OULUN AMMATTIKORKEAKOULU

Huy Bui

Merchant Application

Merchant Application

Huy Bui
Final projects
Winter 2022
Bachelor of Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author(s): Huy Bui
Title of the thesis: Merchant Application
Thesis examiner(s): Janne Kumpuoja
Term and year of thesis completion: 2023 Pages: 41 + 3 appendices

---

Munchi, a food delivery company based in Helsinki, Finland, has been relying on a third-party software program for managing their delivery operations. However, the program has proven to be unreliable, with frequent bugs and a user interface that is difficult for their staff to navigate. The CEO of Munchi, Mario Henningson, experienced these issues first hand and realized that a new, custom-built system was needed to streamline their delivery operations and provide a better experience for both their staff and customers.

To address these challenges, Mario and the team at Munchi worked together to tackle these challenges by creating a new application. Our solution was to create a new merchant delivery application, which I was responsible for implementing. This thesis documents the implementation process of the application. The application was designed to be user-friendly, intuitive, and highly functional. The team implemented features such as real-time order tracking, automated inventory management, and customer feedback options to provide businesses with valuable insights into their operations and enable them to make data-driven decisions to improve their services.

Throughout the development process, the team encountered various challenges, such as integrating the application with existing restaurant management systems and ensuring the security of customer data. However, with effective communication, teamwork, and problem-solving skills, they were able to overcome these obstacles and successfully launch the application.

The merchant delivery application developed by Munchi has been met with positive feedback from both the company's staff and customers, as it offers a seamless and efficient delivery experience. However, it should be noted that the current version of the application is still in the demo stage and is not yet available for distribution to other restaurants. Nonetheless, the Munchi team is actively working to improve the application's features and functionality to ensure that it meets the needs and expectations of the market. The development of this application has provided valuable insights and knowledge to the Munchi team, which will be used to continue to improve the application's performance and stay ahead of the competition. Ultimately, the merchant delivery application has the potential to make a significant impact on the food delivery industry and help Munchi to expand its operations and solidify its position in the market.

Keywords: merchant application, restaurant system

PREFACE

I would like to express my deepest gratitude to my teacher, Janne Kumpoja, for his invaluable guidance and support throughout the process of writing this thesis. His expertise and insightful feedback have been instrumental in shaping the direction and scope of this project. I am also thankful for his dedication to helping me improve my writing skills and for his patience in answering my questions and addressing my concerns. Without his encouragement and advice, this thesis would not have been possible.

I would like to express my sincere gratitude to Mario Henningson, the CEO, for providing me with the opportunity to be a part of this organization.

I would also like to extend my gratitude to Phong, my tech lead, for his unwavering support and mentorship. Phong has been an incredible resource throughout this project, providing me with guidance, motivation, and valuable feedback. His expertise in web technologies and problem-solving skills have been invaluable, and I am grateful for his dedication to helping me succeed.

I would like to express my appreciation to my girlfriend, Penny, for her unwavering support and encouragement throughout this journey. Her love and support have been a constant source of motivation, and I am grateful for her patience and understanding during this challenging time.

Last but not least, I would like to thank Mrs. Nhan for her support and guidance since the beginning of my studies in web technologies. Her passion for teaching and expertise in the field have inspired me to pursue my passion for software development, and I am grateful for the knowledge and skills that she has imparted to me.

Together, these individuals have played a crucial role in the successful completion of this project. Their support, guidance, and encouragement have been instrumental in helping me overcome the challenges that I have faced, and I am grateful for their contributions to this thesis.

LIST OF ABBREVIATIONS

LTS: Long term support
CMD: Command Prompt
VSC: Visual Studio Code
OOP: Object Orient Programming
TS: Typescript,
JS: JavaScript
MUI: Material UI
ICT: Information communications technology
JSX: JavaScript Syntax Extension
HTML: Hypertext Markup Language
CSS: Cascading Style Sheet
VDOM: Virtual Document Object Model
REST: Representational State Transfer
API: Application Programming Interface
RDMS: Relational Database Managing System
ORM: Object–relational mapping

CONTENTS

1 INTRODUCTION

As the world continues to develop, communication becomes increasingly crucial to connect people more closely. For service companies, having an effective communication system is necessary to understand and keep in touch with clients. During the COVID-19 pandemic, the delivery service worked tirelessly to fulfill customer needs, not just for shopping, but for food delivery as well. By ordering food through a mobile application, customers can have their desired meals in about 15-20 minutes. However, the communication system provided by the third-party company for Munchi was not reliable and affected the company's customers.

Using a merchant application from a third-party company can be a risky factor for businesses, as it may not always be reliable. One of the biggest issues with third-party merchant application is their tendency to have bugs and technical glitches, which can severely impact a business's ability to manage its deliveries efficiently. In addition to the reliability issue, many third-party merchant applications also suffer from a buggy interface that can make it difficult for users to navigate and use the application effectively. This lack of user-friendliness can further compound the challenges of using a third-party merchant application. Moreover, some of these applications may lack logic in terms of how they are designed and operate, which can create confusion and frustration for users. In short, while third-party merchant applications may seem like a convenient solution for businesses looking to streamline their delivery process, their unreliability, buggy interface, difficulty of use, and lack of logic make them a risky choice.

Introducing a new merchant application developed by a company that aims to address the key issues that businesses face when using third-party merchant applications. Many businesses have experienced the frustration of using unreliable, buggy, and hard-to-use third-party merchant applications. The company has worked tirelessly to create a solution that addresses these concerns. The new merchant applications boast a reliable and stable performance with no bugs or technical glitches. Additionally, the applications have been designed with a user-friendly interface that is intuitive and easy to navigate, ensuring that businesses can manage their deliveries efficiently and without any hassle. The company has also incorporated advanced logic and algorithms into the applications, making it a smart and efficient tool for managing the delivery process. With the new merchant applications, businesses can enjoy the peace of mind that comes with a reliable and effective tool for managing their deliveries.

2 THEORETICAL BACKGROUND

In order to effectively implement any project, it is essential to have an up-to-date understanding of the relevant technologies. This is particularly important in the context of modern software development, where a range of open-source technologies are commonly employed. As such, before any implementation work can begin, a thorough exploration of the theoretical background is necessary. This section will provide an overview of the key theoretical concepts and technologies that form the foundation of the project and will highlight the open-source technologies that are integral to its success.

2.1 JavaScript

JavaScript is a lightweight, open-source programming language used widely in web development these days. For it is most well known as scripting language for website but it has been enhanced to run not only on websites but on servers as well, typical examples like Nodejs, Apache CouchDB, etc. JavaScript supports multiple programming aspects such as OOP, multi-paradigm, single-threaded, dynamic programming. [1][2].

In 2010s, most website was made with only HTML and CSS. With the help of these two, a website was created and styled perfectly but it couldn't be interactive with hyperlink and modern features. JS is a game changer for developers. JS was designed for speed, dynamic pages, reduce memory loss, responsive content and enable us to play video and music on website. From then it creates the perfect combo included HTML, CSS, and JS to build a complete and responsive modern website. [5].

2.2 Typescript

Typescript is a free, open-source programming language which is developed and maintained by Microsoft. You can say that Typescript is the evolution of JavaScript with extra features. [1].

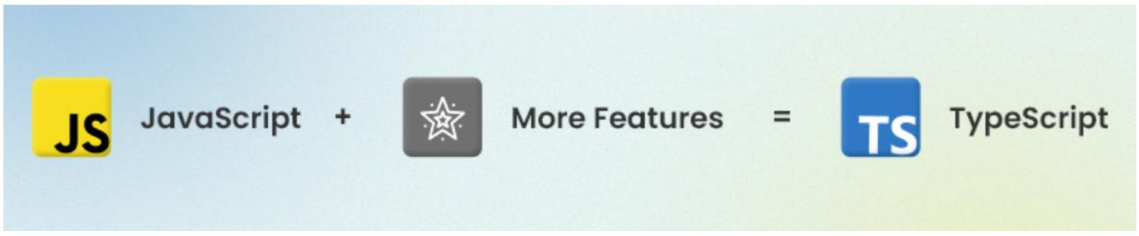


Figure 1. Typescript [1.]

The relationship between TypeScript (TS) and JavaScript (JS) can be seen in Figure 1, where TS is shown as a superset of JS. Although TS inherits all the features of JS, it cannot compile all code to JS. TS requires the addition of type data for each function, variable, and other elements in order to compile. This approach provides several benefits, such as preventing errors that are not detectable when using JS and making it easier to identify and fix problems. With TypeScript, developers can write more robust and reliable code, and debug issues more efficiently.

```
let a = 'a'; a = 1; // throws: error TS2322: Type '1' is not assignable to
type 'string'.
```

Figure 2. JavaScript example [1.]

Based on Figure 2, variable A has been initialized as a string type without being explicitly declared in TypeScript. Despite not being declared with a specific type, TypeScript is capable of inferring the type of data for variable A. If any other type of data is attempted to be assigned to A, TypeScript will raise an error due to a type mismatch.

JavaScript was not originally designed for building large or complex systems and applications. However, with the development of TypeScript, many have reconsidered this stance. TypeScript has gained widespread adoption and is now being used more frequently than JavaScript in certain contexts.

2.3 React JS

ReactJS is a popular open-source framework and library that was originally developed in JavaScript by Facebook. However, it is now also available in TypeScript. By using ReactJS, developers can create modern websites that are faster, more interactive, and efficient, with a reduced amount of code compared to older technologies used in the industry five years ago. Many popular companies such as Facebook, Instagram, Netflix, and others have built their websites and applications using ReactJS. The framework supports several common features, including JSX, unidirectional data flow, and a virtual Document Object Model (DOM). These features contribute to the efficiency and ease of use of the framework, making it a popular choice among developers today. [3].

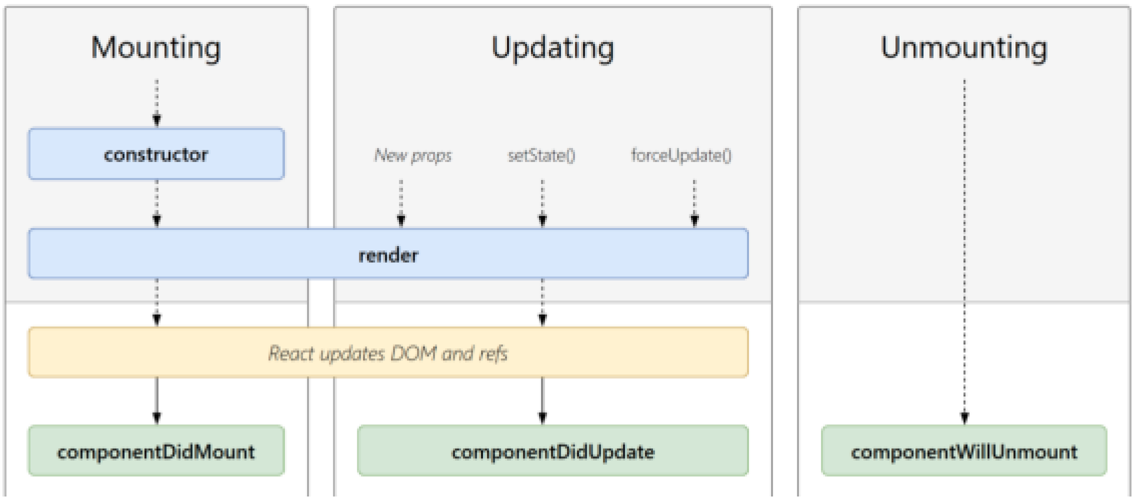


Figure 3. React lifecycle [6.]

ReactJS has a number of built-in methods that are called during the lifecycle of a component. These methods are divided into three categories: mounting, updating, and unmounting as can be seen in Figure 3.

The mounting phase starts when a new component is created and inserted into the DOM. The first method that is called is the constructor. This is where you initialize the state of the component and bind any event handlers. After the constructor, the render method is called, which returns the JSX code that will be rendered on the page. Finally, the “componentDidMount” method is called, which is where you can make API calls and perform other tasks that require the component to be fully mounted.

Once the component is mounted, it can be updated. The updating phase is triggered when the component receives new props or its state changes. The first method that is called during the updating phase is static “getDerivedStateFromProps”, which is used to update the state based on the new props. After that, the “shouldComponentUpdate” method is called, which allows you to control whether the component should be updated or not. If it returns false, the updating stops. If it returns true, the render method is called again to update the component's UI. Finally, the “componentDidUpdate” method is called, which is where you can perform any side effects that need to happen after the component updates.

The final phase is unmounting, which happens when the component is removed from the DOM. The “componentWillUnmount” method is called during this phase, and it's where you can perform any cleanup tasks such as clearing intervals or cancelling API calls. After this method is called, the component is completely removed from the DOM and its memory is freed up.

2.4 Material UI (MUI)

MUI is an open-sourced React component library. It has a prebuild components that we can use for our project. By using it, we will reduce the amount of time we need to create a component from scratch and style it, also we bring a high quality, digital experience for the customers and make the work easier for front-end developers. Here are some examples of prebuild components [9].

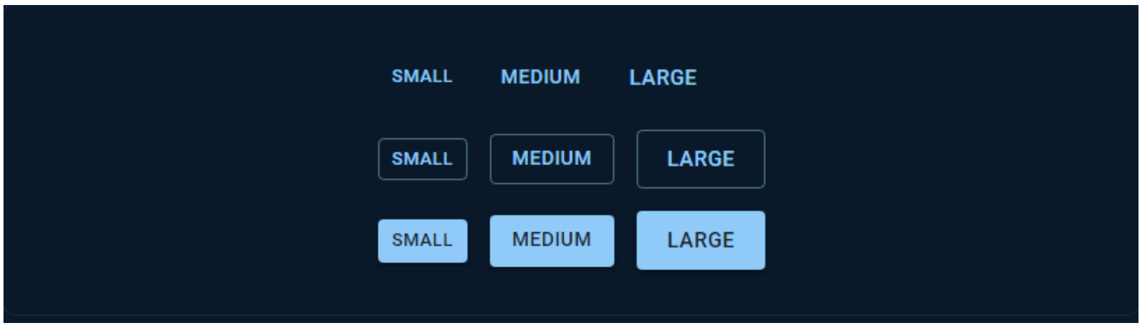


Figure 4. MUI Buttons [9.]

Figure 4 shows a set of buttons that can be created using MUI. The first row is button with normal configuration. Second row are buttons with the outline configuration which will have an addition border outside. The last row are buttons will blue filled background which will fill all the empty space inside the button, and you can change config more for each of them with color, size, font, etc

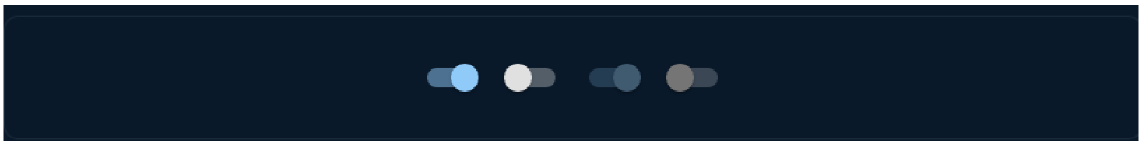


Figure 5. Mui Switches [9.]

Figure 5 shows switches also created by MUI with different state.

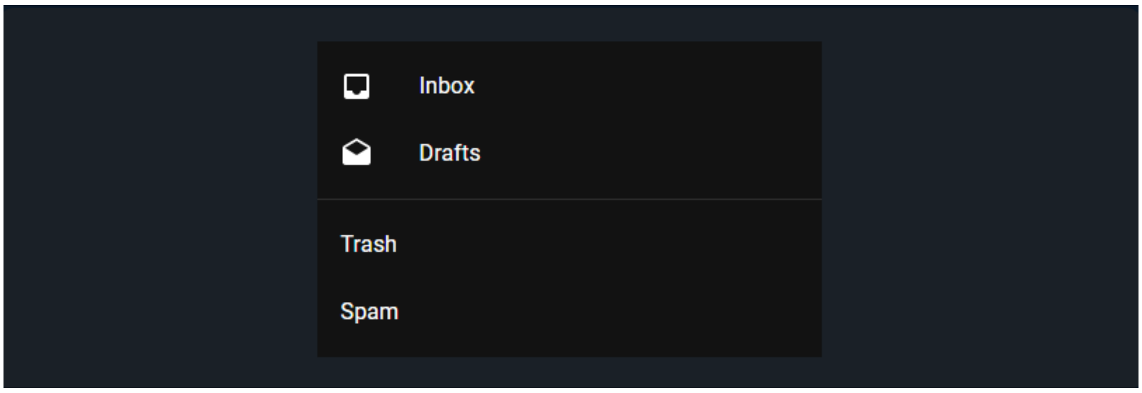


Figure 6. Mui List [9.]

Figure 6 shows a list also created by MUI with two main items and two sub.

2.5 Capacitor JS

Capacitor JS is an open-source cross-platform project that allows running web-based applications on various devices such as mobiles, tablets, and more. The primary objective of this project is to develop a system that can run smoothly on tablets. The process involves building a web application with React and adding styles for both web and tablet interfaces. Once completed, the entire application will be wrapped with Capacitor JS and tested on both Android and iOS platforms.

2.6 Rest API

For those who are in ICT industry now, this definition will not be the first time hearing it. First, API stand for application programming interface what define rules you must follow to communicate with other software systems. Each software system normally will have their own APIs. In real world, you can think it as connection between the client and resources. For example, a customer wants to use a product of your company, so she signed up for an account and buys that product, so how do the company know that they have a customer want to buy their product? Well, they do not, the customer once signing up, on the other line, the data from the customer will be sent from her local machine to the system of the company and the system will response back by sending an email of confirmation. By that mean of data communication, they can get in touch with each other and update the status of the product.

About REST, it is a software architecture that imposes condition how API should work. API can be developed using difference architecture. APIs that follow REST architecture were called REST API or RESTful API. In this project, a RESTful API will be built to play a bridge between the company and 3rd party company that provides the resource. [13].

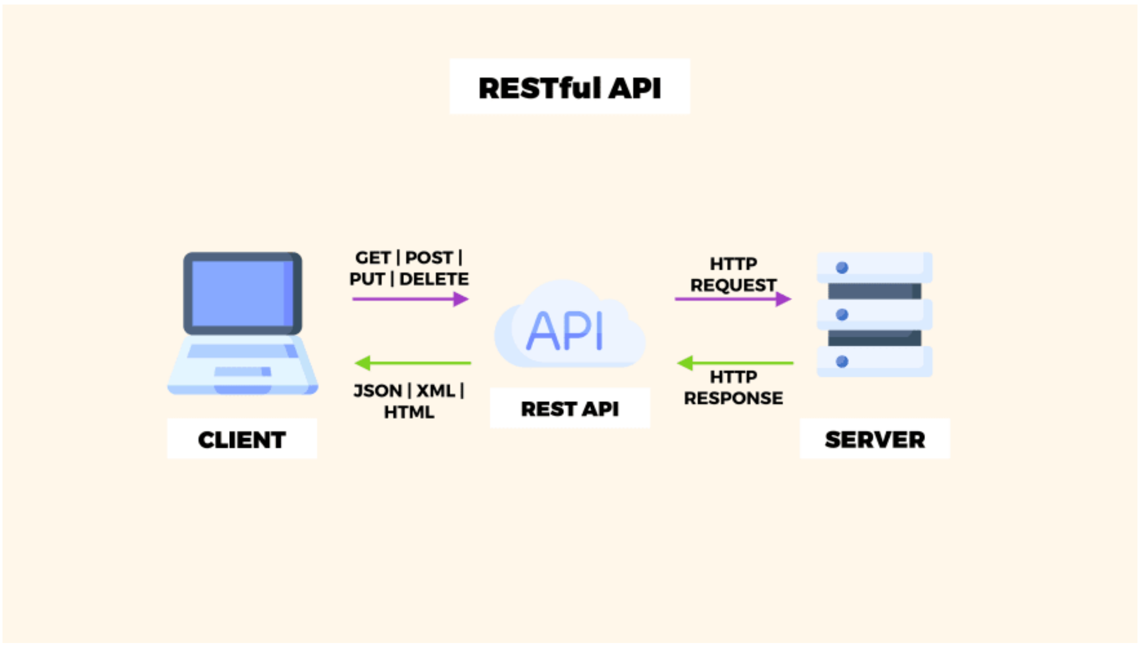
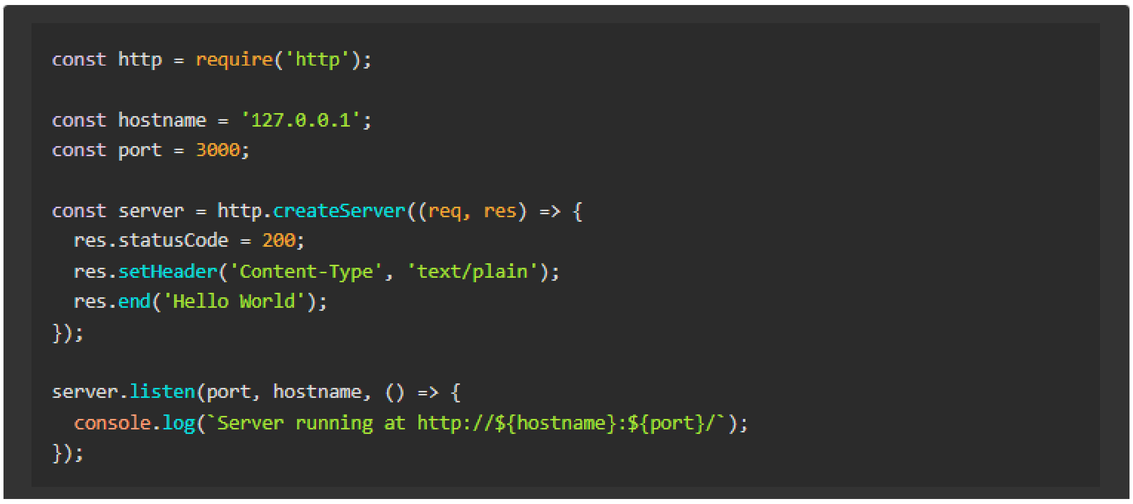


Figure 7. Rest API workflow [4.]

Figure 7 describes the typical flow of a client-server interaction using APIs (Application Programming Interfaces). The client, using basic HTTP methods such as GET, POST, DELETE, and PUT, makes a request to an API server. The API server then forwards this request to a real server that contains the desired data or functionality. The real server processes the request and generates an HTTP response, which is sent back to the API server. The API server then takes this response and generates a response of its own, typically in JSON, HTML, or XML format. This response is then sent back to the client, completing the request-response cycle. This process allows clients to access the functionality and data of a remote server through a standardized API, without needing to understand the internal workings of the server or the protocols used to communicate with it.

2.7 NodeJs

NodeJs is an open-source development platform to run JavaScript code server-side. It was designed building scalable application.



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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figure 8. Nodejs example [15.]

Figure 8 show "hello world" example, many connections can be handled concurrently. Upon each connection, the call-back is fired, but if there is no work to be done, Node.js will sleep. [15].

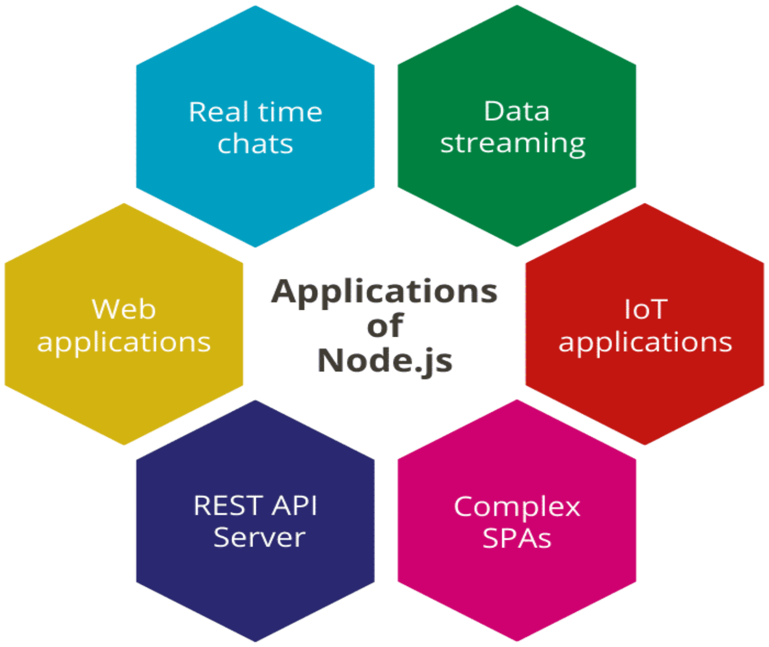


Figure 9. Nodejs's Application [11.]

Figure 9 displays a diverse range of Node.js applications that can be built using this technology. One of the most common types of Node.js applications is web applications, which can be used to create dynamic and responsive websites. Additionally, Node.js can be used to build REST API servers, which can be used to provide data and services to other applications. Another type of Node.js application shown in the image is complex single-page applications (SPAs), which can provide a seamless user experience by loading and updating content dynamically without requiring a page refresh.

In addition to web and API applications, Node.js can also be used to build IoT applications. These applications can communicate with a wide range of devices and sensors and can be used to collect and analyze data from these sources. Node.js can also be used to build real-time chat applications, which require fast and reliable communication between multiple users. Finally, Node.js can be used to build data streaming applications, which can process and analyze large volumes of data in real-time and provide insights and analytics to users.

To run React generate command, Nodejs is needed to be installed in your local machine. A detail tutorial can be found at TutorialsTeacher [18]. LTS is recommended as it will have support for a longer period which help us avoiding some unpredictable problems.

2.8 ExpressJs

ExpressJS is a popular open-source framework developed on top of NodeJS that enables developers to build dynamic applications across multiple platforms. It offers a wide range of features that make it a popular choice for a variety of projects. Not only is it free to use and easy to learn, but it is also fast and efficient, which saves time during development.

One of the major advantages of using ExpressJS is that it is highly customizable, allowing developers to tailor it to meet their specific needs. As it sits on top of NodeJS and manages routes and servers, it is important to ensure that connections are not redundant, which can lead to decreased performance. By optimizing connections and ensuring that they are efficient, developers can maximize the performance of their applications.

2.9 Nestjs

Nest is a powerful framework that enables developers to build efficient and scalable server-side applications in Node.js. It utilizes progressive JavaScript and is fully compatible with TypeScript, while still allowing developers to use pure JavaScript. One of the key strengths of Nest is that it seamlessly integrates elements of Object-Oriented Programming, Functional Programming, and Functional Reactive Programming.

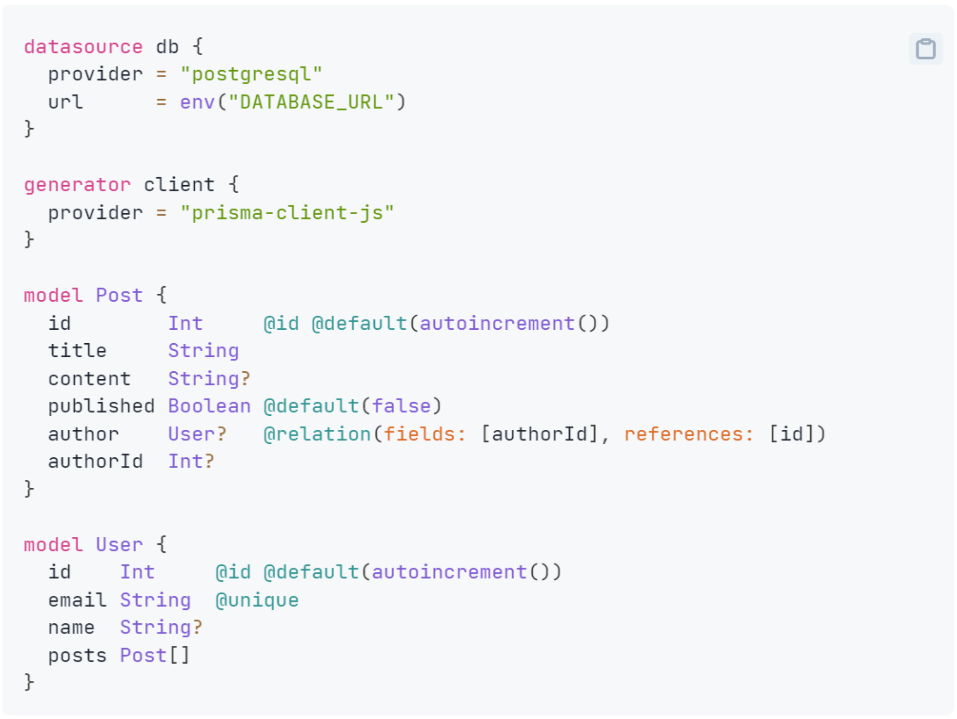
Nest is built on top of robust HTTP server frameworks like Express, which is the default option, but also offers the flexibility to configure. By abstracting away much of the complexity of these frameworks, Nest allows developers to focus on building high-quality applications and exposes the underlying APIs to enable the use of a wide range of third-party modules.

In addition to these features, Nest offers a wide range of tools and libraries to support developers in building reliable and maintainable applications. For example, it provides a powerful dependency injection system, comprehensive testing utilities, and support for real-time applications through WebSockets. Overall, Nest is an excellent choice for developers who want to build modern, scalable, and robust server-side applications in Node.js.

2.10 Prisma

Prisma is an open source next-generation ORM. It can be used in any Node.js or TypeScript backend (including serverless applications and microservices). It is used as an alternative to writing plain SQL or using another database access tool such as SQL query builders (like knex.js) or ORMs (like TypeORM and Sequelize). Prisma currently supports PostgreSQL, MySQL, SQL Server, SQLite, MongoDB and CockroachDB. [20].

The Prisma schema allows developers to define their application models in an intuitive data modelling language. It also contains the connection to a database and defines a generator:



```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String?
  published Boolean @default(false)
  author    User?   @relation(fields: [authorId], references: [id])
  authorId  Int?
}

model User {
  id    Int     @id @default(autoincrement())
  email String  @unique
  name  String?
  posts Post[]
}
```

Figure 10. Prisma schema [20.]

The example schema shown in Figure 10 consists of three components: a data source that specifies the database, a generator that instructs Prisma to generate both the Prisma Client and the data model used to define the application's models.
After defining the data model, Prisma Client can be generated to provide CRUD operations and other queries for the models. With TypeScript, full type-safety is provided for all queries, including when retrieving only subsets of a model's fields.

2.11 Sqlite

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects. [21].

SQLite has a lot of usage since it is scalability, concurrency, centralization, and control. Although it is not directly comparable with client/server SQL databases engines such as MySQL, Oracle, PostgreSQL, etc. SQLite strives to provide local data storage for individual applications and devices.

2.12 Socket IO

Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.

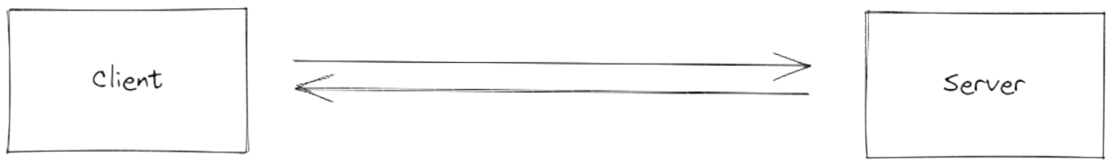


Figure 11. Socket Io basic flow [22.]

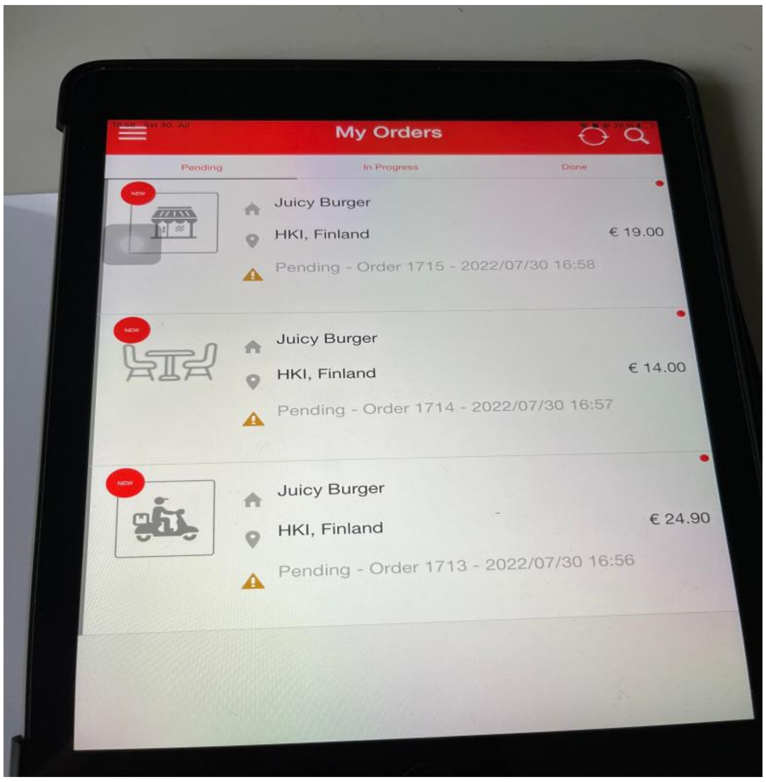
Figure 11 describe the basic flow of socket IO. It involves the client establishing a connection with the server using a WebSocket or HTTP long-polling connection. Once the connection is established, the client can send events to the server using Socket.IO's emit() method, which can trigger additional events that can be sent back to the client. This allows for efficient and reliable real-time communication between the client and server without the need for the client to repeatedly poll the server for updates.

Overall, Socket.IO is a popular choice for a variety of applications that require real-time communication, such as chat rooms, online games, and collaborative document editing. [22].

3 OVERVIEW OF THE OLD SYSTEM AND LIMITATIONS

This section will provide some details about the old system, which will be demonstrated through figures and explanations presented below. The limitations of the old system will be discussed, as well as the pros and cons of the old system compared to the new one.

When an order comes in, the system displays it on the screen and sends a notification to the restaurant. However, there have been issues with the notification function not working as expected, which can result in the user missing orders and negatively impacting their business.



Juicy Burger
HKI, Finland
€ 19.00
Pending - Order 1715 - 2022/07/30 16:58

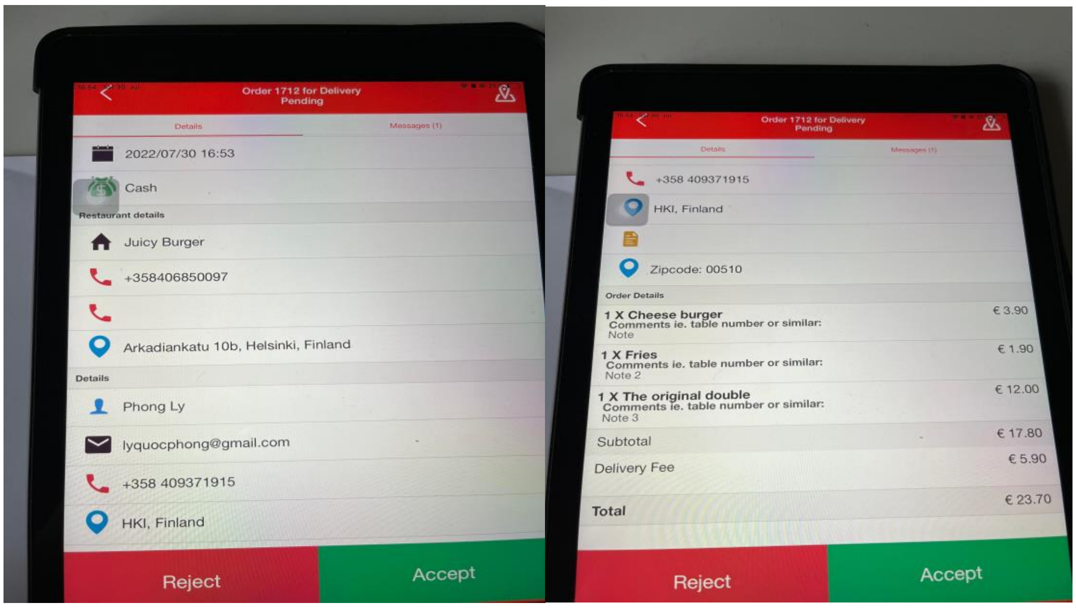
Juicy Burger
HKI, Finland
€ 14.00
Pending - Order 1714 - 2022/07/30 16:57

Juicy Burger
HKI, Finland
€ 24.90
Pending - Order 1713 - 2022/07/30 16:56

Figure 12. Incoming order (old system)

Figure 12 will provide a visual representation of what the pending screen looks like, but it is worth noting that specific orders cannot be viewed from this screen. While this may not be an issue on normal days with lower customer traffic, it can be a problem during weekends and holidays when restaurants tend to be busier. During these times, there is a higher likelihood that they may accidentally accept orders they are unable to fulfill. This could be due to changes in the restaurant's menu or running out of materials for certain dishes on busy days.

In the midst of a busy kitchen, it is not uncommon for mistakes to occur, and accepting an order that cannot be delivered can have negative consequences for both the restaurant and the customer. It can lead to a dissatisfied customer and may result in a loss of business. Therefore, it is important for restaurants to have reliable systems in place to avoid these situations and ensure that orders are only accepted if they can be fulfilled.


Order 1712 for Delivery
Pending

Details | Messages (1)

2022/07/30 16:53

Cash

Restaurant details

Juicy Burger

+358406850097

Arkadiankatu 10b, Helsinki, Finland

Details

Phong Ly

lyquocphong@gmail.com

+358 409371915

HKI, Finland

Reject | Accept

Order 1712 for Delivery
Pending

Details | Messages (1)

+358 409371915

HKI, Finland

Zipcode: 00510

Order Details

1 X Cheese burger Comments ie. table number or similar: Note	€ 3.90
1 X Fries Comments ie. table number or similar: Note 2	€ 1.90
1 X The original double Comments ie. table number or similar: Note 3	€ 12.00
Subtotal	€ 17.80
Delivery Fee	€ 5.90
Total	€ 23.70

Reject | Accept

Figure 13. Pending order screen

Figure 13 displays the details of an order, but it can only be viewed by scrolling down the screen. This can be time-consuming and frustrating for users, especially during busy periods. The system's layout may also cause staff to take longer to process an order and make a decision about whether to accept or reject it.

In fact, a survey was conducted among Munchi's client restaurants, and it was found that most of the staff complained about the user interface of the system. They felt that the system was tiring to use, particularly on busy days. These issues with the system's user interface and design are by far the biggest drawbacks of the old system. To improve the system's usability, it is important to consider the needs of the restaurant staff who will be using it. The user interface should be designed with ease of use in mind, minimizing the time it takes for staff to process orders and make decisions. This will not only improve the overall efficiency of the restaurant, but also enhance the experience of both staff and customers. In addition, incorporating features such as order tracking and real-time updates can provide greater transparency and control, making the system more user-friendly and effective.

4 IMPLEMENTATION

The process was carried on through the first step. It was made with the help of my lead development and my own effort. The difficulty I met through out process was solved based on my self-experience, self-observation, and my own way of thinking. However, sometimes my solution did not solve the whole problem, some small detail can lead to a bigger problem, that where Phong gave me his advice to enhance the solution better.

The project will include 2 parts front end and back end. The front end is nearly done, currently it is being tested and optimized. Nevertheless, in the front-end section 4 steps will be gone through: UI design, architecture, coding, and testing. Each step will have time and goals that need to be achieved to go to the next step.

4.1 ARCHITECTURE

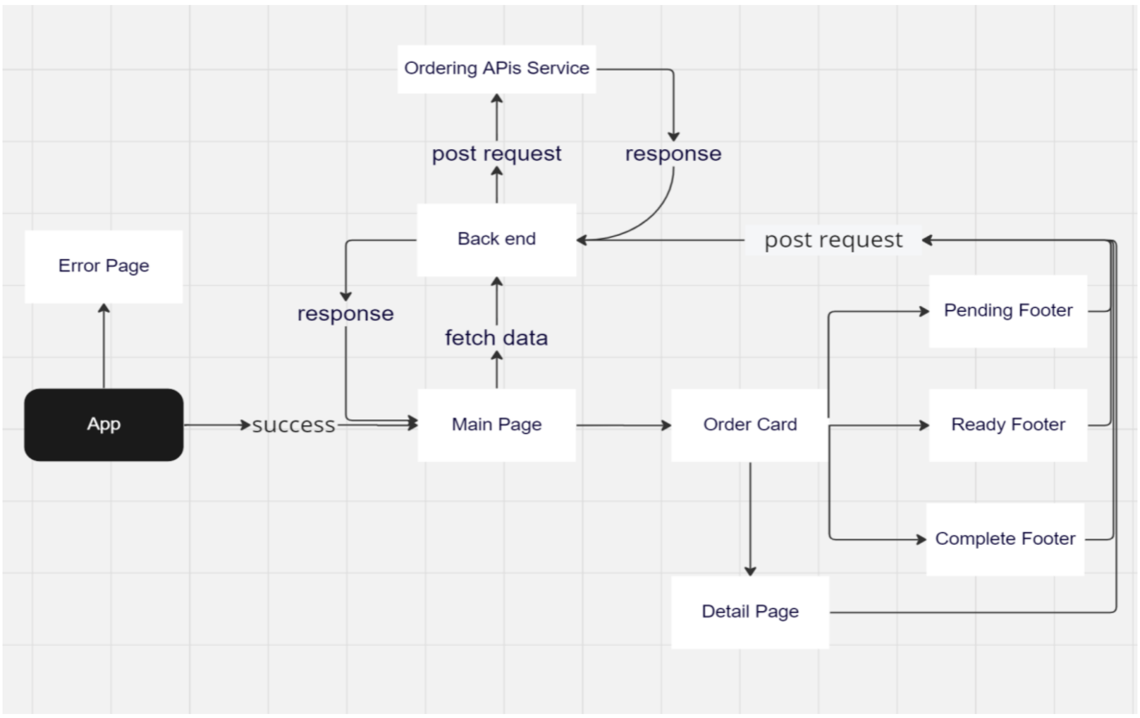


Figure 14. Application Architecture

Figure 14 illustrates the front-end architecture of the application, which is built using React. The application starts with the app.jsx or app.tsx file, depending on the chosen template. Three routes are initialized using react-router, which allows for dynamic routing in the web application. If the address is correct, the user will be redirected to the main page; otherwise, they will be directed to the error page. The main page has two main tasks: fetching data from the backend and passing it down to lower components. The reason for creating a backend instead of sending requests directly to the Ordering Apis service is that the application needs to recognize when a new order is received, and a backend is necessary for this purpose. To establish a connection between the backend and front end, webhook and Socket.io technologies are used. The main page passes the order data down to the order card, which displays similar upper content but different footer content depending on the order's status. For example, for a pending order, it will show the time to choose; for a ready order, it will display a countdown timer and a button to mark it as ready, and for a completed order, it will announce that it is complete. Each footer has a specific function to send a request to the backend to update the order's status and add preparation time. Finally, the detail page allows users to customize preparation time and check the order's details in a wider interface.

4.2 FRONT END

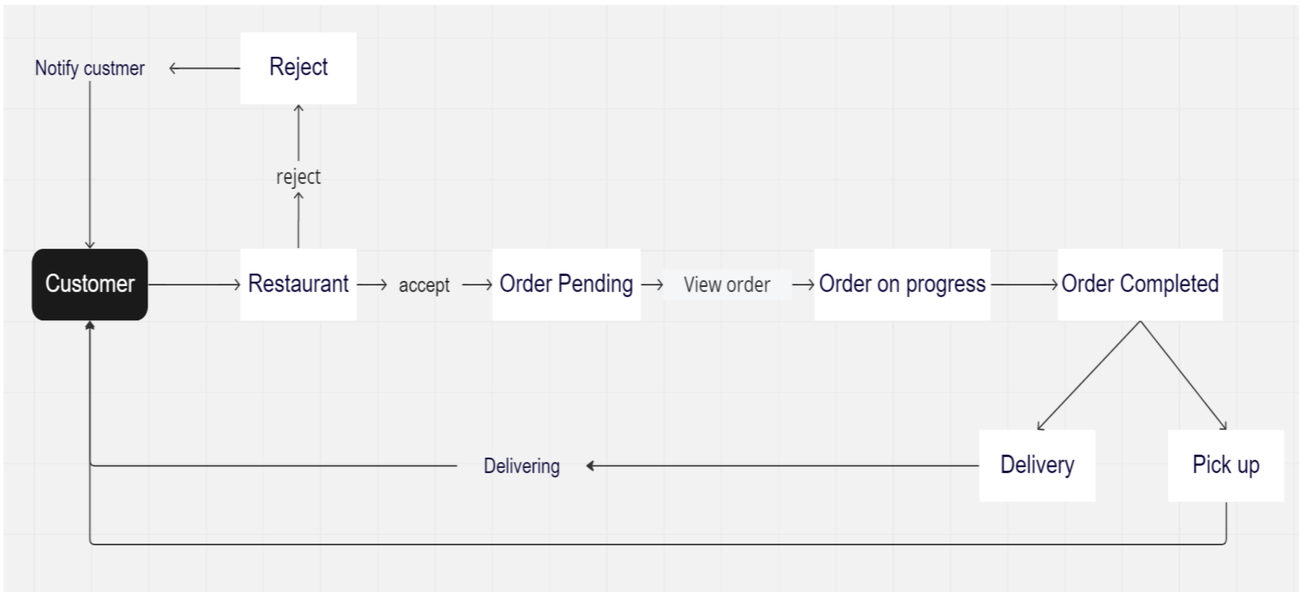


Figure 15. Website flowchart

Figure 15 show the flow of the system. The system will not support the eat in option. To start the process, the customer will first send the order by using a delivery application. More will be explained in the UI design below.

4.2.1 UI Design

Using Adobe XD, the design was built from scratch by the CEO Mario Henningson and Lead Developer Ly Quoc Phong. The user interface was made to be user friendly and easy to use. Therefore, a light soothing palette of colours was chosen.

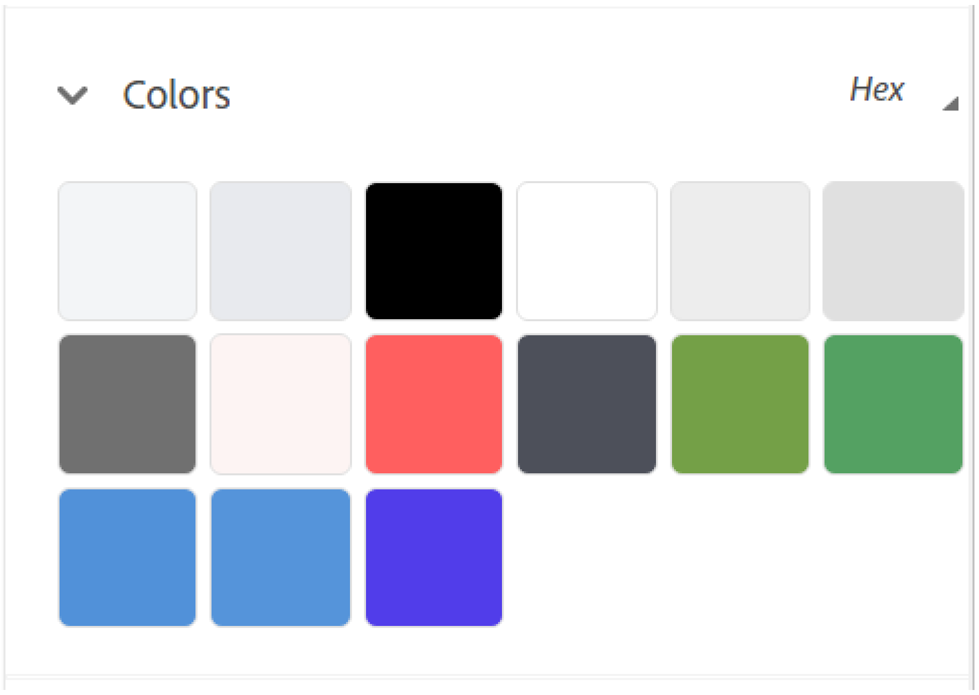


Figure 16. Color palette

Figure 16 show the palette of color. The primary color will be light grey with the code #E8EAEE. The main font will be DM Sans, Bold

The main screen will be divided into three columns: pending, accepted and ready. Every time a new order come and waiting to be processed will be shown in the pending column.

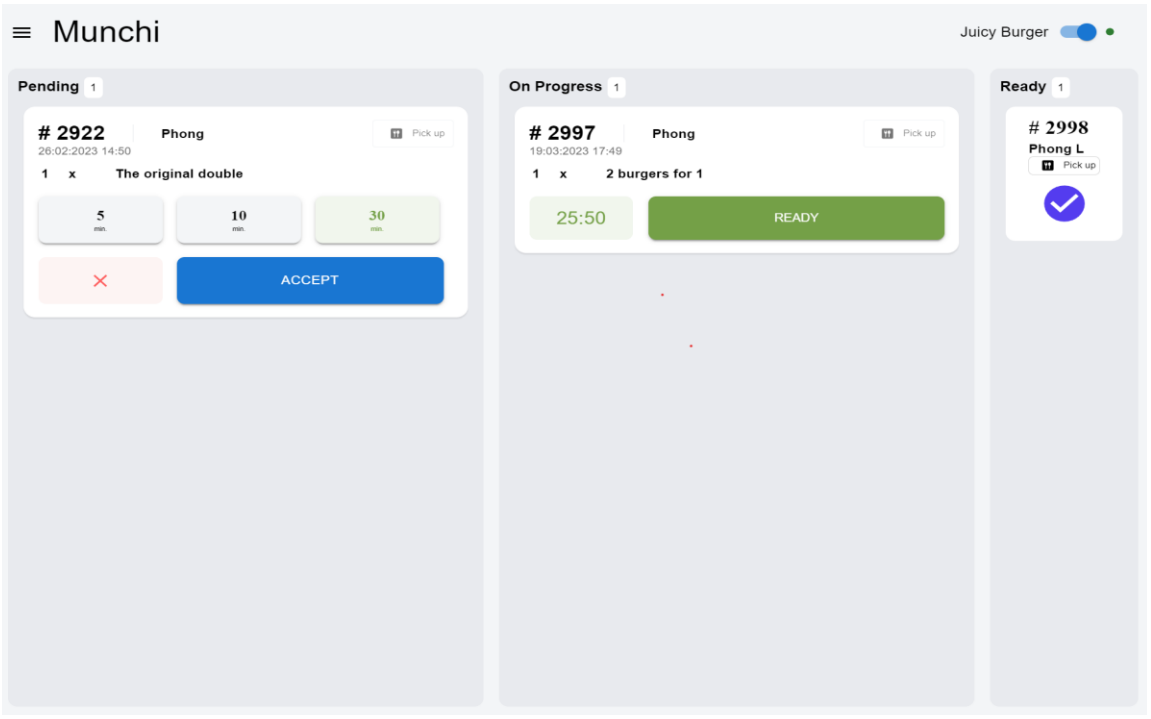
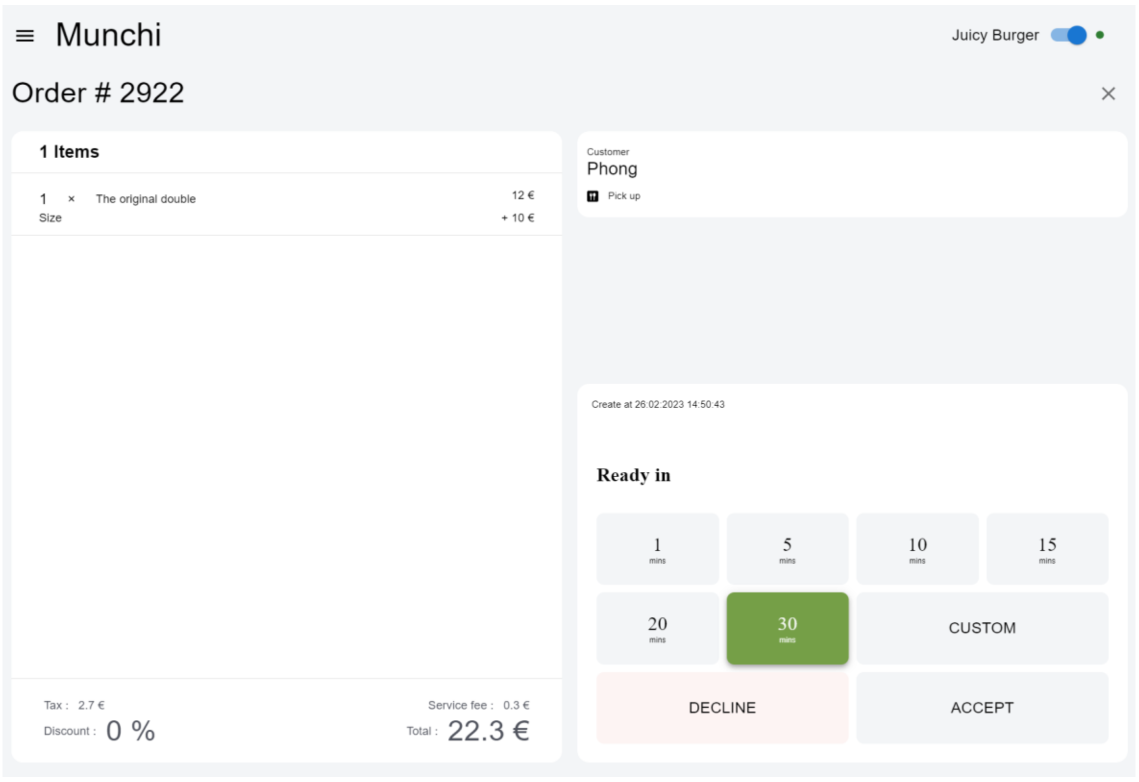


Figure 17. Main Screen

The main screen design of the application is shown in Figure 17, which features a tool bar in the header for accessing settings and other tabs. It also includes an order manager button for returning to the home page and a status indicator for displaying the restaurant's current state. The main content area is divided into three columns: the first column displays incoming orders, which are moved to the second column once the restaurant accepts them. The final column is used to confirm orders when the delivery is finished.

To confirm an order, first select a time from the "Pending" column. Once selected, the time button will change its background and text color. Then, click the "Accept" button to send a request to the backend to update the order status. The main screen will refresh and the order will be moved to the "On Progress" column. In this column, the order card will display a countdown timer based the selected time from the "Pending" column. When the remaining time is less than three minutes, the text color will turn red. Once the timer reaches zero, a dialog box will pop up to notify the user.

Alternatively, if you click or tap on the order card without selecting a time, it will direct you to the detail page


Munchi

Juicy Burger

Order # 2922

1 Items

1 × The original double 12 €
Size + 10 €

Tax : 2.7 €
Discount : 0 %
Service fee : 0.3 €
Total : 22.3 €

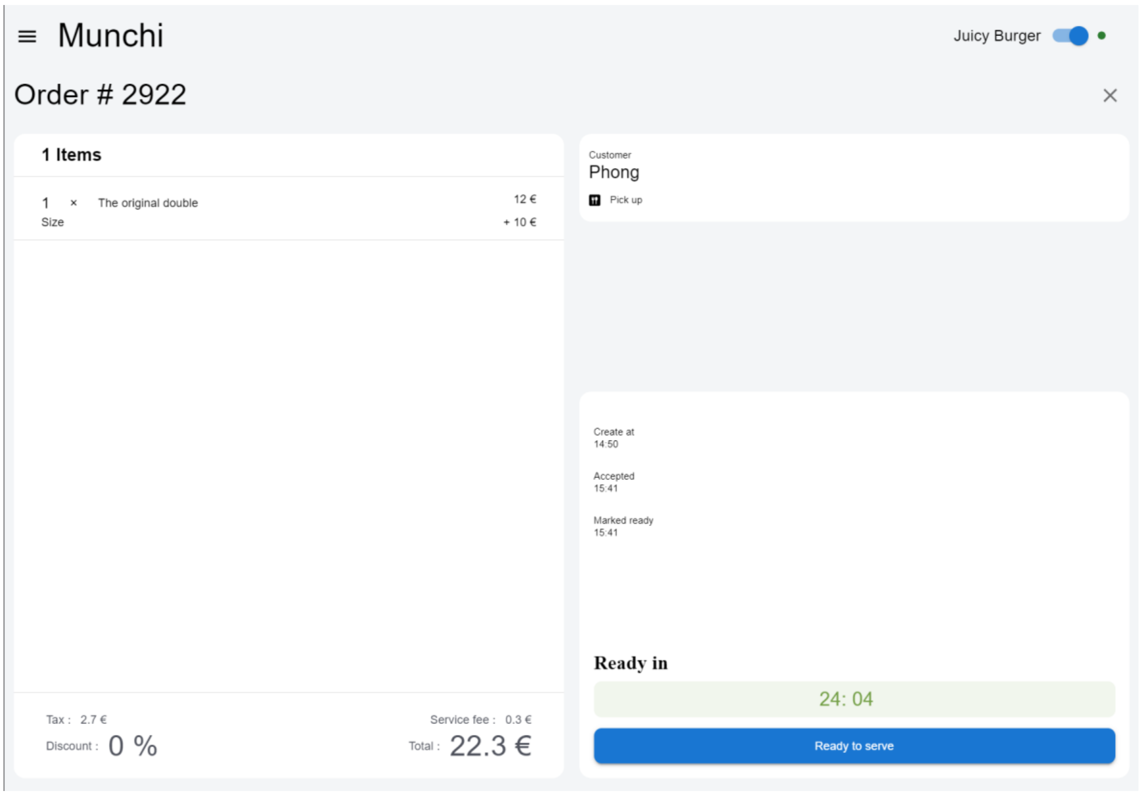
Customer
Phong
Pick up

Create at 26.02.2023 14:50:43

Figure 18. Detail Page

As shown in Figure 18, once you are on the detail page, the order data will be presented in a more user-friendly manner, with the detail order section being wider and displaying additional information. Additionally, users will have the option to customize the time by clicking the "Custom" button.

Similar to the main page, the footer on the detail page will also change based on the status of the order. For example, if the order is still pending, the footer may display options to customize the order or cancel it. If the order is in progress, the footer may show the countdown timer and the option to track the order's status. Once the order is ready, the footer will display the single button to return the user to the main page. This dynamic footer provides users with quick access to relevant information and actions, making the ordering process more intuitive and streamlined.


≡ Munchi

Juicy Burger

Order # 2922

1 Items

1 × The original double 12 €
Size + 10 €

Tax : 2.7 €
Discount : 0 %
Service fee : 0.3 €
Total : 22.3 €

Customer
Phong
Pick up

Create at
14:50

Accepted
15:41

Marked ready
15:41

Ready in
24: 04

Ready to serve

Figure 19. Detail Page (Accepted)

As shown in Figure 19, once you are on the detail page, the order data will be presented in a more user-friendly manner, with the detail order section being wider and displaying additional information. Additionally, users will have the option to customize the time by clicking the "Custom" button. Similar to the main page, the footer on the detail page will also change based on the status of the order. For example, if the order is still pending, the footer may display options to customize the order or cancel it. If the order is in progress, the footer may show the countdown timer and the option to track the order's status. Once the order is ready, the footer will display the single button. This dynamic footer provides users with quick access to relevant information and actions, making the ordering process more intuitive and streamlined.

Moreover, like the main page, the detail page will also include a countdown timer to provide users with real-time information on the status of their order. The button on the detail page will have a similar functionality to the one on the main page, allowing users to confirm their order once the details have been reviewed and customized. In addition, the detail page may also include options for users to provide additional information for the restaurant. This information can be helpful in improving the overall user experience and may be used to inform future updates or changes to the platform.

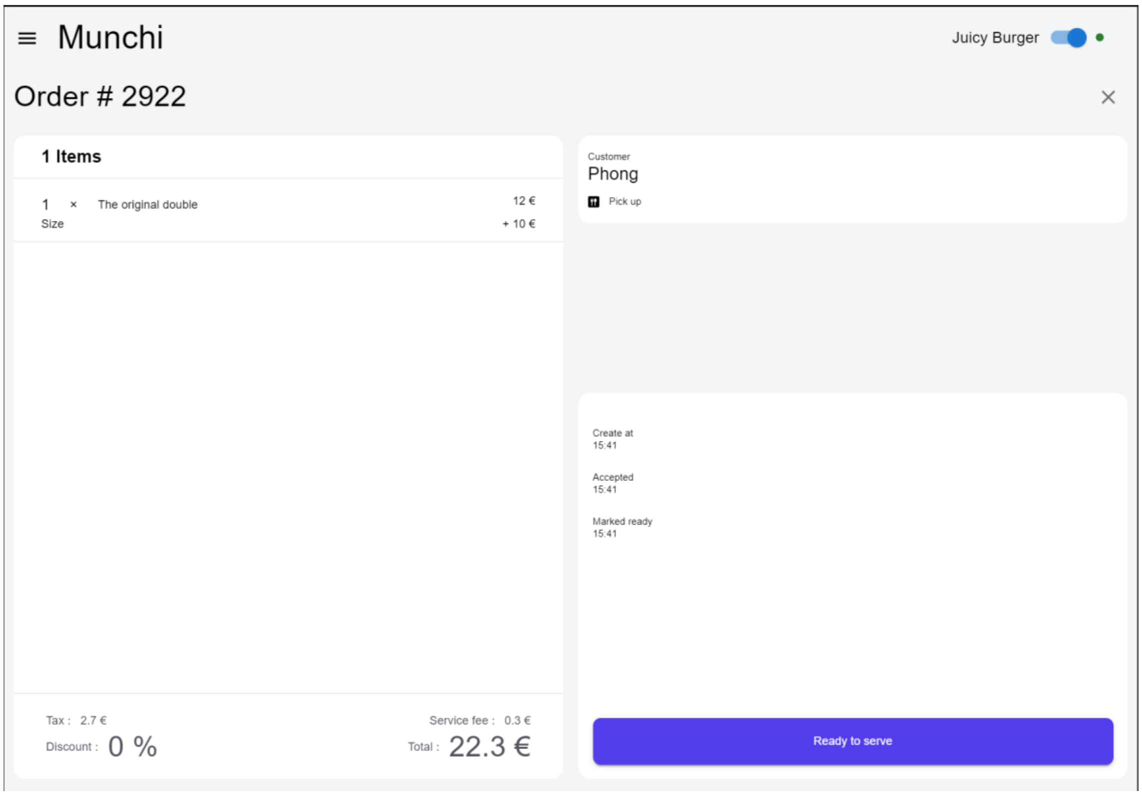


Figure 20. Detail page (Completed)

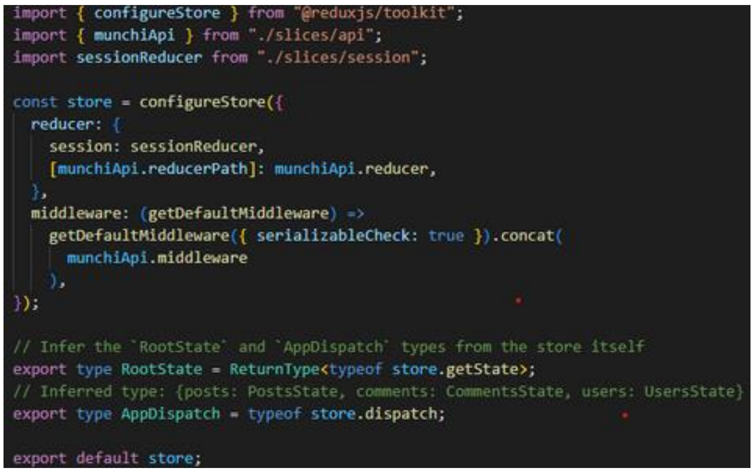
To provide a seamless user experience, the platform includes a dynamic footer on the detail page that changes based on the status of the order. Once the order is ready, the footer will display a single button, as demonstrated in Figure 20, that when clicked, will bring the user back to the main page once the order has been shipped or picked up by the customer. This feature ensures that users can easily navigate the platform and have quick access to relevant information and actions throughout the ordering process.

4.2.2 Code

The merchant application that will be created in this project will be a dynamic and responsive web application built using a variety of modern technologies. The primary front-end framework used will be React, a powerful JavaScript library for building user interfaces. CSS and HTML will be used to define the application’s visual style and layout, while Material UI will be used to provide a rich and customizable set of UI components. To enable real-time communication between the client and server, Socket.IO will be used as the primary web socket library. To provide audio feedback and sound effects, Howler.js will be used, and Capacitor.js will be used to facilitate the application's integration with native mobile devices. By combining these technologies, the resulting merchant application will provide a dynamic and engaging user experience, with seamless integration between the client and server sides of the application.

The application will feature login pages, a business page, and a dashboard page as its main page. If a user attempts to navigate to a non-existent page, they will be redirected to an error page. The user will begin by logging in and selecting their business. Although a user can have multiple businesses, this feature is not applicable due to the involvement of a third-party company. After confirmation on the business page, the user will be directed to the dashboard page. To notify the restaurant that they are online, the user can enable this status with the switch button located at the top right of the screen. Once the dashboard page has been rendered, it will automatically fetch data from the server. Pending orders will be displayed in the "Pending" column. When an order comes in, the application will listen for it using the Socket.IO client, which in combination with React hooks and Howler.js, will trigger a continuous ringing sound to notify the user until the dialog is closed. Simultaneously, the dashboard will refetch data from the server and re-render.

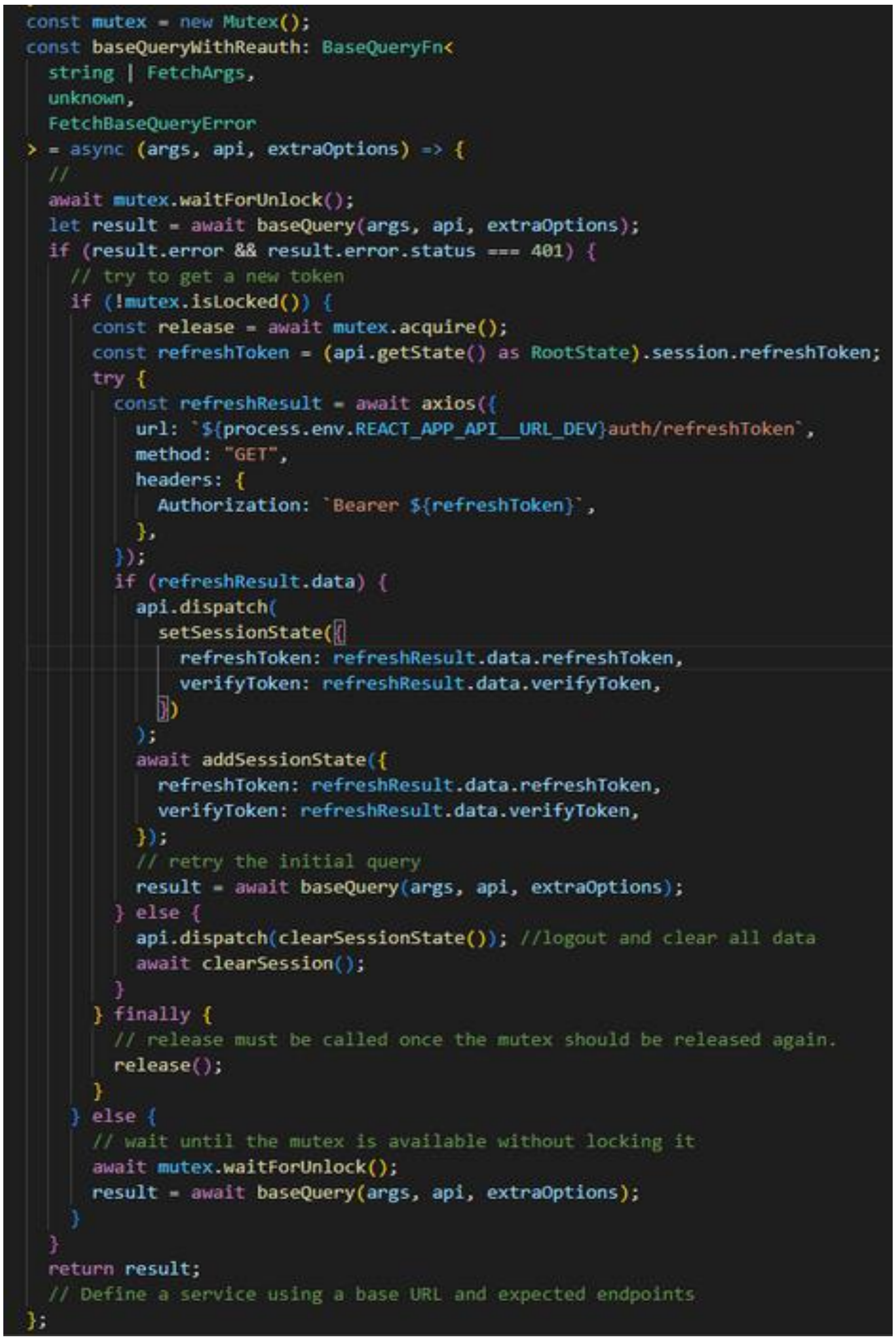
For state management and data fetching, Redux Toolkit will be utilized to efficiently re-render components.



```
import { configureStore } from "@reduxjs/toolkit";
import { munchiApi } from "./slices/api";
import sessionReducer from "./slices/session";

const store = configureStore({
  reducer: {
    session: sessionReducer,
    [munchiApi.reducerPath]: munchiApi.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({ serializableCheck: true }).concat(
      munchiApi.middleware
    ),
});

// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<typeof store.getState>;
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}
export type AppDispatch = typeof store.dispatch;

export default store;
```

Figure 21. Redux store

In Figure 21, the registered slices in the Redux store are depicted. The full code can be found in Appendix 1, with the session slice being the primary slice that manages the application's state. To maintain persistent data, the capacitor preference plugin is utilized to store data in the local storage of the website and other relevant platforms. Whenever an action is dispatched, both Redux and capacitor preference dispatch two actions simultaneously.

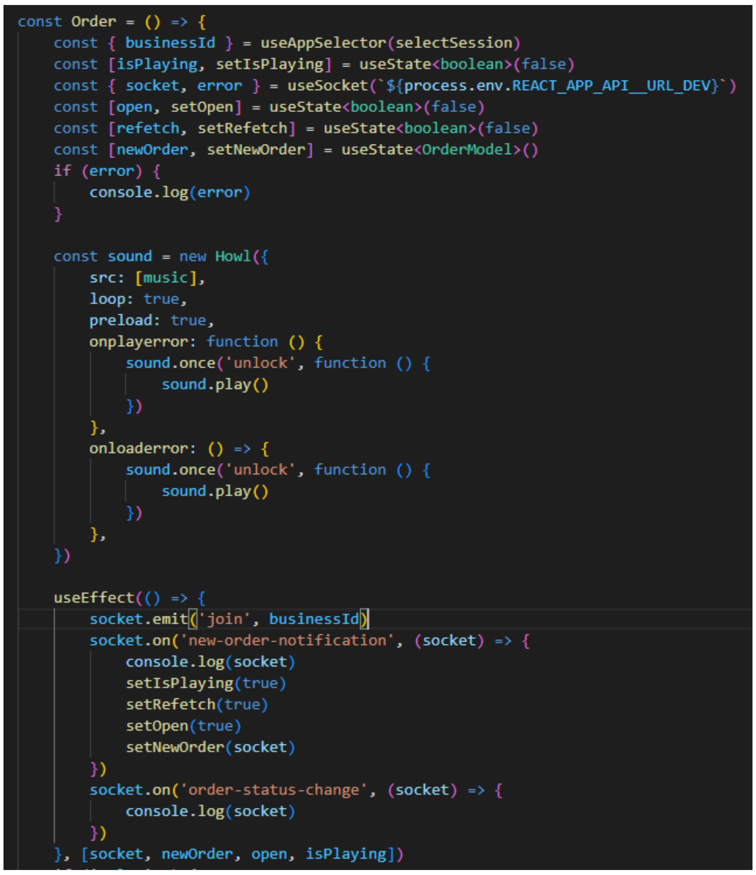


```
const mutex = new Mutex();
const baseQueryWithReauth: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError
> = async (args, api, extraOptions) => {
  //
  await mutex.waitForUnlock();
  let result = await baseQuery(args, api, extraOptions);
  if (result.error && result.error.status === 401) {
    // try to get a new token
    if (!mutex.isLocked()) {
      const release = await mutex.acquire();
      const refreshToken = (api.getState() as RootState).session.refreshToken;
      try {
        const refreshResult = await axios({
          url: `${process.env.REACT_APP_API__URL_DEV}auth/refreshToken`,
          method: "GET",
          headers: {
            Authorization: `Bearer ${refreshToken}`,
          },
        });
        if (refreshResult.data) {
          api.dispatch(
            setSessionState({
              refreshToken: refreshResult.data.refreshToken,
              verifyToken: refreshResult.data.verifyToken,
            })
          );
          await addSessionState({
            refreshToken: refreshResult.data.refreshToken,
            verifyToken: refreshResult.data.verifyToken,
          });
          // retry the initial query
          result = await baseQuery(args, api, extraOptions);
        } else {
          api.dispatch(clearSessionState()); //logout and clear all data
          await clearSession();
        }
      } finally {
        // release must be called once the mutex should be released again.
        release();
      }
    } else {
      // wait until the mutex is available without locking it
      await mutex.waitForUnlock();
      result = await baseQuery(args, api, extraOptions);
    }
  }
  return result;
  // Define a service using a base URL and expected endpoints
};
```

Figure 22. Automatic Re-authorized

The munchiApi reducer is responsible for data fetching, simplified by using the Redux useRTK query in Figure 22, the full code is presented at Appendix 2. By preparing the bearer token header for each request and defining the endpoint for different routes, data can be fetched effortlessly. The function is also equipped to automatically re-authorize when the token expires or a 401 unauthorized error occurs, returning a new token and automatically refetching the data. Additionally, it uses mutex to prevent multiple requests from taking the token.

The application has been designed to provide timely notifications to the user whenever a new order arrives. This is achieved by using a continuous ringing sound that is triggered using React hooks and Howler.js in conjunction with the Socket.IO client. Whenever a new order arrives, the Socket.IO client listens for the incoming order and triggers the continuous ringing sound to notify the user. This sound persists until the user acknowledges the notification by closing the dialog box. The use of React hooks and Howler.js ensures that the sound plays smoothly without any interruptions, providing a seamless experience for the user. By utilizing this method, the application ensures that the user is promptly notified of any new orders, allowing them to respond to them promptly and efficiently.



```
const Order = () => {
    const { businessId } = useAppSelector(selectSession)
    const [isPlaying, setIsPlaying] = useState<boolean>(false)
    const { socket, error } = useSocket(`${process.env.REACT_APP_API__URL_DEV}`)
    const [open, setOpen] = useState<boolean>(false)
    const [refetch, setRefetch] = useState<boolean>(false)
    const [newOrder, setNewOrder] = useState<OrderModel>()
    if (error) {
        console.log(error)
    }

    const sound = new Howl({
        src: [music],
        loop: true,
        preload: true,
        onplayerror: function () {
            sound.once('unlock', function () {
                sound.play()
            })
        },
        onloaderror: () => {
            sound.once('unlock', function () {
                sound.play()
            })
        },
    })

    useEffect(() => {
        socket.emit('join', businessId)
        socket.on('new-order-notification', (socket) => {
            console.log(socket)
            setIsPlaying(true)
            setRefetch(true)
            setOpen(true)
            setNewOrder(socket)
        })
        socket.on('order-status-change', (socket) => {
            console.log(socket)
        })
    }, [socket, newOrder, open, isPlaying])
```

Figure 23. Howler and Socket

Figure 23 displays the use of the useSocket package, the full code is presented at Appendix 3, which has been implemented in the useEffect hook to effectively monitor and receive incoming orders. This integration results in cleaner code and simplifies the process of order detection.

4.3 BACK END

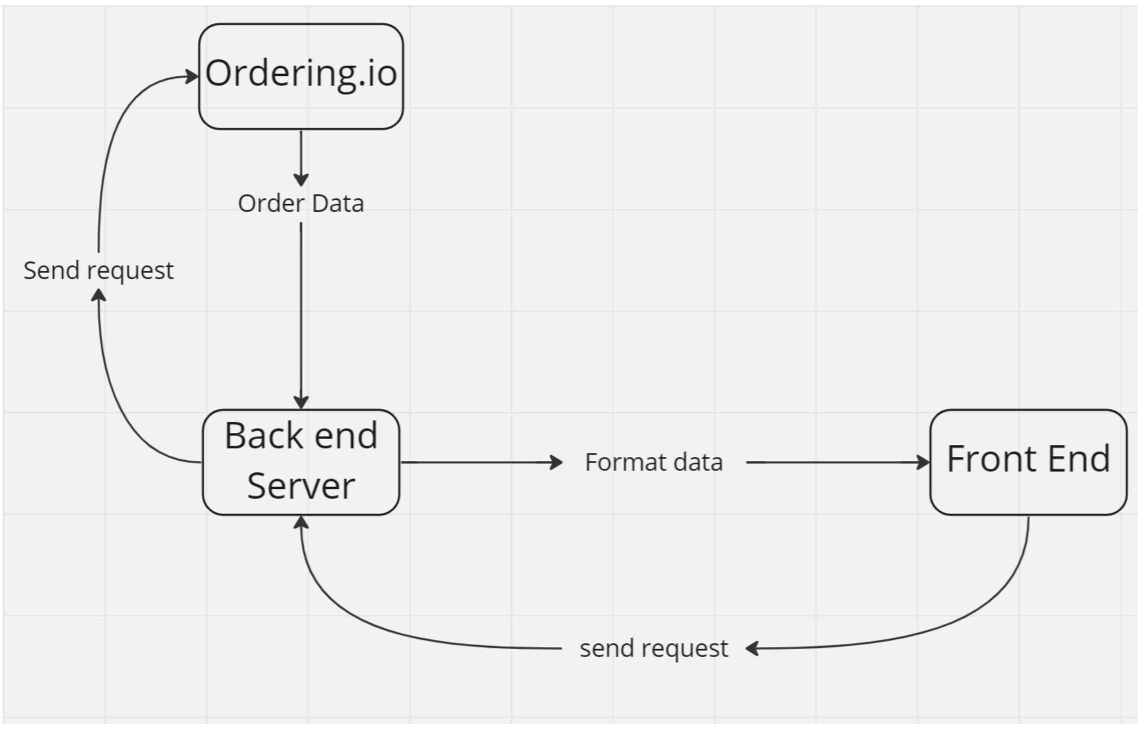
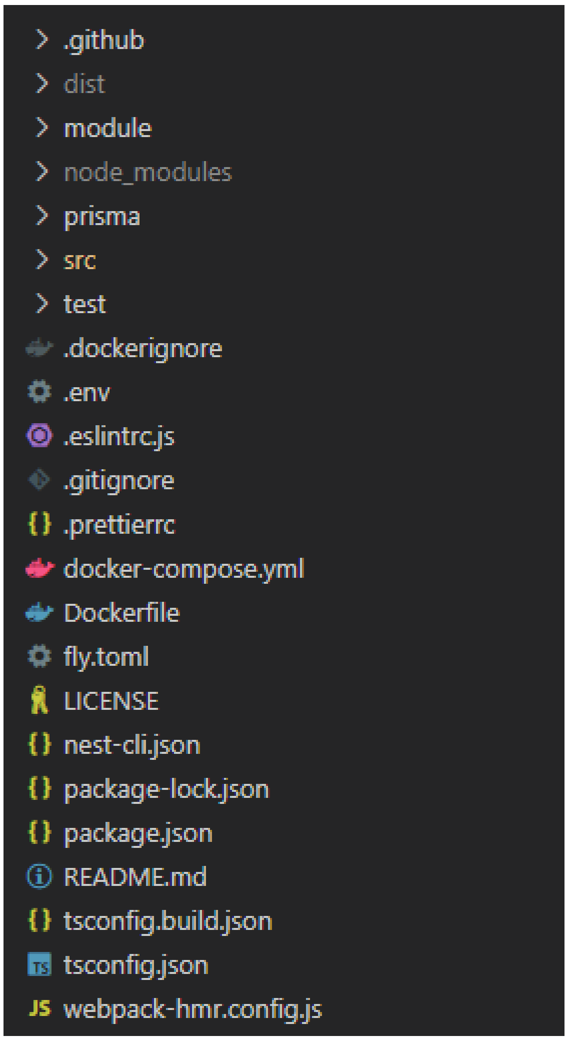


Figure 24. Back-end flow

The intermediary custom-built back end, as explained in Section 4.1 and illustrated in Figure 24, simplifies the information transmitted from the ordering APIs to the front end through data formatting. It also plays the crucial role of establishing connections with both the APIs and the front end, allowing it to receive real-time data from the APIs whenever a new order is placed and transmit it to the front end.

4.3.1 Code

The backend code has been developed using Nestjs (2.9) and Prisma (2.10) to build REST APIs (2.6) for connecting with the frontend.



```
> .github
> dist
> module
> node_modules
> prisma
> src
> test
.dockerignore
.env
.eslintrc.js
.gitignore
{} .prettierrc
docker-compose.yml
Dockerfile
fly.toml
LICENSE
{} nest-cli.json
{} package-lock.json
{} package.json
README.md
{} tsconfig.build.json
tsconfig.json
JS webpack-hmr.config.js
```

Figure 25. Back-end Code

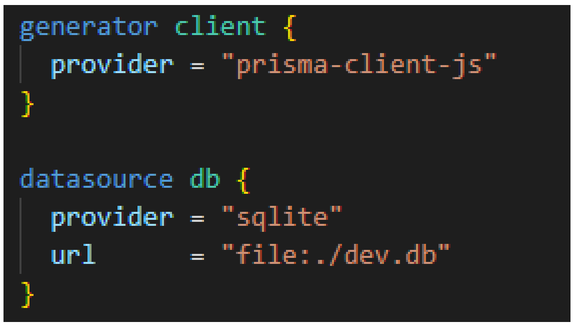
The entire code structure of the backend is depicted in Figure 25, with each folder representing routes for the APIs, consisting of three files: controllers, modules, and services. The APIs are documented using Swagger, which can be accessed through reference [23].

The most crucial part of the backend is the authentication routes, which require an access token to access other routes. Users can use this token for authorization and authentication to access the APIs. The other routes accept the token in the JWT format and proceed accordingly. However, as this backend is acting as a bridge with a third-party provider, it needs a database to store the access token for the provider's routes. Prisma is used for database connection management, which creates a connection to the database and data models by defining the schema. SQLite is used to store data in Prisma.

By far and the most important is the auth routes, to access other routes. An access token is needed which will allow users to access the API to perform authorization and authentication. Other routes will accept the token under JWT format and proceed to it. Unfortunately, as this backend is used as a bridge with the 3rd party provider. Therefore, it needs a database to store the access token for the routes from the provider. For database connection, Prisma is used to manage this operation, by defining the schema, it creates a connection the database and data models. In Prisma, SQLite will be used to store data.

This is a schema definition written in Prisma Schema Language, which is used to define the structure and relationships between tables in a database. The schema defines three models: Session, User, and Business.

The first part of the schema defines the data source and generator client (Figure 26).

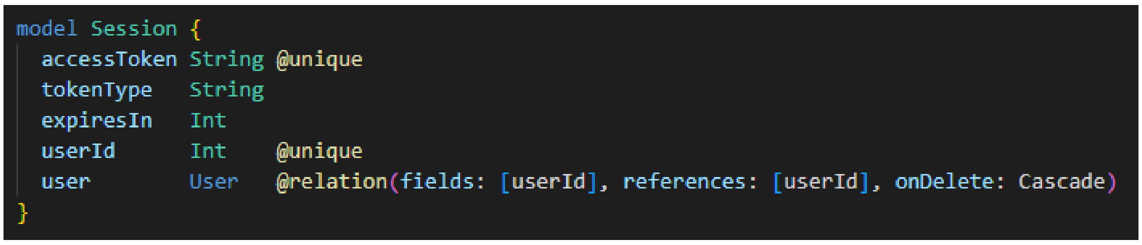


```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}
```

Figure 26. Prisma data source

Figure 26 specifies that the Prisma client should be generated using the prisma-client-js provider, and that the data source should use SQLite as the database provider with a file named dev.db.

Next, the Session model is defined with four fields (Figure 28).

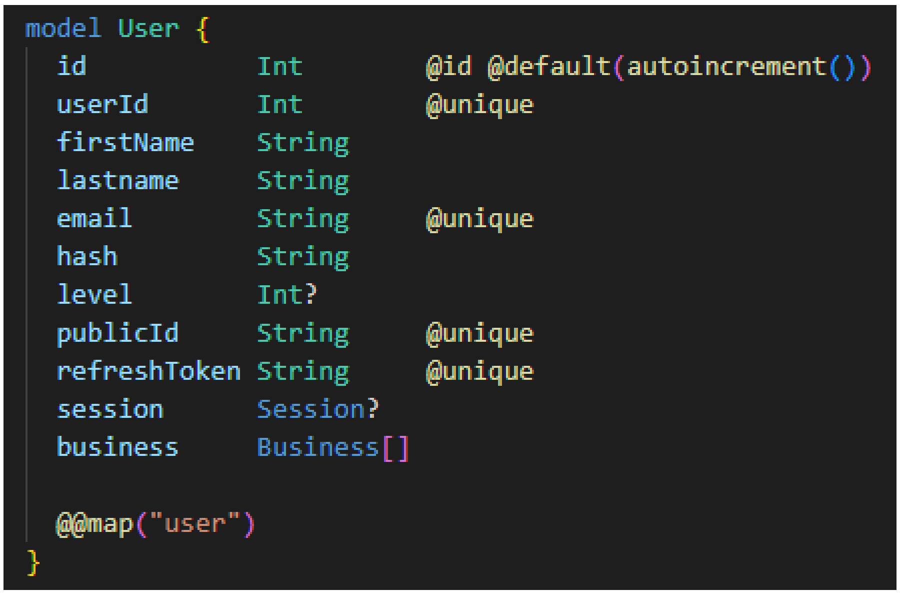


```
model Session {
  accessToken String @unique
  tokenType   String
  expiresIn   Int
  userId      Int    @unique
  user        User   @relation(fields: [userId], references: [userId], onDelete: Cascade)
}
```

Figure 27. Session Model

Figure 27 depicts the Session model, which comprises an “accessToken” field of type String that is unique, a “tokenType” field of type String, an “expiresIn” field of type Int, and a “userId” field of type Int that is also unique. Additionally, the Session model establishes a relationship with the User model by referencing the “userId” field in User via the “userId” field in Session.

The User model is defined next, with several fields and a relationship to the Session and Business models.

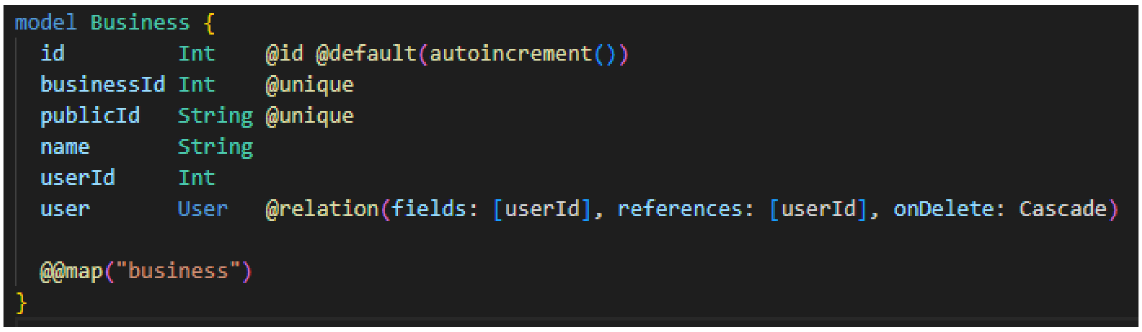


```
model User {
  id           Int       @id @default(autoincrement())
  userId       Int       @unique
  firstName    String
  lastname     String
  email        String    @unique
  hash         String
  level        Int?
  publicId     String    @unique
  refreshToken String    @unique
  session      Session?
  business     Business[]

  @@map("user")
}
```

Figure 28. User Model

The User model (Figure 28) has an id field of type Int that is the primary key and auto-incremented, a “userId” field of type Int that is unique, a “firstName” field of type String, a “lastname” field of type String, an email field of type String that is unique, a hash field of type String, an optional level field of type Int, a “publicId” field of type String that is unique, a “refreshToken” field of type String that is unique, a relationship to the Session model that can be null, and a relationship to the Business model.

Finally, the Business model is defined with several fields and a relationship to the User model (Figure 29).



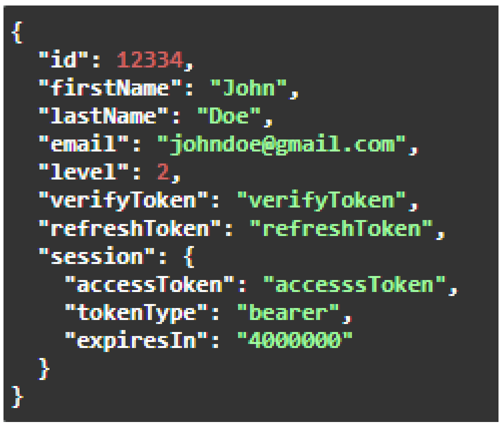
```
model Business {
  id         Int    @id @default(autoincrement())
  businessId Int    @unique
  publicId   String @unique
  name       String
  userId     Int
  user       User   @relation(fields: [userId], references: [userId], onDelete: Cascade)

  @@map("business")
}
```

Figure 29. Business Model

Figure 29 has fields for a unique ID, business ID, public ID, name, user ID, and a reference to a User object. The “businessId” and “publicId” fields are marked as unique.

The @@map directive is used to specify the name of the table that each model is mapped to in the database. In this case, the User model is mapped to a table called user, and the Business model is mapped to a table called business.

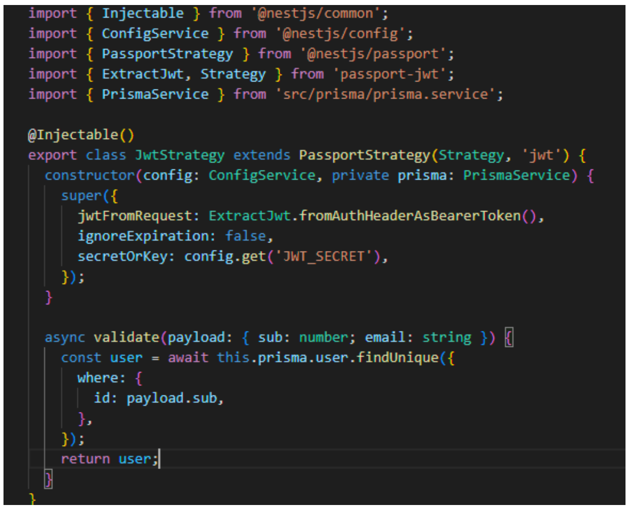
Overall, this schema defines the data models and relationships for an application that involves users, business entities, and sessions for authenticated users. The use of Prisma's DSL and Prisma client can help simplify database interactions and make them more efficient.



```
{
  "id": 12334,
  "firstName": "John",
  "lastName": "Doe",
  "email": "johndoe@gmail.com",
  "level": 2,
  "verifyToken": "verifyToken",
  "refreshToken": "refreshToken",
  "session": {
    "accessToken": "accesssToken",
    "tokenType": "bearer",
    "expiresIn": "4000000"
  }
}
```

Figure 30. Login Response

Upon successful user login, the backend responds with a message containing an access token, as depicted in Figure 30. This access token is then stored in Prisma for future reference. Additionally, a verify token is generated to authorize the API server system. A refresh token is also created to renew the verify token whenever it expires.



```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PrismaService } from 'src/prisma/prisma.service';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt') {
  constructor(config: ConfigService, private prisma: PrismaService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.get('JWT_SECRET'),
    });
  }

  async validate(payload: { sub: number; email: string }) {
    const user = await this.prisma.user.findUnique({
      where: {
        id: payload.sub,
      },
    });
    return user;
  }
}
```

Figure 31. JWT

In Figure 31, the process of extracting and decoding the JWT token using passport-jwt is illustrated. Once the token is extracted from the header, it is decoded to obtain the necessary data. This data is then used to validate the token. If the token is deemed valid, it is used to access Prisma and retrieve the required data. Finally, the retrieved data is passed to the relevant route.

In an application, there may be different routes that perform various tasks. For instance, for order-related routes, the application needs to retrieve orders from the client and update their status upon request. To streamline this process, a protocol called webhook is used to trigger events whenever a new order is created.

To implement this webhook protocol, the application uses Socket.IO version 2.11, which establishes a persistent connection between Nest and Ordering APIs. Whenever a new order is created, the data is sent from Ordering APIs to Nest through this established connection. Nest then forwards the order information to the appropriate client, allowing the client to view the new order and respond accordingly.

The use of webhooks and Socket.IO can provide real-time updates to the clients, making the application more efficient and user-friendly. By keeping the connection open between Nest and Ordering APIs, the application can receive updates quickly and reduce the need for constant requests to retrieve new data.

CONCLUSIONS

The goal of this thesis was to develop a user-friendly system for restaurants to efficiently track orders. The system was designed to be an improvement over the old application in terms of usability. The final result exceeded expectations, meeting all requirements and working seamlessly on tablets. The user interface was significantly improved, providing a more pleasing experience for users.

In addition to the areas for improvement mentioned above, future development of the delivery merchant application made with ReactJS could also include integration with popular payment gateways and third-party logistics providers. This would allow merchants to process payments and track shipments within the application, further streamlining their delivery operations. Additionally, incorporating machine learning algorithms could improve the accuracy of delivery estimates and optimize delivery routes, ultimately reducing delivery times and increasing customer satisfaction. These features would make the application even more attractive to businesses looking to improve their delivery operations and stay competitive in the market. Overall, the delivery merchant application made with ReactJS has great potential for future development and improvement, and I am excited to see how it will continue to evolve and meet the needs of businesses in the future.

During the project, I encountered several challenges that I had to overcome while learning a great deal about developing user-friendly systems for restaurants to efficiently track orders. One of the biggest challenges was designing an intuitive user interface that could be easily navigated by restaurant staff. It required a lot of trial and error to come up with a design that was both functional and aesthetically pleasing. I also faced technical difficulties when it came to integrating the application with existing restaurant management systems, as there were compatibility issues that needed to be resolved. Additionally, ensuring the security of customer data was another challenge that required me to implement various security measures to prevent unauthorized access and protect sensitive information. Despite these obstacles, I gained valuable experience in designing and implementing an application using ReactJS, learning the importance of user interface design and implementing advanced caching and data prefetching techniques. This project provided me with valuable experience that will be useful in my future career as a software developer, teaching me about software development, project management, and problem-solving.

To further enhance the potential of the delivery merchant application made with ReactJS, additional features such as automated inventory management, and customer feedback options could be integrated. These features would provide businesses with valuable insights into their operations and enable them to make data-driven decisions to improve their services. Furthermore, incorporating multi-language support and accessibility features would make the application more inclusive and accessible to a wider range of users.

It is important to note that the successful completion of this project was made possible through the collaborative efforts of a team of professionals. Working alongside my team members taught me the value of effective communication, teamwork, and delegation. These skills are essential for any software development project, and I am grateful to have had the opportunity to develop them during this project.

In conclusion, the delivery merchant app made with ReactJS is a significant improvement over the old application, providing a more user-friendly experience and streamlining delivery operations for businesses. While there are areas for improvement and future development, the potential of this application is promising, and its impact on the industry could be significant. This project has taught me valuable skills and knowledge, and I am excited to apply them in my future career as a software developer.

REFERENCES

1. Nihar Raval, “Typescript and JavaScript: The Difference you Should Know” Oct.13 ,2022. [Online] https://radixweb.com/blog/typescript-vs-javascript [Accessed Oct 22,2022]
2. “MDN Web Doc” [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript [Accessed Oct 22,2022]
3. David Herbert, “What is React.js? (Uses, Examples, & More)” [Online] , June 27,.2022 https://blog.hubspot.com/website/react-js [Accessed Oct 22,2022]
4. “Manipulation of Resources through Representations” [Online] https://restapilinks.com/manipulation-of-resources-through-representations/ [Accessed Oct 30,2022]
5. Lindsay Kolowich Cox, “Web Design 101: How HTML, CSS, and JavaScript Work” [Online] June 21, 2021 https://blog.hubspot.com/marketing/web-design-html-css-javascript [Accessed Oct 24,2022]
6. Rachel Meltzer, “What Is JavaScript Used For?” Nov 15,2021 [Online] https://www.lighthouselabs.ca/en/blog/what-is-javascript-used-for? [Accessed Oct 24,2022]
7. Menard Maranan, “The React lifecycle: methods and hooks explained” Mar 5,2022 [Online] https://retool.com/blog/the-react-lifecycle-methods-and-hooks-explained [Accessed Oct 24,2022]
8. CapacitorJs [Online] https://capacitorjs.com/ [Accessed Oct 24,2022]
9. Material UI [Online] https://mui.com/material-ui/ [Accessed Oct 25,2022]
10. James Denman, “Definition NodeJs” [Online] Oct 23,2022 https://www.techtarget.com/whatis/definition/Nodejs [Accessed Oct 27,2022]
11. “What is Nodejs and Why you should use it?” [Online] https://kinsta.com/knowledgebase/what-is-node-js/ [Accessed Oct 24,2022]
12. MongoDB [Online] https://www.mongodb.com/advantages-of-mongodb [Accessed Oct 28,2022]
13. AWS Amazon [Online] https://aws.amazon.com/what-is/restful-api/ [Accessed Oct 27,2022]
14. ExpressJs [Online] https://expressjs.com/ [Accessed Oct 27,2022]
15. Nodejs [Online] https://nodejs.org/en/about/ [Accessed Oct 28,2022]
16. “Lesson 4: Comparing MongoDB vs SQL Concepts” [Online] https://studio3t.com/academy/topic/mongodb-vs-sql-concepts/ [Accessed Oct 28,2022]

17. "Install Node.js" [Online] https://www.tutorialsteacher.com/nodejs/setup-nodejs-development-environment [Accessed Dec 15,2022]
18. Create React App [Online] https://create-react-app.dev/docs/getting-started [Accessed Dec 15,2022]
19. Nestjs [Online] https://docs.nestjs.com/ [Accessed Jan 24,2023]
20. What is Prisma? [Online] https://www.prisma.io/docs/concepts/overview/what-is-prisma [Accessed Jan 24,2023]
21. What Is SQLite? [Online] https://www.sqlite.org/index.html [Accessed Jan 24,2023]
22. What Socket.IO is [Onlne] Introduction | Socket.IO [Accessed Jan 24,2023]
23. Deployed Server [Onlne] Swagger UI (munchi-apis.fly.dev)

```
import { configureStore } from "@reduxjs/toolkit";
import { munchiApi } from "./slices/api";
import sessionReducer from "./slices/session";


const store = configureStore({
  reducer: {
    session: sessionReducer,
    [munchiApi.reducerPath]: munchiApi.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({ serializableCheck: true }).concat(
      munchiApi.middleware
    ),
});


// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<typeof store.getState>;
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}
export type AppDispatch = typeof store.dispatch;


export default store;
```

```
const baseQueryWithReauth: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError
> = async (args, api, extraOptions) => {
  //
  await mutex.waitForUnlock();
  let result = await baseQuery(args, api, extraOptions);
  if (result.error && result.error.status === 401) {
    // try to get a new token
    if (!mutex.isLocked()) {
      const release = await mutex.acquire();
      const refreshToken = (api.getState() as RootState).session.refreshToken;
      try {
        const refreshResult = await axios({
          url: `${process.env.REACT_APP_API__URL_DEV}auth/refreshToken`,
          method: "GET",
          headers: {
            Authorization: `Bearer ${refreshToken}`,
          },
        });
        if (refreshResult.data) {
          api.dispatch(
            setSessionState({
              refreshToken: refreshResult.data.refreshToken,
              verifyToken: refreshResult.data.verifyToken,
            })
          );
          await addSessionState({
            refreshToken: refreshResult.data.refreshToken,
            verifyToken: refreshResult.data.verifyToken,
          });
          // retry the initial query
          result = await baseQuery(args, api, extraOptions);
        } else {
          api.dispatch(clearSessionState()); //logout and clear all data
          await clearSession();
        }
      } finally {
        // release must be called once the mutex should be released again.
        release();
      }
    } else {
      // wait until the mutex is available without locking it
```

```
const sound = new Howl({
    src: [music],
    loop: true,
    preload: true,
    onplayerror: function () {
      sound.once('unlock', function () {
        sound.play()
      })
    },
    onloaderror: () => {
      sound.once('unlock', function () {
        sound.play()
      })
    },
  })

  useEffect(() => {
    socket.emit('join', businessId)
    socket.on('new-order-notification', (socket) => {
      setIsPlaying(true)
      setRefetch(true)
      setOpen(true)
      setNewOrder(socket)
    })
    socket.on('order-status-change', (socket) => {
      setOpen(true)
    })
  }, [socket, newOrder, open, isPlaying])
  if (isPlaying) {
    sound.play()
  } else {
    sound.stop()
  }
```