



Expertise
and insight
for the future

Tien Pham

Building an online shop application with MERN stack

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

1 November 2020

Author Title	Tien Pham Building an online shop application with MERN stack
Number of Pages Date	54 pages 1 November 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen (Head of Department)
<p>Over the last few years, web development and web technologies have evolved significantly. The advancements in the space of tools and technologies have enhanced developers' experience, user experience and other aspects of web applications. The rise in popularity of React and Single Page Applications concept has made the MERN stack (MongoDB, Express, React and Node) a popular tech stack to build web applications.</p> <p>The thesis's goals were to describe and examine the concepts and functionalities of the MERN stack and implement an online shop application based on the MERN stack. Each technology comprising the MERN stack was explained in detail, along with assistive packages and libraries to build the application, such as Redux and JWT.</p> <p>In the end, a functional and production-ready online shop application was built and deployed. The assessments and possible improvements of the application were also mentioned. The thesis can be used as a comprehensive tutorial about the MERN stack, which targets beginners and people wanting to learn more about the tech stack.</p>	
Keywords	MERN stack, React, Redux, Node, Express, MongoDB, JWT, online shop, web development

Contents

List of Abbreviations		
1	Introduction	1
2	Theoretical background	2
2.1	MERN	2
2.1.1	MongoDB	3
2.1.2	Express	4
2.1.3	React	5
2.1.3.1	React Components	5
2.1.3.2	React Hooks	6
2.1.4	Node	8
2.1.5	Relevance of MERN stack	8
2.2	Mongoose	9
2.3	Redux	10
2.4	Authentication and authorization with JWT	11
3	Application requirements and project setup	12
3.1	Application requirements	13
3.2	Development environment setup	13
3.2.1	Git	13
3.2.2	Visual Studio Code	14
3.2.3	Node	14
3.3	Application setup	14
3.3.1	Frontend initialization	14
3.3.2	Backend initialization	18
3.3.3	Frontend and backend communication	21
3.3.4	Mongoose connection	23
4	Application implementation	24
4.1	Backend development	24
4.1.1	Mongoose models	24
4.3.1.1	User model	24

4.3.1.2	Product model	26
4.3.1.3	Order model	27
4.1.2	Backend user authentication and authorization	28
4.1.3	Routing and APIs	32
4.2	Frontend development	33
4.2.1	Redux integration	33
4.2.2	Home page	35
4.2.3	User authentication	39
4.2.3.1	Login page and login functionality	40
4.2.3.2	Logout functionality	43
4.2.3.3	Private routes and Admin routes	43
4.2.4	Sign up page	45
4.2.5	Product page and Cart page	46
4.2.6	Checkout page and Paypal integration	46
4.2.7	Admin pages	48
4.3	Deployment on Heroku	48
5	Discussion	49
6	Conclusion	51
	References	52

List of Abbreviations

API	Application program interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HOC	Higher-order components
JSON	JavaScript Object Notation
MEAN	MongoDB, Express, Angular, Node
MERN	MongoDB, Express, React, Node
NPM	Node Package Manager
ODM	Object Document Mapping
ORM	Object-relational mapping
SEO	Search Engine Optimization
SPA	Single Page Application
SQL	Structured Query Language
VCS	Version Control Systems
VDOM	Virtual DOM

1 Introduction

In recent years, web development has advanced rapidly. Many tools and technologies have been introduced to improve developers' experience, user experience, and web application performance. Every web application can be constructed with numerous different technologies. The term "stack", which indicates the combination of different technologies used to create an application, received attention since the creation of LAMP stack (Linux, Apache, MySQL and PHP). Nowadays, many options exist for developers to choose when forming a stack to develop new applications.

With the recent developments in web technologies, the concept of Single Page Applications (SPAs) has become popular. The idea which SPAs rely on is to avoid reloading the application and re-fetching an entire web page's content from the server to update the user interface of the application. Compared to the traditional way of reloading the web page to get new data, SPAs improve the user experience by avoiding constant reloads and enhancing application performance by only fetching related data.

The rise in popularity of SPAs has increased the usage of frontend frameworks and libraries, since a lot of the work is carried out on the frontend. One of the earliest stacks that embodied the adoption of SPAs was the MEAN stack; which consists of MongoDB as a database, AngularJS as the frontend framework, Express as a web server and Node as a runtime environment; was one of the most familiar stacks to build web applications a couple of years back. React, an alternative to AngularJS, was introduced and quickly gained traction in the frontend community, thus replacing the "A" in MEAN to form the MERN stack.

The thesis's objectives were to explain and demonstrate the characteristics and core concepts of the MERN stack and build an application utilizing the MERN stack. At the beginning of the thesis, each technology in the MERN stack was discussed in depth, along with the relevance and characteristics of the MERN stack. Furthermore, an application was developed to explore further the concepts and modern practices regarding this tech stack.

2 Theoretical background

In general, the term “stack” refers to a combination of different technologies to build web applications, and MERN (MongoDB, Express, React, and Node) stack is one of the most popular options. In this section, different technologies combining to form the MERN stack were explained in detail of what they demonstrate, how they function and connect. Furthermore, other tools that help to handle application data management, error handling and user authentication and implement the end application were discussed and examined.

2.1 MERN

MERN is one of the formations based on MEAN stack, which was first introduced by an engineering team working at MongoDB in 2013. MEAN stack abbreviates the combination of these languages and frameworks: M for MongoDB, E for Express, A for AngularJS, and N for Node. By replacing the popular framework AngularJS with the library React to cover the frontend and combined as MERN stack, React can accompany the other technologies to create Javascript and JSON oriented applications. [2.]

Within the MERN stack, MongoDB acts as a document database, Express is a web framework and web server, React works as a client-side library, and Node is a runtime environment. Figure 1 below explains the architecture of the MERN stack.

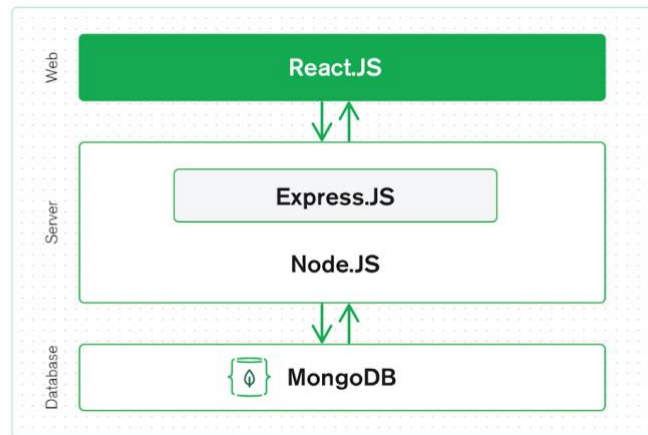


Figure 1. The 3-tier architecture (frontend, backend, database) of MERN stack [1].

Based on Figure 1, it is evident that the MERN stack is a full-stack application development solution, which employs the idea of the long-established 3-tier architecture. The MERN stack consists of the client display tier with React, application tier with Express and Node, and database tier with MongoDB. [1.]

2.1.1 MongoDB

Initially introduced publicly in 2007, MongoDB gradually stood out to be a handy database technology for developers [3]. Unlike SQL, which is known as Structured Query Language, MongoDB is a representative of the NoSQL family generally and document-based language trees specifically. [4.] The terminology for NoSQL is sometimes mentioned as non SQL or not only SQL. Nonetheless, the majority concluded that NoSQL databases store data flexibly in a format of JSON-like documents instead of those collected by relational databases [4].

While a SQL database will gather data in a combination of rows and columns, a NoSQL database will organize information in terms of a document containing arrays and objects. Small-scale projects might not reflect well the difference in utilization of these two databases; nevertheless, a medium-sized program could feasibly demonstrate the team members who directly manage and develop the program, the benefits and difficulties of each database type. [5.]

2.1.2 Express

As stated in the documentation on Express's official website, Express is a minimal Node framework that can deliver a robust set of functionalities for both web and mobile applications [6]. Express is truly minimal by itself as it does little but relies on middleware and implements an elementary layer on the application features [4].

Middleware frankly demonstrates the software running in the middle way of the request-response lifecycle. One or multiple pieces of middleware are executed to perform precise tasks, such as authenticating requests or parsing the request body. The task pipeline could be commonly started with the first middleware called to process the request. That first middleware can end the request and send the response to users or invoke the next middleware to continue the request. The same process will go on as each followed middleware takes the result of the previous one as arguments until the last middleware of the pipeline. [7.]

The figure below describes the process from running the request to the final function sending out the response.

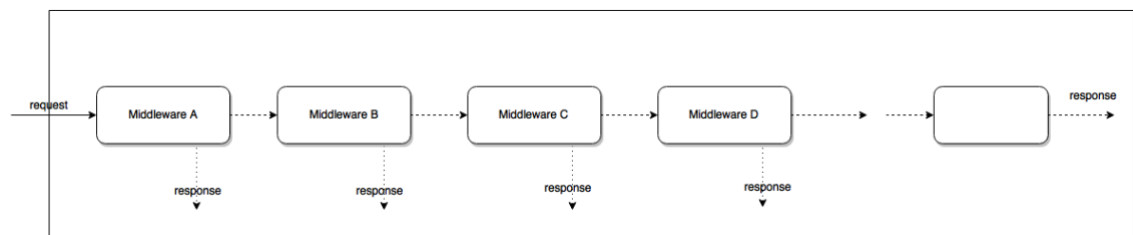


Figure 2. Request-response journey via middleware [7].

Referring to middleware, more specifically, it relates not only to functions that are able to access not only to HTTP requests and return objects but also to sequent actions in the request-response cycle of the application [4]. Precisely determined above, a routing and middleware framework such as Express can execute any lines of code, alter the request and result objects, unsubscribe to the request-response cycle, and call the next middleware method queueing in the stack. If there is such a case that the current middleware function is not capable of ending the cycle, users can follow the instruction guided on Express's website, which is to call a function written as `next()` to pass control to the next middleware approach to avoid an unfinished cycle [8].

There are different types of middleware. One of the worth noting types of middleware is router-level middleware can be used to handle routing in Express. It is similar to any other middleware, but it is constrained to only be an instance of express router. [28.]

2.1.3 React

React is a Javascript library released in 2013 by Facebook. [9.] Originally composed to resolve complex, large-scale user interfaces with real-time changing data and data binding, React keeps growing even stronger in developing a single-page application and advancing frontend utilities for all levels of programmers. One feature of React's rich feature set that satisfies large production applications, which are required to be fast and performant, is virtual DOM or VDOM. This concept presents a virtual representation of a user interface that React can manipulate quickly without touching the real interface, using that virtual object to determine what needs to be done with the real DOM tree and sync these virtual and real trees to match [10].

Coming with React core ideas, other sturdy features introduced by Facebook engineering team got welcomed as well by developers globally to tackle the other problems of any web or mobile applications requiring responsive layout and scalability of growing user data. Within this thesis's scope, only some of the major React attributes are mentioned below in the list of React Components and React Hooks.

2.1.3.1 React Components

Components are the core concept of React, where developers are encouraged to break the user interface into independent and reusable sections. A React component could be written in two ways: functional or class component and the most effortless approach to compose a component is to write a functional one, either as a Javascript function or using ES6 class to illustrate the component. In React's viewpoint, these two techniques, which are demonstrated below, are identical. [11.]

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

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

Figure 3. Functional component versus class component in React [11].

As depicted in Figure 3, functional components and class components can have the same output. The h1 component with the word “Hello ...” is displayed, along with the “name” props passed into the component.

Components can reference each other, as one component could be a parent component that contains many other child components without any limit to any level of whatever details. Whether it is a class or functional component, they both follow one strict rule set by React: all React components are pure functions that do not modify their props. Props are a set of inputs passed as parameters to a component, while pure function illustrates the case the function executes the logic without altering the arguments. Therefore, a React component operates as a pure function, which respects its inputs and always returns the same outcome for the same props. [11.]

2.1.3.2 React Hooks

Handling the global state with Redux is seemingly efficient and effective, notwithstanding, managing local state within a component with Redux is considered excessive and unnecessary. Classes from React have been doing their job well in maintaining local states with a clear syntax structure since the beginning, and it still does nowadays when applied regardless of project scale. Moreover, an additional option to do the same thing as React classes, React Hooks, was introduced by Sophie Alpert and Dan Abramov at React Conf 2018. [12.]

This change is a smooth transition from classic React classes, as Hooks do not replace and involve any new React concepts, for example relating to props, state, component

life cycles. Hooks' introduction does not mean goodbye to React classes either; developers and project managers are free to decide whether they want to try something new and move forward or stay with the same syntax they have been working. Hooks at first might initiate a confusing impression, but in the end, the logic and objectives are not anywhere far away from the core ideas of classes. In practice, one can say that React Hooks have diminished the number of coding lines as well as temporarily removed the use of the keyword "this". There are indeed more dissimilarities than just the appearance and the total number of lines.

React Hooks introduces State Hook, which is also known as the `useState` hook, which handles component level state management. To be more precise, `useState` is a hook that hooks into React's state by initializing a state variable, which is preserved by React. This hook receives and sends two values as results: current state and a function to change it. With `useState` hook, the component state can be initialized, used, and updated easily. [13.]

Another key hook that needs attention is Effect Hook, which is better known as `useEffect`. While `useState` deals with state, `useEffect` helps programmers handle component life cycles. The issue of breaking related logic and data into several class life cycles, for instance, `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`, has been well covered in Effect Hook. A React component can embrace multiple effects to separate data manipulation concerns. [13.]

One thing that React classes miss while Hooks have the answer is sharing functionality logic. To share stateful, visual, or non-visual logic previously before Hooks, programmers either choose higher-order components (HOC) or render props patterns, which expects adjusting the component hierarchy accordingly and makes the application more troublesome to follow. Hooks, on the other hand, allow developers to reuse logic without reshaping the component structure. [12.]

2.1.4 Node

Despite being a newly-born technology from 10 years ago, Node. (or Node.js) has proven itself to be a vital JavaScript runtime environment that leverages the popularity of server-side JavaScript [14]. A runtime environment is neither a language nor a framework; nevertheless, it is a powerful tool built on Chrome's V8 engine, which also runs with Javascript [4]. Using an event-driven, asynchronous, and non-blocking I/O (input/output) model, Node supports developers with another option to build lightweight and real-time applications besides the standard path of waiting and serving requests [4].

For developers who focus heavily on Javascript, NodeJS brings them the perks of building an application written in their favorite programming language in both backend and frontend [15]. To tell further about what other benefits Node also delivers, programmers feasibly highlight the Node package manager or npm, which gives access to not thousands but hundreds of thousands of packages registered in Node's system [4]. The npm registry was considered one of the world's largest package registry with more than 350,000 packages noted in 2017, which many of them are open source and developed by developers around the globe [16].

2.1.5 Relevance of MERN stack

MERN stack is not the only name in the list of combined technologies to develop web applications. To name a few, MEAN (using MongoDB, Express, Angular, NodeJS), LAMP (using Linux, Apache, MySQL or MongoDB, PHP), Django (using Python, Django, Apache, MySQL) [17].

There are different reasons why MERN stack is a popular and well-adopted tech stack to build web applications. MERN stack focuses on a single code base using Javascript and JSON, enabling developers to dwell more profoundly in a specific programming language with and enhancing team communication and workflow throughout various pieces of the whole web application. Stack consistency also indicates less time to build a working product, less struggle to further progress, extend, and maintain the application. Last but not least, each piece of the MERN stack is associated with a large community supporting behind, adding contribution to help grow the technologies and support for

programmers at any level. Plenty of open source materials are available such as documentations, customized packages, add-on libraries. [4.]

2.2 Mongoose

Mongoose is not a part of MongoDB, but rather an object document mapping (ODM) library to enhance working with Node and MongoDB [18]. Not only does it deliver schema validation and data relationship management, but it also connects the objects in code and objects in MongoDB [19]. Figure 4 below describes the connection between Mongoose and others within an application:

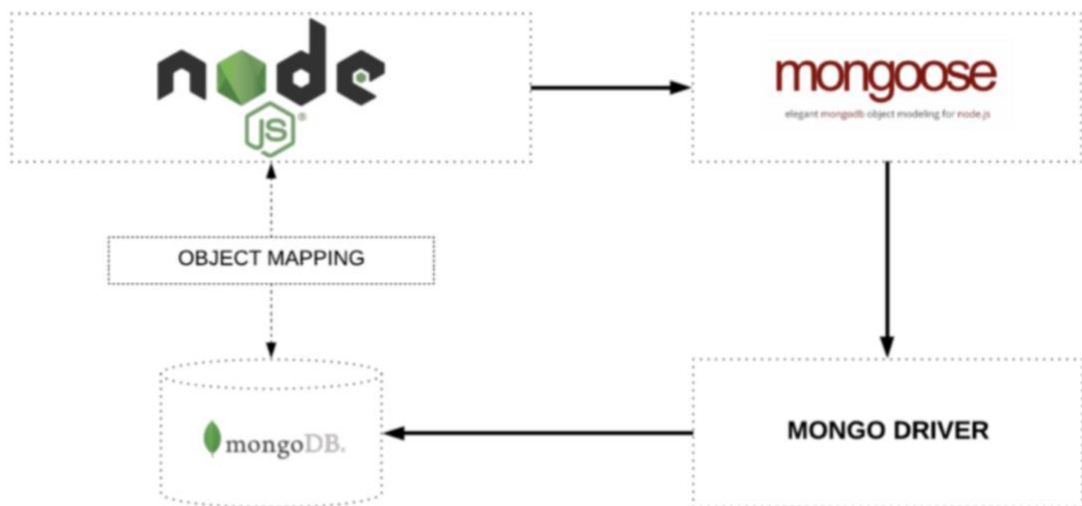


Figure 4. Mongoose maps objects between Node and MongoDB [19].

As depicted in Figure 4, Mongoose is used for Node to connect to MongoDB via object mapping. Mongoose is then connected with MongoDB via a Mongo Driver. Because of the connection between Mongoose, Node and MongoDB, data flow capability is ensured. [19.]

Similar to other ODM libraries, the first step to kick off with Mongoose is a schema [20]. A schema clarifies the data structure and property casting and other practices listed by Mongoose's documentation page: instance methods, compound index, static Model

methods, and middlewares [20]. When the first step is completed, those created schemas will map to MongoDB collections and shape each collection's data documents [20]. The second step that programmers need to follow is to produce a Mongoose model. Models are schema's compiled constructors, whose primary duties are generating and scanning Mongo database's documents. Other abilities of models worth mentioning are querying, removing, and updating documents in the database [21].

2.3 Redux

State management perhaps is not a big or the most prioritized issue to small-scale projects where data from users, server responses, and cached data do not slow down or break the limit that the web browser can handle. Notwithstanding, when the application grows, managing state becomes a real pain for further developing and debugging, especially when two concepts of asynchronicity and mutation are mixed. Asynchronicity defines several changes in an asynchronous sequence, even though the mutation clarifies application state changes. These two concepts are usually put together, in which unpredictable events or response waiting time come into play and might likely show unexpected behaviors. [22.]

Constructed and developed by Dan Abramov in 2015, Redux was proposed to solve the above problem by turning mutation predictable without affecting the asynchronicity's advantages [22]. To be more detailed, Redux stores state in a single source of truth and requires a strict structure on how state modification can occur, where reusable and pure functions are in play. Instead of altering the given object's value, pure functions return the same new values based on the provided arguments wherever they are called. Therefore, Redux allows a more straightforward debugging process, testing, code maintenance, and a smoother developing experience personally or in teamwork [23].

Another outstanding factor that Redux was chosen is its simplicity in utilizing plainly Javascript objects and functions. It could combine well with React and other frontend libraries or frameworks, for instance, AngularJS, Angular, VueJS, Polymer, Ember. To add more points to its flexibility, Redux runs on various environments like browser, server, and native. [23]

Redux's core concept spins around two words: reducer and action. To simplify the terms, every mutation of the state is called an action, and reducer is the pure function where it ties the state and related actions. To start the Redux process, developers need to dispatch an action to mutate the state, which is written as a Javascript object, involving the action name and other information, if necessary, to describe the action in more detail. The second thing to handle is to write a reducer that accepts state and action as parameters and returns the new state. As a result of a pure function, reducers' returned outcome is always the same expected value, making state immutable and following the strict guidelines proposed by Redux. [24.]

2.4 Authentication and authorization with JWT

Authentication is the procedure of finding out who the user is, whereas authorization is the procedure of permitting what the user can access. Authentication is typically done before authorization. After identifying who the user is, access to some resource is either allowed or rejected. [25.]

There are different ways to implement authentication and authorization for web applications, where the most battle-tested and standard technique is to use sessions to handle user status both on client and server [4]. However, a later published technology called JSON Web Token (JWT) has implemented a new concept of a stateless mechanism in which the user status is stored [4]. Figure 5 below explains the general flow of the mechanism:

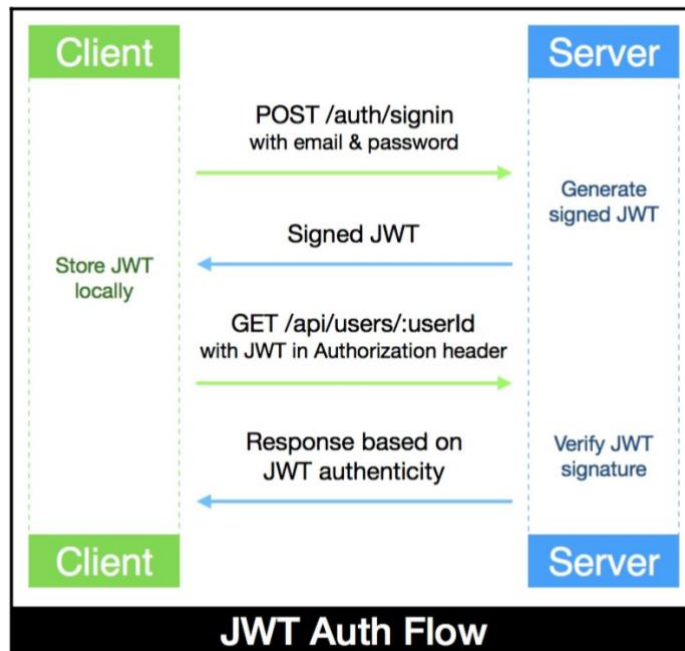


Figure 5. JWT Auth Flow [4].

As depicted in Figure 5, firstly, when a user logs in, the server creates a JWT token signed using the user detail entered and a secret key. Afterward, the token will be sent to the client to be kept in a cookie or local storage, thus handing user state maintenance to the client. After signing in successfully, in any request to protected endpoints, the token must be attached to the Authorization request header following the format of "Authorization: Bearer <JSON Web Token>". When the server handles the requests to protected endpoints, it will verify the token to check if it is valid. If that is the case, access is given to the user. Otherwise, an error occurs. [4.]

3 Application requirements and project setup

The thesis aimed to create an online shop application to examine and explore the MERN stack's modern practices and core concepts. This section was dedicated to identifying and clarifying the requirements for the online shop application, an explaining the development environment setup and the initial application setup.

3.1 Application requirements

After carefully researching a list of online shopping websites and interviewing his network on ecommerce, the thesis author has come up with a list of requirements for his application. Based on the principal concerns regarding an ecommerce website's functionalities, the application performance is divided into three categories of users based on their needs and level of access: visitor, logged user, and admin.

First of all, any visitors who pay a visit to the application should be able to view the list of products together with each item's details, including images, description and price, while also being able to search for products by product name. Meanwhile, not only can logged in users do what visitors are capable of, but also have access to more views such as logging in and out screens and the page to change contact details. They could add products to the shopping cart, pay the shopping receipt via Paypal, view their orders, and check the order delivery process. Lastly, admin is the user role holding the full access to the application. On top of all views and functionalities that visitors and logged in users can approach, the admin manages the application by adding, removing, editing the product database, or the user database.

3.2 Development environment setup

Before using the MERN stack to build the application, it was necessary to set up the development environment so that technologies and tools work well together, in association with helping to improve the development flow.

3.2.1 Git

A development workflow needs to contain a version control system that facilitates collaboration, code sharing, and reviewing code changes [26]. A version control system (VCS) monitors the changes' history and allows developers to contribute code and fix bugs to assure that it is always possible to go back to any previous version [26]. Based on the latest Stackoverflow developer survey, Git is the most popular VCS in the world

[26]. As a result, Git was installed and used for version control while developing this application.

3.2.2 Visual Studio Code

In choosing the most suitable text editor, Visual Studio Code (VS Code) stands out to be a free and open-source option that operates on macOS, Linux, and Windows, which Microsoft released in 2015. [27.] In a few years, VS Code has quickly acquired fame in the developer community with many outstanding features. Those can be listed as built-in support for Javascript and Node development, support for hundreds of languages, Intellisense, syntax highlighting, and a built-in terminal to ease the development process [27]. VS Code was then chosen as the text editor for the development of this application.

3.2.3 Node

The backbone of the backend server of the application is Node. Node needs to be installed locally in the system to develop the application. At the time of writing, Node has the latest stable version of 12.19.0, and the current version with the latest features is 15.0.1. The latest version of 15.0.1 was installed locally to take advantage of the latest features of Node, such as the usage of ES Module.

3.3 Application setup

After setting up the development environment, the basic setup of the application was built. The frontend of the application was initialized, followed by the setup of the backend. The connection between the frontend and the backend of the application was then formed correspondingly.

3.3.1 Frontend initialization

Setting up a typical working React application from scratch might be a daunting task, which often consists of configuring and setting up the Webpack build pipeline besides Babel to transpile the code. Create React App is a tool created to quickly establish a

React application boilerplate to ease the frontend development with React. For the application, a Create React App boilerplate was formed by using the command in Listing 1.

```
npx create-react-app frontend
```

Listing 1. Command to create the Create React App boilerplate

After running the above command, the frontend boilerplate was set up without configuration and generated in the folder named "frontend" of the application. The folder's structure is precisely demonstrated in the following Figure 6:

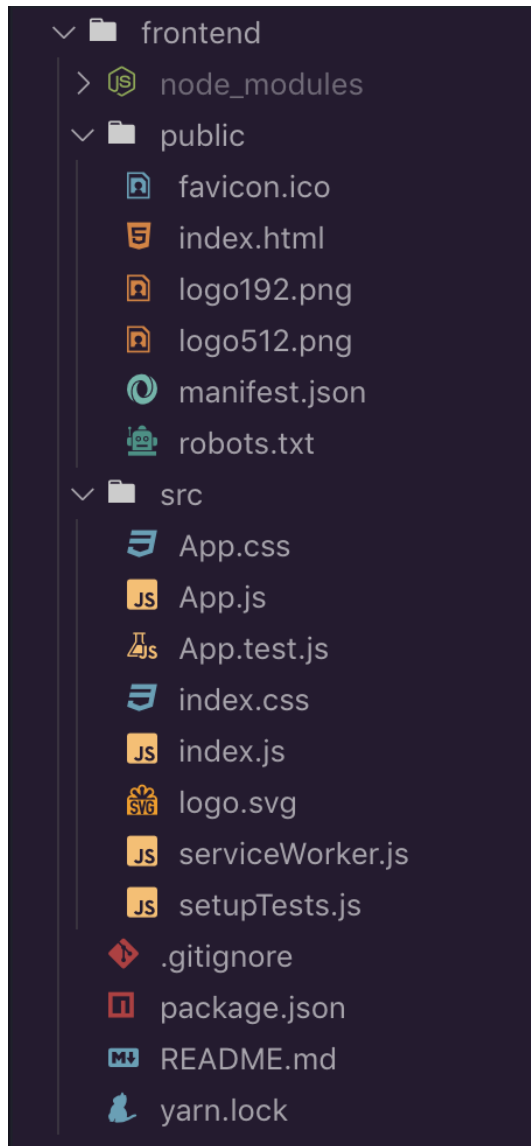


Figure 6. Structure of the React frontend folder.

In the React frontend folder, as shown in Figure 6, the structure includes the "node_modules" folder, which has the installed dependencies, and the "public" folder that contains all the static files. The structure continues with the "src" folder that covers the application components. Besides, the package.json file that lists dependencies and their respective versions, stores metadata of the frontend, and runs defined scripts, is instantiated.

The package.json file contains scripts that can be executed to run, build or develop the application. Listing 2 illustrates the possible commands that come built-in with Create React App:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

Listing 2. Built-in commands of Create React App in package.json folder.

Listing 2 illustrated commands to start, build, test, or eject the frontend. To start the application on `http://localhost:3000` as the default port of the boilerplate, developers can simply change the directory to go inside the "frontend" folder and run "npm start" on the terminal, which will run the "start" command. Inside the src folder, the index.js file is the main Javascript entry point of the application's frontend. On the other hand, the index.html file inside the "public" folder is the page template for the React frontend of the application.

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

ReactDOM.render(<App />, document.getElementById('root'))
```

Listing 3. Content of index.js file.

According to Listing 3's file content, the App component is rendered by the ReactDOM.render method in place of the element that has the id of "root", which is an empty HTML div element inside the index.html file. Initially, the browser reads the index.html file with the id of "root", then it will render the App component, which is the main component of the frontend of the application.

Since React is a library following the Single Page Application (SPA) paradigm, it is crucial to have the ability to navigate between pages without reloading the page to improve user experience. Taking care of the same concern, React Router shows up to be a useful and popular package for React to handle routing and navigation throughout the frontend. The "react-router-dom" package needs to be installed using the "npm install react-router-dom" command to use React Router. More details on using React Router in the frontend of the application are displayed in the listing below.

```
import { Route } from "react-router-dom";
const HomeScreen = () => <div>Home</div>
const ProductScreen = () => <div>Product A</div>

const App = () => (
  <BrowserRouter>
    <Route path="/" component={HomeScreen} exact />
    <Route path="/product/:id" component={ProductScreen} />
  </BrowserRouter>
);
```

Listing 4. Usage of React Router package.

As can be seen from Listing 4, React Router operates with the `BrowserRouter` component, which covers multiple `Route` components carrying a separate React component passed in. The `BrowserRouter` component provides all the context of routing to the React Router components declared inside it, where the `Route` component renders a component if the URL path matches the route's path. To test that React Router is working well, it is possible to navigate to "http://localhost:3000/", which is the "/" path that matches the first route, then the "Home" text inside the `HomeScreen` component will be shown.

3.3.2 Backend initialization

To start the work on the backend, a `package.json` file was created to keep track of dependencies, document the project, run commands, and initialize the backend. The command in Listing 5 was run from the root folder to prompt a series of questions to gather the project's information, such as the author, license, and project name. Then a `package.json` was generated automatically.

```
npm init
```

Listing 5. Command to initialize the backend.

Subsequently, a file called `server.js` was added to be the entry of the application's backend, whereas "express" and "dotenv" packages were installed as dependencies. The content of the initial `server.js` file assembled at the beginning of setting up the backend is demonstrated in Figure 7.

```
1 const express = require('express');
2 const dotenv = require('dotenv');
3 // import express from 'express'
4 const app = express();
5 const PORT = process.env.PORT || 4000;
6
7 app.get('/', (req, res) => {
8   res.send('API is running...');
9 })
10
11
12 app.listen(PORT, console.log(`Server running on port ${PORT}`));
```

Figure 7. Content of initial server.js file.

From Figure 7, the "express" module was imported using CommonJS import syntax on the line 1, followed by the `express()` function initiating the server application on line 4. On line 2, the "dotenv" package was also imported and initialized so that application secrets such as tokens or API keys in the ".env" file can be stored and kept out of the git history for security reasons. The application starts the server that connects to port 4000, then logs "Server is running on port 4000" from the console. The application responds with "API is running" for the requests coming to the root URL afterward.

For Node and Express development, in the expectation of putting ESM module syntax into service, as shown in the commented out line 3 of Figure 7, which is a more modern way of importing and exporting files instead of CommonJS syntax, the update in Listing 6 was added to the root package.json file.


```
{  
  ...  
  "type": "module",  
  ...  
}
```

Listing 6. Update to enable ESM module syntax for Node and Express development.

As shown in Listing 6, the "type" of "module" was set in the package.json file, which enabled ESM module for Node and Express development. After creating a working server, some folders were created in the root of the project, illustrated in Figure 8.

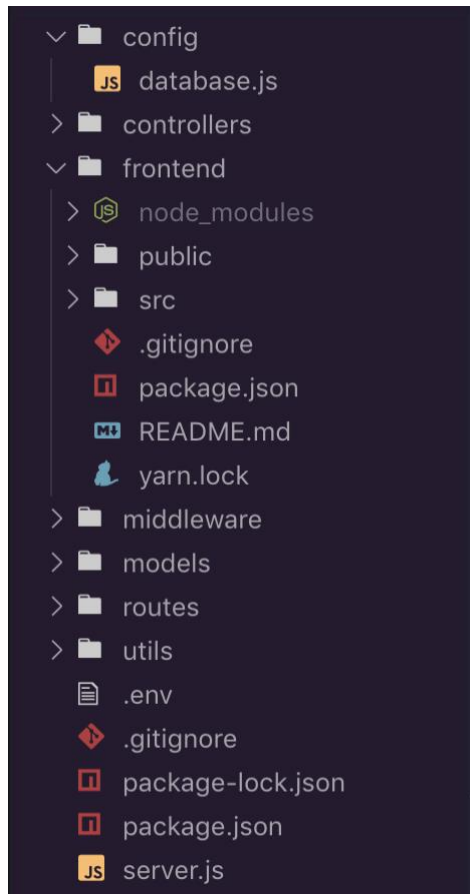


Figure 8. The updated folder structure of the application.

Based on Figure 8, the project's root remained the backend of the application and backend related dependencies would be installed in the root package.json file. The "config" folder hosts the file containing the MongoDB connection, the "controllers" folder was added to contain controllers for the backend, the "models" folder contains models, and so on.

3.3.3 Frontend and backend communication

The proxy from Create React App was used for the frontend to communicate with the backend in a different port. Listing 7 shows the configuration installed to the frontend package.json file.

```
"proxy": "http://localhost:4000"
```

Listing 7. Proxy to enable communication from frontend to backend.

From Figure 7, proxy from Create React App is connected to the backend server, which runs on port 4000. As a result, the frontend and backend communication is then formed. Furthermore, it was necessary to create scripts to run the application seamlessly. The "nodemon" module was added to continually watch changes to the server code so that it is not necessary to keep resetting the server. The upcoming Listing 8 presents the commands to run the frontend and the backend of the application:

```
"scripts": {  
  "server": "nodemon server",  
  "client": "npm start --prefix frontend",  
  "start": "concurrently 'npm run client' 'npm run server'",  
},
```

Listing 8. Commands to run the application.

In Listing 8, the command to start the backend is "npm run server", and the command to start the frontend is "npm run client". It is critical to run both the backend and the frontend of the application simultaneously in order for the application to work. Therefore, the "concurrently" package is employed to run both the backend and the frontend starting commands simultaneously.

3.3.4 Mongoose connection

For the sake of connecting the MongoDB database to the application, a tool called Mongoose is utilized. As discussed in section 2.2, Mongoose is an Object Data Modelling package for Node which lets developers construct a Model and a schema for different database resources. Figure 9 below displays how Mongoose connection is formed:

```
1 import mongoose from "mongoose";
2
3 export const connectDatabase = async () => {
4   try {
5     const connection = await mongoose.connect(process.env.MONGO_URI, {
6       useNewUrlParser: true,
7       useUnifiedTopology: true,
8       useCreateIndex: true,
9     });
10
11     console.info(`MongoDB Connected: ${connection.connection.host}`);
12   } catch (err) {
13     console.error(error.message);
14     process.exit(1);
15   }
16 };
17
18 // In server.js
19 connectDatabase()
```

Figure 9. Mongoose connection.

According to Figure 9, the connectDatabase is first created, which will form the Mongoose connection using the MONGO_URI environment variable. After that, the function is imported into the server.js file and ran to connect the application to MongoDB and Mongoose. After trying to run the server, if the connection is successful, the success message will include "MongoDB Connected: ..." and displayed on the command line. Otherwise, an error message will be shown along with the error returned from Mongoose.

4 Application implementation

The online shop application's implementation process consisted of three steps. First, backend development was conducted. Afterward, the frontend development took place, and finally, the deployment of the application took place. All those steps were explained in detail in this section.

4.1 Backend development

The backend development of the application consisted of creating Mongoose models creation, backend user authentication and authorization, and the creation of routing and APIs.

4.1.1 Mongoose models

A Mongoose schema specifies the document structure stored in MongoDB, whereas a Mongoose model is formed by wrapping a Mongoose schema to create an interface to the MongoDB database for querying, creating, deleting, or updating the records. There are three models in total in the application: user, product, and order models.

4.3.1.1 User model

Figure 10 below displays the details of the user model. The instance of Mongoose schema is instantiated and called `userSchema`, which includes the schema definition and business logic to create the user model. The user schema has four required fields: name, email, password, and `isAdmin`. Email is set to be unique because there must not be many people having the same email address. The “timestamps” second argument to mongoose schema is set to true so that Mongoose generates `createdAt` and `updatedAt` fields automatically so that it is convenient to check when the data is changed or constituted if necessary. The "isAdmin" field has a boolean value and indicates whether the user is admin or not, and it is defaulted to false.

```

1 import mongoose from "mongoose";
2 import bcrypt from "bcryptjs";
3
4 const userSchema = mongoose.Schema({
5   name: { type: String, required: true,},
6   email: { type: String, required: true, unique: true,},
7   password: { type: String, required: true,},
8   isAdmin: { type: Boolean, required: true, default: false,},
9 },
10 { timestamps: true,}
11 );
12
13 userSchema.pre('save', async function (next) {
14   if (!this.isModified('password')) {
15     next()
16   }
17
18   const salt = await bcrypt.genSalt(10)
19   this.password = await bcrypt.hash(this.password, salt)
20 })
21
22 userSchema.methods.matchPassword = async function (enteredPassword) {
23   return await bcrypt.compare(enteredPassword, this.password);
24 };
25
26
27 const User = mongoose.model("User", userSchema);
28
29 export default User;
30

```

Figure 10. User model.

The schema in Figure 10 is later modeled to formulate the user model, which is exported for the rest of the server code to reuse. On line 13, a "pre-save" hook that will run before saving a document to the database is attached to the user schema. For security reasons, passwords must not be stored in the database as plain texts and must always be encrypted. The bcryptjs library was installed to handle encryption and comparison of passwords in this application. On line 18, a salt (random characters) to add to the hashed password was drawn by using the genSalt() method of bcrypt, which took in a number of generation rounds. At a subsequent time, the salt was passed to bcryptjs's hash() function, and the encrypted password was assigned to the password that would be saved to the database. In spite of that, the password field's encryption is only handled if the password has been modified or has only been sent for the first time. Therefore, a check method was introduced from line number 14 to 16 of Figure 10 to examine if the

password is not changed. For instance, if a user updates only the name field or email field but not the password, the `next()` function will be called, and the application moves on and does not update the password. Otherwise, the application will create a new hash, and the user will not be able to login.

From line number 22 to 24 of Figure 10, an instance method called "matchPassword" is attached to the user schema. The method provides a comparison between the password that the user enters when logging in and the encrypted password in the database by using the `compare()` method of `bcryptjs`. Since password encryption is a one-way flow for security reasons, meaning that it is impossible to decrypt an encrypted password to be plain text, the `compare()` method enables comparing the passwords to assess if the user entered the correct password.

4.3.1.2 Product model

Figure 11 explains the details of the product model. The instance of the Mongoose schema called `productSchema` was initialized, which encapsulated the product model's business logic.

```
1 import mongoose from "mongoose";
2
3 const productSchema = mongoose.Schema({
4   user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: "User", },
5   name: { type: String, required: true, },
6   image: { type: String, required: true, },
7   brand: { type: String, required: true, },
8   category: { type: String, required: true, },
9   description: { type: String, required: true, },
10  price: { type: Number, required: true, default: 0, },
11  countInStock: { type: Number, required: true, default: 0, },
12 },
13 { timestamps: true, }
14 );
15
16 const Product = mongoose.model("Product", productSchema);
17
18 export default Product;
19 |
```

Figure 11. Product model.

The product schema contains numerous fields related to the product, as shown in Figure 11. It contains name, image, brand, category, product description, price, and the product count in stock. Furthermore, it contains the user who created that product, in the form of a Mongoose Schema ObjectId with a direct reference to the User model. All fields in the product schema are required.

4.3.1.3 Order model

Compared to the user and product schemas, the order schema has more fields and looks more complicated. Figure 12 indicates the structure of the order schema alongside the exporting of the order model.


```

1 import mongoose from 'mongoose'
2
3 const orderSchema = mongoose.Schema({
4   user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User'},
5   orderItems: [{
6     name: { type: String, required: true },
7     qty: { type: Number, required: true },
8     image: { type: String, required: true },
9     price: { type: Number, required: true },
10    product: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Product' },
11  ]},
12  shippingAddress: {
13    address: { type: String, required: true },
14    city: { type: String, required: true },
15    postalCode: { type: String, required: true },
16    country: { type: String, required: true },
17  },
18  paymentMethod: { type: String, required: true, },
19  paymentResult: {
20    id: { type: String },
21    status: { type: String },
22    update_time: { type: String },
23    email_address: { type: String },
24  },
25  taxPrice: { type: Number, required: true, default: 0.0, },
26  shippingPrice: { type: Number, required: true, default: 0.0, },
27  totalPrice: { type: Number, required: true, default: 0.0, },
28  isPaid: { type: Boolean, required: true, default: false, },
29  paidAt: { type: Date, },
30  isDelivered: { type: Boolean, required: true, default: false,},
31  deliveredAt: {type: Date, },
32  },
33  { timestamps: true, }
34 )
35
36 const Order = mongoose.model('Order', orderSchema)
37
38 export default Order

```

Figure 12. Order model.

As exhibited in Figure 12, the order schema has the keys of shippingAddress, paymentMethod, paymentResult, taxPrice, totalPrice, and shippingPrice. Also, the order schema has isPaid key, which indicates whether the order is paid for in conjunction with isDelivered key, which shows if the order is delivered. The order schema also has paidAt and deliveredAt, which present the time of payment and delivery. Similar to product schema, the order schema includes the user field representing the user who set up that product with a direct reference to the User model. The orderItems field is an array which includes the ordered items.

4.1.2 Backend user authentication and authorization

When a request is made to the Express server, it is possible to have a middleware that can access anything in the request and response object. Listing 9 reveals the addition of a middleware in the application.

```
app.use(express.json())
```

Listing 9. A middleware connected in the server.js file.

The `express.json()` middleware, which parses the requests coming with JSON payload, was utilized in the application. As discussed in section 2.1.2, it is also possible to use router-level middleware, but it has to be an instance of `express.Router()` function. The usage of router-level middleware is illustrated in Listing 10.

```
import express from "express";
import {authUser} from "../controllers/userController.js"
const authRouter = express.Router();
authRouter.post("/login", authUser);

app.use('/api/auth', authRouter)
```

Listing 10. Using router-level middleware in server.js.

The `authRouter` is initialized as an instance of `express.Router()`. Afterward, for requests coming to `/api/auth`, the `authRouter` will handle the routing for child routes inside it. For example in Listing 10, when the user goes to `/api/auth/login`, the only child route of `authRouter`, the `authUser` controller will be used to handle requests.

The only one route for user login in this application is the `/api/auth/login` route with POST method. From Listing 10, the `authUser()` function acts as the controller for this route. When a user logs in through this route, the user's email and password will be checked against the database. Afterward, JWT will be used to grant access to certain parts and protected routes in the API. The application instantiates and signs a JWT with a secret key when a user logs in, then sends it back to the client to be used and stored in the client. If the user needs to access any protected route, the token can be sent in the headers, and the server will decode the token with the same secret key to verify that the token is valid. Figure 14 displays the logic of the `authUser()` controller.

```

1 import jwt from 'jsonwebtoken';
2
3 const generateToken = (id) => {
4   return jwt.sign({ id }, process.env.JWT_TOKEN, { expiresIn: '30d', })
5 };
6
7 const authUser = async (req, res) => {
8   const { email, password } = req.body;
9
10  const user = await User.findOne({ email });
11
12  if (user && (await user.matchPassword(password))) {
13    res.json({
14      _id: user._id,
15      name: user.name,
16      email: user.email,
17      isAdmin: user.isAdmin,
18      token: generateToken(user._id),
19    });
20  } else {
21    res.status(401).send("Invalid email or password");
22  }
23 };|

```

Figure 13. Logic of authUser() controller.

As shown in Figure 13, on line 10, the user entry is checked in the database using email. If a user with that email exists, the password user entered is checked to see if it matches the password entry stored in the database on line 12. On line 18, if the user is successfully verified, the application will send back a JWT token that is signed using the user's unique `_id` that is automatically generated by MongoDB and the secret key. The `generateToken()` method creates a signed JWT with a secret key that will expire in 30 days. The `authUser()` controller sends the JWT token back so that if the user accesses a protected route on the server, the user can send the JWT token on the headers based on the convention "Authentication: Bearer <TOKEN>".

For user authorization, a middleware named "protected" was called into play to restrict only logged in users to access the endpoints that require logged-in users. Listing 11 points out the usage of "protected" middleware.

```
router.route('/api/user/profile').get(protected, getUserProfile)
```

Listing 11. Usage of "protected" middleware.

The route called `/api/user/profile` in Listing 11 is employed to display the user profile for a logged-in user. The route is guarded with the `"protected"` middleware to only allow logged-in users to access the route. The complete code of `"protected"` middleware is listed in Figure 14.

```
1 import jwt from "jsonwebtoken";
2 import User from "../models/user.js";
3
4 const protected = async (req, res, next) => {
5   let token;
6
7   if (
8     req.headers.authorization &&
9     req.headers.authorization.startsWith("Bearer")
10  ) {
11    try {
12      token = req.headers.authorization.split(" ")[1];
13
14      const decoded = jwt.verify(token, process.env.JWT_TOKEN);
15
16      req.user = await User.findById(decoded.id).select("-password");
17
18      next();
19    } catch (error) {
20      console.error(error);
21      res.status(401).send("Not authorized, token failed");
22    }
23  }
24
25  if (!token) {
26    res.status(401).send("Not authorized, no token");
27  }
28 }
```

Figure 14. The `"protected"` middleware.

As shown in Figure 14, the first request headers were checked to see if the authorization header is in the request. After that, on line 12, the JWT token is extracted from the authorization header. Thereupon, the JWT token is decoded using the JWT's `verify()` function to obtain the user id from the JWT token. If the JWT is invalid, the `verify()` function will error out, and the error will be sent back to the client. Afterward, the middleware checks if there is any user in the database that has the same user id. If the user with the same user id in the database is found, the middleware will allow the user to access the route. On the other hand, the request object's `"user"` field will be populated

with the found user entry in the MongoDB database to be used by the middlewares that come after this, which is called "admin". Figure 15 below displays the "admin" middleware:

```
1 const admin = (req, res, next) => {
2   if (req.user && req.user.isAdmin) {
3     next();
4   } else {
5     res.status(401);
6     throw new Error("Not authorized as an admin");
7   }
8 };
9
```

Figure 15. The "admin" middleware.

As displayed in Figure 16, on line 2, the "admin" middleware first checks if the "user" field is in the request object, and it has a truthy "isAdmin" flag. If the test passes, the user is an admin, and the user can access the route. The usage of the "admin" middleware is presented in Listing 12 below:

```
router.route('/api/user').post(registerUser).get(protected, admin, getUsers)
```

Listing 12. Usage of "admin" middleware.

From Listing 12, the "admin" middleware is placed after the "protected" middleware so that the "protected" middleware can pass along the "user" object, which represents the user who is accessing the route. Afterward, the "admin" middleware takes the user object appended to the request object by the "protected" middleware, and checks if the "isAdmin" field in the user object is true, which indicates that the user is an admin. If that is the case, the "admin" middleware will permit the user to access the protected route.

4.1.3 Routing and APIs

To better structure router-level middlewares, in this application, all the router-level middlewares are held in the "routes" folder. Figure 16 below explicates the usage of router-level middlewares in the application:

```
1 import productRoutes from './routes/productRoutes.js'  
2 import authRoutes from './routes/authRoutes.js'  
3 import userRoutes from './routes/userRoutes.js'  
4 import orderRoutes from './routes/orderRoutes.js'  
5 ...  
6 app.use('/api/products', productRoutes)  
7 app.use('/api/auth', authRoutes)  
8 app.use('/api/users', userRoutes)  
9 app.use('/api/orders', orderRoutes)
```

Figure 16. Routes and importing router-level middleware.

Figure 16 shows four main routes of the application: product, auth, user, and order routes. All routes, except the auth route, have multiple sub routes inside it. A sub route may deal with GET, POST, PUT or DELETE requests, and can be gated to prevent unauthorized users from accessing it. For instance, the route `"/api/users/profile"`, which is a sub route of `"/api/users"`, can only be accessed by logged in users, therefore it is guarded with the `"protected"` middleware.

4.2 Frontend development

The development of the application's frontend included the Redux integration, frontend user authentication, Paypal integration, and the development of various different pages in the application.

4.2.1 Redux integration

Redux is used to store the global state in this project. For example, some pieces of state, such as the store products are called in many places, making sense to make them available to all components. Even though it is possible to put all the state to the top level component and pass pieces of state down through props, it will get messy for an application of this size. To integrate Redux into the application, the Redux store was first set up. Figure 17 displays the creation of the application's Redux store:

```
1 import { createStore, combineReducers, applyMiddleware } from "redux";
2 import thunk from "redux-thunk";
3 import { composeWithDevTools } from "redux-devtools-extension";
4 import { productsReducer } from "../reducers/productReducers";
5
6 const reducer = combineReducers({
7   products: productsReducer,
8 });
9
10 const initialState = {};
11
12 const store = createStore(
13   reducer,
14   initialState,
15   composeWithDevTools(applyMiddleware(thunk))
16 );
17
18 export default store;
```

Figure 17. The Redux store of the application.

As explained in Figure 17, on line 6, the main reducer was initialized by combining smaller reducers, each controlling a piece of state in the application. Afterward, the store was implemented using the `createStore()` function from Redux, which took in the main reducer, the `initialState`, and the enhancer composed of the Redux dev tool and the `redux-thunk` middleware. The Redux dev tool helps visualize the Redux store on the browser while developing the application, while the `redux-thunk` middleware provides the ability to use asynchronous actions in Redux, which is very useful in dealing with data fetching.

Down the road, the Redux store was exported to be used to instantiate Redux in the application. Figure 18 below displays the integration of Redux to the application:

```
1 import { Provider } from "react-redux";
2 import store from "./store";
3 import App from "./App";
4
5 ReactDOM.render(
6   <Provider store={store}>
7     <App />
8   </Provider>,
9   document.getElementById("root")
10 );
11
```

Figure 18. Redux integration to the application in index.js file.

The Provider component from the “react-redux” package was imported as described in Figure 18, which took in the earlier exported store. The Provider passed down the Redux store to any nested component inside it. The App component was then imported and used inside the Provider, so that all the App's child components have access to the global store and can get information from the store.

4.2.2 Home page

The Home page is the page displayed by default when a user comes to the application. While building the Home page, the Redux flow with data fetching is carefully discussed since this flow is reused in many other pages. Figure 19 illustrates the appearance of the Home page with products of the shop.

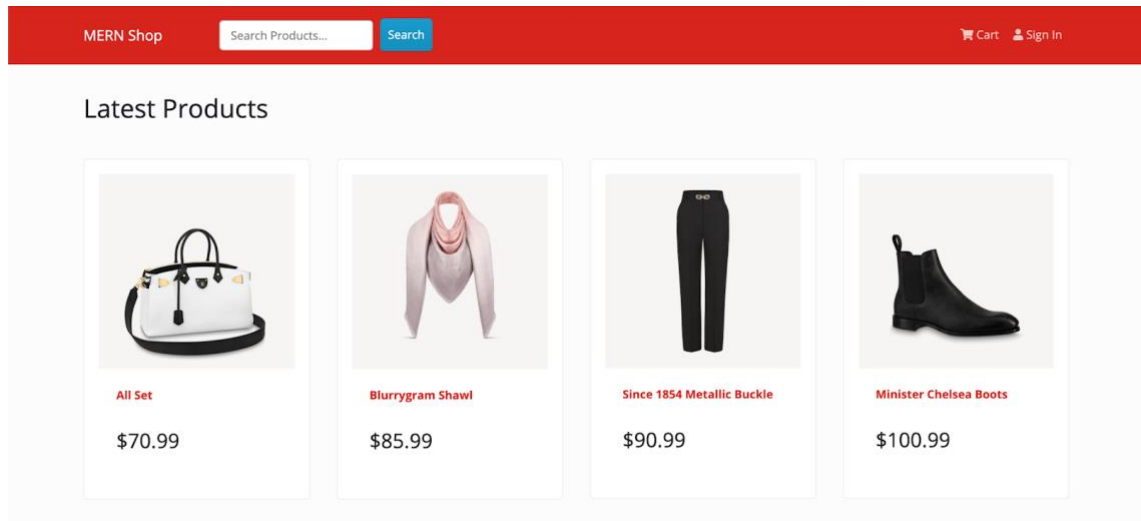


Figure 19. Home page of the application.

As shown in Figure 19, the Home page displays the products of the shop. Initially, when the user comes to this page, the products will be fetched from the server to the client, and the products will be mapped to create the ProductCard components. The ProductCard components will then be displayed in the page.

The products fetching for the Home page was carried out using Redux. First the reducer for the “products” piece of state was created. Figure 20 presents the code for the reducer and action types for products.

```

1 export const PRODUCTS_REQUEST = "PRODUCTS_REQUEST";
2 export const PRODUCTS_SUCCESS = "PRODUCTS_SUCCESS";
3 export const PRODUCTS_ERROR = "PRODUCTS_ERROR";
4
5 export const STATUS = {
6   REQUEST: "REQUEST",
7   SUCCESS: "SUCCESS",
8   ERROR: "ERROR",
9 };
10
11 const initialState = {
12   products: [],
13   error: "",
14   status: STATUS.REQUEST,
15 };
16
17 export const productsReducer = (state = initialState, action) => {
18   switch (action.type) {
19     case PRODUCTS_REQUEST:
20       return { status: STATUS.REQUEST, products: [] };
21     case PRODUCTS_SUCCESS:
22       return { status: STATUS.SUCCESS, products: action.payload };
23     case PRODUCTS_ERROR:
24       return { status: STATUS.ERROR, error: action.payload, products: [] };
25     default:
26       return state;
27   }
28 }

```

Figure 20. Reducer and action types for products.

As illustrated in Figure 20, `PRODUCT_REQUEST`, `PRODUCT_SUCCESS`, and `PRODUCT_ERROR` are only three needed action types for products, representing three statuses of data fetching: data is fetching, data fetching is successful, and data fetching is not successful. The `STATUS` constant also has three properties of `REQUEST`, `SUCCESS`, and `ERROR`, where each mimics a data fetching status. The store's product piece has an object's shape, including `products`, `status`, and `error`. If the action type is `PRODUCTS_REQUEST`, the status in the store's product piece will be `REQUEST`. If the action type is `PRODUCTS_SUCCESS`, the status will be `SUCCESS`, and the products from the action payload will be placed to the Redux store. Finally, if the API call does not succeed, the status will be `ERROR`, and the error will be included in the Redux store.

The reducer and action types are applied for data fetching of the products for Home screen after that. Figure 21 displays the whole data flow of fetching products:

```

1 import React, {useEffect} from "react";
2 import { useDispatch, useSelector } from "react-redux";
3
4 export const fetchProducts = () => (dispatch) => {
5   dispatch({ type: PRODUCTS_REQUEST });
6   axios
7     .get(`/api/products`)
8     .then(({ data }) => dispatch({ type: PRODUCTS_SUCCESS, payload: data}))
9     .catch((error) =>
10       dispatch({ type: PRODUCTS_ERROR, payload:
11         error.response && error.response.data.message
12           ? error.response.data.message
13             : error.message
14       })
15     );
16 };
17
18 const HomePage = () => {
19   const dispatch = useDispatch();
20   const { status, error, products } = useSelector((state) => state.productsList);
21
22   useEffect(() => {
23     dispatch(fetchProducts());
24   }, [dispatch, keyword]);
25   ...
26 };

```

Figure 21. Data flow of fetching products.

As can be seen from Figure 21, on line 19, the Redux store is connected to the application, and the product piece is taken from the Redux store using the `useSelector` hook that was made available. The status, error, and product properties are next destructured from the Redux store's products piece of state and used for displaying information inside the `HomePage` component. On line 18, a `dispatch` variable is instantiated using the `useDispatch` hook from `react-redux` and can dispatch actions to the Redux store. When the `HomePage` component is mounted, the `useEffect` hook runs afterward and dispatches an action by invoking the `fetchProducts` function, which is carried out on line 4.

After being invoked, the `fetchProducts()` function dispatches an action having the type of `PRODUCTS_REQUEST`, which indicates that the data fetching started then. The products' data is later fetched from the `/api/products` route on the backend using the "axios" package. If the data fetching succeeds, `fetchProducts()` function will dispatch the action type `PRODUCTS_SUCCESS` having the server's payload of products' data. If the data fetching fails, the `fetchProducts()` function will dispatch the action type `PRODUCTS_ERROR`, and the action will contain a payload of the error which occurs

while fetching the data. The status, error, and product variables obtained from Redux's products state will display the Home page correspondingly. The logic for displaying the Home page is written in Figure 22:

```

1 const HomePage = () => {
2   ...
3   const { status, error, products } = useSelector((state) => state.products);
4
5   return (
6     <>
7       <h1>Latest Products</h1>
8       {status === STATUS.REQUEST && <Loader />}
9       {status === STATUS.ERROR && <Message variant="danger">{error}</Message>}
10      {status === STATUS.SUCCESS && (
11        <Row>
12          {products.map((product) => (
13            <Col key={product._id} sm={12} md={6} lg={4} xl={3}>
14              <ProductCard product={product} />
15            </Col>
16          ))}
17        </Row>
18      )}
19    </>
20  );
21 };
22

```

Figure 22. Logic to display the Home page.

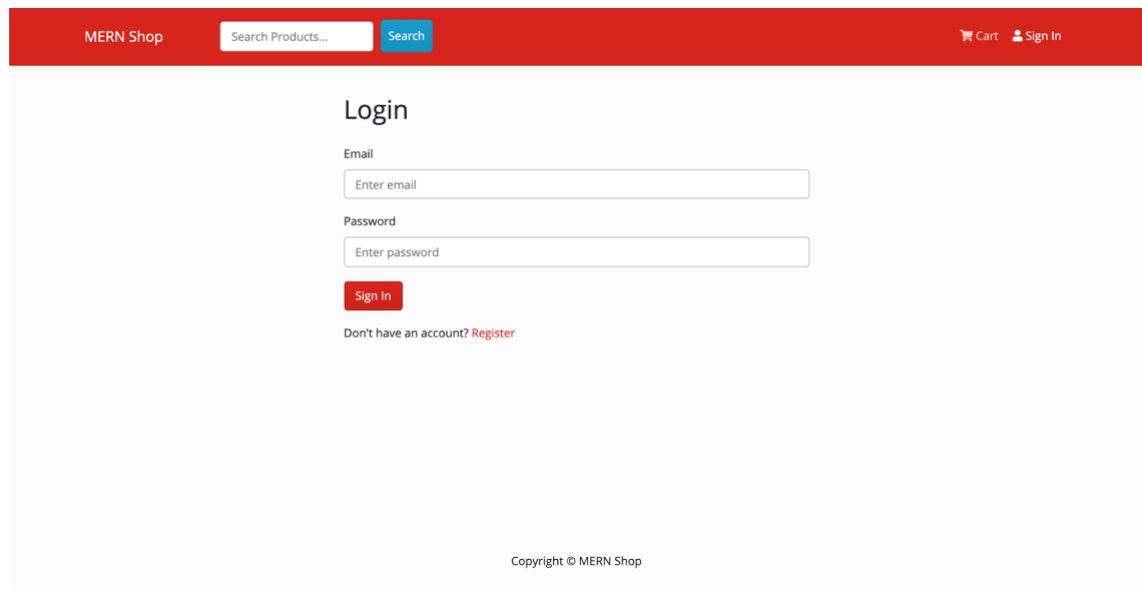
According to Figure 22, based on data fetching status, the Home page component displays differently for each type of status. If the data fetching is in progress, the data fetching status is the REQUEST key of STATUS constant, which will be REQUEST, and the page will show a loader. A loader improves user experience and signifies the user that data fetching is happening. If the status is ERROR, meaning the data fetching fails, a failure message will be rendered with the content of the occurred error. Otherwise, if the data fetching is successful, the products will be populated and mapped through to create the ProductCard components seen in the Home page.

4.2.3 User authentication

The frontend user authentication for the application consists of Login page, login and logout functionalities, in conjunction with private and admin frontend routes.

4.2.3.1 Login page and login functionality

The Login page is where a user can log in to the application to do tasks that only a logged-in user can do, such as add products to cart, order products, or pay for the orders. The Login page also has the header shared for every page in the application. The view of the Login page is specified in Figure 23:



The screenshot shows the login page of the MERN Shop application. At the top, there is a red navigation bar containing the text 'MERN Shop', a search bar with the placeholder 'Search Products...', a blue 'Search' button, and links for 'Cart' and 'Sign In'. Below the header, the page title 'Login' is centered. The form consists of two input fields: 'Email' with the placeholder 'Enter email' and 'Password' with the placeholder 'Enter password'. A red 'Sign In' button is positioned below the password field. Below the button, there is a link that says 'Don't have an account? Register'. At the bottom of the page, the text 'Copyright © MERN Shop' is displayed.

Figure 23. Login page of the application.

As illustrated in Figure 23, the Login page consists of a form with two input components, which governs the email and password that the user passes in, and a submit button to send the login request. When the user enters email and password, the email and password will be saved in state using useState hooks. To better illustrate the Login page's behavior, the code of Login Page is displayed in Figure 24 below:

```

1 const LoginPage = ({ location, history }) => {
2   const [email, setEmail] = useState("");
3   const [password, setPassword] = useState("");
4
5   const dispatch = useDispatch();
6   const { status, error, userInfo } = useSelector((state) => state.userLogin);
7
8   useEffect(() => {
9     if (userInfo) history.push('/');
10  }, [history, userInfo, redirect]);
11
12  const submitHandler = (e) => {
13    dispatch(login(email, password));
14  };
15
16  return (
17    <>
18      <h1>Login</h1>
19      {status === STATUS.ERROR && <Message variant="danger">{error}</Message>}
20      {status === STATUS.SUCCESS && <Loader />}
21      <Form onSubmit={submitHandler}>
22        <Form.Group controlId="email">
23          <Form.Label>Email</Form.Label>
24          <Form.Control
25            type="email"
26            placeholder="Enter email"
27            value={email}
28            onChange={(e) => setEmail(e.target.value)}
29          ></Form.Control>
30        </Form.Group>
31        ...
32      </>
33    );
34  };
35
36  export default LoginPage;

```

Figure 24. The logic of Login page.

As can be viewed in Figure 24, the email and password variables are set initially to empty strings using the `useState` hook. Subsequently, the component is connected to Redux using `useSelector` hook, and after that, the `userLogin` piece of Redux state was taken from the Redux store. The `userLogin` piece of state indicates whether the user has logged in or not. If the user is logged in, the Login page will redirect the user to the Home page showing the shop's products since a logged-in user cannot log in a second time. The form inputs will set email and password values, which are set in state when the user types in the inputs. Finally, when the user submits the login request by clicking the submit button, the page will dispatch actions that are formed by executing the `login()` function with the email and password that the user entered. The `login()` function is used for user login and dispatches actions correspondingly to save the user information to local

storage. Henceforth, the user information will be available to be used throughout the application. For better understanding, Figure 25 displays the details of the login() function.

```

1 export const login = (email, password) => async (dispatch) => {
2   try {
3     dispatch({ type: USER_LOGIN_REQUEST, })
4
5     const config = {
6       headers: {
7         'Content-Type': 'application/json',
8       },
9     }
10
11    const { data } = await axios.post('/api/users/login',
12      { email, password },
13      config
14    )
15
16    dispatch({ type: USER_LOGIN_SUCCESS, payload: data, })
17
18    localStorage.setItem('userInfo', JSON.stringify(data))
19  } catch (error) {
20    dispatch({
21      type: USER_LOGIN_FAIL,
22      payload: error.response && error.response.data.message
23        ? error.response.data.message
24        : error.message,
25    })
26  }
27 }
28

```

Figure 25. Login function.

As can be observed in Figure 25, initially, the login() function dispatches an action of type USER_LOGIN_REQUEST, which enables the loader to be shown to the user while waiting for logging in. A config object specifying the Content-Type header to be "application/json" will be then created to tell the server that the type of content to be sent to the server will be of JSON type. a POST request to "api/users/login" route on the server will be executed, sending the user's email and password to the server and passing the config object initialized before. Afterward, if the request is successful, an action of type USER_LOGIN_SUCCESS will be dispatched, the user info with JWT token will be populated inside the Redux store, and user info will be added to local storage. Besides, the JWT token will be added to the Authorization headers in the subsequent requests to

access protected routes on the backend. On the contrary, if the request fails, an action of type `USER_LOGIN_FAIL` carrying the error will be dispatched.

4.2.3.2 Logout functionality

If the user is logged in, the header will have a dropdown displaying the username. When the user clicks on the dropdown, there will be a logout option, as visualized in Figure 26.



Figure 26. Header of the application with logout functionality.

In Figure 26, the "Logout" button arises when the user clicks on the dropdown that has the username. After clicking the button, the application will dispatch an action to clear the Redux store's user information. Furthermore, the local storage information of the user will be cleared.

4.2.3.3 Private routes and Admin routes

Each of the three types of users, which consists of not logged in users, logged in users, and admin users, has different permissions when using the application. For instance, admins will have access to add and edit products, unlike other types of users. Therefore, gating the view access for each type of user is necessary. In order to facilitate this, protected routes to gate access are implemented. For gating pages that require the user to log in, a component called `PrivateRoute` is employed, whose logic can be seen in Figure 27 below.


```

1 import React from "react";
2 import { Route, Redirect } from "react-router-dom";
3
4 export const PrivateRoute = ({ component: Component, ...rest }) => {
5   const { userInfo } = useSelector((state) => state.userLogin);
6
7   return (
8     <Route
9       {...rest}
10      render={() =>
11        userInfo ? (
12          <Component {...props} />
13        ) : (
14          <Redirect
15            to={{
16              pathname: "/signin",
17              state: { from: props.location }
18            }}
19          />
20        )
21      )
22    />
23  );
24 };
25
26 const App = () => {
27   return (
28     <Router>
29       ...
30       <PrivateRoute path="/profile" component={ProfilePage} />
31       ...
32     </Router>
33   );
34 };

```

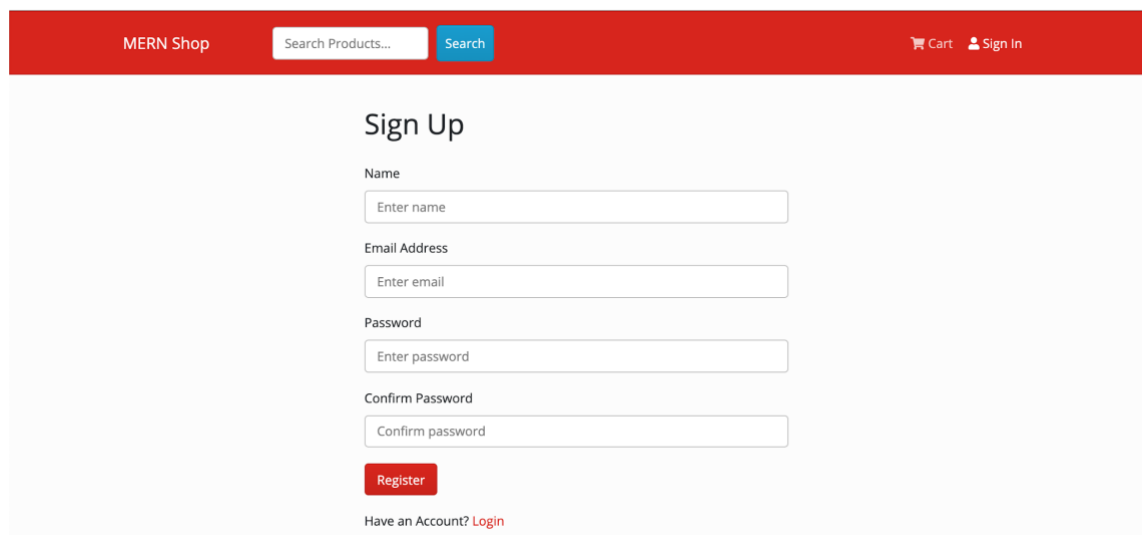
Figure 27. Logic and usage of PrivateRoute component.

From Figure 27, on line 30, the PrivateRoute component takes a path and a component. After that, the PrivateRoute component passes down the path to the Route component inside it, which means that the PrivateRoute component and its child Route component will be displayed when the user goes to the path passed into the PrivateRoute component. Initially, when the PrivateRoute renders, the user information is taken from the Redux store, and if there is such data in the Redux store, the user is logged in. A check happens on line 11 to see whether the user has logged in; if that is the case, the component passed into the PrivateRoute will be rendered. Or else, the user will be redirected to the "/signin" route where the user can sign in with the Login page. As a result, the user profile page is gated so that only logged in users can view it when coming to the "/profile" path.

Similarly, to restrict access to let only admin users view a page, a component named `AdminRoute` was created. The only difference between it and the `PrivateRoute` component is that the `AdminRoute` component checks if the user is admin via `isAdmin` flag inside the Redux store's `userInfo` object. If `isAdmin` is true, indicating the user is an admin, the component passed in will show up. Oppositely, an error page will appear, notifying that the user does not have permission to view the page.

4.2.4 Sign up page

It is possible to view some of the application screens, such as the Home screen, without asking the user to sign up to improve the user experience. On the other hand, if users wish to buy on the platform, they need to sign up and log in. The user interface of the Signup page is depicted in Figure 28 below:



The screenshot shows the 'Sign Up' page of the MERN Shop. The header is red and contains the text 'MERN Shop', a search bar with the placeholder 'Search Products...', and a blue 'Search' button. On the right side of the header, there are icons for 'Cart' and 'Sign In'. The main content area is white and features a 'Sign Up' heading. Below the heading are four input fields: 'Name' (placeholder 'Enter name'), 'Email Address' (placeholder 'Enter email'), 'Password' (placeholder 'Enter password'), and 'Confirm Password' (placeholder 'Confirm password'). A red 'Register' button is positioned below the input fields. At the bottom of the form, there is a link that says 'Have an Account? Login'.

Figure 28. Sign up page.

The Signup page in Figure 28 includes a form with four input components: name, email address, password, and confirmed password. The password and confirmed password are required to match for the purpose of submitting the sign up request. After submitting valid passwords, a POST request will be sent to the `"/api/users"` route on the backend with user name, email and password under JSON format. If a user with the same email address exists, an error stating that the user already exists will be sent back from the

server and displayed on the page. Contrarily the user will be successfully created in the database. If user sign up is successful, the frontend will sign the user in.

4.2.5 Product page and Cart page

The Product page is the place where a user can find more detailed information about the product. Generally, the user can view the product name, description, price, image, and whether the product is in stock or not. Furthermore, a logged-in user will have the ability to select the product's quantity and add the product to the cart. After adding the product to the cart, the product will be sent to the local storage, and the user will be redirected to the cart page.

The Cart page presents all the products that were added to cart. In this page, logged-in users have the ability to remove a product from the cart and change the quantity of the products in the cart. Each product is shown with a small thumbnail, the price, and the quantity. A total price is also presented with a button to proceed to checkout.

4.2.6 Checkout page and Paypal integration

After clicking the proceed to checkout button, the user will be redirected to the Checkout page. The user interface of the Checkout page is illustrated in Figure 29 below.

The screenshot displays the checkout interface for MERNShop. At the top, there is a red navigation bar with the store name 'MERNShop', a search bar, and a user profile 'Tien'. The main content area is titled 'Order 5f9bcefe163a9b381c068806' and is divided into several sections:

- Shipping:** Shows the user's name (Tien), email (tien@example.com), and address (1 Road 1, Helsinki 12341, Finland). A status bar indicates 'Not Delivered'.
- Payment Method:** Shows the selected method as 'PayPal' and a status bar indicating 'Not Paid'.
- Order Items:** Lists three items: 'All Set' (3 x \$70.99 = \$212.99), 'Minister Chelsea Boots' (1 x \$100.99 = \$100.99), and 'Blurrygram Shawl' (1 x \$85.99 = \$85.99). An 'Edit' button is located below the list.
- Order Summary:** A table showing the breakdown of costs:

Items	\$399.95
Shipping	\$0
Tax	\$59.99
Total	\$459.94

 Below the summary are two payment buttons: a yellow 'PayPal' button and a dark grey 'Debit or Credit Card' button. A note at the bottom of the summary states 'Powered by PayPal'.

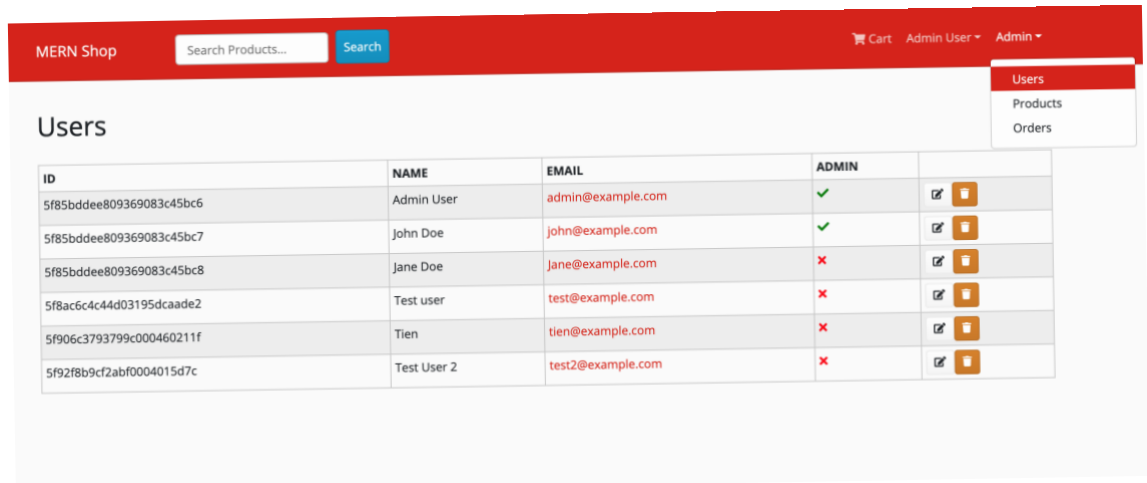
Figure 29. User interface of Checkout page.

In Figure 29, the shipping address, payment method, and order items are laid out in the Checkout page, where logged-in users can edit by clicking the Edit button. Only Paypal payment method is available in the application, but other payment methods can be later employed to facilitate user payment. The page's right side contains the order summary with items' price, shipping price, tax, and total price. Underneath, Paypal payment button is rendered using the "react-paypal-button-v2" package provided by Paypal. The Paypal script is inserted to the page when the page loads using useEffect hook.

When the user clicks on Paypal payment button, a payment modal will be prompted on the page. After signing in to Paypal using the login form inside the modal, the user will be able to make the purchase. In order to test Paypal integration and mimic real transactions, sandbox accounts were created for the shop and the users. Finally, if the payment is successful, the order's status will be set to "paid" and reflected on the frontend.

4.2.7 Admin pages

For all admin pages, access is gated, and users must be admins to view these pages. First and foremost, the admin can view the list of users and update users' information. The following Figure 30 presents the admin user list:



ID	NAME	EMAIL	ADMIN	
5f85bddee809369083c45bc6	Admin User	admin@example.com	✓	✎ 🗑
5f85bddee809369083c45bc7	John Doe	john@example.com	✓	✎ 🗑
5f85bddee809369083c45bc8	Jane Doe	jane@example.com	✗	✎ 🗑
5f8ac6c4c44d03195dcaade2	Test user	test@example.com	✗	✎ 🗑
5f906c3793799c000460211f	Tien	tien@example.com	✗	✎ 🗑
5f92f8b9cf2abf0004015d7c	Test User 2	test2@example.com	✗	✎ 🗑

Figure 30. Admin User list page.

As demonstrated in Figure 30, the admin User list page contains a table featuring all the users in the application, where each table row consists of user id, username, user email, and if the user is admin. Furthermore, options to edit and remove users are provided on each row. Additionally, the application header has one more dropdown for admin users, which will show navigation to admin pages when clicked. Admin users can also view the list of products, edit products, and add new products. When adding a new product, admin users can enter product name, price, product image, brand, category, count in stock, and product description. Besides, admin users can set orders to be delivered.

4.3 Deployment on Heroku

In order to create a production build for the React frontend of the application, the command "npm run build" on the frontend was utilized, which created the production build in the "frontend/build" folder. Afterward, the application was deployed to Heroku, a deployment platform. To deploy to Heroku, Heroku CLI was installed, and the login to Heroku was carried out from the terminal. Subsequently, the command "heroku create"

was run on the terminal to create a new Heroku application. Besides, a file named Procfile including script to run the server, and "heroku-postbuild" command were employed to start the Heroku application. On the Heroku platform, environment variables were then entered, and after a short time, the application was successfully built and deployed.

5 Discussion

In the end, the online shop application using the MERN stack technologies was successfully developed. As a result, it is evident that the MERN stack is capable of constructing a complex full-stack application. MongoDB, Express, React, and Node were used in conjunction with numerous tools and packages to deliver the final application. Those technologies and tools used were carefully explained throughout the thesis.

Overall, the end application met all the requirements defined at the beginning of the development phase. Any visitor coming to the application can view the products on the home page and select a product to see its information. After signing up and logging in, users can update their profiles, add products to cart, proceed to checkout, and pay for the orders. After paying for their orders, users can keep track of the orders on the Order page. Figure 31 displays the user interface of the Order page.

Order 5f9dbb03c79d660ce407ba4d

Shipping
 Name: Tien
 Email: tien@example.com
 Address: 1 Road 1, Helsinki 12341, Finland
 Not Delivered

Payment Method
 Method: PayPal
 Paid on 2020-10-31T19:29:48.176Z

Order Items

All Set	3 x \$70.99 = \$212.96999999999997
Blurrygram Shawl	1 x \$85.99 = \$85.99

Order Summary

Items	\$389.95
Shipping	\$0
Tax	\$58.49
Total	\$448.44

Figure 31: The Order page.

From Figure 31, after paying, this particular user can see that the order was paid, and the user can keep track of whether or not the order was delivered. In the Order screen, users can also see ordered items, the shipping price, tax, total price, and the user's personal information. The Order page is the final step of the purchasing flow in the application.

For admin users with higher privileges, they can update users' information, update products, add products, and update the orders' statuses to be delivered. In general, the end application is intuitive and easy to use. Only after a few steps, any visitor can go from non logged-in to making purchases on the platform.

Nevertheless, the application can still be expanded with more features. Paypal is the only payment method available for users in the application currently. More payment methods, for instance, Stripe, Braintree, or Wepay, can be added. Furthermore, social media signup and sign-in could be integrated into the application to facilitate the sign-in process to make purchases faster and improve user experience. Finally, product rating and review could also be implemented for users to rate products and see product ratings and reviews for the decision making process.

Besides, several advanced techniques and concepts could be applied to improve several aspects of the application further. First, server-side rendering can replace the traditional client-side rendering that comes in with the "create-react-app" package used in the application. Server-side rendering will provide search engines better access to the page's content, which improves Search Engine Optimization (SEO) to rank the page higher while user search in engines such as Google. Testing, which is a vital part of building any reliable application, is not dealt with in this application. Popular and powerful tools, for instance, Jest, react-testing-library, and Cypress, could be brought into the application to perform different testing techniques, such as unit testing, integration testing, and end-to-end testing. Finally, code splitting, which is the technique of only loading needed pieces of the application, can be implemented to improve the application's performance.

6 Conclusion

The thesis aimed to analyze and describe the MERN stack's functionalities and core characteristics and develop an online shop application using the MERN stack. Each technology comprising the MERN stack was examined in detail, together with the traits and the relevance of the whole MERN stack. The modern practices, core concepts of MERN stack, and libraries that complement the MERN stack used to create the application were researched carefully in the thesis.

Eventually, a working and production-ready online shop application was built and deployed successfully. Any visitor can see the products and the details of the products, and within a few steps, any visitor can sign up, add products to cart and pay for the orders using Paypal. With higher privileges, admin users can update products, mark orders as paid, update the user database, and promote users to be admins. Generally, the application satisfied all the predefined requirements from the start of the project.

The thesis can be treated as an extensive guide to the MERN stack and can help those who want to learn more about MERN stack development. Eventually, the MERN stack proved to be capable of building rather complex full-stack applications. However, the end application could still be improved by adding new features, such as new payment methods, social media login, and product rating functionality. Additional advanced concepts, such as server-side rendering, code splitting, and testing, could still be added to improve different aspects of the application.

References

- 1 MERN Stack [online] MongoDB
Available at: <https://www.mongodb.com/mern-stack>. Accessed 1 October 2020.
- 2 Valeri Karpov. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js [online] MongoDB; 2017
Available at: <https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and> Accessed 2 October 2020.
- 3 Eliot Horowitz. A Founder's Reflections on 10 Years of MongoDB [online] MongoDB; 2017
Available at: <https://www.mongodb.com/blog/post/a-founders-reflections-on-10-years-of-mongodb>. Accessed 3 October 2020.
- 4 Shama Hoque. Full-stack React Projects Second Edition. Birmingham, UK: Packt Publishing; 2020.
- 5 Vasam Subramanian. Pro MERN Stack. Bangalore, India: Apress; 2017.
- 6 Express's Official Website [online] Express
Available at: <https://expressjs.com>. Accessed 5 October 2020.
- 7 Can Ho. Understanding the Middleware Pattern in Express.js [online] Dzone; 2016
Available at: <https://dzone.com/articles/understanding-middleware-pattern-in-expressjs>. Accessed 7 October 2020.
- 8 Using middleware [online] Express
Available at: <https://expressjs.com/en/guide/using-middleware.html>. Accessed 9 October 2020.
- 9 Cory Gackenheimer. Introduction to React. New York, USA: Apress; 2015.
- 10 Virtual DOM and internals [online] Facebook
Available at: <https://reactjs.org/docs/faq-internals.html>. Accessed 15 October 2020.
- 11 Components and Props [online] Facebook
Available at: <https://reactjs.org/docs/components-and-props.html>. Accessed 16 October 2020.
- 12 Introducing Hooks [online] Facebook
Available at: <https://reactjs.org/docs/hooks-intro.html>. Accessed 16 October 2020.
- 13 Using the State Hook [online] Facebook
Available at: <https://reactjs.org/docs/hooks-state.html>. Accessed 16 October 2020.

- 14 A brief history of Node.js [online] OpenJS Foundation
Available at: <https://nodejs.dev/learn/a-brief-history-of-nodejs>. Accessed 16 October 2020.
- 15 Differences between Node.js and the Browser [online] OpenJS Foundation
Available at: <https://nodejs.dev/learn/differences-between-nodejs-and-the-browser>. Accessed 17 October 2020.
- 16 Paul Brown. State of the Union: npm [online] Linux; 2017
Available at: <https://www.linux.com/news/state-union-npm>. Accessed 17 October 2020.
- 17 Christoph Heike. LAMP vs. MEAN: Which stack is right for you? [online] Bitbucket; 2019
Available at: <https://bitbucket.org/blog/lamp-vs-mean-which-stack-is-right-for-you>. Accessed 18 October 2020.
- 18 Mardan A. Practical Node.js. Berkeley, USA: Apress; 2018.
- 19 Nick Karnik. Introduction to Mongoose for MongoDB [online] FreeCodeCamp; 2018
Available at: <https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57>. Accessed 18 October 2020.
- 20 Mongoose Guide [online] Mongoose
Available at: <https://mongoosejs.com/docs/guide.html>. Accessed 18 October 2020.
- 21 Models [online] Mongoose
Available at: <https://mongoosejs.com/docs/models.html>. Accessed 19 October 2020.
- 22 Daniel Bugl. Learning Redux. Birmingham, UK: Packt Publishing; 2017
- 23 Alex Bachuk. Redux - An Introduction [online] Smashing Magazine; 2016
Available at: <https://www.smashingmagazine.com/2016/06/an-introduction-to-redux>. Accessed 19 October 2020.
- 24 Core Concepts [online] Redux
Available at: <https://redux.js.org/introduction/core-concepts>. Accessed 19 October 2020.
- 25 Authentication and Authorization [online] Auth0
Available at: <https://auth0.com/docs/authorization/authentication-and-authorization>. Accessed 19 October 2020.
- 26 Git Handbook [online] Github; 2020
Available at: <https://guides.github.com/introduction/git-handbook>. Accessed 19 October 2020.

- 27 Why did we build Visual Studio Code? [online] Visual Studio Code; 2020
Available at: <https://code.visualstudio.com/docs/editor/whyvscode>. Accessed 19 October 2020.
- 28 Using middleware [online] Express
Available at: <https://expressjs.com/en/guide/using-middleware.html#middleware.router>. Accessed 20 October 2020.