Bachelor’s thesis

Sigurd Espedalen Strøm
Sigurður Hallur Jónsson
Samuel Hardeberg

E-commerce in a distributed system of warehouses

May 2021

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

NTNU
Norwegian University of
Science and Technology

Bachelor’s thesis

2021

NTNU

Norwegian University of
Science and Technology

Sigurd Espedalen Strøm
Sigurður Hallur Jónsson
Samuel Hardeberg

E-commerce in a distributed system of warehouses

Bachelor's thesis
May 2021

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

NTNU
Norwegian University of
Science and Technology

Abstract

E-commerce is the process of selling and buying goods or services online. The main goal of this thesis is to create an e-commerce storefront that is able to communicate with a series of distributed warehouses. The storefront is to be made in Vue while the backend and database is made using Spring Boot and Postgres. To make the webpage and backend system communicate an API layer will also be created, which makes it possible to communicate solely through HTTP requests. The result of this thesis includes a distributed warehouse database system, along with an e-commerce storefront that will communicate through HTTP.

Sammendrag

E-handel er prosessen av å selge eller kjøpe varer eller tjenester over internett. Hovedmålet med denne oppgaven er å lage en e-handel nettside som kan snakke sammen med andre distribuerte varehus. Selve nettbutikken skal bli laget i Vue, og backend systemet skal bli laget i Spring Boot og Postgres. For at nettsiden og backend systemet skal kunne snakke sammen lages også en API-tjeneste, noe som muliggjør kommunikasjon gjennom HTTP forespørsler. Resultatet av denne oppgaven inkluderer et distribuert varehus database system, i tillegg til en nettbutikk som vil kommunisere ved bruk av HTTP.

Acknowledgements

We would like to express our deepest gratitude to our advisor Ricardo Da Silva Torres for all the guidance throughout the project and writing.

We would also like to thank Marius Lundbø for all the help and guidance with regard to the project and for setting up a meeting with Olaf Nykrem. The meeting with Olaf gave us a lot of insights, new ideas, and inspiration for developing our project and we would like to thank you for taking your time to help and inspire us. Thank you all.

Table of Contents

List of figures

Terminology

Glossary

E-commerce: Buying/Selling products and/or services electronically.

Warehouse: A location that stores goods to be distributed or sold later.

Full-Stack developer: Is a person who can develop both client and server side of a software.

Storefront: A storefront webpage is a webpage that displays products to potential customers.

Database: Collection of \structured data.

Abbreviations

HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
XML	Extensible Markup Language
NPM	Node Package Manager
JSON	JavaScript Object Notation
VUE	Virtual-User Environment
IDE	Integrated Development Environment
API	Application Programming Interface
REST	Representational State Transfer
MVC	Model View Controller
JPA	Java Persistence API

1 Introduction

1.1 Background

This e-commerce project was defined by DRIW AS. It focuses on the interaction between multiple warehouse applications and a storefront website. The project consists of warehouses with e-commerce and a storefront that allows users to shop goods on the website independently of where they are located. The main goal of this project is to make the storefront able to receive data from all the warehouses and keep stock updated, while showing the available stock on a per warehouse basis.

1.1.1 Why we chose this project

We chose this task since we wanted a project whose development would provide us the opportunity to use existing knowledge and skills, while also learn something new during its development. Everyone in the group is familiar with front-end web design and database programming, but we were still interested in learning and becoming aware of existing and new technologies the project requires, such as Postgres, Vue JS, and Spring Boot.

1.1.2 E-commerce storefront

The storefront can communicate with all the warehouses independently and to provide information about the stock on a per warehouse basis. The storefront also allows the end user to select which warehouse they want to purchase a product from. In the example databases we have created, we consider warehouses from Ålesund, Oslo, and Trondheim. The developed e-commerce system allows the user to add the products they want to a cart and choose between the three warehouses the items will be shipped from, depending on their availability.

1.1.3 Distributed warehouse database

The databases communicate using HTTP requests and the storefront “asks” the databases for the available stock on each item. After items have been added to the cart in the storefront and bought, the stock is decreased in the databases depending on which warehouses the user chose as the shipping points. The user is also able to put multiple items in a cart and set different shipping points per item. For instance, buying two products and choosing Ålesund for one of them and Trondheim for the other.

1.2 The problem to be solved

The main problem to be solved in this project is the communication and stock handling of the databases and storefront. We had to create an API layer that let all the communication happen through HTTP requests and let the warehouses work independently of each other. By doing it this way, we can easily add multiple warehouses with minimal effort. Each warehouse will hold its own stock and product catalog. This would also mean that if, for instance, the warehouse in Oslo is down for maintenance the other warehouses can keep working and take in orders as per usual.

1.3 Project group

Student #	Name	Phone #	E-mail
494669	Sigurdur Hallur Jonsson	98061047	Sigurdur.h.jonsson@ntnu.no
507877	Samuel Hardeberg	47224458	Samuel.hardeberg@ntnu.no
494679	Sigurd Strøm	97503611	Snstrom@ntnu.no

1.4 Goals

In this project, we are expected to design, implement, and validate an e-commerce system. This system consists of a backend database system and a commercial webstore front end.

1.4.1 Effect goals

The effect goal of this project is to design and implement an e-commerce website and distributed system of warehouses that solves the issue of communicating with multiple warehouses simultaneously while keeping stock up to date. Moreover, it should enable users to buy products online and choose which warehouse each of the products should be sent from. And check the stock for each warehouse.

1.4.2 Result goals

At the end of the semester, we will have a product consisting of a storefront website and a main backend warehouse application and API Layer. And secondary warehouse databases which holds inventory information. The storefront will be able to communicate will all the warehouses, while the warehouses themselves cannot communicate with each other. Thus, the website works as a storefront for all the warehouses. On the website, customers can browse products and get detailed information about them. Furthermore, registered and logged in users can add items to shopping cart to buy and choose which warehouse the item should be bought from.

1.4.3 Process goals

As developers, we expect to experience opportunities about how to work on a larger and more complex project. We also expect to develop our skills to work in an agile way. We expect to learn how to use frameworks like Spring Boot and Vue. Also, we will deepen our understanding of database systems.

2 Theory

2.1 Agile development

Agile development is a very broad term for a set of frameworks based on the values and principles of the agile manifesto. Summarized, the manifesto states several prioritizations in a software development context:

> *"Individuals and interactions over processes and tools,*
> *Working software over comprehensive documentation,*
> *Customer collaboration over contract negotiation,*
> *Responding to change over following a plan"*
> (Agile Manifesto Authors, 2021)

The principles and values of the agile manifesto ensures a streamlined and effective software development project. Continuous development, close customer communication and collaboration, and a sustainable development process are some of the agile principles that we have incorporated as a central component into our workflow. As previously mentioned, Agile development is realized through agile frameworks. For our project, we used the core elements of the Scrum framework.[1]

2.2 The Scrum framework

The scrum framework helps create an agile workflow, with emphasis on a high responsibility to change. Simply put, scrum ensures that the product adapts and perseveres trough unforeseen changes and circumstances. The scrum framework is structured around the scrum events, as illustrated in Figure 1 below.

[1] https://www.agilealliance.org/agile101/ as of May 2021

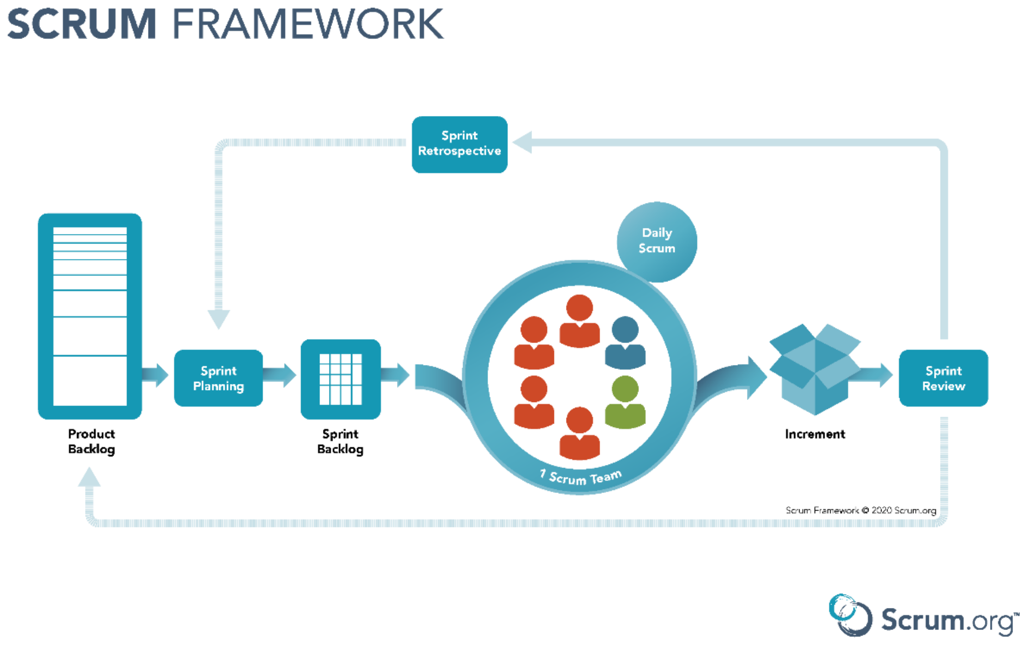


Figure 1: The Scrum Framework diagram (https://www.scrum.org/resource/scrum-framework-poster - As of May 2021).

The process starts by collecting a backlog of tasks/features in the product backlog. During a sprint planning session, a set number of tasks are added to a sprint backlog. This is the tasks that are implemented during the sprint. During the sprint, the Scrum team has daily scrum meetings every day. To visualize the progress, each task is given story points in relation to its difficulty, estimated time to complete and importance to the project. In our project, there has been a focus on 4 core scrum events: The backlog(s), sprints, daily scrum, and story points.

2.2.1 Backlog

The product backlog is the list of all tasks that need to be performed in the product development. The product backlog tasks are not associated with a time specification nor a particular order. This list can be changed throughout the development of the project. During sprint planning session, a number of these tasks is added to a sprint backlog. The number of tasks depends on how much the team members think they can finish during the 2-week sprint duration.

2.2.2 Sprint

During the sprints, the group completed the tasks in which we have added to the scope of the sprint. For instance, we could set a goal for each sprint and add the related tasks from the backlog in Jira, which is our platform for keeping track of issues, tasks, and bugs, as well as also for creating an overview of the progress.

2.2.3 Daily scrum

The group met every day at 09.00 - 14.00 and at the start of the day we used to discuss which task or issue the group member would work on. By having a small meeting each day, we could work seamlessly without overlap of the work that had been done.

2.2.4 Story points

Story points indicate what work has been done in the sprints in terms of the programming project. Our definition for story points is meaningful changes that the end user will make use of or are visual for the end user.

2.3 Git and version control

Version control systems help track and log changes in a project. This is particularly helpful in systems development, as trial, error, and correction are an integral part of the workflow.

The most common version control system, and the one we have used for our project, is GIT. GIT has the added feature of allowing the members of our team to diverge from the "master" version of the project, i.e., creating branches. This means that we can work simultaneously on the same project, as everyone is working only a local version of the project. Only when a project member is finished with a specific task, it is "merged" back to the *master* branch. GIT works locally, so for the GIT repository to be available to every project member it needs a web hosting service.

2.3.1 GitHub

GitHub is a GIT repository web hosting service we have used for our project. GitHub is exclusively cloud-based and specializes in shared projects. It has an intuitive graphical user interface which provides management tools.

2.4 Web development languages

2.4.1 Hyper Text Markup Language (HTML)

HTML is the standard markup language for web development, for formatting the text to be displayed in a webpage.[2] It is structured around layers of nested elements, where different elements represent different text types, such as a heading, paragraph, link, etc. Each element can also hold an attribute, which connects the element to a specific styling (CSS) and behavior (JS). HTML markup is stored in .html files.

2.4.2 Cascading Style Sheets (CSS)

CSS is used to define how HTML elements are displayed on a webpage.[3] It is used to create styling and layout for specific elements, trough HTML attribute tags. CSS is stored in .css files, referred to as stylesheets.

2.4.3 JavaScript (JS)

JavaScript is a programming language specified for web development.[4] JavaScript code is used to create behavior in the webpage, such as user interactions and animations. JavaScript code can also alter HTML element attributes and CSS styling. This functionality allows for a more dynamic website.

2.4.4 JavaScript Object Notation (JSON)

Only pure text (strings) can be delivered from a web server to a browser and vice versa. JSON is a syntax for storing JavaScript objects in pure a string for the main purpose of server-browser data exchange.[5]

The JavaScript object is "stringified" with "`JSON.stringify()`", thus creating a JSON string. The JSON string is then sent to its destination. It is then parsed back to a JavaScript object with "`JSON.parse()`".

[2] https://www.w3schools.com/html/html_intro.asp as of May 2021
[3] https://www.w3schools.com/css/css_intro.asp as of May 2021
[4] https://www.w3schools.com/js/js_intro.asp as of May 2021
[5] https://www.w3schools.com/js/js_json_intro.asp as of May 2021

2.4.5 PostgreSQL

PostgreSQL (also called Postgres) is a relational database management system that extends the SQL database language.[6] It is an open-source project. PostgreSQL has support for many popular programming languages, including java and JavaScript. It is classified as a "General purpose Transaction database"[7] which means that it is optimized to work with transactional data (purchasing orders, sales, shipping documents etc.), and fit with as many applications as possible.

2.5 Vue.js

Vue.js is a JavaScript framework used for building front-end user interfaces. [8] Vue differs from its competitors such as Angular[9] and React[10] in that it is much more lightweight and focuses on the development of user interfaces on only the view layer. Its simplicity is not a limitation, as there is a multitude of extensions and libraries for vue.js tailoring it to specific development cases. Vue.js is optimized for single-page applications.

2.5.1 Vue components

Vue components are nameable, reusable Vue instances – encapsulated html, CSS, and JavaScript to be used as custom html elements. Vue components are stored on ".Vue" files. Each Vue component is divided into three sections, <template>, <script>, and <style>:

```
<template> (HTML) </template>
<script> (JS) </script>
<style> (CSS) </style>
```

Template contains the HTML syntax of the Vue component, style holds the CSS styling of the component, and script holds the JavaScript code to be used in the template. It is in the script section that the *options* for the Vue instance are specified.

[6] https://www.postgresql.org/about/ As of May 2021
[7] https://www.postgresqltutorial.com/what-is-postgresql/ As of May 2021
[8] https://vuejs.org/v2/guide/ As of May 2021
[9] https://angular.io/ As of May 2021
[10] https://reactjs.org/ As of May 2021

2.5.2 Vue options

Name specifies the name of the Vue component. This must be the same as the filename.

Data contains the function which returns data objects. The data object returned will trigger updates in the markup. This is a core part of what makes Vue reactive. Its syntax is as following:

```
data: function() {
  return {
    data: "Some string"
  }}
```

This can also be written without “function()”. The relevant detail here is that whatever data is outputted, the html syntax needs to be explicitly stated in “return{}”.

Methods is where the functionality of the Vue component is defined. Methods can hold several functions in the following syntax:

```
  methods: {
    function1() {
        //Functionality 1
    }
    Function2() {
        //Functionality 2
    }
}
```

In order for these functions to output data to the html syntax, and thus be part of the Vue reactive system, the data to be outputted must be *returned* in the data function.

As previously mentioned, Vue operates through Vue instances. Each of these instances has a set lifetime, and a list of events that transpire during this lifetime. This is where the **Lifecycle hooks** Vue component option comes in. There are many lifecycle hooks attached to a complex lifecycle. Grossly simplified, there is a creation event, a mounting event, several update events (whenever data changes), and a destruction event. The hooks relate to these events by means of corresponding titles: *beforecreated*, *created*, *beforemount*, *mounted*, *beforeupdate*, *updated*, *beforedestroy*, and *destroyed*. Since the hooks are triggered at distinct stages, they can be used to execute code at different stages of the instance.

Props are custom attributes that are used to pass data to child components. They can also be used internally similarly to *return{}*, in the following syntax:

```
Vue.component(‘testComponent', {
  props: ['prop'],
  template: '<h1>{{ prop }}</h1>'
})
```

2.5.3 Vuex

Vuex is a state Management pattern and a library for Vue.[11] It enables the user to make a centralized store in which the user makes states, getters, actions, and mutations. This allows the user to call these functions and save the states from any component by using “this.$store” notation. As an example, we could make a state, which in this case will be an array, called products and make getters, actions, and mutations that would allow us to change the state “products” and add all the products in the store. In this case “axios.get” was used to make an API call to the database and fetch all the products and add them into the state “products.” Thus, we can make states that hold information, such as Cart, Products, User, searchProducts, and so on.

2.5.4 Vue router

Vue router is a way to traverse the website. Upon installation it will create a router folder with a routing index JavaScript file. In the routing file the user will specify which paths and one folder for “views.” Due to Vue being a single page application, the Vue router creates a “views” folder that represents pages in a traditional html structure. As an example, the login page and cart are views, and their path are defined in the router index file. When pressing on the user icon it will then route you to */login and swap out the view files to login page.

2.5.5 Node Package Manager (NPM)

NPM is a software registry that the user can use to keep track of dependencies and keep them up to date. The benefit of NPM is giving the user the ability to update the dependencies and letting a group work on the same project while downloading the same version of dependencies for all users.

2.5.6 Axios

Axios is a library that allows the user to create API calls using URLs. It is a simple to install JavaScript library that works seamlessly with Vue. It has some benefits over other JavaScript libraries, such as fetch, as Axios has some quality-of-life improvements, such as better error handling, and automatic transforms of JSON data.

[11] https://vuex.vuejs.org/ As of May 2021

2.6 Spring framework

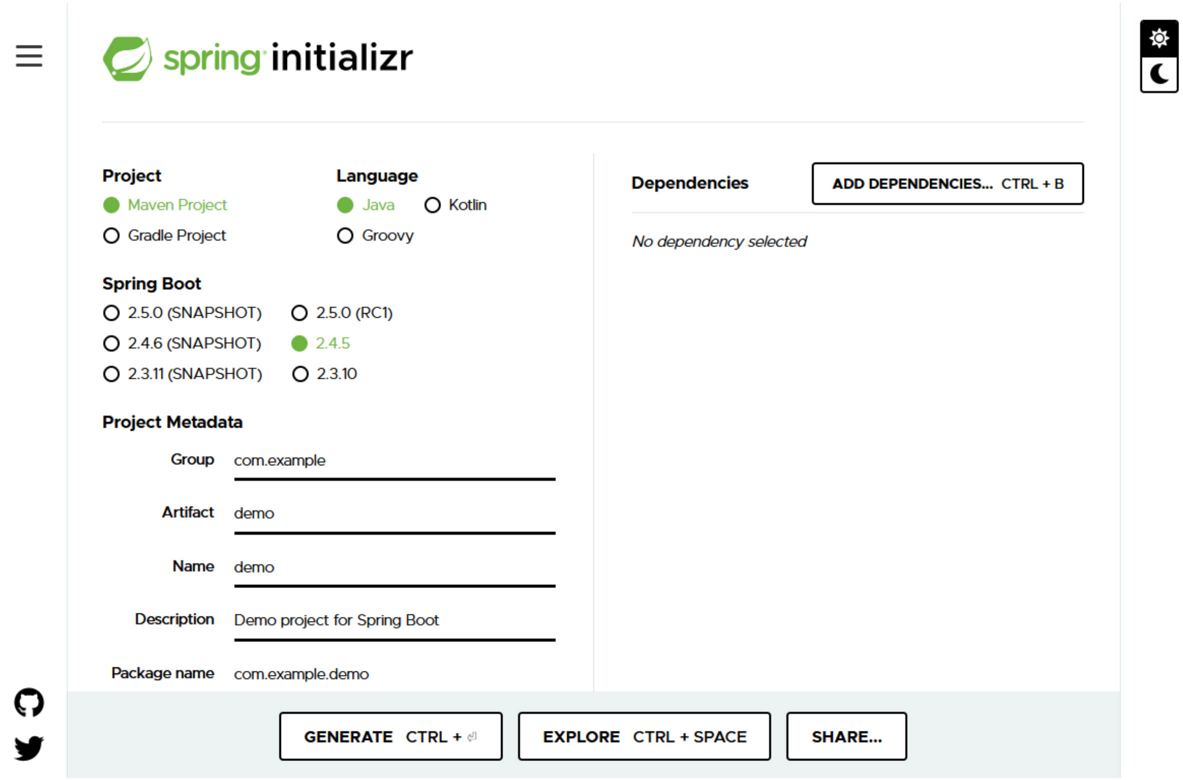
Spring framework is a popular open-source java application framework. Most integral to the framework, is its Inversion of Control (IoC) container, and Dependency Injection (DI). These technologies allow Spring framework to detach dependencies from the code, further abstracting it.[12]

2.6.1 Spring boot

Spring boot is a tool to create Spring applications, with an emphasis on simplicity. It automatically configures third party libraries, eliminates the need for any XML configuration, and provides some lesser features such as health checks and metrics.

2.6.2 Spring initializr

Spring initializr is a web tool that can generate a spring boot project structure.



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.5.0 (SNAPSHOT) ○ 2.5.0 (RC1)
○ 2.4.6 (SNAPSHOT) ● 2.4.5
○ 2.3.11 (SNAPSHOT) ○ 2.3.10

Project Metadata

Group com.example

Artifact demo

Name demo

Description Demo project for Spring Boot

Package name com.example.demo

Dependencies

No dependency selected

Figure 2: Spring initializr (https://start.spring.io - As of May 2021).

[12] https://spring.io/projects/spring-framework (As of May 2021).

It provides a graphical interface that lets you choose between a Maven and Gradle build specification and a choice between Java, Kotlin, and Groovy as the programming language. The app has a list of Spring and third-party dependencies that can be added. The web tool also allows the specification of the version of spring boot and the project metadata.

2.6.3 Monolithic architecture pattern

Monolithic architecture is when one executable application is created. An independent software that is responsible for all functionality and user interface are intertwined in one. One advantage of this implementation is shared resources. For example, it is hosted on one server and shares data via function call instead of http request. However, it can be difficult to maintain as the software scales up and gets more complex.

2.6.4 Microservice architecture pattern

Microservice architecture is when monolithic software is divided into smaller more manageable executables. Those smaller programs should have single responsibility. They are not dependent on languages, as they exchange messages via JSON or XML response.

2.6.5 RESTful API

API is an abbreviation for application programming interface. It is a set of rules allowing application to interact with each other. It will send data in a convenient format like JSON or XML. REST is also an abbreviation and stands for representational state transfer. REST is an architectural style that aims to make data presentation in a convenient way for the client.

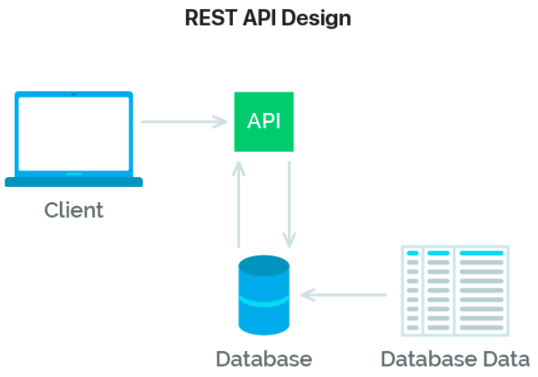


Figure 3: REST API Design (https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api – As of May 2021).

Four basic operations that RESTful API are to receive data in a convenient format, create new data, update data, and delete data.[13] As shown in Figure 3 client will send request to RESTful API that will then perform one of its operation.

[13] https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api – As of May 2021)

3 Method

3.1 Scrum Agile development

In the project, it was used some elements of scrum in our workflow rather than following scrum fully. This was a conscious decision, as scrum is optimized for larger project teams, which allows for more specialization. According to the scrum methodology, there should be a scrum master external to the project itself. Given that our project team consists of three members, this did not make sense to implement. We all collectively oversaw the use of the core scrum events.

3.1.1 Daily scrum

Every weekday morning at around 09:00, we held a daily scrum meeting remotely, where we delegated tasks for the day. This was very helpful to the project as it gave us a more structured workflow, and a specific time to start the workday.

3.1.2 Sprints

Our project timeline was divided into 2-week sprints in the proper scrum fashion, with a sprint meeting at the end of each sprint with the client to conclude the ongoing sprint, compare the projected progress with the actual progress, and plan the next sprint accordingly.

3.1.3 Jira

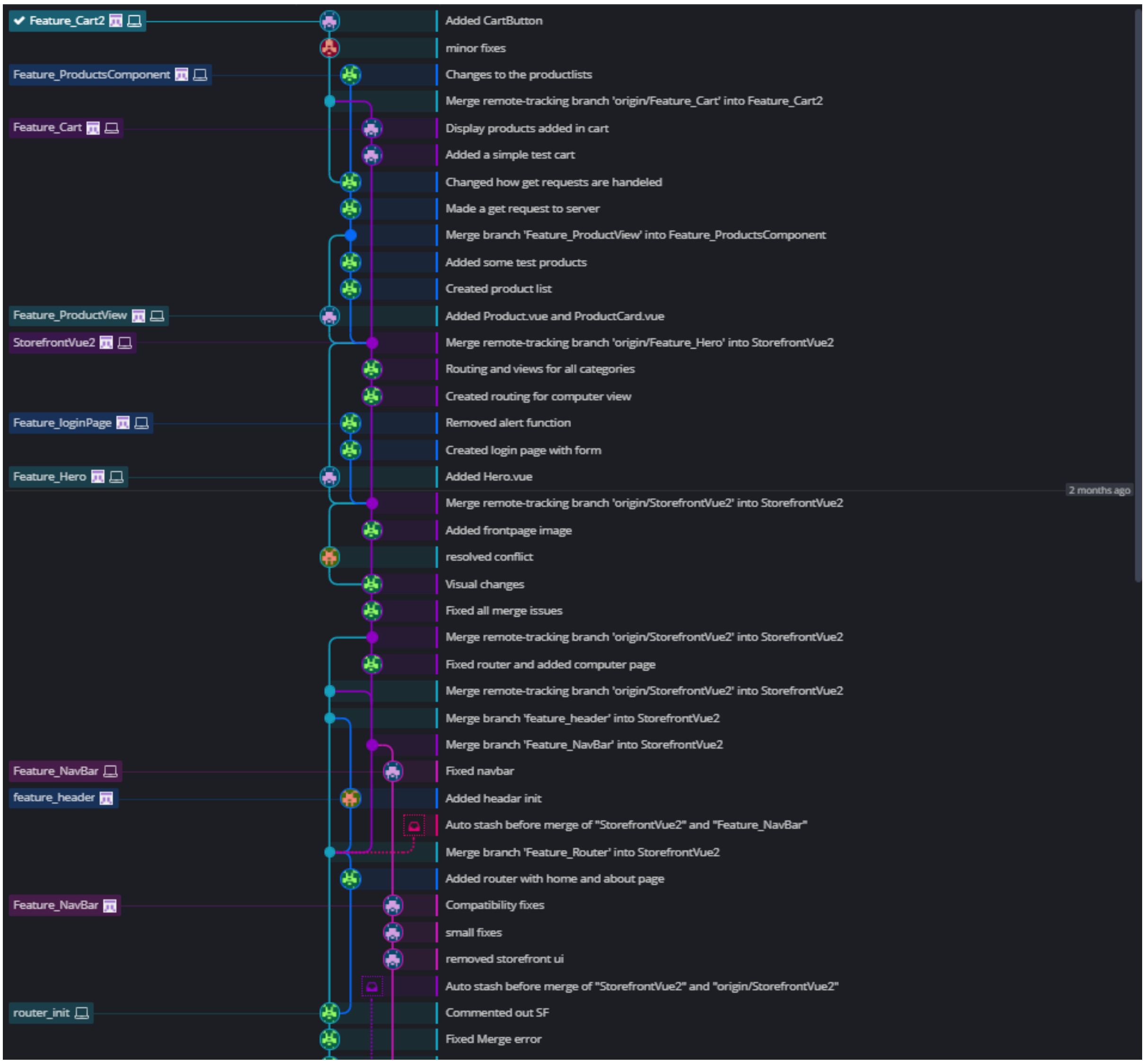
Jira, a scrum management tool, helped us get a clear overview of these scrum events. It allows us to make a board with split in three columns: “To Do”, “In Progress” and “Done”. By using Jira, we can also create a backlog of issues or tasks that needs to be done and have an overview of who reported it, and who is working on it. This makes it easier as a group to know what everyone else is working on and the overall progress being made on the project. Furthermore, we are able to get progress reports directly from Jira which makes it a very useful management tool.

3.2 Git

For the project, we mainly used two git repositories, one for the storefront and one for the warehouse database system. We used the Git graphical user interface GitKraken for our repositories.

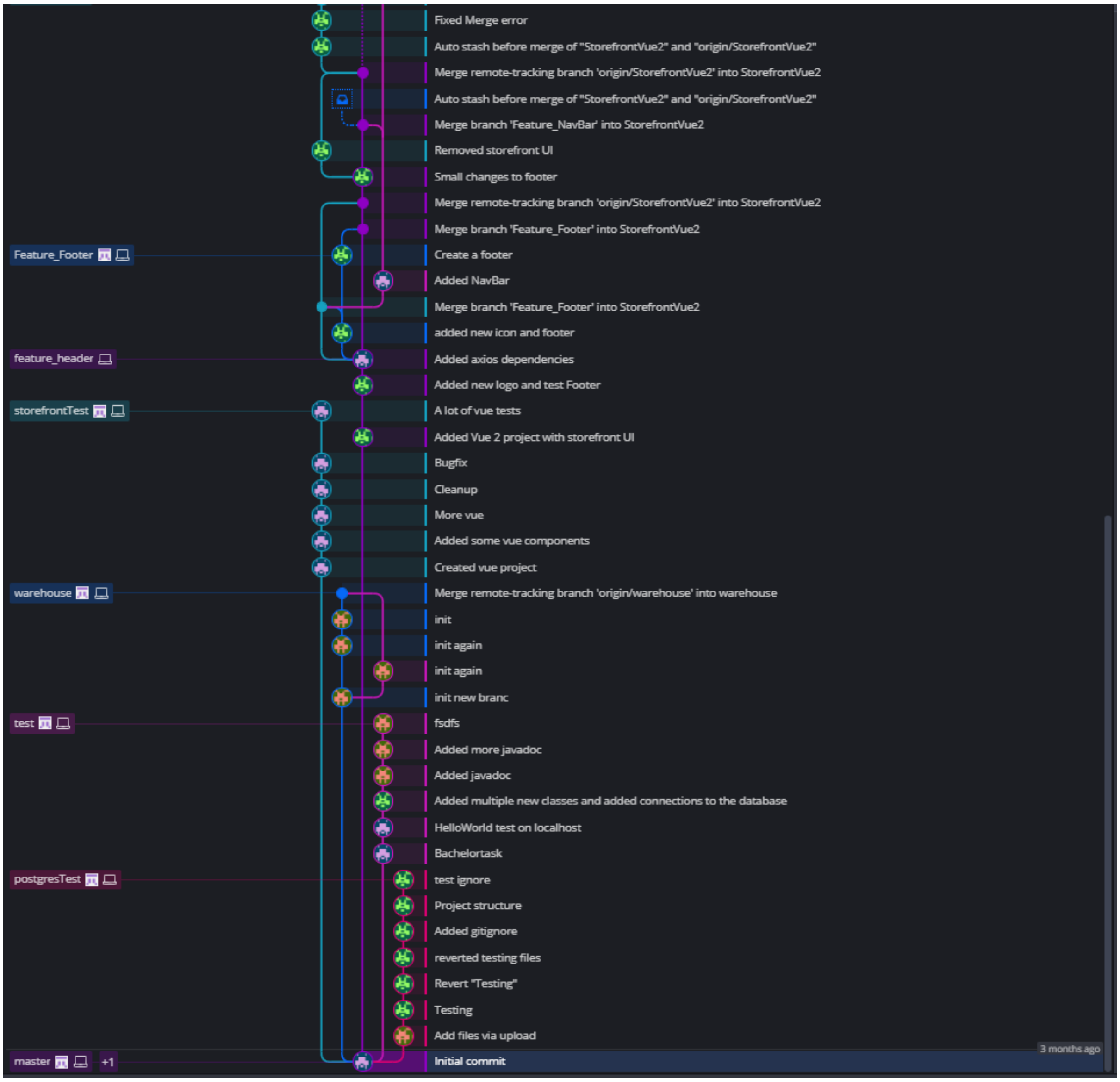
3.2.1 Repositories

We started with the repository "E-commerce-in-a-distributed-system-of-warehouses" Where we started developing the storefront:



```
Added CartButton
minor fixes
Changes to the productlists
Merge remote-tracking branch 'origin/Feature_Cart' into Feature_Cart2
Display products added in cart
Added a simple test cart
Changed how get requests are handeled
Made a get request to server
Merge branch 'Feature_ProductView' into Feature_ProductsComponent
Added some test products
Created product list
Added Product.vue and ProductCard.vue
Merge remote-tracking branch 'origin/Feature_Hero' into StorefrontVue2
Routing and views for all categories
Created routing for computer view
Removed alert function
Created login page with form
Added Hero.vue
Merge remote-tracking branch 'origin/StorefrontVue2' into StorefrontVue2
Added frontpage image
resolved conflict
Visual changes
Fixed all merge issues
Merge remote-tracking branch 'origin/StorefrontVue2' into StorefrontVue2
Fixed router and added computer page
Merge remote-tracking branch 'origin/StorefrontVue2' into StorefrontVue2
Merge branch 'feature_header' into StorefrontVue2
Merge branch 'Feature_NavBar' into StorefrontVue2
Fixed navbar
Added headar init
Auto stash before merge of "StorefrontVue2" and "Feature_NavBar"
Merge branch 'Feature_Router' into StorefrontVue2
Added router with home and about page
Compatibility fixes
small fixes
removed storefront ui
Auto stash before merge of "StorefrontVue2" and "origin/StorefrontVue2"
Commented out SF
Fixed Merge error
```

Figure 4: E-commerce-in-a-distributed-system-of-warehouses (part 1)

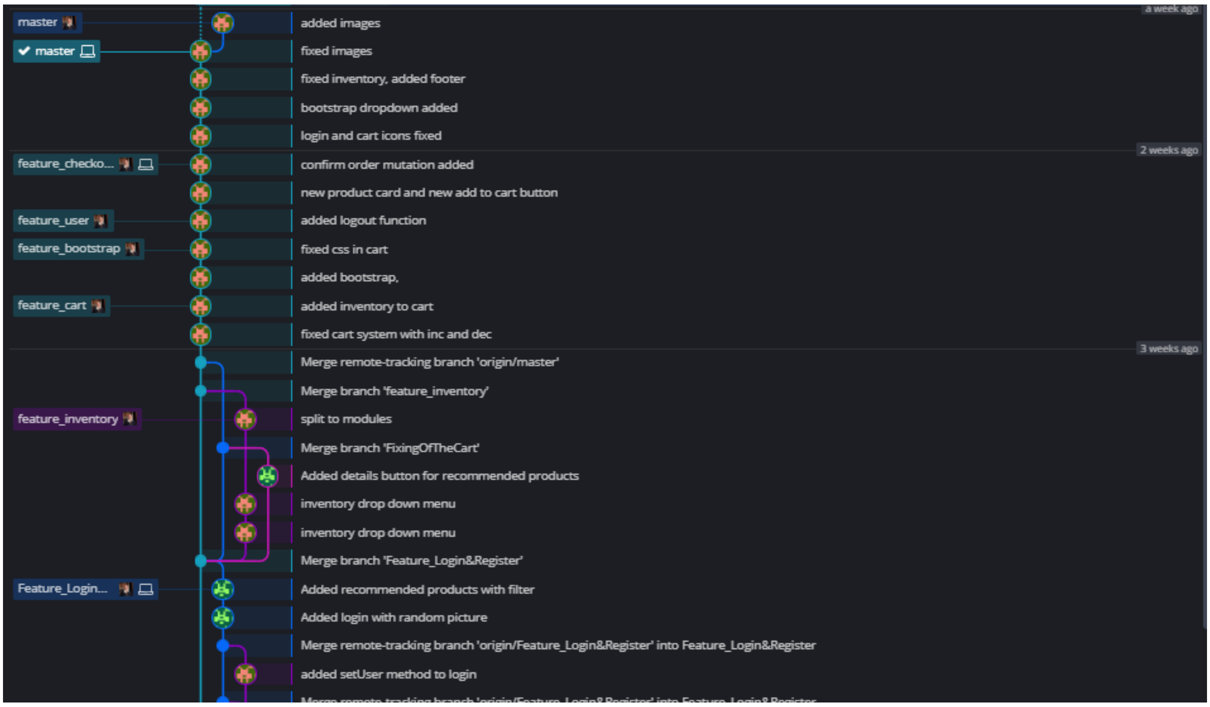


Fixed Merge error
Auto stash before merge of "StorefrontVue2" and "origin/StorefrontVue2"
Merge remote-tracking branch 'origin/StorefrontVue2' into StorefrontVue2
Auto stash before merge of "StorefrontVue2" and "origin/StorefrontVue2"
Merge branch 'Feature_NavBar' into StorefrontVue2
Removed storefront UI
Small changes to footer
Merge remote-tracking branch 'origin/StorefrontVue2' into StorefrontVue2
Merge branch 'Feature_Footer' into StorefrontVue2
Create a footer
Added NavBar
Merge branch 'Feature_Footer' into StorefrontVue2
added new icon and footer
Added axios dependencies
Added new logo and test Footer
A lot of vue tests
Added Vue 2 project with storefront UI
Bugfix
Cleanup
More vue
Added some vue components
Created vue project
Merge remote-tracking branch 'origin/warehouse' into warehouse
init
init again
init again
init new branc
fsdfs
Added more javadoc
Added javadoc
Added multiple new classes and added connections to the database
HelloWorld test on localhost
Bachelortask
test ignore
Project structure
Added gitignore
reverted testing files
Revert "Testing"
Testing
Add files via upload
Initial commit

Figure 5: E-commerce-in-a-distributed-system-of-warehouses (part 2)

This repository has a large amount of test branches, which reflect the fact that the first two sprints were to a large degree used for testing.

The storefront needed a ground-up rework with the website skeleton created in Vuex. The development with Vuex started on a new repository, “E-commerce_Storefront”:



a week ago

master — added images

✔ master — fixed images

fixed inventory, added footer

bootstrap dropdown added

login and cart icons fixed

2 weeks ago

feature_checko... — confirm order mutation added

new product card and new add to cart button

feature_user — added logout function

feature_bootstrap — fixed css in cart

added bootstrap,

feature_cart — added inventory to cart

fixed cart system with inc and dec

3 weeks ago

Merge remote-tracking branch 'origin/master'

Merge branch 'feature_inventory'

feature_inventory — split to modules

Merge branch 'FixingOfTheCart'

Added details button for recommended products

inventory drop down menu

inventory drop down menu

Merge branch 'Feature_Login&Register'

Feature_Login... — Added recommended products with filter

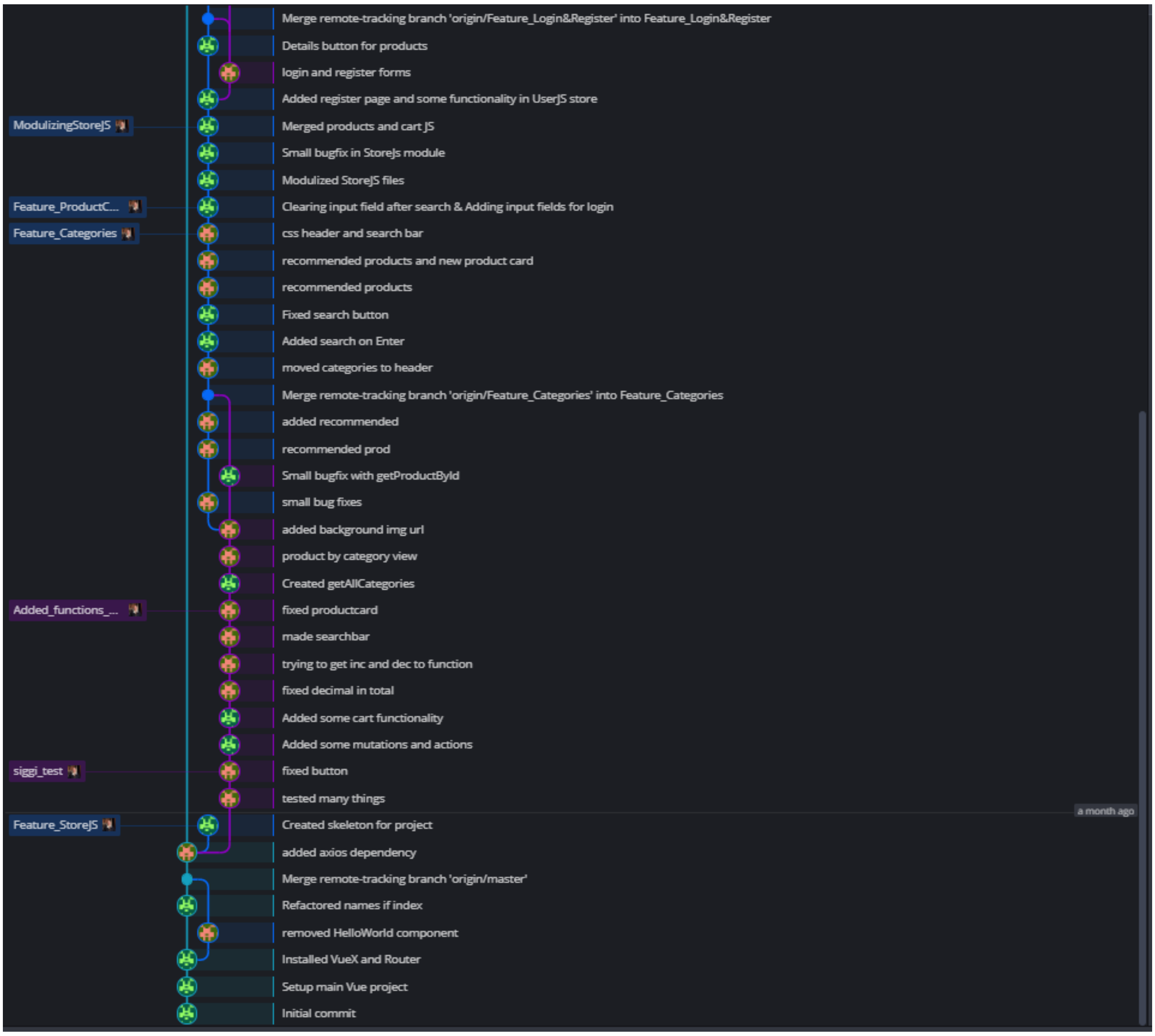
Added login with random picture

Merge remote-tracking branch 'origin/Feature_Login&Register' into Feature_Login&Register

added setUser method to login

Merge remote-tracking branch 'origin/Feature_Login&Register' into Feature_Login&Register

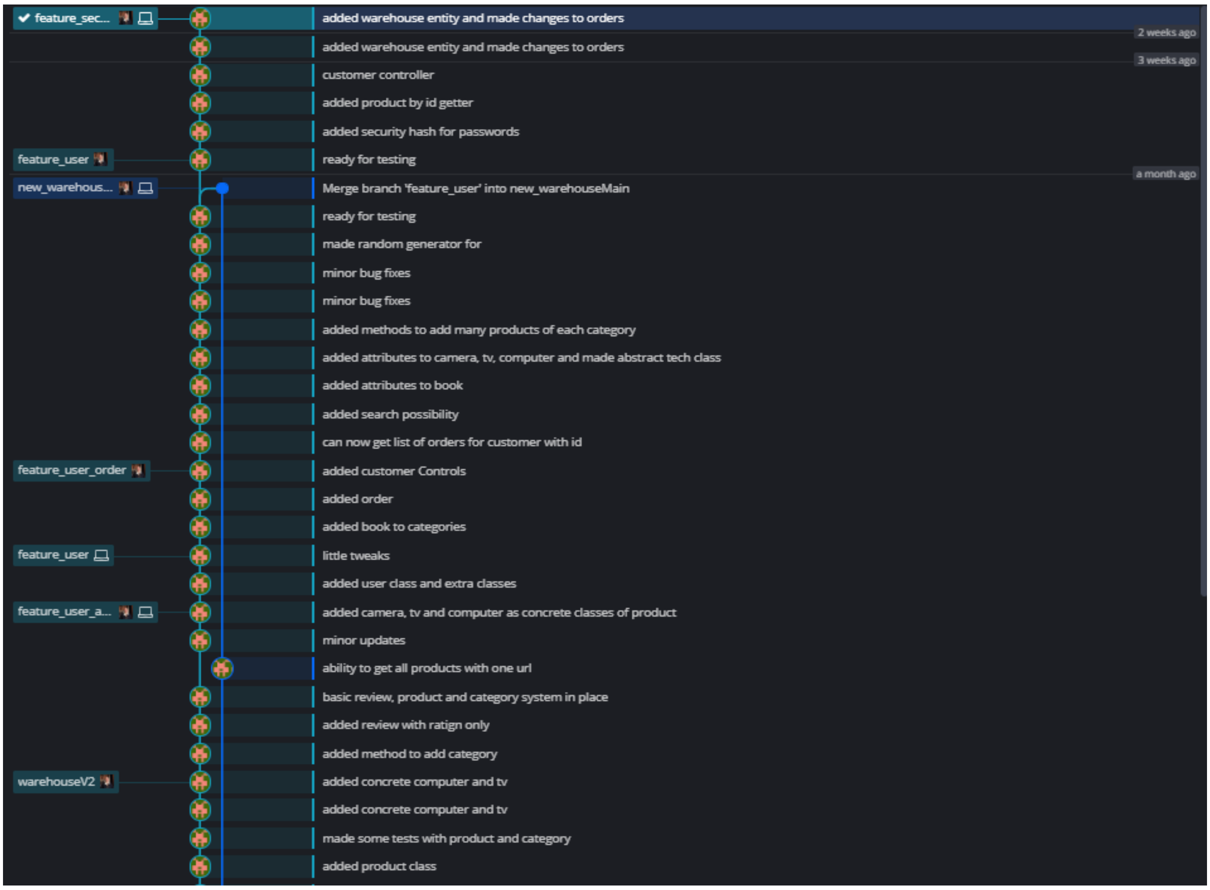
Figure 6: E-commerce_Storefront repository (part 1)



Merge remote-tracking branch 'origin/Feature_Login&Register' into Feature_Login&Register
Details button for products
login and register forms
Added register page and some functionality in UserJS store
Merged products and cart JS
Small bugfix in StoreJs module
Modulized StoreJS files
Clearing input field after search & Adding input fields for login
css header and search bar
recommended products and new product card
recommended products
Fixed search button
Added search on Enter
moved categories to header
Merge remote-tracking branch 'origin/Feature_Categories' into Feature_Categories
added recommended
recommended prod
Small bugfix with getProductById
small bug fixes
added background img url
product by category view
Created getAllCategories
fixed productcard
made searchbar
trying to get inc and dec to function
fixed decimal in total
Added some cart functionality
Added some mutations and actions
fixed button
tested many things
Created skeleton for project
added axios dependency
Merge remote-tracking branch 'origin/master'
Refactored names if index
removed HelloWorld component
Installed VueX and Router
Setup main Vue project
Initial commit

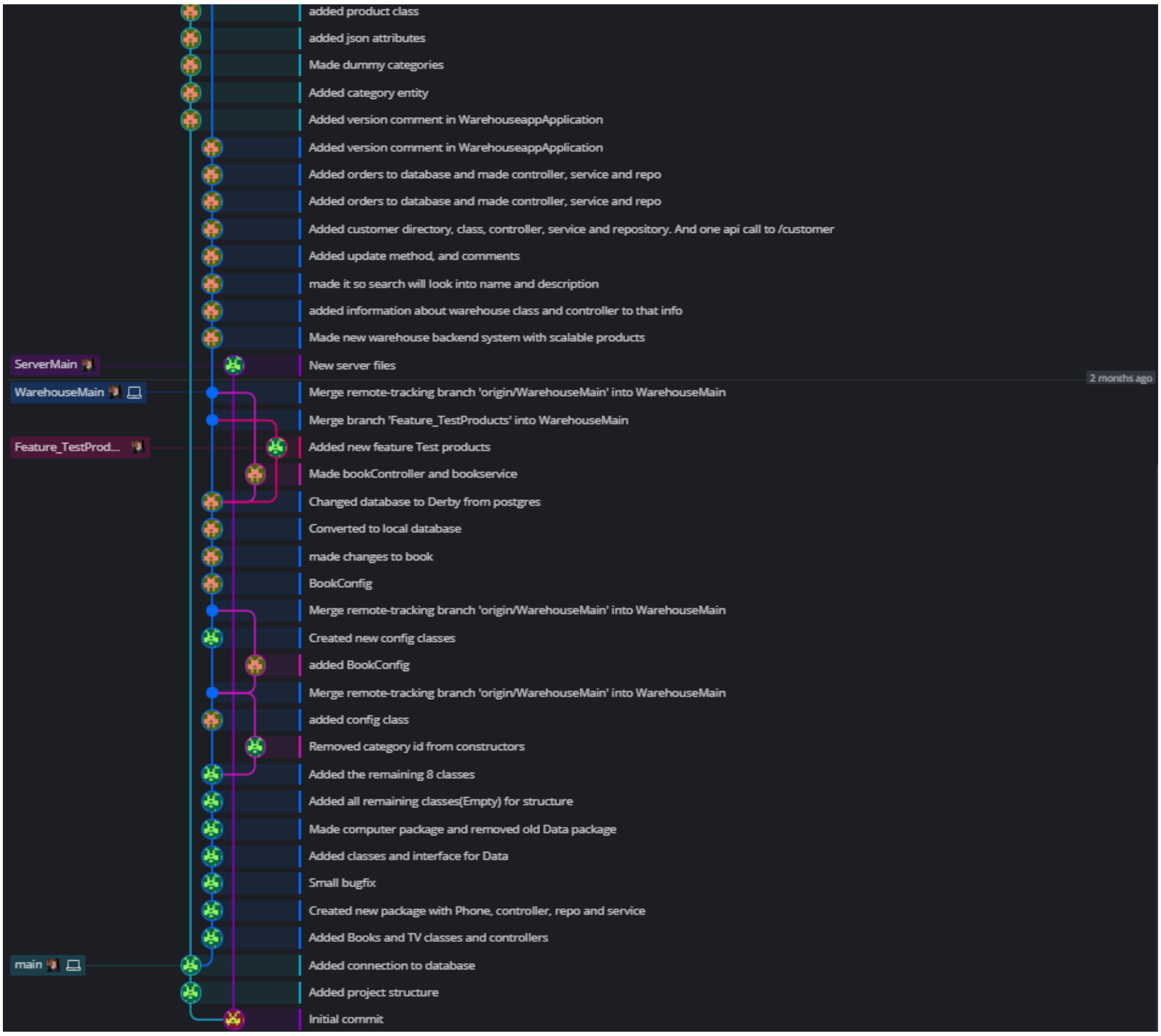
Figure 7: E-commerce_Storefront repository (part 2)

The warehouse database system was developed on the repository “WarehouseApp”:



```
added warehouse entity and made changes to orders
added warehouse entity and made changes to orders
customer controller
added product by id getter
added security hash for passwords
ready for testing
Merge branch 'feature_user' into new_warehouseMain
ready for testing
made random generator for
minor bug fixes
minor bug fixes
added methods to add many products of each category
added attributes to camera, tv, computer and made abstract tech class
added attributes to book
added search possibility
can now get list of orders for customer with id
added customer Controls
added order
added book to categories
little tweaks
added user class and extra classes
added camera, tv and computer as concrete classes of product
minor updates
ability to get all products with one url
basic review, product and category system in place
added review with ratign only
added method to add category
added concrete computer and tv
added concrete computer and tv
made some tests with product and category
added product class
```

Figure 8: WarehouseApp Repository (part 1)



```
added product class
added json attributes
Made dummy categories
Added category entity
Added version comment in WarehouseappApplication
Added version comment in WarehouseappApplication
Added orders to database and made controller, service and repo
Added orders to database and made controller, service and repo
Added customer directory, class, controller, service and repository. And one api call to /customer
Added update method, and comments
made it so search will look into name and description
added information about warehouse class and controller to that info
Made new warehouse backend system with scalable products
New server files
Merge remote-tracking branch 'origin/WarehouseMain' into WarehouseMain
Merge branch 'Feature_TestProducts' into WarehouseMain
Added new feature Test products
Made bookController and bookservice
Changed database to Derby from postgres
Converted to local database
made changes to book
BookConfig
Merge remote-tracking branch 'origin/WarehouseMain' into WarehouseMain
Created new config classes
added BookConfig
Merge remote-tracking branch 'origin/WarehouseMain' into WarehouseMain
added config class
Removed category id from constructors
Added the remaining 8 classes
Added all remaining classes(Empty) for structure
Made computer package and removed old Data package
Added classes and interface for Data
Small bugfix
Created new package with Phone, controller, repo and service
Added Books and TV classes and controllers
Added connection to database
Added project structure
Initial commit
```

Figure 9: WarehouseApp repository (part 2)

3.2.2 Git features used

As a principle, every feature that was developed was developed on a branch, and not merged until it was fully functional. Therefore, one would commit the changes to a branch, test it and then merge if everything is operational.

3.2.3 Merge errors

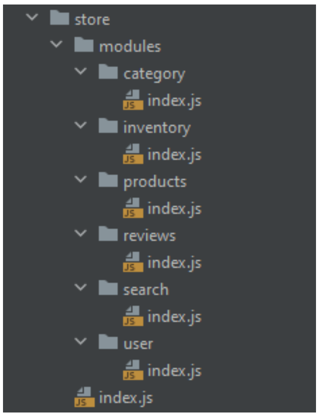
We used Gitkraken and as such we can use the merge tool included in the program. It allows us to see if there are merge conflicts and visualize the code where the merge conflict is located and shows how the output will look like.

3.3 Vue.js

Vue enables us to make as single page application with components that we can change. This makes it quite reactive and enables us to make a scalable platform to create our website on. Because Vue has template section for HTML, script section for JavaScript and style section for CSS in each component, it makes it easy to create a more global styling in the “app.vue” file and specific styling in each of the component. The biggest advantage for us is the availability of libraries such as Vuex, Vue router and Axios.

3.3.1 Vuex

In this project, Vuex was used to introduce a state-based management system that allows us to store arrays of information such as products, product by category, search products, and so on. It also allows us to have states for an object such as a user.



```
store
  modules
    category
      index.js
    inventory
      index.js
    products
      index.js
    reviews
      index.js
    search
      index.js
    user
      index.js
  index.js
```

Figure 10: Vuex Store modules

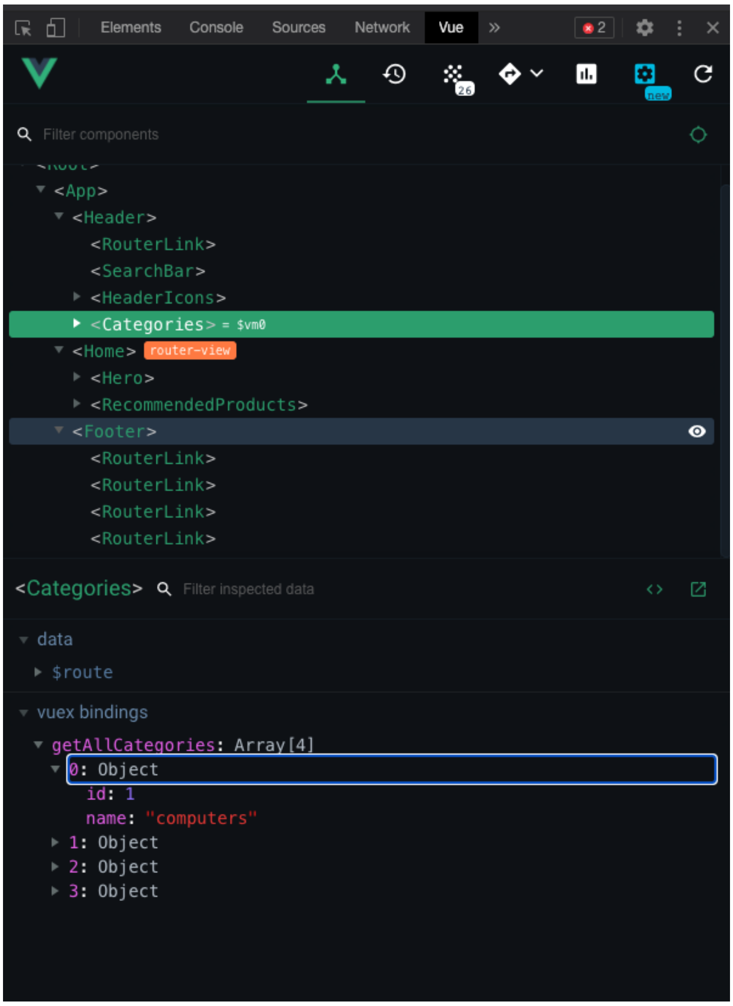
From Figure 10 we can clearly see the modular structure that we have used for the project. By splitting the store up into modules, it makes it easier for others to read and understand the code, while also maintaining better understanding and control of which stores are being used and which states are being changed.

3.3.2 Vue router

As previously stated, Vue is a single paged application which is not directly suitable for an e-commerce website. Therefore, we included the Vue router to be able to traverse the website and create a better user experience.

3.3.3 Vue Devtools

The Vue devtools is a valuable tool as it allows developers to see which value is currently assigned to any given variable or which input or value a method is holding. The Figure 11 below shows how this can be used as a debugging tool and tester. Here we can see that the categories state is an array with 4 objects, whereas the first object is “Computers”.



```
<Root>
  <App>
    <Header>
      <RouterLink>
      <SearchBar>
      <HeaderIcons>
      <Categories> = $vm0
    <Home> router-view
      <Hero>
      <RecommendedProducts>
    <Footer>
      <RouterLink>
      <RouterLink>
      <RouterLink>
      <RouterLink>
```

```
<Categories>
data
  $route
vuex bindings
  getAllCategories: Array[4]
    0: Object
      id: 1
      name: "computers"
    1: Object
    2: Object
    3: Object
```

Figure 11: Vue devtools

3.4 Maven

Maven is a project management tool, used primarily for Java projects. Maven makes it easier for developers to build and managing projects. This is done with XML file called POM holding on to configurations of the project. In that file for example, dependencies are added that are used in the project.

3.5 Spring

3.5.1 Spring Framework

The Spring Framework is open-source Java application developing platform. Spring handles the inner workings of the application or the low-level functionality. Enabling developing teams to focus on high-level or application-level business logic and allowing the development to be untied to specific deployment environments.[14] The first version of the framework was released in June 2003 and was written by Rod Johnson.[15] From version 2.5 on, Spring Framework started using annotations to control its behavior. Annotations provide meta data for the Java compiler instruction, build-time instruction, and runtime instruction. It is used, for example, for packaging compiled code into a JAR file. Prior to annotations XML configurations were used. In this project, there are numerous annotations.

For instance, following annotations are used for this project:

@SpringBootApplication

- Is used instead of using three other annotations, (@Configuration, @EnableAutoConfiguration and @ComponentScan), with their default attributes. [https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-using-springbootapplication-annotation.html]

@Bean

- A bean is another name for object. Instantiated and managed by Spring IoC (Inversion of Control) container. It is a process where object define their dependencies without constructing them.

@Autowired

- Is used to inject object dependency.

@Service

- Is used to annotate the service layer. This spring bean is where the business logic is kept.

[14] https://spring.io/projects/spring-framework

[15] https://en.wikipedia.org/wiki/Spring_Framework As of May 2021

3.5.2 Spring Boot

Spring boot is an extension of Spring framework. It eliminates the common configurations needed for Spring application setup. For example, Spring Boot comes with an embedded server, in which facilitates the application deployment. Spring Boot also has a variety of dependencies. Some of the frequently used are spring-boot-starter-data-jpa, spring-boot-starter-web, and spring-boot-starter-security. Those are also the ones that are used in this project.[16]

3.5.3 Spring Data JPA

Spring Data JPA (Java Persistence API) is a member of Spring Data family. Spring Data family is a group of technologies which is explored to easy the use of data access technologies, such as relational databases. With Data JPA developers write their own methods in a repository interface and Spring will automatically implement them. The JPA repository comes with *create*, *read*, *update*, and *delete* methods built in. This means that instead of writing SQL queries developers map class objects to a table in the database with annotations. In the following, we present some of the most used annotations used in this project.

@Repository

- Is used to indicate the class will provide operations for create, reading updating, and deleting objects.

@Query

- This annotation allows for custom made query in a repository. Here developers can write a normal SQL query.

@Transactional

- Is used to define the scope of each unique database transaction.

@Entity

- Is used to map a java class to a table in database. Each instance variables in a class are mapped to a column in a database table.

@Table

- Is used to configure table settings; for example, to change the name of the table and setting unique constraints.

[16] https://www.baeldung.com/spring-vs-spring-boot – As of May 2021

@UniqueConstraint

- Is used to configure unique constraints on a column. It is used with the @Table annotations unique constraint setting.

@Inheritance

- Is used to map together or join inherited subclasses in java into a table in a database.

@Id

- Is used to declare the primary key. Each @Entity must have an @Id annotation like each table in relational database must have a primary key. For generating unique values for the primary key, the @SequenceGenerator and @GeneratedValue annotations are used in conjunctions with the @Id annotation.

@Column

- Is used to configure the instance variables in java class. For example, setting name of the column in the database and to set constraints like if column allows any null values.

@ManyToOne

- Is used to indicate relations to another table in a relational database. It is used on instance variable that is itself an @Entity. Usually, it is used with @JoinColumn annotation as a configuration method to a foreign key.

3.5.4 Spring Web

With Spring Web starter pack, developers have access to all the tools needed to make a web application. In the spring web pack, there are for example, Spring MVC, REST, and a Tomcat server. The Spring MVC is a Java Framework that is based on the *Model View Controller design pattern. REST stands for Representational State Transfer and is an architectural style for an API. This API uses HTTP request to access data. Tomcat server is an open source where Java code can use HTTP environment. Like other spring boot dependencies annotations are used. In this project, the following annotations were used:

@RestController

- Is used to create RESTful web service. It maps request data to a service that will handle the request and then returns a JSON response.

@RequestMapping

- Is used to map request from client to control method. The annotations come with a value that is the path needed to get a response and method which tells what kind of response is requested. The methods can be GET, POST, DELETE, and PUT.

@RequestBody

- Is a JSON object that is sent as a respond to client request.

@RequestParam and @PathVariable

- Are used to get specific values from client request.

3.6 Java

3.6.1 Java programming language

Java is a high-level object-oriented programming language. Java was designed to run on any platform with Java virtual machine despite underlying computer architecture. Its first version was released in May 1995 and is today one of the most popular and most used programming languages in the world. Java is class based and comes with large library of class-interfaces, classes, and methods. Additionally, there are many 3rd-party libraries that can be used. This project uses a few built-in classes, as well as one 3rd-party class.

3.6.2 List and ArrayList

The List interface is a set of instructions on how to work with collections of objects in Java. It provides algorithms that handle for example adding and removing objects from collections. ArrayList implements the List interface. ArrayList is a resizable array that inherits the methods from List interface. The main difference between ArrayList and array in Java is that ArrayList can change its size dynamically versus a fixed size of array.

3.6.3 LocalDateTime

LocalDateTime class is used to represent date and time in the ISO-8601 calendar system. From this class comes timestamp of the date and time in this default format, yyyy-MM-ddTHH-mm-ss.zzz or like this 2021-05-01T10:21:55321.

3.6.4 CommandLineRunner

CommandLineRunner is a part of Spring framework. It is an interface that is used to run code immediately after a Spring application has started.

3.6.5 Faker

JavaFaker is a library that is used to generate variety of fake data. It is very useful when developing project and large amount of data is needed for testing.

3.7 Postman

Postman is API development tool. It is used, for example, to test API by creating and sending any kind HTTP request and reading their response. It is very useful, quick, and simple to use when trying out new API calls.

3.8 Integrated Development environments

3.8.1 IntelliJ

IntelliJ is a java IDE from the JetBrains IDE collection. It was used to develop the back-end warehouse application in Java for the project. IntelliJ was chosen because of its familiarity to the group, as all members have used the IDE in previous projects. It has several features that was helpful to the project:

- It has a code completion feature that give code recommendations based on the context of the code. This made the developing process more efficient.
- The IDE layout contains several tabs: Text editor, project file structure, local console, etc. Among these tabs, there is the Database tab. The database tab gives an overview of the database hierarchy, as well as a query console.

3.8.2 WebStorm

WebStorm is another IDE from JetBrains that is specifically designed for web development. The graphical user interface proved to be very consistent across the various JetBrains IDE’s. Layout-wise, WebStorm was almost indistinguishable from IntelliJ, but different in terms of functionality.

This IDE was chosen for this project as it had native support for vue.js development and was thus very suitable for the development of the storefront. This IDE also provided a useful feature: by writing “npm install” in the local terminal, WebStorm scans through the project files and detects any npm dependencies and installs them. This feature helped synchronize dependencies across all local instances of the project and meant that we could work without an OS-level virtualization software such as Docker.

4 Development process

4.1 Project timeline

Sprint #	Task	Time started	Time finished
A1	Research of platform and setup of dev area	15.01	29.01
A2	Created testing application	1.02	15.02
A3	Created database and Vue components	12.02	26.02
A4	Created new abstract database & API requests while making new Vuex storefront	26.02	12.03
A5	Created data driven categories, implemented product generator & finalizing API layer	26.03	09.04
A6	Created the register and login for user, order confirmation & display stock for each warehouse	09.04	23.04
A7	Administrative work (logs and weekly reports) & Bachelor Report writing	23.04	20.05

4.2 Sprint overview

4.2.1 A1

In our first sprint, we familiarized ourselves with Vue.js and Spring Boot. This sprint was used to work on the pre-project plan and learning how to use and setup the project. Therefore, this sprint was mainly focused on gathering information, getting familiar with the platform, and creating a backlog of the first steps. Figure 12 illustrates how we started the project by outlining tasks that needs be done. We often discussed in group what is the basic steps needed to start the project and write them all up in Jira.

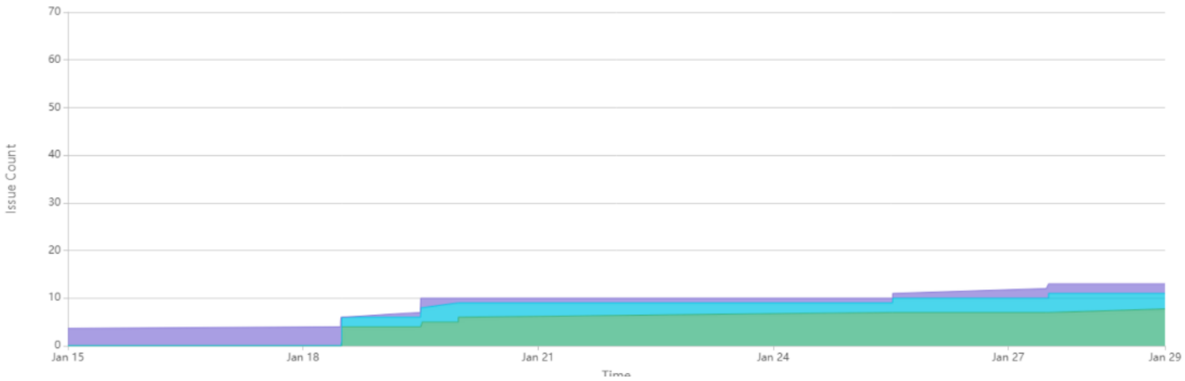


Figure 12: Sprint A1

In Figure 12, we can see the start of Sprint A1 from Jan 15 – Jan 29. The Color coding is purple for "To Do", Cyan for "In Progress" and lastly Green for "Done". The Y axis represents issues in Jira and the X axis is time (By date). The following figures in Chapter 4 will have the same layout.

4.2.2 A2

Sprint 2 started 1st of February and had the main focus on testing spring boot and Vue while using tutorials to gain more knowledge. In this stage, we set up a test database and got a json file with test products from DRIW. This file was not directly used but served as a reference point for what kind of information we might need to add and which to omit. For instance, we found that storing which package size should be used for each product was out of the scope of this project. Furthermore, we created the first test application that communicated with the database and did further research on the API layer in spring boot that we would use to make requests from the website later.

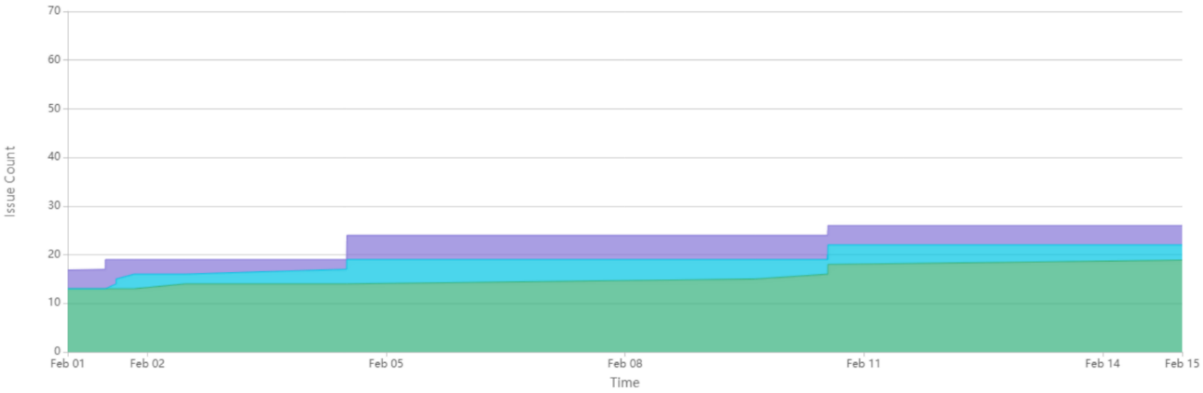


Figure 13: Cumulative flow progress report for Sprint A2.

4.2.3 A3

In Sprint 3, we created "TV" and "book" classes for the database application where we defined each product type. However, we soon found this to go against the "data-driven" goal for the project, as this would require us to create classes in backend for each product, which would also require the need to add classes in backend each time a new product type would be added. We asked for advice from our advisor which guided us towards NoSQL databases. However, it took longer than initially expected to learn Spring Boot and Vue, which put us behind schedule. After discussing the use of NoSQL database with DRIW they advised us to still use Postgres but to create a more abstract class, which our advisor also agreed it was a promising idea for the group.

Early in the sprint, our client at DRIW offered us a meeting with Olaf Nykrem, who has a lot of experience in the e-commerce field and previously was a leader in the digital department of "Byggmakker" for 3 years and also developed trading solutions and the e-commerce department for Malorama from 2016 to 2020. This was a big opportunity for the group to obtain valuable information and inspiration for the project. Olaf Nykrem gave us insights into how the websites function and the design decisions were made. This definitely helped us and gave us another point of view that we could use as a reference for our own project. We are very grateful to Olaf for taking his time to help us, and to Marius for making the meeting possible.

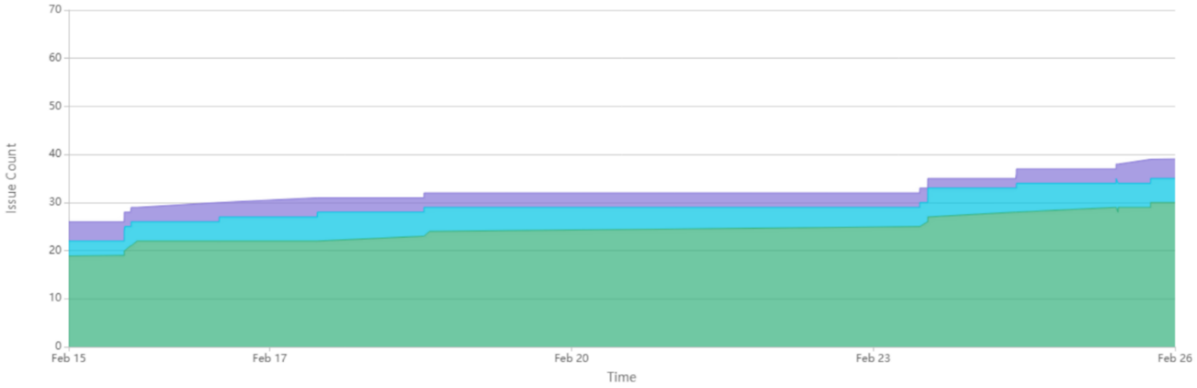


Figure 14: Cumulative flow progress report for Sprint A3.

4.2.4 A4

In this sprint, we experienced the biggest progress in the project so far. We had created an abstract database which allowed us to implement a more data-driven website and implemented Vuex which allows a centralized store file with state-based management. We found this to be a perfect fit for the project. This allowed us to instead of passing props from one component to another, we could store in it states. Not only did this decrease the chances of bugs but also decreased the complexity of the project and allowed us to focus more on new features. For instance, we could now make state arrays like cart and products that would hold on all the information in the cart and all the products in the database, while also making simple calls to them when needed. The feature created during this sprint was the routing for all the categories, login page, a hero component showing new products and displaying all products.

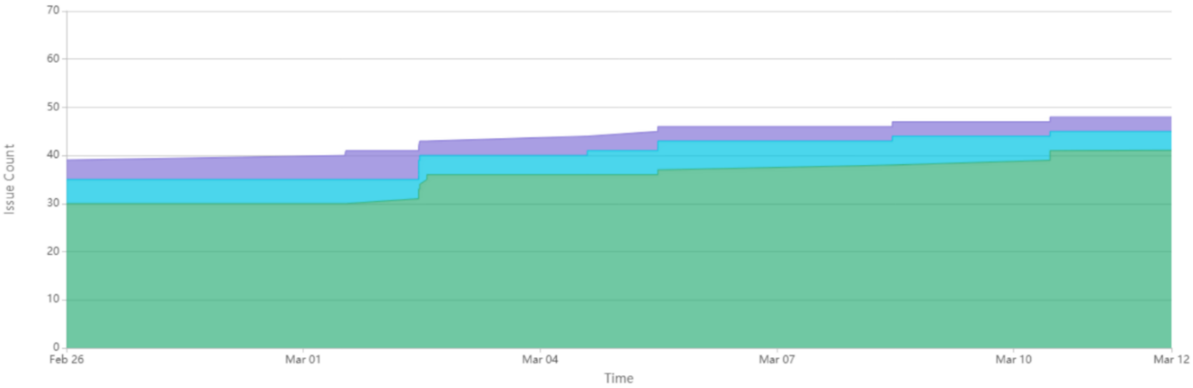


Figure 15: Cumulative flow progress report for Sprint A4.

Note that the end of this sprint 12th of march and the start of the next sprint is moved forward to 26th of march. This is due to the system subject exam preparation.

4.2.5 A5

In Sprint 5, we implemented a product generator that can make “fake” products to be able to test our features and that they are functioning properly. We created all the API calls and made a list all the calls that are available. Furthermore, we created the search logic in the Vuex Store JS that was connected to the search bar and implemented data-driven categories that will get the name of all the categories in the database and create a spot for them on the header with routing to a page that will show all products associated with the category. Lastly the backend system for handling and storing reviews was completed.

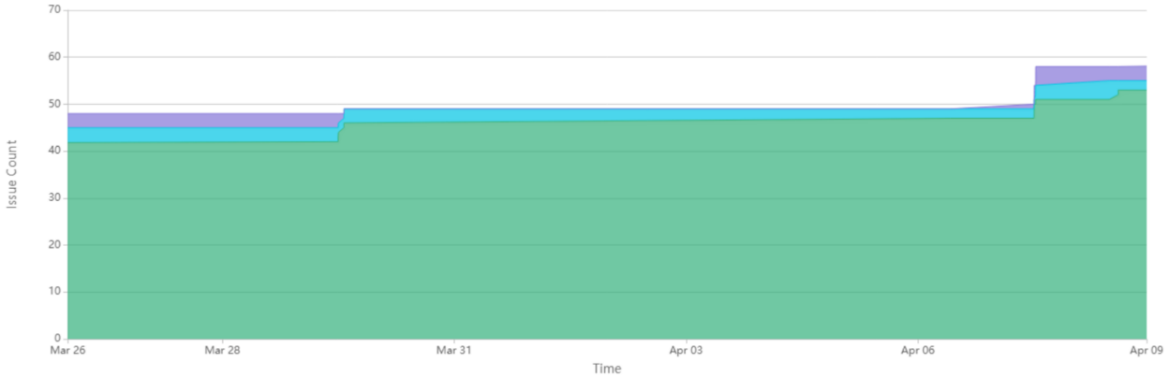


Figure 16: Cumulative flow progress report for Sprint A5.

4.2.6 A6

Sprint 6 was close to the ending of the project as the group has now moved one of three group members on writing the report while the last two are finishing up the last of the work on the project. In this Sprint, the Store JS was split into modules to decrease the complexity and also make it easier for others to understand and easily locate the Store calls in the code. This was important as it makes the code more readable and decreases the time needed to debug as it is more transparent which methods are being used at a given time. Moreover, a couple of bug fixes were completed and the features, such as user registration and login were finalized. Lastly, in this Sprint, the group was able to create all the checkout functionality before starting the writing process.

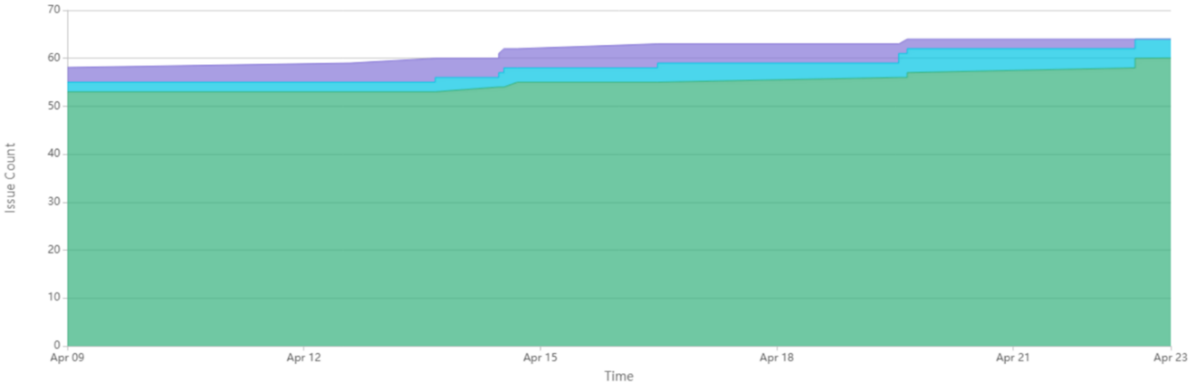


Figure 17: Cumulative flow progress report for Sprint A6.

4.2.7 A7

The last Sprint is being used to write the report and finishing some CSS styling for the website. This Sprint is reserved to finish the report and make sure that all resources are available and finalized.

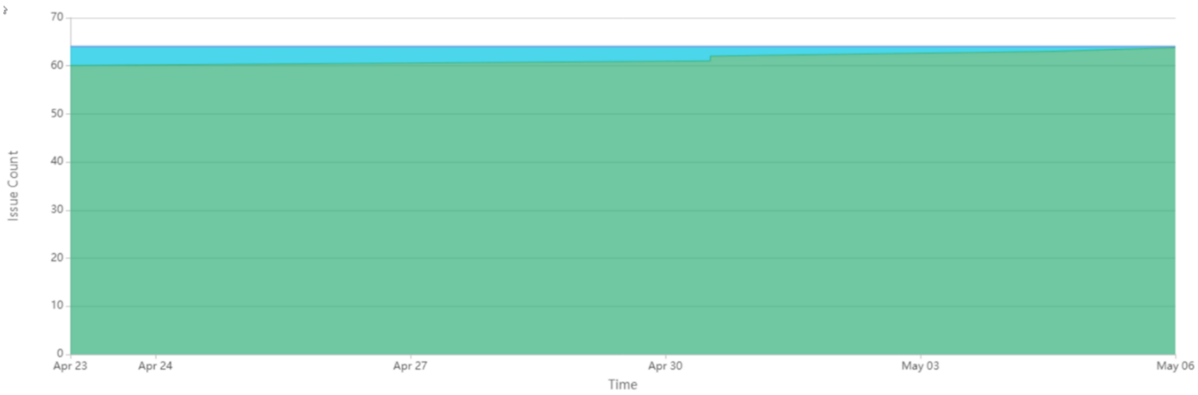


Figure 18: Cumulative flow progress report for Sprint A7. All issues are now listed as complete

4.3 Development model

4.3.1 Project tools

To make sure that all the changes being made to the program are being handled properly, without merge conflicts, we decided to use Gitkraken. This is due to the fact it is a simple graphical interface, which makes it easy to keep track of all the changes and branches being made, and the ability to visually compare the difference between to commits in the case that a merge conflict arises.

4.3.2 Management tools

We are using Jira as a way to guide the project and workflow. This is done by delegating issues or tasks that need to be done. We are also able to create multiple charts that will be used to improve our awareness about our progress and workflow, possibly contributing to the decision-making along the project development.

4.3.3 Development tools

We use:

- GITHUB to host our git repositories.
- IntelliJ IDE for making backend system.
- WebStorm for making the website storefront.
- Vue Devtools for debugging and testing
- DataGrip to manage the databases
- Postman to push information to the databases.
- Gitkraken GUI for GIT

4.4 Project meetings

Meetings with advisor: After every 2-week sprint, we met with both our advisor and our contact in DRIW AS. The exact date and time may vary and were scheduled some days prior to the meeting. We chose to have meetings like this because we could then give a report on the previous sprint and receive guidance for the next sprint.

Group meetings: Every weekday morning at 09:00 we had a SCRUM meeting planning out the day. After the meeting, we used to work together in discord or physical meetings until 14.00.

Work log: We created a sprint chart and rapport using Jira. This gave us opportunity to have access to all the information about the issues in each sprint. This report showed the number of issues that are finished and the number of issues that are still under progress and needs to be continued in the next sprint.

Progress reports: After every sprint, we created progress reports using Jira. By using this platform, we were able to create Burnup charts, velocity charts, and normal sprint reports to visualize the progress and remaining work for the project. This was utilized to support decision making in the group.

5 System architecture & implementation

In this section, we explain how we designed the system and how we implemented it. The system is divided into RESTful API backend system and data driven frontend website. The backend system is implemented with Spring Boot framework and Microservice architecture. The frontend is implemented with VUE.JS framework. HTTP requests are used for communicating between the systems.

5.1 Backend design

The backend system is divided into four main components. The warehouse application and three store applications. The warehouse has its own database and is running on its own server. The warehouse is responsible for all information about products, users, and orders. Each store is its own application with its own database and are running on different servers. The stores are responsible for their individual inventory. With minimum effort, a new store can be added to the system.

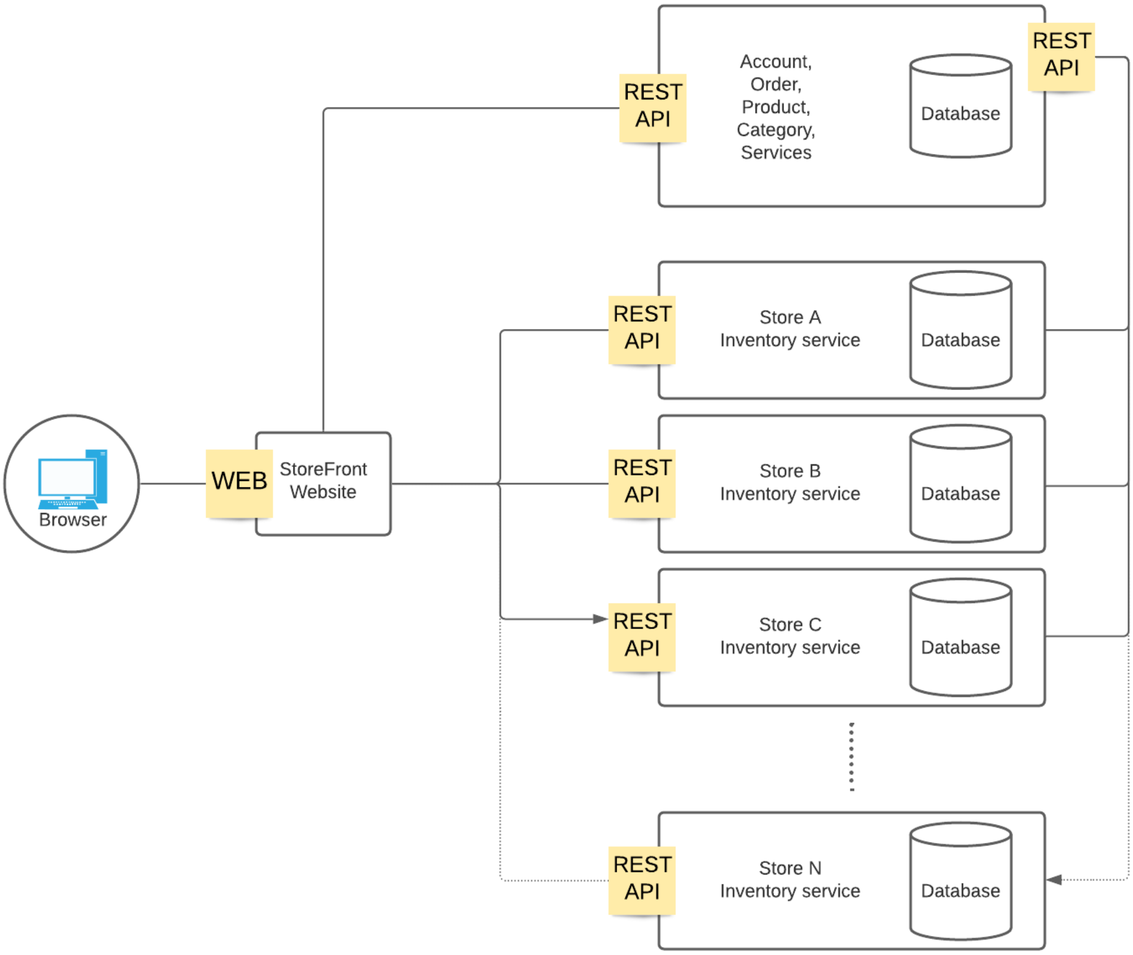
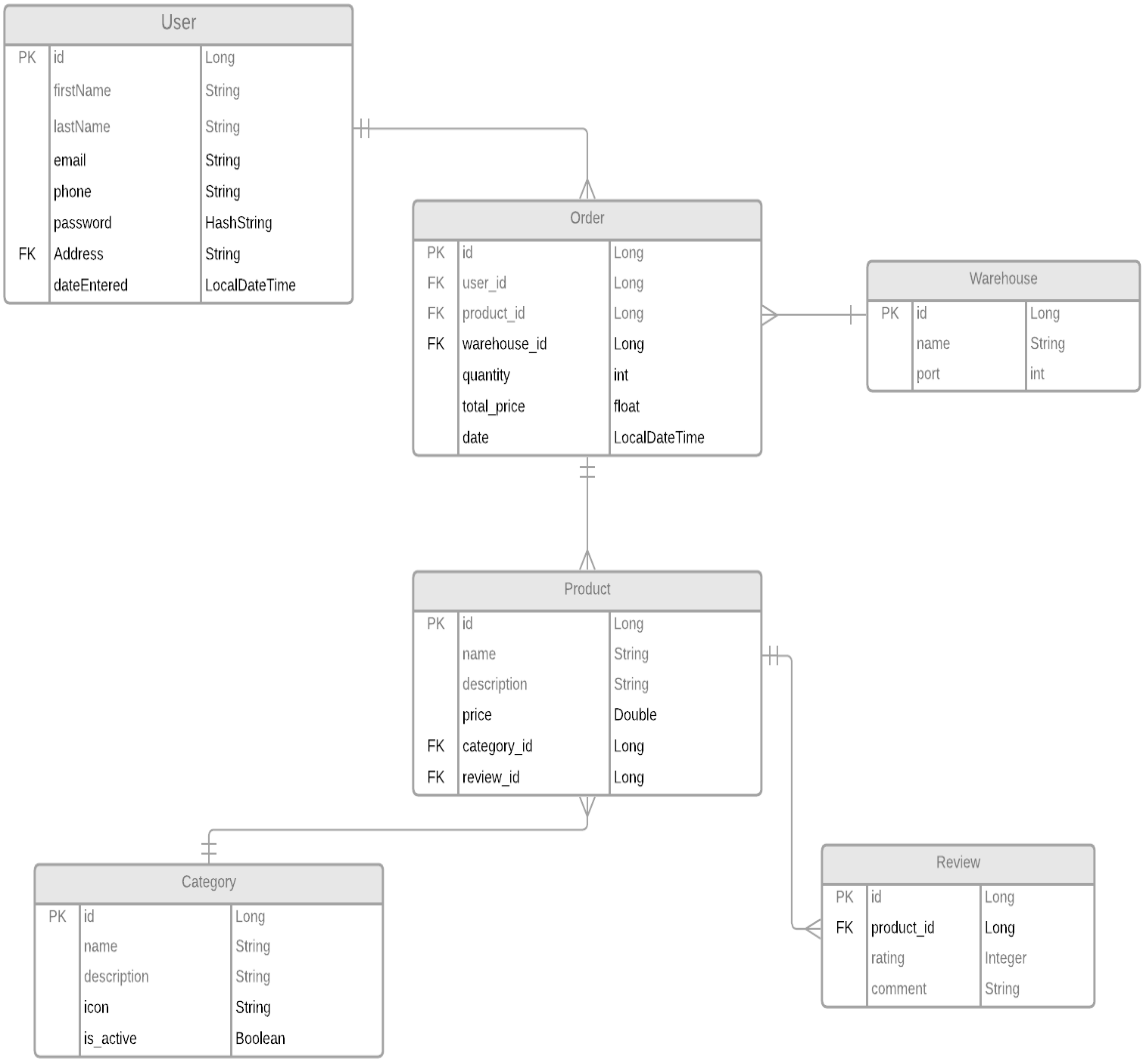


Figure 19: Microservice backend architecture.

Figure 19 shows the design of the system. It follows a Microservice architecture. On the top right, there is the Warehouse service. Below, there are three stores, A, B, and C. Further down, we show that more stores will have the same setup as A, B and C. The stores get product ids from the warehouse via http request. On the left, there is the StoreFront Website and the client browser. The store front communicates with the warehouse and each store via http requests.

5.1.1 Warehouse implementation

The warehouse system is made with the Java Spring Boot Web framework and PostgreSQL relational database system. It is responsible for all customer, product, and order data, as well as for making test data used for this project. Figure 20 illustrates the warehouse database diagram, indicating the entities and their relationships.



User		
PK	id	Long
	firstName	String
	lastName	String
	email	String
	phone	String
	password	HashString
FK	Address	String
	dateEntered	LocalDateTime

Order		
PK	id	Long
FK	user_id	Long
FK	product_id	Long
FK	warehouse_id	Long
	quantity	int
	total_price	float
	date	LocalDateTime

Warehouse		
PK	id	Long
	name	String
	port	int

Product		
PK	id	Long
	name	String
	description	String
	price	Double
FK	category_id	Long
FK	review_id	Long

Category		
PK	id	Long
	name	String
	description	String
	icon	String
	is_active	Boolean

Review		
PK	id	Long
FK	product_id	Long
	rating	Integer
	comment	String

Figure 20: Warehouse database diagram

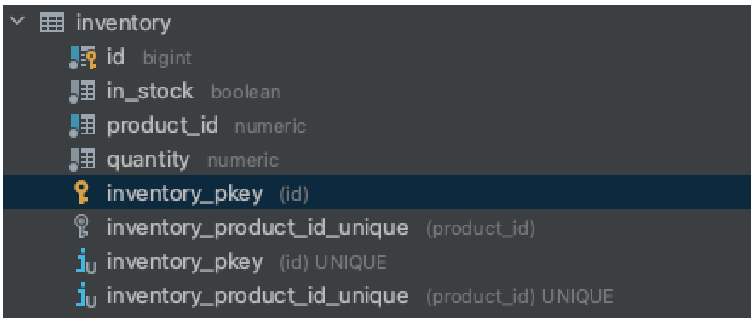
5.1.2 Store implementation

The stores are also made with Spring Boot Framework and use a PostgreSQL database. Each store gets product ids from the warehouse vie API call and is responsible for keeping inventory list of products available in that store. The store has only one entity and that is the inventory entity.

5.2 Entity of the store

5.2.1 Inventory

The inventory entity has four properties. The id property is the primary key and is of type big integer. Then there is the product id that also is of type big integer. Stock property is Boolean, while and quantity property is of type numeric.



```
inventory
    id  bigint
    in_stock  boolean
    product_id  numeric
    quantity  numeric
    inventory_pkey  (id)
    inventory_product_id_unique  (product_id)
    inventory_pkey  (id) UNIQUE
    inventory_product_id_unique  (product_id) UNIQUE
```

Figure 21: Inventory entity

5.3 Entities of the warehouse

In the database, we ended up with six main entities. Customer, product, category, review, warehouse, and orders. There are five other entities in the database that all inherit from the product entity. Figure 22 shows the class diagram for product and children classes of product.

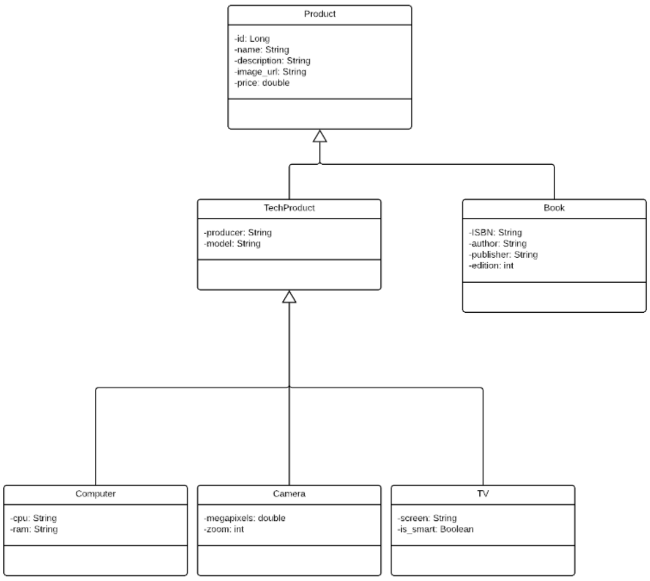
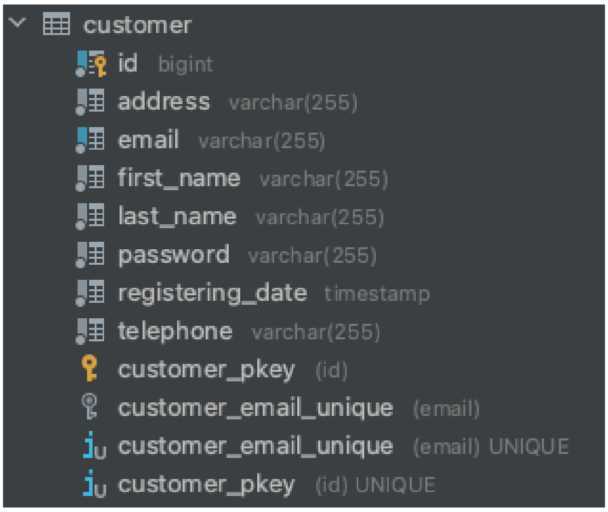


Figure 22: Class diagram showing product hierarchy

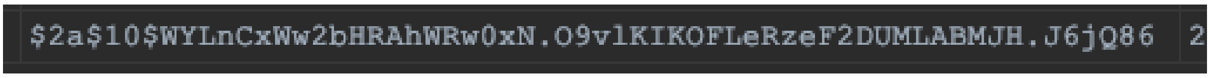
5.3.1 Customer

The customer entity represents users in the database. It contains eight properties. The primary key is the Id property that is auto incremented big integer type. Another attribute is a unique email (text type). Other attributes are first name, last name, telephone, and address, that are all varchar types or text. The timestamp type is used for registering date and the varchar or text type for password. The password is encoded in the database. Figure 23 shows the customer table, while Figure 24 shows the encoded password as it is stored in the database row.



```
customer
    id  bigint
    address  varchar(255)
    email  varchar(255)
    first_name  varchar(255)
    last_name  varchar(255)
    password  varchar(255)
    registering_date  timestamp
    telephone  varchar(255)
    customer_pkey  (id)
    customer_email_unique  (email)
    customer_email_unique  (email) UNIQUE
    customer_pkey  (id) UNIQUE
```

Figure 23: customer entity

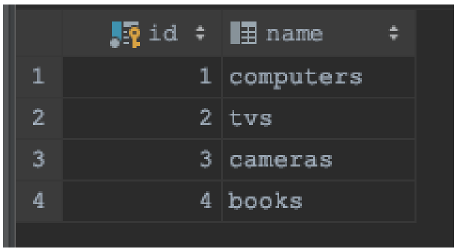


```
$2a$10$WYLnCxWw2bHRAhWRw0xN.O9vlKIKOFLeRzeF2DUMLABMJH.J6jQ86  2
```

Figure 24: encoded password in database

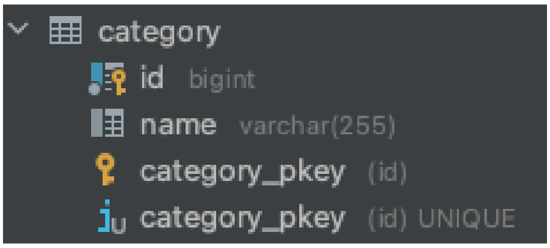
5.3.2 Category

The category entity is a small table only with Id and name properties. The id is auto incremented and unique primary key of type big integer. Name is of type text or varchar.



	id	name
1	1	computers
2	2	tvs
3	3	cameras
4	4	books

Figure 26: Category table

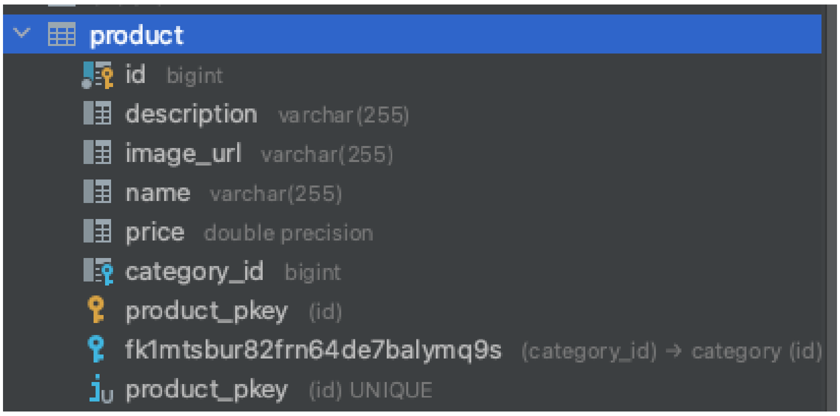


```
category
    id  bigint
    name  varchar(255)
    category_pkey  (id)
    category_pkey  (id) UNIQUE
```

Figure 25: Category entity

5.3.3 Product

The product entity represents what is in common with the all the products. It contains six properties. Primary key is the Id property which is big integer type and is auto incremented and unique. Then there are three text properties: description, images, and name. Image property keeps the URL of an image associated with the product. Also, there is the price property that is of type double. Category id property is a Foreign Key to the id property in category (section 5.3.2) entity.

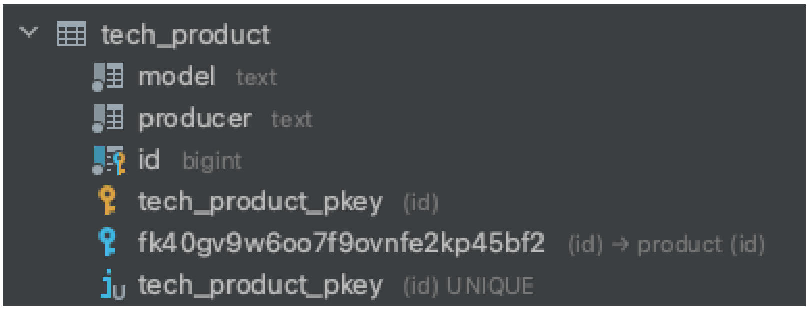


```
product
    id  bigint
    description  varchar(255)
    image_url  varchar(255)
    name  varchar(255)
    price  double precision
    category_id  bigint
    product_pkey  (id)
    fk1mtsbur82frn64de7balymq9s  (category_id) → category (id)
    product_pkey  (id) UNIQUE
```

Figure 27: Product entity

5.3.4 Tech product

Tech product entity is a child entity to the product entity. It has three properties. Model and producer are both text type properties. The id property is a Foreign Key to the id in product entity.

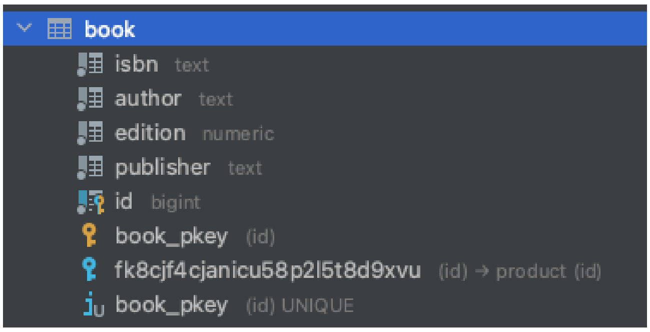


```
tech_product
    model  text
    producer  text
    id  bigint
    tech_product_pkey  (id)
    fk40gv9w6oo7f9ovnfe2kp45bf2  (id) → product (id)
    tech_product_pkey  (id) UNIQUE
```

Figure 28: Tech product entity

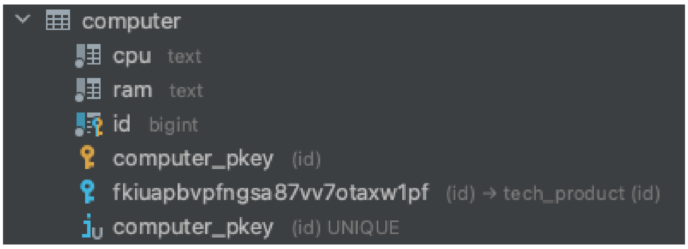
5.3.5 Specific products

In our project, we use four types of products: book, computer, camera, and TV. Four each of those we have an entity in the database. The book entity is child to the product entity. Computer, camera, and TV are child entity to the tech product entity. The book entity has five properties: ISBN, author, and publisher are all the text type. Edition is of type integer and the id is an integer and foreign key to the product entity. Computer has three properties: CPU, RAM, and Id. CPU and ram properties are both of type text. The Id is of type integer and is a foreign key to the tech product entity. The camera entity has also three properties. Megapixel's property is of type numeric and holds floating number; Zoom is of type integer; and the id of camera entity is like the id of computer entity. The TV entity has the same kind of id as camera entity and computer entity. It also has an attribute named smart (Boolean type) and screen (text type).



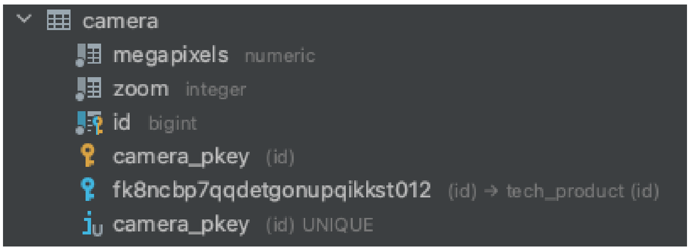
```
book
    isbn  text
    author  text
    edition  numeric
    publisher  text
    id  bigint
    book_pkey  (id)
    fk8cjf4cjanicu58p2l5t8d9xvu  (id) → product (id)
    book_pkey  (id) UNIQUE
```

Figure 29: Book entity



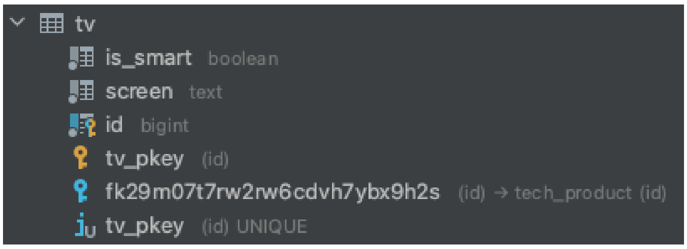
```
computer
    cpu  text
    ram  text
    id  bigint
    computer_pkey  (id)
    fkiuapbvpfngsa87vv7otaxw1pf  (id) → tech_product (id)
    computer_pkey  (id) UNIQUE
```

Figure 30: computer entity



```
camera
    megapixels  numeric
    zoom  integer
    id  bigint
    camera_pkey  (id)
    fk8ncbp7qqdetgonupqikkst012  (id) → tech_product (id)
    camera_pkey  (id) UNIQUE
```

Figure 31: camera entity

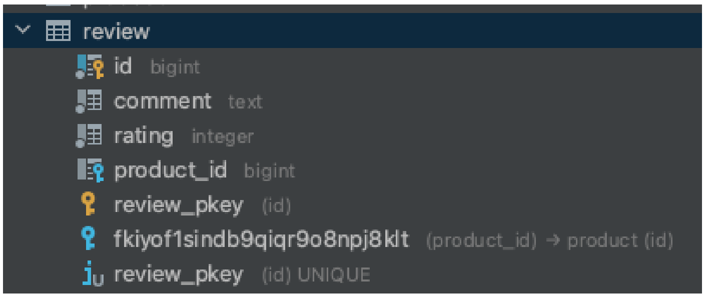


```
tv
    is_smart  boolean
    screen  text
    id  bigint
    tv_pkey  (id)
    fk29m07t7rw2rw6cdvh7ybx9h2s  (id) → tech_product (id)
    tv_pkey  (id) UNIQUE
```

Figure 32: Tv entity

5.3.6 Review

The review entity has four properties: The id is a Primary key and of the data type big integer; The comment property of type text and rating property of type integer; and finally, the product id property, a foreign key of type big integer referring to the id of the product entity.

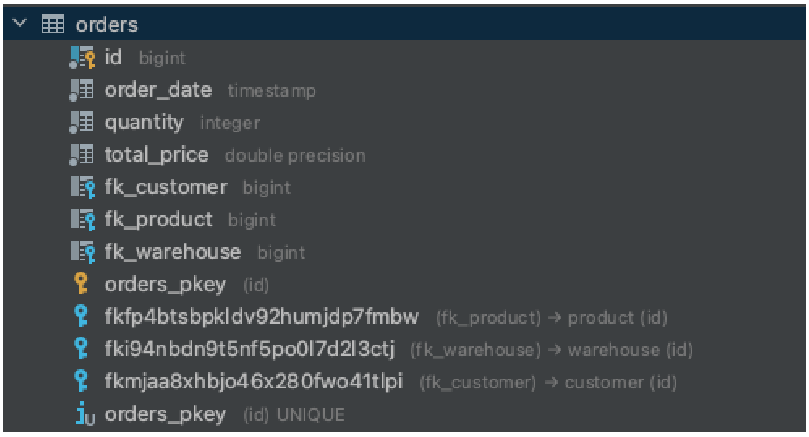


```
review
    id  bigint
    comment  text
    rating  integer
    product_id  bigint
    review_pkey  (id)
    fkiyof1sindb9qiqr9o8npj8klt  (product_id) → product (id)
    review_pkey  (id) UNIQUE
```

Figure 33: Review entity

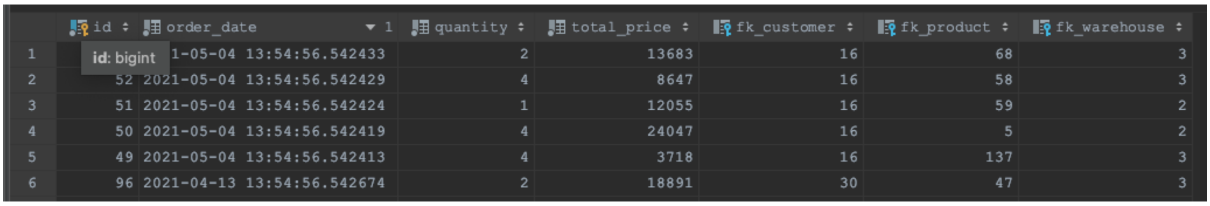
5.3.7 Orders

The orders entity represents an order in the database. It contains seven properties. Id is of “big integer” datatype; it is the primary key of the table. Another attribute, the order date is of type timestamp. Order date gets its value from the LocalDateTime(link) method. Quantity is of type integer and total price is of type double. The table also contains three foreign keys, all of which are of type big integer. Fk customer refers to the id of customer entity (5.3.1). Fk product refers to the id of the product entity (5.3.3). And at last, the fk warehouse that refers to id of the warehouse entity (5.3.8).



```
orders
    id  bigint
    order_date  timestamp
    quantity  integer
    total_price  double precision
    fk_customer  bigint
    fk_product  bigint
    fk_warehouse  bigint
    orders_pkey  (id)
    fkfp4btsbpkldv92humjdp7fmbw  (fk_product) → product (id)
    fki94nbdn9t5nf5po0l7d2l3ctj  (fk_warehouse) → warehouse (id)
    fkmjaa8xhbjo46x280fwo41tlpi  (fk_customer) → customer (id)
    orders_pkey  (id) UNIQUE
```

Figure 34: Orders entity

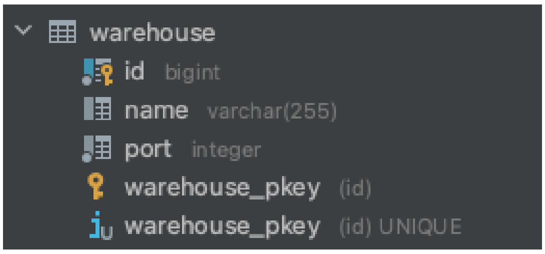


	id	order_date ▾ 1	quantity	total_price	fk_customer	fk_product	fk_warehouse
1		1-05-04 13:54:56.542433	2	13683	16	68	3
2	52	2021-05-04 13:54:56.542429	4	8647	16	58	3
3	51	2021-05-04 13:54:56.542424	1	12055	16	59	2
4	50	2021-05-04 13:54:56.542419	4	24047	16	5	2
5	49	2021-05-04 13:54:56.542413	4	3718	16	137	3
6	96	2021-04-13 13:54:56.542674	2	18891	30	47	3

Figure 35: Orders in postgres database

5.3.8 Warehouse

The warehouse entity has three properties. Id is of type big integer and is the primary key of the table. Name property is of type text and port property is of type integer.



```
warehouse
    id  bigint
    name  varchar(255)
    port  integer
    warehouse_pkey  (id)
    warehouse_pkey  (id) UNIQUE
```

Figure 36: Warehouse entity

5.4 RESTful API for the warehouse and stores

The warehouse system is a Java @SpringBootApplication. It is run from the main class of the system the WarehouseappApplication class. The first thing that happens when the application runs is CORS configuring. This configuration allows for the system to run on localhost as a server. Next the CommandLineRunner runs code that generates and loads the localhosted postgres database with dummy data. Then the RESTful API of the warehouse application is up and running on localhost port 8090. The store system is also a Java @SpringBootApplication like the warehouse. We use CORS configuration as well in the store application to allow the warehouse server to be accessed securely. In this project, we have three store applications. They are all exactly alike only with the application properties configuration of the database different as well as the port they are running on. When store application runs the first thing it does is to get product ids from the warehouse. Next it randomly generates inventory list associated with the ids. Then the store server is running and ready to handle http request from the storefront. We use localhost port 8091, 8092 and 8093 for the three stores. New stores can be added only be setting up new databases and new ports. The client will send http request to the RESTful API controller located in @RestController class. The @Servie class will handle the business logic of the request and communicate with the dataservice (postgres database) via the jpa repository. The client will then get response from the API whether it was and successful request or not. For each of the entities in the warehouse system is a RESTful API. Each entity has its own @Entity class, @RestController class, @Service class and a @Repository interface. For example, as shown in Figure 37 the product package in warehouse application has the following classes. Product is the @Entity class. ProductController is the @RestController. ProductRepository is the JPA-repository interface. And ProductService is the @Service class.

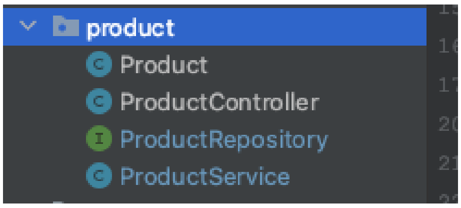
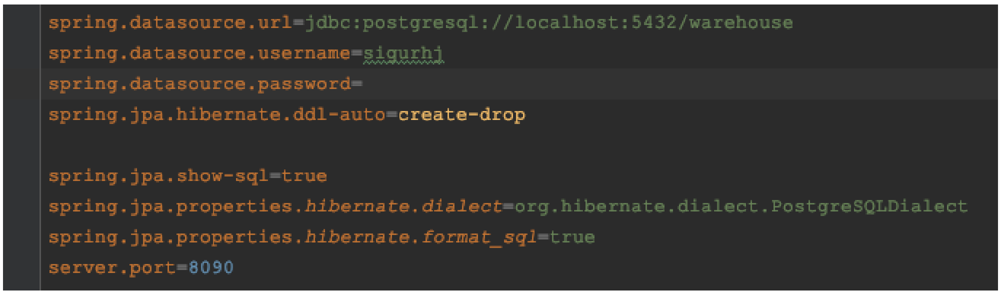


Figure 37: product package with its classes and interface

5.4.1 Application properties

The application properties file of the warehouse system is used for configuring the system. The main purpose is to setup the connection to database server and the JPA handling of that database. Also, is it possible to reconfigure the server port that is set as default 8080.



```
spring.datasource.url=jdbc:postgresql://localhost:5432/warehouse
spring.datasource.username=sigurhj
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
server.port=8090
```

Figure 38: Properties configure for warehouse

5.4.2 Entity classes

The entity class is the class that is mapped to a table in the database. It is a normal Java class annotated with @Entity at the top. It requires an @Id instance variable that becomes the primary key in the database table. In the following, there is a code snippet that is an example of a basic entity class.

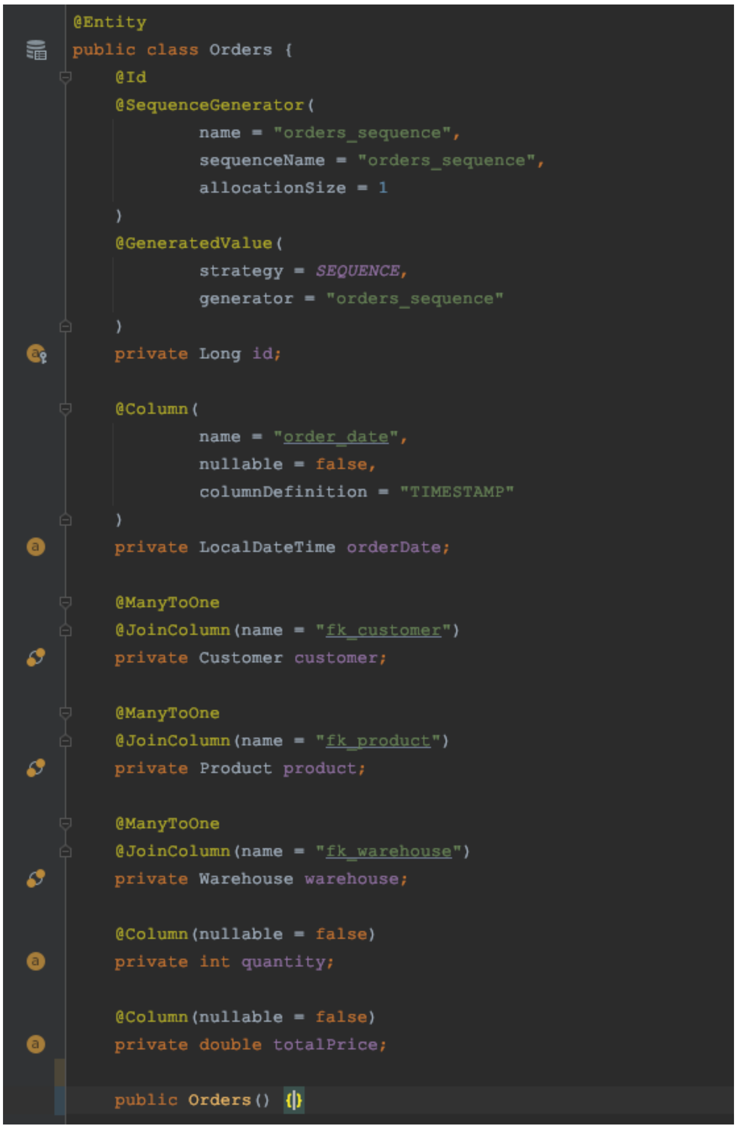
```
@Entity
@Table ( name = “category” )
public class Category {
      @Id
      private Long id;

      @Column
      private String name;
      // other variables

      // empty constructor
      public Category () {}
      // getters and setters omitted.
}
```

The instance variables of the class will become property of the database tables mapped with the class. The @Column annotation is used to configure the property. Some of the configurations are used to define the type and name of the property and if it allows null value in the table column. In this project, we used @SequenceGenerator and @GenerateValue with the @Id to auto generate unique ids for all the products, orders, customers, reviews, and categories.

To make a relation between tables, we used @ManyToOne and @JoinColumn with the instance variable that is itself another entity class. Example of this is showed in following code snippet.

```
@ManyToOne
@JoinColumn (name = “category_id”)
private Category category;
```

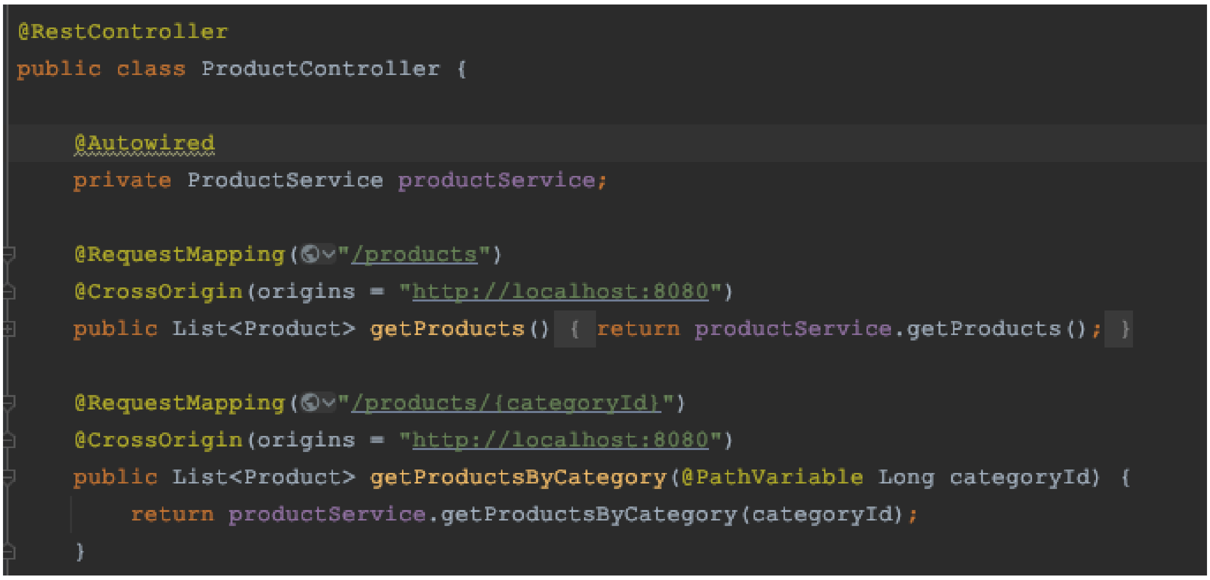
Here the category variable is a foreign key to the id in category table of the database. Another example of this is in the orders class, shown in Figure 39. In that class there are three foreign keys relating orders to the customer table, the product table, and the warehouse table.



```
@Entity
public class Orders {
    @Id
    @SequenceGenerator(
            name = "orders_sequence",
            sequenceName = "orders_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = SEQUENCE,
            generator = "orders_sequence"
    )
    private Long id;

    @Column(
            name = "order_date",
            nullable = false,
            columnDefinition = "TIMESTAMP"
    )
    private LocalDateTime orderDate;

    @ManyToOne
    @JoinColumn(name = "fk_customer")
    private Customer customer;

    @ManyToOne
    @JoinColumn(name = "fk_product")
    private Product product;

    @ManyToOne
    @JoinColumn(name = "fk_warehouse")
    private Warehouse warehouse;

    @Column(nullable = false)
    private int quantity;

    @Column(nullable = false)
    private double totalPrice;

    public Orders() {}
```

Figure 39: orders entity class

5.4.3 REST API control layer @RestController



```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @RequestMapping("/products")
    @CrossOrigin(origins = "http://localhost:8080")
    public List<Product> getProducts() { return productService.getProducts(); }

    @RequestMapping("/products/{categoryId}")
    @CrossOrigin(origins = "http://localhost:8080")
    public List<Product> getProductsByCategory(@PathVariable Long categoryId) {
        return productService.getProductsByCategory(categoryId);
    }
```

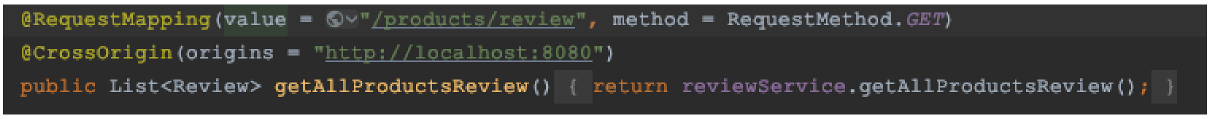
Figure 40: ProductController class

The @RestController class acts as a control layer between the client and the service it requests. The class contains the API request methods. Each method is annotated with @RequestMapping and in this project with @CrossOrigin annotation. Cross origin or CORS stands for Cross Origin Resource Sharing. It defines a way for client and server to determine if it safe to allow cross origin request.[17] Figure 40 shows how @CrossOrigin sets origin at localhost with port 8080 as a secure origin for request.

@RequestMapping annotation is used to map the http request to the method. Figure 41 shows how the top method @RequestMapping is configured to the path “/products” with cross origin set to “http/localhost:8080”. This means that sending the http GET request to “http/localhost:8080/products” will invoke the getProducts method and return JSON response of all products of the warehouse application.

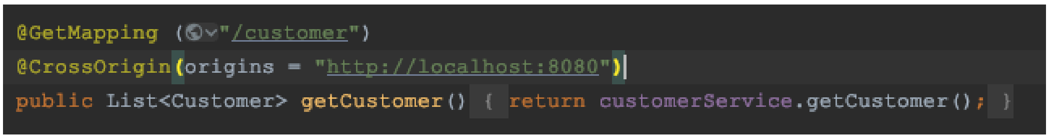
@RequestMapping default configuration is a GET request. With the method configuration in the request mapping one can choose which type of request the method responds to. The most common are GET, POST, DELETE and PUT. Instead of configuring the request mapping it is possible to use directly for example, @GetMapping for GET methods and @PostMapping for POST methods.

[17] https://www.w3schools.com/tags/att_script_crossorigin.asp As of May 2021



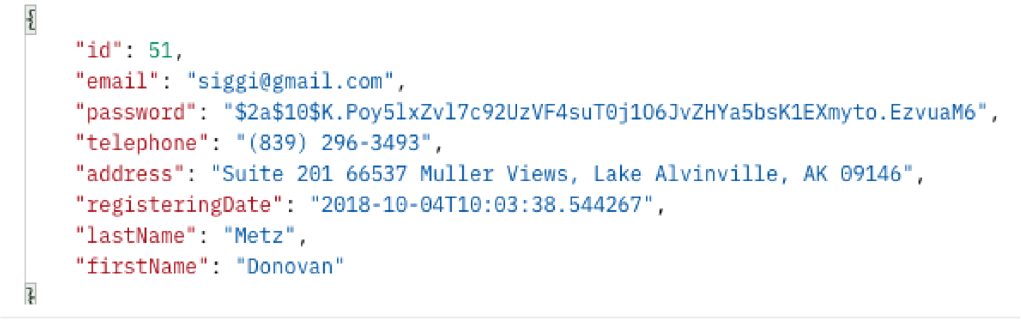
```
@RequestMapping(value = "/products/review", method = RequestMethod.GET)
@CrossOrigin(origins = "http://localhost:8080")
public List<Review> getAllProductsReview() { return reviewService.getAllProductsReview(); }
```

Figure 41: @RequestMapping



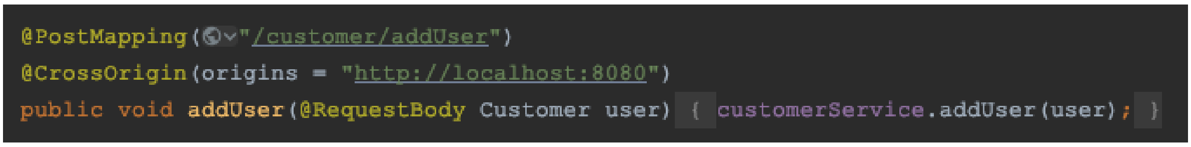
```
@GetMapping ("/customer")
@CrossOrigin(origins = "http://localhost:8080")
public List<Customer> getCustomer() { return customerService.getCustomer(); }
```

Figure 42: @GetMapping



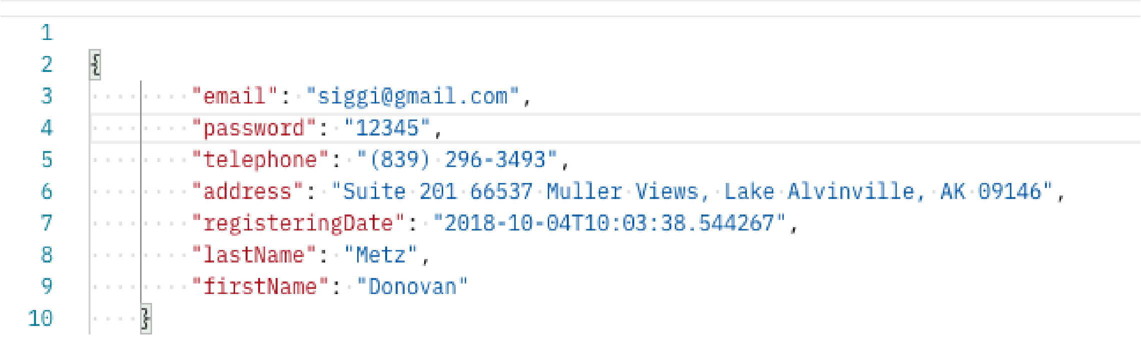
```
{
    "id": 51,
    "email": "siggi@gmail.com",
    "password": "$2a$10$K.Poy5lxZvl7c92UzVF4suT0j1O6JvZHYa5bsK1EXmyto.EzvuaM6",
    "telephone": "(839) 296-3493",
    "address": "Suite 201 66537 Muller Views, Lake Alvinville, AK 09146",
    "registeringDate": "2018-10-04T10:03:38.544267",
    "lastName": "Metz",
    "firstName": "Donovan"
}
```

Figure 43: JSON response from customer GET method with customer id of 51



```
@PostMapping("/customer/addUser")
@CrossOrigin(origins = "http://localhost:8080")
public void addUser(@RequestBody Customer user) { customerService.addUser(user); }
```

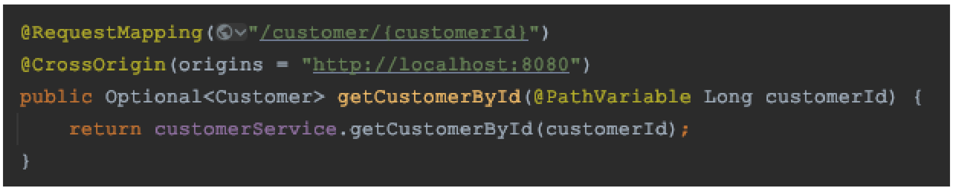
Figure 44: @PostMapping



```

{
    "email": "siggi@gmail.com",
    "password": "12345",
    "telephone": "(839) 296-3493",
    "address": "Suite 201 66537 Muller Views, Lake Alvinville, AK 09146",
    "registeringDate": "2018-10-04T10:03:38.544267",
    "lastName": "Metz",
    "firstName": "Donovan"
}
```

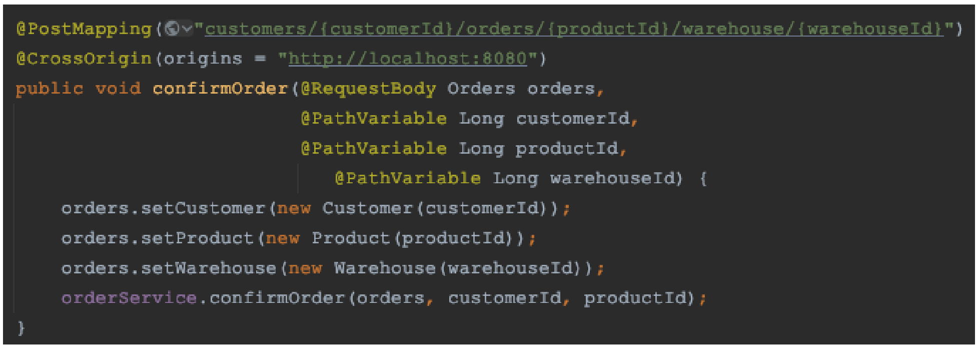
Figure 45: JSON data object sent to warehouse server as Request body in POST method.

By means of POST methods, data is sent to the server. That is done by using @RequestBody annotation. The request body is an JSON object representing the variables needed to instantiate an object on the server. This JSON data is sent as a parameter to the method. Also, as a parameter can be path variable and or request parameter, annotated with @PathVariable and @RequestParam. Both are parameters that come from the request string. One of the ways we use @PathVariable in our project is to get ids. For example, the request string in the controller is “/customer/{customerId}”. The {customerId} is the path variable there. Request call to “http:/localhost:8080/customer/22” will return customer with id 22. @PathVariable can be optional parameter with default setting.



```
@RequestMapping("/customer/{customerId}")
@CrossOrigin(origins = "http://localhost:8080")
public Optional<Customer> getCustomerById(@PathVariable Long customerId) {
    return customerService.getCustomerById(customerId);
}
```

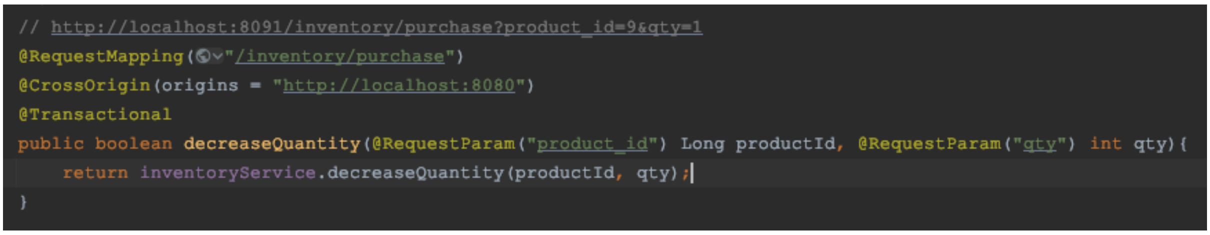
Figure 46: Customer with id @PathVariable.



```
@PostMapping("customers/{customerId}/orders/{productId}/warehouse/{warehouseId}")
@CrossOrigin(origins = "http://localhost:8080")
public void confirmOrder(@RequestBody Orders orders,
                         @PathVariable Long customerId,
                         @PathVariable Long productId,
                            @PathVariable Long warehouseId) {
    orders.setCustomer(new Customer(customerId));
    orders.setProduct(new Product(productId));
    orders.setWarehouse(new Warehouse(warehouseId));
    orderService.confirmOrder(orders, customerId, productId);
}
```

Figure 47: confirmOrder method with three @PathVariables and a @RequestBody.

Request parameter annotated with @Requestparam is also a parameter taken from the request string. Opposed to @PathVariable, request parameter is a required parameter. As shown in Figure 48, the set request string in the controller is “inventory/purchase” but the request from client is on the form “http:localhost8091/inventory/purchase?product_id=9&qty=1”. Here the request parameters product_id and qty have value associated with them as parameters of the method.



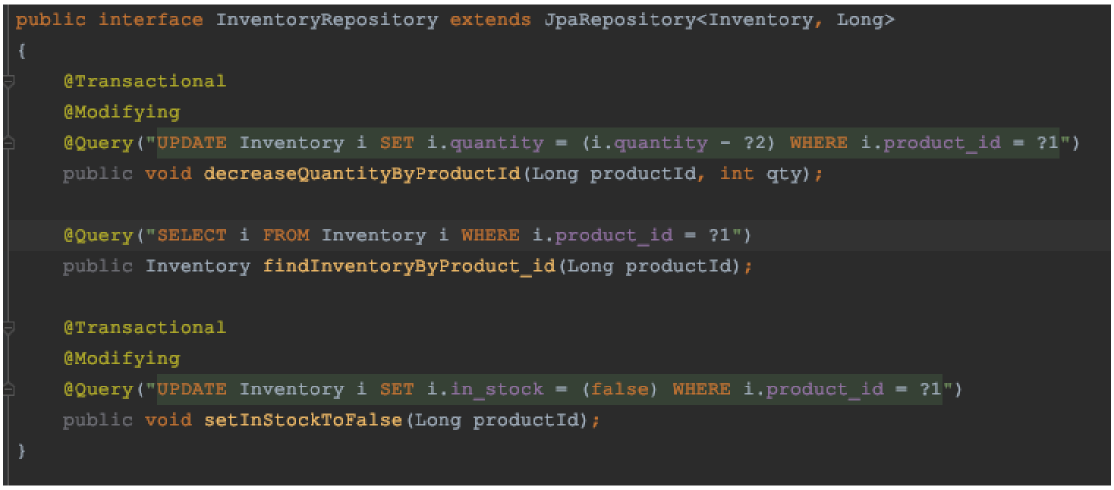
```
// http://localhost:8091/inventory/purchase?product_id=9&qty=1
@RequestMapping("/inventory/purchase")
@CrossOrigin(origins = "http://localhost:8080")
@Transactional
public boolean decreaseQuantity(@RequestParam("product_id") Long productId, @RequestParam("qty") int qty){
    return inventoryService.decreaseQuantity(productId, qty);
}
```

Figure 48: decrease Quantity method with two @RequestParam.

Each of the @RestController class depends on @Service class to handle the business logic of the request.

5.4.4 The service layer @Service, and Data access layer @Repository

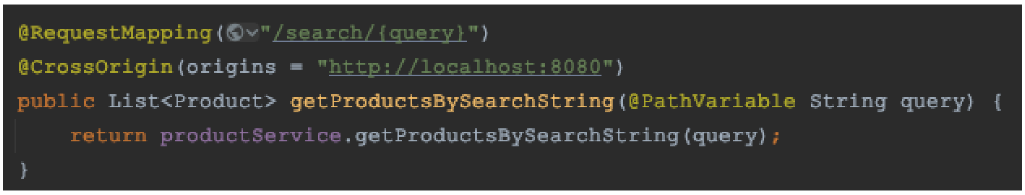
The @Service class handles the business logic of the warehouse application. The class depends on a JPA interface repository or the data access layer. Annotated with @Repository. The JPA is the standard way to save Java object to database as well as retrieve Java objects from database.[18]



```java
public interface InventoryRepository extends JpaRepository<Inventory, Long>
{
    @Transactional
    @Modifying
    @Query("UPDATE Inventory i SET i.quantity = (i.quantity - ?2) WHERE i.product_id = ?1")
    public void decreaseQuantityByProductId(Long productId, int qty);

    @Query("SELECT i FROM Inventory i WHERE i.product_id = ?1")
    public Inventory findInventoryByProduct_id(Long productId);

    @Transactional
    @Modifying
    @Query("UPDATE Inventory i SET i.in_stock = (false) WHERE i.product_id = ?1")
    public void setInStockToFalse(Long productId);
}
```

Figure 49: Inventory repository

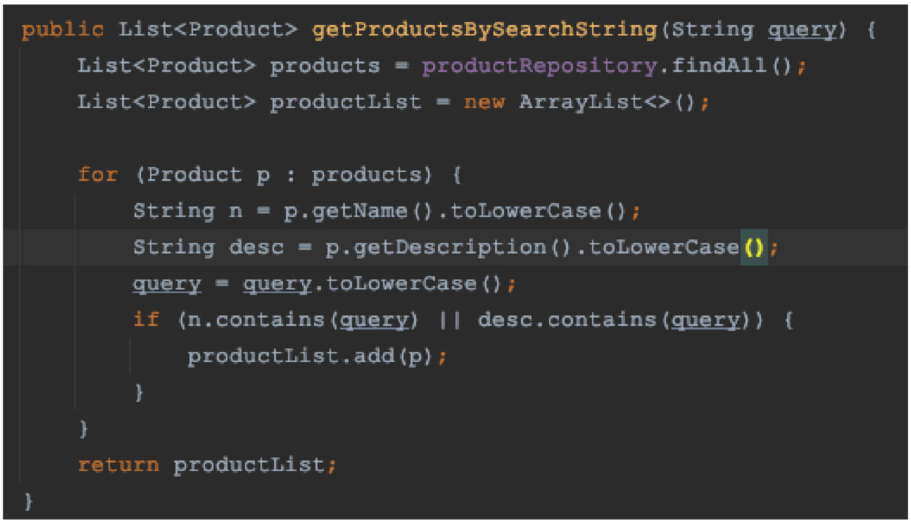
Every method in the control layer is serviced by a method in the service layer. In our project, they have the same method declaration. The service layer methods get request, or method call, from control layer methods. The service method then will use its data access layer to store or retrieve data in the database.



```java
@RequestMapping("/search/{query}")
@CrossOrigin(origins = "http://localhost:8080")
public List<Product> getProductsBySearchString(@PathVariable String query) {
    return productService.getProductsBySearchString(query);
}
```

Figure 50: Get product by search query, from control layer.

[18] https://www.oreilly.com/library/view/spring-data/9781449331863/ch04.html as of May 21

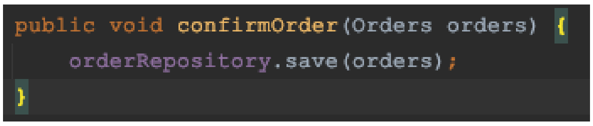


```java
public List<Product> getProductsBySearchString(String query) {
    List<Product> products = productRepository.findAll();
    List<Product> productList = new ArrayList<>();

    for (Product p : products) {
        String n = p.getName().toLowerCase();
        String desc = p.getDescription().toLowerCase();
        query = query.toLowerCase();
        if (n.contains(query) || desc.contains(query)) {
            productList.add(p);
        }
    }
    return productList;
}
```

Figure 51: Get product by search query, from service layer.

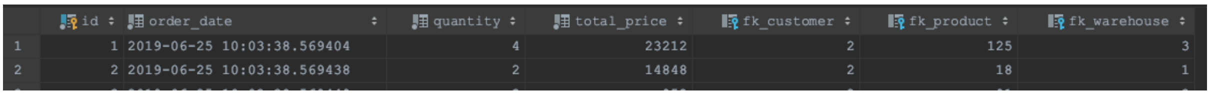
As shown in Figure 51 the method “getProductBySearchString” in service layer gets a search query string from getProductBySearchString method (see Figure 50), in the control layer. The service layer method then gets list of all products from the data access layer. Furthermore, the service layer method instantiates new empty list. It then goes through the product list and matches the string it got and adds to the empty list if it matches. Then it returns this new list with products based on the query string to the control layer and back to the client.

Another example of the service layer is the confirmOrder method. This method takes parameter of type Orders. Java Orders object is generated from JSON data from the @RestController layer and sent to the JPA @Repository layer where it is persisted into the database (see Figure 52). And then the object instance become a row in the database (see Figure 53).



```java
public void confirmOrder(Orders orders) {
    orderRepository.save(orders);
}
```

Figure 52: confirm order method of service layer.



	id	order_date	quantity	total_price	fk_customer	fk_product	fk_warehouse
1	1	2019-06-25 10:03:38.569404	4	23212	2	125	3
2	2	2019-06-25 10:03:38.569438	2	14848	2	18	1

Figure 53: Orders in database.

5.4.5 Full API list for the warehouse and the stores

We use @RestController for each entity in our warehouse application. There are in total seven controller classes with twenty-nine request mappings.

```
http://localhost:8090/products
```

Returns all products in database.

```
http://localhost:8090/products/{categoryId}http://localhost:8090/products/{categoryId}
```

Returns all products by category.

```
http://localhost:8090/product/{productId}
```

Returns product by its id.

```
http://localhost:8090/getProductsIds
```

Returns all products ids. Used by the stores to get the ids.

```
http://localhost:8090/search/{query}http://localhost:8090/search/{query}
```

Returns list of products based on search string or query.

```
http://localhost:8090/products/{categoryId}/addhttp://localhost:8090/products/{categoryId}/add
```

Adds new product based on category.

```
http://localhost:8090/products/{categoryId}/addManyhttp://localhost:8090/products/{categoryId}/addMany
```

Adds list of products based on category.

```
http://localhost:8090/products/{productId}/removehttp://localhost:8090/products/{productId}/remove
```

Deletes product by its id.

```
http://localhost:8090/products/review
```

Returns reviews for all products.

```
http://localhost:8090/products/{productId}/review
```

Returns review for product based on the product id.

```
http://localhost:8090/products/{productId}/addReviewhttp://localhost:8090/products/{productId}/addReview
```

Adds review based on product id.

```
http://localhost:8090/products/{categoryId}/addComputer
```

Adds new computer product.

```
http://localhost:8090/products/{categoryId}/addCamera
```

Adds new camera product.

```
http://localhost:8090/products/{categoryId}/addBook
```

Adds new book product.

`http://localhost:8090/products/{categoryId}/addTVhttp://localhost:8090/products/{categoryId}/addTV`

Adds new tv product.

`http://localhost:8090/categories`

Returns all categories.

`http://localhost:8090/categories/add`

Adds new category.

`http://localhost:8090/customer`

Returns all customers.

`http://localhost:8090/customer/{customerId}http://localhost:8090/customer/{customerId}`

Returns customer by its id.

`http://localhost:8090/customer/addUser`

Adds new customer.

`http://localhost:8090/customer/login`

Returns customer if password is valid.

`http://localhost:8090/orders`

Returns all orders.

`http://localhost:8090/customers/{customerId}/orders`

Returns all orders based on customer id.

`http://localhost:8090/products/{productId}/ordersByProductId`

Returns all orders based on product id.

`http://localhost:8090/customers/{customerId}/orders/{productId}/warehouse/{warehouseId}`

Adds new order to orders based on custormer id, product id and warehouse id.

`http://localhost:8090/warehouse`

Returns list of warehouses.

5.5 Frontend

5.5.1 Frontend Design

We wanted to make a website that will be reactive and also data driven. Therefore, the design needs to be integrated with the warehouse application and have components that will change based on the data it will receive from the warehouses. For instance, the header of the website makes a call to the warehouse to get all the registered categories for the products and will make a spot on the header and a routing to a product component where we will pass the information on which category was selected by the user. By making the design this way, it would make it possible for potential shop owners to simply drop in a new category in the database and the website will show all products related to that category.

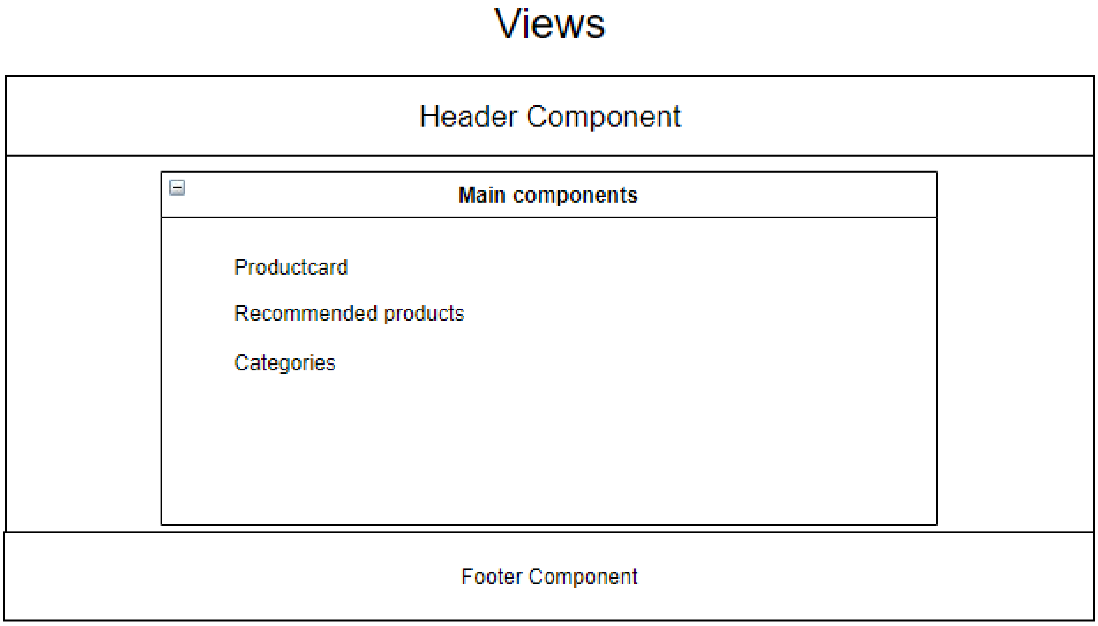
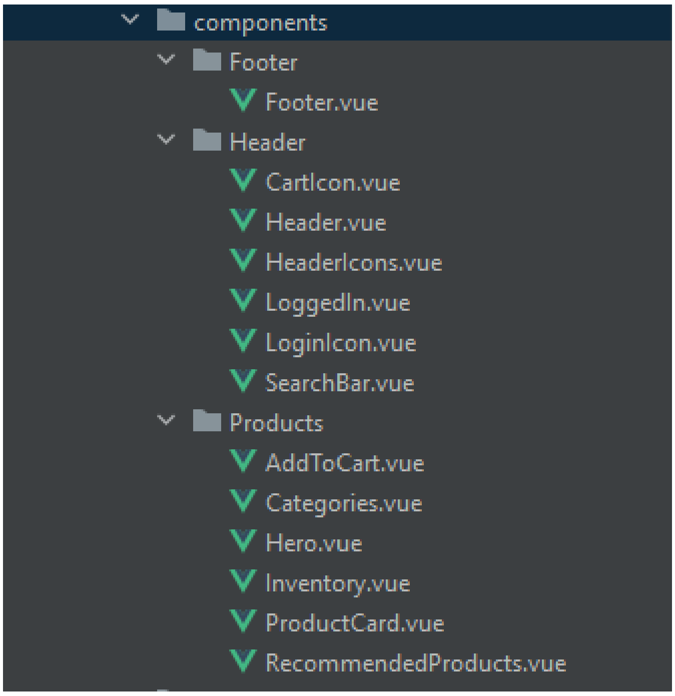


Figure 54: Diagram for views and components

Moreover, it is important to distinguish the difference between a component and a view. The components are swappable, meaning we can reuse them and simply put them into other areas. For instance, the add to cart button is a component in which we can put on a product card or in product information page. Whereas a view can be considered more as a page. An example of this would be login page or the cart.

5.5.2 Components

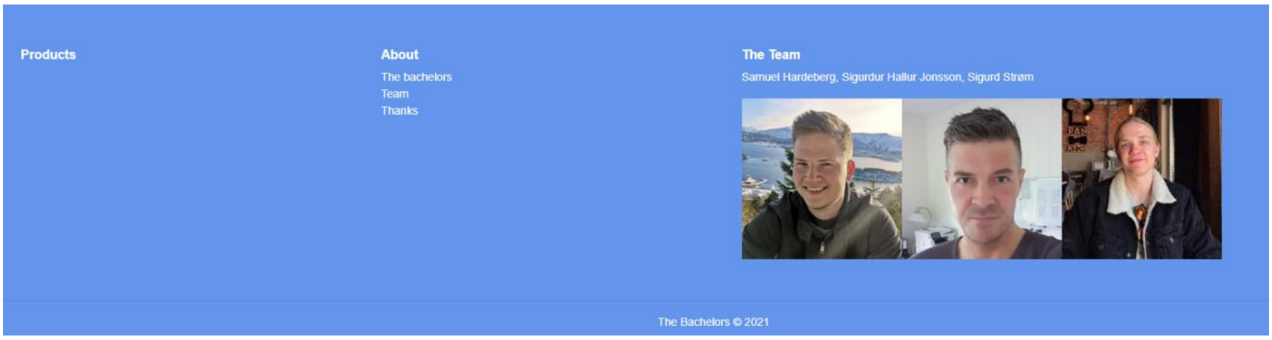
Since Vue is mainly a single paged application, it is important to create components that can swapped in and out of the page when needed. In this project, we have components like: Product card, Categories, search bar, and so on. These can easily be moved and placed in other areas of the website. This makes it very modular and makes it easy to change the layout of the website if needed.



```
components
  Footer
    Footer.vue
  Header
    CartIcon.vue
    Header.vue
    HeaderIcons.vue
    LoggedIn.vue
    LoginIcon.vue
    SearchBar.vue
  Products
    AddToCart.vue
    Categories.vue
    Hero.vue
    Inventory.vue
    ProductCard.vue
    RecommendedProducts.vue
```

Figure 55: Components in the storefront

In Figure 55 we see all the components that make up the storefront. The main function of the footer is to add a couple of "About us" fields where we have some information about the team. (*see Figure 56 below*)



Products

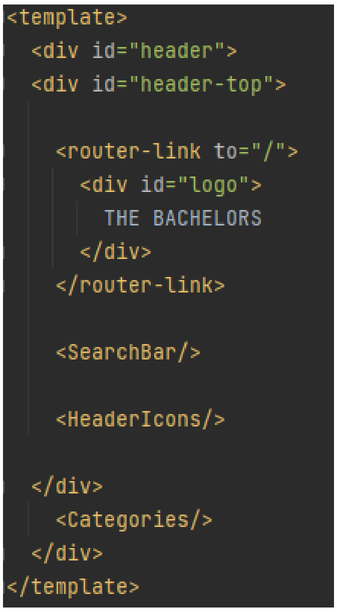
About
The bachelors
Team
Thanks

The Team
Samuel Hardeberg, Sigurdur Hallur Jonsson, Sigurd Strøm

The Bachelors © 2021

Figure 56: Footer component

The Header component is more complex as it contains multiple smaller modular components. By looking at Figure 55, we can see the components included in the header folder is: CartIcon, Header, HeaderIcons, LoggedIn, LoginIcon, and lastly SearchBar. Thus, there are icons for the Cart, Login button and also a changeable icon for LoggedIn. This is because we want to give feedback to the user when they have logged in. Therefore, we change out the normal Icon with a new green logged in icon which gives positive feedback to the users that they are currently logged in. Furthermore, we have small counter on the cart icon that displays the number of products added to cart.



```
<template>
  <div id="header">
  <div id="header-top">

    <router-link to="/">
      <div id="logo">
        THE BACHELORS
      </div>
    </router-link>

    <SearchBar/>

    <HeaderIcons/>

  </div>
    <Categories/>
  </div>
</template>
```

Figure 57: Code snippet from header component

Figure 57 above shows the code from the header component. From this we can clearly see the benefit of defining everything in its own component. As in the header all we need to do it call the components we need using, for instance <SearchBar/>, and importing it in the script section. This adds the search bar inside the header component and makes it very modular and easily changeable.

5.5.3 Vuex Store

The Vuex Store is modular and holds all the states, getters, mutations, and actions that were used in this project. Firstly, I will explain how the states, getters, mutations, and actions operate and tie together. Secondly, the explanation of the modules using an example of category store module.

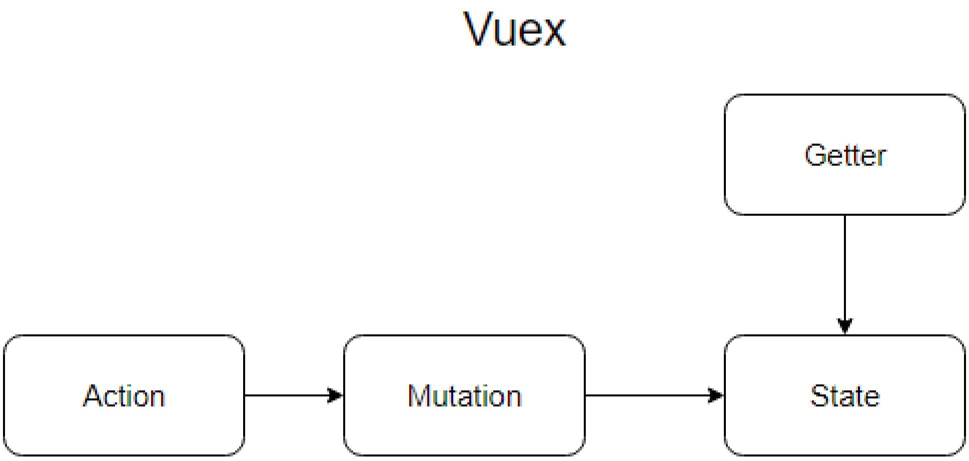
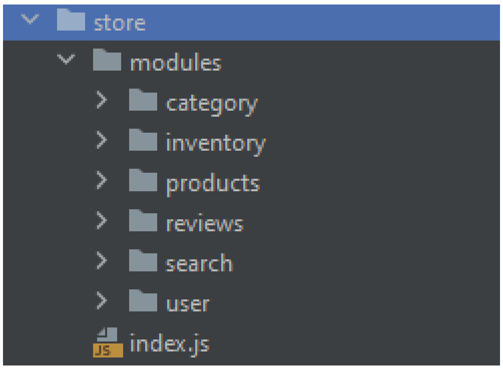


Figure 58: Visualization of Vuex functions

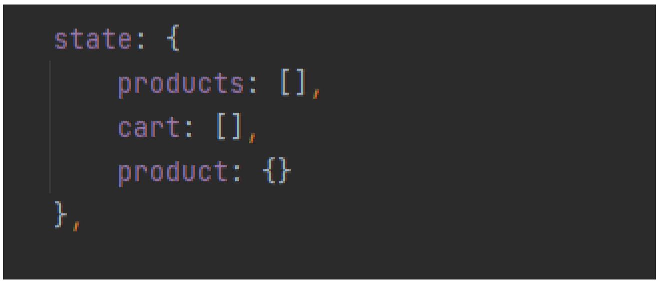
Figure 58 above shows how Vuex functions. We never want to change a state directly, therefore, we make an action that commits its changes to a mutation. An action and mutation are quite similar; however, an action will not change a state directly but instead commit a mutation that will use the payload of the action to change the state. Therefore, only actions and getters will be used in the components of the code as none of this directly change the states' information.



```
store
  modules
    category
    inventory
    products
    reviews
    search
    user
  index.js
```

Figure 59: Vuex Store Modules

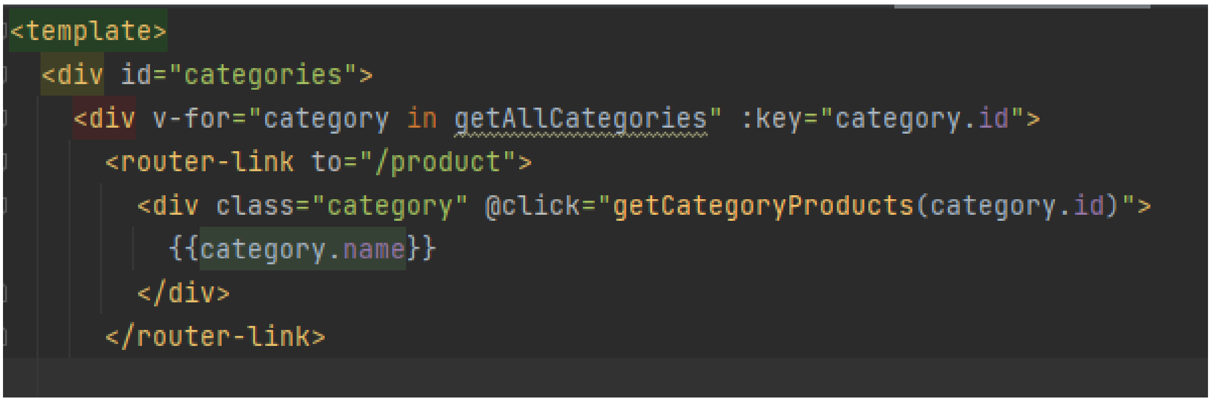
Figure 59 shows all the modules that we have split the store into. Each of these folders contains an index.js file that keeps the relevant states, getters, mutations, and actions. In Figure 60 below, we see the states listed in products module. From the figure, we can see that the module products hold 3 states: Products (Array), Cart (Array), and lastly Product (Object). Each of these states play a valuable role in the storefront. As the products array hold the information of all the products, we will receive from the backend system. Cart will hold all products added to cart and product object holds a single product. The goal of this state is to hold a single product when you click on a product card and want more information on the product.



```
state: {
    products: [],
    cart: [],
    product: {}
},
```

Figure 60: States of products module

The other modules, such as category, hold states that are relevant to obtain categories and displaying them on the website. Therefore, we have category state and category products. The category state will hold the information of all the categories which exists in the database. This state is what we use to display all the categories on the header.



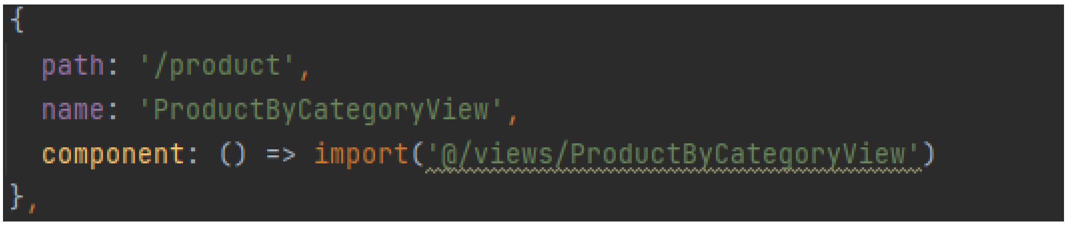
```
<template>
  <div id="categories">
    <div v-for="category in getAllCategories" :key="category.id">
      <router-link to="/product">
        <div class="category" @click="getCategoryProducts(category.id)">
          {{category.name}}
        </div>
      </router-link>
```

Figure 61: Snippet of categories.vue component.

From figure 61 we can see how it will use the Vuex store getter “getAllCategories”, which returns the categories state, to find all categories and create a router link and displays the name for each of the iterated categories. It also includes a method “getCategoryProducts” which will use the category id to find products that are included in the given category.

5.5.4 Routing and views

The routes are defined in a router.js file in the project. This file includes the URL pathing, name and the component that will be shown. In Section 5.5.3, we explained how the categories will be displayed on the website. To continue this example, we will look at what happens once a category has been pressed on the header. From figure 61, we could see the router link leads to “/product”.



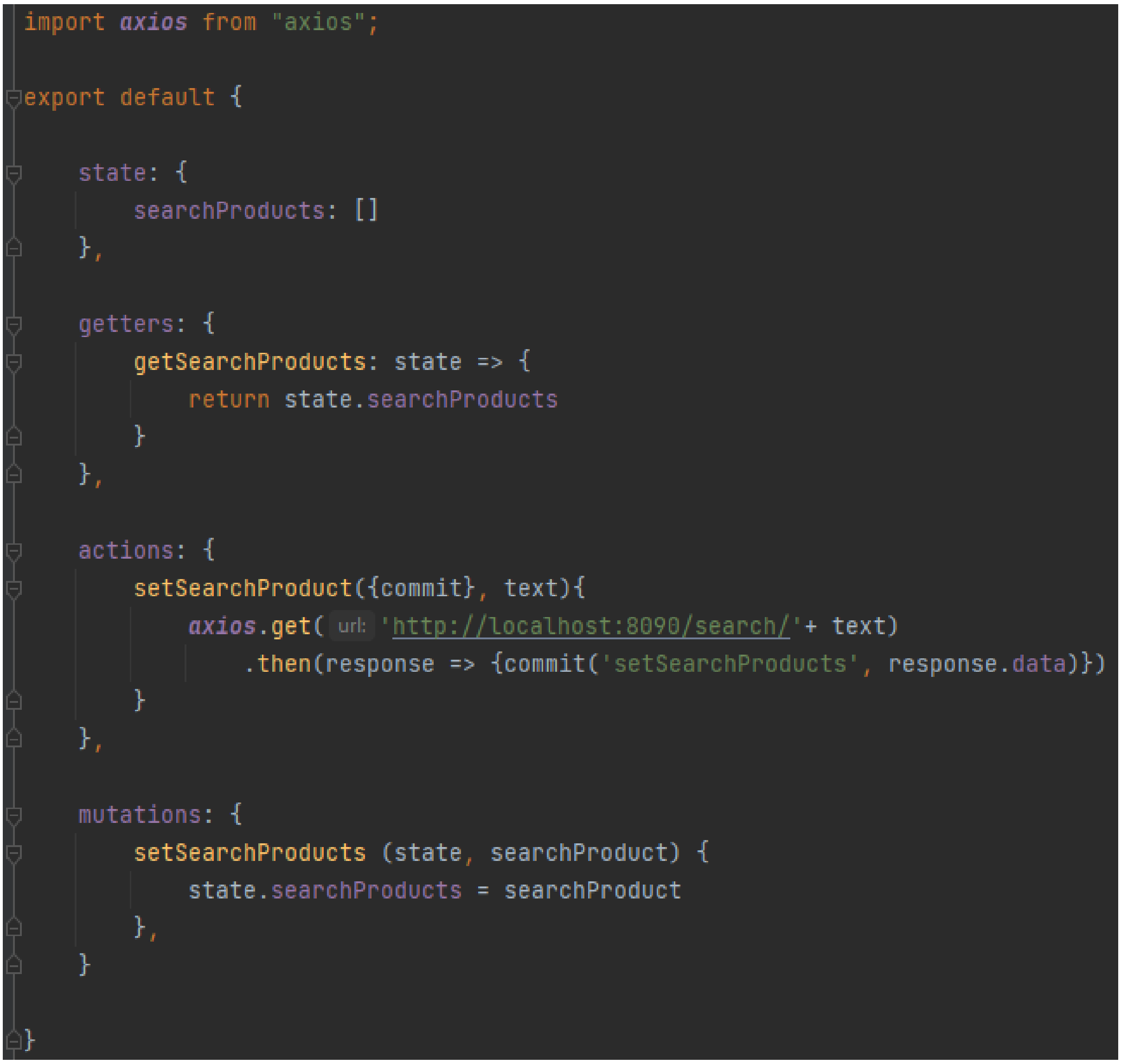
```
{
  path: '/product',
  name: 'ProductByCategoryView',
  component: () => import('@/views/ProductByCategoryView')
},
```

Figure 62: Routing example

Figure 62 essentially shows us that once the router link has been pressed (/product) we will show the “ProductByCategoryView”. The view files are components, but it is easier to think of them as pages. Thus, we move from the page we are currently on and move to the page where the products related to the category will be shown.

5.5.5 Information gathering

For the information gathering the Axios library is used in Vue. Axios allows us to send request to the Restful API. The Axios get requests are stored in the Vuex Store files. When the get requests are called, they will receive the data from the database and put the information into the states. Thus, when we need to display the information we have stored, we will make a call to the store getter and then display it.



```
import axios from "axios";

export default {

    state: {
        searchProducts: []
    },

    getters: {
        getSearchProducts: state => {
            return state.searchProducts
        }
    },

    actions: {
        setSearchProduct({commit}, text){
            axios.get( url: 'http://localhost:8090/search/'+ text)
                .then(response => {commit('setSearchProducts', response.data)})
        }
    },

    mutations: {
        setSearchProducts (state, searchProduct) {
            state.searchProducts = searchProduct
        },
    }

}
```

Figure 63: Vuex search module

In Figure 63 above we can see how the search bar follows these instructions. When the text has been written in the search bar it will be saved as the variable text under actions. Thus, we can see from this example that the action will use text in an Axios get request and send the payload (response data) to the mutation which will change the state. Lastly the getter will simply display the information currently stored in the state “searchProducts”.

6 Results

6.1 Requirement specification

We received a requirement specification from our DRIW AS contact, containing a list of requirements structured into three tiers based on priority:

- **P1** – MVP: Minimum Viable Product (MVP) is the minimum requirements for a working product. The main priority of the project was to ensure that the MVP features were completed.
- **P2** – Wants: After ensuring that the MVP features were in place, some of these features were implemented. Many of the p2 features were closely linked to the development of p1 features.
- **P3 -** Nice-to-haves: Most of these features were deprioritized and thus not implemented, although some of the p3 features were similarly to the p2 features in that they were linked to the development of p1 features.

6.2 Features implemented

All, except for one, P1 (MVP) features were implemented. Most of the P2 features and some of the P3 features were implemented as well. Most of the requirements related to a number of core features:

- Search bar
- Home page with product browsing
- Product pages
- Cart
- Checkout
- User account system
- Warehouse system
- Administrator system
- Rating system

These core features were implemented as following:

6.2.1 Search bar

Requirements:

- P1: The system should allow customers to enter a search query and give back relevant products.
- P3: The system should allow customers to search for properties in a product where name weighs most for relevance.

Implementation:

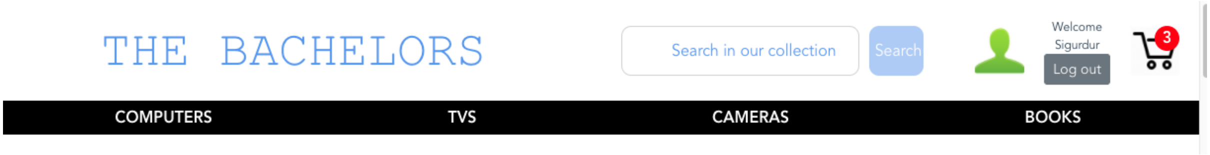


Figure 64: Header with search bar

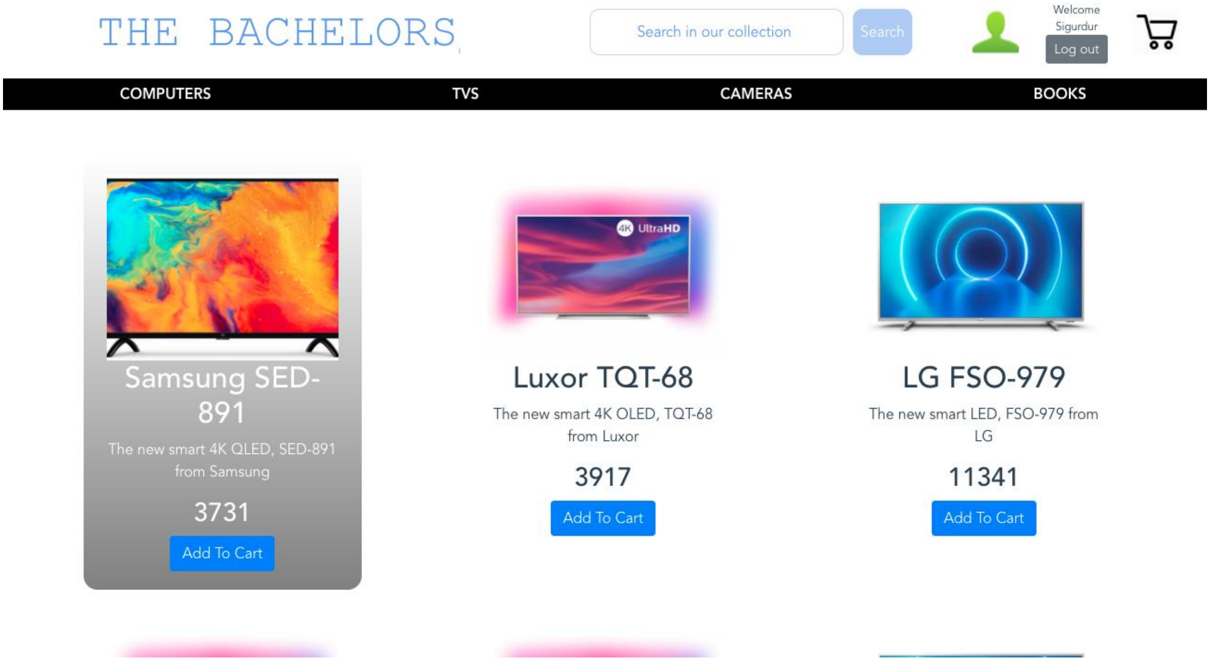
The navigation bar contains a search bar that sends search queries trough Axios request to the database. The search query is applied to various data entry attributes such as the product descriptions and other features, not just the product name.

6.2.2 Product browsing

Requirements:

- P1: The system should display suggested products in the start page.
- P3: The system should display suggested products based on products reviewed.
- P1: The system should build the product categories in the store based on all registered categories in the database.

Implementation:



Samsung SED-891

The new smart 4K QLED, SED-891 from Samsung

3731

Luxor TQT-68

The new smart 4K OLED, TQT-68 from Luxor

3917

LG FSO-979

The new smart LED, FSO-979 from LG

11341

Figure 65: Browsing products

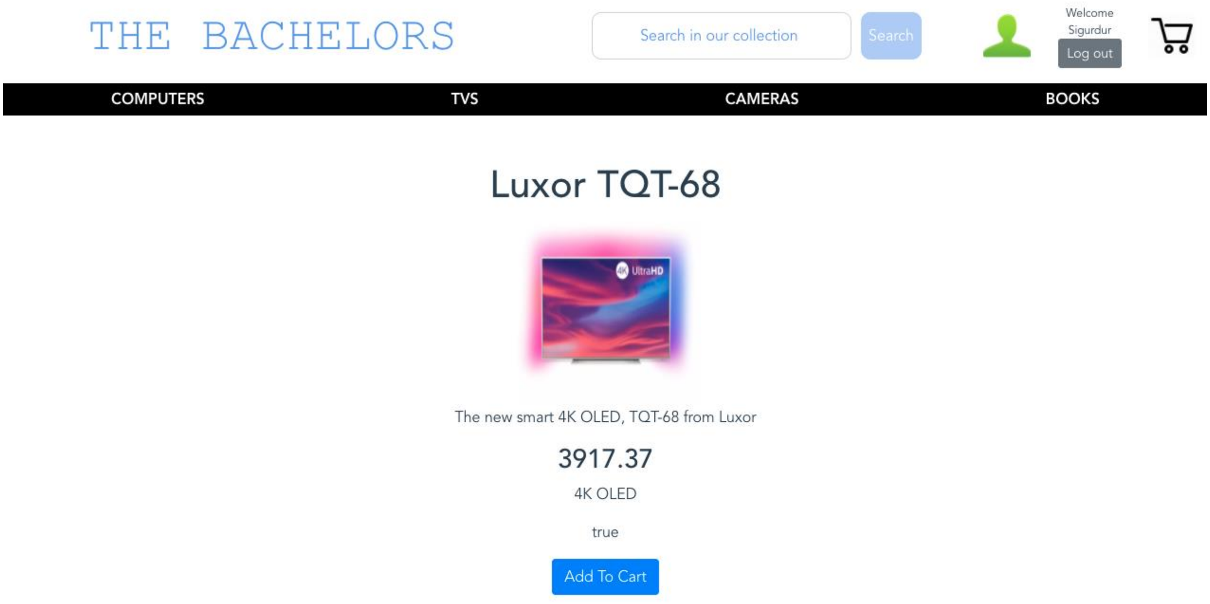
The storefront shows the product categories that are stored in the database. The backend review system determines which products are displayed on the home page.

6.2.3 Product pages

Requirements:

- P1: The system should display product information to customers such as physical properties, name, and description.
- P1: The system should allow customers to check the total available stock available for purchase.

Implementation:



Luxor TQT-68

The new smart 4K OLED, TQT-68 from Luxor

3917.37

4K OLED

true

Figure 66: Product details

Each product from the database(s) are generated a product page for that product. The product page displays:

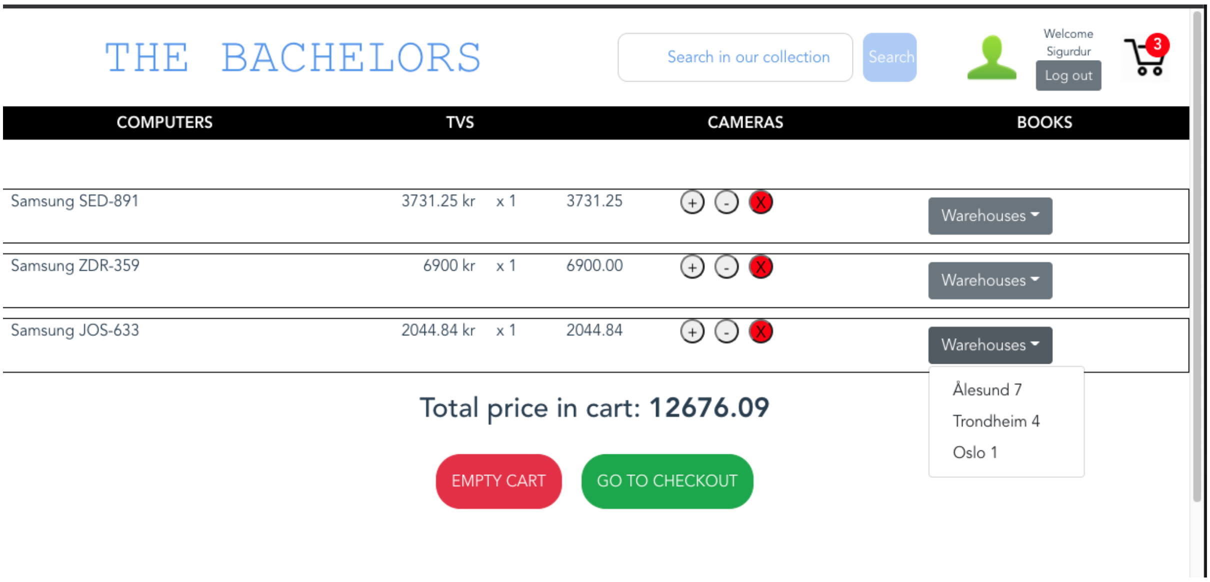
- Picture of the product
- Name of the product
- Description
- Add to cart button

6.2.4 Cart

Requirements:

- P1: The system should allow customers to add products to a shopping cart if it is available.
- P1: The system should allow customers to remove products from a shopping cart.
- P1: The system should allow customers to check the stock available for purchase in each warehouse.

Implementation:



Samsung SED-891	3731.25 kr	x 1	3731.25	Warehouses
Samsung ZDR-359	6900 kr	x 1	6900.00	Warehouses
Samsung JOS-633	2044.84 kr	x 1	2044.84	Warehouses

Ålesund 7
Trondheim 4
Oslo 1

Total price in cart: **12676.09**

Figure 67: Cart with items

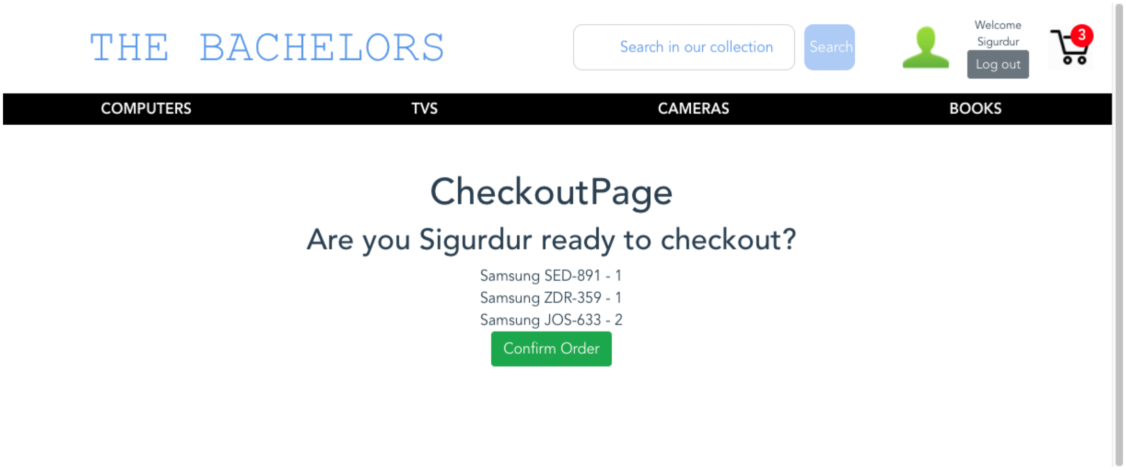
The shopping cart is accessed through an icon of a physical shopping cart in the navigation bar. You can add more of a product or remove products from the shopping cart by clicking the (+) and the (-) on the cart page. You can also choose the warehouse to ship from certain warehouse locations and see the stock in each warehouse.

6.2.5 Checkout

Requirements:

- P1: The system should bring the customer to a checkout page listing the name, price, and quantity selected of each item.
- P1: The system should give the customer an order confirmation after confirming the purchase, this is also stored in the database.
- P1: The system should display to the customer which warehouse each product is sent from after confirming the order.

Implementation:



CheckoutPage

Are you Sigurdur ready to checkout?

Samsung SED-891 - 1
Samsung ZDR-359 - 1
Samsung JOS-633 - 2

Figure 68: Checkout page

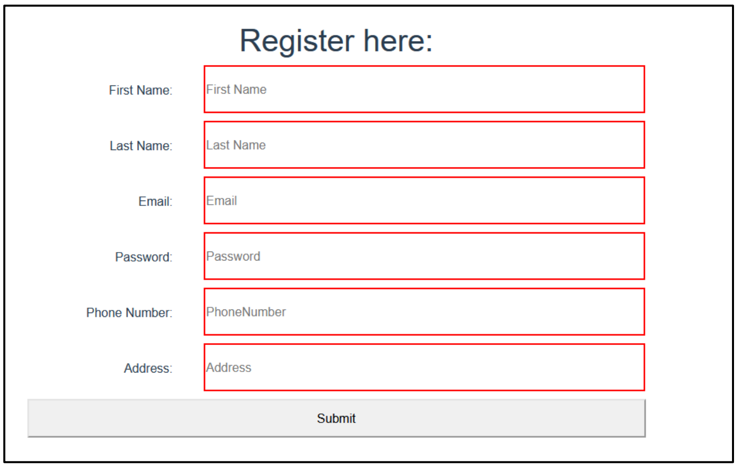
The checkout page shows each product with its corresponding quantity. The page displays a personalized message to the user asking for confirmation to checkout.

6.2.6 User account

- P1: The system should allow the customer to register a user account by providing a username and password.
- P1: The system should allow the customer to log in to the page.

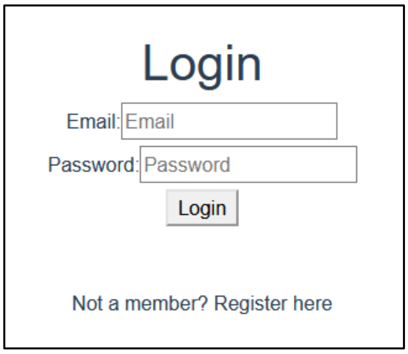
Implementation:

The navigation bar contains an icon that links to the login page. The login page has two input text boxes, *email*, and *password*. It also has a registering page, which is accessed through the link below the login interface.



Register here:

First Name:	First Name
Last Name:	Last Name
Email:	Email
Password:	Password
Phone Number:	PhoneNumber
Address:	Address



Login

Email: Email

Password: Password

Figure 69: Login for users

Figure 70: Register new user

6.2.7 Warehouse

Requirements:

- P1: The system should have its own database (cannot share with other instances) and be its own application instance.
- P1: The system should only be able to communicate with other applications through HTTP(S) and it is not allowed to access other warehouse databases directly.
- P2: The system should have a way of automatically generating test products at scale.
- P1: The system should allow reading product information without authentication.

Implementation:

All the requirements of the warehouse where fulfilled. The warehouse is its own application with database. Each of the stores are its own application with database. The communicate with each other and with the storefront via http requests. We made script within both the warehouse and the stores to auto generate test data. No authentications are needed to read product information.

6.2.8 Non-functional requirements

These requirements are not part of the core features of the system.

Requirements	**Implementation**
P1: The user should only have to visit one URL to reach the store.	The website is accessed through a single URL that links to the storefront's homepage.
P2: The user should always know the content of the shopping cart.	The cart shows a number for each product in the cart, showing how many of that product is in the cart.
P2: The user should know if he/she is logged in.	There is a login-icon that is green when logged in, red when not logged in.
P2: Any user interaction should take no longer than 2 seconds to give feedback to the user	Every action happens virtually instantaneously. (the website is only running locally, as agreed upon by our client)

P1: User credentials must be secured when in storage.	User credentials are encoded before it is stored using the Spring framework dependency “BCryptPasswordEncoder”[19].
P1: The system should have at least 3 warehouses.	The system has 3 warehouses, for three different physical warehouse locations.

6.3 Partly finished features

These features were implemented but remains incomplete.

6.3.1 Rating system

Requirements:

- P2: The system should allow customers to check a total rating for a product of 1-5 stars where half stars are allowed.
- P3: The system should allow customers to read individual reviews with their star rating included.
- P2: The system should let a logged in user submit a review with a rating (1-5) and comment of a product.

Implementation:

The rating system was fully implemented backend-wise, as the products stored has a rating attribute - but the rating is not displayed on the website. There is also no interface to add ratings to products from the user. However, the website uses the stored ratings for each product to determine which ware is displayed on the home page.

[19] https://docs.spring.io/spring-security/site/docs/4.2.20.RELEASE/apidocs/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html as of May 2021

6.3.2 Other features

Requirements	Implementation
P2: The system should display all completed orders for a customer	The database stores every completed order for the user. But this information is not displayed on the website.
P2: The system should be able have more warehouses with minimal work.	It is simple to add new warehouses by programming the backend system, but there is no frontend interface to add new warehouses.
P1: The system should tell the store its stock on start and as it updates.	The stock updates as the user launches the website, but there is no update interval.

6.4 Deprioritized features

Among the P2 and P3 features, there were features that were not implemented:

- P3: The system should display and take into account tax (here 25%) in the total price.
- P2: The system should allow the customer to cancel their order if done within a given timeframe.
- P3: The system should allow the customer to change their password.
- P3: The user should reach the order history page using maximum 2 clicks.
- P3: User credentials must be secured at transport.
- P2: The stock in the store should not be out of sync longer than 5 seconds.

The features were not implemented due to the projects time constraint. *(see 7.1.1)* The reasoning for deprioritizing these features, were the prioritization hierarchy of the requirement specification from our client, DRIW AS. The features not implemented were all p2 and p3 requirements. Furthermore, most of them were classified as non-functional. The p2 and p3 requirements related to creating an *administrator system* was not deprioritized, but rather excluded due to time constraints; This could be implemented in future work. *(See 7.2.1)*

7 Discussion

7.1 Challenges and limitations

7.1.1 Time management

A challenge to our project was managing our time for the duration of the product. In our original project plan, we made a poor estimate of the time needed on some parts of the project. This became most evident during the first and second sprint, when we were learning about the spring framework and vue.js framework for the warehouse management system and storefront, respectively. The two frameworks proved to be much more time consuming to learn than previously estimated.

7.1.2 Vue and Vuex

When we were learning about the Vue framework, we were simultaneously developing the project with Vue. The development of the frontpage were, particularly in focus. However, during the fourth sprint the group discovered more limitations with the vue.js framework. In particular, implementing the cart system in Vue was a challenge. This was because the cart system among other features heavily relied on Vue's prop system, the more complex the Vue system there is, the more flow of data there is. As we reached more complex features that relied on more data transfer, such as the cart system, it was evident that we needed a better foundation for the storefront.

The solution was to re-implement the storefront in Vuex. Vuex's state management pattern meant that the reworked storefront was now state based. In addition to the system becoming much more scalable and data-driven, implementing the product information pages and cart system was much easier.

7.2 Future work

Although the project was finished, there was certain features that could be added, and room for improving existing features:

7.2.1 Administrator system

Among the P2 “wants” category of requirements, there were several requirements relating to the implementation of an administrator system. The requirements were as following:

- P2: The system should allow an administrator to log into administration pages.
- P2: The system should allow an administrator to add, remove, and update product information in any warehouse.
- P2: The system should only allow writing to production data by an authenticated administrator.

Given that they were a P2 requirement, they were not integral to creating a functioning system. However, having an administrator back-door is very helpful for scalability. If the storefront were going to have real clients, it would need administrators to add, remove, and update product information. And for those hypothetical administrators, having access to an administrator page would be a great tool.

In potential future work, an administrator system would be implemented as following:

Admin page made with VUE where admin logs in. From there admin will fill out form with new product to add to the warehouse application. Filled out form will be sent via http request to the RESTful API. Admin will be able to retrieve information about all products and full inventory list from all stores in the system. Furthermore, admin will be able to adjust stock in each store. Some of the backend implementation is already in place.

7.2.2 Web design

Our storefront website was implemented with a purposeful design; It is made clear to the user that the website is a storefront. It also has an inherent simplicity as the styling is minimal. Other than its purposeful design and simplicity, there has not been any explicit development on the design of the website during the project. This was a conscious decision, as there was not a single requirement from DRIW AS related to web design. It was made clear that the aim was for the project to be centered around developing a functional storefront and warehouse system rather than making an attractive website. However, we still recognize the principles of good web design as relevant to any web development project and have made some ideas for potential improvements for future work.

7.2.2.1 Consistent styling

For a more visually appealing website, there should be few colors, less than 5. These colors must also complement each other. Furthermore, there are some colors that are strongly associated with certain actions. For example, red is associated with deletion, warnings, and otherwise negative messages. Green is associated with addition and otherwise positive messages. Typography also needs to be appealing and consistent. There are a multitude of fonts available, and each one of them affect the user experience differently. Images as well play a role in the visual aesthetics of the website. All of these visual elements also need to complement each other. Given more time, there would have been some focus on styling the website.

7.2.2.2 Responsive web design

As stated, there were no requirements related to web design. As such, there were no requirements in implementing responsive web design with mobile devices in mind. But, with mobile browsing being mainstream and increasingly so, responsive web design and mobile web design is somewhat expected.

7.3 Testing

Testing of the Vue program was done using Vue’s own “DevTool” in the browser. This tool allows us to see all information being stored in the browser and the current value of all methods and states. By using this tool, we could test methods and make sure that the values are the same as expected. Serving as a double check but also as a debug tool when bugs were found. If the values were not matching our expectations, we could trace the values and figure out the root of the issue.

8 Conclusion

The main objective of this bachelor’s thesis was to create an e-commerce storefront which will be able to communicate with multiple warehouses through HTTP and keep stock information of each warehouse. We managed to do all but one of the P1 Requirements and quite a few of the P2 requirements as well. There were two quite big hurdles in our progress, namely the discussion about using NoSQL and the change of project structure to fit with Vuex. Whereas the biggest change was switching to Vuex, as we created a new repository for the project and had to rebuild the foundation and main functions of the project. After changing it to work with Vuex we had great progress as most of the main problems we faced without Vuex could now be solved using the Vuex store states. Furthermore, focus could now be put on implementing new features instead of testing and bug fixing. Despite being behind schedule we managed to solve the task and create a feature rich storefront which communicates to the three warehouses through HTTP. Moreover, all but one P1 requirement was completed and quite a few P2 requirements. The main constraint for not finishing all the P1, P2 and P3 requirements was time. Nonetheless, we consider the project a success as it is fully functional, is feature rich and mostly lack visual styling as this was not a requirement and focused on how we would solve the communication between multiple warehouses and not the aesthetical design.

To summarize, we have created an e-commerce storefront with the addition of being able to check stock information per warehouse. Created a platform where it can take in data such as categories in the database and display and create buttons for it automatically. And furthermore, the ability to choose which warehouse the items added to cart would be shipped or bought from.

9 Bibliography

[1] Agile Manifesto Authors,“ Agile 101”. https://www.agilealliance.org/agile101/ (visited April 2021)

[2] Refsnes data, ”HTML Introduction”. https://www.w3schools.com/html/html_intro.asp (visited April 2021)

[3] Refsnes data, “CSS Introduction”. https://www.w3schools.com/css/css_intro.asp (visited April 2021)

[4] Refsnes data, ”JavaScript Introduction”. https://www.w3schools.com/js/js_intro.asp (visited April 2021)

[5] Refsnes data, “JSON Introduction”. https://www.w3schools.com/js/js_json_intro.asp (visited April 2021)

[6] The PostgreSQL Global Development Group, “About”. https://www.postgresql.org/about/ (visited May 2021)

[7] The PostgreSQL Global Development Group, “What is PostgreSQL”. https://www.postgresqltutorial.com/what-is-postgresql/ (visited May 2021)

[8] Evan You, “Introduction”. https://vuejs.org/v2/guide/ (visited May 2021)

[9] Google, “Angular.io”. https://angular.io/ (visited May 2021)

[10] Facebook, “React”. https://reactjs.org/ (visited May 2021)

[11] Evan You, “VueX”. https://vuex.vuejs.org/ (visited May 2021)

[12] Vmware, “Spring Framework”. https://spring.io/projects/spring-framework (visited May 2021)

[13] Vasyl Redka, “A Beginner’s Tutorial for Understanding RESTful API”. https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api (visited May 2021)

[14] Wikipedia, “Spring Framework”. https://en.wikipedia.org/wiki/Spring_Framework (visited May 2021)

[15] Baeldung, “A Comparison Between Spring and Spring Boot”. https://www.baeldung.com/spring-vs-spring-boot (visited May 2021)

[16] Refsnes data, “HTML <script> crossorigin Attribute”. https://www.w3schools.com/tags/att_script_crossorigin.asp (visited May 2021)

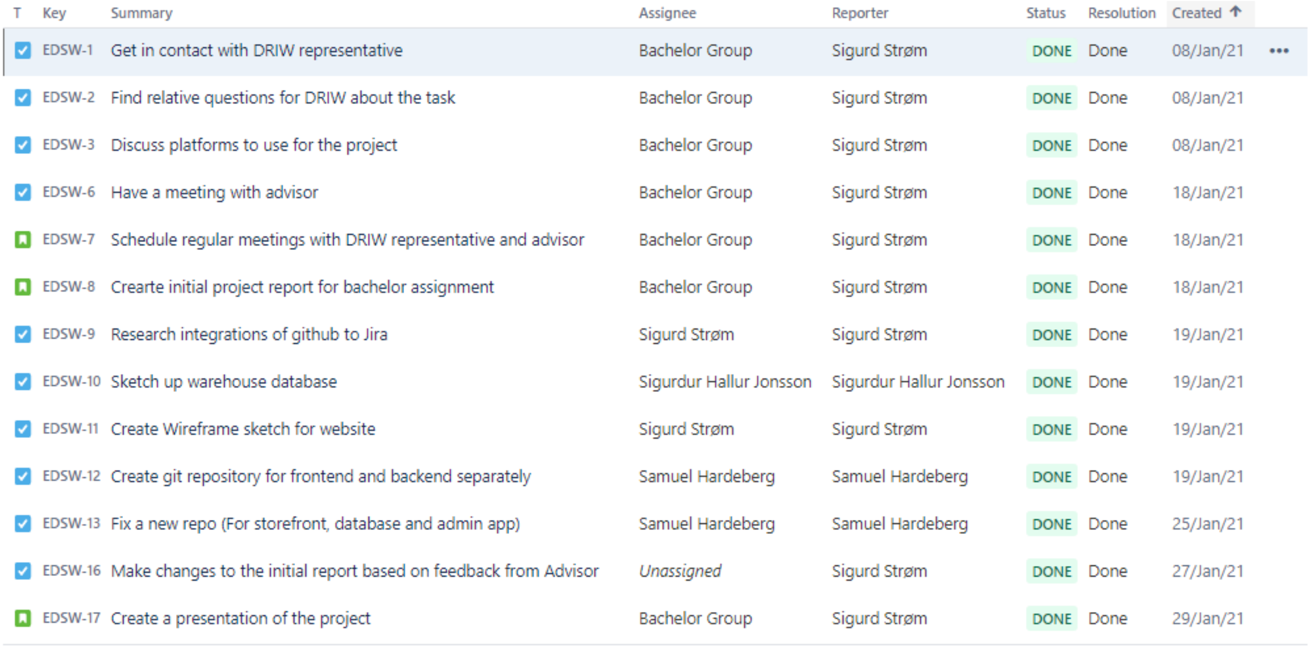
[17] O’Reilly Media, Inc, “Chapter 4. JPA Repositories”. https://www.oreilly.com/library/view/spring-data/9781449331863/ch04.html (visited May 2021)

[18] Vmware, “Class BCryptPasswordEncoder”. https://docs.spring.io/spring-security/site/docs/4.2.20.RELEASE/apidocs/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html (visited May 2021)

10 Appendix

10.1 Jira Issues Log starting from sprint 1 → 6:

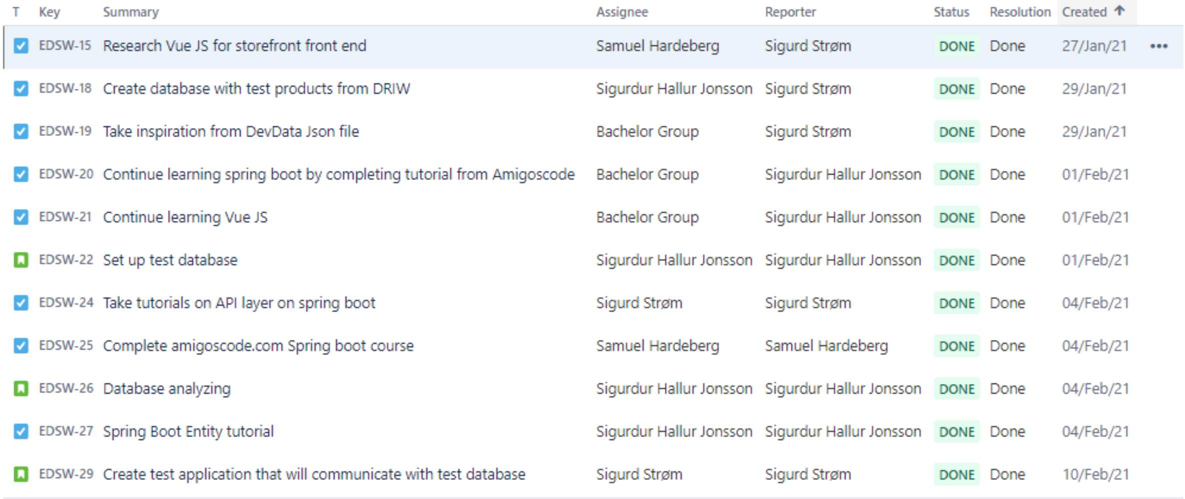
Sprint 1:



T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑	
☑	EDSW-1	Get in contact with DRIW representative	Bachelor Group	Sigurd Strøm	DONE	Done	08/Jan/21	•••
☑	EDSW-2	Find relative questions for DRIW about the task	Bachelor Group	Sigurd Strøm	DONE	Done	08/Jan/21	
☑	EDSW-3	Discuss platforms to use for the project	Bachelor Group	Sigurd Strøm	DONE	Done	08/Jan/21	
☑	EDSW-6	Have a meeting with advisor	Bachelor Group	Sigurd Strøm	DONE	Done	18/Jan/21	
🟩	EDSW-7	Schedule regular meetings with DRIW representative and advisor	Bachelor Group	Sigurd Strøm	DONE	Done	18/Jan/21	
🟩	EDSW-8	Crearte initial project report for bachelor assignment	Bachelor Group	Sigurd Strøm	DONE	Done	18/Jan/21	
☑	EDSW-9	Research integrations of github to Jira	Sigurd Strøm	Sigurd Strøm	DONE	Done	19/Jan/21	
☑	EDSW-10	Sketch up warehouse database	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	19/Jan/21	
☑	EDSW-11	Create Wireframe sketch for website	Sigurd Strøm	Sigurd Strøm	DONE	Done	19/Jan/21	
☑	EDSW-12	Create git repository for frontend and backend separately	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	19/Jan/21	
☑	EDSW-13	Fix a new repo (For storefront, database and admin app)	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	25/Jan/21	
☑	EDSW-16	Make changes to the initial report based on feedback from Advisor	*Unassigned*	Sigurd Strøm	DONE	Done	27/Jan/21	
🟩	EDSW-17	Create a presentation of the project	Bachelor Group	Sigurd Strøm	DONE	Done	29/Jan/21	

Figure 71: Jira log sprint 1

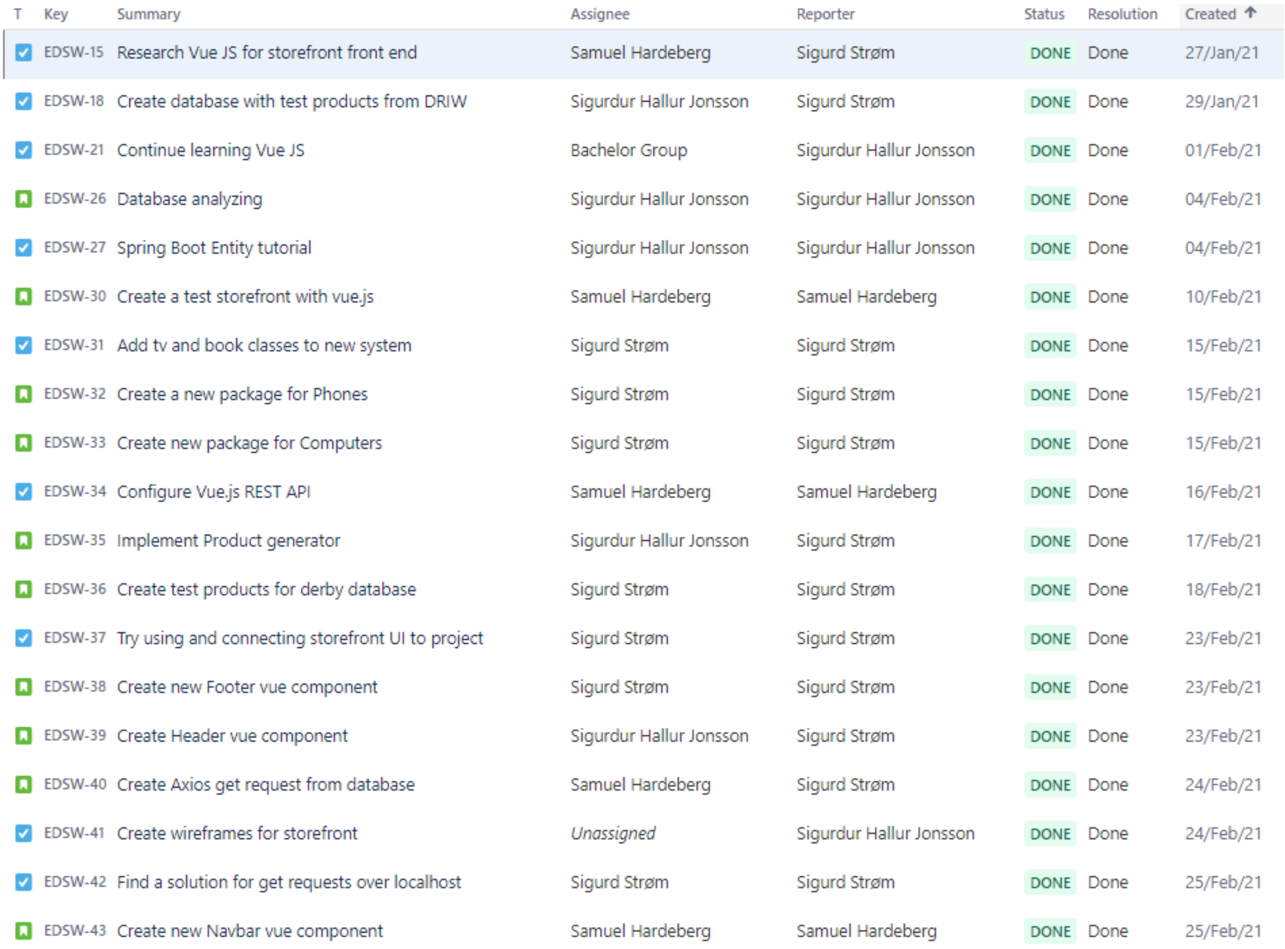
Sprint 2:



T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑
☑	EDSW-15	Research Vue JS for storefront front end	Samuel Hardeberg	Sigurd Strøm	DONE	Done	27/Jan/21
☑	EDSW-18	Create database with test products from DRIW	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	29/Jan/21
☑	EDSW-19	Take inspiration from DevData Json file	Bachelor Group	Sigurd Strøm	DONE	Done	29/Jan/21
☑	EDSW-20	Continue learning spring boot by completing tutorial from Amigoscode	Bachelor Group	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
☑	EDSW-21	Continue learning Vue JS	Bachelor Group	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
🟩	EDSW-22	Set up test database	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
☑	EDSW-24	Take tutorials on API layer on spring boot	Sigurd Strøm	Sigurd Strøm	DONE	Done	04/Feb/21
☑	EDSW-25	Complete amigoscode.com Spring boot course	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	04/Feb/21
🟩	EDSW-26	Database analyzing	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
☑	EDSW-27	Spring Boot Entity tutorial	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
🟩	EDSW-29	Create test application that will communicate with test database	Sigurd Strøm	Sigurd Strøm	DONE	Done	10/Feb/21

Figure 72: Jira log sprint 2

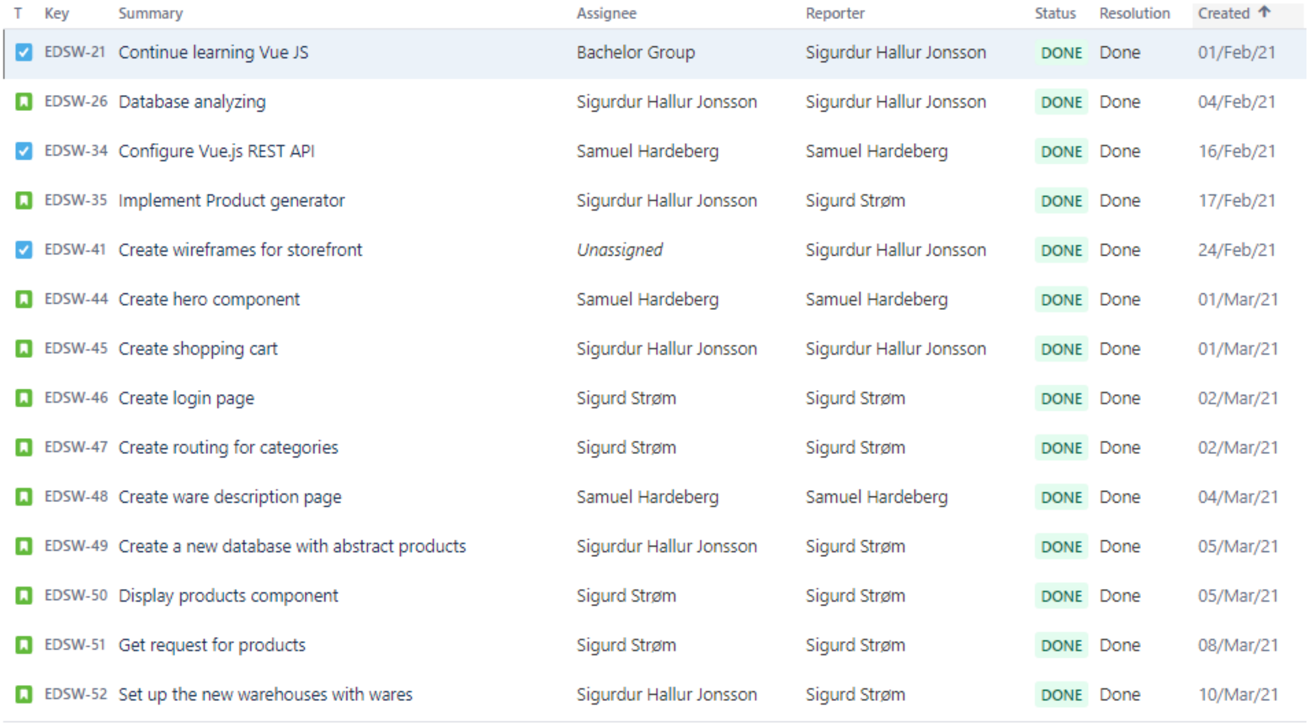
Sprint 3:



T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑
☑	EDSW-15	Research Vue JS for storefront front end	Samuel Hardeberg	Sigurd Strøm	DONE	Done	27/Jan/21
☑	EDSW-18	Create database with test products from DRIW	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	29/Jan/21
☑	EDSW-21	Continue learning Vue JS	Bachelor Group	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
🔖	EDSW-26	Database analyzing	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
☑	EDSW-27	Spring Boot Entity tutorial	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
🔖	EDSW-30	Create a test storefront with vue.js	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	10/Feb/21
☑	EDSW-31	Add tv and book classes to new system	Sigurd Strøm	Sigurd Strøm	DONE	Done	15/Feb/21
🔖	EDSW-32	Create a new package for Phones	Sigurd Strøm	Sigurd Strøm	DONE	Done	15/Feb/21
🔖	EDSW-33	Create new package for Computers	Sigurd Strøm	Sigurd Strøm	DONE	Done	15/Feb/21
☑	EDSW-34	Configure Vue.js REST API	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	16/Feb/21
🔖	EDSW-35	Implement Product generator	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	17/Feb/21
🔖	EDSW-36	Create test products for derby database	Sigurd Strøm	Sigurd Strøm	DONE	Done	18/Feb/21
☑	EDSW-37	Try using and connecting storefront UI to project	Sigurd Strøm	Sigurd Strøm	DONE	Done	23/Feb/21
🔖	EDSW-38	Create new Footer vue component	Sigurd Strøm	Sigurd Strøm	DONE	Done	23/Feb/21
🔖	EDSW-39	Create Header vue component	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	23/Feb/21
🔖	EDSW-40	Create Axios get request from database	Samuel Hardeberg	Sigurd Strøm	DONE	Done	24/Feb/21
☑	EDSW-41	Create wireframes for storefront	*Unassigned*	Sigurdur Hallur Jonsson	DONE	Done	24/Feb/21
☑	EDSW-42	Find a solution for get requests over localhost	Sigurd Strøm	Sigurd Strøm	DONE	Done	25/Feb/21
🔖	EDSW-43	Create new Navbar vue component	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	25/Feb/21

Figure 73: Jira log sprint 3

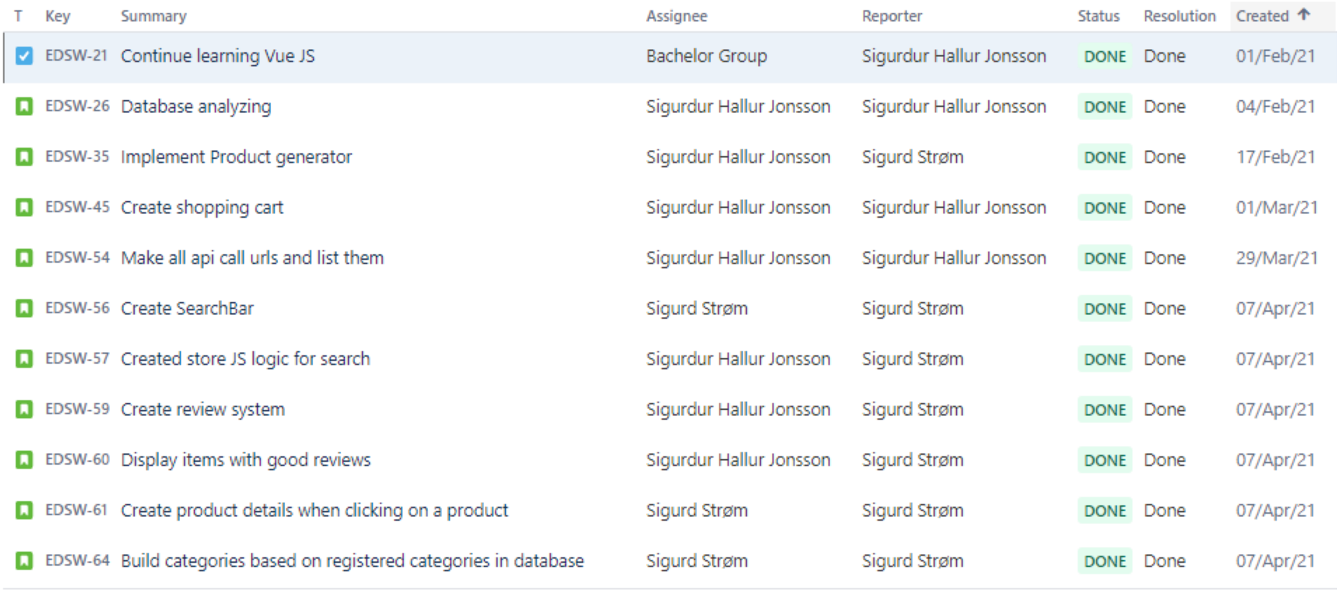
Sprint 4:



T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑
☑	EDSW-21	Continue learning Vue JS	Bachelor Group	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
🟩	EDSW-26	Database analyzing	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
☑	EDSW-34	Configure Vue.js REST API	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	16/Feb/21
🟩	EDSW-35	Implement Product generator	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	17/Feb/21
☑	EDSW-41	Create wireframes for storefront	*Unassigned*	Sigurdur Hallur Jonsson	DONE	Done	24/Feb/21
🟩	EDSW-44	Create hero component	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	01/Mar/21
🟩	EDSW-45	Create shopping cart	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	01/Mar/21
🟩	EDSW-46	Create login page	Sigurd Strøm	Sigurd Strøm	DONE	Done	02/Mar/21
🟩	EDSW-47	Create routing for categories	Sigurd Strøm	Sigurd Strøm	DONE	Done	02/Mar/21
🟩	EDSW-48	Create ware description page	Samuel Hardeberg	Samuel Hardeberg	DONE	Done	04/Mar/21
🟩	EDSW-49	Create a new database with abstract products	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	05/Mar/21
🟩	EDSW-50	Display products component	Sigurd Strøm	Sigurd Strøm	DONE	Done	05/Mar/21
🟩	EDSW-51	Get request for products	Sigurd Strøm	Sigurd Strøm	DONE	Done	08/Mar/21
🟩	EDSW-52	Set up the new warehouses with wares	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	10/Mar/21

Figure 74: Jira log sprint 4

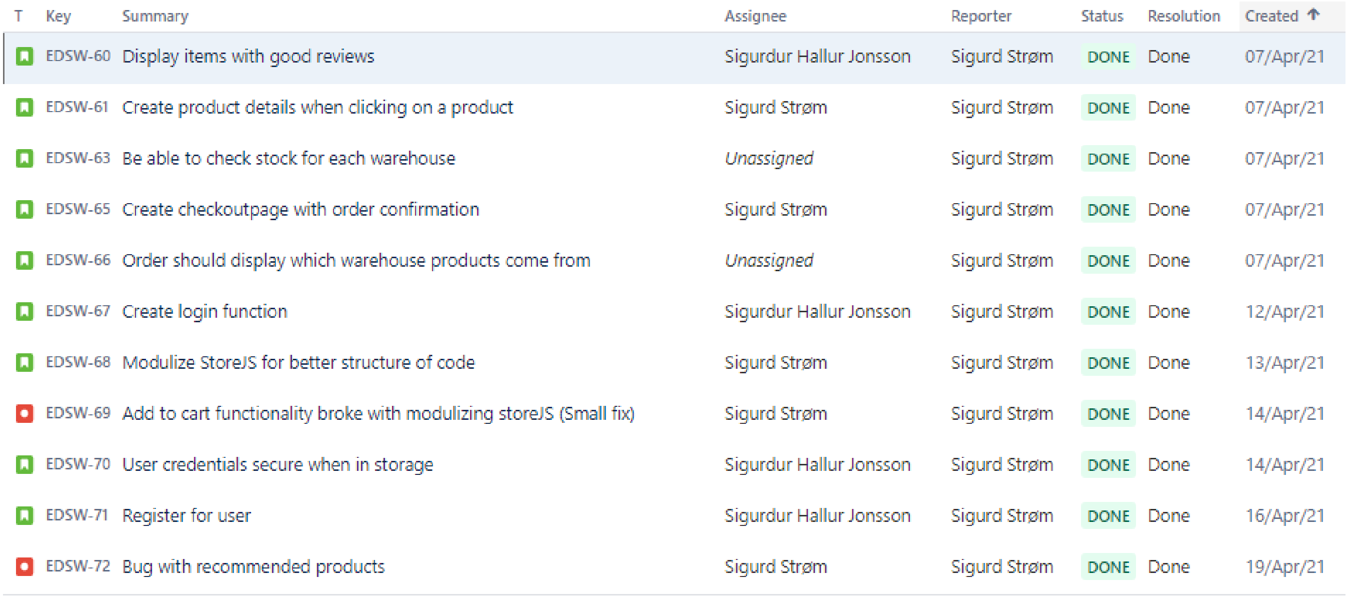
Sprint 5:



T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑
☑	EDSW-21	Continue learning Vue JS	Bachelor Group	Sigurdur Hallur Jonsson	DONE	Done	01/Feb/21
🔖	EDSW-26	Database analyzing	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	04/Feb/21
🔖	EDSW-35	Implement Product generator	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	17/Feb/21
🔖	EDSW-45	Create shopping cart	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	01/Mar/21
🔖	EDSW-54	Make all api call urls and list them	Sigurdur Hallur Jonsson	Sigurdur Hallur Jonsson	DONE	Done	29/Mar/21
🔖	EDSW-56	Create SearchBar	Sigurd Strøm	Sigurd Strøm	DONE	Done	07/Apr/21
🔖	EDSW-57	Created store JS logic for search	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	07/Apr/21
🔖	EDSW-59	Create review system	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	07/Apr/21
🔖	EDSW-60	Display items with good reviews	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	07/Apr/21
🔖	EDSW-61	Create product details when clicking on a product	Sigurd Strøm	Sigurd Strøm	DONE	Done	07/Apr/21
🔖	EDSW-64	Build categories based on registered categories in database	Sigurd Strøm	Sigurd Strøm	DONE	Done	07/Apr/21

Figure 75: Jira log sprint 5

Sprint 6:

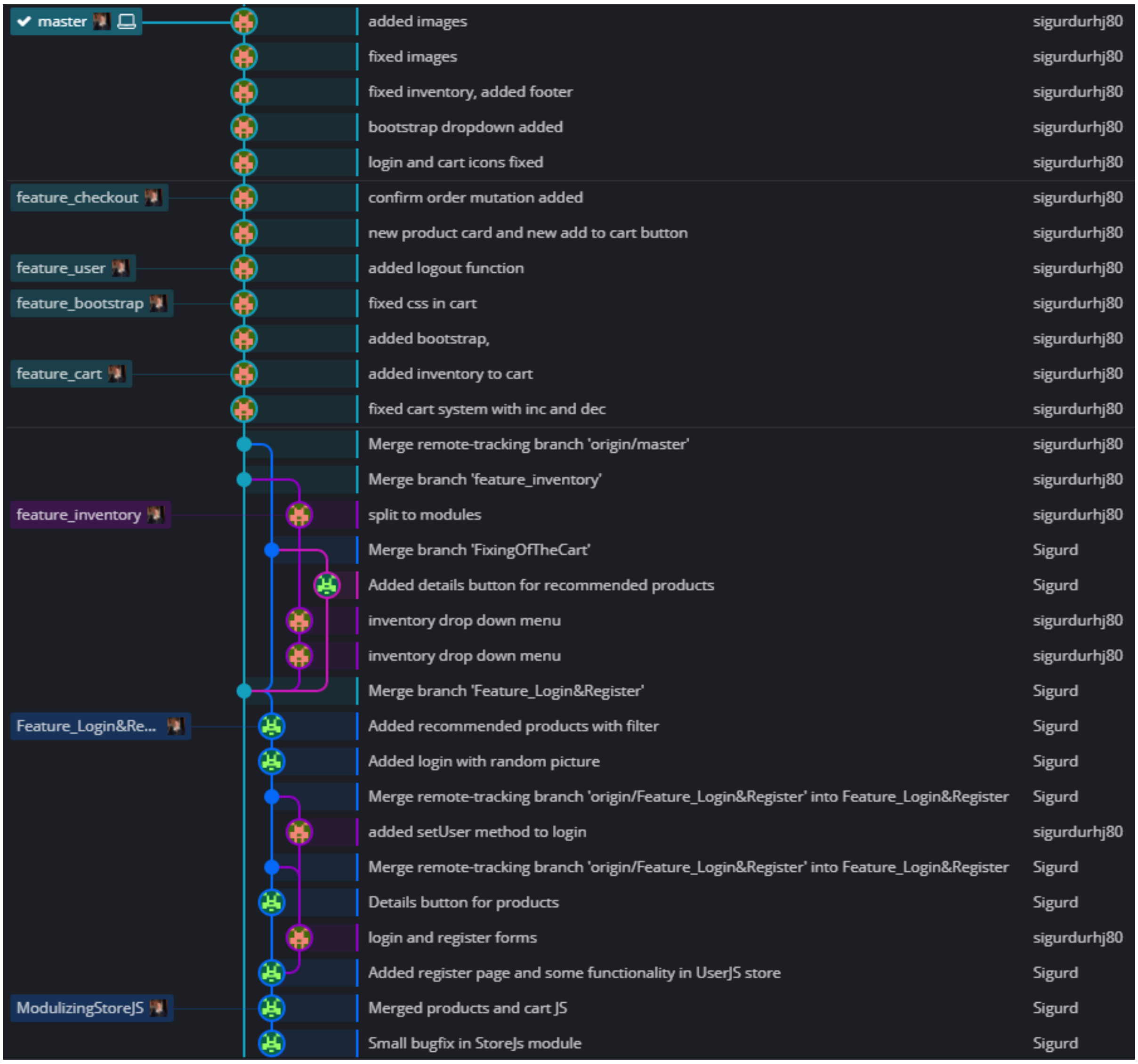


T	Key	Summary	Assignee	Reporter	Status	Resolution	Created ↑
🟩	EDSW-60	Display items with good reviews	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	07/Apr/21
🟩	EDSW-61	Create product details when clicking on a product	Sigurd Strøm	Sigurd Strøm	DONE	Done	07/Apr/21
🟩	EDSW-63	Be able to check stock for each warehouse	*Unassigned*	Sigurd Strøm	DONE	Done	07/Apr/21
🟩	EDSW-65	Create checkoutpage with order confirmation	Sigurd Strøm	Sigurd Strøm	DONE	Done	07/Apr/21
🟩	EDSW-66	Order should display which warehouse products come from	*Unassigned*	Sigurd Strøm	DONE	Done	07/Apr/21
🟩	EDSW-67	Create login function	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	12/Apr/21
🟩	EDSW-68	Modulize StoreJS for better structure of code	Sigurd Strøm	Sigurd Strøm	DONE	Done	13/Apr/21
🟥	EDSW-69	Add to cart functionality broke with modulizing storeJS (Small fix)	Sigurd Strøm	Sigurd Strøm	DONE	Done	14/Apr/21
🟩	EDSW-70	User credentials secure when in storage	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	14/Apr/21
🟩	EDSW-71	Register for user	Sigurdur Hallur Jonsson	Sigurd Strøm	DONE	Done	16/Apr/21
🟥	EDSW-72	Bug with recommended products	Sigurd Strøm	Sigurd Strøm	DONE	Done	19/Apr/21

Figure 76: Jira log sprint 6

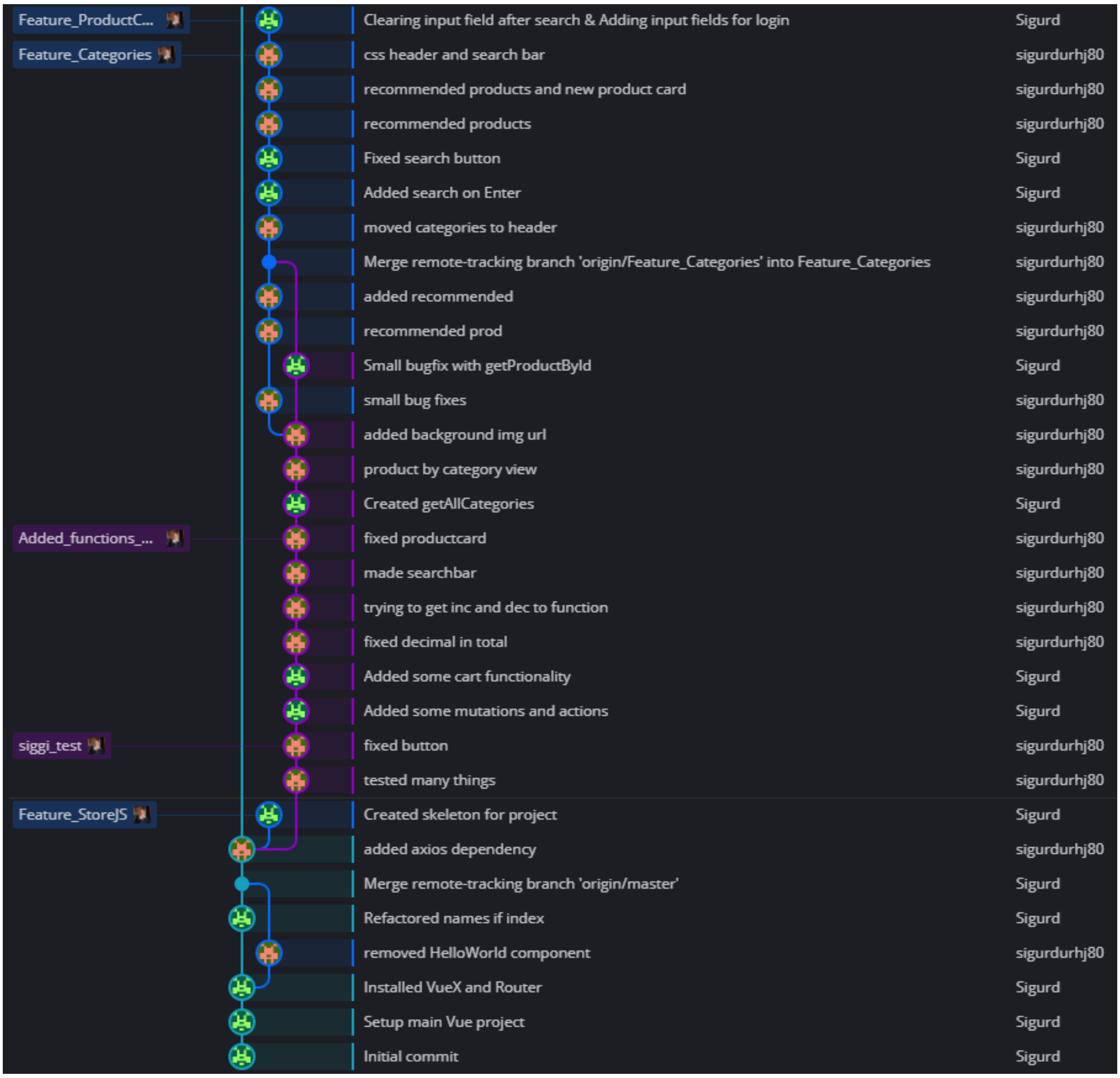
10.2 Git Log

10.2.1 E-commerce Storefront (The final repository for frontend)



✓ master	added images	sigurdurhj80
	fixed images	sigurdurhj80
	fixed inventory, added footer	sigurdurhj80
	bootstrap dropdown added	sigurdurhj80
	login and cart icons fixed	sigurdurhj80
feature_checkout	confirm order mutation added	sigurdurhj80
	new product card and new add to cart button	sigurdurhj80
feature_user	added logout function	sigurdurhj80
feature_bootstrap	fixed css in cart	sigurdurhj80
	added bootstrap,	sigurdurhj80
feature_cart	added inventory to cart	sigurdurhj80
	fixed cart system with inc and dec	sigurdurhj80
	Merge remote-tracking branch 'origin/master'	sigurdurhj80
	Merge branch 'feature_inventory'	sigurdurhj80
feature_inventory	split to modules	sigurdurhj80
	Merge branch 'FixingOfTheCart'	Sigurd
	Added details button for recommended products	Sigurd
	inventory drop down menu	sigurdurhj80
	inventory drop down menu	sigurdurhj80
	Merge branch 'Feature_Login&Register'	Sigurd
Feature_Login&Re...	Added recommended products with filter	Sigurd
	Added login with random picture	Sigurd
	Merge remote-tracking branch 'origin/Feature_Login&Register' into Feature_Login&Register	Sigurd
	added setUser method to login	sigurdurhj80
	Merge remote-tracking branch 'origin/Feature_Login&Register' into Feature_Login&Register	Sigurd
	Details button for products	Sigurd
	login and register forms	sigurdurhj80
	Added register page and some functionality in UserJS store	Sigurd
ModulizingStoreJS	Merged products and cart JS	Sigurd
	Small bugfix in StoreJs module	Sigurd

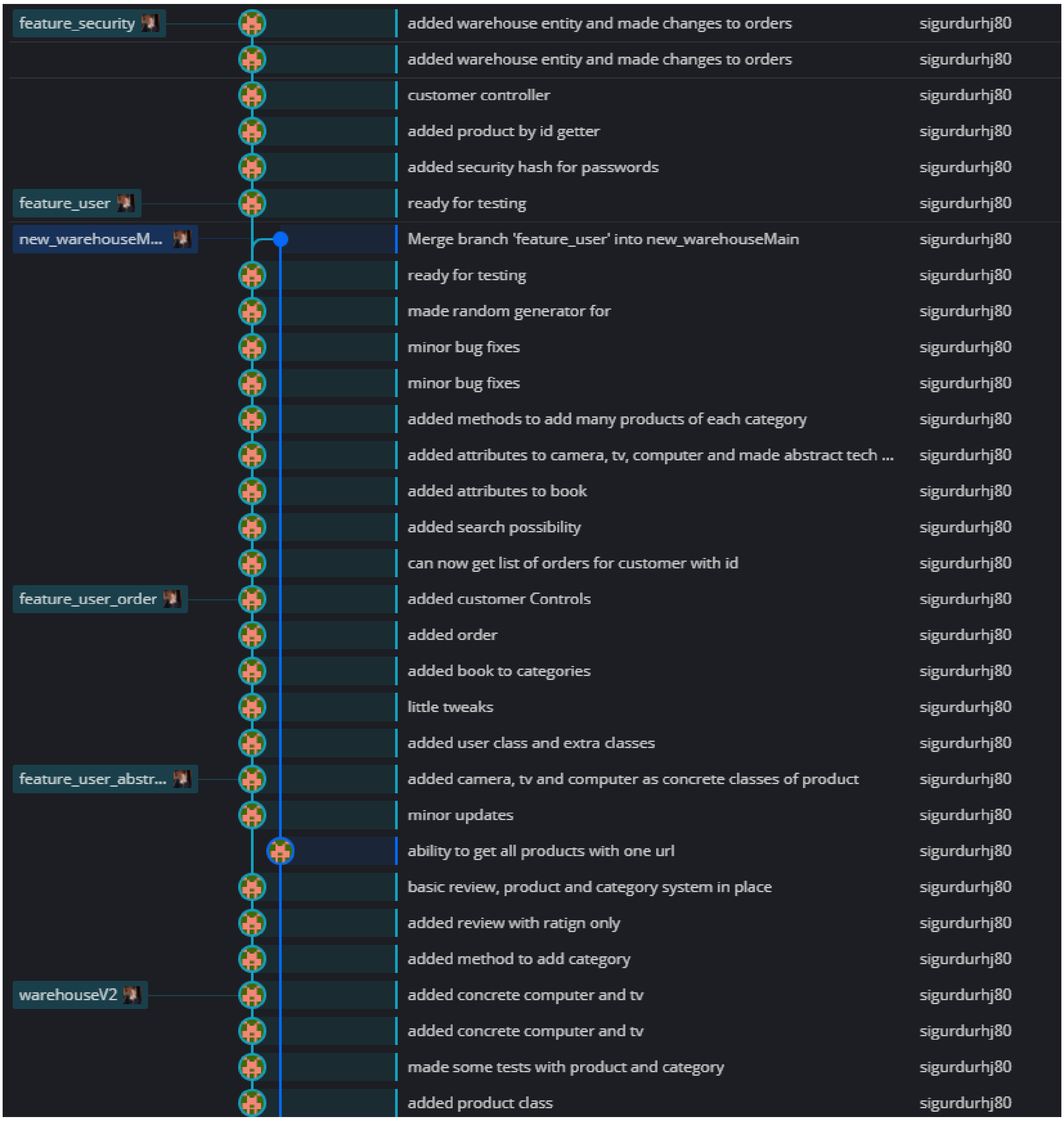
Figure 77: E-commerce Storefront (part 1)



Feature_ProductC...	Clearing input field after search & Adding input fields for login	Sigurd
Feature_Categories	css header and search bar	sigurdurhj80
	recommended products and new product card	sigurdurhj80
	recommended products	sigurdurhj80
	Fixed search button	Sigurd
	Added search on Enter	Sigurd
	moved categories to header	sigurdurhj80
	Merge remote-tracking branch 'origin/Feature_Categories' into Feature_Categories	sigurdurhj80
	added recommended	sigurdurhj80
	recommended prod	sigurdurhj80
	Small bugfix with getProductById	Sigurd
	small bug fixes	sigurdurhj80
	added background img url	sigurdurhj80
	product by category view	sigurdurhj80
	Created getAllCategories	Sigurd
Added_functions_...	fixed productcard	sigurdurhj80
	made searchbar	sigurdurhj80
	trying to get inc and dec to function	sigurdurhj80
	fixed decimal in total	sigurdurhj80
	Added some cart functionality	Sigurd
	Added some mutations and actions	Sigurd
siggi_test	fixed button	sigurdurhj80
	tested many things	sigurdurhj80
Feature_StoreJS	Created skeleton for project	Sigurd
	added axios dependency	sigurdurhj80
	Merge remote-tracking branch 'origin/master'	Sigurd
	Refactored names if index	Sigurd
	removed HelloWorld component	sigurdurhj80
	Installed VueX and Router	Sigurd
	Setup main Vue project	Sigurd
	Initial commit	Sigurd

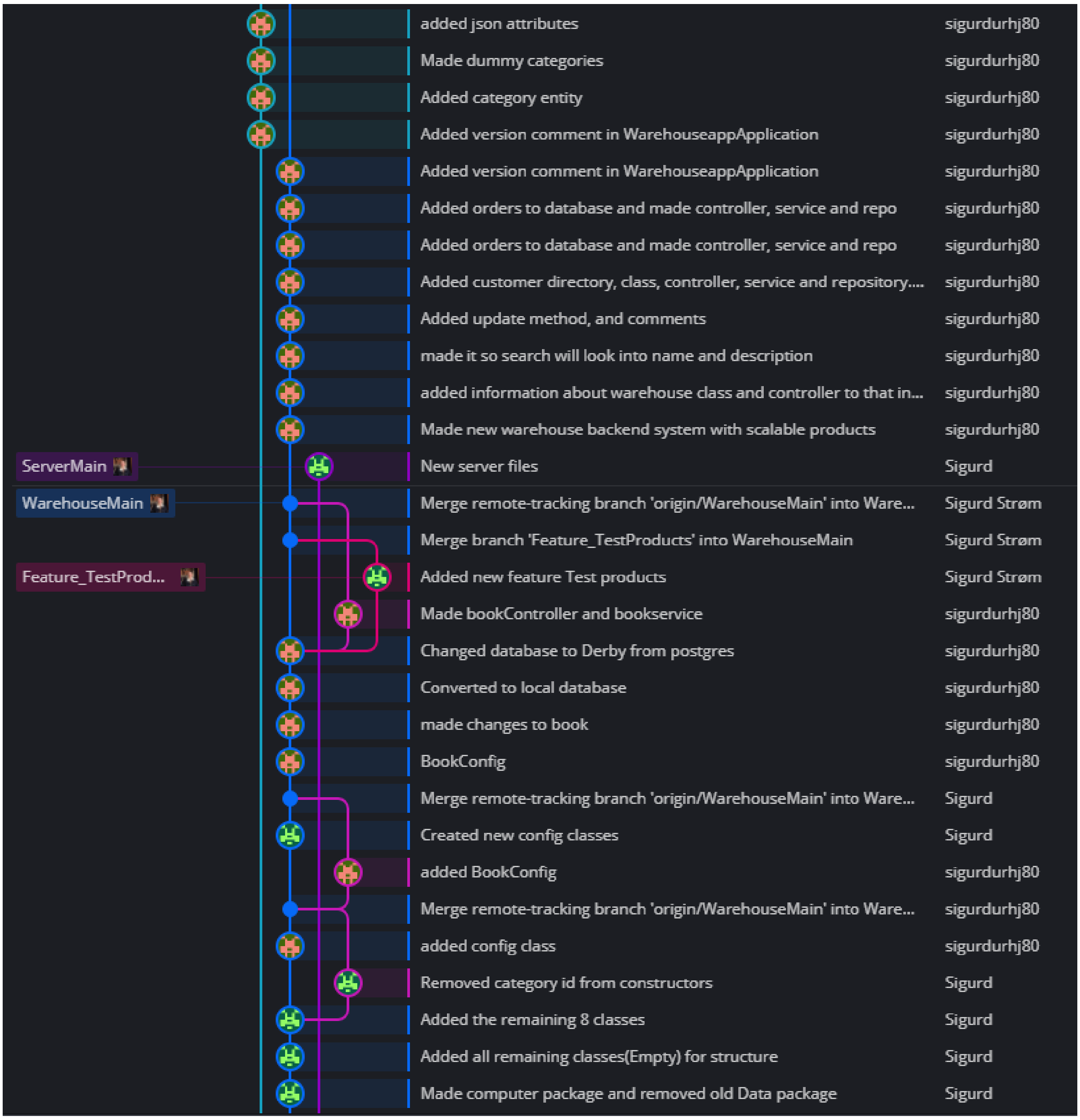
Figure 78: E-commerce Storefront (part 2)

10.2.2 Warehouse App repository for backend system



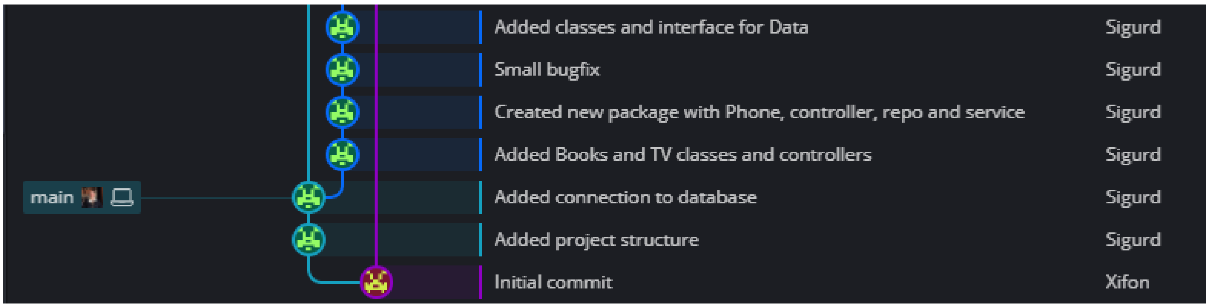
feature_security	added warehouse entity and made changes to orders	sigurdurhj80
	added warehouse entity and made changes to orders	sigurdurhj80
	customer controller	sigurdurhj80
	added product by id getter	sigurdurhj80
	added security hash for passwords	sigurdurhj80
feature_user	ready for testing	sigurdurhj80
new_warehouseM...	Merge branch 'feature_user' into new_warehouseMain	sigurdurhj80
	ready for testing	sigurdurhj80
	made random generator for	sigurdurhj80
	minor bug fixes	sigurdurhj80
	minor bug fixes	sigurdurhj80
	added methods to add many products of each category	sigurdurhj80
	added attributes to camera, tv, computer and made abstract tech ...	sigurdurhj80
	added attributes to book	sigurdurhj80
	added search possibility	sigurdurhj80
	can now get list of orders for customer with id	sigurdurhj80
feature_user_order	added customer Controls	sigurdurhj80
	added order	sigurdurhj80
	added book to categories	sigurdurhj80
	little tweaks	sigurdurhj80
	added user class and extra classes	sigurdurhj80
feature_user_abstr...	added camera, tv and computer as concrete classes of product	sigurdurhj80
	minor updates	sigurdurhj80
	ability to get all products with one url	sigurdurhj80
	basic review, product and category system in place	sigurdurhj80
	added review with ratign only	sigurdurhj80
	added method to add category	sigurdurhj80
warehouseV2	added concrete computer and tv	sigurdurhj80
	added concrete computer and tv	sigurdurhj80
	made some tests with product and category	sigurdurhj80
	added product class	sigurdurhj80

Figure 79: Warehouse App repository (part 1)



	added json attributes	sigurdurhj80
	Made dummy categories	sigurdurhj80
	Added category entity	sigurdurhj80
	Added version comment in WarehouseappApplication	sigurdurhj80
	Added version comment in WarehouseappApplication	sigurdurhj80
	Added orders to database and made controller, service and repo	sigurdurhj80
	Added orders to database and made controller, service and repo	sigurdurhj80
	Added customer directory, class, controller, service and repository....	sigurdurhj80
	Added update method, and comments	sigurdurhj80
	made it so search will look into name and description	sigurdurhj80
	added information about warehouse class and controller to that in...	sigurdurhj80
	Made new warehouse backend system with scalable products	sigurdurhj80
ServerMain	New server files	Sigurd
WarehouseMain	Merge remote-tracking branch 'origin/WarehouseMain' into Ware...	Sigurd Strøm
	Merge branch 'Feature_TestProducts' into WarehouseMain	Sigurd Strøm
Feature_TestProd...	Added new feature Test products	Sigurd Strøm
	Made bookController and bookservice	sigurdurhj80
	Changed database to Derby from postgres	sigurdurhj80
	Converted to local database	sigurdurhj80
	made changes to book	sigurdurhj80
	BookConfig	sigurdurhj80
	Merge remote-tracking branch 'origin/WarehouseMain' into Ware...	Sigurd
	Created new config classes	Sigurd
	added BookConfig	sigurdurhj80
	Merge remote-tracking branch 'origin/WarehouseMain' into Ware...	sigurdurhj80
	added config class	sigurdurhj80
	Removed category id from constructors	Sigurd
	Added the remaining 8 classes	Sigurd
	Added all remaining classes(Empty) for structure	Sigurd
	Made computer package and removed old Data package	Sigurd

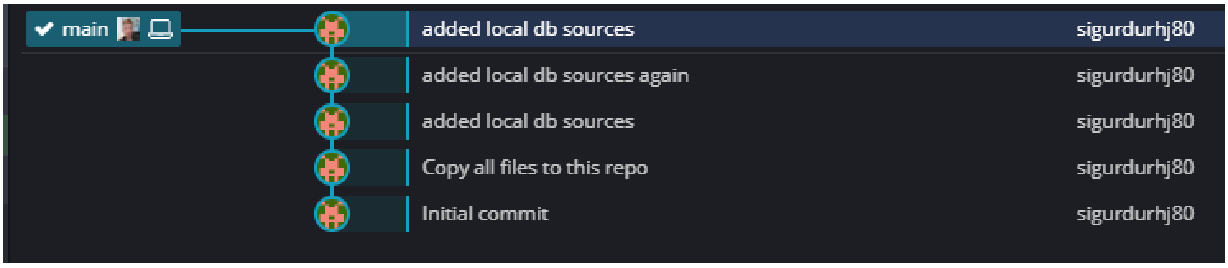
Figure 80: Warehouse App repository (part 2)



	Added classes and interface for Data	Sigurd
	Small bugfix	Sigurd
	Created new package with Phone, controller, repo and service	Sigurd
	Added Books and TV classes and controllers	Sigurd
main	Added connection to database	Sigurd
	Added project structure	Sigurd
	Initial commit	Xifon

Figure 81: Warehouse App repository (part 3)

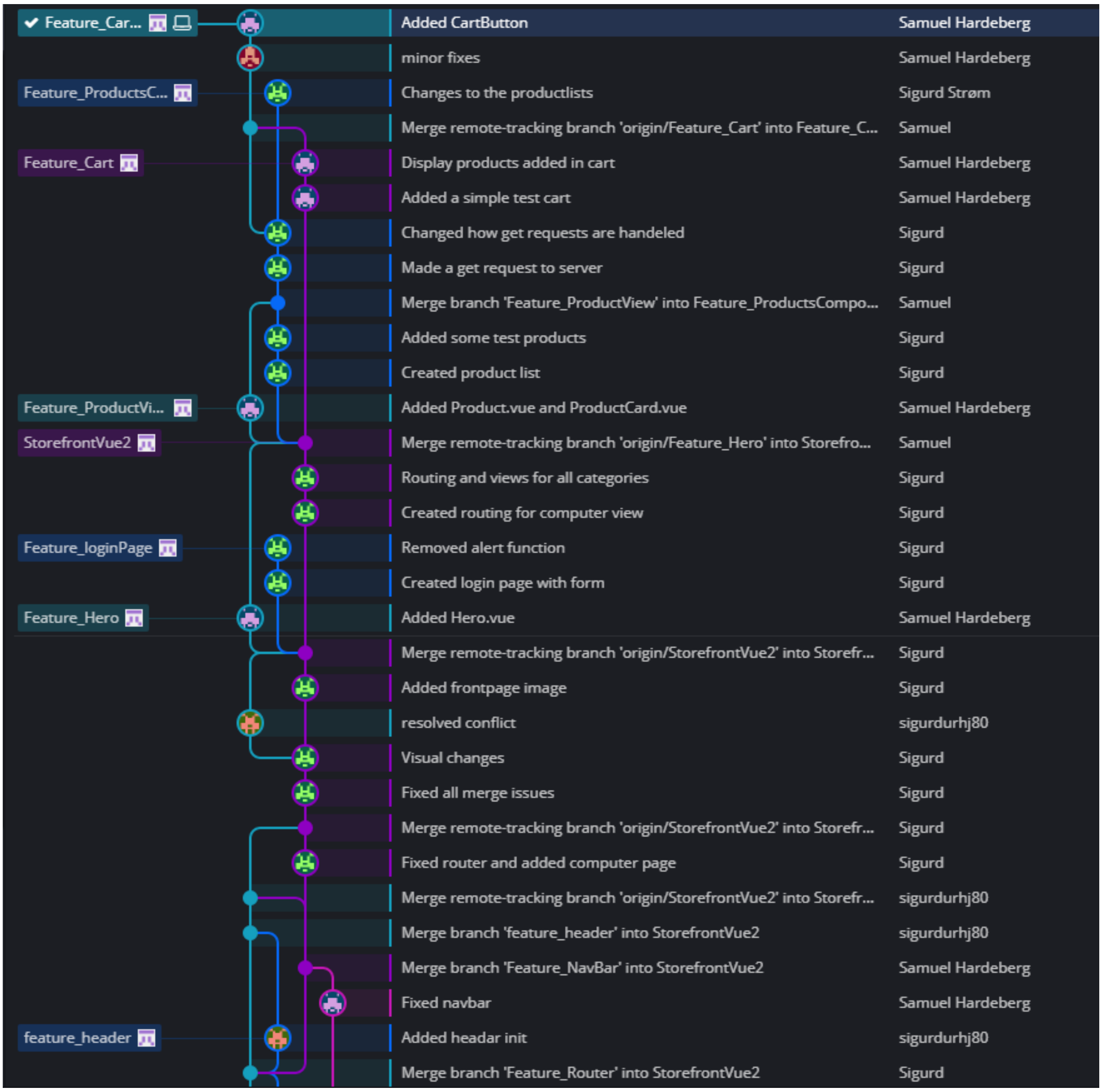
10.2.3 Warehouse Aalesund



added local db sources	sigurdurhj80
added local db sources again	sigurdurhj80
added local db sources	sigurdurhj80
Copy all files to this repo	sigurdurhj80
Initial commit	sigurdurhj80

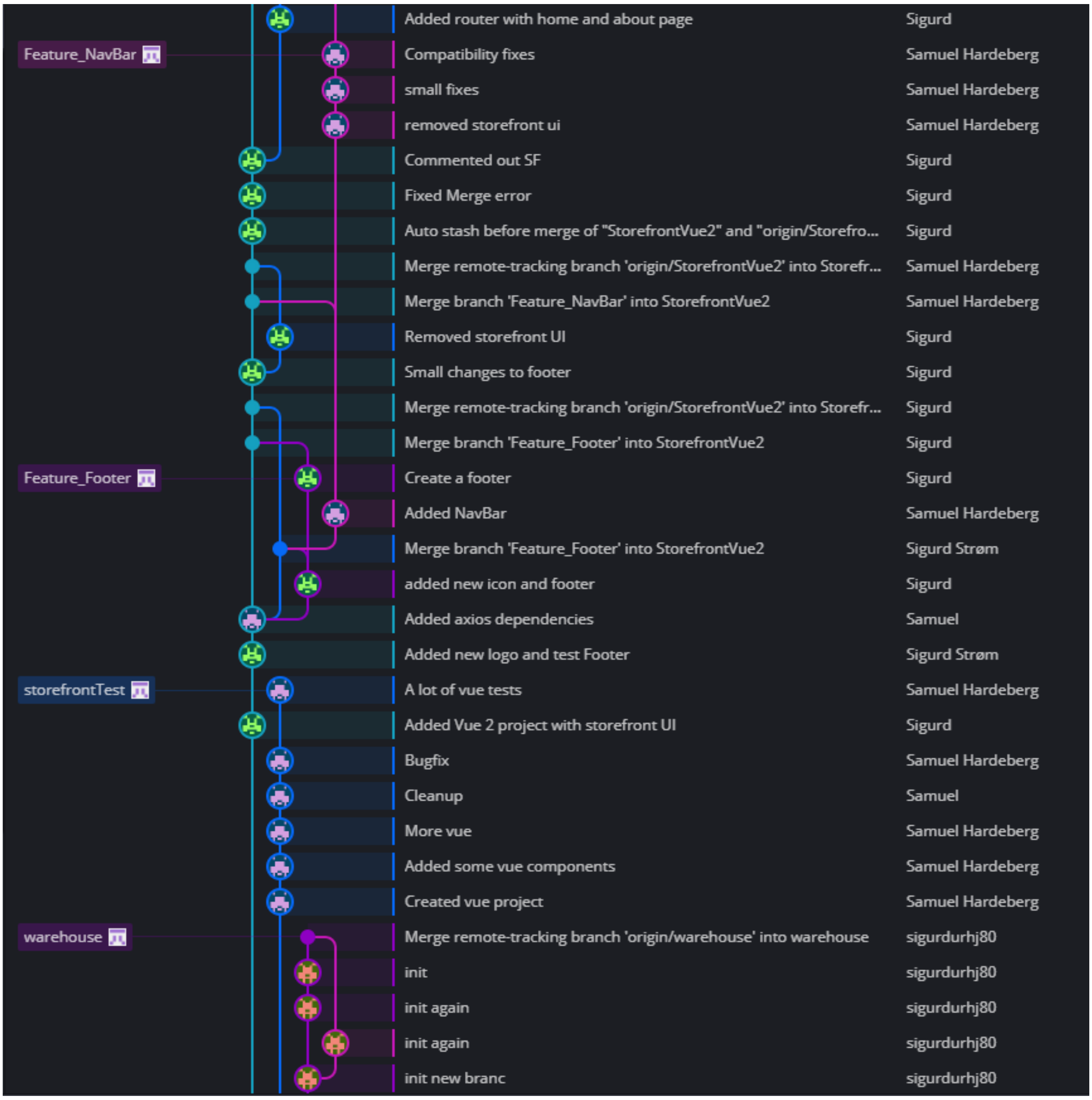
Figure 82: Warehouse Aalesund repository

10.2.4 E-commerce in a distributed system of warehouses (Old testing repository)



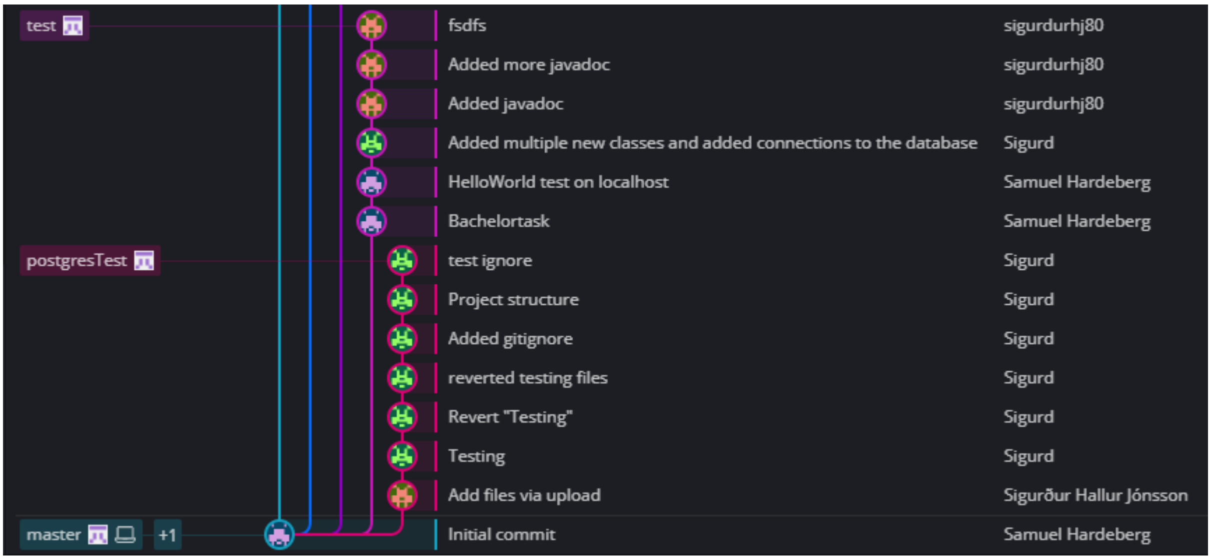
✓ Feature_Car...	Added CartButton	Samuel Hardeberg
	minor fixes	Samuel Hardeberg
Feature_ProductsC...	Changes to the productlists	Sigurd Strøm
	Merge remote-tracking branch 'origin/Feature_Cart' into Feature_C...	Samuel
Feature_Cart	Display products added in cart	Samuel Hardeberg
	Added a simple test cart	Samuel Hardeberg
	Changed how get requests are handeled	Sigurd
	Made a get request to server	Sigurd
	Merge branch 'Feature_ProductView' into Feature_ProductsCompo...	Samuel
	Added some test products	Sigurd
	Created product list	Sigurd
Feature_ProductVi...	Added Product.vue and ProductCard.vue	Samuel Hardeberg
StorefrontVue2	Merge remote-tracking branch 'origin/Feature_Hero' into Storefro...	Samuel
	Routing and views for all categories	Sigurd
	Created routing for computer view	Sigurd
Feature_loginPage	Removed alert function	Sigurd
	Created login page with form	Sigurd
Feature_Hero	Added Hero.vue	Samuel Hardeberg
	Merge remote-tracking branch 'origin/StorefrontVue2' into Storefr...	Sigurd
	Added frontpage image	Sigurd
	resolved conflict	sigurdurhj80
	Visual changes	Sigurd
	Fixed all merge issues	Sigurd
	Merge remote-tracking branch 'origin/StorefrontVue2' into Storefr...	Sigurd
	Fixed router and added computer page	Sigurd
	Merge remote-tracking branch 'origin/StorefrontVue2' into Storefr...	sigurdurhj80
	Merge branch 'feature_header' into StorefrontVue2	sigurdurhj80
	Merge branch 'Feature_NavBar' into StorefrontVue2	Samuel Hardeberg
	Fixed navbar	Samuel Hardeberg
feature_header	Added headar init	sigurdurhj80
	Merge branch 'Feature_Router' into StorefrontVue2	Sigurd

Figure 83: E-commerce in a distributed system of warehouses (part 1)



Added router with home and about page	Sigurd
Compatibility fixes	Samuel Hardeberg
small fixes	Samuel Hardeberg
removed storefront ui	Samuel Hardeberg
Commented out SF	Sigurd
Fixed Merge error	Sigurd
Auto stash before merge of "StorefrontVue2" and "origin/Storefro...	Sigurd
Merge remote-tracking branch 'origin/StorefrontVue2' into Storefr...	Samuel Hardeberg
Merge branch 'Feature_NavBar' into StorefrontVue2	Samuel Hardeberg
Removed storefront UI	Sigurd
Small changes to footer	Sigurd
Merge remote-tracking branch 'origin/StorefrontVue2' into Storefr...	Sigurd
Merge branch 'Feature_Footer' into StorefrontVue2	Sigurd
Create a footer	Sigurd
Added NavBar	Samuel Hardeberg
Merge branch 'Feature_Footer' into StorefrontVue2	Sigurd Strøm
added new icon and footer	Sigurd
Added axios dependencies	Samuel
Added new logo and test Footer	Sigurd Strøm
A lot of vue tests	Samuel Hardeberg
Added Vue 2 project with storefront UI	Sigurd
Bugfix	Samuel Hardeberg
Cleanup	Samuel
More vue	Samuel Hardeberg
Added some vue components	Samuel Hardeberg
Created vue project	Samuel Hardeberg
Merge remote-tracking branch 'origin/warehouse' into warehouse	sigurdurhj80
init	sigurdurhj80
init again	sigurdurhj80
init again	sigurdurhj80
init new branc	sigurdurhj80

Figure 84: E-commerce in a distributed system of warehouses (part 2)



fsdfs	sigurdurhj80
Added more javadoc	sigurdurhj80
Added javadoc	sigurdurhj80
Added multiple new classes and added connections to the database	Sigurd
HelloWorld test on localhost	Samuel Hardeberg
Bachelortask	Samuel Hardeberg
test ignore	Sigurd
Project structure	Sigurd
Added gitignore	Sigurd
reverted testing files	Sigurd
Revert "Testing"	Sigurd
Testing	Sigurd
Add files via upload	Sigurður Hallur Jónsson
Initial commit	Samuel Hardeberg

Figure 85: E-commerce in a distributed system of warehouses (part 3)