



University  
of Stavanger

Faculty of Science and Technology

## BACHELOR'S THESIS

Study program/Specialization:

Bachelor of Computer Science

Spring semester, 2021

Open access

Writers:

Osama Billa  
Yohannes Dawit Kassaye

(Writer's signature)

(Writer's signature)

Supervisor: Krisztian Balog

Thesis Title: LMS and Smart Text-Editor to Enhance Teacher – Student Collaboration

Credits (ECTS): 20

Keywords:

Web Development, TypeScript, JavaScript,  
HTML, CSS, MySQL, Database, Vue.js,  
Node.js

Pages: 128

Stavanger, May 15<sup>th</sup> / 2021

# Contents

<b>Innhold</b>	<b>i</b>
<b>Summary</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Project Organization . . . . .	2
1.4 Outline . . . . .	4
<b>2 Technology and Resources</b>	<b>5</b>
2.1 Programming Languages . . . . .	5
2.1.1 HTML . . . . .	5
2.1.2 CSS . . . . .	5
2.1.3 JavaScript . . . . .	6
2.1.4 Typescript . . . . .	6

## CONTENTS

---

2.2	Front-end Technologies . . . . .	6
2.2.1	Vue Framework . . . . .	6
2.2.2	Bootstrap . . . . .	9
2.2.3	Typeform . . . . .	10
2.2.4	Vue Highcharts . . . . .	10
2.2.5	Axios . . . . .	12
2.2.6	Quill . . . . .	13
2.2.7	Vue Test Utils . . . . .	14
2.2.8	Mocha . . . . .	15
2.2.9	Chai . . . . .	15
2.3	Back-end Technologies . . . . .	16
2.3.1	Node.Js . . . . .	16
2.3.2	Express.Js Framework . . . . .	17
2.3.3	JSON Web Token . . . . .	17
2.3.4	Sequelize . . . . .	19
<b>3</b>	<b>System Overview</b>	<b>21</b>
3.1	Main Concepts . . . . .	21
3.2	Requirements . . . . .	22
3.3	Application Flowchart . . . . .	23
3.4	Directory Structure . . . . .	24

## CONTENTS

---

3.5	Key Components . . . . .	25
<b>4</b>	<b>Back-end Structure</b>	<b>26</b>
4.1	Server Setup . . . . .	28
4.2	Database . . . . .	29
4.3	Routes Structure . . . . .	36
4.3.1	Controllers and Routes . . . . .	37
<b>5</b>	<b>Front-end Structure</b>	<b>39</b>
5.1	User Interface . . . . .	39
5.1.1	Login Page . . . . .	40
5.1.2	Home Page . . . . .	41
5.1.3	Courses Page . . . . .	43
5.1.4	Course Page . . . . .	44
5.1.5	Documents Page . . . . .	46
5.1.6	Question Set Page . . . . .	48
5.1.7	Create Question Set Page . . . . .	49
5.2	Front-end Components and Technologies . . . . .	50
5.2.1	Vue Framework . . . . .	52
5.2.2	Components . . . . .	54
5.2.3	Views . . . . .	56
5.3	Vuex State Management . . . . .	58

## CONTENTS

---

5.3.1	Interfaces . . . . .	59
5.3.2	Modules . . . . .	60
5.4	Router . . . . .	67
5.5	Axios Web Service . . . . .	68
5.5.1	Customize Default Axios Settings . . . . .	69
5.5.2	Axios Requests . . . . .	72
5.6	Quill Text-Editor . . . . .	73
5.6.1	Quill Building-Blocks . . . . .	74
5.6.2	Customizing Quill . . . . .	76
<b>6</b>	<b>Test Driven Development</b>	<b>81</b>
6.1	Implementation . . . . .	81
6.2	Github Actions . . . . .	86
6.2.1	BachelorCI and Build . . . . .	87
<b>7</b>	<b>Student Monitoring and Data Visualization</b>	<b>90</b>
7.1	Monitoring Mechanisms . . . . .	94
7.1.1	Implementation of Monitoring Mechanisms . . . . .	95
7.2	Visualizing Data . . . . .	100
<b>8</b>	<b>Application Evaluation</b>	<b>102</b>
8.1	Google Form Survey . . . . .	104
8.2	Feedback and Assessment . . . . .	105

## CONTENTS

---

<b>9 Conclusion</b>	<b>107</b>
9.1 Main Contributions . . . . .	107
9.2 Moving Forward . . . . .	107
9.3 Notable Lessons Learnt . . . . .	110
9.3.1 Best Practices and File structure . . . . .	110
9.3.2 TypeScript . . . . .	110
9.4 Reflection . . . . .	111
<b>References</b>	<b>113</b>
<b>Appendix</b>	<b>113</b>
<b>A Setup Instructions</b>	<b>114</b>
<b>B Database Diagram</b>	<b>115</b>
<b>C Components and Views</b>	<b>117</b>
<b>D Directory Structure</b>	<b>119</b>
<b>E Datablad</b>	<b>121</b>

# Summary

Our vision with this project was to create an application which collects data of students as they interact with course material. The collected data would then be visualized and presented to the students' teacher. What we hoped to achieve with this was an automated system that provides teachers with useful information and better insight into how the students are faring in a course so that the teachers could better facilitate the education to match with each individual students' needs.

This was achieved by creating a complete learning management system in which teachers can create a course, add to it course material and create question sets that can be linked to the course. Students could then join a course and interact with the course material. We monitor the students as they are reading through a course document and collect data on which topics they spend the most time on and how they perform on question sets related to a course. The collected data is then processed and visualized in a graph for the teacher to see.

# Preface

We would like to thank Krisztian Balog, full professor and head of the Information Access and Artificial Intelligence research group at the University of Stavanger for accepting this project and providing pivotal guidance and much needed structure in the development process. Meetings with him were as informational as they were humorous and pleasant, and for that we would like to again give our thanks to him.

# Chapter 1

## Introduction

### 1.1 Background

As current students, we've had years of first-hand experience with how the educational system works. Throughout these years, it has become apparent that there is room for improvements in certain areas. The area we've decided to tackle on is the one comprising of teachers' assessment of the individual students' progression and understanding of the curriculum.

A lecturer might pause during a lecture after covering a part of the curriculum and ask the class if anything is unclear and if there are any questions. This way of gathering valuable data on the students understanding of what was recently covered is inconsistent and not reliable as it is solely based on the students' ability to speak up.

Another tool teachers have to assess their students are compulsory assignments and graded tests. In many cases a students' weaknesses will be spotted after a test and the class moves on to the next topic in the curriculum without providing the students a chance to properly learn from their mistakes.

Gathering information on how students fare in a course and teaching the course are two processes that go hand-in-hand but in today's system are separated and done individually from one another. We wish to combine those two into one platform.

We want to make it possible for teachers to detect students' weaknesses and provide them with the necessary help and guidance before an important test, and not after.



## 1.2 Problem Statement

---

## 1.2 Problem Statement

Our goal is to build an application that will give teachers a better and more detailed overview of how the students are doing in a course

With this application, we aim to solve two problems.

The first problem we would like to solve is changing the way teachers collect data from users. To achieve this, we need to build an application that provides tools that enhances this process. The way we decided to do this is to automatically monitor the progress of each individual student on specific sections of the curriculum and present the data to the teacher.

By solving the first problem, we also eliminate our second problem, which is to give each student a voice on how they are progressing in the course. Many students are not comfortable with speaking up in a room full of people. The consequence of this is that the teacher lacks information regarding to where the students are experiencing difficulties. Automating the process of collecting student data ensures that everyone is heard and the teacher will have a better understanding of where the students struggle.

## 1.3 Project Organization

We used Github for version control. Link to our Github repository:  
<https://github.com/YohannesDK/BachelorThesis>

We had a high-level idea for what we were going for in terms of functionality and design.

For the back-end we created a block-diagram consisting of all the tables we would need and the relation between those. The diagram is presented in appendix B.

For the front-end we created a rough sketch of the different pages and the functionality on each of these pages.

We then created a table where we put a priority on different functionalities of the application and started with implementing the functions with the highest priority and worked towards the ones with lower priority.

### 1.3 Project Organization

---

Priority	Teacher	Student	General
P0 (v0.1)	Create Course, Assign syllabus, questions and other resources to course,	Join and interact with course material	Role-based login system  Dashboard (minimal)
P0 (v0.2)	Visualize student data	Collect data from students taking a course  Visualize personal student data	Dashboard (extended)
P1	Add questions to specific topics	Student notes,  Inactivity detection	Nice UI  responsive design  Security  Search for resources
P2		Share Notes	Notification System  Bug Fixing

**Figure 1.1:** Priority Table

Yohannes functioned as a front-end developer and Osama worked primarily on the back-end. But we were not strictly restricted to one end, and we were involved in both parts of the project. This made it easy to divide the workload between us and ensured that the development process was smooth.

There were two layers to our workflow, one external and one internal.

The external workflow consisted of weekly structured meetings between us and our supervisor. We wrote down an agenda prior to the meeting of things we wanted to discuss. During the meeting we presented the work we had done since previous meeting and set up new goals to work towards the next week.

The internal workflow consisted of daily meetings between us, the writers, where we discussed different possible solutions to our weekly goals and gave each other updates on our progress.

## 1.4 Outline

---

## 1.4 Outline

### **Chapter 2 - Technology and Resources**

In this chapter we give an overview of all the technologies, tools and frameworks used in this project.

### **Chapter 3 - System Overview**

In this chapter we give a complete overview of the project and the file structure.

### **Chapter 4 - Back-end Structure**

In this chapter we explain the setup and structure of the back-end.

### **Chapter 5 - Front-end Structure**

In this chapter we explain the setup and structure of the front-end.

### **Chapter 6 - Application Overview**

In this chapter, we go through the different views of the application and explain the main features in each of them.

### **Chapter 7 - Application Evaluation**

In this chapter, we evaluate our application in various ways and present the results.

### **Chapter 8 - Conclusion**

In this chapter, we conclude our project and discuss what we've learned, improvements to be made, and how we're moving forward.

## Chapter 2

# Technology and Resources

In this chapter we will provide a brief introduction to the different technologies and resources we've taken use of during this project and discuss the reasoning behind the choice of said tools.

## 2.1 Programming Languages

### 2.1.1 HTML

*Hypertext Markup Language*

HTML 5 is the latest version of HTML, and is the go-to standard for structuring and presenting web applications to the user. HTML felt like a natural choice as we were introduced to it in our Web Programming course.

### 2.1.2 CSS

*Cascading Style Sheets*

Is a style sheet language that is used to change the style and design of HTML web-pages. It is one of the core building blocks in web development. It was used in this

## 2.2 Front-end Technologies

---

project as it is simple and effective.

### 2.1.3 JavaScript

JavaScript is a high-level scripting language that allows us to add functionality to our web application. Many of the frameworks used in this project are built on JavaScript, and since it is supported by all web-browsers it became an optimal choice for us. The version of JavaScript used in this project was ECMAScript 2020.

### 2.1.4 Typescript

TypeScript is a superset of JavaScript that offers optional types. It is a strictly typed language that is designed for large-scale JavaScript applications. It compiles to JavaScript and is developed and maintained by Microsoft. It is licensed under the Apache-2.0 license.

We chose to use TypeScript v.4.2.4 in the front-end of our application mainly because JavaScript is very forgiving in the sense that it is not a strictly typed language. In other words, JavaScript doesn't warn or care about the assignment of wrong data types. As the application grows in complexity and the number of components that need to share data becomes large, the application will then be prone to bugs when passing data from one component to another. This is exactly what TypeScript tries to solve, by warning the developer long before the application is deployed to production.

## 2.2 Front-end Technologies

### 2.2.1 Vue Framework

*Vue 3* Vue v.3 is a progressive component based front-end framework based on JavaScript that is used for building user interfaces and single-page applications. Vue makes it possible to extend basic HTML with attributes called directives. The directives provide functionality to HTML elements. There are a variety of different directives, the ones worth of mention that we made use of were:

**v-bind:** Binds the value of an HTML element to a variable in our script

**v-for:** Iterates through an array and renders an HTML element for each element in

## 2.2 Front-end Technologies

---

the array

**v-if:** Renders an element if a specified condition is met

**v-else:** Renders an element if a specified condition is not met

Examples of the v-if and v-else directives in use in use:

```
<h1 v-if="Math.random() > 0.5">The number is greater than 0.5!</h1>
<h1 v-else>The number is not greater than 0.5!</h1>
```

Vue is licensed under the MIT License.

A good alternative to Vue is React, which builds on the same principles and is more widely adopted within the web-development industry. React is maintained by Facebook and used by over 6.2 million people, whereas Vue is not backed by a big company and is used by 142 thousand people. To understand why we settled for Vue instead of React, we would need to look at how the two frameworks differ.

The fact that React has a vast amount of users makes it so that it has a very active community. When searching for solutions to problems online, it is much easier to find what you're looking for if it is a React related issue compared to an issue in Vue. However, the official documentation of Vue is superb, and it is known to be a lot better than the documentation of React. This makes up for the lack of a large and active community.

The learning curve of the Vue framework was a big selling point for us. Vue is the framework we were introduced to in our Web Programming course. The choice of Vue over React by our lecturer was a strategical one as Vue has a much less steep learning curve than React, yet it achieves the same results. We found Vue to be intuitive and comfortable to work with.

We had decided to use TypeScript in our application as it is better suited for large projects. Therefore, support for TypeScript would be essential for us. Although React is written in JavaScript, it offers good support for TypeScript. Vue version 3 was recently released and the whole codebase of Vue was rewritten in TypeScript to offer its full support, thus making it level with React and also a choice that fit our needs.

The biggest difference between Vue and React is how the HTML is written and rendered.

React strictly uses JSX to write HTML code. JSX stands for JavaScript XML and allows users to write HTML code in JavaScript. Below is an example of JSX to output

## 2.2 Front-end Technologies

---

an h1 element with the text Hello World.

```
const message = 'Hello World!';
const element = <h1>Hello, {message}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Vue primarily uses basic HTML syntax and also provides support for JSX syntax. The code in Vue to achieve the same output as the code above written in JSX would be as follows:

The HTML part:

```
<div id="app">
  {{ message }}
</div>
```

The JavaScript part:

```
new Vue({
  el: '#app',
  data: {
    message: 'Hello World!'
  }
})
```

Everything React has to offer so does Vue. Vue has more in-built features like routing and state management, but React being as big as it is has libraries to cover everything it lacks.

We were already familiar with Vue and the syntax it uses. The framework covers our needs and achieves the same results as React but with an easier learning curve. We came to the conclusion that React had no real advantage over Vue, and thus we settled for Vue.

## 2.2 Front-end Technologies

---

### 2.2.2 Bootstrap

Bootstrap v.5 is the most popular UI framework. It makes for easy styling of HTML elements by the use of bootstrap classes that are essentially pre-made CSS classes that we can import and use. It also has a collection of more than 85 ready to use Vue components. These components can be as big as complete navigation bars or as small as a button. Bootstrap is licensed under the MIT License.

The services Bootstrap provide us are also provided by other frameworks, most notable ones are Vuetify and Quasar.

Quasar and Vuetify are both based on Vue. They do not differ much from each other, but where they differ from Bootstrap is in the design guidelines they follow. Quasar and Vuetify follow Googles Material Design guidelines which was created back in 2014. It was created with the intention of giving users a familiar user experience across different applications and platforms, in the same way that Apple has managed to have a very distinct and familiar user experience between all its products.

The benefit of this approach is benefiting from a design created by professionals that provide a good UI and UX experience for users. The drawback is that it follows a specific corporate design that might not appeal to everyone. It also restricts customization as the frameworks have guidelines to follow, whereas Bootstrap does not.

Bootstrap is the most popular UI framework for Vue. This fact signalizes that although Bootstrap does not follow what is considered the golden-standard in terms of design guidelines, it is still a framework that provides components with a design that users are very fond of. The graph below shows the amount of times the different frameworks have been downloaded through npm.

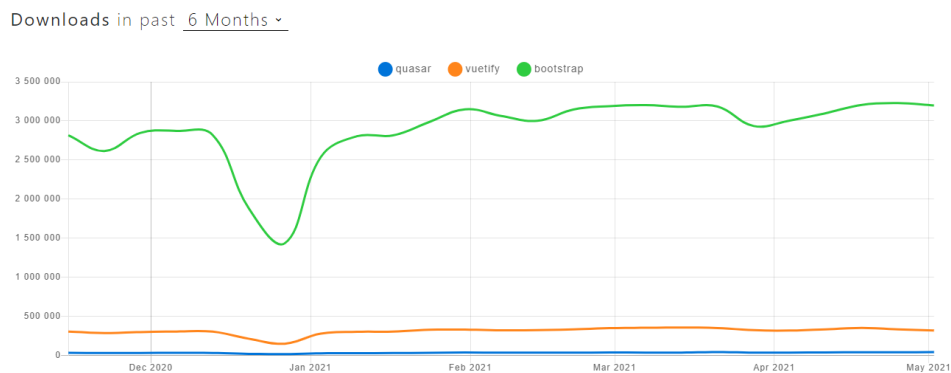


Figure 2.1: UI Framework Popularity Comparison



## 2.2 Front-end Technologies

---

The components and style of Bootstrap fell at a better taste to us in comparison to what Vuetify and Quasar had to offer. It also offers full support for Vue 3, whereas Vuetify, which is the top competitor for Bootstrap, does not. And thus we decided to go for Bootstrap.

### 2.2.3 Typeform

Typeform is an online service that provide complete and professional ready-to-use forms. It is easy to use as it comes with a URL for each form it offers which can simply be embedded in a project.

### 2.2.4 Vue Highcharts

Vue Highcharts v.1.3.5 is the library we used to visualize data in our application. It offers template code in their website that we can copy into our project and customize to our needs. Here is an example of a simple implementation of a line chart from Vue Highcharts:

## 2.2 Front-end Technologies

---

```
<template>
  <vue-highcharts :options="chartOptions"/>
</template>

<script lang="ts">
import VueHighcharts from "vue3-highcharts";

export default {
  components: {
    VueHighcharts
  },
  data() {
    return {
      options: {
        chart: {type: "line"},
        xAxis: {"Monday", "Tuesday", "Wednesday"},
        yAxis: {"Temperature"},
        Data: { [9, 11, 10] }
      }
    }
  }
}
</script>
```

The code above would render a line-graph with the days Monday, Tuesday and Wednesday on the x-axis and temperature on the y-axis. The value for Monday would be 9, Tuesday would be 11 and Wednesday would be 10.

An alternative to Highcharts is a library called Vue-ChartJs. It offers the same service as Highcharts and its implementation follows the same steps although slightly different. Implementing the same graph as the one shown in the code block above in Vue-ChartJs would look like this:

## 2.2 Front-end Technologies

---

```
<template>
  <line-chart :chartdata="chartdata" :options="chartOptions"/>
</template>

<script lang="ts">
import { Line } from 'vue-chartjs'

export default {
  extends: Line,
  data() {
    chartdata: {
      labels: ["Monday", "Tuesday", "Wednesday"],
      datasets: [
        {
          data: [9, 11, 10]
        }
      ]
    },
    options: {
      responsive: true,
      maintainAspectRatio: false
    }
  }
}
</script>
```

The structure of the two implementations are very similar and we decided to settle for Vue Highcharts. The reason for this decision was because Vue-ChartJs only offered basic charts that did not meet our needs. Vue Highcharts allows for more customization of the graphs and can be used for more sophisticated data visualization.

An example to highlight is that we first tried to visualize data with the implementation of Vue-ChartJs due to its simplicity. It got the job done in the beginning but when we were in need of a scatter plot it wasn't able to provide that whereas Vue Highcharts managed to do so in an elegant way. Vue Highcharts also scores better in terms of official documentation.

### 2.2.5 Axios

Vue does not come with any built-in tools that handle API calls. In their official documentation they recommend their users to install the Axios library to take care of API calls and provide documentation on how to easily implement it in Vue applications.

## 2.2 Front-end Technologies

---

Axios is a JavaScript library that offers promise-based HTTP requests. We used it in our project to send and fetch data from our database. An Axios request is structured in the following way:

```
axios.get('/user', {
  params: {
    ID: 1
  }
})
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
})
```

We perform a GET request to the /user API. We pass a params object with an ID field to the request object that is being sent to the server. The server can then access this ID and perform a SELECT query based on the ID. When the query is done the server sends a response object back to our axios function. We can then decide what to do with the response. If the server sends an error object, we can catch it and handle it as we see fit.

Axios is licensed under the MIT license.

### 2.2.6 Quill

Quill is an open source rich-text editor that can be implemented in modern web applications. It stores the input as a Delta format that we store in our database. The delta format is a subset of JSON that stores information about a Quill document that can easily be translated to HTML code with functions provided by the Quill library.

Here is an example of a delta format:

## 2.2 Front-end Technologies

---

```
{
  ops: [
    { insert: 'This text is bolded', attributes: { bold: true }},
    { insert: 'This is normal text' },
    { insert: 'This text is grey', attributes: { color: '#cccccc' }}
  ]
}
```

This delta can be converted to HTML code by using the following function:

```
Editor.setContents(delta);
```

An alternative to Quill is an editor called TinyMCE. It provides more features and is more advanced than Quill, but Quill provides a very key feature which was essential for our needs. It allows us to customize the attributes given to HTML elements which helps greatly in the work of monitoring students progress. With the help of Quill, we can pass unique id's to each header as an attribute as they are being made. The id's can then be used to distinguish one topic from another, and that allows us to collect data from a specific topic in a document.

### 2.2.7 Vue Test Utils

Vue Test Utils v.2 is the recommended and official testing library for Vue. The library makes it possible to mount Vue components into an object like this:

```
import { shallowMount } from '@vue/test-utils'
import dummyComponent from './dummyComponent.vue'

const wrapper = shallowMount(dummyComponent)
```

We can then write tests for a component using the wrapper object.

When installing Vue Test Utils it gives you the option of installing recommended libraries for writings tests. The two options provided were Jest and Mocha. We chose Mocha.

## 2.2 Front-end Technologies

---

### 2.2.8 Mocha

Mocha is an open source JavaScript test framework that is used for writing and running tests. It provides useful test reports and the tests run asynchronously. An alternative to the Mocha framework is Jest.

Jest is also a JavaScript test framework and is built as an open source project by developers from Facebook. The project is also maintained by Facebook and was originally created as a testing framework for React applications but it has been expanded to offer support for other frameworks such as Angular and Vue.

Jest is primarily used for front-end testing whilst Mocha is for both front-end and back-end testing. Mocha also provides more flexibility as it is compatible with a lot of other libraries that provide useful functionality.

Given that we weren't entirely sure on which scale and parts of the application we would implement testing, we were in need of a flexible testing framework. Therefore we went for Mocha.

### 2.2.9 Chai

The Mocha framework does not have built-in assertion methods, so in order to write tests with assertions we needed to make use of the Chai library. The assertion method we made the most use of was the `expect()` method. A demonstration of this method can be seen below:

```
expect(dummyComponent.exists()).to.equal(true);
```

The assertion below checks that the component exists and is rendered in the DOM.

## 2.3 Back-end Technologies

### 2.3.1 Node.Js

Node.Js is an open-source server runtime environment for executing JavaScript code. It runs on the V8 JavaScript Engine, which was created by Google specifically with speed in mind so that their browsers would be faster. They released the V8 Engine Project as open-source so that other developers could build projects on top of this engine.

In addition to running on a fast engine, Node.Js has features of its own that make it very efficient. It is able to run several concurrent requests at once and is only active when it is needed. If there are no jobs to be executed, Node.Js goes into standby mode, which saves RAM. These factors combined make Node.Js one of the fastest back-end technologies available.

The best alternative to Node.Js for server-side programming is ASP.Net which was released by Microsoft in 2002. But since ASP.Net uses CSharp which neither of us were comfortable with we decided against it as there is no significant advantages to be using ASP.Net over Node.Js.

The creator of Node.Js, Ryan Dahl, recently released a new server-side framework called Deno. The motivation behind this was what he claimed was many regrettable choices during his role as a tech-lead under the development of Node.Js which he now has addressed in his new framework. The most notable regrets of his was the security of the framework and how the framework handles dependencies.

The biggest security flaw of Node.Js is that it gives full network and file access to scripts, such as lint. With Deno, no scripts you run in your environment has access to your network or disk without you granting it with special commands.

The way dependencies are handled in Node.Js is unnecessarily complicated and messy. To use a dependency, a user must first install it. The reference to the dependency is then stored in an auto-generated file called package.JSON. The actual code of the dependency is installed into a folder called node-modules. In addition to this, there is an auto-generated file called package-lock.JSON. This file keeps track of all the changes made in the node-modules folder. These auto-generated files become very large and messy. The package-lock.JSON file in our project contains 41907 lines of JSON objects. The framework Deno skips all this unnecessary bookkeeping of dependencies and lets users import dependencies with one line of code that consists of a function and the source of the dependency, as following:

## 2.3 Back-end Technologies

---

```
import { dependency } from "https://dependencyURL";
```

This makes the project tidier and more lightweight.

Deno is written in Rust and is made for JavaScript and TypeScript. It does everything Node.js can do and it does it better in many ways. The reason why we chose Node.js over Deno is a very trivial one; Deno is still under development and very new. Node.js has been around for a long time, and during this time, the community has created many great libraries for the framework that make things easier, such as the Express.js framework.

Node.js is licensed under the MIT License.

### 2.3.2 Express.js Framework

Express.js is a Node.js framework with built-in functions that make setting up a server and handling requests a lot easier than it would be doing it with just Node.js. Its functionalities are designed in a way that makes it easy to set up an MVC structure in the back-end.

### 2.3.3 JSON Web Token

JSON Web Token is an alternative way of authorizing users to the traditional way of using sessions. Traditionally Session id's are stored on the server and passed onto the client's cookie. When the client makes a request to the server, the client's session id is passed through the header field in the HTTP request to the server. The server then compares the session id received from the client with the one stored in the server. If they match, the request is resolved, if they don't the request is rejected.

JWT operates in a different way. A token is only stored in the client side of the application, and not on the server. Scalability wise it is better to use JWT for authorizing requests since it is not stored on the server, hence does not make use of the server's memory.

A JWT token is sent through the HTTP request to the server. The server then decodes and validates the token. If the token is validated, the request is resolved, if not, the request is rejected. JWT is proposed as an open standard by IETF and is by many chosen as the go-to technology for authorization.



## 2.3 Back-end Technologies

---

JWT makes authorization easy and intuitive, and although it is proposed as an open standard by IETF, it does have well known security flaws.

This is what a typical JWT token looks like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
eyJzdWIiOiIyMyIsIm5hbWUiOiJUb20gd29vZCI6ImFkbWluIjpmYXxzZX0
.qf6z0DKxLwzzpBu4bvvo_or3xM0vveku0ta-VGBHG48
```

A JWT consists of three parts: A header, a payload, and a signature used to verify the token. The three different parts are separated by a period, as can be seen in the token above.

The first two parts, the header and payload, are Base64Url encoded, which can easily be decoded by either simple terminal commands or using websites that will decode it for you.

Decoding the first part of the token will give us the header, and it looks like this:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The field of interest here is the "alg" field, which specifies what algorithm will be used when encrypting the token.

Decoding the second part of the token will yield us the payload, which look like this:

```
{
  "userID": "23",
  "name": "Tom Wood",
  "admin": false
}
```

## 2.3 Back-end Technologies

---

The payload consists of information stored of the user. The field of interest in this object is the "admin" field. In this case it is set to false, which means that this user is not an admin. If one were to change the "admin" field to true, encode it in base64url and send it back as a forged token, it would not be authorized by the server. The reason for this is the last part of the token. We cannot decode the last part of the token as it is encrypted using the algorithm specified in the header. The last part is what is used by the server to verify a token's authenticity.

Since we have access to the header, and the header specifies which algorithm will be used to encrypt the last part of the token, attackers can modify the "alg" field and set it to none, therefore telling the server that this token does not need a verification part, and thus dropping the last part of the token altogether. The resulting token will look like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIyMyIsIm5hbWUiOiJUbnR5Z29vZCI6ImFkbWluIjp0cnV1fQ
```

We can use this forged token to make requests to the server as an admin user. This vulnerability has been addressed by many libraries that rely on JWT. The way it was addressed was by simply not verifying tokens that had the "alg" field set to none. People who work with cyber-security have suggested removing the "alg" field altogether and implementing a more secure way of assigning an encryption algorithm.

Despite having security flaws, JWT remains the accepted standard for authorization tokens. There are no viable replacements for JWT as of yet, but there are authentication frameworks that handle tokens in a more secure manner. One example of such framework would be OAuth2 which is developed by Google.

OAuth2 becomes useful if an application is in need of fetching authorization tokens from other applications. For instance, Overleaf allows users to log in by using their Google account. In our case, this is not needed, and therefore we implemented JWT as it is.

JWT is licensed under the Apache License 2.0.

### 2.3.4 Sequelize

Sequelize is a JavaScript library that is defined as an ORM (Object Relational Mapper). The installation of Sequelize comes with a CLI that allows us to create models in our

## 2.3 Back-end Technologies

---

database through terminal commands. It has a wide variety of functions that enables us to execute queries using JavaScript code. We used Sequelize in our project in combination with Axios to effectively communicate with our SQL database.

A good alternative to Sequelize is TypeORM. TypeORM offers the same services as Sequelize but in its own style. The structure of code that executes queries in TypeORM differs from that of Sequelize.

Sequelize passes objects into functions that execute queries based on the fields of the object. To create a user in Sequelize one would write the following code:

```
User.create({
  username: 'Tom Wood',
  birthday: new Date(1999, 4, 27)
}))
```

To create a user in TypeORM one would first initialize an empty object, set the data of the empty object, and then call a save function that saves the object. The structure of the code would be as following:

```
const user = new User();
user.username = 'Tom Wood';
user.birthday = new Date(1999, 4, 27);
user.save();
```

What TypeORM offers that Sequelize does not, is full support for TypeScript. Our back-end is built in JavaScript, therefore there was nothing we were losing by opting for Sequelize instead of TypeORM. Since the code structure of Sequelize felt more natural to us we decided to go for it.

Sequelize also offers protection against SQL-injection attacks by escaping strings before they are sent to the database. The database will then treat all characters as normal strings and not as special characters despite what they may be.

## Chapter 3

# System Overview

In this chapter we will give a brief overview over the application system. Below is a high-level flow chart of our application.

### 3.1 Main Concepts

- **Teacher:** A user that can create a course, question sets, and view statistical information on how the students are progressing.
- **Student:** A user that can join a course and interact with course material.
- **Document:** HTML content created by a teacher that can be linked to a course.
- **Question Set:** An array of specific questions related to a document. A question set can be linked to a document or a course.
- **Questions:** A question that belongs to a question set. Stored as an element in the array that represents a question set.
- **Course:** Created by a teacher user. Contains documents and question sets that are related to a specific course.
- **Course Modules** A course module contains documents and questions that cover a section of the course syllabus.
- **Hidden Course Modules** A course module that has been created by the teacher but not visible to students.

### 3.2 Requirements

---

## 3.2 Requirements

In order to achieve our goal, there are a certain set of functionalities required.

A teacher needs to be able to:

- Register an account with an assigned Teacher role
- Create a course
- Create course material
- Create question sets and link them to a course / document
- View students' results in courses

A student needs to be able to:

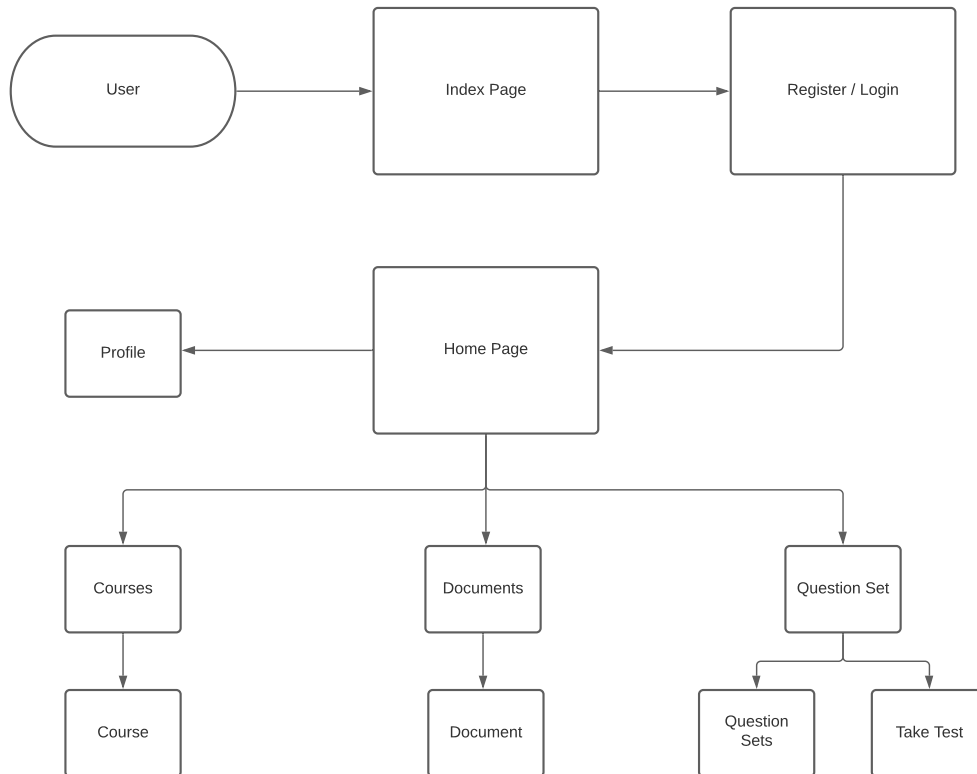
- Register an account with an assigned Student role
- Join a course
- Read through course material
- Create documents in which they can take notes
- Create question sets for rehearsal purposes
- Answer questions related to a course / document and submit the answers

### 3.3 Application Flowchart

---

## 3.3 Application Flowchart

Below is a high-level flowchart of our application to provide an overview of the different front-end components in this project.



When a user enters our web application, he is presented with our index page. From here, the user can either register himself or log in.

When logged in, the user can navigate through the different views from the home page. We will give a brief explanation of the different views and their functionality.

#### **Courses:**

In this view, the user can view the different courses that they are linked to. A teacher can view the courses they have created and a user can view the courses they are enrolled in. This template has two main functionalities: The first being that a teacher

### 3.4 Directory Structure

---

can create a course, and the second being that a student can join a course.

**Course:**

The user can navigate to this template by clicking on a course located in the Courses template. In this template, the user will be presented with the course material for that particular course.

**Question Sets:**

This view contains a list of question sets. The user can create new question sets from here.

**Take Test:**

A user can navigate to this view from the Question Sets view by interacting with a specific question set from the list of question sets. In this view, the user is presented with the questions stored in the question set. Main functionality in this view is being able to answer the questions and submitting the answers.

**Documents:**

In this view, the user can view the different documents that they have created. Main functionality in this view is being able to create new documents.

**Document:**

A user can navigate to this view from the Documents view by clicking on a document. In this view the user is presented with the contents of the document. Main functionality of this view is being able to edit and saving the contents of the document.

**Document:**

A user can navigate to this view from the Documents view by clicking on a document. In this view the user is presented with the contents of the document. Main functionality of this view is being able to edit and saving the contents of the document.

**Profile:**

In this view the user is presented with the user information.

## 3.4 Directory Structure

For a visual representation of the directory structure please refer to Appendix C.

The project file structure has two main parts; app and server. The app folder contains the front-end code and the server folder contains the back-end code.

### 3.5 Key Components

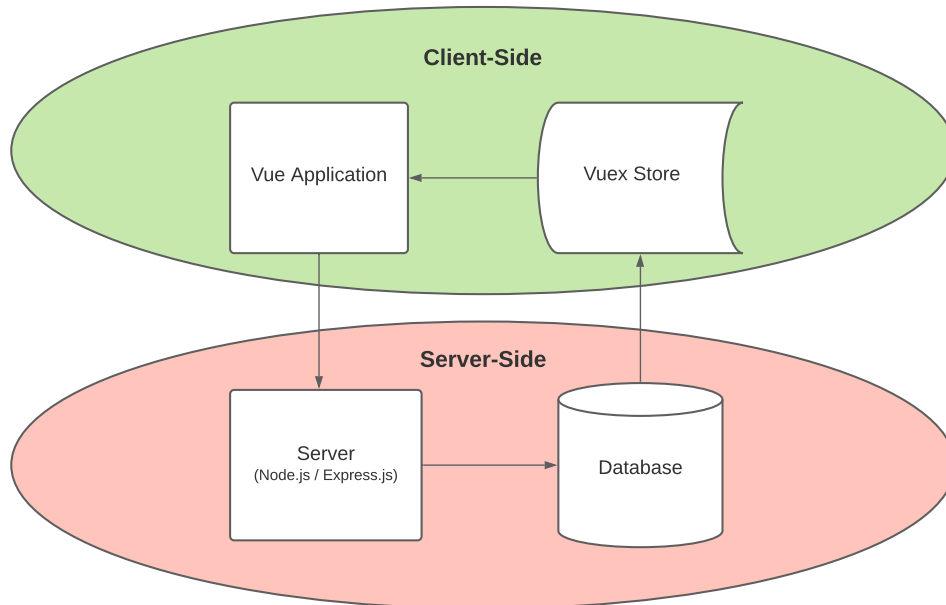
---

A lesser part of the application resides in the `.github` folder. The folder contains github related files that allow for the setup of a pipeline that automates workflow upon pushing to the main branch or creating a pull request to main.

We will dive deeper into the contents of some key files and folders in their respective sections.

## 3.5 Key Components

The back-end consists of two main components: Server application and the database. The server communicates with the front-end. A key component in the front-end that we need to abstract from the Vue application is the Vuex Store. The Vuex Store functions as a temporary client-side database. Below is a high-level overview of the interaction between these components:



**Figure 3.1:** High-level Overview of Application Components

The server-side components will be explained in more detail in Chapter 4, while the client-side components and its interaction with the server-side will be explained in Chapter 5.

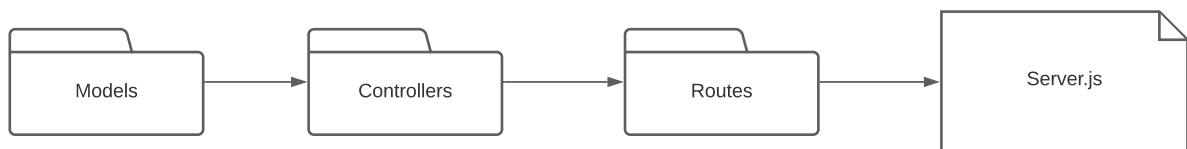


## Chapter 4

# Back-end Structure

In this chapter we will discuss the setup of a server, creation of database tables and models, and the architectural design pattern of the routes and their contents.

There are four files to keep track of, and each file builds on top of the other. A high-level overview of the composition of these files can be seen below:



**Figure 4.1:** High-level Overview of Application Components

The relationship between the files is as follows:

- The Server.js file imports routes from the Routes folder.
- Routing files import the routing logic from the Controllers folder.
- Controller files import database models from the Models folder in order to create functions that execute queries.

A brief walk-through of this process with dummy-code:

## Back-end Structure

---

Create a User model:

```
"use strict";
const {
  Model
} = require("sequelize");
module.exports = (sequelize, DataTypes) => {
  class User extends Model {
    static associate(models) {

    }
  }
  User.init({
    firstName: DataTypes.STRING,
    lastName: DataTypes.STRING,
  }, {
    sequelize,
    modelName: "User",
  });
  return User;
};
```

Create a controller file.

Import the model and write the logic for creating a user.

```
// filename: user.controller.js

const models = require("../models/index.js");

const register = (request, response) => {
  let user = await models.User.create({
    firstName: request.body.firstName,
    lastName: request.body.lastName,
  });
  return response.status(200).json({user})
}

module.exports = {
  register
}
```

## 4.1 Server Setup

---

Create a router file.

Import the controller file and assign a URL to the imported functions:

```
// filename: user.routes.js  
  
const userController = require("../controllers/user.controller")  
  
router.get("/api/register", userController.register )
```

Import the router file in the server.js file.

Tell the server to make use of the imported router.

```
// filename: server.js  
  
const userRoutes = require("./routes/user.routes")  
  
app.use(userRoutes)
```

The following subsections will give an explanation at a lower-level of the structure and contents of the different files.

## 4.1 Server Setup

Getting a server up and running using Node.js and Express requires no more than a few lines of code.

We import the express module in our server.js file using the `require()` statement.

```
var express = require("express");  
  
var app = express()  
  
var port = 3000;  
app.listen(port, () => {  
  console.log("listening on port 3000");  
});
```

## 4.2 Database

---

We create a variable and set it equal to the function `express()` provided by the `express` module. The function creates an Express application.

The Express application has a function called `listen()`. The function returns an http server we can communicate with on the port provided to the function as a parameter.

To start the server, we run the `server.js` file with the following script in our terminal: `"npm server:dev"`. The script runs the following command: `"nodemon ./src/server.js"`.

Now that the server is up and running, the next step is to create a database.

## 4.2 Database

For our database, we used MySQL.

Before we started working on the project, we created a database diagram of the different tables and relations we would need for our application. The database diagram can be found in Appendix B.

This database was implemented into our project by using a JavaScript library called the Sequelize ORM.

Sequelize comes with a CLI that allows us to create models, and provides functions to modify models and execute MySQL queries using JavaScript functions.

Below is a walk-through on the creation of a course table, defining the relations between tables and writing queries to fetch their data.

In our project, users can join a course and later interact with the courses they have joined. The way we solved this was by creating three tables:

- User table
- Course table
- Junction table to link the user to the course

To create the tables we use the Sequelize CLI and write the following commands:

Create user table:

## 4.2 Database

---

```
npx sequelize-cli model:generate --name users --attributes
username:String,fullname:String,password:String,role:String
```

Create course table:

```
npx sequelize-cli model:generate --name courses --attributes
courseName:String,shorthand:String,coursePassword:String,
userID:Integer
```

Create junction table:

```
npx sequelize-cli model:generate --name StudentCourseJunction
--attributes userID:Integer,courseId:Integer
```

After running these commands, Sequelize generates two files for each table: A migration file, and a model file.

The migration file contains all the attributes of the table. In this file we can alter the table by adding additional attributes or adding additional fields to an attribute. Below is the migration file for the courses table:

## 4.2 Database

---

```
"use strict";
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable("courses", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      courseName: {
        type: Sequelize.STRING
      },
      shorthand: {
        type: Sequelize.STRING
      },
      coursePassword: {
        type: Sequelize.STRING
      },
      userId: {
        type: Sequelize.INTEGER
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable("courses");
  }
};
```

Here we can observe that three attributes we did not specify in our command exist, the `id`, `createdAt` and `updatedAt` attributes. These are included by default by Sequelize. They can be easily removed by simply deleting the objects, but we chose to let them be.

At this point, our database is still empty and no tables have been created. In order

## 4.2 Database

---

to create the tables in our database, we need to run the migrations. To run the migrations, we write the following command in our terminal:

```
npx sequelize-cli db:migrate
```

The tables have now been created in our database, and we can move on to the next step, which is defining relations between the tables.

The models file contains all the specified attributes of the table and a class in which one can define associations between tables. Associations is the term Sequelize uses for relations. Below is a modified version of the courses model file:

```
"use strict";
const {
  Model
} = require("sequelize");
module.exports = (sequelize, DataTypes) => {
  class courses extends Model {
    static associate({StudentCourseJunction}) {
      this.belongsTo(StudentCourseJunction, {foreignKey: 'id',
        onDelete: 'cascade'})
    }
  }
  courses.init({
    courseName: DataTypes.STRING,
    shorthand: DataTypes.STRING,
    coursePassword: DataTypes.STRING,
    userId: DataTypes.INTEGER
  }, {
    sequelize,
    modelName: "courses",
  });
  return courses;
};
```

The function `associate` takes an object as a parameter. The object is the model you would like to create an association with. In this case it is the `StudentCourseJunction` model.

## 4.2 Database

---

The function `this.belongsTo()` takes in a required parameter and optional parameters. The required parameter is the object we passed to the associate function. The optional parameters are additional information on how the association should behave.

In this case, we specify that the association should be created upon the 'id' field of the `StudentCourseJunction` table. The `onDelete: 'cascade'` parameter specifies that upon deletion of a course, related entries in `StudentCourseJunction` shall also be deleted.

The function `this.belongsTo()` is one of many functions provided by Sequelize to define associations. Other functions include:

- `this.hasOne()`
- `this.hasMany()`
- `this.belongsToMany()`

Now that the tables are created and the associations are defined, we can move to the last part, which is to write queries for these tables.

In our project, a teacher user can create a course which a student can later join. The student can then view all the courses that he has joined on his personal page.

To achieve this, we need to write queries to create a course, join a course, and fetch a course. Following will be an explanation of the process of creating a course:

First thing we do, is verify the token received from the front-end and check the role of the user. We would only like to execute the query if the user is a teacher.



## 4.2 Database

---

```
const create_course = (request, response) => {  
  
  let token = request.headers.token;  
  
  jwt.verify(token, "secretkey", (err, decoded )=> {  
    if(err) return response.status(401).json({  
      title: "unauthorized",  
      error: err  
    });  
  
    if (decoded.role.toLowerCase() !== "teacher") {  
      return response.status(401).json({  
        title: "unauthorized",  
      });  
    }  
  }  
  
})
```

The function `jwt.verify()` has an error and an object called `decoded`. The `decoded` object contains the information stored in the token, such as `userId`, `fullname` and `role`.

The `decoded` object is only defined if the verification of the token is successful. The error object is only defined if the verification of the token is not successful.

We write an `if`-statement to check if an error exists, if it does, we return an HTTP response of status 401 to the front-end with the title "unauthorized" and the error object.

If the error does not exist, we write an `if`-statement to check the role of the user. This query should only be executed if the role of the user is teacher. If the role of the user is not equal to teacher, we return a response of status 401.

If the user is verified and has the role of a teacher, we can go ahead and execute our query:

## 4.2 Database

---

```
models.users.findOne({where: {id: request.body.userId}})
  .then( async (users) => {
    let course = await models.courses.create({
      courseName: request.body.courseName,
      coursePassword: request.body.coursePassword,
      userId: users.id,
      shorthand: request.body.shorthand
    });

    let course_right_format = {
      courseId: course.id,
      courseName: course.courseName,
      courseShorthand: course.shorthand,
      Teacher: users.id,
      documents: [],
      courseModules: [],
      AssignmentModules: [],
      QuestionSets: []
    };

    return response.status(200)
      .json({course: course_right_format})
  });
}
```

We would only like to create a course if the user exists in our database. We first run a query to find the user.

The function `models.users.findOne()` is provided to us by the Sequelize framework and executes a `SELECT` query. We pass it arguments to specify which user we want to fetch from the database,

When the user is found, we have access to the 'users' object that contains the information stored about the user stored in our database. We can use this information in combination with the data in the request object to create a course.

In line 3, we declare a variable called `course`. We set the value of the variable equal to the response of the `models.courses.create()` function. The courses table has four attributes:  
`courseName`, `coursePassword`, `userId` and `shorthand`.

We set each attribute equal to its correspondent in the request and users object. The

### 4.3 Routes Structure

---

function then executes a CREATE query and creates an entry in our database.

We reformat the the newly created course to a format that will fit the structure of the course type in the front-end before we return it.

## 4.3 Routes Structure

To structure our routes, we decided to use the MVC architectural pattern and extend it a little bit further.

The pattern separates the routing into three main folders: Models, Views and Controllers. In our case, we added an additional folder called Routes.

The Models folder contains all the model files for the models in our database. These files are imported in the files that want to access the models in order to execute queries.

The Views folder is located in the front-end. It is from here we call the API endpoints. We will take a look at these files in Chapter 5.

The controllers folder contains the files that hold the logic of all the endpoints, but not the actual endpoint. We grouped the functions by their relevance. For instance, all functions related to a course would be stored in the `course.controller.js` file.

The routes folder contains files that hold the endpoints of the functions in the controller files. We import the functions from the controller files to our routes file, and assign them their corresponding endpoint.

## 4.3 Routes Structure

---

### 4.3.1 Controllers and Routes

The structure of a controller file is as follows:

```
// import the models from the models folder.  
// import JWT to use for authorizing API calls  
const models = require("../models/index.js");  
const jwt = require("jsonwebtoken");  
  
// Create functions for different calls  
const getCourses = (request, response) => {  
  
}  
  
const create_course = (request, response) => {  
  
}  
  
//Export the functions  
module.exports = {  
  getCourses,  
  create_course  
}
```

These functions are imported in the file `course.routes.js` located in the `routes` folder. The structure of the file is as follows:

```
const express = require("express");  
const router = express.Router();  
  
// We import the course controller file  
const courseController =  
require("../controllers/course.controller")  
  
// We create a post method  
router.post("/api/createCourse", courseController.create_course)  
  
// get all courses using user id  
router.get("/api/getCourses", courseController.getCourses)  
  
module.exports = router;
```

### 4.3 Routes Structure

---

Commenting on the code block above, first off, we import the express module. We create a constant called router and assign it the `express.Router` class. The class is a complete routing system that can be used to create route handlers such as get and post methods.

Then we import the course controller file.

We create post and get methods and assign them an endpoint URL and a function from the course controller file.

We then export the router variable.

In order to access these routes, we need to import them in our `server.js` file and tell the Express server application to use the files. Below is an example of this process:

Importing a route file in our `server.js` file:

```
const courseRoutes = require("./routes/course.routes.js")
```

Tell our express application to use the imported file:

```
app.use(courseRoutes)
```

Now our server has access to the endpoints in `course.routes.js` and will be able to handle their requests.

The motivation behind refactoring our code in this way was to get a more modular and cleaner code architecture. At the beginning of the development process we wrote our API calls directly into the `server.js` file. As we got further into the development the file got very long and all the calls were unorganized. Every time we wanted to modify a specific function we had to scroll through hundreds of lines of code to find the function of interest.

After refactoring the code, it is clear which file contains which routes and functions, making it easier to work with.

## Chapter 5

# Front-end Structure

In this chapter we will go through the UI and some of the main components in the front-end of our application.

### 5.1 User Interface

When deciding on the external design of our application we decided to not re-invent the wheel and opted for a simplistic and lightweight design. This allowed us to focus more on essential functionality of the application while at the same time developing an intuitive and beginner friendly user interface.

Our philosophy whenever implementing a new interface was to keep things simple wherever possible. However, sticking true to this philosophy wasn't always easy as some functionality such as creating a course module required a more sophisticated design.

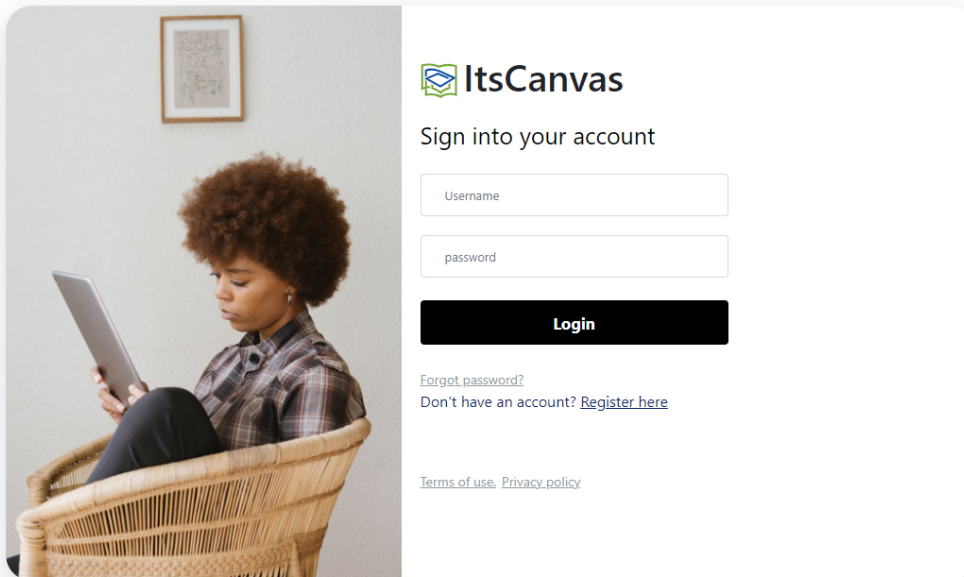
The following subsections will briefly cover the main views of the application and their purposes.

## 5.1 User Interface

---

### 5.1.1 Login Page

The /login route renders the following view.



**Figure 5.1:** Login View

Its main purpose is to allow users to log in. There are links to give it a professional look, such as the "Forgot password?" and "Terms of use", "Privacy Policy". These are at the moment dead links with no functionality. The "Register here" link works as supposed to and redirects the user to the registration page.

## 5.1 User Interface

---

### 5.1.2 Home Page

When a user is logged in, he is directed to the home page of the application.

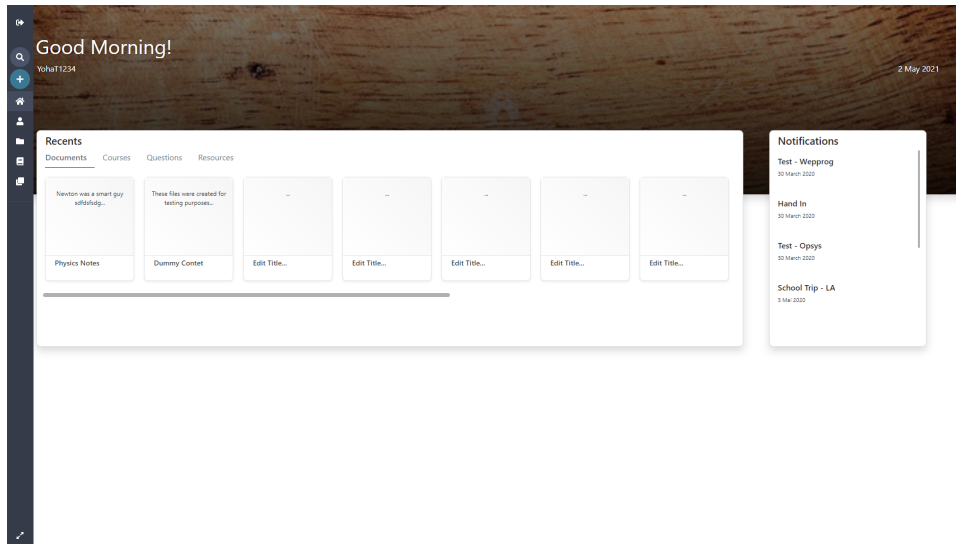


Figure 5.2: Home Page

The home page is equipped with a component that displays the most recent documents and courses interacted with. Alongside that is another component with the title Notifications. The notification bar was a feature that we wanted to implement, but were not able to due to time constrictions. The data inside it is hard-coded values with no functionality.

To the left side of the home page is a navigation bar. In this figure the navigation bar is contracted, but it can be expanded by clicking the button with arrows to the bottom left. When expanded it looks like this:



## 5.1 User Interface

---

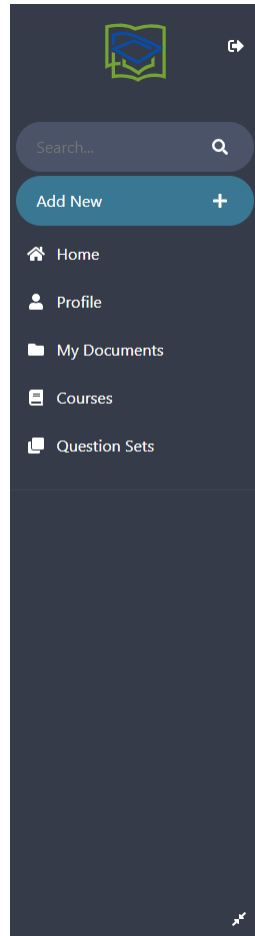
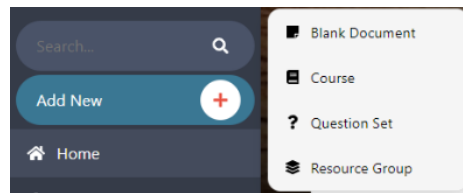


Figure 5.3: Navigation bar

The navigation bar makes it easy to navigate around in the application. The links have clear names and icons that represent where they redirect to.

Hovering over the + icon on the blue Add New bar activates a menu of items

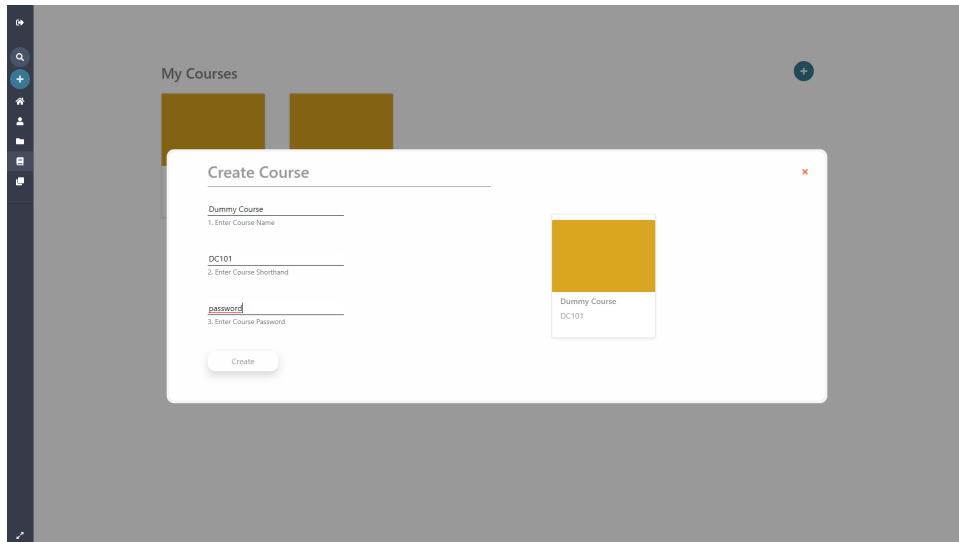


Clicking on one of the links will redirect the user to the appropriate page of creating the item that was clicked on.

## 5.1 User Interface

---

### 5.1.3 Courses Page



**Figure 5.4:** Courses Page

This page contains an overview over the courses a teacher has created and the courses a student has joined. A teacher is able to create courses from this page and a student is able to join courses. The modal for creation of courses is illustrated in the figure below. Joining a course follows the same design but slightly modified.

When clicking on a course, a user is redirected to the page of that particular course.

## 5.1 User Interface

---

### 5.1.4 Course Page

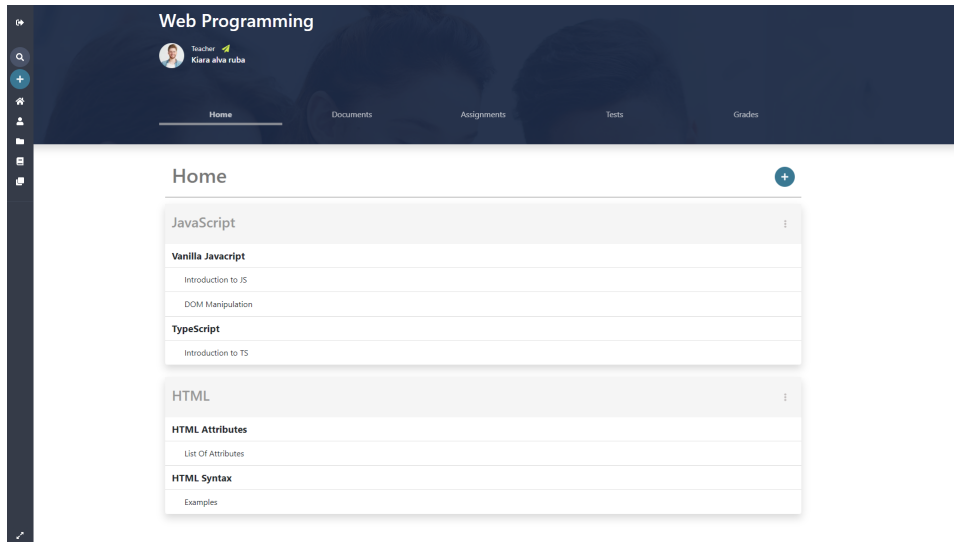


Figure 5.5: Course Page

In this page, the user can get an overview over the different modules and sections of the course and will be able to navigate directly to the course material of those sections by clicking on them. A user can view the documents linked to a course by clicking on the Documents tab located in the header. A teacher can view student data of that course by clicking on the Grades tab.

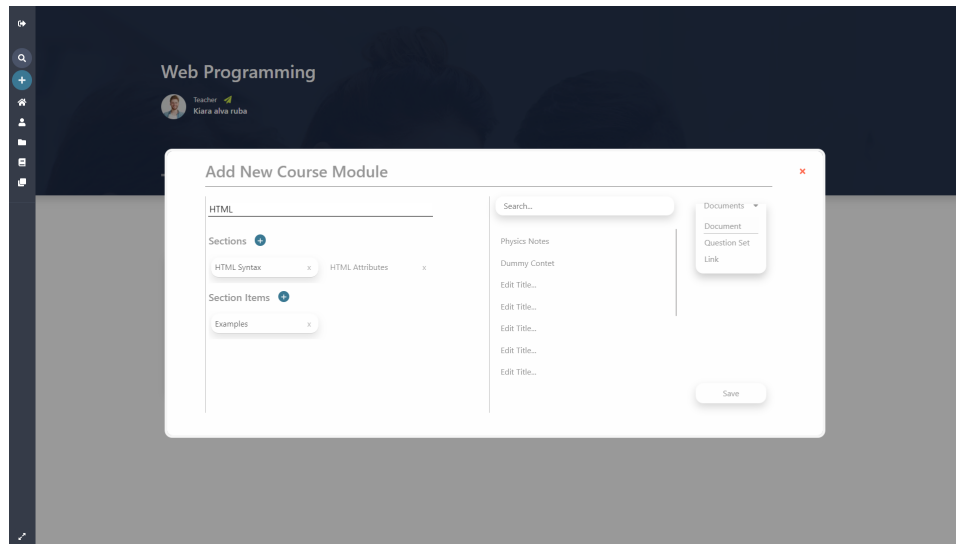
The other tabs in the header, Assignments and Tests, were ideas that we had in mind that we could not get around to implement due to time constrictions.

Teachers can add and modify the course material by clicking on the blue circle with a + icon located on the upper right part of the page.

The interface for modifying / adding a course module is illustrated in the figure below.

## 5.1 User Interface

---



**Figure 5.6:** Create Course Module Interface

A teacher can add sections to the module, and items to each section. The items can be a document, a question set, or a link.

## 5.1 User Interface

---

### 5.1.5 Documents Page

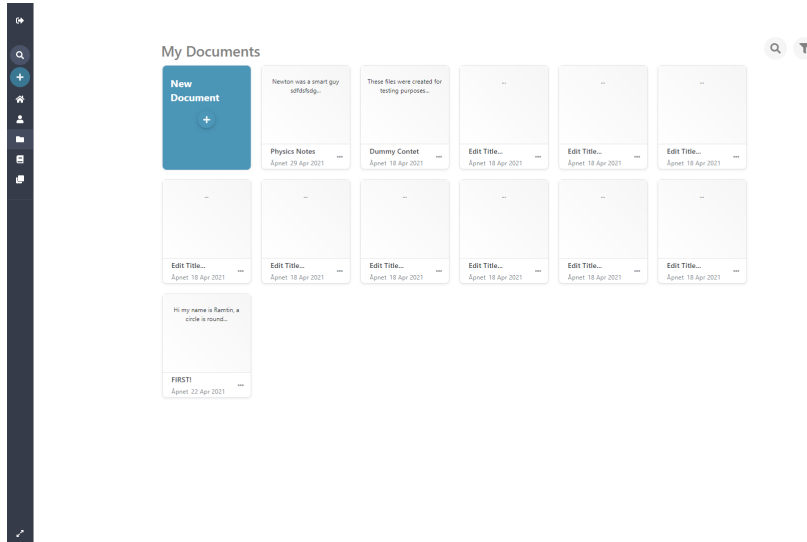


Figure 5.7: Documents Page

This page gives the user an overview over all their documents. By clicking on the blue card labeled "New Document" users will be redirected to a new document that they can edit.

## 5.1 User Interface

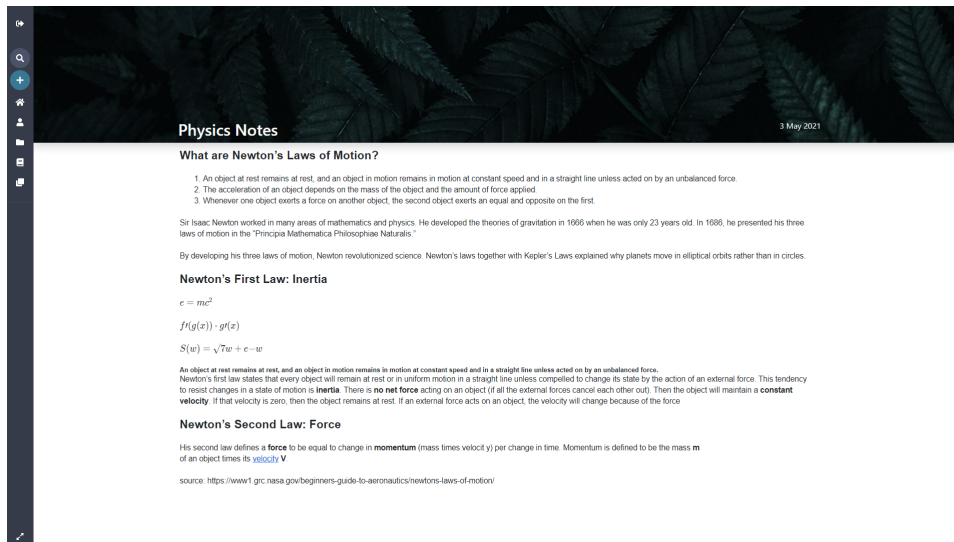


Figure 5.8: Editor View

This page is of a document belonging to a user. The user can edit the title and the text in the document as he wishes. By selecting some text or right clicking on the editor a customization bar appears:



Figure 5.9: Editor View

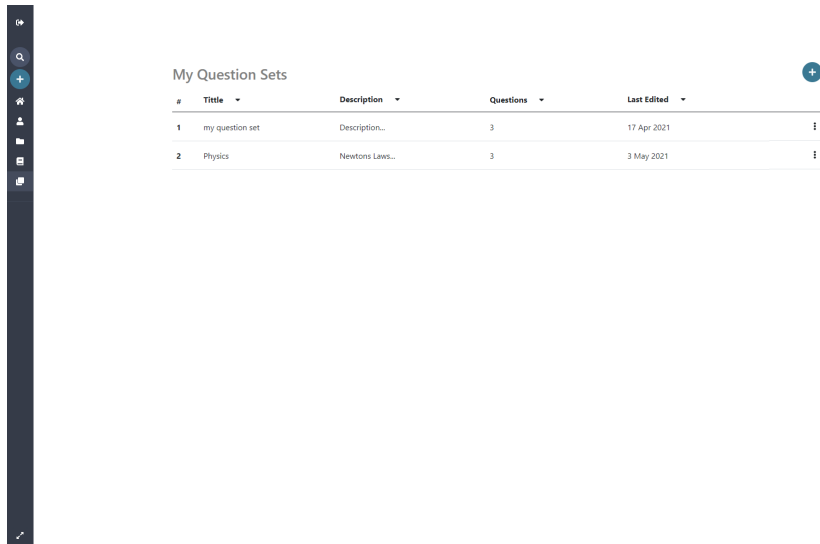
This bar has a lot of options for customizing the text, such as the placement of the text, font, weight and even allows for typing mathematical formulas that are formatted correctly.

## 5.1 User Interface

---

### 5.1.6 Question Set Page

The question set page contains a list of all the question sets a user has created.



#	Title	Description	Questions	Last Edited
1	my question set	Description...	3	17 Apr 2021
2	Physics	Newtons Laws...	3	3 May 2021

**Figure 5.10:** Question Set Page

A user can interact with the question sets in two different ways. They can open an existing question set to modify its contents, or they can choose to take a test and answer the question sets.

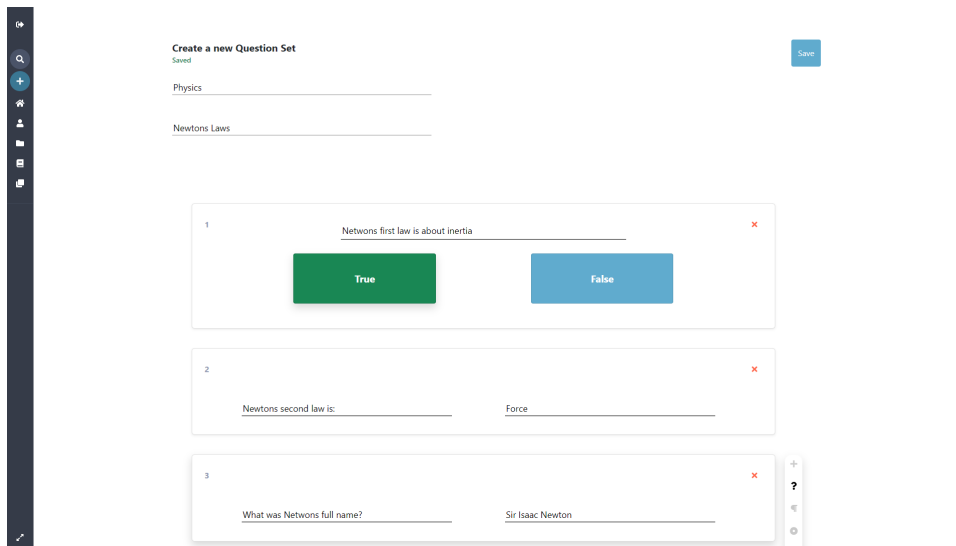
The user can create a new question set that will be added to the list by clicking on the blue circle with a + icon on the top right of the page.

## 5.1 User Interface

---

### 5.1.7 Create Question Set Page

The following figure illustrates the page in which a teacher can create and modify a question set.



**Figure 5.11:** Create Question Set Page

There are three different types of questions that can be made: short answer, long answer, true or false and multiple choice.

To create a new question one can simply click on any question and a list of icons will appear next to the question component. On top of the list is a + icon. Clicking it will generate a new question component under the last one. The other icons are for changing the type of the question.

The page for answering the questions follows the same design, but differs in the way that the answers are not shown to the user. Additionally, the user will have a side bar that shows the amount of questions there are in the question set in a numbered order.



## 5.2 Front-end Components and Technologies

Before proceeding, a clarification must be made regarding to terms used in this section. Previously when we have talked about components in a diagram we have been referring to an entity in that diagram. The front-end contains files called component files. To avoid confusion, throughout this section when mentioning a component we refer to the component file and not a component as in a piece of technology. A component as in a technology will be referred to as an entity.

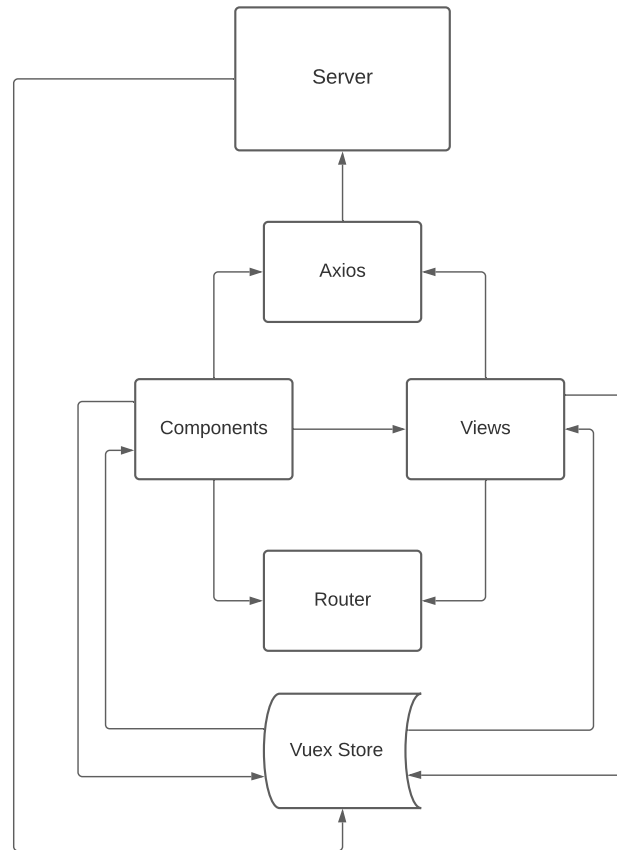
The files and entities that will be discussed are the following:

- **Vue Framework** - The framework in which we build our application
- **Components** - Files that contain reusable code that we import in our views
- **Views** - Files that contain the code that is rendered to the user
- **Vuex Store Library** - A temporary in-memory data storage that is stored in the client-side
- **Vue Router Library** - Used to navigate through the different views
- **Axios Library** - Used to communicate with the back-end

Figure 5.1 below aims to illustrate how the different entities work together:

## 5.2 Front-end Components and Technologies

---



**Figure 5.12:** High-level Overview of Application Entities

Walk-through of the figure:

Components are imported by Views.

Both components and views communicate with all the other entities located in the client-side. This is because those are the files that the user will see and interact with.

Whenever a user wants to navigate from one view to another, it must make use of the routes that are defined in the Router folder and the functions provided by the Vue Router library:

When a user wants to create new data, he must do so through the functions provided by the Axios library that send requests to the server.

## 5.2 Front-end Components and Technologies

---

When the server sends a response back to the front-end, it is stored in the Vuex Store entity.

When a user wants to fetch information, it executes Getter() methods located in the Vuex Store entity that retrieve the requested information.

All these entities will be explained at a lower-level in their respective sections.

### 5.2.1 Vue Framework

We create a Vue application by executing the following command in our terminal:

```
1 vue create Bachelorthesis
```

It will create a folder called Bachelorthesis with the complete structure of the project.

The structure of the project initialized by Vue contains the following folders which are all located in the app/src folder:

- **assets:** Contains static images for our application
- **components:** Contains reusable component files that are imported by View templates
- **Router:** Contains an index.ts file where the routes are defined
- **Store:** Contains files that keep track of the state of the application
- **Views:** Contains the template files that will be rendered to the user
- **main.ts** This is a file in which we initialize our Vue application and its libraries

we will dive deeper into the different folders to better explain their functionality. But before we do, we would need to explain the structure of Vue files.

## 5.2 Front-end Components and Technologies

---

A Vue file is divided into three sections:

- HTML section (lines 1-3 in the code block below)
- Script section (lines 5-14)
- CSS section (lines 16-21)

```
1 <template>
2   <h1 class="Header" @click="DummyFunction">Hello World</h1>
3 </template>
4
5 <script lang="ts">
6   import Vue from 'vue'
7   export default Vue.extend({
8     methods: {
9       DummyFunction() {
10         console.log("Hello World")
11       }
12     }
13   })
14 </script>
15
16 <style>
17   .Header {
18     font-size: 36px;
19     color: teal;
20   }
21 </style>
```

The above file is named HelloWorld.vue and is a component. It can be imported and rendered by other Vue files.

To do so, we import the component in the script section of the file and render it in the HTML section:

## 5.2 Front-end Components and Technologies

---

```
<template>
  <HelloWorld/>
</template>

<script lang="ts">
import Vue from 'vue';
import HelloWorld from '@components/HelloWorld.vue';

export default Vue.extend({
  name: 'Home',
  components: {
    HelloWorld,
  }
});
</script>
```

This structure and synergy between components and views makes writing reusable code very straightforward and effective.

### 5.2.2 Components

Our project has 25 different components. A list of the components can be found in Appendix D alongside a list of the views in our application.

In this section, we will give a brief overview of the `courseCard.vue` component.

Below is the code of the HTML section of the `courseCard.vue` file:

## 5.2 Front-end Components and Technologies

---

```
1 <template>
2   <div
3     class="course-card-container card shadow-sm rounded"
4     @click="OpenCourse()"
5     v-test="{ id: 'course-card-container' }"
6   >
7     <img
8       src=""
9       alt=""
10      class="card-img-top course-image"
11      v-test="{ id: 'course-card-thumbnail' }"
12    />
13    <div class="card-body">
14      <h6 v-test="{ id: 'course-card-name' }">
15        {{ course.courseName }}</h6>
16      <p v-test="{ id: 'course-card-shorthand' }">
17        {{ course.courseShorthand }}
18      </p>
19    </div>
20  </div>
21 </template>
```

It contains a div element that has two child elements: an image and a div. All the elements have v-test directives. These will be discussed in chapter 6. Another thing of note is the double curly brackets. They contain the value of variables and are dynamic. course refers to a list of course objects, and courseName and courseShorthand are attributes of that particular course object.

The result of an arbitrary course card would look like this:



## 5.2 Front-end Components and Technologies

---

### 5.2.3 Views

A view in Vue is the same as a template file, which is the file that is rendered to the user.

We would like to let the user have an overview of all the courses they have joined. To achieve this, we have a template that renders those courses. Each course the user has joined is represented by one of the components pictured in the figure above.

Below is an extract of the piece of code in the Courses.vue file that renders those components:

```
</template>
  <div class="course-container-inner p-1">
    <course-card
      v-for="(course, index) in courses"
      :key="index"
      :course="course"
    />
  </div>
</div>
</template>
```

The element `<course-card>` is the imported component, and we pass it the `v-for` directive. The `v-for` directive renders a `courseCard` component for each element in the `courses` list.

Below is a stripped down example of the courses list:

## 5.2 Front-end Components and Technologies

---

```
courses {
  0:{
    courseId: 2
    courseName: Web Programming,
    courseShorthand: DAT101
  }
  1:{
    courseId: 2
    courseName: dummyCourse,
    courseShorthand: DC100
  }
  0:{
    courseId: 3
    courseName: History,
    courseShorthand: HIS101
  }
}
```

By the syntax `course.courseName` that we use in the `courseCard.vue` component file we are able to access the name of that specific course belonging to that card component. The result is as following:

### My Courses





### 5.3 Vuex State Management

Application state refers to how the program functions at any given time based on the contents in its memory. Keeping track of the data in memory is important in order for things to render correctly and for the application to function as expected. We therefore implemented a centralized state management system that stores all the data in our application.

The state management system was implemented by the use of the Vuex library. The implementation makes use of a design pattern called store pattern. The reason why it is called the Store pattern is because all the files that manage the state of the application are stored in a folder called Store.

The store folder consists the following folders:

- Helpers
- Interfaces
- Modules
- index.ts

The Interfaces and Modules folders are the two central folders in the state management system.

In Interfaces we define an object of a custom type. In this example we create a course object and assign it the type `courseType`.

```
export type courseType = {
  courseId: number;
  courseName: string;
  courseShorthand: string;
};
```

In modules we create a state object. In it we pass it the type interface we imported. Then we create functions to alter the data of the state object and functions to retrieve the data in the state object:

## 5.3 Vuex State Management

---

```
import { courseType } from "../interfaces/course";

export default {
  state: {
    courses: [] as courseType[]
  },
  mutations: {
    AddCourse: (state: any, course: courseType) => {
      state.courses.push(course);
    },
  },
  actions: {
    AddCourse: (context: any, course: courseType) => {
      context.commit("AddCourse", course);
    },
  },
  getters: {
    getCourses: (state: any) => {
      return state.courses;
    },
  }
};
```

Mutations are functions that alter the state object.

Actions are functions that trigger mutations.

Getters are functions that retrieve the data from the state object.

The examples above contain simplified code for demonstration purposes. In the following subsections we will dive into the interfaces and modules in greater detail with code that is more representative of what is found in our codebase.

### 5.3.1 Interfaces

In this folder we create files in which we define specific types.

For instance, we have a file called `user.types.ts`. In it we define types related to a user. Below is an example of types related to a user:

## 5.3 Vuex State Management

---

```
export enum RoleType {
  Student = "STUDENT",
  Teacher = "TEACHER"
}

export type UserType = {
  UserID: number;
  UserName: string;
  Role: RoleType.Student | RoleType.Teacher;
  FirstName?: string;
  LastName?: string;
};
```

The benefit of working with objects in this way is that every attribute of the object has a defined type. Errors due to invalid types or missing attributes are eliminated when working with objects as a pre-defined type.

Defining types are not required to implement a state management system, but they make updating our state management system much easier, as making changes to our types is caught by the typescript compiler, and notified to all other places the same type is in use. Objects in modules are assigned the types we create in this folder.

### 5.3.2 Modules

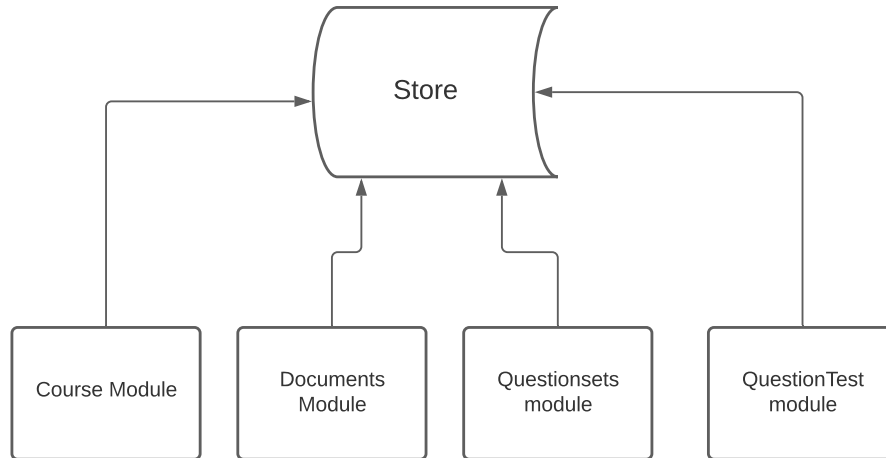
The definition of a store in Vuex is a container that contains the in-memory data of the application at any give moment.

The data in the store is dispersed through different modules, where each module contain their own store that keeps track of the data of a specific component.

We can therefore define a module as a mini-container that contains the data of a specific component in the application, and the store as a big container that contains all the different modules. A visual representation of this can be seen in the figure below:

### 5.3 Vuex State Management

---



**Figure 5.13:** The structure of a store

A module file contains a store.

To create a store, we import the Vuex library and the user type we had created earlier:

```
import { createStore } from "vuex";
import { UserType, RoleType } from "../interfaces/user.types";

const store = createStore({
  state: {
    isAuthenticated: false,
    user: {
      UserID: -1,
      UserName: "",
      Role: RoleType.Student,
      FirstName: "",
      LastName: ""
    } as UserType,

    activeUser: {},
    loading: false
  },
  ...
});
```

### 5.3 Vuex State Management

---

We create a store using the `createStore()` function provided by the Vuex library and initialize it with some default values.

The user object we pass in the state object is defined as the `UserType` interface.

A store has specific attributes that we are going to discuss in this section. The attributes are:

- Mutations
- Actions
- Getters

Mutations is an collection of functions that change the values of the state in the store when called upon.

```
mutations: {
  login: state => {
    state.isAuthenticated = true;
  },
  logout: state => {
    state.isAuthenticated = false;
  },
  setUser: (state, user: UserType) => {
    state.user.UserID = user.UserID;
    state.user.UserName = user.UserName;
    state.user.Role = user.Role;
    state.user.FirstName = user.FirstName;
    state.user.LastName = user.LastName;
  },
  ...
}
```

We write these functions ourselves and use them to change the state of the variables / objects defined in the state array, but we do not call the functions directly. Instead, we define an action for each mutation that we call upon whenever we want to execute one of the functions. The actions are defined like this:

### 5.3 Vuex State Management

---

```
actions: {
  loading: (context, loadingstatus) => {
    context.commit("loading", loadingstatus);
  },
  login: context => {
    context.commit("login");
  },
  logout: context => {
    context.commit("logout");
  },
  setUser: (context, user: UserType) => {
    context.commit("setUser", user);
  }
},
...
```

The keyword "context" is the state object. Through this keyword we have access to the elements in the state. In order to execute a mutation, we need to commit it to the state. It might seem counter intuitive to call on a function in order to perform a function. The reason for this is that mutations have to be synchronous, while actions do not. Therefore, we can run asynchronous actions and commit their result to the mutation which then mutates the state.

The order of operations to change a state is as following:

We make a call to an action using the `store.dispatch()` function from a view / component:

```
store.dispatch("login");
```

The function executes the following action:

```
login: context => {
  context.commit("login");
},
```

The action executes the following mutation:

### 5.3 Vuex State Management

---

```
login: state => {
  state.isAuthenticated = true;
},
```

The isAuthenticated attribute of state is now changed from false to true.

Below is a more comprehensive example of this process:

```
export function Login(username: string, password: string) {
  axios
    .post("/user", {
      username: username,
      password: password
    })
    .then(async (response: AxiosResponse) => {
      if (response.status === 200) {
        const user: UserType = {
          UserID: response.data.id,
          UserName: response.data.username,
          Role: response.data.role.toUpperCase(),
          FirstName: response.data.firstname,
          LastName: response.data.lastname
        };
        store.dispatch("setUser", user);
        store.dispatch("login");
        localStorage.setItem("token", response.data.token);

        store.dispatch("loading", true);
        await LoadStore();

        setTimeout(() => {
          store.dispatch("loading", false);
        }, 1000);

        router.push({ name: "Home" });
      }
    });
}
```

### 5.3 Vuex State Management

---

We perform an async API call to the database which returns a user object. We create a constant user and give it the type of UserType. The constant user object of type UserType is then passed on the data from the server. When we have our user, we dispatch the setUser() action which triggers the mutation that mutates the state.

We also dispatch the "login" action which changes the value of "isAuthenticated" from false to true to signalize that a user is logged in. When a user is logged in.

To retrieve values from the state, we need to define getter methods the same way we defined mutations and actions:

```
getters: {
  getIsAuthenticated: () => {
    return !localStorage.getItem("token");
  },
  getActiveUser: state => {
    return state.user;
  },
  getIsTeacher: state => {
    if (state.user) {
      return state.user.Role.toLocaleLowerCase() === "teacher";
    }
    return -1;
  },
  ...
},
```

After storing a response from the server into the store, we can call on the getter methods next time we would like to retrieve the object instead of making another request to the server.

In our course view, there are elements we would only like to render to a teacher. We can call onto the getter methods to retrieve their value in the following way:

```
const IsTeacher = computed(() => store.getters.getIsTeacher);
```

The constant IsTeacher is then used for conditional rendering.

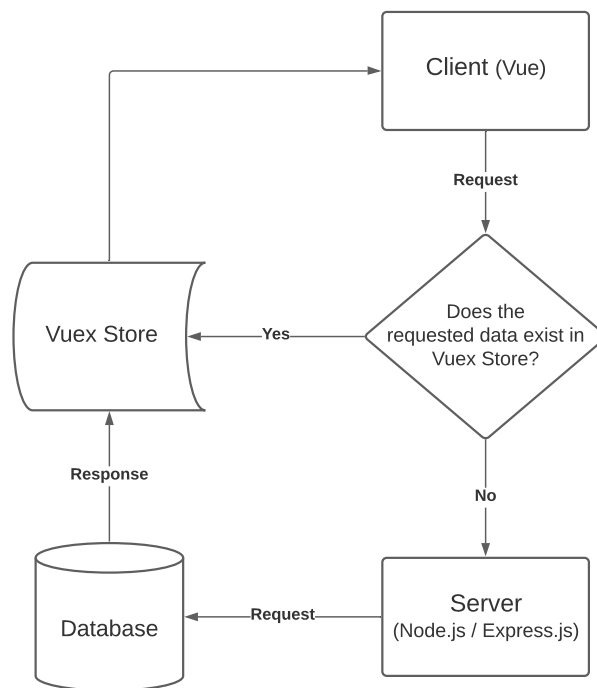


### 5.3 Vuex State Management

---

While most of the other libraries were a necessity for our application, the Vuex library was an optional one. There were two main benefits of implementing a store management system. The first one being that we have more control of the data that exist in-memory in our application, thus making it less likely that the application will behave in unexpected ways due to missing data. It also allows for easier debugging since the data is organized in a structured manner.

The other benefit is that it reduces the need for API calls by a great amount. When a user is logged in, we make requests to the database to retrieve all the existing information about the user and store it in the Vuex store. From then on, every time the user requests data it will be retrieved from the store and not from the server, thus eliminating excessive API calls. The only time we make requests to the server apart from when a user logs in is if the user wants to create new data that does not exist in the database. A diagram that illustrates this can be seen below:



**Figure 5.14:** High-level Overview of Application Components

This design choice puts less stress on the server and has a positive impact on the scalability of the project.

### 5.4 Router

Vue comes with a built-in library that handles routing. Upon creating a Vue-project, it auto-generates a router folder with an index.ts file with a complete route setup. The index.ts file looks like this straight out the box:

```
import Vue from 'vue'
import VueRouter, { RouteConfig } from 'vue-router'
import Home from '../views/Home.vue'

Vue.use(VueRouter)

const routes: Array<RouteConfig> = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    component: () =>
      import(/* webpackChunkName: "about" */ '../views/About.vue')
  }
]

const router = new VueRouter({
  routes
})

export default router
```

To define new routes we can simply add an object to the routes array. The object consists of three attributes:

path: The URL to access the route

name: The name of the route. Used as a reference to the route in certain functions

component: The file this route renders.

## 5.5 Axios Web Service

---

In the component field we can either specify the name of the file directly in the component attribute or pass it into a function which implements lazy-loading. The function extracts the code of the route from the js bundle and creates a separate chunk for it. This chunk of code is only loaded when a user enters the route. It is easy to implement and improves the performance of the application significantly.

The Vue-router library has functions that allows us to have more control over the routes. Take this function as an example:

```
{
  router.beforeEach((to, from, next) => {
    const publicRoutes = ["Login", "Welcome", "Register"];
    const authRequired =
      !publicRoutes.includes(to.name as string);

    if (authRequired && !IsAuthenticated()) {
      Logout();
      next({ name: "Login" });
    } else next();
  });
}
```

The function `router.beforeEach` is executed before each navigation.

We would like to check if the user is authenticated when navigating through the application.

We write an if-statement that checks if the user is trying to navigate to a route that requires authentication and if the user is authenticated.

If the user is not authenticated, we redirect him to the Login page, and if he is, we go ahead and let him navigate to the desired route.

## 5.5 Axios Web Service

Axios is used to make asynchronous promise-based HTTP requests.

It is easy to use out of the box, we import it into our views / components and start making requests to the server:

## 5.5 Axios Web Service

---

```
const axios = require('axios');

axios.post("/createCourse", {
  courseName: "Physics",
  coursePassword: "1234",
  userId: "3",
  shorthand: "PHY100"
})
.then(response => {
  console.log(response)
});
}
```

The code above will hit the /createCourse endpoint in our server which will execute a query to create a course in our database based on the data received from the Axios call.

### 5.5.1 Customize Default Axios Settings

It is possible to customize the way requests are handled by Axios, and in our case this was needed. In order to do this, one needs to create a new instance of Axios and define the custom settings.

Following the singleton pattern, we created a new file in which we created a new instance of Axios that could be reused and imported whenever needed:

```
import axios, { AxiosError,
  AxiosRequestConfig,
  AxiosResponse } from "axios";
import { ApiConfig } from "@config/api.config";
import router from "@router";

const axiosInstance = axios.create({
  baseURL: ApiConfig.API_URL
});
```

We configure it with a URL which it will send requests to. In this case during devel-

## 5.5 Axios Web Service

---

oment it is "http://localhost:3000/api".

When the instance is created we can go ahead customize the way requests are handled by using interceptors. Interceptors are functions that are called whenever a request is made by Axios.

A lot of our requests required a token being sent in the header of the request. This process was very repetitive and required excess code that could be automated. We wrote the following function to address this issue:

```
axiosInstance.interceptors.request.use(
  (request: AxiosRequestConfig) => {
    if (
      !(router.currentRoute.value.name === "Login") &&
      !(router.currentRoute.value.name === "Welcome") &&
      !(router.currentRoute.value.name === "Register")
    ) {
      const token = localStorage.getItem("token");
      if (token) {
        request.headers.token = token;
        return Promise.resolve(request);
      }
      return Promise.reject(request);
    }
    return Promise.resolve(request);
  },
  (error: AxiosError) => {
    return Promise.reject(error);
  }
);
```

All requests executed in routes that were not "Login", "Welcome" or "Register" were in need of a token.

We wrote an if-statement to check that the current route were none of the three. If it was not we retrieved the token from the local storage and passed it onto the header of the request.

If there were no token in the local storage, we would reject the request and not go through with it.

The last scenario is if the current route is one of the routes mentioned above. In that case we resolve the request with no token.

## 5.5 Axios Web Service

---

In addition to automatically passing in the token in the request header, it also adds an extra layer of security by checking if the token exists in local storage before resolving the request.

The second interceptor function we made was to handle errors. It addresses the same issue as the first one in terms of redundant code. Errors with status 401 or 404 were handled the same in all requests, so instead of writing the same code for every request we wrote a function that automatically handles those errors the way we want:

```
axiosInstance.interceptors.response.use(
  (response: AxiosResponse) => {
    return Promise.resolve(response);
  },
  (error: AxiosError) => {
    if (error.response && error.response.status) {
      console.log(error.response.status);
      if (error.response.status === 401) {
        localStorage.removeItem("token");
        router.push({ name: "Login" });
      } else if (error.response.status === 404) {
        router.push({ name: "PageNotFound" });
      }
      return Promise.reject(error.response.status);
    }
  }
);
```

If we receive an error, we check the status code of that error.

If it is 401 which is returned when the user is unauthorized, we remove the token from local storage if it exists and redirect the user to the Login page.

If the status of the error is 404, this means that the view the user wants to access is not found / does not exist. In that case, we redirect them to our "PageNotFound" route which displays our project logo and the text "Page Not Found".

Now that we have created a new instance of Axios and configured it the way we want, we can start sending requests to the server.

## 5.5 Axios Web Service

---

### 5.5.2 Axios Requests

We created a file for each group of requests, such as `courses.service.ts` and `documents.service.ts`. We wrote the API calls in these files instead of writing them directly into the view / component files. Structuring the code in this way made it cleaner and more readable.

Below is an example of a request to create a course in the `courses.service.ts` file:

```
export function CreateCourse(course: courseType,
                             coursePassword: string) {
  axios
    .post("/createCourse", {
      courseName: course.courseName,
      coursePassword: coursePassword,
      userId: course.Teacher,
      shorthand: course.courseShorthand
    })
    .then((response: AxiosResponse) => {
      if (response.status && response.status === 200) {
        store.dispatch("AddCourse", response.data.course);
      }
    });
}
```

We create a function called `CreateCourse()`. The function takes in an object of type `courseType` and a password of type `string`. We then send the object to the server which then executes a create query to create the course in our database. The result from the query is then sent back to the function as a response object.

We add the course object received from the server into our store by dispatching the `"AddCourse"` action which updates the state of the course module.

To call the function from a view / component, we import it and call on it:

## 5.6 Quill Text-Editor

---

```
import { CreateCourse } from "@services/api/course.service";

const create = () => {
  if (!Object.entries(user.value).length) {
    return;
  }
  if (user.value.Role.toLowerCase() !==
      RoleType.Teacher.toLowerCase()) {
    return;
  }
  course.value.Teacher = user.value.UserID;
  CreateCourse(course.value, coursePassword.value);
};
```

The function create performs checks to see if the user exists and if the user is a teacher. If it is, we call on CreateCourse() and pass it the required arguments.

## 5.6 Quill Text-Editor

The Quill text-editor is a central piece of technology in our application. It is what allows our teachers to create course documents and students to take notes. It will also provide important functionality regarding the monitoring of student interactions with a document.

Setting up Quill requires just a few basic steps. After installing the library we simply import it in our code and create a new Quill instance in the following way:

```
<script lang="ts">
import Quill from "quill";

var quill = new Quill(root.value, {
  placeholder: "Write something cool...",
  theme: "bubble",
  modules: {
    toolbar: true
  }
});
</script>
```



## 5.6 Quill Text-Editor

---

The function `new Quill()` takes in a container as the first argument. This container is usually a `div` HTML element. The way we display the Quill editor in our web application is to create a `div` element with a unique identifier. This unique identifier will be passed in the `new Quill()` function as the first argument, and the Quill editor will render itself in that specified HTML element.

As can be seen in the code above, in our case the identifier for our HTML element is the `root` attribute. Our HTML element that renders the editor can be seen in the code below:

```
1 <template>
2   <div ref="root">
3     </div>
4 </template>
```

### 5.6.1 Quill Building-Blocks

Quill documents use a parallel tree structure to the DOM tree which is called a **Parch-ment** as its document model. The Parchment tree consists of **Blots**, which are essentially a copy of a node in the DOM. The blots are what provide the structure, formatting and content in the document.

When we type in the Quill editor, Blots create HTML elements to represent the content we put in it. Below is a figure to illustrate this:

## 5.6 Quill Text-Editor

---

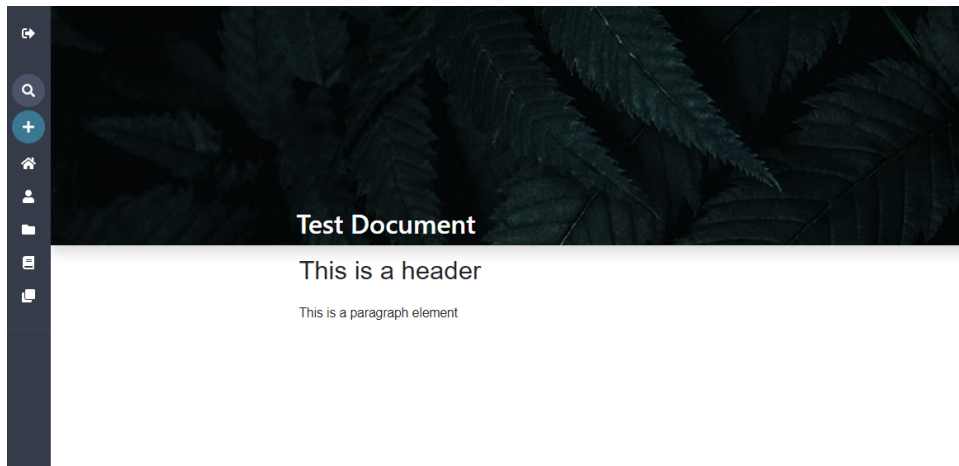


Figure 5.15: Quill Content Demo

The content in the figure above is represented by the following HTML:

```
<h1>This is a header</h1>
<p>
  <br>
</p>
<p>This is a paragraph element</p>
```

Figure 5.16: Quill HTML Demo

HTML elements are created and changed dynamically real-time as we change the content of the editor. If we were to change the header to normal text the h1 element would change to a p element.

One of the main reasons for why we choose quill as our editor, was because of how easy it is to extend the editor. To extend quill you need to extend the blot that provides the feature you want to change.

We needed to extend quill to give the created Blots custom attributes, so we could use that to define **Topic Scopes**. Quill has 2 basic text Blots, a **Header Blot** and a **Paragraph Blot**. We extended both of these blots so that each of them stored a custom attribute called **Topic-Id**. The next subsections in this chapter will explain how this was implemented at a lower-level.

## 5.6 Quill Text-Editor

---

### 5.6.2 Customizing Quill

As mentioned earlier, we needed to extend the different text blots to have a unique Topic ID. This would help us with defining what we have called a **Topic Scope**. A Topic Scope is what we use to define where a Topic begins and where it ends. This would later be used to calculate the time used on each Topic, so we can provide the teacher with feedback on where the students are spending their time.

In order to achieve this we created a class that overrides / extends Quill's built in header-blot and paragraph-blot class so that we can add some custom functionality in quills life-cycle.

## 5.6 Quill Text-Editor

---

```
import Quill from "quill";
import { TopicAttributeName, TopicHeaderClassName }
from "../constants";
import uuid from "uuid";

const Block = Quill.import("blots/block");

// This object will keep IDs to prevent duplicates,
// maps topicID to header level
export const TopicIDContext: {[topicID: string] : number} = {}

type IHeaderValue = {
  level: 1,
  ref: string // reference topic id
}

export default class HeaderBlot extends Block {

  public static create(value: IHeaderValue | number) : Element

  public static formats(domNode: Element): IHeaderValue

  private static AssignTopicId(domNode: Element, level: number)
  :void

  public attach() : void

  private DispatchAddedEvent() : void

  public get quill() : Quill
}

HeaderBlot.blotName = "header";
HeaderBlot.tagName = ["H1", "H2", "H3"]; // allowed header levels
HeaderBlot.className = TopicHeaderClassName;

MyQuill.register(HeaderBlot, true);
```

Out of these class methods the most important ones are **create**, **AssignTopicId** and **DispatchAddedEvent**.

The create method is called once the header blot is created, but before the blot is added to the document tree. This is where we either assign a existing Topic ID,

## 5.6 Quill Text-Editor

---

typically when its loaded from the database, or assign a new Topic ID through the AssignTopicId method.

The AssignTopicId method generates a uuid, universally unique identifier, which is a long string of characters, and assigns it to the domNode passed from the create method. To keep everything as extensible as possible we defined some constants, one of them being out TopicAttributeName. This TopicAttributeName, is used through out the project so that we can change at one place and it will be replicated through out the whole project.

Once the header blot has been created and the blot has been assigned a Topic-ID, quill's lyfe cycle continues until we again interrupt it right before its attached to the document tree. At this point the Attach method is called and we override that method to call **DispatchAddedEvent**, so that we can notify the editor containing all of these blots that a new header-blot has been added.This is done as shown below:

```
import Quill from "quill";

const Block = Quill.import("blots/block");

export default class HeaderBlot extends Block {

  private DispatchAddedEvent() {
    if (this.quill) {
      const TopicID = this.domNode
        .getAttribute(TopicAttributeName);
      if (TopicID && this.domNode.innerHTML !== "<BR>") {
        this.quill.container
          .dispatchEvent(new CustomEvent("header-added", {
            detail: {
              TopicID: TopicID,
              Topic: (this.domNode as any).innerText
            }
          }));
      }
    }
  }

  public get quill() : Quill
}
```

## 5.6 Quill Text-Editor

---

The process above was repeated in a similar manner with paragraphs as well. The major difference between these to blots is that the header-blot defines where a topic scope begins, while the paragraph-blots are used for keeping track of the different topic scopes.

```
import Quill from "quill";
import { TopicAttributeName }
from "../constants";
import uuid from "uuid";

const Block = Quill.import("blots/block");

type IParagraphValue = {
  block: "p",
  ref: string // reference topic id
}

export default class ParagraphBlot extends Block {

  public static create(value: IParagraphValue | string) : Element

  public static formats(domNode: Element): IParagraphValue

  public attach() : void

  private AssignTopicId() : void

  private AddEventListeners() : void

  private ParagraphInReadZone() : void

  public get Prev() : Element | null

  public get quill() : Quill
}
ParagraphBlot.blotName = "block";
ParagraphBlot.tagName = ["P"];

MyQuill.register(ParagraphBlot, true);
```

Out of these class methods the most important ones are **create**, **AssignTopicId** and **ParagraphInReadZone**. Just as with the header-blot the create method creates the blot. The difference here is that we are not generating a uuid, if the paragraph blot

## 5.6 Quill Text-Editor

---

doesn't have one, because as explained that paragraph blots are only used to define the length of a topic, but not for defining one.

Topic IDs are assigned through the `AssignTopicId` method which is called only when the blot has been **attached**, to the document tree. This is because it then has a reference to the previous node. That node is accessed through the getter method `Prev`, and we assign the previous node's Topic ID to the current. This then results in a document like this,

```
<h1 class="topic-header" data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">Calling a function</h1>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<pre class="ql-syntax" spellcheck="false">...</pre>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
  <p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">the function my_function() will print "Hello from a function".</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="cf2a9970-b4c3-11eb-aa29-010481db79dd">...</p>
<h1 class="topic-header" data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">Arguments</h1>
<p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">...</p>
  <p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">Information can be passed into functions as arguments.</p>
<p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">...</p>
  <p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">The following example has a function with one argument (fname).</p>
<p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">...</p>
<p data-topic-id="0496e640-b4c4-11eb-aa29-010481db79dd">...</p>
```

Figure 5.17: Quill HTML Demo

As you see at the top we have an `h1` element that is highlighted in yellow. It has been given an attribute `"data-topic-id"` which is equal to a unique id provided by the function `uuid.v1()`. All the following paragraph elements below the header have also been assigned the same attribute up until when the next header is defined.

Lower on the figure we can see we have another `h1` element which also has a `"data-topic-id"`, but its value is different from the other `h1` element, and all the following paragraphs under it have also been assigned the same attribute with the same value.

The reason we do this is to be able to define the scope of a topic. By giving a unique id to all the elements in a topic, we are able to retrieve their position in our DOM and use those values to determine where a topic begins and where it ends. We can use that information to monitor which topic a student is reading, and then take the time on how much time a student spends on a certain topic. The implementation of how this is achieved is discussed in Chapter 7.

## Chapter 6

# Test Driven Development

We wrote many unit tests for our application in the front-end in order to implement a continuous integration environment to our project. The purpose of the tests was to check that all the components and their content were rendered properly.

### 6.1 Implementation

To write tests in Vue, we need to install three different libraries:

- Vue Test Utils
- Mocha
- Chai

Vue Test Utils provides tools that make it possible to write unit tests for components in Vue, but it is not the library that we use to write the actual tests. The main tool that we have made use of is the function called `shallowMount()`. It essentially stores a component inside an object called `wrapper`. We then have access to the HTML elements in the component through the `wrapper` object.

Mocha library provides functions for writing the tests and is the library that runs the tests.



## 6.1 Implementation

---

Chai provides functions that allow us to make assertions that we check for.

Before we take a look at an example of a unit test, we will first look at the component that we will write a test for.

Below is the HTML code for a course card:

```
<template>
  <div
    class="course-card-container card shadow-sm rounded"
    @click="OpenCourse()"
    v-test="{ id: 'course-card-container' }"
  >
    <img
      src=""
      alt=""
      class="card-img-top course-image"
      v-test="{ id: 'course-card-thumbnail' }"
    />

    <div class="card-body">
      <h6 v-test="{ id: 'course-card-name' }">
        {{ course.courseName }}</h6>
      <p v-test="{ id: 'course-card-shorthand' }">
        {{ course.courseShorthand }}
      </p>
    </div>

  </div>
</template>
```

The course card has three elements. All elements are equipped with a v-test directive in which we pass an id, such as: v-test=" id: 'course-card-shorthand' ".

We will create a wrapper on this component so that we have access to all the elements and their attributes.

Here is how the different libraries come together to create a working unit test:

## 6.1 Implementation

---

```
import { expect } from "chai";
import { shallowMount } from "@vue/test-utils";
import CourseCard from "@components/courseCard.vue";
import { dummyCourse } from "./courseCard.utils";

const wrapper = shallowMount(CourseCard, {
  props: {
    course: dummyCourse
  }
});

describe("Course Card", () => {
  it("Course Card should exists", () => {
    const CourseCardContainer = wrapper.find(
      "[data-test-id='course-card-container']"
    );
    expect(CourseCardContainer.exists()).to.equal(true);
  });
});
```

By the use of the Vue Test Utils library we create a constant wrapper and set it equal to the CourseCard component by using the shallowMount() function.

The function describe() and it() are provided by the Mocha library. These are the functions that run the tests.

We write what we would like to test inside the it() function.

We create a constant called CourseCardContainer and set it equal to a specific HTML element inside the CourseCard component that is stored in the wrapper object. The way we achieve this is by using the function wrapper.find(). Recall that the HTML elements in the component were passed v-test directives with a test id. This test id is passed as an argument into the wrapper.find() function so that it is able to fetch the desired HTML element. In this case, it is the course-card-container element.

The function expect() is provided by the Chai library. It allows us to make an assertion about the HTML element stored in the CourseCardContainer constant. In this case, we expect it to exist in our DOM. If the component is rendered in our view, the expect() function will return true and the test will pass, if not, it will return false and the test will fail.

We can write more advanced tests by passing data to the wrapper and test if the data is handled correctly. Looking at the shallowMount() function in the code block above, we can see that we pass it a course object. The course object is set equal to

## 6.1 Implementation

---

dummyCourse which is stored in a separate file and looks like this:

```
import { courseType } from "@store/interfaces/course";

export const dummyCourse: courseType = {
  courseId: 0,
  courseName: "Web Programming",
  courseShorthand: "DAT310",
  Teacher: 0,
  documents: [],
  courseModules: [],
  AssignmentModules: [],
  QuestionSets: []
};
```

The component will then be injected with the data from the dummyCourse created above. This allows us to create other assertions than just checking whether an element exists or not, now we can check if an element has the expected output based on the given input.

In the two tests below we check if the component has the correct output in the course-card-name and course-card-shorthand elements.

```
describe("Course Card - body", () => {
  it("Course Card should contain course name", () => {
    const CourseCardName =
      wrapper.find("[data-test-id='course-card-name']");
    expect(CourseCardName.text()).to.equal(dummyCourse.courseName);
  });

  it("Course Card should contain course shorthand", () => {
    const CourseCardShorthand = wrapper.find(
      "[data-test-id='course-card-shorthand']"
    );
    expect(CourseCardShorthand.text()).to
      .equal(dummyCourse.courseShorthand);
  });
});
```

## 6.1 Implementation

---

There are two main approaches for writing tests for an application: Black-box unit testing and White-box unit testing.

Black-box unit testing involves writing tests for a unit that does not yet exist. You first write the tests and then write your code to match with the expectations of your tests.

White-box unit testing operates in the opposite way. You first write a piece of code and then write tests for it

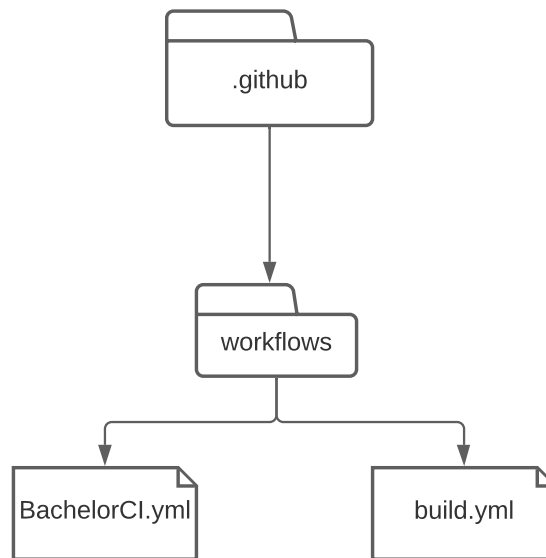
We chose the approach of White-box unit testing when writing our tests. The Black-box approach would be more fitting in a well-planned and strictly structured development model such as the Waterfall model where you have an overview of all the components that will go into your application before the development process starts. Since we worked in a more agile manner there were often things and ideas that came up under ways throughout the development process that we decided to implement. We then implemented the piece of code and then wrote tests for it afterwards.

We have over 100 unit tests that check that everything is rendered correctly.

## 6.2 Github Actions

The purpose of writing tests is to automate the process of checking that everything works as expected whenever we change the code. We would like to run the tests on every pull request, and in order to do so we need to create workflows that we automate with Github Actions.

In order to set it up, we created a `.github` folder in which we store github related files. We create a `workflows` folder in the `.github` folder and in it create files. The file structure for the github actions looks like this:



We will go through the `BachelorCI.yml` and `build.yml` files.

## 6.2 Github Actions

---

### 6.2.1 BachelorCI and Build

```
name: Bachelor CI - server and app

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  #server job
  server:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./server
    strategy:
      matrix:
        node-version: [12.x, 15.x]
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v1
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm install
      - run: npm test

  #app job
  app:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./app
    strategy:
      matrix:
        node-version: [12.x, 15.x]
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v1
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm ci
      - run: npm run test:unit
```

## 6.2 Github Actions

---

The file has two parts: `on` and `jobs`.

Most of the code is provided as a template from the official documentation of Github, we edit the code to match with our project and requirements.

The `"on"` part which is declared in the beginning of the file is used to specify when this workflow will be executed. We can specify on which action and on which branch. In our case, we want to run the tests whenever we push to the main branch and when we make a pull request to the main branch.

The `"jobs"` part is for specifying what actions should be ran upon a push or pull request to the main branch, and on what specifications the tests should be ran on. We defined two different jobs; one called `server job` and one called `app job`.

The `server job` is configured to run the tests located in the `server` folder, while the `app job` is configured to run the tests in the `app` folder. Each job has a field called `steps`. In this field we can tell Github what commands to execute to run the tests. We tell it to run the following script for the `app job`:

The tests can be run by executing the following script: **`npm run test:unit`**

The script runs the following command:

```
vue-cli-service test:unit tests/unit/**/*.spec.ts
--require tests/unit/setup.js
```

The following script is ran for the `server job`: **`npm test`**

Which runs the command:

```
mocha test/**/*.js
```

The `build.yml` file follows the exact same structure but differs slightly in two ways.

It only has one job called `app job` instead of two jobs.

The script the `build.yml` file runs is **`npm run build`** which creates a production build of our application. This is especially important for our application since we are using

## 6.2 Github Actions

---

TypeScript. As discussed earlier, typescript doesn't run in the web browser, so it will need to be compiled down to Javascript. By building the application for production each time we create a pull request to our main branch, we make sure that the main branch is always working, and that the application can be safely deployed.



## Chapter 7

# Student Monitoring and Data Visualization

This chapter will talk about a central topic in our project: The gathering of student data and its representation to the teacher.

When it comes to monitoring the performance of students in question sets the implementation was straight forward. We increment the time spent on the question set and collect data on the amount of correct answers the student has. This data is then represented in a graph to the teacher.

Monitoring students as they read through a document was not as straight forward. A practical problem which occurred when working on the gathering of student data was to determine where to collect data from. For instance, we would like to figure out how much time a student spends on a certain topic of a document, and in order to do so we need to be able to find out when a student is reading a topic and when he is not.

Our solution to this problem was to define what we call a **topic scope**, which spans from the beginning of a topic which is a header, to the end of the topic, which is the last paragraph element under that header. If the scope of a topic was within a certain area on the screen, we would interpret it as if the student is reading that topic and thus increment the time spent on that topic.

An example of this can be seen in the figure below:

## Student Monitoring and Data Visualization

The screenshot shows a web page titled "The World Wide Web" with a date of "10 May 2021". The main content area is titled "Web 1.0" and contains text about the first version of the World Wide Web. A diagram labeled "Web 1.0" shows a globe with two people icons connected by lines. Below the diagram is text about Sir Tim Berners-Lee's design and the Archie search engine. To the right, a "Topics Data" sidebar lists three topics: "Topic Web 1.0" (Time: 17, InReadZone: true), "Topic Web 2.0" (Time: 0, InReadZone: false), and "Topic Web 3.0" (Time: 0, InReadZone: false). A vertical bar on the left side of the page is divided into red, green, and red sections.

Figure 7.1: Topic Scope Example 1

The document the user is reading contains three different topics as can be seen to the right under Topics Data. Topics highlighted in blue are the topics we assume the student is reading. In this case it is Topic Web 1.0.

To the left is a red and green bar. If a topic scope is within the green zone we will increment the time spent on that topic. If it is below or above the green zone then the topic is not visible to the student in such a way that we would assume that the student is reading it, and thus we do not increment time spent on those topics.

There are flaws to this implementation. Take the figure below as an example:

## Student Monitoring and Data Visualization

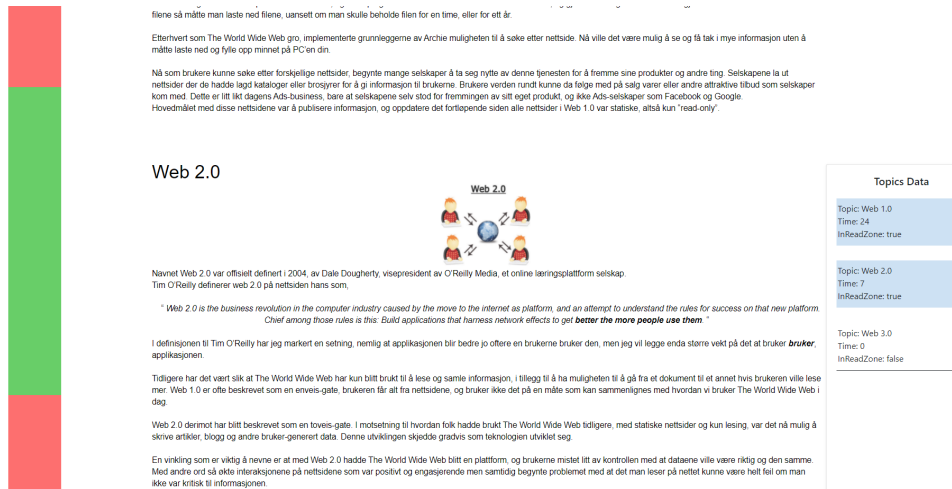


Figure 7.2: Topic Scope Example 2

In this case there are two topics within the green zone. The student might be finishing reading the last part of the first topic and then jumping to the second topic. Since the last part of the first topic is still in the green zone, time spent on that topic is still being incremented although the student might be reading the beginning of the second topic, as can be seen to the right in the Topics Data container.

An idea we had for counter-acting this flaw is perhaps shrink the green zone and have a lower opacity on text that was in the red zone. This way the student would be forced to scroll so that the text he wants to read is in full opacity which would be in the green zone. An argument against this implementation is that it might affect the user experience in a negative way.

## Student Monitoring and Data Visualization

The collected data on each topic is represented in a graph where the topic names are listed in the x-axis and the time spent on each topic is listed on the y-axis:

### Stats

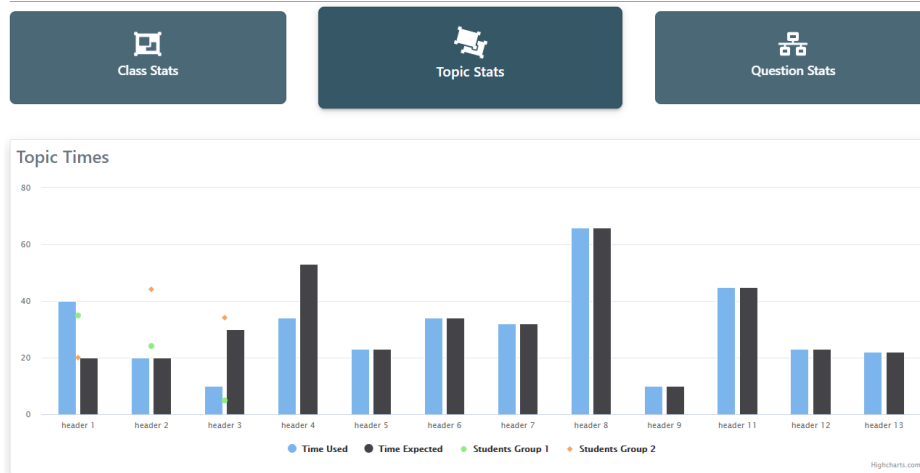
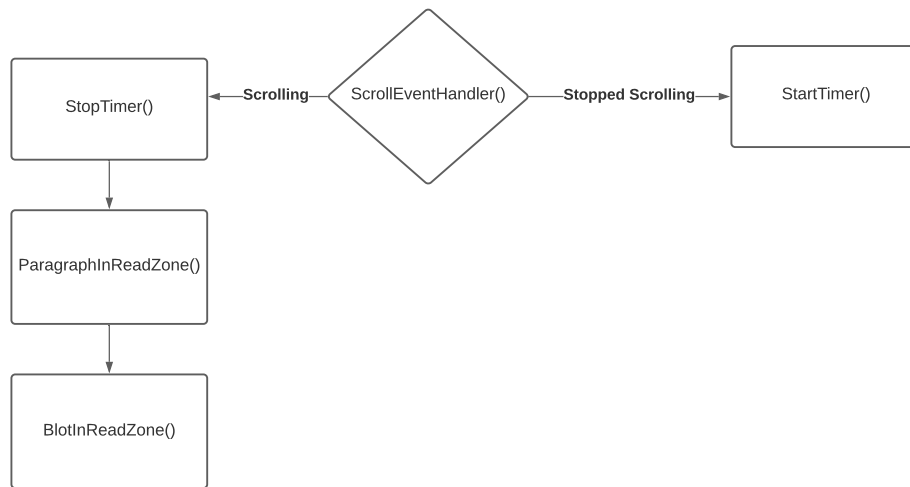


Figure 7.3: Graph Example

### 7.1 Monitoring Mechanisms

The implementation of monitoring students as they go through a document required a number of connected functions. An overview and brief walk through of them is seen below:



**Figure 7.4:** Monitoring Functions

When a user is scrolling, we call on a function called `StopTimer()` which stops incrementing the time value of a topic and triggers the function `ParagraphInReadZone()` by emitting a custom scroll event. The function `ParagraphInReadZone()` is bound to every paragraph in the editor and retrieves the coordinates of paragraph elements and passes them into the function `BlotInReadZone()` and executes it. The function `BlotInReadZone()` checks if a paragraph is within the topic scope as the user scrolls. If it is, it sets the value of an attribute that each topic has that is called `InReadZone` to true.

When a user stops scrolling we execute the function `StartTimer()`. The function loops through a list of topics and checks which topics have the value `InReadZone` set to true and begins to increment the `Time` value of those topics.

## 7.1 Monitoring Mechanisms

---

### 7.1.1 Implementation of Monitoring Mechanisms

This section will walk through the implementation of the different mechanisms that allow for collecting data as the user interacts with a course.

The bar to the left that sets boundaries for what we call a "reading zone" was created by creating three div elements and giving them classes and id's which we used to set their height in vh using css:

```
<template>
  <div class="topic-readzones">
    <div id="topic-non-readzone-1"></div>
    <div id="topic-readzone"></div>
    <div id="topic-non-readzone-2"></div>
  </div>
</template>

<style scoped>
.topic-readzones{
  position: fixed;
  left: 4%;
  top: 0;
  height: 100vh;
  width: 100px;
  opacity: 0;
}

#topic-non-readzone-1, #topic-non-readzone-2{
  height: 20vh;
  background: #fd6f6f;
  width: 100%;
}

#topic-readzone {
  height: 60vh;
  background: #68ce68;
  width: 100%;
}
</style>
```

Note that the opacity of the div elements is set to 0 so that the user will not be able to see them when reading through documents.

## 7.1 Monitoring Mechanisms

---

The next step is to find the position of the different HTML elements so that we can store the positional values of a topic scope and match them up against the positional values of the reading zone. In order to do this we used the function `element.getBoundingClientRect()`. It returns a `DOMRect` object which contains these properties:

- **`DOMRect.x`** returns the x coordinate of the element
- **`DOMRect.y`** returns the y coordinate of the element
- **`DOMRect.top`** returns the top coordinate value of the element
- **`DOMRect.right`** returns the right coordinate of the element
- **`DOMRect.bottom`** returns the bottom coordinate of the element
- **`DOMRect.left`** returns the left coordinate of the element
- **`DOMRect.width`** returns the width of the element
- **`DOMRect.height`** returns the height of the element

The ones that we made use of was the `DOMRect.top` and `DOMRect.bottom` properties in order to retrieve values for our reading zone. We also used it to determine the position of the data in a certain topic. If the data in a topic was within our desired range, we would start a counter that would increment the time spent on that topic. The code is as follows:

First we defined variables to hold the values of the position of our reading zone:

```
this.readZone = document.getElementById("topic-readzone");  
this.readZoneClientRect = this.readZone!.getBoundingClientRect();
```

`readZone` is set equal to the element with the id "topic-readzone" which is the green div that can be seen on the left of figure 7.1 and figure 7.2.

## 7.1 Monitoring Mechanisms

---

We then created a function which checks if a Blot is in the range of the reading zone:

```
public BlotInReadZone(blotRect: DOMRect) : boolean {
  return (
    blotRect.top >= this.readZoneClientRect.top &&
    blotRect.bottom <= this.readZoneClientRect.bottom
  )
}
```

The function returns true if the coordinates of the element passed in as an argument is within our reading zone. In Chapter 5 Section 5.6.1 we showcased how we can customize the way our editor handles HTML elements, such as passing custom attributes to elements.

We also customized the way Blots handled paragraphs. We created a file called paragraphBlot.ts and following the same steps as shown in Section 5.6.1 we added additional functionality to how paragraphs are handled:

```
private ParagraphInReadZone() {
  const ParagraphRect = this.domNode.getBoundingClientRect();
  if (this.quill) {
    if (this.quill.BlotInReadZone(ParagraphRect)) {
      const TopicID = this.domNode.getAttribute(TopicAttributeName);
      this.quill.EnableTopicUpdating(TopicID);
      (this.domNode as Element).classList.add("in-readzone")
    }
    else {
      (this.domNode as Element).classList.remove("in-readzone")
    }
  }
}
```

We use the function `this.domNode.getBoundingClientRect()` in order to retrieve the positioning properties of a paragraph element. This paragraph element is passed in as an argument in the `BlotInReadZone()` function we had created earlier. The function determines if the paragraph is within the reading zone.

If it is, we do two things:  
we call on the function `EnableTopicUpdating()` and pass it the id of the topic and we



## 7.1 Monitoring Mechanisms

---

give the paragraph a class attribute of name "in-readzone". This class attribute leaves us with the possibility of styling paragraph elements within the reading zone differently than the ones that are outside of it.

Below is the function EnableTopicUpdating():

```
public EnableTopicUpdating(topicID: string) {
  if (topicID in this.TopicData) {
    this.TopicData[topicID].InReadZone = true;
  }
}
```

All it does is set the value of the boolean InReadZone to true. When it is true, we can start counting the time spent on that topic.

The function ParagraphInReadZone() is bound to a scroll event, so whenever a user stops scrolling the function is executed and checks which paragraphs are within the topic scope. Details on how we bind functions to events will follow in the next paragraphs.

We have two functions that handle the time counting: StartTimer() and StopTimer(). These functions are executed on specific conditions which is handled by a function called ScrollEventHandler(). The function ScrollEventHandler() is bound to scroll events in this manner:

```
window.addEventListener("scroll",
  this.ScrollEventHandler.bind(this))
```

We bind the function ScrollEventHandler() to the scroll event listener, so that whenever the user scrolls in a document, the function ScrollEventHandler() is executed.

Below is a walkthrough of the function ScrollEventHandler():

## 7.1 Monitoring Mechanisms

---

```
private ScrollEventHandler() {
  if (this.ScrollTimeout !== -1) clearTimeout(this.ScrollTimeout);
  this.StopTimer()
  this.ScrollTimeout = window.setTimeout(() => {
    this.StartTimer()
  }, 500)
}
```

What this function does is to stop the timer by executing the function `StopTimer()` when the user is scrolling and reset our variable called `ScrollTimeout`. The variable checks to see how long it has been since the last time the user scrolled.

If the time since last scroll exceeds 500 milliseconds we assume that the user has stopped scrolling and execute the function `StartTimer()`:

```
public StartTimer() {

  this.root.dispatchEvent(new CustomEvent("scroll-started"));

  this.Time = 0
  this.Timer = setInterval(() => {
    ++this.Time;
    for (const TopicID in this.TopicData) {
      if (Object.prototype.hasOwnProperty.call(this.TopicData,
        TopicID)) {

        if (this.TopicData[TopicID].InReadZone) {
          ++this.TopicData[TopicID].Time;
        }
      }
    }

    if (this.Time > 300) {
      clearInterval(this.Timer);
      this.StopTimer()
    }
  }, 1000)
}
```

## 7.2 Visualizing Data

---

This function loops through a list called TopicData. The list contains topics which all have a boolean attribute called InReadZone. For every topic we check if the value of InReadZone is true. If it is, we increment the value of the Time attribute of that topic.

At the end of the function we have an if-statement that we put as a measure against inactivity. If the user has not scrolled for more than five minutes we assume that the user is inactive and stop the timer until the next scroll event. The next step is to visualize the data now that we are able to monitor and collect data from students.

## 7.2 Visualizing Data

This section will briefly cover how we process and present our collected data.

The data collected from monitoring the time spent on the different parts of a document is visualized in a stacked bar graph. Below is a demonstration of this graph:

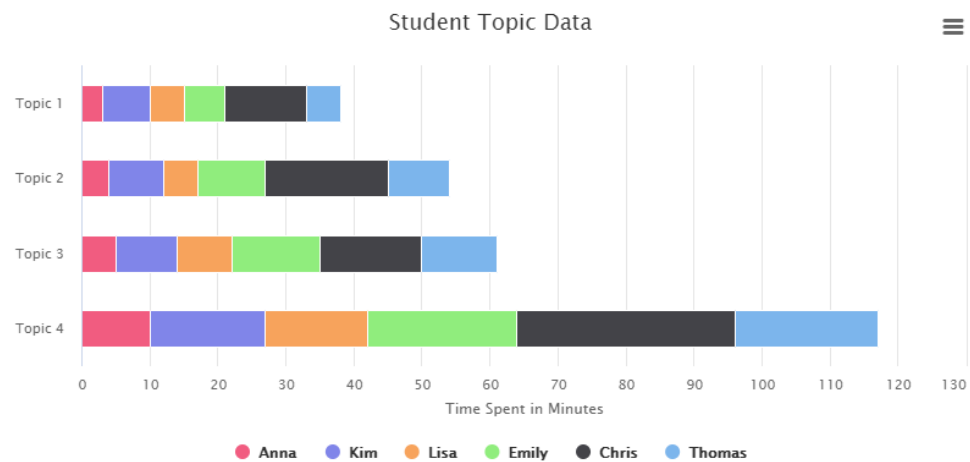


Figure 7.5: Stacked Bar Graph

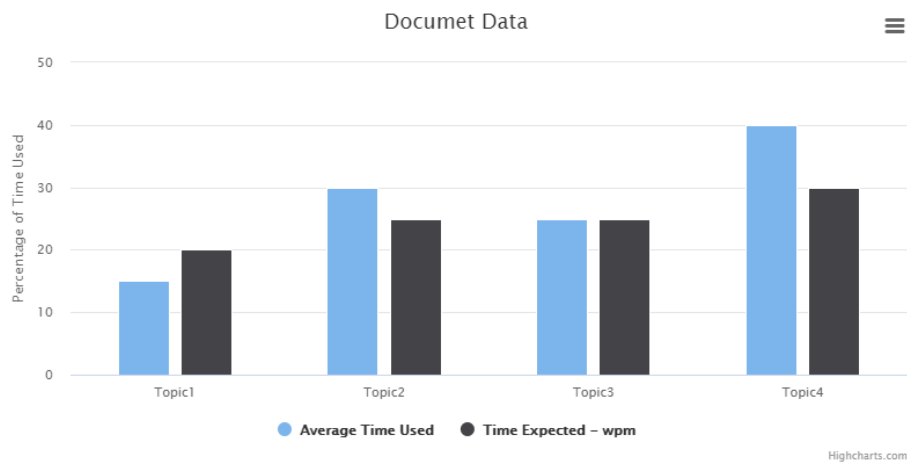
The graph provides insight in how much time each student spends on a certain topic. We can tell that the student Anna goes through the topics very quickly, whereas the student Chris is spending a lot more time on each topic than the others. The conclusion that can be drawn by the teacher is that the student Chris might be in need of assistance in the course.

## 7.2 Visualizing Data

---

Additionally, we can observe that the students are allocating a large amount of time on topic 4, while topic 1 for most students does not require much time. This can signalize to the teacher that the content in topic 4 might be hard for the students to understand whilst the content in topic 1 is easier to grasp.

However, students might be spending more time on a certain topic due to the sheer fact that it might be longer than others and not necessarily harder to digest. We therefore provide an additional graph to compliment the graph in figure 7.5. The additional graph displays the time expected on a topic versus the average time used on that topic.



**Figure 7.6:** Class Stats Graph

The expected time used on a topic is calculated by taking the total amount of words in a topic divided by 200 which is equal to the average word per minute read by an adult. The expected time for all topics in a document are added up to a total. We then calculate how many percentages of that total each topic covers.

The calculation of time used follows a similar process. We add the total time used on all topics for each student to get a total time used. Then we calculate how many percentages each topic account for of that total. For instance, a student might spend 20 percent of their time reading a document on topic 1, and 40 percent on topic 2.

The data for time used on a topic is matched up against the expected time used for that same topic. This provides additional data for the teacher to support claims made by analysing the graph in figure 7.5 which projects the allocated time on a topic by each individual student.

## Chapter 8

# Application Evaluation

For evaluating our application we decided to create a survey with questions of interest to us as developers.

We then gathered a group of testers and gave them instructions on how to test the application and afterwards asked them to fill out the survey based on their user experience.

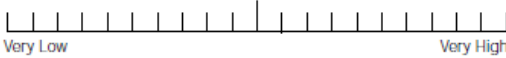
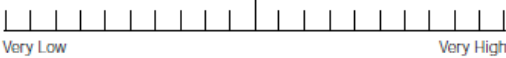

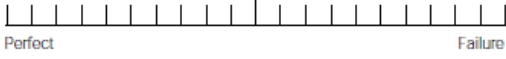
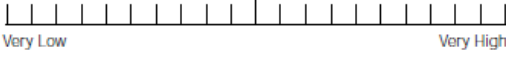
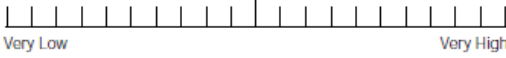
The form created was inspired by Nasas Task Load Index which looks like this:

## Application Evaluation

---

### **NASA Task Load Index**

*Hart and Staveland's NASA Task Load Index (TLX) method assesses work load on five 7-point scales. Increments of high, medium and low estimates for each point result in 21 gradations on the scales.*

Name	Task	Date
Mental Demand      How mentally demanding was the task?		
		
Physical Demand      How physically demanding was the task?		
		
Temporal Demand      How hurried or rushed was the pace of the task?		
		
Performance      How successful were you in accomplishing what you were asked to do?		
		
Effort      How hard did you have to work to accomplish your level of performance?		
		
Frustration      How insecure, discouraged, irritated, stressed, and annoyed were you?		
		

**Figure 8.1:** Nasa Task Load Index

The survey sticks to asking "How" questions that can be answered by providing a value within a specified range, usually defined as from "Very Low" to "Very High".

## 8.1 Google Form Survey

---

### 8.1 Google Form Survey

We created our survey by the help of Google Form Survey.

It consists of 11 questions; 8 of them are formatted in a linear scale and three of them are of the type long answer in case the testers would like to provide additional information.

The first eight questions are:

- How hard was it to navigate the application?
- How hard was it to create a document?
- How hard was it to create a question set?
- How hard was it to submit the answers to a question set?
- How mentally demanding was it to navigate between course modules?
- How physically demanding was it to create a user?
- How hard was it to create a course?
- How useful were the graphs to you as a teacher?

They are answered by providing a value from 1 to 5 where 1 signalizes "Very easy" and 5 signalizes "Very hard" with the last question being an exception where the value 1 signalizes "Very useful" and 5 signalizes "Not useful"

The last two questions are optional as they are only relevant for testers that were assigned to test as a Teacher account, while the others are relevant for all users and therefore required.

The last three questions are the following:

- Is there any negative feedback you would like to include?
- Is there any positive feedback you would like to include?
- Are there any suggestions for improvements to the application?

These are optional and answered by providing a text input.

## 8.2 Feedback and Assessment

---

### 8.2 Feedback and Assessment

The feedback data from the testers were plotted in a stacked bar graph in order to get a better overview over the results:

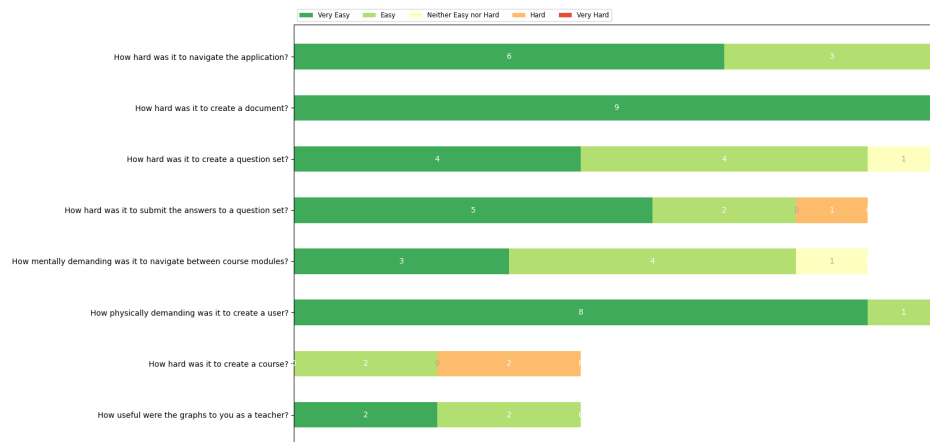


Figure 8.2: Survey Data

The answers for the remaining three questions were generic and neutral. We had no negative feedback and most of the positive feedback had to do with the design of the application, it was by many described as simplistic and clean.

Suggestions for improvements were to add a navigation arrow in the application to go back and forth from previous routes instead of navigating through the browser, and have some modals such as the modal for joining a course to automatically close by itself after clicking the join button.

A teacher also requested a graph in which the score of a student on a test was graphed together with the time spent reading the course documents to easier look at the correlation between test scores and time used.

We had 9 test users. Most of the feedback was positive, with a majority finding the application appealing to the eye and easy to use and navigate through. What the testers had the most negative experience with was creating a course.

There were a couple of bugs that were reported:



## 8.2 Feedback and Assessment

---

- **Registration**

There were times when user registration was not working as intended for all users. After looking into it we realized it had to do with how TypeForm handled requests. TypeForm processes 25 requests at a time, in cases where the number of requests exceeded that limit the registration would not work properly. The fix to this was to clear all requests to TypeForm.

- **Saving Documents**

A bug was discovered where if certain commands were written in a document it would not save. When creating documents about Python we were writing about If, Elif and Else statements. Examples which included the elif Python statement would cause the document to not save. We are yet to find a fix to this bug.

- **Questions**

A rare bug that we had only stumbled upon once during the development phase and were unable to recreate was rediscovered during the testing process. Some true or false questions would have their answer options swapped. The option False was located in the place of where the option True should have been and vice versa.

The application was loading slower than expected upon logging in. The course page and images had a noticeable loading time. Our web application is hosted on the university server. We are unsure of whether the delay is caused by the server or the application itself, but it was clear that there are improvements to be made in regards to optimizing the app.

## Chapter 9

# Conclusion

### 9.1 Main Contributions

The goal of this project was to create a web application that will serve as a smart learning management system for teachers and students. The main concept of this system was to be able to automatically monitor and collect data on the performance of a student in sections of a course. This data would then be presented to a teacher to give the teacher a greater insight on which parts of the course the class in general find the most challenging and where each student struggles the most.

Our main goal was dependent of many other pieces falling into place. In order to achieve our objective, we had to build a whole platform in which teachers can create course material that students can interact with. Only after the learning management system was in place were we able to work on our goal and start monitoring and collecting data of students. This included the creation of a database, setup of a server, and a web application with a complete user interface.

### 9.2 Moving Forward

This was a fairly large and ambitious project for a team of two members. Therefore, there were a few functionalities and ideas that we wanted to implement but didn't have the time to. This leaves many doors open in regards to continuation of this project.

## 9.2 Moving Forward

---

Referring to figure 1.1 in Chapter 1, Section 1, we had created a table of different functionalities that we had planned to implement. Looking back at it now we can take a look at what we managed to implement and what was left out:

Priority	Teacher	Student	General
P0 (v0.1)	Create Course Assign syllabus questions and other resources to course.	Join and interact with course materia	Role-based login system Dashboard (minimal)
P0 (v0.2)	Visualize student data	Collect data from students taking a course Visualize personal student data	Dashboard (extended)
P1	Add questions to specific topics	Student notes Inactivity detection	Nice UI responsive design Security Search for resources
P2		Share Notes	Notification System Bug Fixing

Figure 9.1: Colour Coded Priority Table

The colours represent the following:

- **Green:** Implemented
- **Yellow:** Partially Implemented
- **Red:** Not Implemented

We managed to implement the functions of the highest priority, but didn't manage to implement all the functions that we had in mind.

Here are some key features that we would like to continue to develop:

### Security

Security is a vital part of every application. We did not give it the attention and time

## 9.2 Moving Forward

---

that we would like and therefore there is a lot of room for improvement in this area. The way we would secure the application further would be through protection against cross-site scripting and applying a WAF (Web Application Firewall). We would also have to do more research on protecting our application from SQL-injections although Sequelize does offer built-in protection against it.

### **Adding Questions To Specific Topics**

An important feature that would aid us in achieving our main goal in a more sophisticated manner would be to have the possibility of adding a question set to a specific topic in a document. Apart from being able to measure time on specific topics, we would be able to couple that with the score on the questions to the same topic. This would give more detailed data that we could work with and present to the teacher.

**More research on statistics and graphs** The data we collect is presented in a simple manner. We feel that the data can be processed in a better way through statistical analysis and perhaps also presented in other various ways.

### **Responsiveness**

Our application was not developed with mobile-friendliness in mind. It is best suited for PC usage at this stage. Mobile phones have become an integral part of many peoples daily lives, thus it would be a sought-out feature by users. It would make the application more accessible and attractive.

### **Notification System**

We had a plan on implementing a notification system which would allow teachers to send notifications to students about upcoming events. This would be a nice feature to implement in the future.

### **Sharing System**

A nice feature that we had in mind was to allow students to share notes and question sets with each other.

**Bug Fixing** A lot of bugs were discovered very late in the development process due to our application not reaching a state in which it could be tested until late into the project. Although we were able to fix some bugs, there still remain a few bugs that need to be addressed.

### 9.3 Notable Lessons Learnt

#### 9.3.1 Best Practices and File structure

Since we both had not worked on a project of this scale before, we hadn't had much experience with good coding practices and clean code principles.

When we started off with our project we didn't keep these practices and principles in mind and tackled the project as we would any other small project. We quickly came to the realization that our initial approach would not be optimal if we were to continue down the path we had chosen.

Midways through the development process we took a couple days off from writing more code and started researching and learning more about good coding practices and optimal code structure which we then implemented. We noticed a big difference when it came to code readability and the code base became easier to work with and modify.

#### 9.3.2 TypeScript

In prior to starting on this project, we had never had any experience with TypeScript. The discussion of whether to make use of this technology or not arose in the beginning when we researched what technologies would fit our projects needs. A lot of articles online spoke highly of TypeScript in applications of larger sizes, and we decided to take our chances and go for it.

It became a decision we both are thankful for. TypeScript has proven to be a very useful tool that we both will most definitely make more use of in the future. Although its modular approach to coding and type definitions make for a more strict coding style, it helps a great deal with ensuring good project structure which is much needed for larger applications.

## 9.4 Reflection

There was a lot to take in from this project. It was a challenging and intriguing process that took us into the realms of both the knowns and unknowns. There were a lot of technologies that we encountered for the first time, such as TypeScript, Node.js and many other libraries. But still the project had elements from a lot of different courses that we have had in our study plan, such as web programming, databases, data security and even statistics, which made it a fun and interesting experience.

We are grateful for having been given the opportunity to work on this project as it has let us put our knowledge and skills to test, and proud of what we have achieved in the past few months, being able to build a prototype which demonstrates what we are trying to achieve.

The process of going from an idea to an actual product has been extremely enjoyable and instructive. We worked with a lot of flexibility which ultimately was a double-edged sword as the amount of flexibility gave us a lot of freedom to experiment and implement new features that we hadn't planned for earlier, but at the same time it could be a draw-back in the sense of drawing our attention and time away from the main goal.

Having been through the process of building an application from scratch has definitely given us a feeling of empowerment and a confidence boost. We've proved to ourselves that we are capable of putting ideas and sketches into realization. The knowledge and experience gained from this project will come to great help in tackling daunting challenges in the future.

# Bibliography

- [1] Axios. <https://github.com/axios/axios>.  
[Accessed April 2021].
- [2] Bootstrap. <https://getbootstrap.com/>.  
[Accessed February 2021].
- [3] Chai. <https://www.chaijs.com/>.  
[Accessed February 2021].
- [4] Download typescript. <https://www.typescriptlang.org/download>.  
[Accessed January 2021].
- [5] Downloads. <https://nodejs.org/en/download/>.  
[Accessed January 2021].
- [6] Google survey form. <https://www.google.com/forms/about/>.  
[Accessed May 2021].
- [7] Highcharts demos. <https://www.highcharts.com/demo>.  
[Accessed April 2021].
- [8] How to customize quill.  
<https://quilljs.com/guides/how-to-customize-quill/>.  
[Accessed March 2021].
- [9] Introduction to json web tokens. <https://jwt.io/introduction>.  
[Accessed February 2021].
- [10] Migrations. <https://sequelize.org/master/manual/migrations.html>.  
[Accessed January 2021].
- [11] Mocha. <https://mochajs.org/>.  
[Accessed February 2021].

## BIBLIOGRAPHY

---

- [12] Model basics. <https://sequelize.org/master/manual/model-basics.html>.  
[Accessed February 2021].
- [13] Nasa task load index. <https://en.wikipedia.org/wiki/NASA-TLX#/media/File:NasaTLX.png>.  
[Accessed May 2021].
- [14] Quill configuration. <https://quilljs.com/docs/configuration/>.  
[Accessed March 2021].
- [15] Quill parchments. <https://github.com/quilljs/parchment>.  
[Accessed March 2021].
- [16] Quill quickstart. <https://quilljs.com/docs/quickstart/>.  
[Accessed March 2021].
- [17] Typeform. <https://vue-test-utils.vuejs.org/>.  
[Accessed March 2021].
- [18] Vue highcharts installation. <https://www.npmjs.com/package/vue-highcharts>.  
[Accessed April 2021].
- [19] Vue installation. <https://vuejs.org/v2/guide/installation.html>.  
[Accessed January 2021].
- [20] Vue test utils. <https://vue-test-utils.vuejs.org/>.  
[Accessed February 2021].



## Appendix A

# Setup Instructions

## Node.Js Installation

1. Go to <https://nodejs.org/en/download/>
2. Click on either the Windows or iOS installer depending on your IO
3. Run the downloaded software and follow the instructions
4. Go to your command-line and type the command `node -v` and `npm -v` to check the installed version.

## Vue.Js Installation

1. Open a command line interface
2. Run the command `"npm install vue"`
3. Run the command `"vue -V"` to verify and check the installed version

## Vue.Js Project Setup

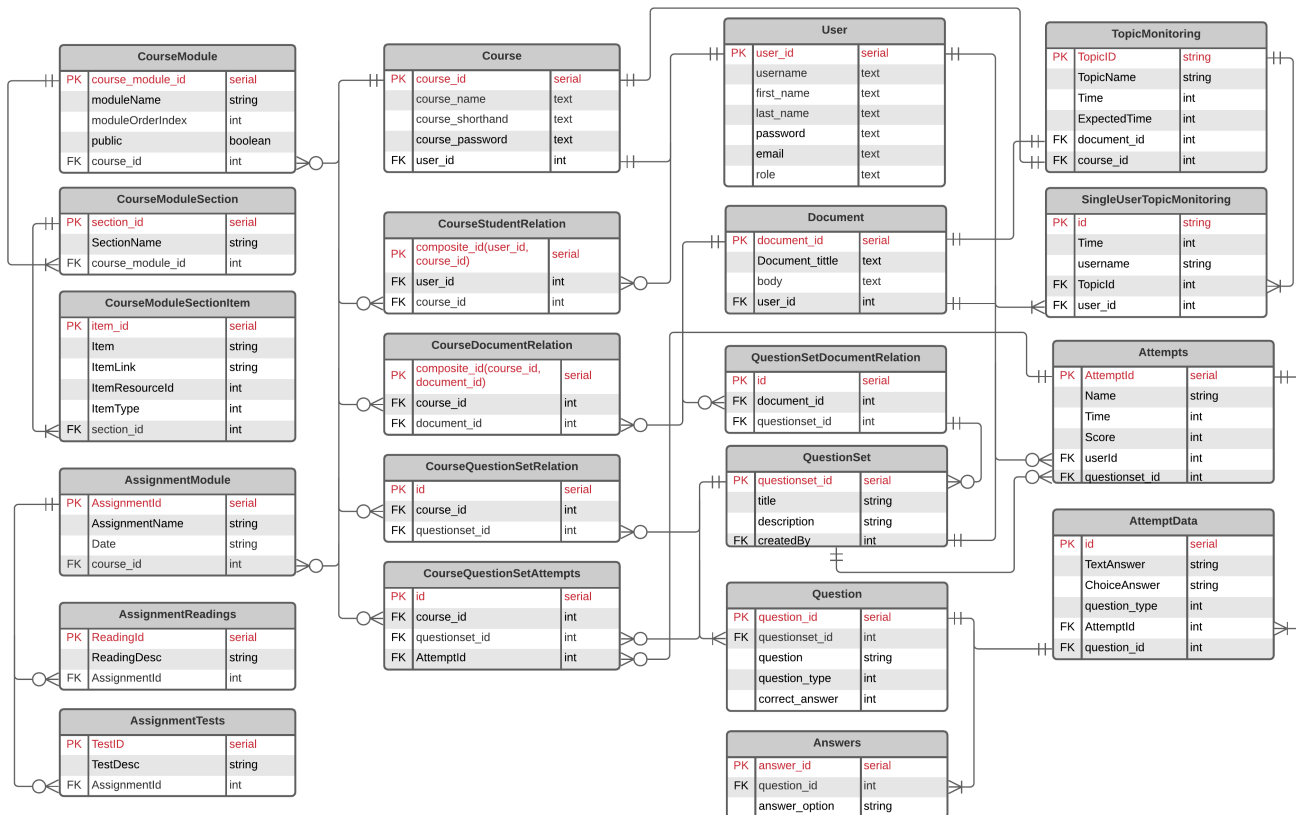
1. Create a folder with desired project name
2. Open the folder in your preferred IDE
3. Open a command-line interface and `cd` into your project folder
4. Run the command `"vue create <application name>"`
5. `cd` into the generated `<application name>` folder
6. Run the command `"npm run serve"` to start your vue application



## Database Diagram

# Appendix B

# Database Diagram





## Appendix C

# Components and Views

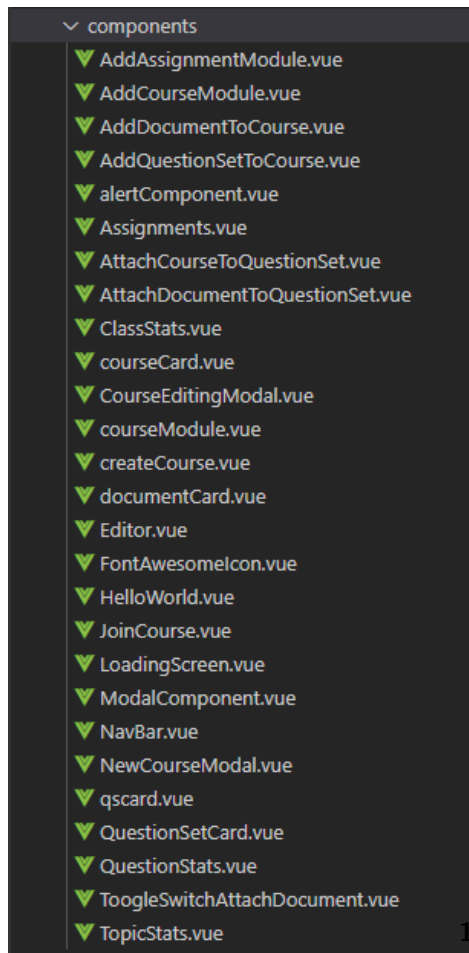


Figure C.1: Component Files

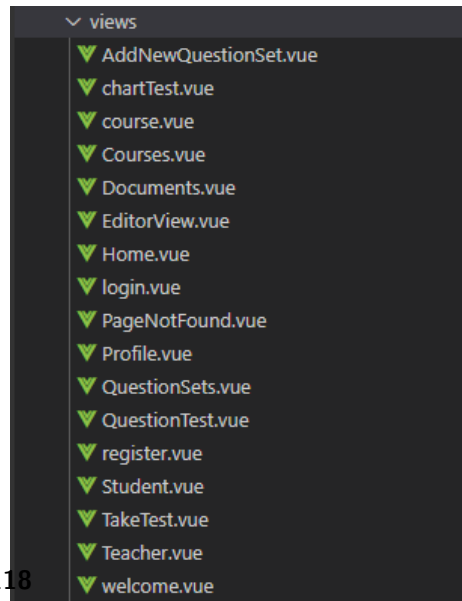
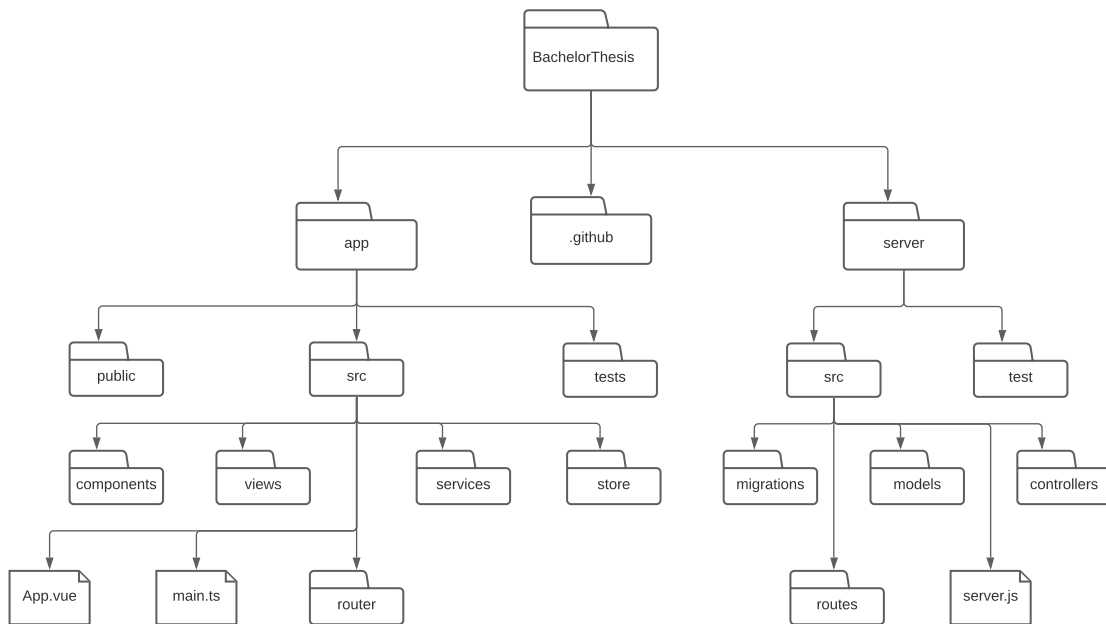


Figure C.2: View Files



## Appendix D

# Directory Structure



Appendix E

Datablad