside the HTML-element with the id or class of the CSS. The styles are often structured and identified by using either a dot (.) for id, or a hashtag (#) for class. The id is mostly used on a single HTML-element, while the class is used on the larger elements such as a container.

2.1.3 SCSS

Sassy CSS (SCSS) is often described as CSS with superpowers (sass-lang, 2020). It is a powerful CSS extension language that allows for a more easy and flexible styling experience. It is compatible with every CSS version and includes more features and abilities to give the styling experience more fluidity.

2.1.4 JavaScript

JavaScript is a high-level programming language used to create functions to make websites more dynamic and interactable (developer.mozilla, 2020). It is an easy-to-understand, yet powerful programming language and is one of the most recommended languages to use when it comes to web programming (Veeraraghavan, 2020).

2.1.5 TypeScript

TypeScript is a powerful open-source programming language used to develop JavaScript applications. It can be used to develop both the client-side and the server-side part of the application. TypeScript is a superset of JavaScript, meaning that a code written in JavaScript can also work in a TypeScript program (tutorialspoint, 2020). Typescript is compiled to Javascript code.

2.1.6 Riot.js

Riot.js is a free open-source component-based UI library (Guarini, riot.js, 2020). Its syntax is a combination of HTML layout and JavaScript logic. It uses tags to identify each element just like how HTML and CSS are used together. A Riot component is structured with a named tag, and the scripts are implemented inside that tag. Riot.js supports all modern browsers, but browsers such as Internet Explorer 11 requires an older installation of Riot

2.1.7 Java

Java is an object-oriented programming language used almost everywhere. It is designed to run on all platforms that supports Java (edureka, 2020). The syntax is similar to C and C++ but does not require any knowledge of memory allocation since it does it automatically compared to the C language. Java is also one of the most popular programming languages used by developers and is also used on web applications on both the client and server side (Veeraraghavan, 2020).

2.1.8 Play 2

Play 2 is an open-source web application framework that uses the Model-View-Controller (MVC) structure. It is written in Scala and is compiled using the Java Virtual Machine Bytecode (JVM Bytecode) (Play, 2020). It is used by developers to optimize productivity when developing web applications.

2.2 Database Technology

2.2.1 PostgreSQL

PostgreSQL is a powerful open-source relational database management system (The PostgreSQL Global Development Group, 2020). It is known for its extensibility, reliability and optimized performance. PostgreSQL comes with many features that helps develop applications aimed at protection and managing data. It is also a very flexible system, allowing developers to use different programming languages without having to recompile the database.

2.2.2 Amazon Web Services

Amazon Web Services (AWS) is a well-known cloud computing platform that provides excellent services and APIs to both individuals and large companies (Amazon web services, 2020). It comes with multiple tools that helps developers maintain, manage and administrate web applications or other projects.

AWS uses a “pay-as-you-go” business model, meaning that the services used is only limited by how much budget an individual or a company is willing to spend. These services include, but are not limited to:

- Computing
- Storage
- Database
- Analytics
- Application services
- Deployment
- Management

- Mobile
- Developer tools

The AWS technology is maintained and managed by Amazon. They are implemented at server farms all around the world. A server farm is a collection of computer servers that often consists of thousands of these. In case of emergency, these server farms often have a backup servers that will automatically take over the primary server.

2.2.3 AWS Aurora

Amazon Aurora (AWS Aurora) is a relational database management service developed by Amazon. It is compatible with both PostgreSQL and MySQL (Amazon web services, 2020). Aurora aims to improve performance, flexibility and reliability while also having automatic allocation of database storage. It also provides performance metrics and fast database cloning for developers.

2.3 Integrated Development Environment

2.3.1 IntelliJ IDEA

IntelliJ IDEA is an integrated development environment (IDE) developed and maintained by JetBrains (tutorialspoint, 2020). It is written in Java and Kotlin, and is available in multiple operative systems including Microsoft, MacOS and Linux. IntelliJ IDEA offers multiple versions to suit individual developers or large companies.

The IDE offers powerful tools that aims to improve developers coding experience. Like other IDE's, IntelliJ IDEA provides coding assistance, built in tools integration, plugin systems and supports multiple programming languages. Java and Kotlin are the standard programming languages, while the ultimate edition offering more. It also offers technologies and frameworks for both the community and the ultimate edition (jetbrains, 2020).

IntelliJ IDEA also offers an in-built version control that supports Git, GitHub, Azure DevOps and more, giving developers more options on how they want to manage their project.

Installing plugins also allows the IDE to be compatible with other version control services like Bitbucket (jetbrains, 2020).

2.3.2 Visual Studio Code

Visual Studio Code (VSC) is a source-code editor developed by Microsoft. It is available on MacOS, Windows and Linux (Microsoft, 2020). Despite having features that mimics an IDE, Visual Studio Code is not an integrated development environment. Instead it focuses on being highly customizable and optimized with the installation of extensions to fit the developer's needs.

It comes with embedded features such as debugging tool, syntax highlighting, code completion and more. With the option to install extensions, Visual Studio Code can become a very powerful tool, giving its users the freedom to develop and code however they want. It is basically an IDE creator tool that allows developers to create their own development environment akin to an actual IDE (Microsoft, 2020).

Unlike an IDE, which is often limited to certain programming languages, Visual Studio Code is not limited and can be customized to fit any language. This is because VSC is an editor and can be optimized and built in a way that allows the user to code in different languages. Despite being highly customizable, Visual Studio Code does lack some features that is only offered using an IDE. However, this does not limit VSC as it can be used in tandem with an IDE. This is because unlike an actual IDE, Visual Studio Code does not structure projects in a system like IntelliJ IDEA. Instead it allows the user to open multiple directories in a workspace. This allows developers to open projects created in an IDE and still develop it through the use of Visual Studio Code.

2.4 Version Control

2.4.1 Git

Git is a powerful version control system that tracks the development of a source code. It is used by programmers to coordinate the changes on a project during development but is not

limited to source codes only. Git can also be used to track changes on any type of files including word documents, multimedia and texts.

Version controlling a project is done using the commit, push and pull commands. Commit is a command which allows you to save your code either in a local branch or a remote branch, it records changes in the repository. Pushing sends the recent commit history from your local repository to Github or other version control applications. Pulling is when you download any changes from the GitHub repository, to merge with your local repository (sparkfun, 2020). Each project can also “branch” out of the master file, allowing developers to work on their own branch of the project. This can be done by using command lines such as “checkout”. The developer can then ask for a “pull request”, which basically asks a senior programmer to view the branched code to be merged with the master file.

Each commit are stored in a directory called “repository”. Here, the user can view and manage the changes made on the project. Git changes and commits can be tracked through version control services. The most common is GitHub, which is used to track projects both private and public. Larger companies who prioritizes privacy and security, will most likely use other services to track changes and commits.

2.4.2 Bitbucket

Bitbucket is a version control repository service like GitHub. It is owned by Atlassian and has a lot of similarities to GitHub in terms of functionality (Atlassian, 2020). It is clearer and appealing in terms of visual presentation and is favored by large companies for its easy-to-use management systems.

Bitbucket has many features that helps developers manage a project. It keeps tracks of commits and changes made on the project, while also being simple and easy to understand (Atlassian, 2020). Bitbucket also offers a clean visual presentation over branches and pull requests. It also offers an in-build Kanban board to help manage and develop features for a project.

Most large companies prefer to use Bitbucket version control instead of GitHub. This is because Atlassian offers multiple services for managing large projects. The most common services used together with bitbucket are Jira Software and Confluence. This helps with coordination, management, deployment, and communication on large companies. It is also used by individuals or small groups working on larger projects.

2.4.3 Jira Software

Jira Software is an issue tracking and project management product owned by Atlassian (Atlassian, 2020). It is most used together with Bitbucket to view, manage and develop projects. Jira offers multiple features and tools to help developers coordinate and communicate when solving an issue or managing a project.

Jira has an in-built Kanban board, making it a versatile tool for software developers to keep track of features to be implemented. A Kanban board is a tool for project management and is often used by larger groups of developers when working on a large project (Atlassian, 2020). The board is divided into columns, each representing a stage of its development process. The most common naming for each column is: “to-do”, “under development”, “testing” and “done”. The columns can contain features that is to be implemented or is finished depending on how the project is managed.

Jira Software also offers additional features that keeps track of project issues (Atlassian, 2020). These includes, but is not limited to:

- Filtering of issues
- Project Releases
- View history
- Report management and analysis
- Create user stories & issues
- Create a Git branch directly from Jira

It also offers Slack integration. Slack is a communication software used by developers. It is

comparable to Skype and helps when coordinating issues or managing a project on-the-go. Jira also allows developers to create pages to help visualize and write features and ideas.

2.4.4 Confluence

Confluence is a collaboration tool used to help developers collaborate, coordinate, and manage information efficiently (Atlassian, 2020). The contents are created, organized, and managed using pages, blogs and spaces (Atlassian, 2020). It is used in coordination with Jira Software by large companies to efficiently share information and manage projects.

Confluence has a robust and clean interface that allows for easy interaction. It offers its own collaboration tools that can create, edit, comment, and manage work. Confluence also has its very own text editor, allowing for an advanced and easy way to edit pages.

Confluence offers a way to manage user permissions and restrictions, making administration and user management simple to manage. The permission and restrictions can prevent or limit users from viewing certain contents and information within the Confluence page.

2.5 Scrum

2.5.1 Sprint

Scrum is an agile method for project development and has multiple stages, one of them being sprint. A sprint is a way to continuously add incremental features during a short period of time, which can range from two weeks up to a month (Scrum, 2020).

The scrum process is usually facilitated by a “scrum master”, who plans the meetings and coordinates the team. The scrum master is responsible of making sure that the development process stays on track and making sure that there are no issues during the sprint duration.

2.5.2 Sprint Planning

A sprint usually starts off with a sprint meeting led by the scrum master, and usually takes place at the start of the week. During this sprint meeting, a goal and objective will be planned for implementation during the sprint duration (scrum, 2020).

Both the goal and objective can vary depending on the current situation. This could either be a fix for an issue, or a new feature. Once the team has settled on a certain goal, the scrum master will then organize the team and set everything up for development. This is usually visualized by using a Kanban board. When everything is ready for implementation, the sprint will start.

If the goal or objective is finished before the sprint duration ends, then the team can start working on a new feature or issue right away. The scrum master will notify the team and address the current situation. This usually happens during a daily scrum meeting.

2.5.3 Daily Scrum

A daily scrum is a short meeting taken place every day. During this meeting, the team will update each other about the development process (Scrum, 2020). Each developer on the team takes turn on updating each other about their current situation, what they are currently working on and if there are any issues.

2.5.4 Sprint Review

Sprint reviews are done after a sprint has ended, usually by the end of the week. During this review, the team will go through what they have implemented, and which issues has been addressed (Scrum, 2020). In short, it is a summary of the sprint process. A large company usually has multiple teams working on a project. During a sprint review, each team will present what they have accomplished and give a quick summary of their progression.

3 Objectives & Requirements

3.1 Objectives

Our objective is to move the current product wizard from the old JSF-project ProductCatalogAdmin (PKAdmin) over to the new support application. This is done through the implementation of user stories, a list of features assigned to us by the product owner.

The current product catalog is outdated and needs a new home. Ambita's current support-application is fast and up to date with the current modern technology. Our group will be redesigning and implementing both new and old features over to the new support application.

These features each have a description and an image of how it should look and function. Most of these are already implemented on the back-end part, and just needs to be converted over to the new framework with a new design.

According to the product owner, the scope of this project is bigger than a bachelor project. It means that we work with as much as we can. The Infoland developers will take over the project and complete it. Our group will start with the simple ones first to familiarize ourselves with their system. Once we get the hang of it, we will then proceed to implement the other features for the product catalog.

3.2 Rules

Ambita is a relatively large company with a professional work environment. They have rules and standards that we must follow. During the development process, our group are required to follow these certain requirements and ruleset:

Technical requirements:

- Use the computer they have provided for us (For security reasons).
- Follow the project & code structure.
- Join daily scrum meeting (Not required but recommended).

The rules are, but is not limited to:

- Not share or leak any private information or direct source code to the public.
- Not lose the ID-card given to us by the company (Required to enter building).
- Wear pants during work.
- Follow the contract regarding anti-corruption.

There are no requirements for when or how long we will have to work on the project every week. So, we decided to set a minimum requirement for ourselves. Our group decided to meet up and work during Mondays, Tuesdays and Fridays.

3.3 Requirements

The requirements was given to us by the product owner. Our group had to implement the “user stories”, which are basically features that is to be implemented in the PKAdmin. It shows a description and an example of how it should work and function. It is not required for our group to finish all user stories given to us by the product owner. We can choose which feature to implement first and continue working on the other incrementally.

3.4 User Stories

3.4.1 The modal

Our first objective is to setup the modal, which is a dialog box window that is displayed on top the current page (w3schools, 2020). It is a simple feature and should be the main skeleton and background for the user stories. The description for this feature is as follows:

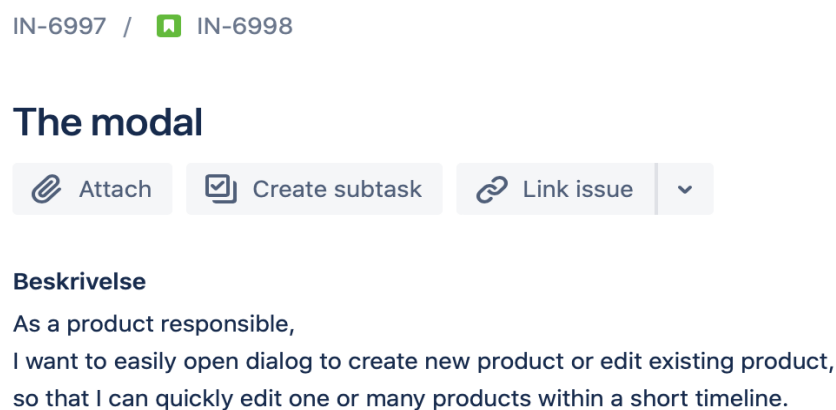
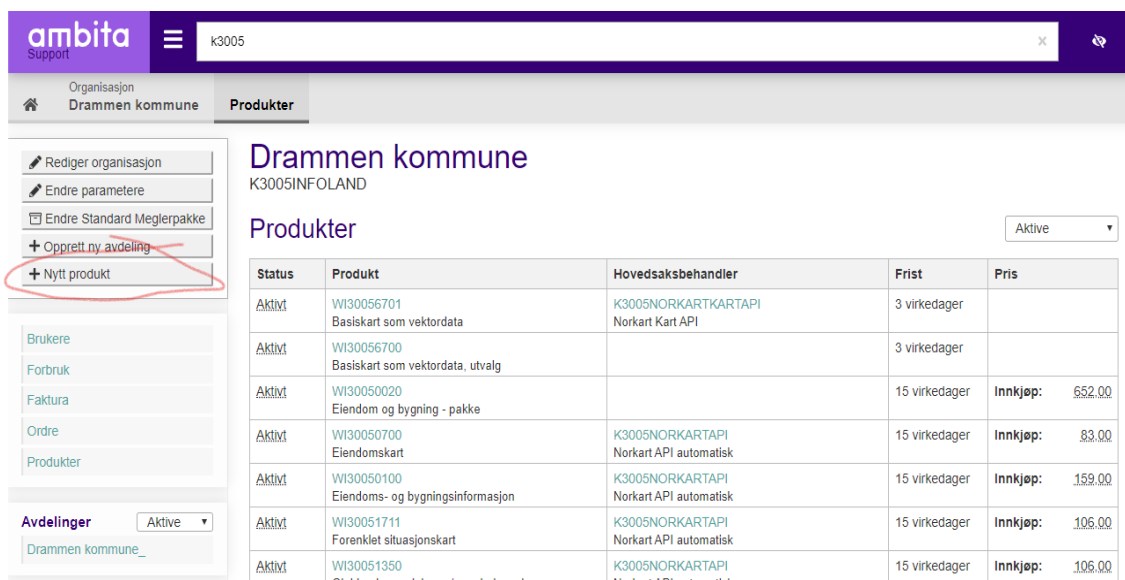


Figure 1 The Modal description

The objective of this user story is to add two new buttons in the support application, one that creates a new product and one that edits the existing products. The “edit products” button already exists and opens the old wizard for editing products in Ambita’s product-catalog. The only thing that our group needs to implement is the “create product” button, which should

also open the new modal. Here are some illustrations that shows how it should be implemented:



ambita Support

Organisasjon
Drammen kommune

Produkter

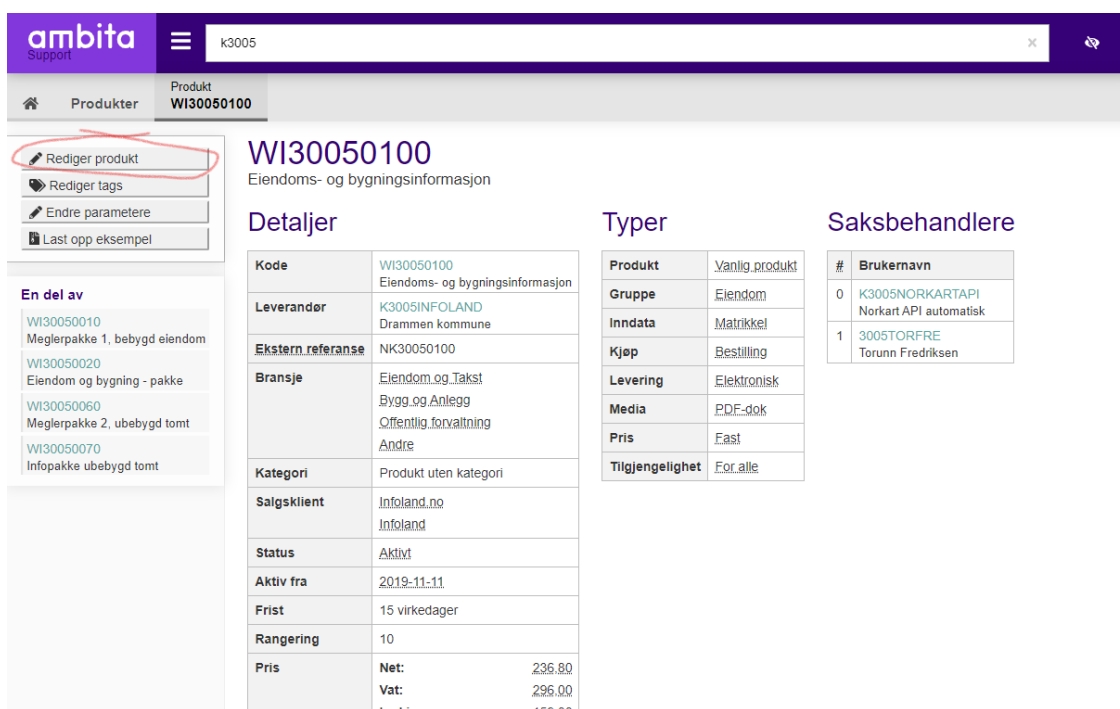
Drammen kommune
K3005INFOLAND

Produkter

Aktive

Status	Produkt	Hovedsaksbehandler	Frist	Pris
Aktivt	WI30056701 Basiskart som vektordata	K3005NORKARTKARTAPI Norkart Kart API	3 virkedager	
Aktivt	WI30056700 Basiskart som vektordata, utvalg		3 virkedager	
Aktivt	WI30050020 Eiendom og bygning - pakke		15 virkedager	Innkjøp: 652.00
Aktivt	WI30050700 Eiendomskart	K3005NORKARTAPI Norkart API automatisk	15 virkedager	Innkjøp: 83.00
Aktivt	WI30050100 Eiendoms- og bygningsinformasjon	K3005NORKARTAPI Norkart API automatisk	15 virkedager	Innkjøp: 159.00
Aktivt	WI30051711 Forenklet situasjonskart	K3005NORKARTAPI Norkart API automatisk	15 virkedager	Innkjøp: 106.00
Aktivt	WI30051350 Gjeldende arealplaner / reguleringsplaner	K3005NORKARTAPI Norkart API automatisk	15 virkedager	Innkjøp: 106.00

Figure 2 New Product button



ambita Support

Produkter

Produkt
WI30050100

WI30050100
Eiendoms- og bygningsinformasjon

Rediger produkt

Rediger tags

Endre parametere

Last opp eksempel

En del av

- WI30050010
Meglerpakke 1, bebygd eiendom
- WI30050020
Eiendom og bygning - pakke
- WI30050060
Meglerpakke 2, ubebygd tomt
- WI30050070
Infopakke ubebygd tomt

Detaljer

Kode	WI30050100 Eiendoms- og bygningsinformasjon
Leverandør	K3005INFOLAND Drammen kommune
Ekstern referanse	NK30050100
Bransje	Eiendom og Takst Bygg og Anlegg Offentlig forvaltning Andre
Kategori	Produkt uten kategori
Salgsklient	Infoland.no Infoland
Status	Aktivt
Aktiv fra	2019-11-11
Frist	15 virkedager
Rangering	10
Pris	Net: 236.80 Vat: 296.00 Innkjøp: 159.00

Typer

Produkt	Vanlig produkt
Gruppe	Eiendom
Inndata	Matrikkel
Kjøp	Bestilling
Levering	Elektronisk
Media	PDF-dok
Pris	Fast
Tilgjengelighet	For alle

Saksbehandlere

#	Brukernavn
0	K3005NORKARTAPI Norkart API automatisk
1	3005TORFRE Torunn Fredriksen

Figure 3 Edit product button

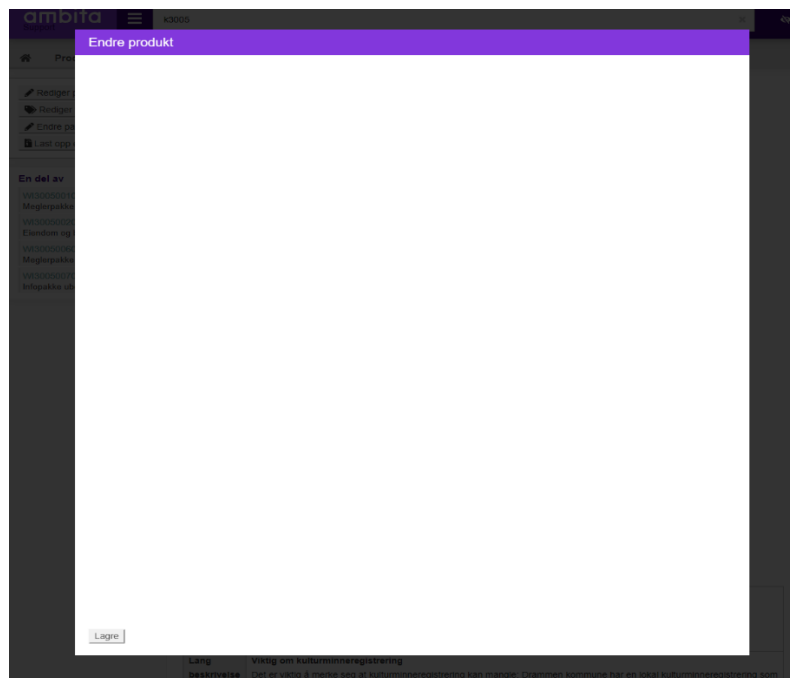


Figure 4 Blank modal

3.4.2 The wizard

A wizard is a way to help the user setup a program. In our case, we will create a wizard to guide the user through the process of creating or editing a product. The wizard will be the entirety of all the other user stories that are to be implemented. These user stories will be separated and can be navigated to by the navigation menu. Our goal for this user story is to develop a main frame for the other user stories to be implemented, including a navigation bar. We already have a visual representation of how each wizard step should look like. The description for this feature is as follows:

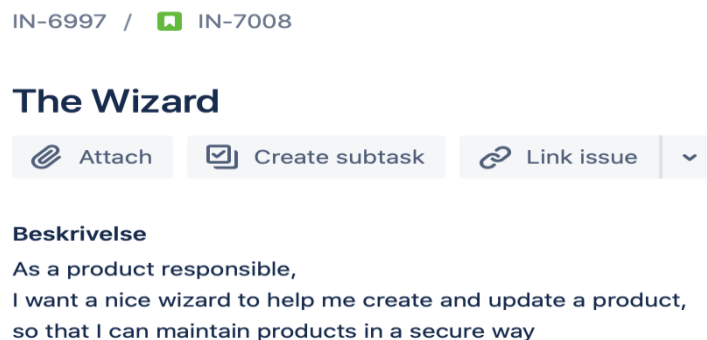






Figure 5 The Wizard description

3.4.3 The identity

The first step of the wizard setup is the identity wizard. This will have all the general information about a product. It should display the product name, product code, product number, product type etc. Our goal for this user story is to develop a feature that can edit the identity of a product. We will also create a “details” page on the top left of the wizard that shows information about the product. The description for this feature is as follows:

IN-6997 /  IN-7009


The Identity

 Attach  Create subtask  Link issue 

Beskrivelse

As a product responsible,
I want to easily create new products and maybe change names and numbers on existing products,
so that I can make sure the customers wants to buy them.

Figure 6 The Identity description




Brukstillatelse og ferdigattest

Midlertidig brukstillatelse er attest som viser at bygningen kan tas i bruk i henhold til byggetillatelsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetillatelsen.

Produktkode: WI30151000

Leverandør: K3015INFOLAND Skiptvedt

Pris VAT: 185,-

Status: Aktiv 

ID

BESKRIVELSE

LEVERING

RELASJONER

BESTILLING

SAKSBEHANDLER

PRIS

PARAMETERE

TILGANG

GODKJENN

Systemnavn

Web Infoland (WI)

Leverandørid

K3015INFOLAND

Nytt produkt

Det er mulig å bruke et annet produkt som utgangspunkt for dette nye produktet. Søk det opp og hent det inn som forslag.

Produktnummer

Hent forslag

Produktnummer

Produktnavn

Brukstillatelse og ferdigattest

Ekstern referanse

Produkttype

Vanlig produkt

Produktpristype

Fast

Aktuelle bransjer ☒ Eiendom & Takst ☐ Bank og Finans ☐ Bygg og Anlegg ☐ Offentlig forvaltning ☒ Andre

Lagre, neste steg

Figure 7 Presentation of the identity feature

3.4.4 The description

The description wizard is a way to edit the description of a product. It should be simple and clean, yet powerful enough to act as an editor. Our goal for this user story is to implement a wizard that can edit and setup the products description. The description for this feature is as follows:

IN-6997 /  IN-7010

The description



Attach



Create subtask



Link issue



Beskrivelse

As a product owner,

I want to quickly and easy change descriptions on the products,
so that the customers knows what they are getting

Figure 8 The description

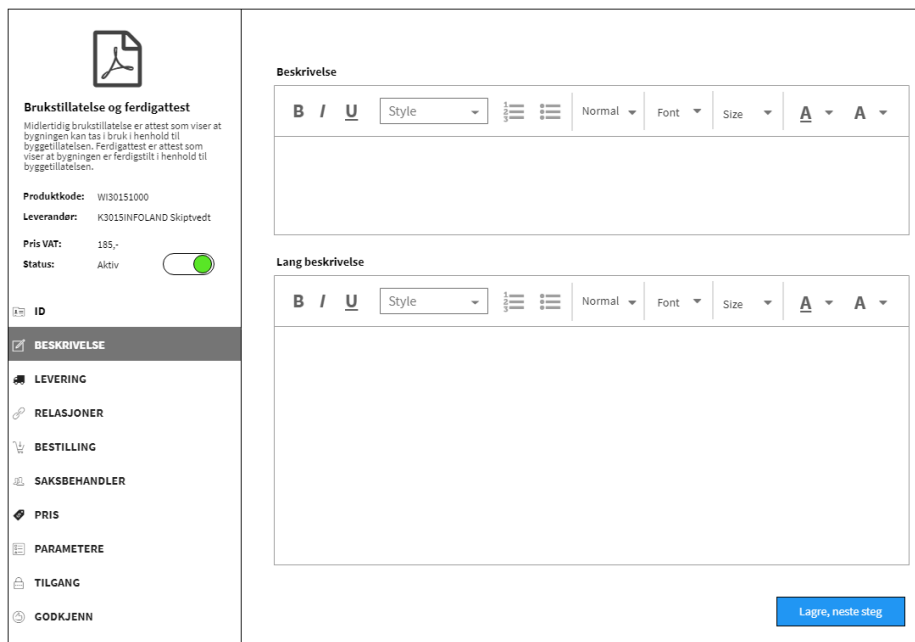


Figure 9 Presentation of the description feature





3.4.5 The delivery

The delivery wizard is a way to edit the products method of delivery. This should include the products availability, purchase type and other necessary features that it needs. Our goal for this user story is to develop a wizard that can setup and edit the current products delivery methods. This should be a simple and a clean wizard that is easy to use for everyone.

The description for this feature is as follows:

IN-6997 /  IN-7011

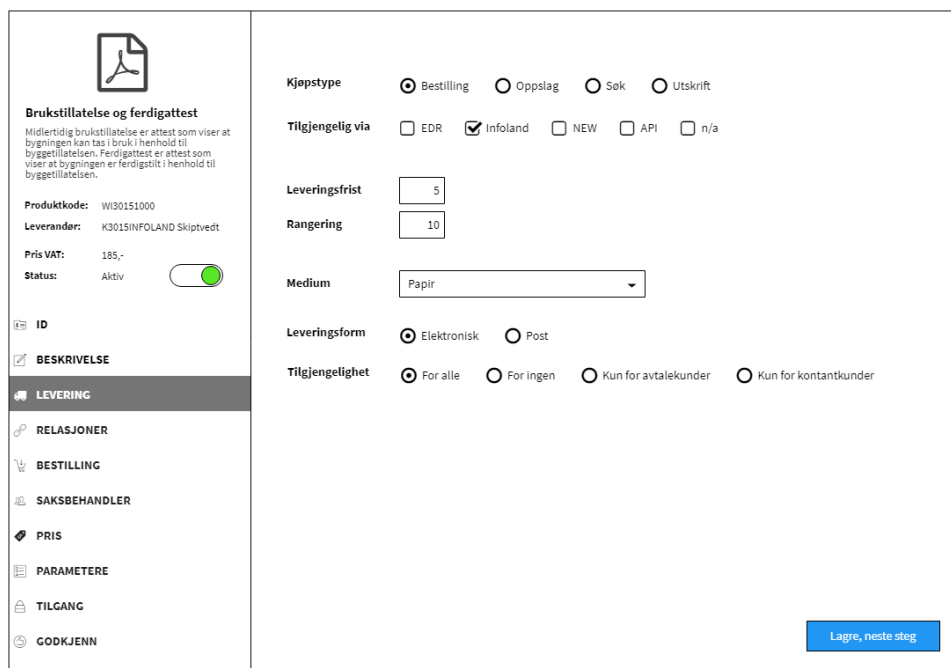
The delivery

 Attach  Create subtask  Link issue 

Beskrivelse

As a product responsible,
I want to say something about delivery time and availability,
so that my customers sees as many products as possible

Figure 10 The delivery description



The screenshot shows a web form for the 'The delivery' feature. On the left is a sidebar with a navigation menu containing: ID, BESKRIVELSE, LEVERING (highlighted), RELASJONER, BESTILLING, SAKSBEHANDLER, PRIS, PARAMETERE, TILGANG, and GODKJENN. The main content area is divided into two columns. The left column contains a document icon, a title 'Brukstillatelse og ferdigattest', a descriptive paragraph, and fields for 'Produktkode' (WI30151000), 'Leverandør' (K3015INFOLAND Skiptvedt), 'Pris VAT' (185,-), and 'Status' (Aktiv with a green toggle switch). The right column contains several sections: 'Kjøpstype' with radio buttons for Bestilling (selected), Oppslag, Søk, and Utskrift; 'Tilgjengelig via' with checkboxes for EDR, Infoland (checked), NEW, API, and n/a; 'Leveringsfrist' and 'Rangering' as input fields with values 5 and 10 respectively; 'Medium' as a dropdown menu set to 'Papir'; 'Leveringsform' with radio buttons for Elektronisk (selected) and Post; and 'Tilgjengelighet' with radio buttons for For alle (selected), For ingen, Kun for avtalekunder, and Kun for kontantkunder. A blue button labeled 'Lagre, neste steg' is at the bottom right.

Figure 11 Representation of the delivery feature


3.4.6 The relations


The relations wizard should display, edit and setup the relations between products. Some products are tied or linked to another product or is simply just a part of a package. Our goal for this user story is to create a wizard that can edit the products relations.


The description for this feature is as follows:


IN-6997 /  IN-7012

The relations

 Attach

 Create subtask


 Link issue




Beskrivelse

As a product responsible,
I want to say something about the relations between products,
so that I can create sexy packages

Figure 12 The relations description



Brukstillatelse og ferdigattest
Midlertidig brukstillatelse er attest som viser at bygningen kan tas i bruk i henhold til byggetillatelsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetillatelsen.

Produktkode: WI30151000
Leverandør: K3015INFOLAND Skiptvedt
Pris VAT: 185,-
Status: Aktiv 

- ID
- BESKRIVELSE
- LEVERING
- RELASJONER**
- BESTILLING
- SAKSBEHANDLER
- PRIS
- PARAMETERE
- TILGANG
- GODKJENN

WI30150010 er en pakke. Her spesifiserer du hvilke produkter som skal tilknyttes.

Søk...

Ingen valgt

☐ Velg alle

☐ WI30150100 Eiendomsinformasjon

☐ WI30150102 Kommunale erklæringer

☐ WI30150104 Landbruks eiendom

☐ WI30150310 Situasjonsskilt

☐ WI30150700 Målebrev

☐ WI30150900 Bygningstegninger

☐ WI30151000 Brukstillatelse og ferdigattest

☐ WI30151350 Reguleringsplan m/bestemmelser

☐ WI30151700 Oversiktskart

☐ WI30151800 Tilknytning til offentlig vann og avløp

Ingenting her...

Lagre, neste steg

Figure 13 Representation of the relations feature

3.4.7 The ordering


The ordering wizard is a way to choose the product type and ordering schema. This wizard should cover the essentials for ordering and should be simple enough for customers to use. The main objective for this user story is to develop a wizard that qualifies as an ordering method. It should also be very simple to understand.


32


The description for this feature is as follows:


IN-6997 /  IN-7013

The ordering

 Attach

 Create subtask


 Link issue



Beskrivelse


As a product responsible,
I want to choose product type and ordering schema,
so that the customers are able to fill out and order what they need










Figure 14 The ordering description



Brukstillatelse og ferdigattest

Midlertidig brukstillatelse er attest som viser at bygningen kan tas i bruk i henhold til byggetillatelsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetillatelsen.

Produktkode: W130151000
Leverandør: K3015INFOLAND Skiptvedt
Pris VAT: 185,-
Status: Aktiv 

- ID
-  BESKRIVELSE
-  LEVERING
-  RELASJONER
-  **BESTILLING**
-  SAKSBEHANDLER
-  PRIS
-  PARAMETERE
-  TILGANG
-  GODKJENN

Påkrevet informasjon ved kjøp av produktet

☒ Matrikkel ☐ Boretts ☐ Forretningsfører ☐ Geografisk utsnitt ☐ Dokument ☐ Province

☒ Benytt standardskjema for påkrevet informasjon

Skjema

Default matrikkel

Lagre, neste steg

Figure 15 Representation of the ordering feature


3.4.8 The executives


The executives wizard will function relatively close to that of the relations wizard. This wizard should display the products current executives, as well as the other executives available. Our main goal for this user story is to create a wizard that can edit, display and select executives for the product.


The description for this feature is as follows:


IN-6997 /  IN-7014

The executives

 Attach


 Create subtask

 Link issue



Beskrivelse
As a product responsible,
I need to set the executive officers of a product, quickly,
so that I'm sure the ordered products are produces and delivered asap

Figure 16 The executives description




Brukstiltalelse og ferdigattest
Midlertidig brukstiltalelse er attest som viser at bygningen kan tas i bruk i henhold til byggetilførsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetilførsen.


Produktkode: W130151000


Leverandør: K3015INPOLAND Skiptvedt


Pris VAT: 185,-


Status: Aktiv 


ID


 **BESKRIVELSE**


 **LEVERING**


 **RELASJONER**


 **BESTILLING**


 **SAKSBEHANDLER**

 **PRIS**

 **PARAMETERE**

 **TILGANG**

 **GODKJENN**

Merk! Endringer her påvirker ikke allerede eksisterende ordre, kun ordre opprettet etter endringen. 

Søk...

Velg alle


☒ Servicekontor


☒ Grethe Monica Bogen

☐ Skiptvet kommune

☐ Svein Magne (Skiptvet)

2 valgt

Servicekontor 

Grethe Monica Bogen 

Lagre, neste steg

Figure 17 Representation of the executives feature

3.4.9 The price





The price wizard is a way for the users to edit and display the product price. This wizard should be very detailed so that it covers all the essentials of the products price. It should also display a history of the product that shows the date and price of the product during that period. Our main objective for this user story is to develop a wizard that can edit and add new product prices, as well as display the products history for price changes.

34

The description for this feature is as follows:

IN-6997 /  IN-7015

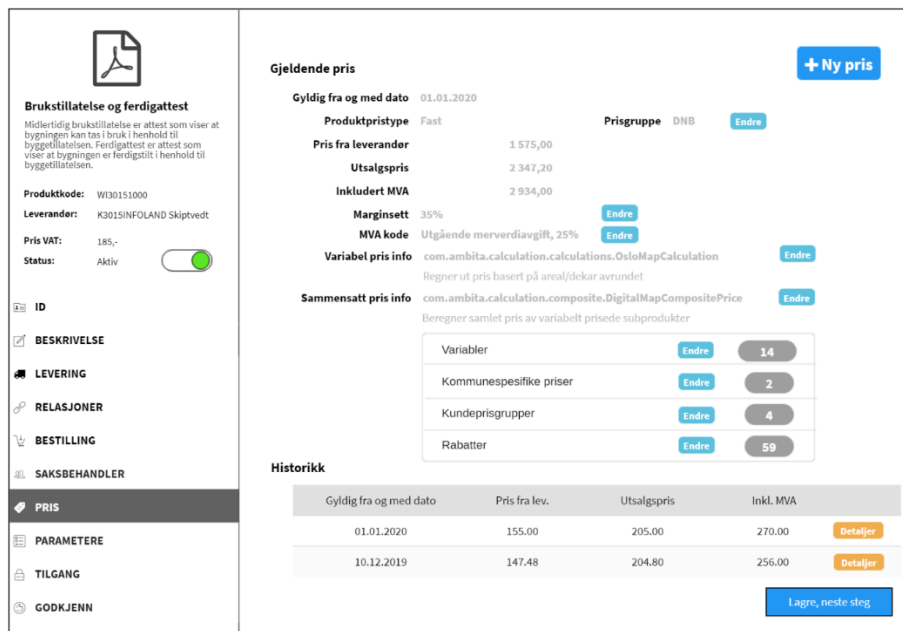
The price

 Attach  Create subtask  Link issue 

Beskrivelse

As a product responsible,
I want to be able to set the correct price or price formula,
so that all parties remains happy

Figure 18 The price description



Brukstillatelse og ferdigattest
Midlertidig brukstillatelse er attest som viser at bygningen kan tas i bruk i henhold til byggetillatelsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetillatelsen.

Produktkode: W130151000
Leverandør: K3015INFOLAND Skiptvedt
Pris VAT: 185,-
Status: Aktiv ☒

ID
BEKRIVELSE
LEVERING
RELASJONER
BESTILLING
SAKSBEHANDLER
PRIS
PARAMETERE
TILGANG
GODKJENN

Gjeldende pris + Ny pris

Gyldig fra og med dato: 01.01.2020

Produktpristype: Fast

Prisgruppe: DNB Endre

Pris fra leverandør: 1 575,00

Utsalgspris: 2 347,20

Inkludert MVA: 2 934,00

Marginsett: 35% Endre

MVA kode: Utgående merverdiavgift, 25% Endre

Variabel pris info: com.ambita.calculation.calculations.OsloMapCalculation Endre
Regner ut pris basert på areal/dekar avrundet

Sammensatt pris info: com.ambita.calculation.composite.DigitalMapCompositePrice Endre
Beregner samlet pris av variabelt prisede subprodukter

Variabler	Endre	
Kommunespesifikke priser	Endre	14
Kundeprisgrupper	Endre	2
Rabatter	Endre	4
	Endre	59

Historikk

Gyldig fra og med dato	Pris fra lev.	Utsalgspris	Inkl. MVA	
01.01.2020	155.00	205.00	270.00	Detaljer
10.12.2019	147.48	204.80	256.00	Detaljer

Lagre, neste steg

Figure 19 Representation of the price feature


3.4.10 The parameters


The parameters wizard should display all the parameters that the product currently has setup. The backend part of this wizard has already been implemented, and the frontend should be the same as before. This means that the goal for this user story will not be to develop, but to re-implement the current parameter feature over to our new wizard. It should also be optimized to fit the current wizard proportion and size.

The description for this feature is as follows:


IN-6997 /  IN-7016

The parameters

 Attach

 Create subtask


 Link issue



Beskrivelse


As a product responsible,
I want to set all the nits and bits, cogs and wheels and bolts correctly,
so that the customers are amazed about what the magic machine can produce

Figure 20 The parameters description









Brukstillatelse og ferdigattest
Midlertidig brukstillatelse er attest som viser at bygningen kan tas i bruk i henhold til byggetillatelsen. Ferdigattest er attest som viser at bygningen er ferdigstilt i henhold til byggetillatelsen.

Produktkode: W30151000
Leverandør: K3015INFOLAND Skiptveed

Pris VAT: 185,-
Status: Aktiv 

- ID
- BESKRIVELSE
- LEVERING
- RELASJONER
- BESTILLING
- SAKSBEHANDLER
- PRIS
- PARAMETERE**
- TILGANG
- CODKJENN

Parametere

Navn	Verdi	
AxaptaAccount	3002	 
AxaptaProductGroup	1221	 
<div>AxaptaAccount</div>		 

Tags

KART ✕

KOMMUNALT ✕

Lagre, neste steg

Figure 21 Representation of the parameters feature





3.4.11 The authorization

The authorization wizard is a way to set the correct authority for each product so that only a selected few can view the right products. This wizard should display the different layer of authorization that Ambita has, and whether it is available for the client or not. Our goal for this user story is to create a wizard that can both edit and set the authority of the product.

The description for this feature is as follows:

IN-6997 /  IN-7017

The authorizations

 Attach  Create subtask  Link issue 

Beskrivelse

As a product responsible,
I want to set the correct authorizations on a product,
so that only the selected few can see my little children

Figure 22 The authorizations description

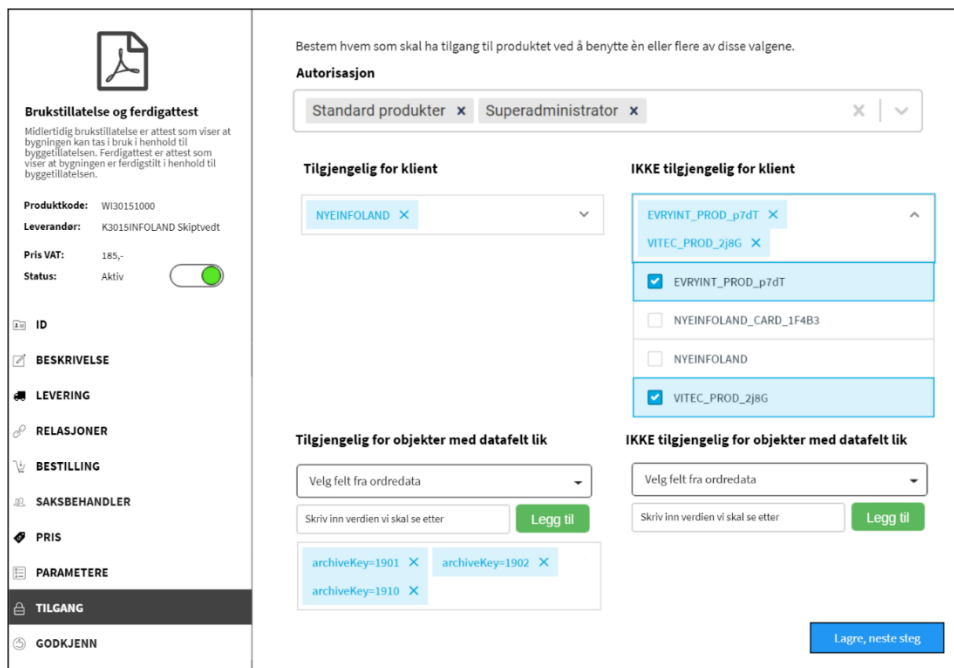


Figure 23 Representations of the authorizations feature


3.4.12 The approval


The approval wizard will be a quick summary over all the previous steps. It should display the most relevant changes and have an approval functionality. We have not yet received any information regarding the visuals, as well as the content it should have. This includes features like buttons, tabs and other variables.


The description for this feature is as follows:


IN-6997 /  IN-7018

The approval

 Attach

 Create subtask

 Link issue



Beskrivelse

As a product responsible,
I want to look over my product, approve and activate it,
so that the customers can buy it as hot bread with butter on

Figure 24 The approval description

4 Method & Material

4.1 Project Group

Our group (Group 2) consists of two member, one that studied Information Technology and the other studied Computer Engineering. This group goes way back and have known each other for a while making us a good team. We would know our strength and weaknesses. When we were stuck on something, we could rely on the partner to know how to solve this issue.

4.2 Project Organization

4.2.1 Task giver

Our Task giver is the company named Ambita. Ambita is filled with professional coworkers and has a good atmosphere. The working culture is serious, but at the same time very casual. We will be receiving our tasks from the product owner. These tasks will be in the form of “user stories”, that has a description and an image of how the feature should look and function.

4.2.2 Project advisor

Our project advisor is Sidney Pontes-Filho who is a PhD-student at Oslo Metropolitan University. We have monthly meetings with our advisor to report back on our progress. Then asking questions if we are wondering about something. When we started writing the report, the meetings with our supervisor became more frequent. Sidney provided us with useful information and even gave us tips and tricks to make our report quality be better.

4.3 Development tools

4.3.1 Atlassian

Ambita uses Atlassian for its products. Some of the products that was used was: Jira Software and Confluence. We used Jira to review the status of the sprint. We saw the usage of the Kanban board in a professional setting. Sometimes we would use these programs to see what

kind of updates that were made either in the backlog or in the specifications file of the project in Confluence.

4.3.2 Git

In this project we used Git in a way that it would not destroy the actual application. Before features and patches goes to production (application that the end users use). It would go through a beta phase, then it would later be merged with the master branch. We worked in our own branch. We created a branch out from “dev” which is development. When the featured was finished, a pull request was made and then waiting for it to be approved. With git, you can both sync your local branch, by “pulling” and give updates to the remote branch by “pushing”. Sometimes merge conflicts will appear, these must be solved before pulling again. A merge conflict will start when the remote repository cannot merge with your local repository because of the pending changes that could be written over by the commits that are being merged in (Atlassian, 2020).

4.3.3 Bitbucket

Bitbucket was used in this project to review pull requests that were made into “dev” and other branches. Developer(s) must approve the pull requests to be able to merge it to another branch. Sometimes a developer sees trailing whitespaces, an extra newline here and there. We would then fix the comments that were posted by other developers to get the pull request approved. There were times when we pushed and pulled new code, thing would not work as they used to. We open the file in bitbucket then see what changed from last commit. We can then resolve the issues we were having. New code is highlighted in green and removed coded is highlighted with red.

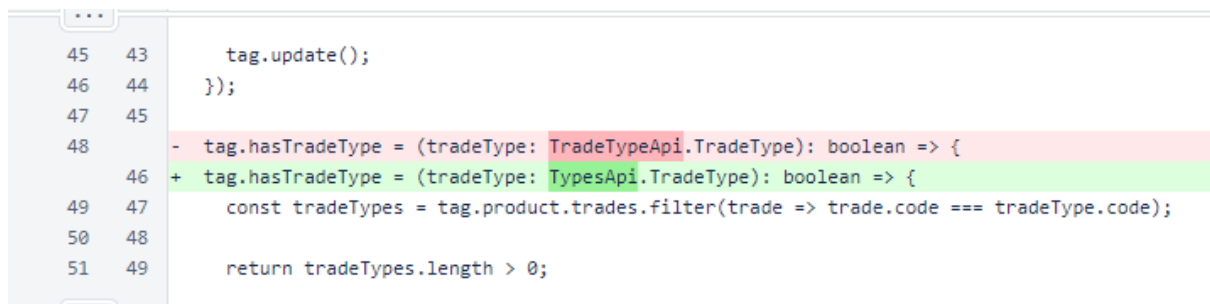


Figure 25 Picture of highlights

4.3.4 Jira Software

We did not use this tool much, the scrum master used it the most. During our daily scrums, the scrum master would open the Jira software, and go to our current sprint. To show us status of each task. If something had been updated it would be moved to another column like “in pull request” or the “done” column. Jira was also used to create sprint tasks and give each task a story point. Sprint planning is already explained above, see chapter [2.5.2](#).

4.3.5 IntelliJ IDEA

IntelliJ was mostly used to write the code for the different API endpoints which was written in Java. IntelliJ was better than VSC to write the APIs with because its better with Java development. The reason is that we could access the database values and see these values. Color highlighting for syntaxes. The swagger notations are highlighted in yellow, so it was easier to distinguish them. It was also easier to create a database connection to PostgreSQL.

4.3.6 Visual Studio Code

VSC was used to write 90% of our code. This is because it was better to write JavaScript and Typescript with. The Ambita-support features/user stories were written in VSC. A lot of customizations of the VSC was done, this was possible by downloading different plugins like: auto-bracket close, color highlighting for syntax, riot plugin, typescript plugin and more. We could customize it to our liking, making our work environment optimized for us. The project was run from the terminal and all git functions here too. Since we used a Git bash terminal instead of cmd.

4.3.7 Slack

Slack was mainly used for communicating with each other. There were text-channels dedicated for different things. Sometimes you would sit far away from the team, so one of the best ways to reach the team, was to write in a slack channel. When we had to work from home, this was our main way of communication with the development team.

4.3.8 Google Docs

Google docs was used to write log, reports and to-do note lists. It was mostly used for writing our daily documentation.

4.3.9 Daily documentation

Every day for four months Google docs were used to log what we did. This was done in a way that we would not have to take screenshots later, and keywords to help us write the report later. The daily documentation was structured in a way that helps us navigate through it. It had the date and month of when a certain feature was worked on.

The daily log was like a daily scrum for us. A little script to what to report the day after. we would always be prepared to daily scrum. If we made progress or not, we would log that, so that we know what we struggled with the day before, so we could solve it the next day.

4.4 Implementation

4.4.1 Implementation requirements

Our group is required to follow Ambita's design structure during the implementation process. This includes the code structure and folder structure. We are also required to use their tools to coordinate and get updates on the project. We had to use their current technology, programming language and communication tools during implementation.

4.4.2 Design structure

The new support application has a very structured design, and we are required to follow it. The folders are structured in a way that is very easy to navigate through and is named properly. Each folder is named after its functionality or feature and holds all the core files of the feature.

The files under each folder are named properly after their feature. It follows the Model View Controller framework (MVC) and has a code structure designed for Riot.js implementation. Ambita uses TypeScript for the controller.ts and index.ts file. The HTML file is named view.tag

because it uses Riot.js tag syntax. The styling is named styles.scss since we will be using SCSS for styling each class.

Infoland folder structure (Only shows Project Phoenix):

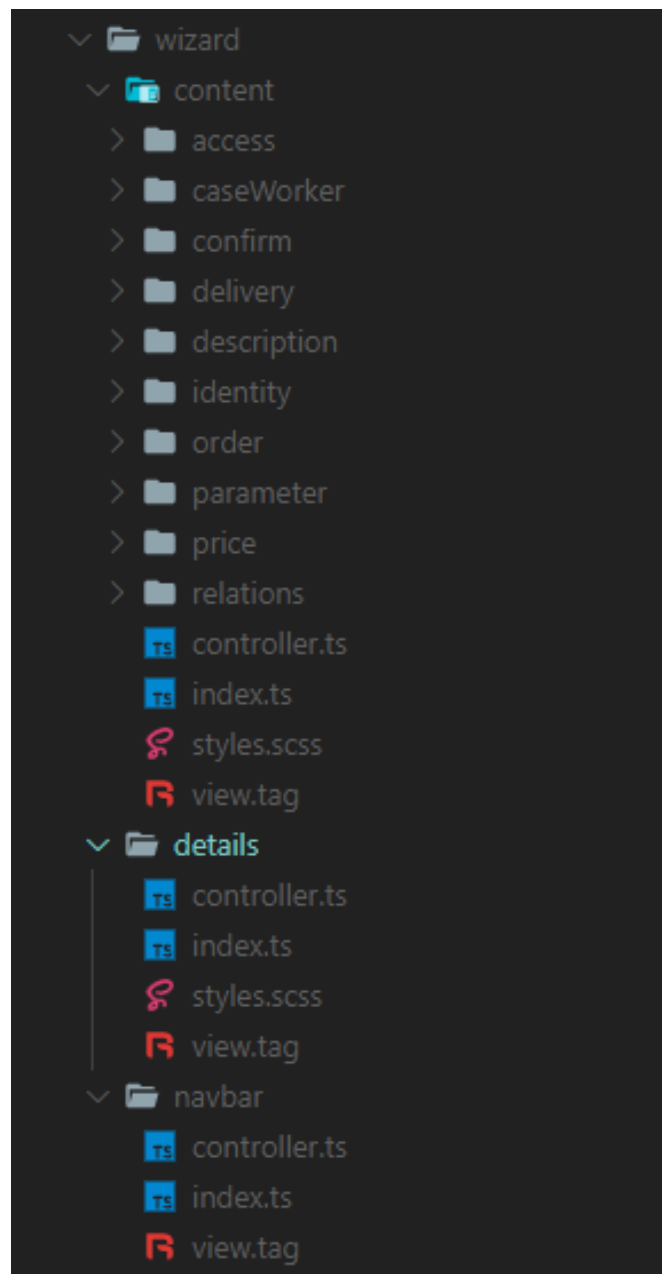


Figure 26 Picture of our project folder

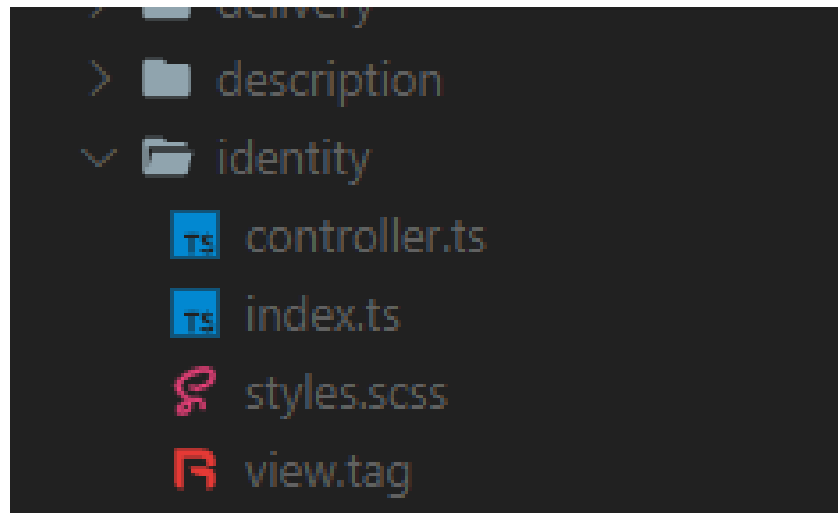


Figure 27 Picture of files in each folder

We will be following this folder structure during the development of Project Phoenix. Each feature will also be named after their respective user story as shown on the figure above. The code structure will also be followed during the projects implementation.

The code structure is very close to a HTML-file but does not have a head or a body tag that encloses its respective contents. In our case, the entire code structure will be enclosed with a tag named after its user story. The content and functionality of the feature are written inside the tag. The script is enclosed in its own script tag similar to how we enclose scripts inside a HTML code.

Since the support application has a massive structure, the classes inside the div tag will have to be named accurately. The scripts are not directly written inside view.tag, but instead inside the controller.ts. We will have to import controller.ts file as well to work together with the view.tag, this goes for the styles.scss as well.

```

view.tag x styles.scss
app > components > product > wizard > content > identity > view.tag > {} "1" > wizard-content-identity.wizard-content-identity > button.an
1 <wizard-content-identity class="wizard-content-identity">
2   <h3>ID</h3>
3
4   <div class="ambita-support-modal_block">
5     <label class="bold space">Systemnavn</label>
6     <div class="wizard-content-identity_container">
7       <select name="systemBelongings" class="width">
8         <option each="{ systemBelonging in systemBelongings }"
9           value="{ systemBelonging.code }"
10          selected="{ systemBelonging.code === product.systemBelonging.code }">
11           { systemBelonging.description } { systemBelonging.code }
12        </option>
13      </select>
14    </div>
15  </div>
16  <div class="ambita-support-modal_block">
17    <label class="bold space">Leverandørid</label>
18    <input type="text" class="wizard-content-identity_input" placeholder="{ product.supplier.code }">
19  </div>
20
21  <div class="ambita-support-modal_block">
22    <div class="wizard-content-identity_new-product">
23      <label class="bold space">Nytt produkt</label>
24      <p class="wizard-content-identity_notice">
25        Det er mulig å bruke et annet produkt som et utgangspunkt for dette nye produktet.
26        Søk det opp og hent den inn som forslag
27      </p>
28      <label class="bold space">Produktnummer</label>
29      <input type="text" class="wizard-content-identity_input">
30      <button class="wizard-content-identity_recommend-button">Hent forslag</button>
31    </div>
32  </div>
33
34  <div class="ambita-support-modal_block">

```

Figure 28 View.tag code structure

```

view.tag styles.scss ...identity styles.scss ...modal x
app > components > _common > modal > styles.scss > .ambita-support-modal > &__wizard
1 @import '.././.././../styles/globals';
2
3 .ambita-support-modal--active {
4   overflow: hidden;
5 }
6
7 .ambita-support-modal {
8   &__overlay {
9     position: fixed;
10    top: 0;
11    left: 0;
12    width: 100%;
13    height: 100%;
14    background-color: transparentize($color-black, 0.2);
15    z-index: $z-index-modal;
16    overflow-y: auto;
17
18    &:focus {
19      outline: none;
20    }
21  }
22
23  &__block {
24    padding-top: 10px;
25    padding-bottom: 10px;
26  }
27
28  &__container {
29    width: 1000px;
30    margin: 50px auto 30px;
31    background-color: white;
32  }
33

```

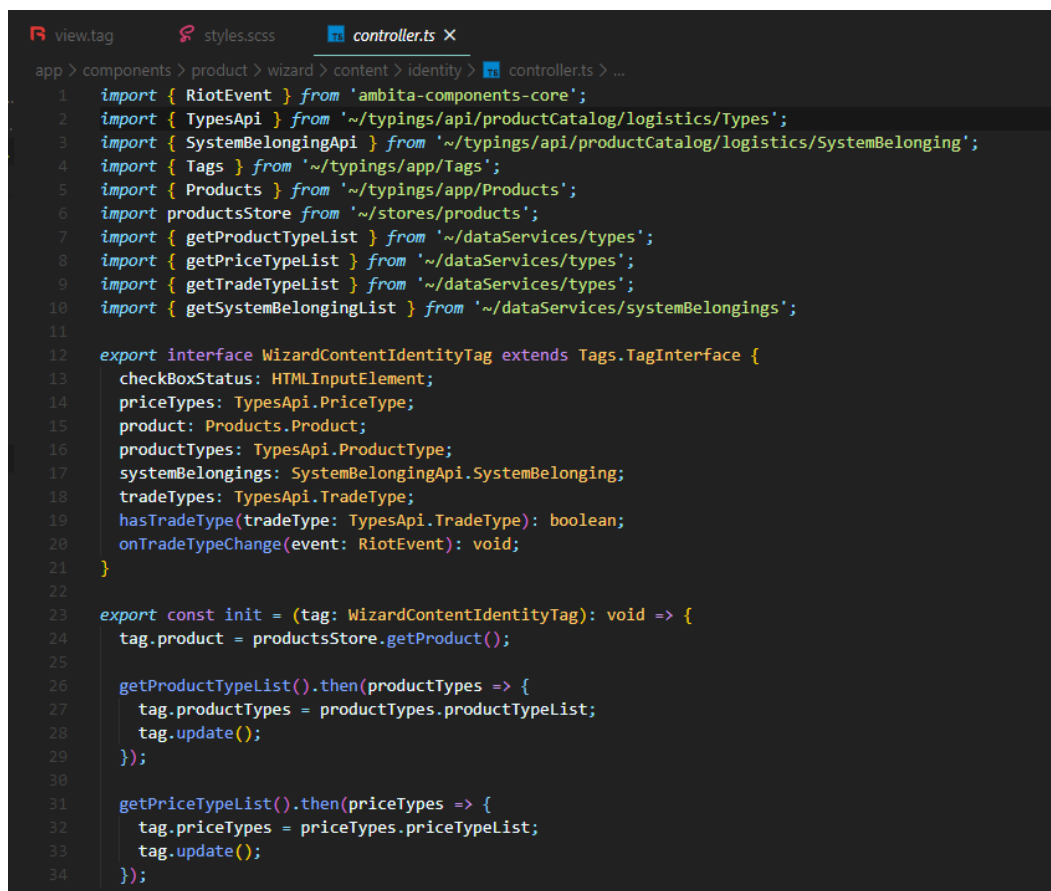
Figure 29 styles.scss code structure

Figure 29 shows an example of how the nesting structure should look. It imports from the global styles in their support application. Most of the files imports from functionalities from other files, this is especially true for the controller.ts file.

The styles.scss file is very similar to a regular CSS-file. It uses almost the identical syntax but is more powerful and has a lot more features. We will structure our styles.scss files in the same way it has been structured in the other support application features.

The controller.ts is written in TypeScript and shares a structure close to a Java or C-language. The first lines import the dependencies. This often includes the main API types, Tags and also Riot Events to link with the view.tag. In figure 30 from line 1-10 are the imports for the controllers. From line 12 to 21 we have the interface, which basically adds all the required variable and tags that can be called in the view.tag.

Controller.ts code structure (Identity wizard example):



```
app > components > product > wizard > content > identity > controller.ts > ...
1  import { RiotEvent } from 'ambita-components-core';
2  import { TypesApi } from '~/typings/api/productCatalog/logistics/Types';
3  import { SystemBelongingApi } from '~/typings/api/productCatalog/logistics/SystemBelonging';
4  import { Tags } from '~/typings/app/Tags';
5  import { Products } from '~/typings/app/Products';
6  import productsStore from '~/stores/products';
7  import { getProductTypeList } from '~/dataServices/types';
8  import { getPriceTypeList } from '~/dataServices/types';
9  import { getTradeTypeList } from '~/dataServices/types';
10 import { getSystemBelongingList } from '~/dataServices/systemBelongings';
11
12 export interface WizardContentIdentityTag extends Tags.TagInterface {
13   checkBoxStatus: HTMLInputElement;
14   priceTypes: TypesApi.PriceType;
15   product: Products.Product;
16   productTypes: TypesApi.ProductType;
17   systemBelongings: SystemBelongingApi.SystemBelonging;
18   tradeTypes: TypesApi.TradeType;
19   hasTradeType(tradeType: TypesApi.TradeType): boolean;
20   onTradeTypeChange(event: RiotEvent): void;
21 }
22
23 export const init = (tag: WizardContentIdentityTag): void => {
24   tag.product = productsStore.getProduct();
25
26   getProductTypeList().then(productTypes => {
27     tag.productTypes = productTypes.productTypeList;
28     tag.update();
29   });
30
31   getPriceTypeList().then(priceTypes => {
32     tag.priceTypes = priceTypes.priceTypeList;
33     tag.update();
34   });
35 }
```

Figure 30 controllers.ts code structure

Our group will be implementing all the controller.ts, index.ts, view.tag and styles.scss to match the code structure similar to the examples shown above. This will also help us immensely by having a clean structure that is easy to navigate through.

4.4.3 Used technologies

Team Infoland utilizes multiple technologies and tools for their projects. For the implementation of Project Phoenix, we will be using the computer they have provided for us. This is because we require Virtual Private Network (VPN) access, as well as Secure Shell (SSH) connection to be able to access their servers.

Here are the current specs for the PC provided to us:

View basic information about your computer

Windows edition

Windows 10 Home

© 2019 Microsoft Corporation. All rights reserved.

System

Manufacturer:	ASUSTek Computer Inc.
Processor:	AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx 2.10 GHz
Installed memory (RAM):	16.0 GB (15.8 GB usable)
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

Figure 31 Laptop spec

We will also be utilizing the Atlassian user they have provided us with. This allows us to coordinate, communicate and update each party during project development.

4.4.4 Used programming languages

The used programming languages during the implementation process were:

- Java
- Riot.js
- TypeScript
- JavaScript
- A bit of HTML & SCSS (Tags and class structure)

5 Results

5.1 Functionality & Design

5.1.1 The Modal

The modal was the very first thing we implemented. This was to help us understand the main structure and ropes of how to work with their support application. We managed to create a button that opens the modal, and also completely removed all the features inside the existing “edit product” button. The modal is divided into three containers. These containers each have their own functionality being navigation, product details (description) and the schema (contents).

The First container is the description, which gives a small description of the product. The second container is the navigation menu, allowing the users to navigate through the wizard. The third container is the schema, which includes the content and main feature for editing a product.

The Modal feature:

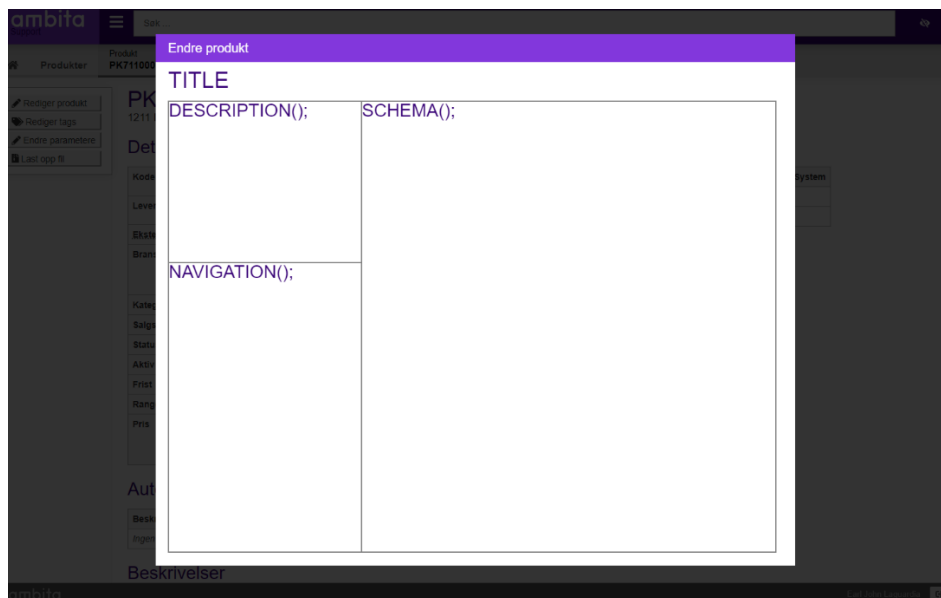


Figure 32 Modal feature

5.1.2 The Wizard

The wizard is a setup process of the product. We managed to implement a navigation bar that switches between the contents, allowing the user to navigate through the modal. The top-left container gives a small description of the selected product. It also gives the user a quick summary of the name, price and status of the viewed product.

The navigation bar uses the “Font Awesome” icons and will get highlighted when the user hovers over the selected feature. We tried implementing the navigation bar to be as close to the request as possible.

The schema container switches its content depending on the selected navigation menu. This was done using Riot Events. The default page is the identity wizard, which allows the user to edit the identity information of a product. We have also implemented the “Lagre, neste steg” button as mentioned in the user stories.

The Wizard feature:

The screenshot displays the 'Endre produkt' (Edit product) wizard. The left sidebar contains a list of navigation options: 'Godkjente bygningstegninger', 'ID', 'BESKRIVELSE', 'LEVERING', 'RELASJONER', 'BESTILLING', 'SAKSBEHANDLER', 'PRIS', 'PARAMETERE', 'TILGANG', and 'GODKJENN'. The main content area is divided into two columns. The left column, under the 'Godkjente bygningstegninger' tab, shows product details: 'Produktkode: W99990000', 'Leverandør: K9999INFOLAND', 'Kostpris: 115,-', 'Pris NET: 150,-', 'Pris VAT: 200,-', and 'Status: Aktiv'. The right column, under the 'ID' tab, contains fields for 'Systemnavn' (Web Infoland (WI)), 'Leverandør' (K9999INFOLAND), and a 'Nytt produkt' section with a 'Hent forslag' button. Below this are fields for 'Produktnummer' (W99990900), 'Produktnavn' (Godkjente bygningstegninger), 'Ekstern referanse', 'Produkttype' (Vanlig produkt), and 'Produktpristype' (Fast). At the bottom right is a 'Lagre, neste steg' button.

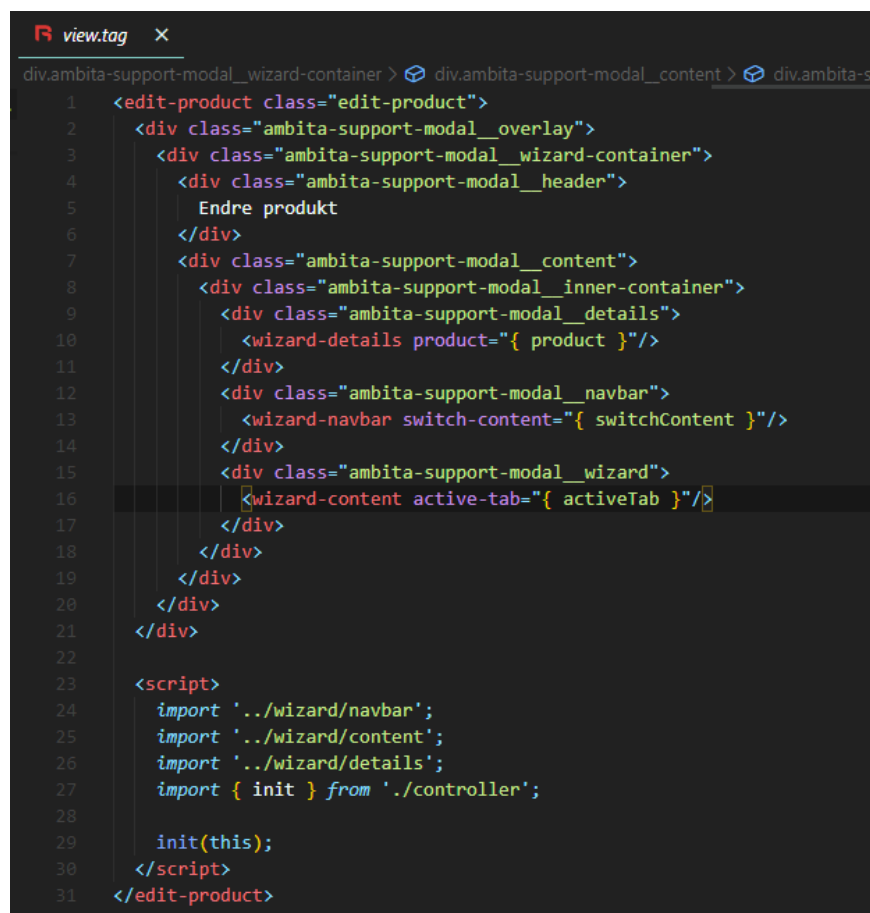
Figure 33 The identity feature

5.2 Frontend Development

5.2.1 The modal

The modal was the very first feature to be implemented. It allowed us to get familiar with the support application folder structure. This also let us experience how a decently large company coordinate using version control.

The modal wizard view.tag:



```
1 <edit-product class="edit-product">
2   <div class="ambita-support-modal__overlay">
3     <div class="ambita-support-modal__wizard-container">
4       <div class="ambita-support-modal__header">
5         Endre produkt
6       </div>
7       <div class="ambita-support-modal__content">
8         <div class="ambita-support-modal__inner-container">
9           <div class="ambita-support-modal__details">
10             <wizard-details product="{ product }"/>
11           </div>
12           <div class="ambita-support-modal__navbar">
13             <wizard-navbar switch-content="{ switchContent }"/>
14           </div>
15           <div class="ambita-support-modal__wizard">
16             <wizard-content active-tab="{ activeTab }"/>
17           </div>
18         </div>
19       </div>
20     </div>
21   </div>
22
23   <script>
24     import '../wizard/navbar';
25     import '../wizard/content';
26     import '../wizard/details';
27     import { init } from './controller';
28
29     init(this);
30   </script>
31 </edit-product>
```

Figure 34 view.tag of edit-products modal

The view.tag file is a simple structure written in Riot.js. It uses the tag-syntax, where each tag has its own contents of codes written in it. This allows us to display contents by just simply adding an enclosed tags on the view.tag file.

The navigation bar, wizard content and product details were also imported on the same view.tag file as seen on the image above.

The modal does not import any styles.scss files. Instead it uses the global styles.scss that was already made inside the support application global files. This was done so that we did not have to write a duplicate code on the other user stories. It was also advised and instructed to us by the Infoland team to not write duplicated code.

The modal wizard styles.scss:

```
.ambita-support-modal {
  &__overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background-color: transparentize($color-black, 0.2);
    z-index: $z-index-modal;
    overflow-y: auto;

    &:focus {
      outline: none;
    }
  }

  &__block {
    padding-top: 10px;
    padding-bottom: 10px;
  }

  &__container {
    width: 1000px;
    margin: 50px auto 30px;
    background-color: white;
  }

  &__header {
```

Figure 35 The modal scss

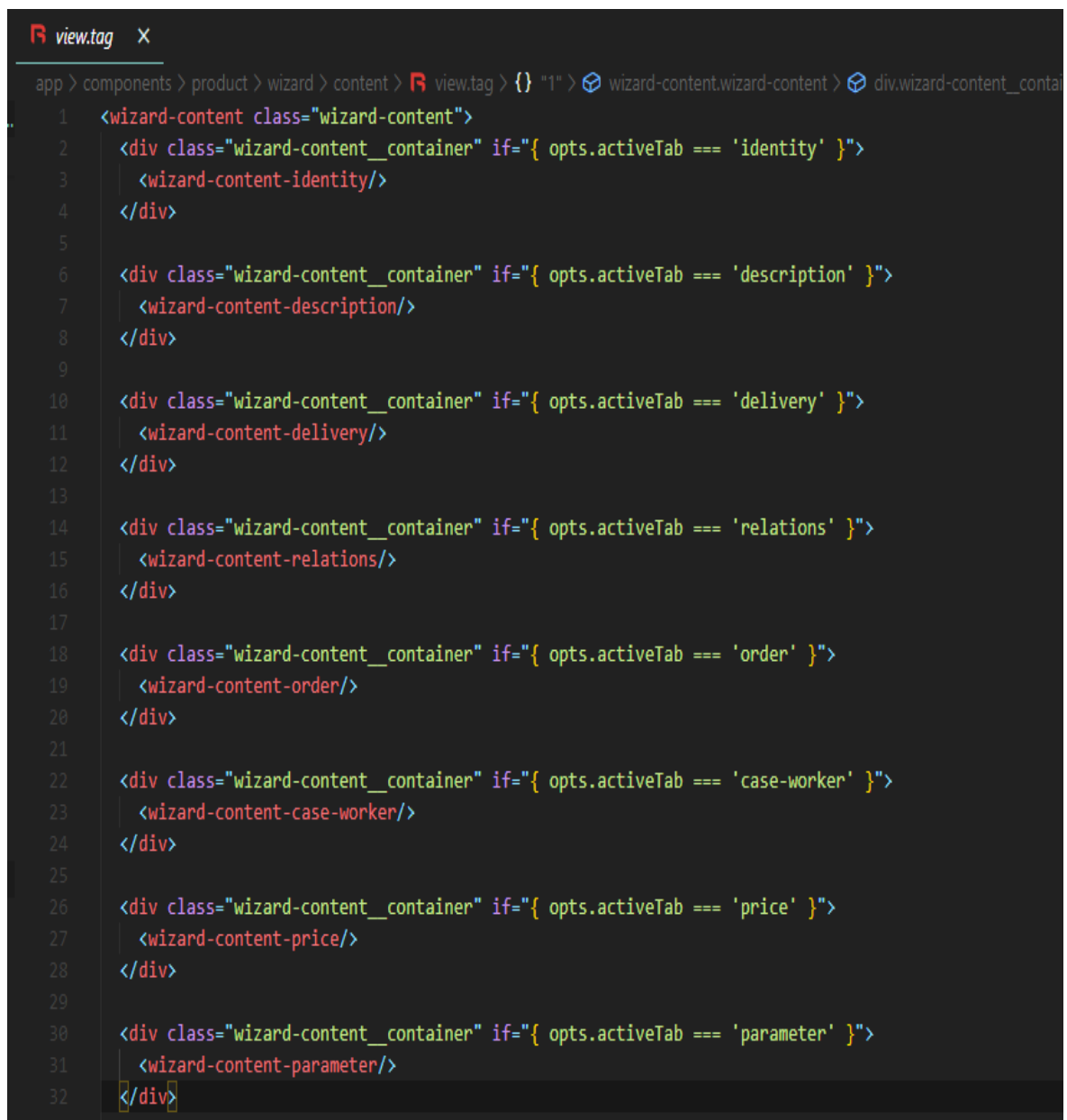
The image above shows how we stylized the modal using scss. It does not show the entire code as it is too long and is also used by the other support application features.

5.2.2 The wizard

The wizard is heavily reliant on the Riot Events to function. Each user story we implemented are basically features of their own and is a part of the wizard. To create a setup wizard, we had to somehow switch between these contents when clicking on a navigation menu.

To switch between contents, we created a Riot Event that connects each Riot.js tag on an interface controller. It displays and swaps out the current default content with the chosen content. This is done through the navigation menu and is imported through the view.tag under the scripts tag.

The wizard view.tag:



```
1 <wizard-content class="wizard-content">
2   <div class="wizard-content__container" if="{ opts.activeTab === 'identity' }">
3     <wizard-content-identity/>
4   </div>
5
6   <div class="wizard-content__container" if="{ opts.activeTab === 'description' }">
7     <wizard-content-description/>
8   </div>
9
10  <div class="wizard-content__container" if="{ opts.activeTab === 'delivery' }">
11    <wizard-content-delivery/>
12  </div>
13
14  <div class="wizard-content__container" if="{ opts.activeTab === 'relations' }">
15    <wizard-content-relations/>
16  </div>
17
18  <div class="wizard-content__container" if="{ opts.activeTab === 'order' }">
19    <wizard-content-order/>
20  </div>
21
22  <div class="wizard-content__container" if="{ opts.activeTab === 'case-worker' }">
23    <wizard-content-case-worker/>
24  </div>
25
26  <div class="wizard-content__container" if="{ opts.activeTab === 'price' }">
27    <wizard-content-price/>
28  </div>
29
30  <div class="wizard-content__container" if="{ opts.activeTab === 'parameter' }">
31    <wizard-content-parameter/>
32  </div>
```

Figure 36 styles.scss of navigation bar in modal

The styles for the wizard contain multiple container, each having similar style. The wizard can be divided into three sections. The first one displays the details of the product, the second one displays the navigation menu, and the third displays the selected content (Identity wizard by default).

The wizard styles.scss:

```
&__wizard-container {  
  width: 1140px;  
  margin: 50px auto 30px;  
  background-color: #white;  
}  
  
&__inner-container {  
  width: 1100px;  
  height: 790px;  
}  
  
&__details {  
  position: static;  
  float: left;  
  margin: auto;  
  padding: 20px;  
  width: 220px;  
  height: 300px;  
  border: 0 solid #gray;  
  outline: #gray solid thin;  
  background-color: #white;  
  overflow: hidden;  
}  
  
&__navbar {  
  position: static;  
  float: left;  
  margin: 340px -260px 0;  
  width: 260px;  
  height: 450px;  
  border: 0 solid #gray;  
  outline: #gray solid thin;  
  background-color: #white;  
  overflow: hidden;
```

Figure 37 styles.scss of the wizard

The image above shows how we implemented the style for the wizard frame. They share almost an identical style and layout but has different proportions to contain the features that is to be implemented in it. This is a code snippet of styles.scss of the wizard showing the structure of the code. It does cover the main concept about how we went to implement the style.

5.2.3 Navigation bar

The navigation bar is a simple container that contains customized buttons. Each button changes the content on the wizard. It was a relatively simple implementation and takes advantage of using the Font Awesome icons.

The navigation bar:

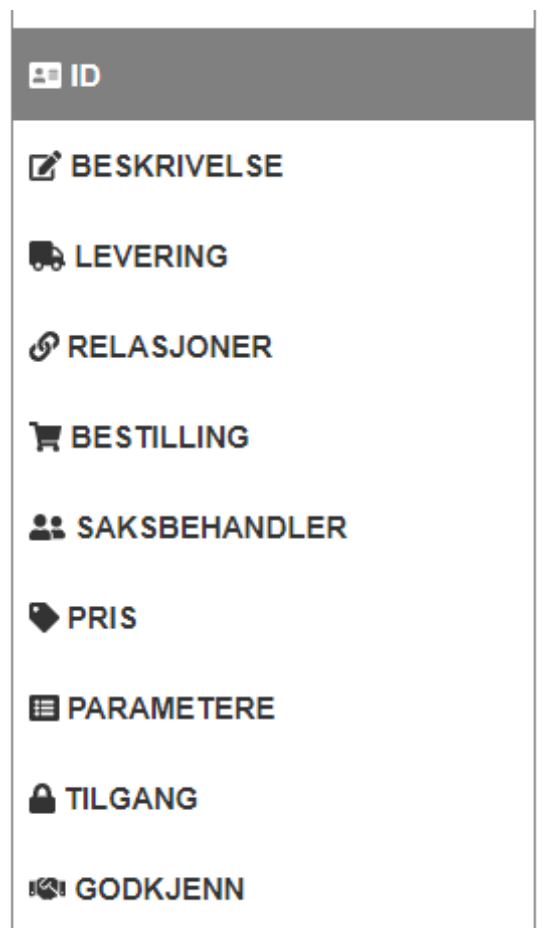


Figure 38 Navigation bar

The image above shows a visual representation of the navigation bar we implemented. We tried to make it as close to the requested user story image as possible. The buttons get highlighted when hovered on as requested by the product owner.

The view.tag code uses the wizards Riot Events to call in its functions when clicking on each buttons. The `opts.switchContent` changes the content shown on the wizard content container depending on the button id.

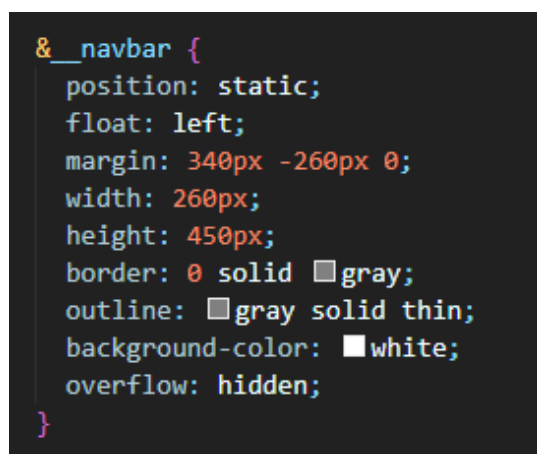
The navigation bar view.tag



```
1 <wizard-navbar>
2   <button id="identity" onclick="{ opts.switchContent }" class="ambita-support-modal__wizard-button">
3     <i class="fas fa-address-card"></i> ID
4   </button>
5
6   <button id="description" onclick="{ opts.switchContent }" class="ambita-support-modal__wizard-button">
7     <i class="fas fa-edit"></i> BESKRIVELSE
8   </button>
9
10  <button id="delivery" onclick="{ opts.switchContent }" class="ambita-support-modal__wizard-button">
11    <i class="fas fa-truck"></i> LEVERING
12  </button>
13
14  <button id="relations" onclick="{ opts.switchContent }" class="ambita-support-modal__wizard-button">
15    <i class="fas fa-link"></i> RELASJONER
16  </button>
17
18  <button id="order" onclick="{ opts.switchContent }" class="ambita-support-modal__wizard-button">
19    <i class="fas fa-shopping-cart"></i> BESTILLING
20  </button>
```

Figure 39 view.tag of the navigation bar

The navigation bar styles.scss



```
&__navbar {
  position: static;
  float: left;
  margin: 340px -260px 0;
  width: 260px;
  height: 450px;
  border: 0 solid #gray;
  outline: #gray solid thin;
  background-color: #white;
  overflow: hidden;
}
```

Figure 40 styles.scss of the navigation bar

5.2.4 The identity

The identity wizard was the third feature to be implemented. It allowed us to get used to their code structure and how everything is connected and imported. This wizard contains some input fields, dropdown menu and checkboxes.

Endre produkt

Godkjente bygningstegninger
Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Produktkode: WI99990900
Leverandør: K9999INFOLAND
Kostpris: 115 -
Pris NET: 160 -
Pris VAT: 200 -
Status: Aktivt ☒

ID

BESKRIVELSE
LEVERING
RELASJONER
BESTILLING
SAKSBEHANDLER
PRIS
PARAMETERE
TILGANG
GODKJENN

ID

Systemnavn: Web Infoland (WI)
Leverandør: K9999INFOLAND

Nytt produkt
Det er mulig å bruke et annet produkt som et utgangspunkt for dette nye produktet. Søk det opp og hent den inn som forslag

Produktnummer: [Hent forslag](#)

Produktnummer: WI99990900
Produktnavn: Godkjente bygningstegninger
Ekstern referanse:
Produkttype: Vanlig produkt
Produktpristype: Fast

Aktuelle bransjer
☒ Eiendom og Takst
☐ Bank og Finans
☒ Bygg og Anlegg
☒ Offentlig forvaltning
☒ Andre

[Lagre, neste steg](#)

Figure 41 Final result of the wizard feature

Figure 41 shows the final result of the identity wizard. Some of the fields displays the selected products general variables. This is done through the backend part of the development and connects to a database that receives the products general information.

Figure 42 shows the final code for the identity wizard we implemented. It has a simple code structure similar to that of an HTML-file except that these are enclosed inside the wizard-content-identity tag. Most of divs classes are imported from the global styles.scss to avoid duplication of code following the DRY principle. DRY principle or Don't repeat yourself is a basic software development principle aimed at reducing repetition of information (Baghel, 2020). Keep in mind that the image does not show the entire code structure as it was very long due to the amount of dropdown menus and input fields.

The identity wizard view.tag:

```
<wizard-content-identity class="wizard-content-identity">
  <h3>ID</h3>

  <div class="ambita-support-modal_block">
    <label class="bold space">Systemnavn</label>
    <div class="wizard-content-identity__container">
      <select name="systemBelongings" class="width">
        <option each="{ systemBelonging in systemBelongings }"
          value="{ systemBelonging.code }"
          selected="{ systemBelonging.code === product.systemBelonging.code }">
          { systemBelonging.description } ({ systemBelonging.code })
        </option>
      </select>
    </div>
  </div>
  <div class="ambita-support-modal_block">
    <label class="bold space">Leverandørid</label>
    <input type="text" class="wizard-content-identity__input" placeholder="{ product.supplier.code }">
  </div>

  <div class="ambita-support-modal_block">
    <div class="wizard-content-identity__new-product">
      <label class="bold space">Nytt produkt</label>
      <p class="wizard-content-identity__notice">
        Det er mulig å bruke et annet produkt som et utgangspunkt for dette nye produktet.
        Søk det opp og hent den inn som forslag
      </p>
      <label class="bold space">Produktnummer</label>
      <input type="text" class="wizard-content-identity__input">
      <button class="wizard-content-identity__recommend-button">Hent forslag</button>
    </div>
  </div>

  <div class="ambita-support-modal_block">
```

Figure 42 Final code for the identity

The styles.scss for the identity wizard contained mostly of unique classes and id. They exclusively belong to the identity class and is directly imported inside the view.tag under the scripts section. The code focuses on customizing the identity wizards input fields and button. There are two different buttons on the identity wizard. One uses the global styles, while the other one uses the unique style written inside the identity wizard styles.scss.

Figure 43 shows how the classes for the identity wizard was written. Even though it is written in SCSS, it looks identical to a CSS-file. This was our first take on how to use SCSS and its code structure.

The identity wizard styles.scss

```
.wizard-content-identity {  
  &__container {  
    width: 250px;  
    overflow: hidden;  
  }  
  
  &__input {  
    width: 250px;  
    box-sizing: border-box;  
  }  
  
  &__new-product {  
    background-color: #eee;  
    padding-bottom: 10px;  
  }  
  
  &__notice {  
    font-size: 12px;  
    clear: both;  
  }  
  
  &__recommend-button {  
    background-color: #2196f3;  
    color: white;  
    padding: 5px;  
    font-size: 12px;  
    border-color: black;  
    border-width: thin;  
    margin-left: 5px;  
  }  
}
```

Figure 43 styles.scss of the identity feature

5.2.5 The description

The description wizard was simple to implement. It consists of a short description field and a long description field. This wizard is supposed to let the user write a short or a long description for the product. It is supposed to be simple yet powerful enough to be an editor.

Figure 44 an image of the actual result of the description tab. Some things that are worth to mention is that this is a test/demo product. Hence the weird descriptions, the text formatting bar is not the same as the one on the requirement, because of the code was not in this project but rather a different project. Therefore, the text formatting tool looked like that.

The description wizard:

Godkjente bygningstegninger

Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Produktkode:

W199990900

Leverandør:

K9999INFOLAND

Kostpris:

116,-

Pris NET:

160,-

Pris VAT:

200,-

Status:

Aktivt

ID

BESKRIVELSE

LEVERING

RELASJONER

BESTILLING

SAKSBEHANDLER

PRIS

PARAMETERE

TILGANG

GODKJENN

Beskrivelse

Kort beskrivelse

Normal

B

I

U

Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Lang beskrivelse

Normal

B

I

U

Lang lang lkasjdiksaj Iskajd Iskajd Iskajd Iskajd Iskajd Iskajd Iskajd Iksaj dksaj dlksaj dlksaj dlksaj dlksaj dlksaj dsaj dlsajdlksaj dksaj dlsa Idksaj d uuvvuvuevvwvewv oneyetyenewvwvuwuwue ughewebubewem ossas

Lagre, neste steg

Figure 44 Final result of the description feature

The description wizard view.tag:

```
<wizard-content-description class="wizard-content-description">
  <h3>Beskrivelse</h3>

  <div class="ambita-support-modal_block">
    <p class="bold">Kort beskrivelse</p>
    <div class="wizard-content-description__short-description" ref="shortDescription">{ product.description }</div>
  </div>

  <div class="ambita-support-modal_block">
    <p class="bold">Lang beskrivelse</p>
    <div class="wizard-content-description__long-description" ref="longDescription">{ product.long_description }</div>
  </div>

  <button class="ambita-support-modal__button-next">Lagre, neste steg</button>

</script>
import './styles';
import { init } from './controller';

init(this);
</script>
</wizard-content-description>
```

Figure 45 Final code for the description feature

The actual code in the view.tag looked like the figure above. All that was done was referring to the short and longDescription which is in the controller. The content of the div class is the products description called with { **product.long_description** } and { **product.description** }.

```
.wizard-content-description {  
  &_short-description {  
    resize: none;  
    height: 120px;  
  }  
  
  &_long-description {  
    resize: none;  
    height: 250px;  
  }  
}
```

Figure 46 styles.scss for the description feature

The only scss that was written is shown in the figure above. Width was already predefined in the other project and was just imported over on the view.tag file.

5.2.6 The delivery

The delivery wizard was a relatively simple task. It has multiple radio and checkboxes for the user to set the proper delivery settings. The user story displayed a horizontal type of implementation for these radio boxes, but we decided that a vertical one would be much cleaner and easier to use. The product owner also approved this.

Figure 47 shows how the delivery wizard turned out. There is no logic behind this figure, the only thing that had to be made was the input boxes. So, the user can select the different values.

Endre produkt

Godkjente bygningstegninger

Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Produktkode:

W09990900

Leverandør:

K0999/INFOLAND

Kostpris:

118,-

Pris NET:

160,-

Pris VAT:

200,-

Status:

Aktivt

ID

BESKRIVELSE

LEVERING

RELASJONER

BESTILLING

SAKSBEHANDLER

PRIS

PARAMETERE

TILGANG

GODKJENN

Levering

Kjøpstype

Bestilling

Oppslag

Søk

Utskrift

Tilgjengelig via

EDR

Infoland

NEW

API

n/a

Leveringsfrist

Rangering

Medium

Bildefil

Leveringsform

Elektronisk

Post

Tilgjengelighet

For alle

For ingen

Kun for avtalekunder

Kun for kontantkunder

Lagre, neste steg

Figure 47 Final result of the delivery feature

```

<wizard-content-delivery class="wizard-content-delivery">
  <h3>Levering</h3>

  <div class="ambita-support-modal_block">
    <label class="bold space">Kjøpstype</label>
    <div class="wizard-content-delivery_container">
      <label class="ambita-support-modal_radio-container"> Bestilling
        <input type="radio" name="purchaseType">
        <span class="ambita-support-modal_radio-checkmark"></span>
      </label>
      <label class="ambita-support-modal_radio-container"> Oppslag
        <input type="radio" name="purchaseType">
        <span class="ambita-support-modal_radio-checkmark"></span>
      </label>
      <label class="ambita-support-modal_radio-container"> Søk
        <input type="radio" name="purchaseType">
        <span class="ambita-support-modal_radio-checkmark"></span>
      </label>
      <label class="ambita-support-modal_radio-container"> Utskrift
        <input type="radio" name="purchaseType">
        <span class="ambita-support-modal_radio-checkmark"></span>
      </label>
    </div>
  </div>
</div>

```

Figure 48 Final code for the delivery feature

Because of slow pull request and no API-endpoints available. They were temporarily hardcoded in the code. If it were not hard coded it would have looked something like this. Taking “Kjøpstype” as an example:

```
<div class="ambita-support-delivery__container">
  <label class="ambita-support-modal__radio-container"
    each="{ purchaseType in purchaseTypes }">
    <input name="branches" type="radio" value="purchaseType.code">
      { purchaseType.name } </input>
    <span class="ambita-support-modal__radio-checkmark"></span>
  </label>
</div>
```

Figure 49 Example code 1

What it does is making a loop which then iterates through the object purchaseTypes array, and then put every purchaseType.name as a radio button value.

The delivery wizard styles.scss:

```
.wizard-content-delivery {
  &__container {
    width: 250px;
    overflow: hidden;
  }
}
```

Figure 50 styles.scss of the delivery

5.2.7 The relations

This task was surprisingly straightforward but also hard. As shown in the user story it had to have sides, where you can move a product relation back and forth. Five different containers must be made to make this work. Some of the containers had overflow problems, so we fixed that by limiting which container could have visible overflow.

The relations wizard:

Endre produkt

Meglerpakke
Pakken inneholder:

Produktkode: WI99990033
Leverandør: K9999INFOLAND
Kostpris: 1200 -
Pris NET: 1650 4 -
Pris VAT: 2003 -
Status: Aktivt

ID

BESKRIVELSE

LEVERING

RELASJONER

BESTILLING

SAKSBEHANDLER

PRIS

PARAMETERE

TILGANG

GODKJENN

Relasjoner

WI99990033 er en pakke. Her spesifiserer du hvilke produkter som skal tilknyttes

Søk...	Valgt (6)	Fjern Alle
<input type="checkbox"/> Velg alle	WI99990400 Grunnkart	X
<input type="checkbox"/> WI99996040 Arkivdokument	WI99990900 Godkjente bygningstegninger	X
<input type="checkbox"/> WI99991710 Basiskart	WI99990901 Midlertidig brukstillatelse og ferdigattest	X
<input type="checkbox"/> WI99996010 Basiskart som vektordata	WI99991001 Tilknytning til offentlig vann og kloakk - skjemaløsning	X
<input type="checkbox"/> WI99996001 Basiskart som vektordata utvalg (Bergen test)	WI99991600 Kommunale avgifter og eiendomsskatt	X
<input type="checkbox"/> WI99996002 Basiskart som vektordata utvalg (localhost test)	WI99992350 Planutsnitt med bestemmelser	X
<input type="checkbox"/> WI99996000 Basiskart som vektordata utvalg (Oslo test)		
<input type="checkbox"/> WI99990301 Bestille kart		
<input type="checkbox"/> WI99990011		

Lagre, neste steg

Figure 51 Final result of the relations feature

```

<div class="wizard-content-relations__left-bar-search">
  <input type="text" class="wizard-content-relations__search-bar" placeholder="Søk...">
</div>

<div class="wizard-content-relations__left-bar">
  <ul class="wizard-content-relations__check-all">
    <li>
      <label class="ambita-support-modal__checkbox-container"> Velg alle
      <input type="checkbox" ref="checkAll" onclick="{ checkAll }">
      <span class="ambita-support-modal__checkbox-checkmark"></span>
    </li>
  </ul>
</div>

<div class="wizard-content-relations__col-left">
  <ul class="wizard-content-relations__products">
    <li each="{ availableProducts }">
      <label class="ambita-support-modal__checkbox-list-container">
        <linked-code code="{ this }" link-prefix="#/products/">
        <input type="checkbox" class="wizard-content-relations__item" checked="{ isChecked() }" onclick="{ add }" name="c
        <span class="ambita-support-modal__checkbox-list-checkmark"></span>
      </label>
    </li>
  </ul>
</div>

```

Figure 52 Final code of the relations feature part 1

```

<div class="wizard-content-relations__right-bar">
  <label class="wizard-content-relations__done" ref="counter" style="overflow: none">Valgt ({ counter() })</label>
  <div class="wizard-content-relations__remove-all" onclick="{ removeAll }"> Fjern Alle</div>
</div>
<div class="wizard-content-relations__col-right">
  <ul class="wizard-content-relations__products">
    <li each="{ selectedProducts }">
      <label class="ambita-support-modal__checkbox-list-container">
        <linked-code code="{ this }" link-prefix="#/products/">
        <input type="checkbox" class="wizard-content-relations__item" ref="checkboxes" onclick="{ remove }" name="checkboxes" />
        <span class="wizard-content-relations__remove">X</span>
      </label>
    </li>
  </ul>
</div>

<button class="ambita-support-modal__button-next">Lagre, neste steg</button>

```

Figure 53 Final code of the relations feature part 2

The code snippets in Figure 52 and 53 show what the relations user story is made of. As mentioned earlier different containers had to be made to make this work. So how did we get the products? We used **linked-code** to make this work by calling an element **link-prefix="#/products/"** this was not made by us and already existed. Got the information from Scrum master that we could do it like that. Each of these containers are listing different products. One is listing **availableProducts** and the other one lists **selectedProduct**. These are explained more in detail what they are in chapter [5.3.6](#)

The code snippets below is the scss that was written to make the relations user story.

```

.wizard-content-relations {
  &__col-left {
    position: absolute;
    top: 180px;
    width: 400px;
    height: 450px;
    outline: 1px solid gray;
    background-color: white;
    overflow: auto;
  }

  &__col-right {
    position: absolute;
    width: 395px;
    height: 490px;
    top: 140px;
    left: 400px;
    outline: 1px solid gray;
    background-color: white;
    overflow: auto;

    label {
      display: block;
      height: 40px;
      padding-left: 0;
    }
  }
}

```

Figure 54 styles.scss of the relations part 1

```

&__check-all {
  list-style-type: none;
  padding: 0;
  margin: 0;

  li {
    border-bottom: 1px solid #ddd;
    padding: 10px;
  }
}

&__products {
  list-style-type: none;
  padding: 0;
  margin: 0;

  li {
    padding: 10px;

    &:hover:not(.header) {
      background-color: #eee;
    }
  }
}

&__remove {

```

Figure 55 styles.scss of the relations part 2


```

&__remove {
  position: absolute;
  top: 10px;
  left: 350px;
  cursor: pointer;
  font-size: 18px;
  color: gray;
  display: inline-block;
}

&__remove-all {
  position: absolute;
  top: 10px;
  left: 320px;
  cursor: pointer;
  font-size: 14px;
  color: rgb(0, 140, 255);
  display: inline-block;
}

&__remove-all:hover {
  text-decoration: underline;
}

&__search-bar {
  width: 100%;
  height: 100%;
  font-size: 12px;
  padding: 0 10px;
  border: 0;
}

```

Figure 56 styles.scss of the relations part 3

5.2.8 The ordering

The ordering wizard was implemented in an identical way as the identity wizard. It has radio boxes, a checkbox and a dropdown menu that allow the user to choose between schemas. It also strays away from the original user story design, and instead uses the vertical style for the radio boxes instead of a horizontal layout.

The ordering tab is not properly finished, since no API-endpoints were available. Due to slow pull request approval the APIs were made but never implemented to the front end. Since the approval and merging came later. Hence the same results as the **delivery** tab. The radio buttons are vertically instead of horizontally to save space. Then again there is no special logic here, only a bunch of different input types and a select tag.

The ordering wizard:

Endre produkt

Godkjente bygningstegninger
Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Produktkode:	W00000000
Leverandør:	K000000000
Kostpris:	116,-
Pris NET:	180,-
Pris VAT:	200,-
Status:	Aktivt

Bestilling

Påkrevet informasjon ved kjøp av produktet

☐ Matrikkel
☐ Boret
☐ Forretningsfører
☐ Geografisk utsnitt
☐ Dokument
☐ Province

☐ Benytt standardskjema for påkrevet informasjon

Skjema
Default matrikkel

Lagre, neste steg

Figure 57 Final result of the ordering wizard

The ordering wizard view.tag:

```
<div class="ambita-support-modal_block">
  <label class="ambita-support-modal_checkbox-container"> Benytt standardskjema for påkrevet informasjon
    <input type="checkbox">
    <span class="ambita-support-modal_checkbox-checkmark"></span>
  </label>
</div>

<div class="ambita-support-modal_block">
  <p class="bold">Skjema</p>
  <div class="wizard-content-order_container">
    <select class="width">
      <option value="#">Default matrikkel</option>
      <option value="#">test1</option>
      <option value="#">test2</option>
    </select>
  </div>
</div>

<button class="ambita-support-modal_button-next">Lagre, neste steg</button>
```

Figure 58 Final code of the delivery feature

Due to no API available, the values were hardcoded. If it were not hardcoded it would have looked something like the code snippet in figure 59. Using **Skjema** as an example:

```

<div clas="wizard-content-delivery__container">
  <select name="orderAvailability" class="width">
    <option each="{ orders in orderAvailability }"
      value="{ orders.code }">
      { orders.name }
    </option>
  </select>
</div>

```

Figure 59 Example code 2

It does the same thing as the **purchaseType**. It iterates through **orderAvailability** which is an array then creates options in the select tag based on the **orders.name**

The ordering styles.scss:

```

.wizard-content-order {
  &__container {
    width: 250px;
    overflow: hidden;
  }
}

```

Figure 60 styles.scss of the delivery

5.2.9 The executives

The executives wizard:

Figure 61 Final result of the executives feautre

This looks very much like the relations. That is because the only difference is some minor changes. Caseworkers were listed instead of listing products. The reason for this feature

being partially finished were because of we did not know which API to use to get these executives. There was a mutual decision to just focus on the relations feature. Then finish this feature later.

The executive wizard view.tag:

```
<div class="wizard-content-case-worker__left-bar-search">
  <input type="text" class="wizard-content-case-worker__search-bar" placeholder="Søk...">
</div>

<div class="wizard-content-case-worker__left-bar">
  <ul class="wizard-content-case-worker__check-all">
    <li>
      <label class="ambita-support-modal__checkbox-container"> Velg alle
      <input type="checkbox" ref="checkAll" onclick="{ checkAll }">
      <span class="ambita-support-modal__checkbox-checkmark"></span>
      </label>
    </li>
  </ul>
</div>

<div class="wizard-content-case-worker__col-left">
  <ul class="wizard-content-case-worker__products">
    <li>
      <label class="ambita-support-modal__checkbox-list-container"> test
      <input type="checkbox" class="wizard-content-case-worker__item" ref="checkboxes" checked="{ isChecked() }" o
      <span class="ambita-support-modal__checkbox-list-checkmark"></span>
      </label>
    </li>
  </ul>
</div>
```

Figure 62 Final code of the executive feature

A checkAll function was made, this was used to check all the executives. The checkAll function was called in view.tag to use it on click. More about how the different functions were made, can be read in chapter 5.3.6. The executives styles.scss file is very similar to that of the relations wizard. It has two container with multiple nested containers inside for its content.

5.2.10 The price

The price wizard could not be implemented as the user story for this was given to us relatively late of the semester. Development process was also further delayed by the coronavirus pandemic.

5.2.11 The parameters

The parameters wizard had already been implemented. The product owner and scrum master requested that we just move it over to the new modal. The design stayed original, and it still had the duplicate functionality as before. It allows the user to add, edit, delete and save a parameter, as well as add or remove tags.

Endre produkt

Godkjente bygningstegninger
Tegninger som viser bygningen slik den ble godkjent. Hentes fra byggesaksarkiv. Ja!

Produktkode: W00000000
Leverandør: K20000NFOLAND
Kostpris: 110 -
Pris NET: 160 -
Pris VAT: 200 -
Status: Aktiv ☒

Parametere

Navn	Verdi	
AxaptaAccount	3002	<input type="button" value="Slett"/> <input type="button" value="Lagre"/>
AxaptaProductGroup	1221	<input type="button" value="Slett"/> <input type="button" value="Lagre"/>

Tags

Lagre, neste steg

Figure 63 Final result of the parameters feature

```
<wizard-content-parameter class="wizard-content-parameter">
  <h3>Parametere</h3>

  <div class="ambita-support-modal_block">
    <product-edit-parameters-editor/>
  </div>

  <div class="ambita-support-modal_block">
    <p class="bold">Tags</p>
    <div class="wizard-content-parameter_json-editor-container" ref="jsonEditor"></div>
  </div>

  <button class="ambita-support-modal_button-next">Lagre, neste steg</button>

  <script>
    import '.././../editParameters/editor';
    import './styles';
    import { init } from './controller';

    init(this);
  </script>
</wizard-content-parameter>
```

Figure 64 Final code of the parameters feature

The parameters styles.scss

```
.wizard-content-parameter {  
  table {  
    width: 100%;  
  
    thead {  
      th:last-child {  
        width: 22%;  
      }  
    }  
  
    tbody {  
      input {  
        width: 94%;  
      }  
    }  
  }  
  
  &_json-editor-container {  
    .ambita-json-editor__header,  
    .ambita-json-editor__control-label {  
      display: none;  
    }  
  }  
}
```

Figure 65 styles.scss of the parameter

5.2.12 The authorizations

The authorizations wizard could not be implemented as the user story for this was given to us relatively late of the semester. Development process was also further delayed by the coronavirus pandemic.

5.2.13 The approval

The approval wizard could not be implemented as the user story for this was given to us relatively late of the semester. Development process was also further delayed by the coronavirus pandemic.

5.3 Backend Development

5.3.1 API

Application programming interface, also known as API, is a way to let products and services communicate with each other without having to know how they are implemented.

(Redhat.com, 2020) An API is the messenger that takes requests, and tells the system what to get. Then the messenger will wait for a response from the server, which then you will ultimately get. Think of an API as a waitress in a restaurant. You are presented a menu full of items, and the kitchen is the provider who will fulfill your order. But how do you get what you want? That is where the API comes in. The API in this example is the waitress. The waitress is the missing link in this communication. The waitress will bring your order (request) to the kitchen then deliver the food (response) to you. (mulesoft.com, 2020) From here on Application programming interface will be referred as API.

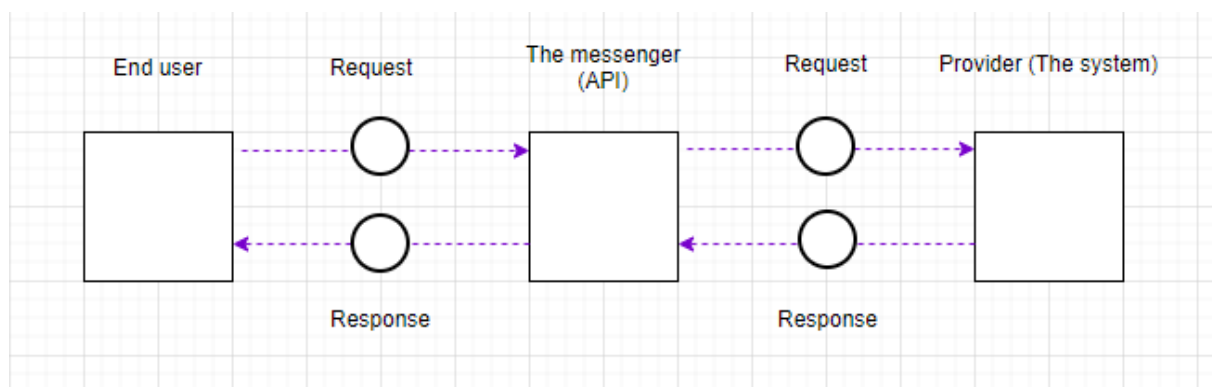


Figure 66 Figure of how API works

During our time working with Project Phoenix, there was some APIs that had to be created. These APIs were not created because the company didn't have any use for them until now. There was no API for listing: **product types**, **price types**, **delivery types** and others which will be mentioned later in this chapter. There was a total of ten API endpoints that had to be created, they were created like the others, following the similar code structure and style, with an exception for one. Which will be mentioned later.

When making API endpoints, eight classes had to be made for each API endpoints. An example of how the API for product types was made:

The classes that had to be made was:

Api/ProductTypes.java

Api/ProductType.java

LogisticsProductTypeMapper.java

ProductTypeService.java

LogisticsProductTypeController.java

Models/ProductType.java

Two classes for testing:

LogisticsProductTypeServiceTest.java

LogisticsProductTypeControllerTest.java

Two classes that had to be modified:

/mapping/Metamapper.java

Logistics.routes

In this class the only thing that was written was the types of the data which you could find in the table (See figure 68). In figure 67 there is a clear representation of what the class should contain. These are the fields we want to expose from the API.

Api/ProductType.java:

```
import ...

@ApiModel("ProductType Object")
@ArgsConstructor
public class ProductType extends Item {

    @ApiModelProperty
    public Long id;

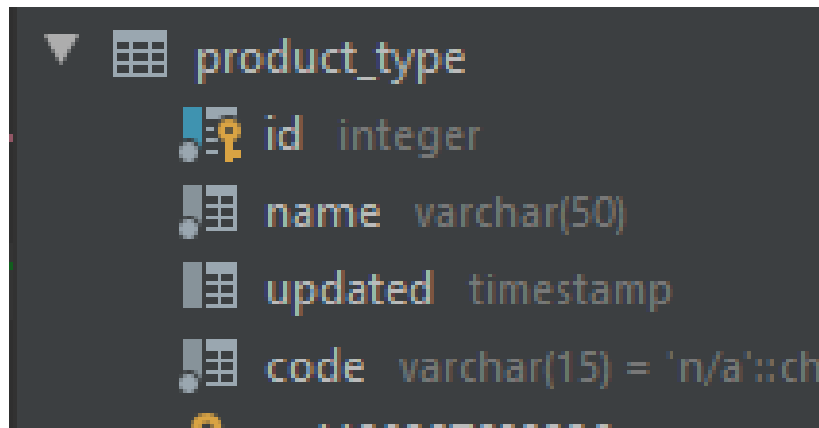
    @ApiModelProperty
    public String name;

    @ApiModelProperty
    public String code;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonSerialize(using = LocalDateTimeSerializer.class)
    @JsonFormat(shape = STRING, pattern = API_DATE_TIME_FORMAT_ZONED)
    @ApiModelProperty
    public LocalDateTime updated;

}
```

Figure 67 Figure of /api/ProductType.java



product_type	
id	integer
name	varchar(50)
updated	timestamp
code	varchar(15) = 'n/a'::ch

Figure 68 Figure of product_type database table

Api/ProductTypes.java:

```
package com.ambita.productcatalog.api;

import ...

@ApiModel("List of product types")
@JsonPropertyOrder({"meta", "productTypeList"})
@ArgsConstructor
public class ProductTypes extends Item {

    @ApiModelProperty
    public List<ProductType> productTypeList;

    @ApiModelProperty
    public Meta meta;
}
```

Figure 69 api/ProductTypes.java

Unlike the other API class, this class contains the list where we store the exposed fields of the API. So that the API user does not get more information than the user should. As shown in the figure 69 there is a swagger notation “@ApiModel” and inside the brackets there is a string: “list of product types”. The swagger notation “@ApiModel” provides the programmer the choice of describing the class (docs.swagger.io, 2015). Taking this description of the swagger notation to account, the class was described as “list of product types”.

The mapper classes are used to expose only the fields we want those who use the APIs to see. In figure 69 displays a function called **mapToProductTypes** is creating an object of type **ProductTypes** by using the api class: **api/ProductTypes.java**, and then use the list **productTypeList** as shown in figure 69. You can see that in figure 68 there is a list called **productTypeList**. It uses stream to convey elements from a source like an array, a data structure or an I/O channel. (docs.oracle.com, 2020) Then add it to the collection using the Class “Collectors” and its built-in method name “toList”.

LogisticsProductTypeMapper.java

```
public class LogisticsProductTypeMapper {

    private final MetaMapper metaMapper;

    @Inject
    public LogisticsProductTypeMapper(MetaMapper metaMapper) {
        this.metaMapper = metaMapper;
    }

    public ProductType mapToProductType(models.ProductType productType) {
        ProductType apiProductTypes = new ProductType();
        apiProductTypes.name = productType.name;
        apiProductTypes.id = productType.id;
        apiProductTypes.code = productType.code;
        apiProductTypes.updated = productType.updated;

        return apiProductTypes;
    }

    public ProductTypes mapToProductTypes(PagedList<models.ProductType> productTypes,
        FilterParameters parameters) {
        ProductTypes apiProductTypes = new ProductTypes();
        apiProductTypes.productTypeList = productTypes.getList().stream()
            .map(t -> this.mapToProductType(t)).collect(Collectors.toList());
        apiProductTypes.meta = metaMapper.mapToLogisticsProductTypeListMeta(parameters, productTypes);

        return apiProductTypes;
    }
}
```

Figure 70 LogisticsProductTypeMapper.java

ProductTypeService.java:

```
public class ProductTypeService {

    private static final String TYPE_NOT_FOUND_MESSAGE = "Could not find type with name: ";
    private LogisticsProductTypeMapper logisticsProductTypeMapper;

    @Inject
    public ProductTypeService(LogisticsProductTypeMapper logisticsProductTypeMapper) {
        this.logisticsProductTypeMapper = logisticsProductTypeMapper;
    }

    public ProductTypes getProductTypes(FilterParameters parameters) {
        PagedList<ProductType> productTypes = ProductType.findBy(parameters);

        return logisticsProductTypeMapper.mapToProductTypes(productTypes, parameters);
    }

    public com.ambita.productcatalog.api.ProductType getProductType(String productName) {
        ProductType productType = ProductType.findByName(productName);

        if (productType == null) {
            throw new NotFoundException(TYPE_NOT_FOUND_MESSAGE + productName);
        }

        return logisticsProductTypeMapper.mapToProductType(productType);
    }
}
```

Figure 71 ProductTypeService.java

The service class is the class that is organizing everything that is going to be done. This includes input validation, update of the database and mapping to the correct response. It updates the database by calling the functions in model class, see figure 73. This will be covered later. The Service class also maps to the correct response by calling the functions in the mapper class. As shown in figure 71 both the functions call the method **mapToProductTypes** from the class **LogisticsProductTypeMapper** (see figure 70). The bottom function is a function that will fetch a specific API response. It first calls a method from the **models/ProductType.java** class called **findByName**. (see figure 73) What this function **getProductType** does is that if the specific response does not exist, then it will be return an error message which is declared as **TYPE_NOT_FOUND_MESSAGE** including the name of the specific response the user tried to get. Then it will return the mapping results if there are any. If there is none, then it will return as empty.

LogisticsProductTypeController.java

```
@CorsComposition.Cors
@ResponseTimeLoggingComposition.ResponseTimeLogging
@ErrorHandlingComposition.ErrorHandling
@Api(value = "/productcatalog/v1/logistics/producttypes", produces = JSON, consumes = JSON, tags = "logistics")
@CheckToken(corsEnabled = true, scopes = ProductCatalogProperties.SCOPE_PRODUCT_CATALOG_READ)
public class LogisticsProductTypeController extends LogisticsBaseController {

    private ProductTypeService productTypeService;

    @Inject
    public LogisticsProductTypeController
        (AuthenticationService authenticationService, ProductTypeService productTypeService) {
        super(authenticationService);
        this.productTypeService = productTypeService;
    }

    @ApiOperation(
        value = "List all product types",
        notes = "Returns a paginated list of all product types",
        httpMethod = GET,
        responseContainer = "List",
        response = ProductTypes.class,
        nickname = "listProductTypes")
    @ApiResponses({
        @ApiResponse(code = UNAUTHORIZED, message = "Authorization header or session variable missing"),
        @ApiResponse(code = FORBIDDEN, message = "User is not allowed to list product types"),
        @ApiResponse(code = INTERNAL_SERVER_ERROR, message = "Internal server error")
    })
}
```

Figure 72 LogisticsProductTypeController.java

The function of the controller class is to receive HTTP call and then forward it to the services. It also functions as access controls to the APIs. The access control part is declared in the constructor of the class. **Super(authenticationService)** means that it inherits all the properties and behavior from the parent class. Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance is that you can create new classes built upon existing classes. (javatpoint.com, 2020)

HTTP calls are handled with the **@ApiOperation**. Inside the **@ApiOperation** scope there is an element called `httpMethod`. What this does is read what kind of 'method' it is. It will then return the response type which is **ProductTypes.class** (the api class). There are seven different acceptable values for HTTP 'method'. These are:

- GET
- HEAD
- POST
- PUT

- DELETE
- OPTIONS
- PATCH

Table 1 will explain what the different HTTP method does.

httpMethod	Function
GET	Requests that uses GET retrieves data. It requests a representation of the specified resource.
HEAD	The same as GET method, but instead of returning everything, it will return what the GET method returns but without its response body
POST	POST method replaces the current representation of a given resource, which will cause a change in state on the server
PUT	PUT replaces all the current representations of the target value with the request payload.
DELETE	DELETE method deletes the specified resource
OPTIONS	The OPTIONS method is used to describe the communications options for the target resource
PATCH	PATCH Is used to apply partial modifications to a resource

Table 1 Different HTTP methods

Table 1 was retrieved from (developer.mozilla, 2020).

Models/ProductType.java:

```

@Entity
@Table(name = "product_type")
public class ProductType extends Model {

    @Id
    @GeneratedValue
    public Long id;

    @Column(nullable = false)
    @NotEmpty
    public @Required String name;

    @Column(nullable = false)
    @NotEmpty
    public @Required String code;

    @Column
    public LocalDateTime updated;
}

```

Figure 73 models/ProductType.java part 1

The model class is a representation of the database table `product_type`. Which is a part of the eBean framework to write queries to the database easier. The model class contains the same values as the table in figure 68. That is because the model is a representation of the database table as mentioned above. eBean Query is used to get the resource we want from the database.

```
@Column(nullable = false)
@NonEmpty
public @Required String code;

@Column
public LocalDateTime updated;

private static final String SORT_DELIMITER = ",";
private static final Pattern SORT_PATTERN = Pattern.compile( regex: "-", Pattern.LITERAL);
private static final Finder<Long, ProductType> find = new Finder<>(ProductType.class);

public static ProductType findByName(String typeName) {
    return find
        .query().where()
        .eq( propertyName: "name", typeName.toUpperCase())
        .findOne();
}

public static PagedList<ProductType> findBy(FilterParameters parameters) {
    Query<models.ProductType> query = find.query()
        .setFirstRow(parameters.offset)
        .setMaxRows(parameters.limit);

    if (parameters.sort != null) {
        String[] params = parameters.sort.split(SORT_DELIMITER);
        for (String param : params) {
            String fieldName = SORT_PATTERN.matcher(param).replaceAll(Matcher.quoteReplacement( " "));
            String sortDirection = param.startsWith("-") ? " DESC" : " ASC";
            query.orderBy(fieldName + sortDirection);
        }
    }
    query.findList();
    return query.findPagedList();
}
```

Figure 74 models/ProductType.java part 2

In figure 73 a table that going to be used was declared as `@Table(name = product_type)` meaning that the table named `product_type` is the table that the model class wants to use.

The query in the function `findByName` is the same as:

SELECT id, code, name, updated

FROM product_type

WHERE name = UPPER(typeName)

LIMIT 1

But It does not say what to select. The finder object is declared as:

```
private static final Finder<Long, ProductType> find = new Finder<>(ProductType.class);
```

The eBean Finder uses the class as a base for implementations that can be injected or used as public static entity static fields on the associated entity. (ebean.io, 2020)

See figure 67 to see what kind of values that are implemented in ProductType.class.

Files that had to be modified:

/mapping/Metamapper.java

Logistics.routes

/mapping/Metamapper.java

```
public Meta mapToLogisticsProductTypeListMeta(FilterParameters parameters, PagedList<ProductType> productTypes) {  
    Meta meta = mapToMeta(parameters, productTypes);  
    meta.createLinks( path: "logistics/producttypes", productCatalogProperties.getSelfUrl());  
  
    return meta;  
}
```

Figure 75 /mapping/MetaMapper.java

The Metamapper class already existed. There were many existing methods and functions, we created a mapping function based on the others. The MetaMapper takes care of the pagination of the values based on the APIs offset, limit etc. which is declared in the Logistics.routes file. Pagination is like page numbers in a book, if too much information is published on one page, the user may get overwhelmed. The use of pagination is to present a lot of information in small manageable chunks (SEOptimizer, 2020).

Logistics.routes

```
# Product types  
GET          /productcatalog/v1/logistics/producttypes
```

Figure 76 Logistics.routes part 1

```
controllers.logistics.LogisticsProductTypeController.list(sort: String ?= null, offset: Integer ?= 0, limit: Integer ?= 10, callback ?= null)
```

Figure 77 Logistics.routes part2

In **Logistics.routes** the application route for the logistics producttypes was declared. The HTTP method GET is being used as shown in figure 76. When it's being used, we would like to call the function: `controllers.logistics.LogisticsProductTypeController.list`

This is what the MetaMapper takes care of: The offset, Limit, Callback and the sorting.

Now that all the files required to make the API were created, it was time to check whether the API worked or not. Hence the test classes had to be made. The API endpoints that was created:

- Product type
- Purchase type
- Delivery type
- Order availability
- System belongings
- Sales client
- Price type
- Trade type
- Input type
- Media type

Someone in Infoland already made the Sales client API. The API did not include all the files that we needed to get from the database. A new API of the same type had to be created.

5.3.2 API testing

When making the test classes, we needed two new classes. A test class for the controller, and a test class for the service. And then insert some data in the appropriate files. See figure 78 to 81.

Product-data.yaml:

```
productTypes:
  - &productType1 !!models.ProductType
    id: 1
    code: NORMAL
    name: Vanlig produkt
    updated: 2020-01-01 01:02:03
  - &productType2 !!models.ProductType
    id: 2
    code: PACKAGE
    name: Pakke
    updated: 2020-01-01 01:02:03
```

Figure 78 Product-data.yaml

First, values have to be fetched from the database. This can be done by looking in the database and pick the values. PostgreSQL was used to access the database. The resources from the database was chosen randomly, they would not be affected by the tests. These values were then put in Product-data.yaml file which already existed beforehand.

1.sql:

```
create table product_type (
  id          bigint not null auto_increment primary key,
  name        varchar(50) not null,
  code        varchar(15) not null,
  updated     timestamp,
);
```

Figure 79 1.sql part 1

```
drop table if exists product_type;
```

Figure 80 1.sql part 2

A test table was created in 1.sql file. Which is ultimately an SQL file which contained different tables. If the table already existed, we would 'drop' it, which means deleting the table.

Util.java:

```
Ebean.insertAll(all.get("productTypes"));
```

Figure 81 Util.java

The last thing that had to be done was to insert the **productTypes** entry from the product-data.yaml in Util.java. This was added to an already existing function called **loadDataForMemoryDB**.

The Test Classes

LogisticsProductTypeServiceTest.java:

```
@Test
public void getProductTypeTotal() {
    FilterParameters parameters = new FilterParameters();
    TokenClient tokenClient = new TokenClient();
    tokenClient.internal = true;

    ProductTypes productTypes = productTypeService.getProductTypes(parameters);
    assertEquals( expected: 2, productTypes.meta.totalCount);
    assertEquals( expected: 2, productTypes.productTypeList.size());
}

@Test
public void getTypesLimit() {
    FilterParameters parameters = new FilterParameters();
    TokenClient tokenClient = new TokenClient();
    tokenClient.internal = true;
    parameters.limit=1;

    ProductTypes productTypes = productTypeService.getProductTypes(parameters);
    assertEquals( expected: 2, productTypes.meta.totalCount);
    assertEquals( expected: 1, productTypes.productTypeList.size());
    assertNotNull(productTypes.meta.next);
}

@Test
public void getTypesOffset() {
    FilterParameters parameters = new FilterParameters();
    TokenClient tokenClient = new TokenClient();
    tokenClient.internal = true;

    ProductTypes productTypesFromZero = productTypeService.getProductTypes(parameters);
    parameters.offset = 1;
    ProductTypes productTypesFromOffset = productTypeService.getProductTypes(parameters);

    assertEquals(productTypesFromZero.productTypeList.get(1).name, productTypesFromOffset.productTypeList.get(0).name);
}

@Test
```

Figure 82 LogisticsProductTypeServiceTest.java part 1

```

@Test
public void getTypesSort() {
    FilterParameters parameters = new FilterParameters();
    TokenClient tokenClient = new TokenClient();
    tokenClient.internal = true;

    parameters.sort = "name";
    ProductTypes productTypesBeforeSort = productService.getProductTypes(parameters);
    parameters.sort = "-name";
    ProductTypes productTypesAfterSort = productService.getProductTypes(parameters);

    assertEquals(productTypesBeforeSort.productTypeList.get(0)
        .name, productTypesAfterSort.productTypeList.get(productTypesAfterSort.productTypeList.size() - 1).name);
}

@Test
public void getType() {
    ProductType productType = productService.getProductType( productTypeName: "PAKKE");
    assertNotNull(productType);
}

@Test(expected = NotFoundException.class)
public void getProductTypeNotFound() {
    ProductType productType = productService.getProductType( productTypeName: "NOT_FOUND");
}

```

Figure 83 LogisticsProductTypeServiceTest.java part 2

The tests in serviceTest class that was made:

getProductTypeTotal()

In this function we are using assertEquals, to check if the API returns the correct product type Total declared in product-data.yaml. If the expected value which is "2" is equal to the size of the list. The test result will give us a green light meaning the test passed.

getTypesLimit()

This test checks if it can return 1 value back since the value of limit is set to 1 in this test. It then checks if the next element is null or not. If its null, it will return an AssertionError.

getTypesOffset()

in this test there are two things to be aware of. The **productTypesFromZero** and **productTypesFromOffset** (see figure 82) in this test, two different values will check if the first object which is .value IDK

getTypesSort()

This test will check if the first element's name in the **productTypeList** is the same as the last element in the **productTypeList** after it is sorted. It will sort the list based on the names. (see figure 83).

getType()

getType() is a function that is trying to get a specific resource from the database. The resource the test wants to get is the name of a resource in capital letters. In this case it is "PAKKE"

getProductTypeNotFound()

This test will return the result "NOT_FOUND" if there is no such product type in the database. NB! This function is connected to the service class. See figure 71 to see the full function of **getProductType**.

LogisticsProductTypeControllerTest.java:

```
public class LogisticsProductTypeControllerTest extends BaseControllerTest {  
    private static final String BASE_URL = "/productcatalog/v1/logistics/producttypes";  
    private static final String USER_WITH_ACCESS = "120000BHA";  
    private static final String USER_WITHOUT_ACCESS = "K1201INFOLAND";  
  
    @Test  
    public void listOk() { assertEquals(OK, statusForRequest(GET, BASE_URL, putInternalClientTokenInCache(USER_WITH_ACCESS))); }  
  
    @Test  
    public void listUnauthorized() { assertEquals(UNAUTHORIZED, statusForRequest(GET, BASE_URL)); }  
  
    @Test  
    public void listForbidden() { assertEquals(FORBIDDEN, statusForRequest(GET, BASE_URL, putTokenInCacheForUser(USER_WITHOUT_ACCESS))); }  
}
```

Figure 84 LogisticsProductTypeControllerTest.java

This class tests the controller. It contains three functions:

listOK()

It will check if the variable: **USER_WITH_ACCESS** has access to the **BASE_URL** or not. This **BASE_URL** is the same one that was declared in **logistics.routes**

listUnauthorized()

This function will check whether the API user has access to elements from the database.

listForbidden()

it will check whether the user **K1201INFOLAND** has access to the **BASE_URL** or not.

All the APIs have been made. But how exactly do we access them from the front-end application? These APIs were made in productcatalog which belongs to another package. A pull request was made in bitbucket to merge this code to the master branch. When the merge was finished, we could then start writing the code in Ambita-support application to

get the API data. In the support application three files were made: (i) **types.ts** in the url folder (ii) **types.ts** in dataservices folder (iii) **Types.ts** in the api folder. The reason they were called “types” and not productType etc. are because of the repetition of the code, which made them look very much alike. Instead of making three different files times the number of APIs made, we made a single file.

Url/productcatalog/types.ts:

```
> services > url > productCatalog > types.ts > ...
1 import { url } from '../api';
2 import { LOGISTICS } from './base';
3
4 const LIMIT = '?limit=999';
5 const PRICE_TYPE_BASE = `${LOGISTICS}/pricetypes${LIMIT}`;
6 const PRODUCT_TYPE_BASE = `${LOGISTICS}/producttypes${LIMIT}`;
7 const TRADE_TYPE_BASE = `${LOGISTICS}/tradetypes${LIMIT}`;
8
9 export const knownPriceTypesListUrl = (): string => {
10   return url(PRICE_TYPE_BASE);
11 };
12
13 export const knownProductTypesListUrl = (): string => {
14   return url(PRODUCT_TYPE_BASE);
15 };
16
17 export const knownTradeTypesListUrl = (): string => {
18   return url(TRADE_TYPE_BASE);
19 };
```

Figure 85 url/productCatalog/types.ts

Trying to fetch the API route defined in **Logistics.routes** the reason that a limit is being added to the base url, is because in productcatalog we set the default limit was ten. If there is a database containing more than ten database values, then it would not be possible to list all of them. Hence adding the limit of 999. A constant containing the base url of the different types is made. The next thing that must be made is the api class for the types in the support application.

Api/productCatalog/logistics/Types.ts:

```
export namespace TypesApi {  
  export interface BaseType {  
    code: string;  
    id: number;  
    name: string;  
    updated: string;  
  }  
  
  export interface ProductType extends BaseType {  
    productList: TypesApi.ProductType;  
  }  
  
  export interface PriceType extends BaseType {  
    priceTypeList: TypesApi.PriceType;  
  }  
  
  export interface TradeType extends BaseType {  
    tradeTypeList: TypesApi.TradeType;  
  }  
}
```

Figure 86 api/productCatalog/logistics/Types.ts

In the api class we are defining the fields we want to get from the productcatalog. As shown in the figure above. Instead of making three different interfaces where we declare code, id, name and updated three times. There was a single interface that was made instead, so that the other interfaces could inherit that with the “extends” keyword. This will improve the quality of the code and will also look prettier. The reason for the “extends” keyword is because of the three API interfaces. The only thing that makes them different from each other is the typeList. These are the list from the productcatalog package.

The dataservices class can now be created since it depends on the imports of the api and the url class.

App/dataServices/types.ts:

```
dataServices / types.ts / ...
import { dataServices, Promise } from 'ambita-components-core';
import { TypesApi } from '~/typings/api/productCatalog/logistics/Types';
import { knownTradeTypesListUrl } from '~/services/url/productCatalog/types';
import { knownPriceTypesListUrl } from '~/services/url/productCatalog/types';
import { knownProductTypesListUrl } from '~/services/url/productCatalog/types';

export const getTradeTypeList = (): Promise<TypesApi.TradeType> => {
  return dataServices.xhr.getJSON(knownTradeTypesListUrl());
};

export const getPriceTypeList = (): Promise<TypesApi.PriceType> => {
  return dataServices.xhr.getJSON(knownPriceTypesListUrl());
};

export const getProductTypeList = (): Promise<TypesApi.ProductType> => {
  return dataServices.xhr.getJSON(knownProductTypesListUrl());
};
```

Figure 87 app/dataservices/types.ts

All the lists are being imported so that we can use them in dataservices file. Exporting constants were made since these have to be implemented in the front-end controllers for each page for them to work. Making an arrow function with the type

TypesApi.ProductTypeApi as shown in the figure 87. The dataservices class contains functions to get objects from the API through Ajax requests.

When making these classes a problem occurred. When trying to add the API endpoint for “system belongings”. It did not work as expected. The weird thing is that it worked for the other three API endpoints but not for the system belongings. But it then worked if it was a separate class instead. Our final solution was to let system belongings be it own class, so that it would continue to work.

5.3.3 Identity functionality

To display our API results in the front-end application, data has to be implemented in the appropriate controller.


```

import { RiotEvent } from 'ambita-components-core';
import { TypesApi } from '~/typings/api/productCatalog/logistics/Types';
import { SystemBelongingApi } from '~/typings/api/productCatalog/logistics/SystemBelongi
import { Tags } from '~/typings/app/Tags';
import { Products } from '~/typings/app/Products';
import productsStore from '~/stores/products';
import { getProductTypeList } from '~/dataServices/types';
import { getPriceTypeList } from '~/dataServices/types';
import { getTradeTypeList } from '~/dataServices/types';
import { getSystemBelongingList } from '~/dataServices/systemBelongings';

export interface WizardContentIdentityTag extends Tags.TagInterface {
  checkBoxStatus: HTMLInputElement;
  priceTypes: TypesApi.PriceType;
  product: Products.Product;
  productTypes: TypesApi.ProductType;
  systemBelongings: SystemBelongingApi.SystemBelonging;
  tradeTypes: TypesApi.TradeType;
  hasTradeType(tradeType: TypesApi.TradeType): boolean;
  onTradeTypeChange(event: RiotEvent): void;
}

```

Figure 88 The identity controller.ts code

We first have to make tags and declare what type they are. As shown in the figure 88 the API that was created is used in this class. The reason for this, is because the product requires that these types of field to be shown. Then the names in yellow are functions. **HasTradeType** is a function of the **TypesApi.TradeType** which returns a Boolean value. A Boolean value is either true or false (1 or 0). The other function is an event listener function Riot. Since this application is written in Riot.js and Typescript. This is exportable so that we can use the predefined functions and tags in our **init** function. See figure below.

```

export const init = (tag: WizardContentIdentityTag): void => {
  tag.product = productsStore.getProduct();

  getProductTypeList().then(productTypes => {
    tag.productTypes = productTypes.productTypeList;
    tag.update();
  });

  getPriceTypeList().then(priceTypes => {
    tag.priceTypes = priceTypes.priceTypeList;
    tag.update();
  });

  getTradeTypeList().then(tradeTypes => {
    tag.tradeTypes = tradeTypes.tradeTypeList;
    tag.update();
  });

  getSystemBelongingList().then(systemBelongings => {
    tag.systemBelongings = systemBelongings.systemBelongingList;
    tag.update();
  });

  tag.hasTradeType = (tradeType: TypesApi.TradeType): boolean => {
    const tradeTypes = tag.product.trades.filter(trade => trade.code === tradeType.code);
    return tradeTypes.length > 0;
  };
};

```

Figure 89 init function in the identity controller class

This **init** is called in the front-end to access the functions and tags that resides in this controller class. The four API that we declared earlier is being used here inside the **init**. The List of each API endpoint is being called by accessing the **TypesApi**. **NOTE! These functions already exist in the TypesApi**. See figure 87.

We then used **Promise.Prototype.then()** method. What it does is it returns a promise (developer.mozilla, 2020). A **promise** object represents the results of an asynchronous operation whether it failed or not. (developer.mozilla, 2020). instead of using **dispatch()** method, and declare other things within the scope. See figure 90. This was done easier and more efficient as shown in figure 89.

```
return (getConsumptionList.pending = dataServices
  .getConsumptionList(parametersOrHref)
  .then(consumptions =>
    dispatcher.dispatch({
      actionType: ACTIONS.GET_CONSUMPTION_LIST,
      consumptions
    })
  )
  .catch(errorAction(ACTIONS.GET_CONSUMPTION_LIST)));
```

Figure 90 Example code 3

Then the **update()** method was used so that the components state will be updated. Now the different APIs can be called to the frontend. This have been covered in the front-end chapter on how they were called.

Aktuelle bransjer	<input checked="" type="checkbox"/> Eiendom og Takst
	<input checked="" type="checkbox"/> Bank og Finans
	<input checked="" type="checkbox"/> Bygg og Anlegg
	<input type="checkbox"/> Offentlig forvaltning
	<input type="checkbox"/> Andre

Figure 91 checkbox states

In the screenshot above, there are 5 different elements from the API. These elements are static, but not the state of these elements. A product can have up to five checked checkboxes. Not all products have the same state, so a function had to be made to take care

of this little problem. Created a function named **hasTradeType** was created.

```
tag.hasTradeType = (tradeType: TypesApi.TradeType): boolean => {  
  const tradeTypes = tag.product.trades.filter(trade => trade.code === tradeType.code)  
  
  return tradeTypes.length > 0;  
};
```

Figure 92 The tag.hasTradeType function

What does this little bit of code do? These API endpoints that were created are in a form of an array of objects. An object can contain multiple fields. In the tradeType array, an object can have four different fields: Id, code, name and updated.

What this function does is taking in a parameter of type **TypesApi.TradeType** then it will return true or false depending on the parameter that was used. “**tag.product.trades.filter(...)**” is actually making a new array of an existing one. If the test passes a certain condition it will then be passed into our new array named **tradeTypes**. It will return the length if it is bigger than zero. if it was zero you wouldn’t need to check because there is nothing to check. Inside the **filter()** method **trade** is the name of iterable objects in the **tag.product.trades** array. It will check if the trade.code is the same as the parameters code. If the condition is true, the value will then be put in tradeTypes.

In the view.tag we have this:

```
<input name="branches" type="checkbox"  
value="{ tradeType.code }"  
checked="{ hasTradeType(tradeType) }"
```

Figure 93 Ccalling hasTradeType(tradeType) in view.tag

Inside the input tag in view.tag each input field will get a unique value which is the **tradeType.code** then there is the input element **checked**. If checked is true, the checkbox will be checked. The top checkbox is checked. The bottom one is unchecked, see figure 94.

☒ Bygg og Anlegg
☐ Offentlig forvaltning

Figure 94 Checked and unchecked checkboxes

The function that was made in the controller.ts class is being used here. Calling the function **hasTradeType(tradeType)** then use **tradeType** as a parameter, it will then check the inputs

tradeType if it's true or not. Since the function is only checking a parameter at a time. When we store it in an array and then the **return tradeTypes.length > 0** is because the **tradeTypes.length** can both be zero and one. If it is return $1 > 0$ then it will always return, since $1 > 0$ will always be true. So, all the checkboxes will be checked no matter what. Then this function is useless.

5.3.4 Description functionality

```
import { Tags } from '~/typings/app/Tags';
import Quill from 'quill';
import { Products } from '~/typings/app/Products';
import productsStore from '~/stores/products';

export interface WizardContentDescriptionTag extends Tags.TagInterface {
  longDescription: Quill;
  product: Products.Product;
  refs: {
    longDescription: HTMLDivElement;
    shortDescription: HTMLDivElement;
  };
  shortDescription: Quill;
}
```

Figure 95 The description controller.ts code

To implement the text formatting tool, Quill was used to do this. First, Quill has to be installed. Installing Quill can be done by writing this line in a git bash terminal:

npm install quill

After Quill has been installed. Import Quill to the project.

Inside the TagInterface tags were declared like what was done in the identity part. In the view.tag file there are two of these description boxes just like in the product description the product owner provided us, which is a tag for long and short description. Then there are two references to these description boxes named **longDescription** and **shortDescription** with the type of **HTMLDivElement**. Tags that we need are **Product.product** which is the API for product. The API is used for finding the product.code so that is possible to connect the correct description of each product.

Inside the **init** once again the tags are declared.

```
export const init = (tag: WizardContentDescriptionTag) => {
  tag.product = productsStore.getProduct();
  tag.on('mount', () => {
    tag.shortDescription = new Quill(tag.refs.shortDescription, {
      placeholder: 'Kort beskrivelse...',
      theme: 'snow'
    });
    tag.longDescription = new Quill(tag.refs.longDescription, {
      placeholder: 'Lang beskrivelse...',
      theme: 'snow'
    });
  });

  tag.on('unmount', () => {
    delete tag.longDescription;
    delete tag.shortDescription;
  });
};
```

Figure 96 init function in the description controller class

Quill requires a container where the editor will be appended. Either a CSS selector or a DOM object can be passed in (Chen, 2020). A DOM object was passed in which we made earlier. Refs = reference. Onto the configuration, a placeholder was put, and 'snow' theme was chosen. The options can be found at this site: <https://quilljs.com/docs/configuration>

```
tag.on('mount', () => {
```

Figure 97 Mounting tag

The mounting is to initialize the component upon runtime (Guarini, riot.js, 2020).

```
tag.on('unmount', () => {
  delete tag.longDescription;
  delete tag.shortDescription;
});
};
```

Figure 98 Unmounting the tags

Unmount is used to destroy the component and remove it from the DOM (Guarini, riot.js, 2020). How it was implemented in the frontend is already explained in chapter [5.2.5](#).

5.3.5 Delivery functionality

Because of slow pull requests, this was not possible to be done in time. But it can be done by doing the same thing as in the identity.

5.3.6 Relations functionality

One of the hardest controllers to make was the controller for relations.

```
export interface WizardContentRelationsTag extends Tags.TagInterface {
  availableProducts: ProductCatalog.ApiProductSupplier[];
  product: ProductApi.Product;
  refs: {
    check: HTMLInputElement;
    checkAll: HTMLInputElement;
    checkBoxes: HTMLInputElement;
    removeText: HTMLLabelElement;
  };
  selectedProducts: ProductApi.ProductBit[];
  add(event: RiotEvent): void;
  checkAll(): void;
  counter(): number;
  isChecked(): boolean;
  remove(event: RiotEvent): void;
  removeAll(): void;
}
```

Figure 99 WizardContentRelationsTag

The functions and tags that were used in this controller is shown in the code snippet above (figure 98). Here are some weird things, like **availableProducts** and **selectedProducts**. They belong to two different kind of objects which have different fields. Because of this you can't simply cast these two tags like this: **tag.availableProducts = tag.selectedProducts**. Because **selectedProducts** is missing a handful of fields.

Søk...	Valgt (2) (0)	Fjern Alle
<input type="checkbox"/> Velg alle	WI17190010	X
<input type="checkbox"/> WI17191073 Opplysninger om adgang til utleie	Eiendomsmeglerpakke	
<input type="checkbox"/> WI17191600 Ortofoto	WI42950070	X
<input type="checkbox"/> WI17196003 Plandata i vektorformat	Infopakke, samlet oppgave	
<input type="checkbox"/> WI17192990 Plandata i vektorformat		
<input type="checkbox"/> WI17197113 Restanser og legalpant		

Figure 100 Presentation of the relation boxes

In the view.tag. The table is divided into two parts. The left side is for the **tag.availableProducts** while the left side is for **tag.selectedProducts**. The selectedProducts are the objects that is already a part of the product. Which is usually a package. The available products are the products which you can add to the package. Selected product can be called using the product API. See figure 101.

```
export const init = (tag: WizardContentRelationsTag): void => {
  tag.product = productsStore.getProduct();

  tag.selectedProducts = tag.product.children;
```

Figure 101 init function of the relations class part 1

Available products can be called using organizationsStore api. See Figure 102.

```
organizationsActions.getProductList(tag.product.supplier.code);

const onProductsNotFound = (): void => {
  tag.availableProducts = [];
  tag.update();
};

const onProductsLoaded = () => {
  tag.availableProducts = organizationsStore.getProductList().filter(product => {
    return (
      !tag.selectedProducts.find(selectedProduct => selectedProduct.code === product.code) &&
      !(product.code === tag.product.code)
    );
  });
  tag.update();
};
```

Figure 102 init function of the relations class part 1

In the **tag.availableproducts** the code is written like that so that the duplicates do not appear on both side. Hence returning **!tag.selectedProducts.find(...)** which means, that it does not

return the same selected products. If the **selectedProducts.code** is the same as **product.code** then it will not display the product from **selectedProduct** array on the left side of the table. One more important thing that was done was also not listing the same product within the array of **availableProduct**. This was done by putting in an extra condition to the return value.

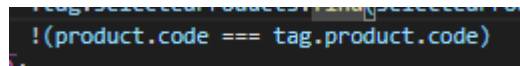


Figure 103 Value check

When the checkbox of a product is checked, it should move to the other side. Hence a tag function was made. A remove button is also required to move it from the selected products to available products again.

Add function:

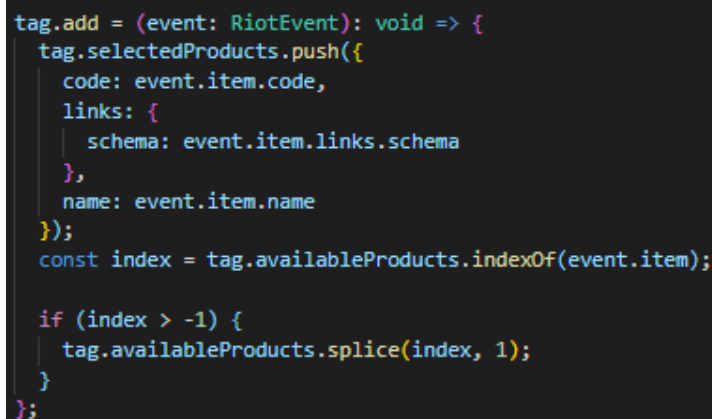


Figure 104 The relations: add function

We are using the array method push to put the value we want into the selectedProduct's array. We want only specific things, in this case there was a need of: code, name and links of the event.item. When these were added to the other side, the elements from the other array have to be removed. First of all, finding the index of the **event.item** can be done using **indexOf()** method. The returned value is stored in a variable named index. As long the index is not negative, we use splice from **tag.availableproducts** on the **index** and replacing/ removing 1 element.

Sometimes the user of this application regrets its decision of moving the product. So, they want to remove it from the list. Therefore, a remove method was also made. First thing that

was done was to find the index of **event.item** then the index of that **event.item** is then spliced, to remove it from **tag.selectedProducts**. Then sort **tag.availableProducts** and put it first in the list using **unshift()** method. See figure 105.

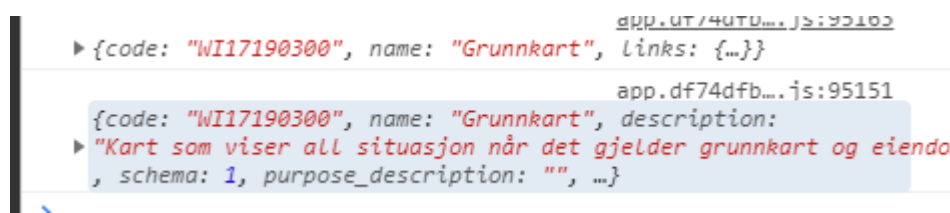
```
tag.remove = (event: RiotEvent): void => {
  const index = tag.selectedProducts.indexOf(event.item);

  if (index > -1) {
    tag.selectedProducts.splice(index, 1);
  }

  tag.availableProducts.sort();
  tag.availableProducts.unshift(event.item);
};
```

Figure 105 The relations: remove function

The hard thing was moving to the other side again, because it goes now from a smaller object with less fields, to a bigger object with more fields. When the remove button is clicked, the function will find the index, splice it from the current array. We sort the array, and then **unshift()** it is identical to the **push()** method but puts the element in front of the array instead. When you insert the item back from **selectedProduct**, you want the whole **event.item** and not only name, code, links. But all the fields. So, the easiest way was just remove it from the array, **unshift()** the **event.item** then calling **tag.availableProducts** again.



```
app.UT/4UTD...js:95103
▶ {code: "WI17190300", name: "Grunnkart", links: {...}}

app.df74dfb...js:95151
▶ {code: "WI17190300", name: "Grunnkart", description:
  "Kart som viser all situasjon når det gjelder grunnkart og eiendo",
  schema: 1, purpose_description: "", ...}
```

Figure 106 Google chrome console results

Figure 106 contains a screenshot from the console. The first bit is the **event.item** when removing from **tag.selectedProducts**, the other line are the same product added back again. As shown above, all the fields are back.

Sometimes the user chooses too many product, and do not know how many, a counter was therefore made. See figure 107 for counter function. This was made easily by finding the `tag.selectedProducts.length` then return the `sum`.

```
tag.counter = (): number => {
  let sum = 0;
  for (let i = 0; i < tag.selectedProducts.length; i++) {
    sum += 1;
  }

  return sum;
}
```

Figure 107 The relations: counter function

CHECK ALL and REMOVE ALL:

```
tag.checkAll = (): void => {
  tag.availableProducts.filter(products => {
    tag.selectedProducts.push({
      code: products.code,
      links: {
        schema: products.links.schema
      },
      name: products.name
    });
  });

  tag.availableProducts.length = 0;
  tag.update();
};

tag.removeAll = (): void => {
  tag.availableProducts = organizationsStore.getProductList().filter(product => {
    return tag.selectedProducts && !(product.code === tag.product.code);
  });

  tag.selectedProducts.length = 0;
  tag.update();
};
```

Figure 108 The relations: check all and remove function

Using the filter to put all values in the new array called products, then push the products.code, links and name into the `tag.selectedProducts` array. When everything has been done. We set “tag.availableProdcuts.length = 0” to empty the array. Then `update()` to update the DOM component. The Document Object Model (DOM) is a programming API for HTML and XML documents. The logical structure as well as the way the document is accessed is defined by the DOM (w3, 2020). DOM components are html tags, like divs, p and so on.

This cannot be done the same way as **checkAll** because of different types. What can be done instead? Simply list all the tag.availableProducts again, including **tag.selectedProducts**. Sometimes the dumbest solution is actually the wisest one.

5.3.7 Ordering functionality

Same as identity, because of slow pull request approval. The API for this task is not available. But it should be able to implement it the same way as in the identity controller

5.3.8 Price functionality

The price calculation database had to be moved so that it will be easier to maintain, and also stopped using AXAPTA. Microsoft Dynamics AX or formerly known as AXAPTA is a powerful Enterprise Resource Planning solution, that helps global enterprises organize, automate and implementation (Nair, 2020).

5.3.9 Parameters functionality

This was already created, so we didn't have to create it. Only move it to the folder.

5.3.10 Authorizations functionality

The authorizations functionality was provided but due to little time let working on the projects and the Coronavirus pandemic. This task also came late, due to product owner being busy.

5.3.11 Approval functionality

This task has not yet been provided therefore no results.

5.4 Final results

User Story	Objective	Frontend	Backend
The Modal	Easily open dialog box to create new or existing product, so that it is possible to edit one or many products within a short time	Done	Done
The Wizard	Develop a main frame for the user stories to be implemented including a navigation bar	Done	Done
The Identity	Develop a feature that can edit the identity of a product	Done	Done
The Description	Implement a wizard that can edit and setup products description	Done	Done
The Delivery	Develop a wizard that can setup and edit the current products delivery methods	Done	partially done
The Relations	Develop a wizard that can change the relations between products so that is possible to make different packages	Done	Done
The Ordering	Develop a wizard that can choose product type and ordering schema so the customers are able to fill out and order what they need	Done	partially done
The Executives	edit, display and select executives	Done	Done
The Price	Develop a wizard that can edit and add new product prices, as well as display the products history for price changes.	not done	not done
The Parameters	Re-implement the current parameter feature over to our new wizard.	Done	Done
The Authorization	create a wizard that can both edit and set the authority of the product.	not done	not done
The Approval	No information regarding this yet	not done	not done

Table 2 Final results of features

Seven of twelve of these features were able to finish as shown in Table 2. The reason for some of them are partially finished is because the APIs were made, but never implemented due to slow pull request approval. The Approval, Authorizations and The Price is not finished because of late update, and feature defining.

6 Discussion

6.1 Design Evaluation

The final designs of the features we implemented turned out to be on par or better than first requests from product owner. Our group managed to develop multiple features that looked as close to the user story design as possible. We were also very proud of the way it was structured, as this was something we learned along the way.

The project structure and code structure for each user story did not initially look that great. With each implementation and feedback from the development team, we adapted and improved our programming skills to meet their expectations. As a result, the finalized code structure for each feature we implemented looked much cleaner and more professional.

The visibility of the design was something we forgot to take into consideration, as the initial size of the modal was a bit small. It was later fixed to make sure it was easy for the eye, and comfortable to read through. This was something we accidentally stumbled across when implementing a feature that was too big to fit inside the wizard. In the end, the modal turned out great, and we were proud of with the results.

Accessibility was also important to us, and we made sure to make the modal as optimized and accessible for everyone. The reliability of the wizard was already taken into consideration by the product owner. Each design example for each feature was designed to be as reliable as possible. It was therefore important that we implemented these features as close to the design examples as possible.

We believe that each user will have an easy experience using the new wizard, whether it be editing or managing a product. Our group used everything we learned from web programming to our advantage. We made sure to develop a very reliable, accessible and optimized modal that is also easy to use.

6.2 Development process

The development of Project Phoenix had a slow start. We had to get used to the technologies and tools they used over at Ambita. With the guidance of the developers from team Infoland, our group eventually got the hang of it.

During the first weeks of development, our group focused solely on getting familiar with the current environment. We have had multiple projects while studying at Oslo Metropolitan University (OsloMet), but we never had any experience working with a large team. Tools such as Atlassian's Bitbucket and Jira software were something we have never used before, but we quickly adapted to it. This was also the case for the other tools used at Ambita, such as Slack and version control.

Agile development methods such as scrum were something we had only learned about during lectures. Experiencing it for the first time left quite the impression on us. It was a really solid way to coordinate and solve issues together with the team. We went on to use these meetings to report problems such as bugs or other technical issues.

Having experienced different programming languages, our group quickly got used to Riot.js and Play 2 Framework's syntaxes. We started with a simple button, then started implementing features as we progressed. The basics were important, so we started with those first. After a while, our group started working on the first user story.

We made sure to write a log for each implementation and progress done every single day. This was to make sure that we did not forget what we worked on and how it was done. This was sure to come in handy when working on future implementation. We also experimented a lot with scss and its features.

The development process was not as easy as we had thought at first. We experienced a lot of problems with certain features along the way. Some of these features also took days and countless hours of testing to get working. Thankfully, we received a lot of help from the Infoland team during the daily scrum meetings.

After having worked on the frontend part of the project, we started familiarizing ourselves with their API to work on the backend. Most of the codes inside the API had everything we needed to update, receive and remove data from the database. It was therefore advised to us that we just copied and pasted some of the lines of codes. This turned out to be extremely time consuming, so we decided to divide our manpower and split into two. One focusing on working with the API, while the other one focused on working with the frontend features. We made sure to write down our progress with detail so we could read what each of us had done.

After a few months into development, our group had become very efficient and familiar with working on the project. We used every tool and information to our advantage as we progressed through each user story. Every time we came across an issue, we took it up with the Infoland development team during scrum meetings.

We did not manage to finish every user stories requested by the product owner. However, this was not required by Ambita. Our group could decide which user stories to implement at the start of each Sprint. We also lost one week of work since we were not able to take our PC home with us during the pandemic. It also slowed down our progression and the amount of assistance we could get.

Despite not finish every user stories, our group enjoyed working on this project. The look, design and optimization turned out to be better than we expected. The product owner was also very pleased with the design of the project.

7 Conclusion

The development of the project started slow, but we picked up the pace after getting used to their tools and technologies. With each implementation, we learned something new and used it to further improve the features. Our group used all available tools to our advantage and became more proficient in using them. The modal and wizard we had developed exceeded our expectation.

We had developed a modal that was very reliable, accessible, optimized and easily usable by anyone. It could edit or manage the variables and data on a product. Even though we did not finish all the user stories, we came out with what looked like an exact replica of the example designs.

It was not always a downhill experience for us, as we did encounter some issues uphill. But with the help from the Infoland development team at Ambita, we managed to climb past it. Working together with their team and learning from them improved our skills as programmers.

The development of Project Phoenix, named after the bird of resurrection, was a very enjoyable learning experience for us. Working in a professional environment together with their development team taught us a lot. As a result, we ended up with a wizard that was visually appealing and satisfying to use.

8 References

- Amazon web services. (2020, 5 7). *aws.amazon*. Retrieved from [aws.amazon.com](https://aws.amazon.com/about-aws/):
<https://aws.amazon.com/about-aws/>
- Amazon web services. (2020, 5 7). *aws.amazon*. Retrieved from [aws.amazon.com](https://aws.amazon.com/blogs/aws/highly-scalable-mysql-compatible-rds-db-engine/):
<https://aws.amazon.com/blogs/aws/highly-scalable-mysql-compatible-rds-db-engine/>
- Atlassian. (2020, 5 22). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/software/jira):
<https://www.atlassian.com/software/jira>
- Atlassian. (2020, 5 22). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/software/jira/features):
<https://www.atlassian.com/software/jira/features>
- Atlassian. (2020, 5 22). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/software/confluence):
<https://www.atlassian.com/software/confluence>
- Atlassian. (2020, 5 22). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/software/confluence/features):
<https://www.atlassian.com/software/confluence/features>
- Atlassian. (2020, 5 19). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts):
<https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts>
- Atlassian. (2020, 5 24). *atlassian*. Retrieved from [atlassian.com](https://www.atlassian.com/agile/kanban/boards):
<https://www.atlassian.com/agile/kanban/boards>
- Atlassian. (2020, 6 22). *bitbucket*. Retrieved from [bitbucket.org](https://bitbucket.org/product/features/pipelines):
<https://bitbucket.org/product/features/pipelines>
- Atlassian. (2020, 5 22). *bitbucket*. Hentet fra bitbucket.org: <https://bitbucket.org/>
- Baghel, A. S. (2020, 5 19). *dzone*. Retrieved from [dzone.com](https://dzone.com/articles/software-design-principles-dry-and-kiss): <https://dzone.com/articles/software-design-principles-dry-and-kiss>
- Chen, J. (2020, 5 8). *quilljs*. Retrieved from [quilljs.org](https://quilljs.com/docs/configuration/): <https://quilljs.com/docs/configuration/>
- developer.mozilla. (2020, 5 8). *developer.mozilla*. Retrieved from [developer.mozilla.org](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then):
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
- developer.mozilla. (2020, 5 5). *developer.mozilla*. Retrieved from [developer.mozilla.org](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods):
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>
- developer.mozilla. (2020, 5 7). *developer.mozilla*. Retrieved from [developer.mozilla.org](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript):
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
- docs.oracle.com*. (2020, 3 12). Retrieved from [docs.oracle.com](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html):
<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>
- docs.swagger.io*. (2015, 4 1). Retrieved from <http://docs.swagger.io/swagger-core/v1.5.0/apidocs/swagger/annotations/ApiModel.html>
- ebean.io*. (2020, 4 9). Retrieved from <https://ebean.io/apidoc/11/io/ebean/Finder.html>
- edureka. (2020, 5 7). *edureka*. Retrieved from [edureka.co](https://www.edureka.co/blog/object-oriented-programming/): <https://www.edureka.co/blog/object-oriented-programming/>

Guarini, G. (2020, 5 8). *riot.js*. Retrieved from riot.js.org: <https://riot.js.org/api/>

Guarini, G. (2020, 5 7). *riot.js*. Retrieved from riot.js.org: <https://riot.js.org>

javatpoint.com. (2020, 5 6). Retrieved from <https://www.javatpoint.com/inheritance-in-java>

jetbrains. (2020, 5 7). *jetbrains*. Retrieved from jetbrains.com: <https://www.jetbrains.com/datagrip/features/>

jetbrains. (2020, 5 7). *jetbrains*. Retrieved from jetbrains.com: <https://www.jetbrains.com/idea/features/#built-in-developer-tools>

Microsoft. (2020, 5 7). *code.visualstudio*. Retrieved from code.visualstudio.com: <https://code.visualstudio.com/docs/editor/whyvscode>

mulesoft.com. (2020, 5 4). Retrieved from <https://www.mulesoft.com/resources/api/what-is-an-api>

Nair, M. (2020, 5 22). *synoptek*. Retrieved from synoptek.com: <https://synoptek.com/insights/it-blogs/what-is-microsoft-dynamics-ax-used-for/>

Oracle. (2020, 5 8). *developer.mozilla.org*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Play. (2020, 5 7). *playframework*. Retrieved from playframework.com: <https://www.playframework.com/documentation/2.8.x/Introduction>

Redhat.com. (2020, 5 4). Retrieved from Redhat.com: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>

sass-lang. (2020, 5 7). *sass-lang*. Retrieved from sass-lang.com: <https://sass-lang.com/>

scrum. (2020, 5 22). *scrum*. Retrieved from scrum.org: <https://www.scrum.org/resources/what-is-sprint-planning>

Scrum. (2020, 5 22). *scrum*. Retrieved from scrum.org: <https://www.scrum.org/resources/what-is-a-sprint-in-scrum>

Scrum. (2020, 5 22). *scrum*. Retrieved from scrum.org: <https://www.scrum.org/resources/what-is-a-daily-scrum>

Scrum. (2020, 5 22). *scrum*. Retrieved from scrum.org: <https://www.scrum.org/resources/what-is-a-sprint-review>

SEOptimizer. (2020, 5 22). *seoptimizer*. Retrieved from seoptimizer.com: <https://www.seoptimizer.com/blog/what-is-pagination/#What-use-pagination>

sparkfun. (2020, 5 19). *learn.sparkfun.com*. Hentet fra learn.sparkfun.com: <https://learn.sparkfun.com/tutorials/using-github-to-share-with-sparkfun/all>

The PostgreSQL Global Development Group. (2020, 5 7). *postgresql*. Retrieved from postgresql.org: <https://www.postgresql.org>

Tomagruppen. (2020, 5 19). *blogg.toma.no*. Retrieved from blogg.toma.no: <https://blogg.toma.no/hva-er-prop-tech>

tutorialspoint. (2020, 5 7). *tutorialspoint*. Retrieved from tutorialspoint.com: https://www.tutorialspoint.com/intellij_idea/index.htm

- tutorialspoint. (2020, 5 7). *tutorialspoint*. Retrieved from tutorialspoint.com:
https://www.tutorialspoint.com/typescript/typescript_overview.htm
- Veeraraghavan, S. (2020, 5 19). *simplilearn*. Retrieved from simplilearn.com:
<https://www.simplilearn.com/best-programming-languages-start-learning-today-article>
- Videos, M. (Regissør). (2015, 5 4). *What is an API?* [Film]. Hentet fra
<https://www.youtube.com/watch?v=s7wmiS2mSXY>
- w3. (2020, 5 7). *w3*. Retrieved from w3.org:
<https://www.w3.org/standards/webdesign/htmlcss.html>
- w3. (2020, 5 19). *w3*. Retrieved from w3.org: <https://www.w3.org/TR/WD-DOM/introduction.html>
- w3schools. (2020, 5 19). *we3schools*. Retrieved from w3schools.com:
https://www.w3schools.com/bootstrap/bootstrap_modal.asp