Developing a content management system with drag and drop functionality

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor thesis

Degree Programme in Business Information Technology
or Computer Applications
Fall, 2022

Fiona Neumann

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology Abstract

Author Fiona Neumann Year 20222

Subject Developing a content management system with drag and drop functionality

Supervisors Lasse Seppänen, Elina Hietaranta

ABSTRACT

This thesis was written in corporation with Financial Labs Oy. Financial Labs Oy is a software development company with offices in Oulu, Helsinki and New York and was founded in 2017. The purpose of this thesis was to develop a feature for an existing Intranet, which would be the dashboard landing page of the Intranet and comes with an editor to give certain users the possibility to create posts for the dashboard. The research questions were:

1. is it possible to develop a content management system that furthermore gives the user the possibility to rearrange and deactivate posts as needed?

2. How can user rights be handled in the frontend application to give certain people the right to access editing pages and make a functionality for reactions to posts created via the content management system?

The implementation was done with TypeScript using the React and Node.js framework. Additional technologies used include TypeORM, MySQL and Amazon Web Services.

The thesis is practice based and the author made an implementation plan and developed the feature. The result of this thesis is a functional feature with possibilities for further development. The outcome of this thesis can be seen as a guide for developers who want to achieve something similar.

Keywords React, Node.js, Drag and Drop, TypeORM

Pages 38 pages and appendices 1 pages

Glossary

HTML	HyperText Markup Language for web pages
npm	Node Package Manager
CSS	Cascading Style Sheets
CRUD	Create Read Update Delete

Contents

Figures and program codes

Annexes

1 Introduction

This thesis was made in corporation with Financial Labs Oy. The company had started developing their own Intranet in 2021 and after a break the development continued in 2022. One of my tasks was to develop a Dashboard page which was required to include a dedicated editor associated with it.

This required Full-Stack development with React and Node.js and planning the infrastructure of the application, the needed database tables, API endpoints and features in the frontend. Some of the previously created parts of the application needed to be used for the project, such as for example user account and authentication.

The process therefore included making a development plan, adjusting to changes during the process, actually developing the features and writing this thesis. The research questions are as follows:

How is it possible to develop a content management system that furthermore gives the user the possibility to rearrange and deactivate posts as needed?

How can user rights be handled in the frontend application to give certain people the right to access editing pages and make a functionality for reactions to posts created via the content management system?

The result of the thesis was a functional feature that might still be developed further in the future. Suggestions for the feature will be discussed in the results section of this thesis.

2 Need for the thesis

The company started developing their own Intranet in summer 2021, but it is not yet in use. At the time being, they are using Atlassian's confluence to document important information about the workplace and news are usually spread through their slack channels.

The functionality developed so far includes user accounts and profile management done with AWS Cognito. There is a page about projects that are currently being worked on including revenue and costs and the amount of time spent working on it and a Teams page showing who is currently assigned to which Team.

Discussions with co-workers revealed that there were certain wishes about what should be included in the Intranet. The most significant one was a staff directory where it is possible to look up people and get an idea on what they are working on. One of the benefits of having such directory is having an easier time to find a co-worker with a certain skillset if running into a problem during the development process and in need of guidance.

A centralized area for finding news and documents was the next aspect which is the topic of this thesis. This could eventually be used to replace confluence entirely. A designer had created a prototype for this dashboard in Figma, a tool for designing interfaces, and during the development process it became obvious that the company would appreciate to have a dashboard editor associated with it. The main goal with this was that anybody in the company, regardless of their technological background can create, edit and delete posts on the dashboard. They should also be able to influence the design and arrange the page in a way they desire. It became clear that developing such a feature is quite a challenging and complex task and requires understanding the concept of the application thoroughly – and this is how the idea of ultimately turning it into a thesis came to be.

3 Full-Stack Development Theory

Creating an Intranet and developing the required additional features discussed in this thesis, it was necessary to setup a Full-Stack Application environment. This chapter will have a closer look into the technologies chosen to accomplish this and describe them to a certain extent to improve the readers' capability to understand the practical part of this thesis.

3.1 JavaScript

Javascript is a scripting programming language originating from Brendan Eich who wrote the initial version of Javascript back in 1995. He is also the founder of the software community Mozilla (Knorr, 2018). JavaScript "is a lightweight, intepreted, or just-in-time compiled programming language with first-class functions" (JavaScript, n.d.).

In its most plain form, JavaScript can be implemented in any HTML website. As the official Documentation explains, HTML is only the markup language, CSS is used for creating styling rules and JavaScript can be used to update content dynamically and do everything that requires higher logic such as implementing functions. (What Is JavaScript?, n.d.)

Popular tools and frameworks using JavaScript include React, Vue, Angular and the Express Web Framework using Node.js. According to Stackoverflow's developer survey in 2022, 67,9% of professional developers use JavaScript as one of their programming languages, making it the most used one at the time being. Notably, TypeScript also took up around 40%, which is built upon JavaScript. (Stack Overflow Developer Survey 2022, n.d.)

In the most current version, the most notable features include let and const variables. Let variables cannot be redefined, arrow functions that can make use of callbacks, the possibility for asynchronous requests and maps instead of normal for loops which increase readability of the code. (Crowder, 2022, p. 4-5)

3.1.1 TypeScript

TypeScript extends JavaScript and introduces a syntax for types. By default, JavaScript does not require the type of a variable to be specified. TypeScript therefore aims at avoiding errors in the code caused by the usage of the wrong type or interface. It also allows the user to specify the way an object is structured in advance. (Figure 1)

Figure 1 TypeScript Interface example (TypeScript, n.d.)

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

In the example of Figure 2 x and y can only be assigned numbers (TypeScript, n.d.). If the type is changed during the development process, the user will receive an error and it is not possible to proceed. A variable can also accept multiple types or be inferred from usage. A variable of an interface can also be made optional by adding a question mark. If we changed interface Point to x?: number the usage of variable x would be optional and if it does not exist will be ignored. It is also possible to assign any to a variable – which means that the type can be anything and be changed during its lifecycle- theoretically making the use of TypeScript obsolete. An OR operator can also be used in certain circumstances – for example we might want the variable to be a string but know that it will be undefined when starting the application. In this case it is possible to assign the type string | undefined (TypeScript, n.d.).

It therefore gives the developer useful functionality during the development process, as it is possible to define complex interfaces and verify that the information is correct. For example, when fetching data, the developer can verify that it is actually in the exact same format they want it to be.

TypeScript code will often be transpiled back to JavaScript code for built versions and is thus mainly a tool during the development process. It makes it easier to avoid errors caused by continuing with a variable or object in a slightly different type.

3.1.2 React

React is a library built on JavaScript that allows building user interfaces. It is declarative and component based and open source, created by Facebook. For simplicity, this chapter will only handle React used with hooks and not with class-based components. One of the most notable features of React is the possibility to create components that can be reused. State and Props which are commonly used in React furthermore simplify how data can be stored and handled. (React Tutorial, n.d.) Through the Node Package Manager React a new React app can be initialized – if desired also with TypeScript instead of plain JavaScript.

The official documentation provides a clear example of how props can be used for building individual reusable components. In Figure 2, the function Welcome is used as a reusable component. In this example it is used in the same file, but it could also be imported from another React / JavaScript file. Using this component in another file allows the user to change content depending on where the component is used. In this example, the name could come from the backend and be changed according to who is currently signed in (React Tutorial, n.d.).

Figure 2 Example of a component using props (React, n.d.)

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

The example above also gives us a hint on another important aspect of React. The Welcome components being returned need to be in an enclosing div. In a way one could say that no part of

the code can be “left open”. We can solve this either by wrapping our contents into divs or alternatively use <React.Fragment> </React.Fragment> or its shorter form <> </>. The advantage is that it does not add extra node to the DOM (Document Object model), making it slightly more efficient when thinking about the performance of our application.

Since version 16.8 released, Hooks were introduced to React. The standard look of a Hook is as follows: const [amount, setAmount] = useState(). Hooks are one of the most important features in React as they make it possible to listen to changes and update content accordingly. Hooks can originally be empty, but also initialized with a value. Hooks allow content to be updated dynamically without refreshing the entire page. They can be changed multiple times during their life cycle and be initialized with a default value. (Hooks at a Glance – React, n.d.)

A very useful feature is the useEffect() Hook. It is a function that can be called either initially when loading a component or based on a state change. The example below shows a basic example of using this hook. We have a state object called count (const [count, setCount] = useState(0)). Whenever there is a change on this state object in our program by using setCount, for example we add 1, we will see a console log. If we leave the brackets in the end empty, the code will only run once as soon as the component mounts. (React Tutorial, n.d.)

```
useEffect(() => {
  console.log(“hello world”)
}, [count]);
```

useEffect() hooks are often used when fetching data from an API, as we need to retrieve the data as soon as our page loaded and before content gets displayed. To achieve this, conditional rendering is often used which makes it possible not to render content until the data was fetched.

In the past, global states were often handled with React redux, which can be installed as a npm package. Nowadays, React also provides an internal feature for this called Context. Some data that could be stored in the Context could include the theme of the website (dark or light mode), the currently authenticated user or language of the program. (Context – React, n.d.)

Conditional rendering is commonly used in React applications. We can basically decide which content to render depending on a Boolean or any if statement created by us. Figure 3 Conditional Rendering shows us an example of how this can be done. We receive a Boolean with the value

false to a Greeting component we created. If the user is logged in, we render a UserGreeting component to the screen, if he is not logged in – as shown in this example – a component called GuestGreeting will be rendered. (React, n.d.)

Figure 3 Conditional Rendering (React, n.d.)

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Try changing to isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

It is however also possible to use inline rendering as seen in Figure 4. We can therefore have a component and decide to render content it contains depending on our decided condition. In this example, we look whether our messages are more than 0. If there are, we print the message to the screen. We need to look for more than zero as the index of Arrays starts at 0 and not at 1.

Figure 4 Inline Conditional Rendering (React, n.d.)

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

Conditional rendering is thus one of the most use features in any React Application.

3.2 Amazon Web Services

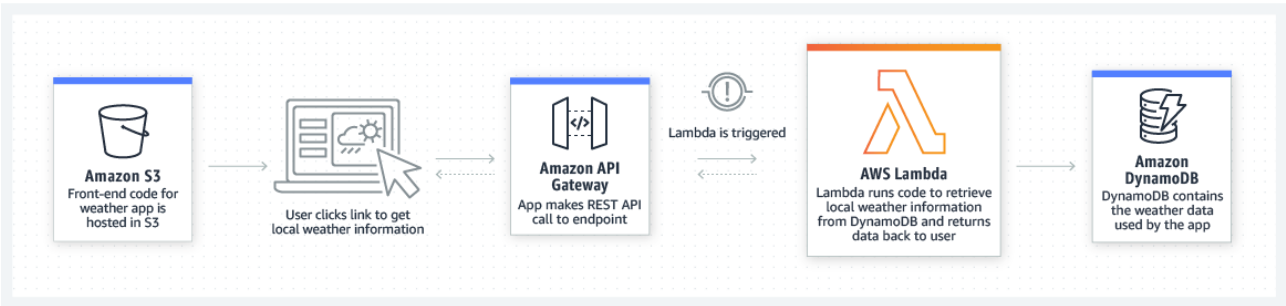
Amazon Web Services provides cloud computing platforms and APIS to their customers. Cloud Computing includes services such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

Infrastructure as a Service "provides a capability for users to provision processing, storage, and network resources on demand." (Sarkar, A. & Shah, A., 2018, p. 9). The user can deploy his own applications on the server. In Platform as a Service certain services such as email, databases and workflow engines are made available, and the user can use these while building his own applications. Software as a Service describes that an application is made available to the end user by paying for a subscription and the user might have certain rights to influence the functionality of said application. (Sarkar & Shah., 2018, p. 9) As cloud computing is accessed on-demand the pricing is usually pay-as-you-go.

There are advantages of using cloud computing, such as paying only for what you actually use, scaling, and the possibility to deploy the application in different regions which can reduce latency. Also, services can be used immediately and the need for maintaining data centers on one's own is not there anymore. (Amazon Web Services, n.d.)

The Serverless framework used in this project differs from traditional hosting as you only pay for what you use based on requests made and uses Lambda functions. Lambda can be used for any kind of application or backend service. (Amazon Web Services, n.d.) The application runs on Amazon's servers and Figure 5 AWS Lambda give us an idea of the workflow happening in the background.

Figure 5 AWS Lambda (Amazon Web Services, n.d.)



Amazon Cognito is used to handle user accounts and besides traditional accounts created also supports sign in technologies with Apple, Facebook, Twitter, and Amazon. It is highly secure and provides multi-factor authentication and users can be managed from the directory. (Amazon Web Services, n.d.)

3.3 MySQL

MySQL is a database system that has been available since 1995 and owned by Oracle since 2010. It is available as a free, open-source version and paid enterprise version. SQL stands for Structured Query Language. (TutorialsPoint, n.d.) SQL Queries can be created that the database can understand – such as adding, updating or deleting data from it. A database can hold a structure of information which can be created, updated, read, combined, and deleted. A database needs to run on a server, but a single instance can host multiple databases.

Databases mainly consist of tables that can have multiple columns – tables can be related to each other by sharing for example a foreign key. By this, it is easily possible to perform different joins between the tables, which allows to combine the data searched. The data entered must be defined to accept a certain data type, such as numeric data, Strings, or date formats. (MySQL Introduction, n.d.)

MySQL queries can be done in the console if installed, but there are many visualization tools available such as Adminer or MySQL Workbench, which make it easier to read the data and in case of Adminer also allows to create, update, and delete information in the tables manually without SQL queries.

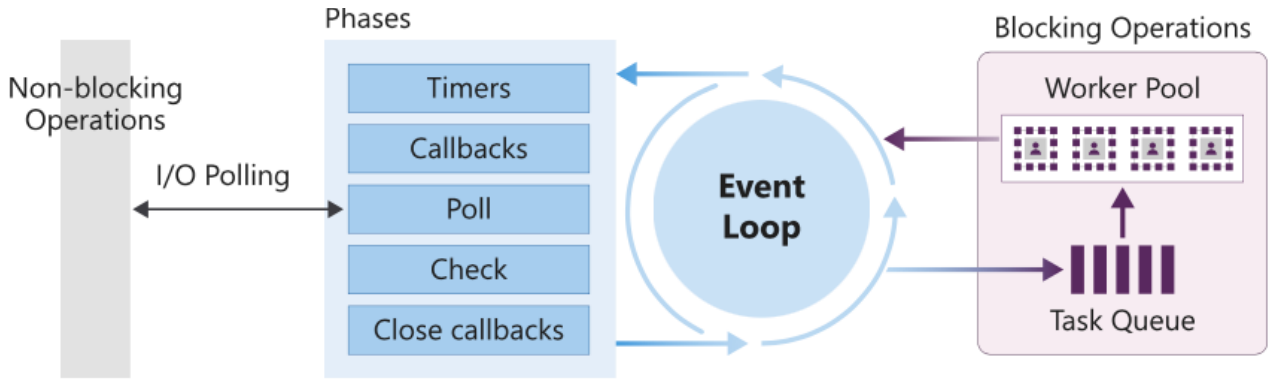
Databases can also make use of user account privileges, allowing only specific users to do certain operations. These can for example mean that a user can only select and insert, but an admin user can do all operations possible. In this application, most database operations and even the setup of the database tables will be handled with typeORM, which will be described further in the Node.js section of the theory part.

3.4 Node.js

Node.js is a server-side JavaScript environment and is mainly used in the backend for creating REST APIs, real-time services and microservices (Nandaa, 2018, p. 6). It is built on Google Chrome's V8 engine which makes it very fast, and the Node Package Manager has approximately 50000 packages that can be used to implement different features (What Is Node.Js, n.d.).

It is open-source and uses a single-threaded event loop., which means that it does only one thing at a time, before it runs the next task in the queue. It supports asynchronous programming, which means that the system gets notified when a process – for example processing data from a database is ready and the following code can be executed. The process can be seen in Figure 6. (Microsoft, n.d.)

Figure 6 Architecture of Node.js (Microsoft, n.d.)



Node.js can be installed as an executable, through Homebrew (a package manager for Linux and MacOS) or through the Node Version Manager.

TypeORM is a tool that can be installed on top of Node.js and is an ORM, which stands for object-relational mapping. It can be used in applications that use JavaScript or TypeScript and is an installable feature that can be used to make working with databases easier. The features it offers include but are not limited to creating entities and columns, an entity manager, relations between database tables, cascades – which means being able to delete a relational data table if we want to and data types. Tables can also be joined, and schemas can be defined. Changes can be migrated

and synced, such that new tables can be created completely from scratch through TypeORM. (TypeORM, n.d.)

Figure 7 shows an example Entity created with TypeORM. TypeORM will “translate” this for us and create a table called User, where the id must be of type number and will be our Primary Key, which is generated automatically. firstName and lastName refer to strings and the age is of type number. (TypeORM , n.d.)

Figure 7 TypeORM Entity Example (TypeORM , n.d.)

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number
}
```

There are inbuilt TypeORM functions that allow us to create, read, update or delete data from a table. In Figure 8 we can see an example where we are dealing with a database table called Photo. At first, a new photo object is created. Afterwards, the reference to the repository – which is the Entity – is made and the item gets saved. We can use the function find() to retrieve all data saved in a table. (TypeORM , n.d)

Figure 8 Saving and Retrieving Data using TypeORM (TypeORM , n.d)

```
import { Photo } from "./entity/Photo"
import { AppDataSource } from "./index"

const photo = new Photo()
photo.name = "Me and Bears"
photo.description = "I am near polar bears"
photo.filename = "photo-with-bears.jpg"
photo.views = 1
photo.isPublished = true

const photoRepository = AppDataSource.getRepository(Photo)

await photoRepository.save(photo)
console.log("Photo has been saved")

const savedPhotos = await photoRepository.find()
console.log("All photos from the db: ", savedPhotos)
```

As demonstrated in Figure 9, we can also update specific data by defining a condition – for example the id. We can change certain columns of the table, in this case the name or also multiple of them. Deleting is done with a similar logic, where we need to have a search condition to find a specific item we want to delete. (TypeORM , n.d.)

Figure 9 TypeORM Updating in the database (TypeORM , n.d.)

```
import { Photo } from "./entity/Photo"
import { AppDataSource } from "./index"

const photoRepository = AppDataSource.getRepository(Photo)
const photoToUpdate = await photoRepository.findOneBy({
    id: 1,
})
photoToUpdate.name = "Me, my friends and polar bears"
await photoRepository.save(photoToUpdate)
```

In more complex scenarios, there might be foreign keys in our Entities, which allows us to link two or more tables together. Depending on the desired functionality, these can be OneToOne, ManyToOne, OneToMany or ManyToMany relations. An example for this is that when we have a user table and a reactions table. There will always only be one user, but there may be many reactions giving the User table the need of a OneToMany relation and the reaction table the need of a ManyToOne relation. (TypeORM, n.d.)

For these more complex scenarios it is possible to use a so called QueryBuilder. Besides doing all normal operations, it is possible to perform joins on our MySQL tables if we have created the necessary relations. In the example below we want to select from the table user, but also have a reference to a linkedSheep and linkedCow with a specific id. The result will be all 3 tables joined together.

Figure 10 QueryBuilder example (TypeORM, n.d.)

```
const result = await dataSource
    .createQueryBuilder('user')
    .leftJoinAndSelect('user.linkedSheep', 'linkedSheep')
    .leftJoinAndSelect('user.linkedCow', 'linkedCow')
    .where('user.linkedSheep = :id', { id: sheepId })
    .andWhere('user.linkedCow = :id', { id: cowId });
```

As we can see, TypeORM can be a powerful tool for handling everything that has to do with our database and a convinient way instead of writing traditional MySQL queries from scratch.

4 Methods

In this chapter we will go through the methods used to develop the application. Agile Software Development was an approach used, but also an in-depth plan of the application was created in advance.

4.1 Agile Software Development

Agile is a Software Development Method that follows a certain mindset and one of the main ideas in to create and respond to change. It may contain methods such as Scrum, pair-programming, stand-ups, and sprints. The agile manifesto focuses on “individuals and interactions over processes and tools (…) working software over comprehensive documentation (…) customer collaboration over contract negotiation” and “responding to change over following a plan” (Layton et al. ,2020, p. 23).

A conversation can be more beneficial than using tools as communication is an effective way to solve and discuss issues and teamwork will strengthen the ability of a team to work together and the team itself can decide how they want to organize their work which means that processes can be adjusted quickly as needed. According to agile principles, documentation can be beneficial if it helps the team to succeed in developing their project – this includes documenting the requirements and technical specifications. As the focus is on communication, some documentation becomes inevitably obsolete. (Layton et al. ,2020, p. 23)

Our team followed these principles to a certain degree. The focus was continuously on developing the main functionality over the details and the documentation was limited. Meetings were held 1-2 times a week to update each other on the current status of the project and often the meeting was a turning point after which changes and suggestions by other team members were implemented. Additionally, meetings between me and the second developer were held occasionally as needed if problems occurred.

4.2 In depth plan of the application

Several features had already been developed for the application in advance. The idea of how the dashboard should look was made by a designer in Figma. The most important aspect was that the layout was supposed to use a Masonry Layout. Masonry is more versatile than Flexbox as it allows for different widths and heights of individual objects. A traditional Flexbox design only allows for the same height, and the next object in the second row will start at the height where the highest item ends. Masonry on the other hand is more versatile as it adjusts with the height and fits as many items as possible into the row taking the height into account. The design also included the possibility to react to specific components with Emojis. According to this design I started developing the project for this thesis.

Therefore, the goal of this thesis was to develop a content management system, as for now with the components predesigned by the designer but with possibility to expand on that idea by for example giving the user the opportunity to partially design their own components. This required setting up an editing page. It was decided early on that this should have a drag and drop functionality. Other requirements were setting up a basic CRUD system for the application and the ability to save the position where an item was dropped in the editor to show it to the user on the dashboard.

This included using the previously set up frontend for implementing the dashboard, editor, and Emoji Picker. This then had to be connected to the Node.js backend and combined by using an API which was done using the serverless framework provided by Amazon Web Services. TypeORM was used for creating and updating data tables in the dockerized MySQL database. The simplified process of the flow of the application can be seen in Figure 11.

Figure 11 Flow of the application

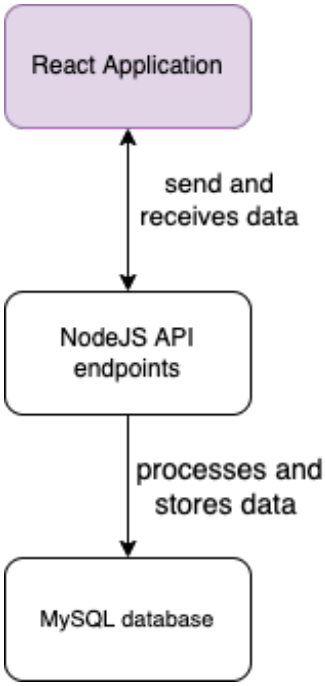


Figure 12 shows the original prototype of the dashboard editor. The idea was that created components initially get stored on the left side of the page, where it is possible to edit or delete a post. These components are inactive and only shown in the editor. The right side of the page shows the preview of the dashboard in the same style (Masonry) as on the actual dashboard. Items can be dropped from the left to the right and right to the left. The items on the right can also be rearranged in a way the user desires. Not included in the sketch is a button for creating a post. When creating or editing a post, a modal will open – either with empty default values or filled initially with the information the user has provided when making a post.

Deleting a post will create an alert to warn the user about the action. Only after accepting the alert the post will actually be deleted from the database. Users can react to posts with different emojis, which will be described in the Emoji Picker section of this thesis. Deleting a post also means deleting any emojis associated with it.

Figure 12 Prototype design

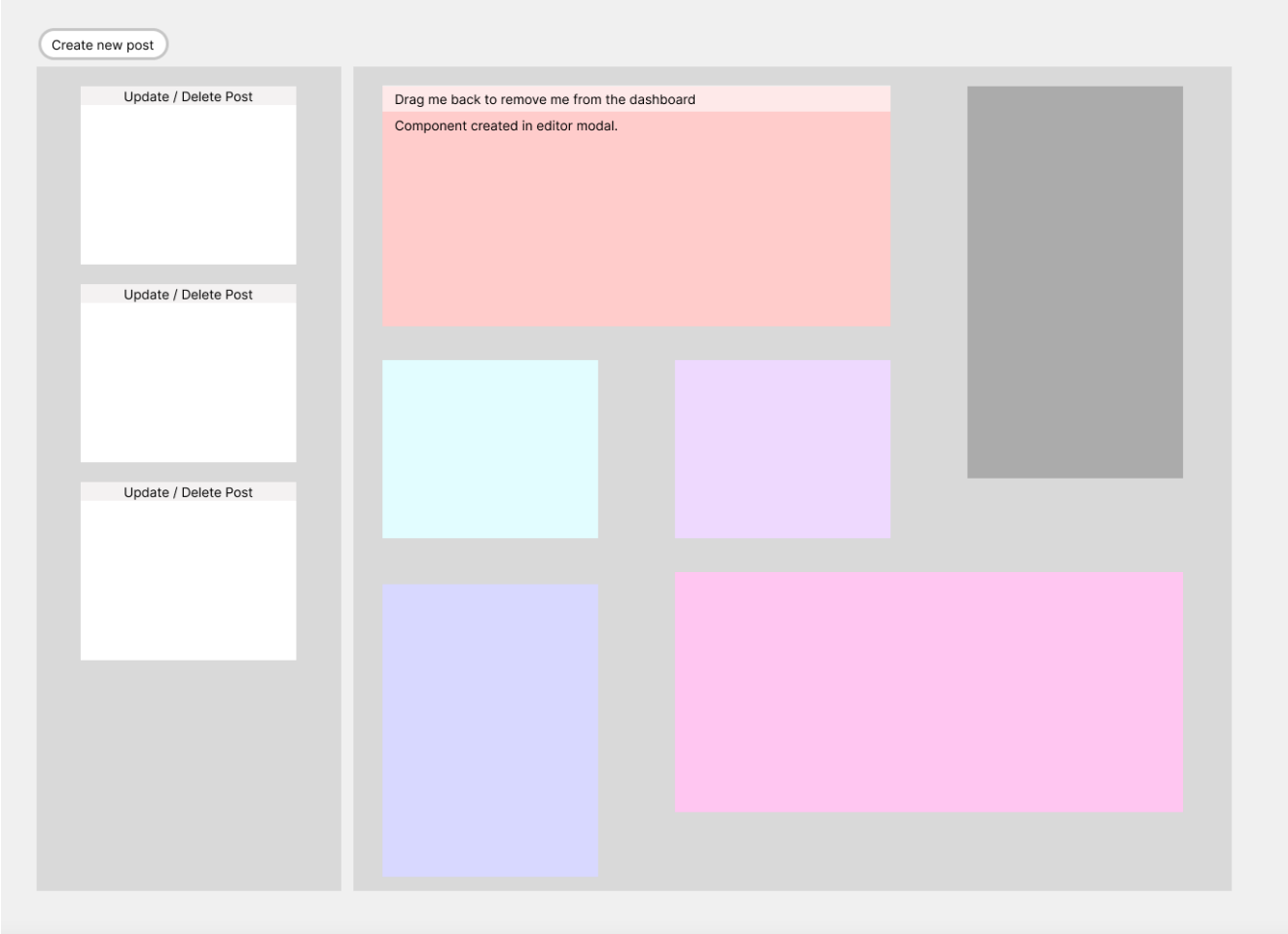
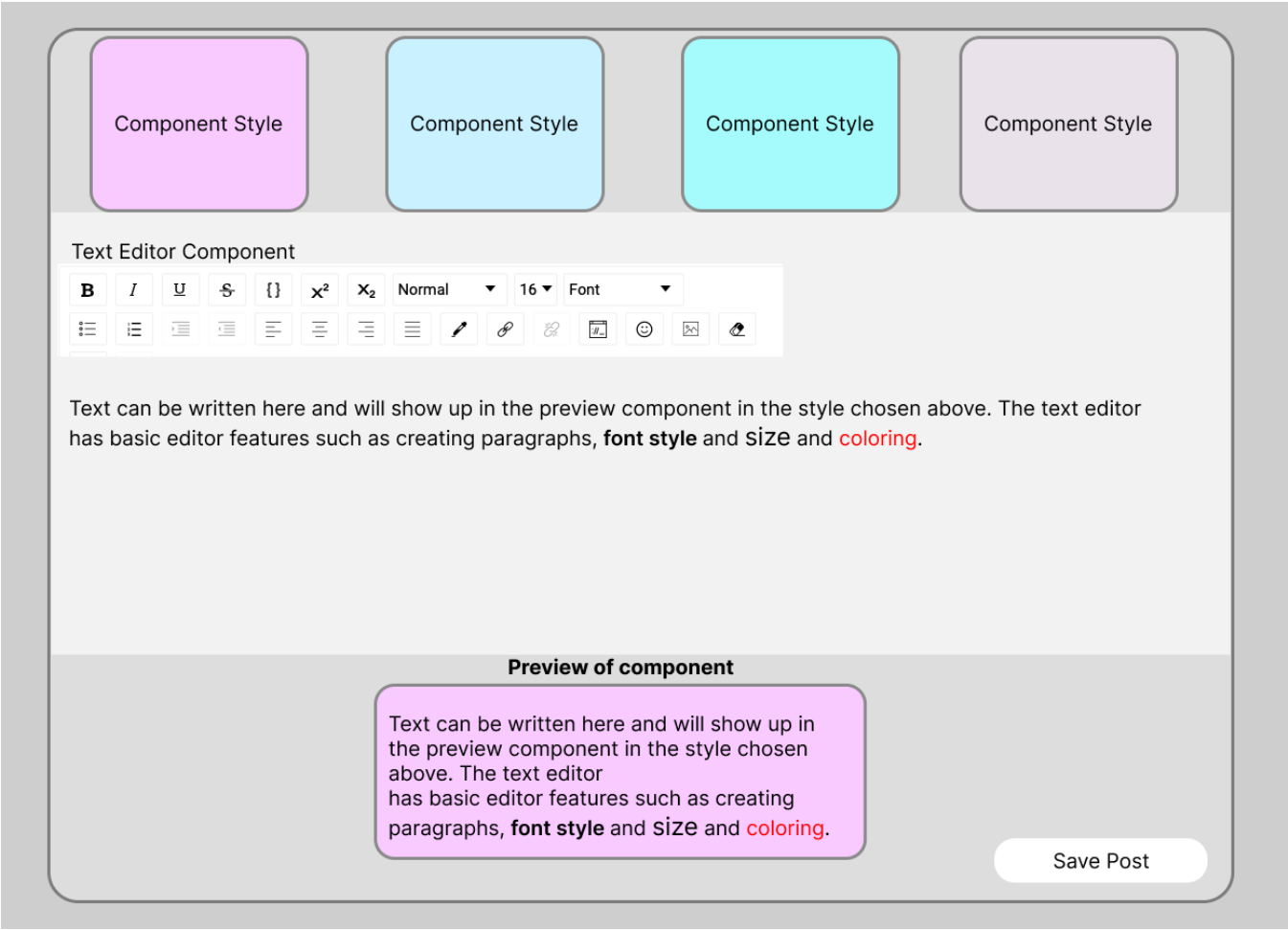


Figure 13 shows a simplified version of a modal that will open when creating a new post. The editor allows choosing a specific component, editing the text, and choosing an employee's photo from a list of registered accounts. A npm package called React Draft Wysiwyg (short for what you see is what you get) was used to accomplish this. Below the text editor, the user can see a live preview of the component until finally being able to save the post to the Dashboard.

Figure 13 Prototype design of editor modal



The approach for the feature was to first develop the most significant parts for its base functionality and going into the details at a later point. The system is made in a way that expandable features could be developed at any time.

Figure 14 shows all the endpoints that have been planned for this application and will be needed to ensure that the core functionality can be established.

Figure 14 Planned API endpoints

ENDPOINT	METHOD	FUNCTION
“/posts/posts”	POST	Save a new post to the dashboard database or updates a previously made post
“posts/dashboard”	POST	Save the current order in the Masonry component as decided by the user with drag and drop
“posts/posts”	GET	Retrieves all posts from the database
“posts/posts/{id}”	DELETE	Delete a post by using its id
“posts/reactions”	POST	Post or update and emoji reactions associated with a profile and dashboard post
“posts/reactions”	GET	Retrieve all reactions including profile and the specified dashboard post

4.3 Used Technologies

The project was written with the help of Visual Studio Code – a code editor with built-in support for JavaScript and TypeScript, which allowed us to use it for the backend and frontend application. As additional features, ESLint and Prettier were installed, which allow us to configure error messages, that do not directly break the code, but there might be something missing, such as a return statement – Prettier on the other hand does exactly what the name implies: it makes our code prettier.

If all developers use the same configuration, it makes submissions easier as for example Git will understand that the code is exactly the same, as there are no differences in spacing or length of a line of code. (Prettier · Opinionated Code Formatter, n.d.)

The frontend application is a React application with TypeScript. The backend application as Node.js application with Amazon's Serverless Framework and TypeORM. The Node Package Manager was used repeatedly to install needed dependencies. The MySQL database was run in a Docker container.

5 Implementation of the dashboard and editor

In this chapter I will go through the actual implementation of the project. I will start with the logic of the React application briefly explained in general and then investigate the dashboard and its editor. Then I will show how the Emoji Picker was created on top of that. This will give the reader an important overview of the data being handled; thus, we will continue by looking into the necessary database tables that were created with Typeorm. After that, the backend application's endpoints and functions required for handling data will be discussed. This thesis can be used as an inspiration for creating a similar dashboard handling system, but some aspects such as CSS and the design are intentionally left out as they are not required for describing the logic of the application.

5.1 Logic of React application

Developing the feature required adding information in different places of the application. There are many ways to structure a React App, and this is often down to personal preference. For this feature everything that gets shown to the user is provided in a folder called pages. Additionally, there is a folder called service that will handle the communication to the backend.

5.1.1 Logic of Dashboard

As explained in the project plan the dashboard was made with a Masonry design. To accomplish this, a package called **react-masonry-component** was installed. The setup of the dashboard page is rather simple – the data is fetched and displayed. For this I need to make use of our service that holds all our axios requests. Axios is a package that can be installed onto a React App and makes it easier to handle API endpoints. (Axios Docs, n.d.) A simple GET request is performed to get our stored data. Looking back at the plan made in advance, we recognize that the editor holds both active and inactive items – therefore there is a need to filter out inactive items. The database table has a column for the position where the order of the posts created by the user will be stored. Therefore, the items will be ordered by this position number to display it correctly and if the position is equal to NULL, it will be displayed in our inactive list.

Code Snippet 1 shows how the data will be displayed. Mapping through the items makes them visible in the Masonry Component. Each item is a so-called Card component which hold different information depending on where it will be displayed.

Code Snippet 1 Masonry

```
<Masonry
  elementType={"div"}
  options={masonryOptions}
  disableImagesLoaded={false}
  updateOnEachImageLoad={false}
>
  {itemList.map((card, i) => renderCard(card, i))}
</Masonry>
```

The card component (see Code Snippet 2) holds all our individual posts and processes them depending on where they are used. At the time being this component occurs in four places in our dashboard and editor – the first one being in our dashboard, the second and third one in our editor and the fourth one is the preview in the modal.

Code Snippet 2 Card Component

```
const renderCard = (card: any, index: any) => {
  return (
    <Card
      index={index}
      text={card.content}
      id={card.id}
      key={card.id}
      column={card.column}
      email={state.email}
      user={user}
      emojiPicker={true}
      reactions={fetchedEmojis}
      type={card.type}
      photo={card?.photo}
      dashboardId={card.id}
      moveCard={moveCard}
      typeOfCard="editor"
      onClick={function (): void {
        throw new Error("Function not implemented.")
      }}
      onClick2={function (): void {
        throw new Error("Function not implemented.")
      }}
    />
  )
}
```

```
interface CardProps {
  id: number
  text: string
  column: string
  index: number
  moveCard: (dragIndex: number, hoverIndex: number) => void
  type: string
  photo: string
  typeOfCard: string // fintranet, editor, preview
  onClick: () => void
  onClick2: () => void
  emojiPicker?: boolean
  emoji?: string
  dashboardId: number
  email?: string
  user?: any | undefined
  reactions?: any | undefined
}
```

For our dashboard the most relevant types that are required are the column, which defines whether the specific post should be a full-width post or not. We also need to know the type and photo which will define how our post will look and whether a photo has been chosen for the post or not.

The reactions made with the Emoji Picker, dashboardId, user and his email are needed to display all data correctly.

The data fetched is therefore our posts that were created, and the reactions made for those posts. We also need to know who the current user is to save a possible reaction to a post. The reactions are directly linked to a profile data table, and it is therefore possible to also display the username.

Code Snippet 3 Filtering data

```
const sorted = temporaryList.sort(
     (a: { position: number }, b: { position: number }) => a.position - b.position
   )
```

```
const actvieItems = itemList.filter(function (e: any) {
   return e.position !== null
 })
```

The Dashboard Editor saves the position of the content. Therefore, the posts need to be retrieved and before rendering them sorted in ascending order by position. In the next step, all objects that do not have a position will be filtered out – as seen in Code Snippet 3 - as this means that the post has been created but is currently inactive.

5.1.2 Logic of Dashboard Editor

The main user interface makes use of React Drag and Drop. For this, a npm package called **react-dnd** was installed. If we take a closer look at the plan made in advance, it becomes clear that for this editor three different drag and drop operations are needed. First, we want the user to be able to drag and item from the left flexbox to the right one and back. The second feature we want to accomplish is dragging items within the Masonry and being able to change their order. React-dnd provides us with features that make it possible to accomplish both operations. To use react-dnd we need to provide the area / item that we want to be draggable and a droppable area where the item is allowed to be dropped. We also need to define what data will be transferred and what we want to do with this data. (React DnD, n.d.) An example of this can be seen in Code Snippet 4.

Code Snippet 4 React-dnd

```
const [{ isDragging }, drag] = useDrag({
  type: ItemTypes.CARD,
  item: { index: index, id: id, content: text, column: column, type: type, photo: photo },

  collect: (monitor: any) => ({
    isDragging: monitor.isDragging(),
  }),
})
```

```
const [{ isOver }, drop] = useDrop(() => ({
  accept: ItemTypes.CARDMOVERIGHT,
  drop: (item) => addImageToBoard(item),
  collect: (monitor) => ({
    isOver: !!monitor.isOver(),
  }),
}))
```

In our example, all data related to our card item is transferred. A function (addImageToBoard) is used to add the card to the list it is supposed to be appended to. It can be defined what kind of item should be accepted and the drop area has to be named accordingly. Using drag and drop to only change the order of items, we receive the new index of the item and can thus save the position according to our item list Array. The inactive items also have a small icon on the top to either edit the post or delete it all together.

The Dashboard Editor gives us the possibility to press a button to create a new post. This opens a modal which entails our editor. It is mainly divided into three parts: in the upper area we have a Selector Component which shows us seven different styles for our post to choose from, followed by the React Draft Wysiwyg (npm package react-draft-wysiwyg) and another selector component to choose a photo. In the bottom area we get shown a preview of our created post with the text added in the editor being displayed in real-time. The Selector Component with the styles uses almost the same logic throughout the entire application, the only difference being whether a photo can be added and whether reacting with Emojis should be possible or not. If we look back at our Card Component, we can see that it requires a value called type. This defines which of the seven is supposed to be rendered.

Content entered in the editor will be saved in in our MySQL database as a string containing raw html. It can then be parsed back to HTML that will be compiled which means that it keeps all its formatting, such as bold text or colouring decided by the author who was using the editor. (Html-to-Draftjs, n.d.)

When creating a new post, it needs to be identified whether this is a newly created post or an update to a previous post. If the user presses the edit icon, the state of updating is set to true and according to this either a request to update or create to our backend API is sent. If creating or updating a post, this information is rendered in the frontend of our application. There are two variables to store our data called itemList and board. itemList stores our inactive or recently created post, board those that are supposed to be shown on the dashboard.

Code Snippet 5 Updating a post

```
const updatePost = (id: any, content: any, type: any) => {
  setShowModal(true)
  setType(type)
  setContent(content)
  const contentBlock = htmlToDraft(content)
  const contentState = ContentState.createFromBlockArray(contentBlock.contentBlocks)
  const editorState = EditorState.createWithContent(contentState)
  setEditorState(editorState)
  setId(id)
  setUpdate(true)
}
```

Updating a post (Code Snippet 5) therefore requires us to show the modal and send forward the type, id and content. The type identifies which styled component should be rendered in the preview, the id is relevant for knowing which post exactly we are dealing with, and the content is the information the user previously entered into the text editor. setUpdate needs to be set to true to true which will tell our function for creating the post how to handle the request. For deleting a post, the user gets shown an alert to confirm whether he really wants to delete the post. For handling the request, only the id of the current post is required.

Code Snippet 6 Axios Request example

```
fetchPosts(): Promise<any> {
  return new Promise(async (resolve, reject) => {
    const url = new URL(HTTP_API_URL + "/posts/posts")
    try {
      const response = await axios.get(url.toString(), this.addAuthorizationHeaders())
      resolve(response.data)
    } catch (e) {
      reject(e)
    }
  })
}
```

All of our API requests are handled in a separate folder and files called service. This needs to be imported in the file where the functions will be used. In Code Snippet 6 we can see the fetch request I created for getting posts. I am using axios for this – the most important aspect is that I need to add our Authorization Headers. These will not be described in detail in this thesis as they were not developed by me, but a short introduction is that it identifies what kind of user is trying to access a page or make a request. As of now, the user types are admin, HR, and regular user. In our Dashboard, we probably only want admins and HR to be able to create posts to the Dashboard, but the backend also allows adding more user types if needed.

5.1.3 Emoji Picker component

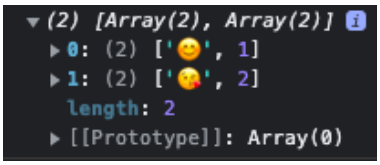
For the Emoji Picker component, a npm package called emoji-picker-react was used. It provides us with a standard Emoji Picker that has the functionality to save the chosen Emoji in a React State. It has an onClick event to update this state. (Emoji-Picker-React, n.d.) Figure 15 shows the idea of rendering the chosen Emojis as a reaction to a post by the user. Multiple Emojis should be summarized, and a number be displayed next to them – by hovering over a chosen Emoji we can see an overlay giving us the names of the users having reacted. If a user reacts twice, the emoji should be replaced by the new reaction.

Figure 15 Emoji Picker

To achieve this the Emoji Picker has been created as its own component that can be added to our Card Object. Therefore, the data is fetched in the Editor itself but processed it in the component to provide reusability if so desired. Our Emoji consists of an id, the emoji itself, the dashboard it is associated with and the profiles that have reacted to it. It also provides a light and dark mode depending on the style of the post which slightly changes the coloring of the interface. Saving a reaction is rather straight forward but rendering the view has some complexity – we need to consider that we receive a list of all reactions and therefore need to write functions to check for double occurrences and to save the usernames. A filter function is used to get the reactions that are relevant to our post by comparing the dashboard id.

Code Snippet 7 Counting Occurrences

```
const map = emojis.reduce(function (prev: any, cur: any) {
  prev[cur.emoji] = (prev[cur.emoji] || 0) + 1
  return prev
}, {})
setOccurences(Object.entries(map))
```



A reduce function is used count the amount of double occurrences and the result is an Array containing the emoji followed by an Integer. The information is used to remove the duplicates and add the usernames at the same time – the process of this can be seen in Code Snippet 8. The Array of reactions retrieved from the backend is used to remove all duplicate emojis from the list – in the next step I add the occurrences I calculated in the previous step to the Array, before finally using our initial array again to add all the usernames to their respective emoji – without having any duplicate Emoji entries anymore.

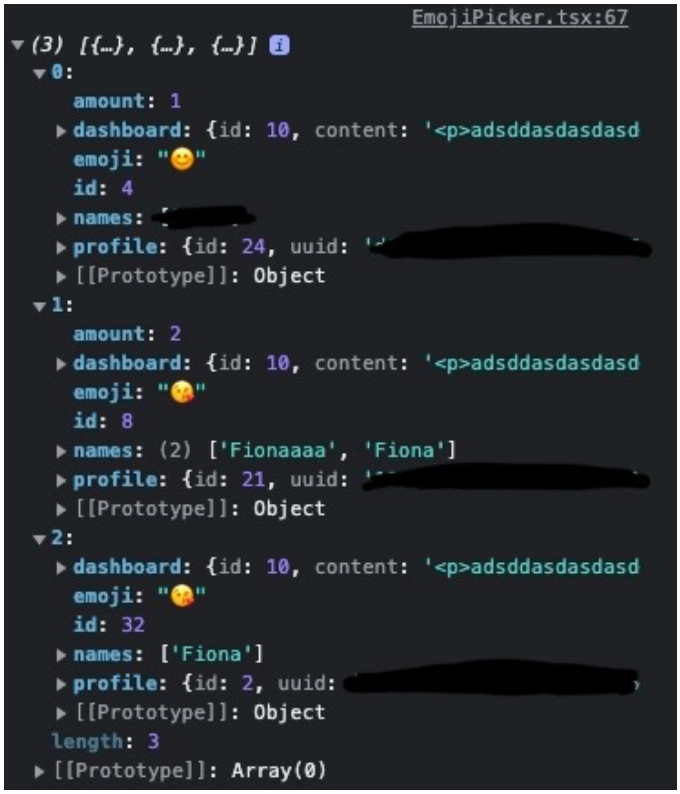
Code Snippet 8 Removing duplicated and adding names

```
const occuring = occurences.map((occ: any, index: any) => {
  const removeDuplicates = emojis.filter(
    (value: any, index: any, self: any) =>
      index === self.findIndex((t: any) => t.emoji === value.emoji)
  )
  withoutDuplicates = removeDuplicates
  if (withoutDuplicates.length > 0) {
    withoutDuplicates[index].amount = occ[1]
  }
  for (let y = 0; y < withoutDuplicates.length; y++) {
    withoutDuplicates[y].names = []

    for (let i = 0; i < emojis.length; i++) {
      if (withoutDuplicates[y].emoji === emojis[i].emoji) {
        withoutDuplicates[y].names.push(emojis[i].profile.firstName)
      }
    }
  }
})
```

Having done this, the result is an Array containing all the information needed to render our Emojis in the component. The structure of the final array looks as follows:

Code Snippet 9 Emoji Array Structure



The emoji list can now be rendered in a desired way to achieve the look we want to. The names included in the array can be used to show who has reacted to a post in a hovering overlay. For this the HTML options onMouseOver and onMouseOut are used as they can identify when a user hovers over a specific div – in our case an individual emoji.

5.2 Setting up database tables with TypeORM in NodeJS

Handling the data for the dashboard required us to make use of totally four database tables. Two of these were developed previously by another developer: our user and his profile. Our **user** database table stores the most basic information about our user, such as his email, phone number, name and registration status and is linked to his profile. The **profile** has more detailed information that is not directly relevant to our Dashboard, but it stores for example the profile photo which may be used in the application and allows us to know who has reacted with emojis to a post. During the process of handling our API requests I will make use of the relations to both of those tables, but as they were not developed by myself, they will not be described in too much detail.

Code Snippet 10 Entities Typeorm

```typescript
@Entity()
export class Dashboard {
  @PrimaryGeneratedColumn()
  id: number

  @Column({ nullable: false })
  content: string

  @Column({ nullable: false })
  column: string

  @Column({ nullable: false })
  type: string

  @Column({ nullable: false })
  active: boolean

  @Column({ nullable: false })
  date: string

  @Column({ nullable: true })
  position: number

  @Column({ nullable: true })
  photo: string

  @Column({ nullable: true })
  emojiPicker: boolean

  @OneToMany("DashboardReactions", "dashboard")
  @JoinColumn()
  reactions: DashboardReactions[]
}
```

```typescript
@Entity("dashboard_reactions")
export class DashboardReactions {
  @PrimaryGeneratedColumn()
  id: number

  @Column({ charset: "utf16", collation: "utf16unicode_ci" })
  emoji: string

  @ManyToOne("Dashboard", "id", { cascade: true, onDelete: "CASCADE" })
  dashboard: Dashboard

  @ManyToOne("Profile", "profileReactions")
  @JoinColumn()
  profile: Profile
}
```

The majority of columns created in typeorm are very basic columns without special relations to for example other tables. The id of both the Dashboard and Reactions table is auto generated. Most columns are either of type string, number or Boolean (TypeORM, n.d.). However, storing emojis requires us to use a charset. In our case the emojis require using utf16, which includes enough bits to display all the emojis our Emoji Picker provides. I also take advantage of TypeORM's possibility for making relations. Our Dashboard table has a OneToMany relation to our Dashboard Reactions, whereas the Reaction tables has a ManyToOne relation to the Dashboard and the Profile. In this case, OneToMany means that while there will always just be one post saved into our dashboard, this post may have multiple reactions assigned to it. There can however be many reactions linked to one post; therefore, ManyToOne is used. On the reactions table, onDelete needs to be set to cascade. (TypeORM, n.d.) This makes it possible to delete a post in the dashboard table and will delete all occurrences of reactions related to this post – this avoids not being able to delete content from the data tables due to foreign constraints in MySQL.

Figure 16 MySQL Datatables

Column	Type	Comment
id	int(11) *Auto Increment*	
emoji	varchar(255)	
dashboard_id	int(11) *NULL*	
profile_id	int(11) *NULL*	

Indexes

PRIMARY	*id*
INDEX	*dashboard_id*
INDEX	*profile_id*

Alter indexes

Foreign keys

Source	Target	ON DELETE	ON UPDATE	
dashboard_id	dashboard(*id*)	CASCADE	NO ACTION	Alter
profile_id	profile(*id*)	NO ACTION	NO ACTION	Alter

Column	Type	Comment
id	int(11) *Auto Increment*	
content	varchar(255)	
column	varchar(255)	
type	varchar(255)	
active	tinyint(4)	
date	varchar(255)	
position	int(11) *NULL*	
photo	varchar(255) *NULL*	
emoji_picker	tinyint(4) *NULL*	

Figure 16 shows us how both data tables look like in Adminer – a tool for handling and visualizing MySQL databases. The reaction datable has Indexes related to the Dashboard table and profile. We receive the id, which allows us to easily join the tables to handle data. It can also be seen that the dashboard_id and profile_id are foreign keys.

5.3 API endpoints for dashboard and reactions in NodeJS

As the serverless framework is used in our application all our routes need to be defined in our serverless.yml file. Code Snippet 11 shows an example of configuring these paths. A handler needs to be defined in which our API endpoints are created. For each possible request type, an event will be created in which the method need to be specified (such as POST, GET, PUT, DELETE), the URL path and in our case cors – which is mainly used for development purposes as it allows access to our API from any origin - and authorizers, which are related to user authentication.

Code Snippet 11 Example from Serverless.yml file

```
posts:
  handler: src/handlers/posts.handler
  events:
    - http:
        method: post
        path: posts/posts
        cors: true
        authorizer:
          name: authorizer
          arn: arn:aws:cognito-idp:#{AWS::Region}:#{AWS::AccountId}:userpool/${self:custom.config.userPoolId}
    - http:
        method: get
        path: posts/posts
        cors: true
        authorizer:
          name: authorizer
          arn: arn:aws:cognito-idp:#{AWS::Region}:#{AWS::AccountId}:userpool/${self:custom.config.userPoolId}
```

For handling our dashboard, editor, and reactions all together seven API endpoints had to be created, which will now be described in more detail. The functions used to handle the data and to do typeORM operations will be explained in the next subchapters.

Our dashboard requires us to use at least five different types of requests. Firstly, two post requests are needed. The first one for just saving the data after a post has been created and the second one storing the order of the items created. Logically, a GET request is used to retrieve the data, a DELETE request to delete posts and a PUT request for updating posts. Furthermore, a GET and POST request for handling our reactions is created.

Code Snippet 12 Example API endpoint

```
    path: "/posts/posts",
    method: "POST",
    action: async (request): Promise<ProxyIntegrationResult> => {
      const data = request.body as PostRequest
      const dbConnection = await new Database().getConnection()
      const item = await postService.createItem(dbConnection, data)
      return { body: JSON.stringify(item) }
    },
  },
```

Code Snippet 12 shows an example of how the routes in the handler need to be defined. The path and method need to match the configuration in our serverless.yml file. A postService file has been created to handle the data. We receive our request which is defined as a type of ProxyIntegrationResult – which is defined by us and means that our request cannot fail or it will throw an error. Our received body is our data sent from the frontend which we will receive as a PostRequest – which is a TypeScript interface defined by us to validate that the data received is in

the exact format as expected. The item is returned after receiving it after handling our typeorm operations which is not necessary in the deployed version but can help us during the development process to see whether our operation was successful.

5.3.1 Functions for Dashboard

TypeORM provides relatively simple ways to do basic operations – all functions use async and return a promise. Async is a concept in programming and JavaScript that makes sure that the program will wait for data to be retrieved before the code will compile the next line – it is also very common in making API calls in the frontend of the application, as the response time from the server may vary. We first need to define the repository we want to work with, followed by the operation we want to do. Repository.find() returns all items in the database table, whereas findeOne allows us to search and delete() to delete by the primary key id (TypeORM, n.d.). An example of how the posts are retrieved can be seen in Code Snippet 13.

Code Snippet 13 Typeorm find operation

```
export const getPosts = async (dbConnection: Connection): Promise<Dashboard[]> => {
  const repository = dbConnection.getRepository(Dashboard)
  const items = await repository.find()
  return items
}
```

Things become a little more complex once we want to save the order of our items in the dashboard. Our frontend rearranges the array according to the position of the items on our dashboard. Looking back at our MySQL tables we find the column position, which is the order our items are arranged in. It can either be a number or null. Null automatically means that the item is inactive now and therefore only visible in our editor but not in the dashboard itself. This implies that the order needs to be created from scratch every time the user decided to save the dashboard.

Code Snippet 14 Saving Dashboard Order

```
await repository
  .createQueryBuilder()
  .update(Dashboard)
  .where(`position IS NOT NULL`)
  .set({ position: null })
  .execute()

if (item.position.length > 0) {
  const ourItem = item.position[0].id
  for (let i = 0; i < item.position.length; i++) {
    const ourItem = item.position[i].id
    const newItem = await repository.findOne({
      id: ourItem,
    })
    newItem.position = i
    await repository.save(newItem)
  }
}
```

Code Snippet 14 shows the solution to this problem. As explained previously, items that are not active have a position of null. I therefore use TypeORM's integrated query builder and set all positions to null that currently still have a number saved. Afterwards I make sure that we still have any items in the Array and thus received any data from our Dashboard – without any posts, there can't be an order of posts. I go through the sorted Array and set the position accordingly ascending from zero and save the position in the next step. By doing this the data that was fetched can afterwards be sorted by position in the frontend and the order of posts will be maintained.

Code Snippet 15 Creating a new post

```
export const createItem = async (dbConnection: Connection, data: DashboardSchema): Promise<Dashboard> => {
  const repository = dbConnection.getRepository(Dashboard)
  const item = repository.create()
  item.content = data.content
  item.column = data.column
  item.type = data.type
  item.active = data.active
  item.date = data.date
  item.photo = data.photo

  await repository.save(item)
  return item
}
```

It is possible to create a new post with the data received. The connection needs to be established first and it is possible to get the repository. Afterwards, the columns that need to be updated will

be defined (see Code Snippet 15) and finally, the entry will be saved to the database. The item is returned only for debugging reasons and this is not necessary in a deployed version.

5.3.2 Functions for Reactions

Saving and retrieving reactions is slightly more advanced but we can make use of very beneficial TypeORM features. For posting and updating I search for a duplicate entry of the same dashboard post and profile in our Reactions data table – if the query returns undefined, I know that the person does not have reacted to that specific post yet and can create a new entry in our database. If the query returns something else than undefined – thus an existing item – I take that item and update only the info on the emoji used.

Code Snippet 16 Typeorm QueryBuilder Leftjoin

```
export const getReactions = async (dbConnection: Connection): Promise<any> => {
  const repository = dbConnection.getRepository(DashboardReactions)

  const items = await repository
    .createQueryBuilder("reactions")
    .leftJoinAndSelect("reactions.profile", "prof")
    .leftJoinAndSelect("reactions.dashboard", "dashb")
    .getMany()

  return items
}

```

To get all the items stored in our reaction table, I can take advantage of our OneToMany and ManyToOne relation I set up when creating the MySQL tables as can be seen in Code Snippet 16. This allows us to make use of joining tables and I can retrieve all the information I need at once – I need all the information on the reactions, but they also need to include the profile they were sent from and the dashboard – more specifically the dashboard post that they are attached to.

6 Results

The implementation of the project was successful, and the result is a fully functional feature. While writing this thesis, there were a few functionalities that I would write different in the future – some of them might still be changed in the near future. Both research questions can clearly be answered with a yes, as it was possible to develop this feature.

The biggest learning curve for myself is that I would more often handle and structure the data already in the backend before fetching it. A good example for this is the functionality of the Emoji Picker. While it works, it would probably be more feasible to receive the already filtered data from the backend including all the details we need, and this would furthermore improve readability and reusability of the code and consistency. Considering that I worked on both the frontend and backend, this option was feasible, but if there were separate developers the data processing should probably be happening in the backend primarily, as the person working on the backend knows about the data structure in more detail.

Also, at the time being, reactions cannot be deleted which is a feature that is being implemented in the future. The feature could be developed further to allow deletion when clicking one's own emoji and adding a reaction when pressing a reaction of somebody else.

Handling multiple Card component in different places that partly need to look completely different was challenging. In the future, I would definitely rather create a second Card component in places where the CSS and logic differ immensely.

7 Summary

The goal of this thesis was to develop a dashboard and editor for an existing Intranet for Financial Labs Oy. This was done using Full Stack JavaScript development with React in the frontend and Node.js, Amazon Webservice's Serverless framework and typeORM in the backend. It required studying these frameworks in depth and understanding the previously built application which helped to understand the structure of the code and requirements.

The application works as expected and all the main features were implemented successfully, giving it the basic functionality needed.

The project was successful, but there remain details that can be developed in the future. This includes CSS improvements, recreating the Emoji Picker to work more like comparable Pickers that are widely available, allowing the user to add a reaction by clicking on someone else's reaction and deleting a reaction if so desired.

References

AWS Serverless Application Repository—Amazon Web Services. (n.d.). Amazon Web Services, Inc. Retrieved September 27, 2022, from https://aws.amazon.com/serverless/serverlessrepo/

Components and Props – React. (n.d.). Retrieved September 9, 2022, from https://reactjs.org/docs/components-and-props.html

Compute Services—Overview of Amazon Web Services. (n.d.). Retrieved September 27, 2022, from https://docs.aws.amazon.com/whitepapers/latest/aws-overview/compute-services.html#aws-lambda

Conditional Rendering – React. (n.d.). Retrieved October 10, 2022, from https://reactjs.org/docs/conditional-rendering.html

Context – React. (n.d.). Retrieved October 10, 2022, from https://reactjs.org/docs/context.html

Crowder. (2020). *JavaScript*. John Wiley & Sons, Inc.

Discuss the history of MySQL. (n.d.). Retrieved September 27, 2022, from https://www.tutorialspoint.com/discuss-the-history-of-mysql

Documentation—Everyday Types. (n.d.). Retrieved September 9, 2022, from https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

Emoji-picker-react. (n.d.). Npm. Retrieved November 7, 2022, from https://www.npmjs.com/package/emoji-picker-react

Getting Started | Axios Docs. (n.d.). Retrieved November 7, 2022, from https://axios-http.com/docs/intro

Hooks at a Glance – React. (n.d.). Retrieved November 6, 2022, from https://reactjs.org/docs/hooks-overview.html

Html-to-draftjs. (n.d.). Npm. Retrieved November 7, 2022, from https://www.npmjs.com/package/html-to-draftjs

Introducing Hooks – React. (n.d.). Retrieved September 9, 2022, from https://reactjs.org/docs/hooks-intro.html

JavaScript | MDN. (n.d.). Retrieved September 8, 2022, from https://developer.mozilla.org/en-US/docs/Web/JavaScript

Knorr, E. (2018, August 17). *Interview: Brendan Eich on JavaScript's blessing and curse*. InfoWorld. https://www.infoworld.com/article/3294999/interview-brendan-eich-on-javascripts-blessing-and-curse.html

Layton, M. k., Ostermiller, S. J. & Kynaston, D. J. (2020). *Agile project management for dummies* (3rd edition.). John Wiley & Sons, Inc.

Microsoft. (n.d.). *How Node.js works—Training*. Retrieved September 29, 2022, from https://learn.microsoft.com/en-us/training/modules/intro-to-nodejs/3-how-works

MySQL Introduction. (n.d.). Retrieved November 6, 2022, from https://www.w3schools.com/MySQL/mysql_intro.asp

Nandaa, A. (2018). Beginning API development with Node.js. Packt Publishing.

Prettier · Opinionated Code Formatter. (n.d.). Retrieved November 7, 2022, from https://prettier.io/index.html

React DnD. (n.d.). Retrieved November 7, 2022, from https://react-dnd.github.io/react-dnd/about

React Tutorial: An Overview and Walkthrough. (n.d.). Retrieved September 9, 2022, from https://www.taniarascia.com/getting-started-with-react/

Sarkar, A. & Shah, A. (2018). *Learning AWS: Design, build, and deploy responsive applications using AWS cloud components* (Second edition.). Packt.

Select using Query Builder | TypeORM. (n.d.). Retrieved October 10, 2022, from https://typeorm.io/select-query-builder#

Serverless Computing—AWS Lambda—Amazon Web Services. (n.d.). Amazon Web Services, Inc. Retrieved September 15, 2022, from https://aws.amazon.com/lambda/

Six advantages of cloud computing—Overview of Amazon Web Services. (n.d.). Retrieved September 27, 2022, from https://docs.aws.amazon.com/whitepapers/latest/aws-overview/six-advantages-of-cloud-computing.html

Stack Overflow Developer Survey 2022. (n.d.). Stack Overflow. Retrieved September 8, 2022, from https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022

TypeORM - Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms. (n.d.). Retrieved October 10, 2022, from https://typeorm.io/

Vanier, E., Birju Shah, Tejaswi Malepati, Malepati, T. & Shah, B. (2019). Advanced MySQL 8. Packt Publishing.

What is Amazon Cognito? - Amazon Cognito. (n.d.). Retrieved September 27, 2022, from https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html

What is JavaScript? - Learn web development | MDN. (n.d.). Retrieved September 8, 2022, from https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

What is Node.js: A Comprehensive Guide. (n.d.). Simplilearn.Com. Retrieved September 27, 2022, from https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs

Annex 1: Material management plan

Development project:

During the development project, information and to-do lists were kept in Obsidian, which stores data only on the personal drive, in which technical information about the project is collected. This information is analyzed for the thesis. The project and to-do lists are stored on drive C of the author's computer and is regularly backed up to an external hard drive. The information and project are kept at station C for at least one year after the completion of the thesis. The project is also backed up in the company’s Bitbucket account.