Bachelor's thesis

Information and Communication Technology

2022

Thi Thu Hien Tran

THE DEVELOPMENT OF AN E-COMMERCE WEB APPLICATION USING MERN STACK

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | 77 pages

Thi Thu Hien Tran

THE DEVELOPMENT OF AN E-COMMERCE WEB APPLICATION USING MERN STACK

The web development industry has grown significantly with the evolution of technology. Software technology is improving at the same rate as hardware. Increasingly, electronic devices with Internet and real-time capabilities have made performance essential. Unfortunately, despite their general use and long history of development and maintenance of traditional technology, some fail to meet today's customer performance expectations. The MERN stack was recently built to overcome this performance issue.

E-commerce has exponentially risen over the last decade, offering more benefits and conveniences than traditional businesses. Furthermore, the COVID-19 era has irreversibly changed the way businesses interact with customers, allowing merchants to approach clients more promptly. By recognizing this need, an E-commerce web application is created in the form of an online bookshop.

This thesis will demonstrate and comprehend each MERN stack technology's essential concept before building a functional E-commerce web application to help small companies formulate their business strategy.

The functionality and browser capabilities tests were all conducted on various platforms. The outcome was satisfactory since the web application fulfilled all of the objectives.

Finally, an operational and production-ready web store application was successfully constructed and deployed. Additionally, the application's results and potential enhancements were discussed. The thesis can be utilized as a reference on the MERN stack, aimed at novices and anyone enthusiastic about exploring the technology stack.

Keywords:

React, MongoDB, NodeJS, Express, MERN Stack, E-commerce

Content

Figures

Tables

List of abbreviations

API	Application Programming Interface
BSON	Binary encoded JavaScript Object Notation
B2C	Business-to-Consumer
CLI	Command-line Interface
CSS	Cascading Style Sheets
CORS	Cross-Origin Resource Sharing
DOM	Document Object Model
CSRF	Cross-site Request Forgery
ECMA	European Computer Manufacturer’s Association
Express	Acronym to ExpressJS
E2E	End-to-End
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
MEAN	MongoDB, AngularJS, ReactJS, NodeJS
MERN	MongoDB, ExpressJS, ReactJS, NodeJS
NPM	Node Package Manager
ODM	Object Data Modeling
RDBMS	Relational Database Management System
REST	Representational State Transfer

SQL	Structured Query Language
SEO	Seach Engine Optimization
SSR	Server-side Rendering
React	Acronym to ReactJS
UI	User Interface
UNCTAD	United Nations Conference on Trade and Development
VSCode	Visual Studio Code
XML	Extensible Markup Language
XSS	Cross-site Scripting

1 Introduction

Nowadays, with the advancement of technology, the web development industry has seen significant growth. As physical equipment improves at an accelerated rate, so does software technology. In addition, a rise in the number of electronic gadgets with Internet and real-time capabilities has made the performance more critical. Traditional technologies such as Servlets, ASP.NET, and PHP have been the most used web development frameworks in the last several years. However, despite widespread usage and prolonged history of development and maintenance, these technologies fall short of fulfilling the performance standards of today's customers. The MERN stack, including MongoDB, Express, React, and NodeJS, was recently designed to address this performance problem owing to its simplicity and consistency.

E-commerce is a massive platform growing at an incredible rate over the previous decade, delivering more benefits and conveniences than traditional businesses. Furthermore, due to the COVID-19 era, E-commerce has irrevocably transformed the engagement procedure between businesses and customers while simultaneously enabling merchants to approach consumers more diligently (UNCTAD, 2021). As a result of recognizing this demand, the author decided to construct an E-commerce web application in the shape of an online bookstore.

The main objective of this thesis is to demonstrate and comprehend the core concept of each technology in the MERN stack. As a result, an E-commerce web application has been developed to facilitate the small companies in formulating their business strategy by taking advantage of the MERN stack as well as its relevant technologies. Both frontend and backend development will be explained in detail to assist readers in understanding the whole process of constructing this application.

This thesis consists of eight chapters. The first chapter contains the primary objective of the thesis and the technologies used. The second chapter explains the theoretical background of each technology in the mentioned stack used for

web application development. The third chapter illustrates different users' functionality along with the application's environmental setup, while the fourth chapter demonstrates the application development process from the backend to the frontend comprehensively and thoroughly. Chapter five demonstrates the application's deployment process in detail. Chapter six demonstrates the testing process, while the result of the application with potential further improvements is discussed in chapter seven. In the end, the conclusion of the thesis is presented in chapter eight.

2 Theoretical background

The thesis application is constructed based on the MERN stack as the primary fullstack technology. This chapter will briefly discuss each technology in the MERN stack as well as its third-party support library Mongoose to develop the project.

2.1 The MERN stack

MERN is one of the notable variants based on the MEAN stack. Basically, the MEAN stack was initially established in 2013 by a MongoDB engineering team as a JavaScript-based stack in order to aid in the development (MongoDB Inc., 2013). MEAN comprises four open-source components: MongoDB acts as the database, Express serves as the server framework, Angular as the client framework, and Node works as the environment for running JavaScript. By substituting the popular framework Angular with React – a client-side library – and combining them as the MERN stack, React can become a companion to the other technologies for developing JavaScript and JSON-oriented applications. Figure 1 below illustrates the architecture of the MERN stack technology:

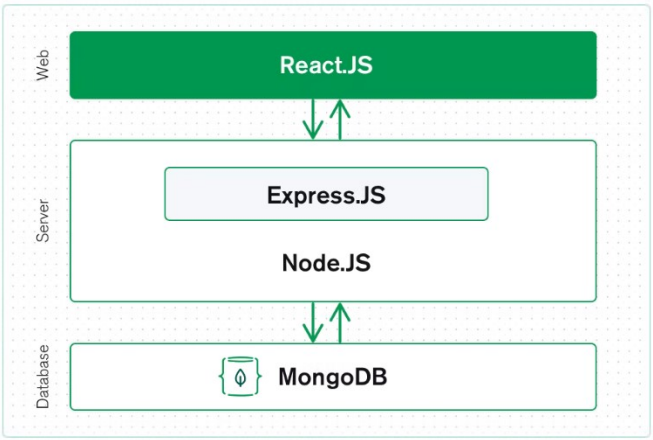


Figure 1. The architecture of the MERN stack (MongoDB n.d.).

As illustrated, the MERN stack is constructed based on the well-known 3-tier architecture, which confirms that the MERN stack is fullstack application development. The MERN stack consists of three components: display layer powered by React, application layer with NodeJS and Express, and database tier provided by MongoDB.

2.1.1 MongoDB

MongoDB is a cross-platform, open-source, NoSQL database that is mainly used for scalable large-volume data applications and tasks that do not function well in a relational database. It utilizes a document storage format known as BSON (Binary encoded JavaScript Object Notation). It is a non-relational database management system created by Dwight Merriman, Eliot Horowitz, and Kevin Ryan and became popular in the mid-2000s (Wikipedia, 2022b). MongoDB's design is based on collections and documents, as illustrated in Figure 2 below, which replace the usage of tables and rows in conventional relational databases.

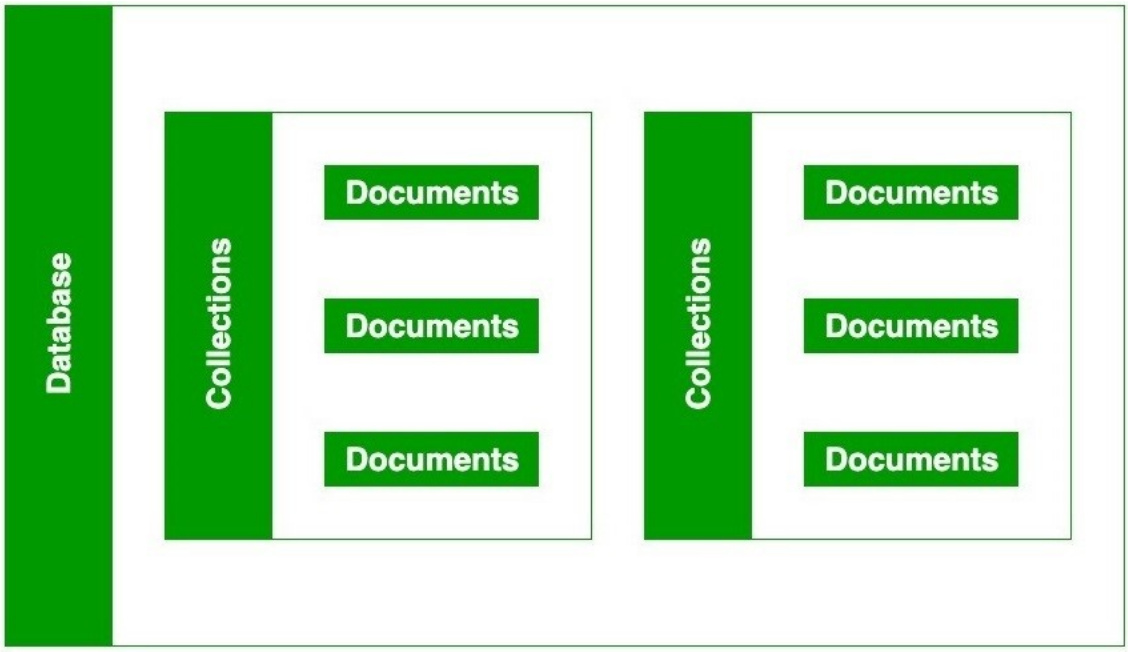


Figure 2. The architecture design of MongoDB (GeeksforGeeks, 2021).

MongoDB also supports a variety of document operations, including adding, querying, updating, and deleting. MongoDB is suitable for various use cases due to the diversity of field values and powerful query languages. In addition, its ability to scale out to accommodate larger data volumes horizontally has contributed to its increasing success as the world's most popular NoSQL database.

There are some crucial features of MongoDB:

- Scheme-less Database: This feature allows a single collection to store numerous documents, each of which contains a varied amount of fields, content, and size. Therefore, MongoDB offers tremendous flexibility to databases thanks to this fantastic feature.
- Document Oriented: All the data is kept in fields containing a clear structure with key-value pairs rather than rows and columns, which provides more flexibility than RDBMS.
- Indexing: Each field in MongoDB documents is indexed using primary and secondary indices, making retrieving and searching for data more straightforward and faster. Without the accurate indices, the database must manually search each document for the matching query, which is time-consuming and inefficient.
- Scalability: The horizontal scalability from MongoDB was offered through sharding, which refers to the process of distributing data across numerous servers. By utilizing the shard key, a significant quantity of data is divided into data chunks, and these data chunks are equally spread among shards that involve multiple physical servers. Moreover, it will add new machines to an existing database.
- Replication: With the benefit of replication, MongoDB enables high availability and redundancy by producing several copies of data and storing them on a separate server to protect the database against hardware failure, ensuring that the data can be recovered from another if one server fails.

- Aggregation: procedures on the dataset are enabled to provide a single or calculated output with three distinct aggregating methods, including pipeline aggregation, map-reduce function, and single-purpose aggregation.

2.1.2 Express

Representing the letter “E” in the MERN stack, Express is a lightweight and versatile web application framework built on top of NodeJS (OpenJS Foundation, 2022). Thanks to the large community of support, it includes a rich collection of functionality for developing web and mobile applications. Even though a large number of support packages along with the functionality for better software creation, Express does not affect the performance of NodeJS.

Based on the GitHub repository, Express was established on May 22, 2010, by T.J. Holowaychuk. After that, StrongLoop acquired the project management rights in June 2014 until IBM owned the company in September 2015. Then, in January 2016, the NodeJS Foundation took over the management of Express, and Express is now the primary function of the NodeJS platform (Wikipedia, 2022a).

Express.js is a routing and middleware framework for managing the many routing options for a website. It operates between the request and response cycles. Middleware is invoked after the server receives the request and before the controller actions transmit the response. One or more pieces of middleware are executed to perform particular tasks, such as authorizing requests or parsing request content. Express applications are composed of a sequence of middleware function calls. Typically, the first middleware executed to process the request initiates the task pipeline. The initial middleware can either complete the request and provide it to the users or call the subsequent middleware to continue the request. The same approach will be continued until the pipeline's last middleware takes the result of the preceding middleware as an input (Ho, 2016).

2.1.3 React

React, representing the letter ‘R’ in the MERN stack, focuses on creating the View Layer, which is well-known for all visible parts of the page of an application. React is a multi-purposed, open-source JavaScript library used for building user interfaces based on UI components (Facebook Inc., 2022d).

Since React was established to cope with sophisticated, large-scale user interfaces combined with real-time dynamic data and data binding, it has been steadily improving its single-page application development and frontend utilities for programmers of all levels.

Compared to other popular libraries and frameworks such as jQuery, Angular, and VueJS, React is considered the most popular framework for web development, as illustrated in Figure 3 below.

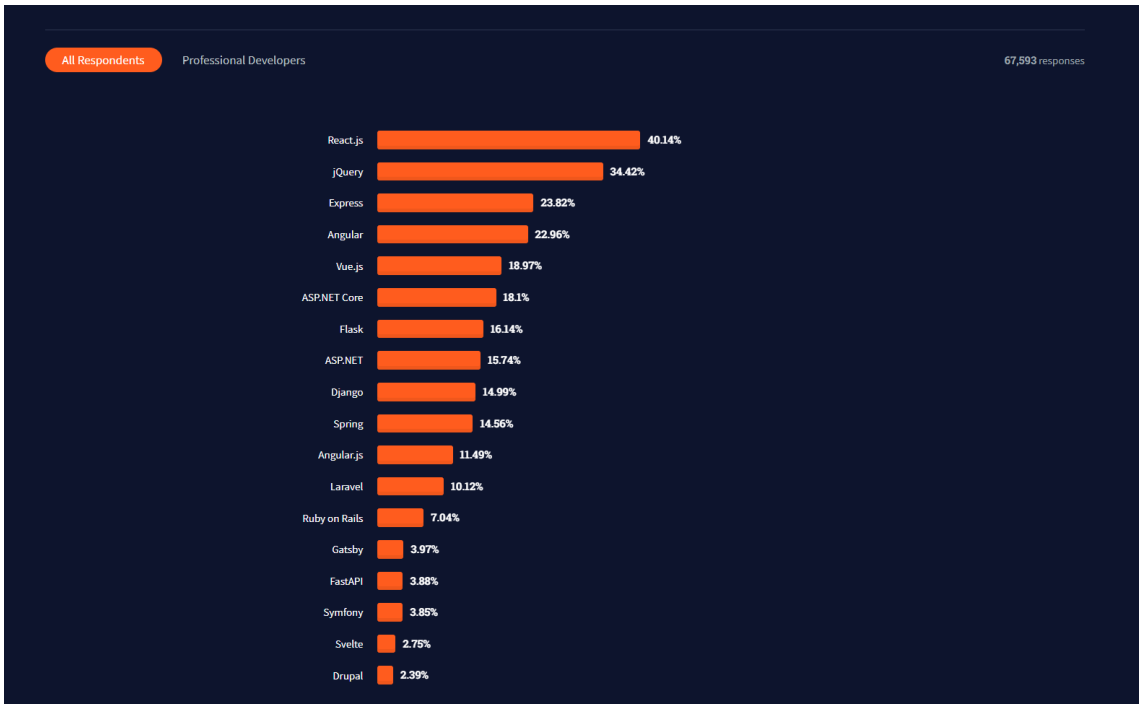


Figure 3. Most popular web frameworks (Stack Overflow, 2021).

Along with offering reusable component code, which reduces development time and the likelihood of bugs and errors, Reacts introduced many essential

characteristics that contribute to its developer appeal, which are discussed below.

JSX

JSX (JavaScript XML) is a syntactic extension to the JavaScript that is similar to HTML. Because JSX optimizes the translation to standard JavaScript and leverages its full power, it is much faster than the regular one. Even though React does not need the usage of JSX to construct the React-based applications, it is suggested as it simplifies the development for developers whenever markup components and the binding events are required (Aggarwal, 2018). Rather than dividing markup and logic into different files, JSX enables developers to generate cleaner and more manageable codebases for their websites by combining rendering logic and user interface logic in the same components (Facebook Inc., 2022c).

Virtual DOM

Virtual DOM (or VDOM) is an abstract representation of DOM (Document Object Model), and its solutions are constructed on top of the regular DOM. DOM represents the UI of the program, which its model portrays the document as a collection of different nodes and objects to interact with the structure, layout, and content of the website through programming languages (MDN Web Docs. 2022b). Figure 4 below indicates an example of the DOM structure.

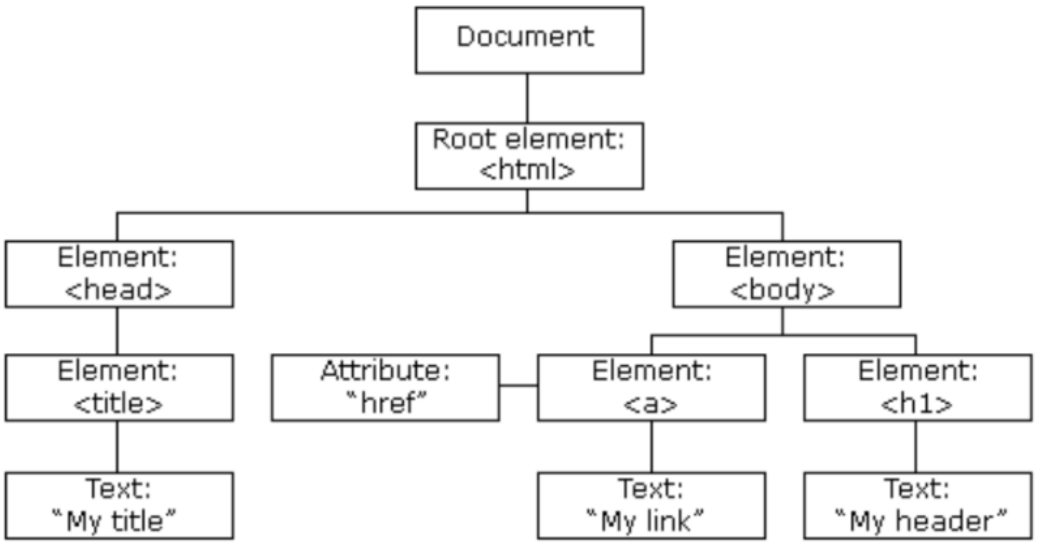


Figure 4. An example of tree-structure of Document Object Model (W3schools n.d.).

While regular DOM manipulation is slow due to the re-rendering of both the updated element and its children after getting new data, the Virtual DOM re-renders only the components that need updating, which fastens the rendering process and increases the performance.

In order to further understand why Virtual DOM is fast and practical, the functionality of Virtual DOM must be discussed. The state of the DOM tree hierarchy is stored when Virtual DOM is utilized to render a page. Instead of constructing a new tree, a diffing approach is employed when UI modifications are necessary. At that point, the React library employs the Virtual DOM, which allows it to do the calculations inside this domain without involving the actual DOM. Therefore, whenever a component's state changes, React keeps track of it and updates the Virtual DOM tree by comparing the current version to the prior one (Facebook Inc., 2022h). This comparison employs a diffing algorithm to reduce the number of DOM operations/refreshes, leading to considerable boosting speed. The whole procedure is titled Reconciliation (Facebook Inc., 2022e).

Figure 5 below demonstrates DOM and the Virtual DOM update process differences.

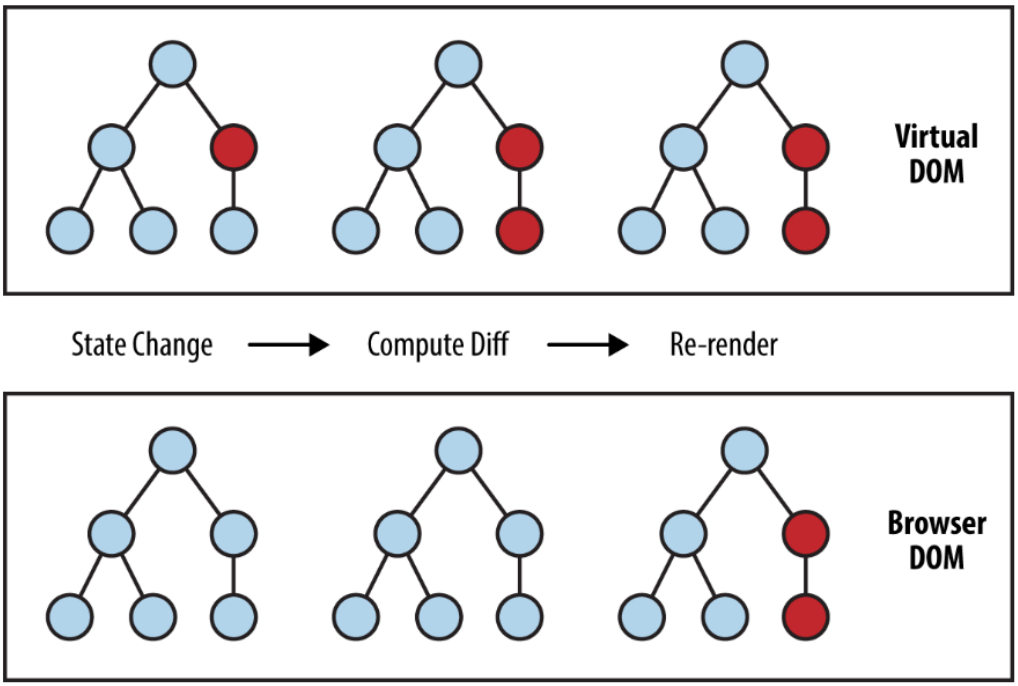


Figure 5. The difference between Virtual DOM and real DOM update process (Hamedani, 2018).

Components

Components are the primary concept of React, which encourages developers to separate the complicated user interfaces into reusable and independent parts. React components are classified into two types: class-based component and functional component. The functional component is considered the most straightforward method to construct as it can be implemented as a JavaScript function to return JSX. The class-based component is constructed using the ECMAScript6 class syntax combined with the built-in React library class “Component” (Facebook Inc., 2022a).

Components can refer to one another, which means one component can be a parent component containing several child components with no restriction on

the amount of complexity. Moreover, both functional and class-based components adhere to one strict rule assigned by React: all React components must be pure functions in which their props are immutable. Props, which stands for Properties, are a collection of inputs passed as parameters to a component, while pure function depicts the state in which the function performs the logic without affecting the arguments. As a result, a React component behaves like a pure function while respecting its inputs and rendering the same result for the same props.

Hooks

Prior to version 16.8, the majority of React components were class-based as class-based components offer life-cycle methods for component state management. Since version 16.8, however, React has introduced a new notion called Hooks, which provides a new method to leverage state and other React capabilities inside a functional component. By using Hooks, a component's stateful logic can be isolated, tested separately, and reused without affecting the component hierarchy. In addition, Hooks enables the developer to break down a component into separate functions based on the relationship instead of life-cycle methods (Facebook Inc., 2022c).

There are two built-in React hooks that should be discussed. First, the State hook, commonly known as the useState hook, enables the component-level state management. It hooks into React's state by creating a state variable that React maintains. useState hook accepts and returns two results: the current state and a function to alter it. The component state can be efficiently initialized, utilized, and modified using the useState hook (Facebook Inc., 2022g). The second hook is the Effect hook, also known as useEffect hook. useEffect hook assists programmers in managing component life cycles. The difficulty of separating related functionality and data into several class life cycles, such as componentDidUpdate, componentDidMount, and componentWillUnmount, has been thoroughly addressed by Effect Hook. A React component can support multiple effects to isolate data manipulation issues (Facebook Inc., 2022f).

2.1.4 NodeJS

NodeJS is an open-source, cross-platform JavaScript runtime environment designed for constructing scalable applications. NodeJS is independently built on top of Google Chrome's V8 runtime engine, which is well-known for working effectively outside of a browser.

By utilizing an event-driven design and operating on a single-thread event loop, NodeJS allows asynchronous and non-blocking I/O optimization to enhance web application performance and scalability, as shown in Figure 6 below. Therefore, it provides an alternative approach for developers to wait and fulfill requests for developing lightweight and real-time applications.

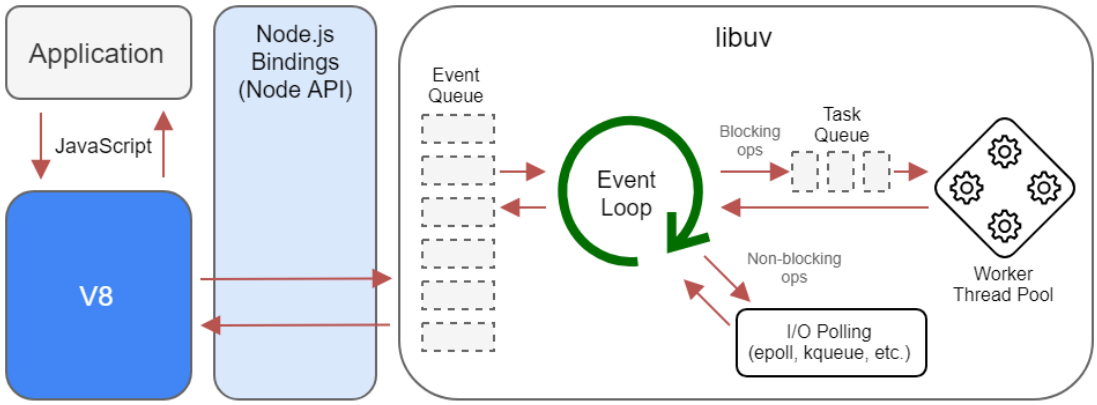


Figure 6. NodeJS event loop (Keshishyan, 2019).

Node package manager (NPM)

Node package manager (NPM) is the NodeJS default package manager for applications, and it is utilized to maintain all of the NodeJS packages and modules along with the command line client npm. Therefore, it facilitates time-consuming manual tasks by automated managing third-party packages, allowing developers to spend more time on the development process.

NPM was first published on January 12, 2010, by Isaac Z. Schlueter (Wikipedia, 2022c). It is installed alongside NodeJS and is utilized to install the required packages and modules in the NodeJS project.

It is currently the world's largest software registry, with approximately two million packages at the end of March 2022.

2.2 Mongoose

Mongoose is an object document mapping (ODM) library that is utilized for facilitating Node and MongoDB development. It is responsible for managing data relationships, performing schema validation, and serving as a middleman between objects in code and object representations in MongoDB. In addition, Mongoose offers multiple methods and functions that effectively facilitate the communication between NodeJS and MongoDB. Figure 7 below illustrates the relationship between Mongoose, NodeJS, and MongoDB.

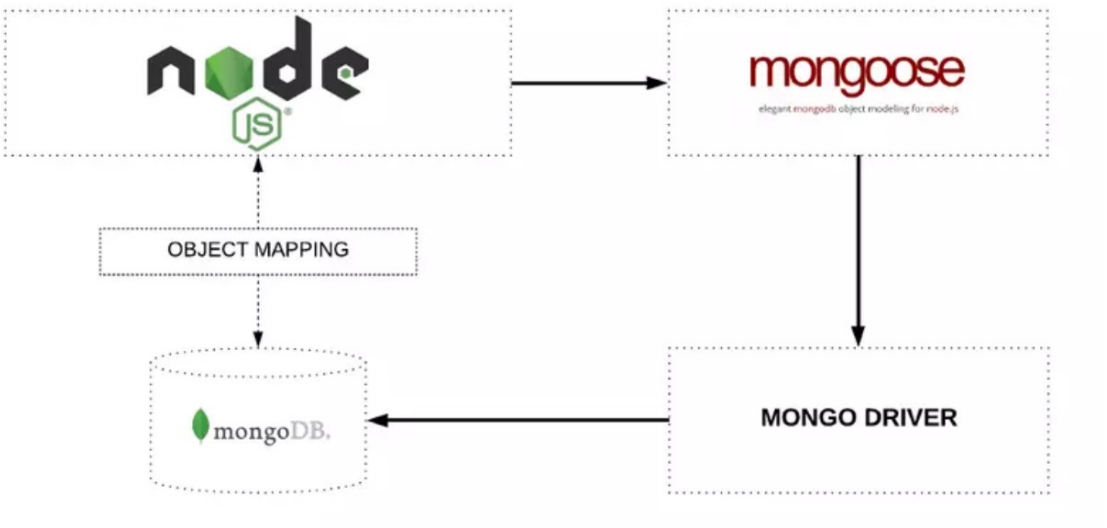


Figure 7. The relationship between Mongoose, NodeJS and MongoDB (Karnik, 2022).

As shown in Figure 7, Mongoose is utilized to create the interaction between Node and MongoDB through object mapping. After that, Mongoose forms the

connection with MongoDB using a Mongo Driver. Therefore, the relationship between Mongoose, NodeJS, and MongoDB ensures data capabilities.

The first step in getting started with Mongoose, like with other ODM libraries, is to create a schema. As Mongoose's documentation page described, a schema specifies the data structure and property casting, along with the following techniques: instance methods, compound indexes, static Model methods, and middlewares (Mongoose, n.d. (b)). Once the first stage is finished, the developed schemas will be utilized to map to MongoDB collections and shape the data documents included inside each collection. The second stage required by programmers is to construct a Mongoose model. Models are composed of builders of schemas, with the primary responsibility of producing and scanning documents in the Mongo database. Querying, deleting, and updating documents in the database are additional capabilities of models worth mentioning (Mongoose, n.d. (a)).

3 Project requirements and environmental setup

This thesis project is an E-commerce web application for an online bookstore. This chapter presents the project requirements and environmental setup for both the frontend and backend of the web application.

3.1 Project requirements

The project was based on the B2C E-commerce web application, which contains two types of users: administrator and user. Administrators are accountable for a variety of management tasks, including creating, modifying, and deleting products from the databases. On the other hand, the user can navigate and explore the information of the product and purchase the product by adding to the shopping cart and completing the transaction for that product. Some certain pages and web routes are visible to the public, whilst others are restricted to administrators and logged-in users.

The user behaviors are demonstrated in Table 1 below, illustrating some of the essential features of an E-commerce web application.

Table 1. The different user behaviors.

Customers	**Administrators**
Create a new account	Not valid
Sort the product list by price, newest added date, sold number and category	Sort the product list by price, newest added date, sold number and category
View the product list	View the product list
View product information with images	View product information with images
Add products to the shopping cart	Not valid
Modify the number of products inside the shopping cart	Not valid

(continue)

Table 1. (continued).

Customers	**Administrators**
Pay the added products inside the shopping cart	Not valid
Not valid	Add new products to the database
Not valid	Remove products from the database
Not valid	Update products to the database
Not valid	Add category to the database
View own orders	View all user orders

After identifying the behaviors of different application users, it is necessary to specify their dependencies so as to close the gaps in the initial deliverables breakdown. Therefore, the flowchart of E-commerce application dependencies constructed based on Table 1 is shown in Figure 8 below:

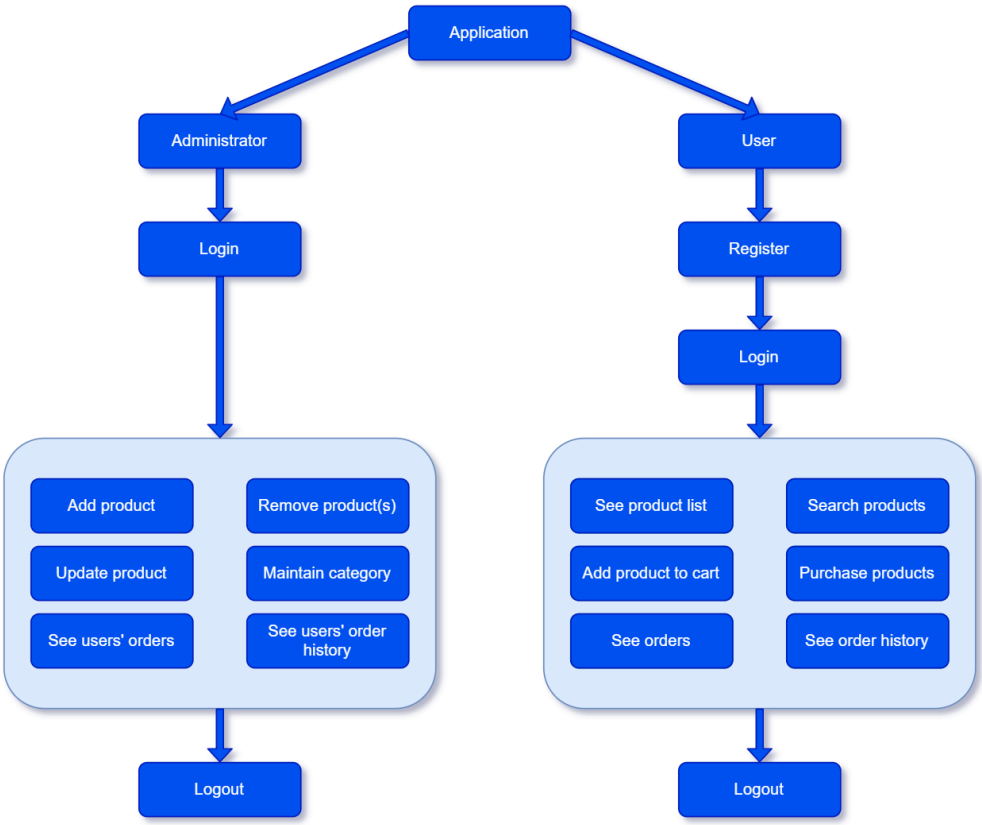


Figure 8. The dependencies flowchart.

3.2 Environmental setup

Prior to the MERN stack implementation, it is required to configure the development environment to ensure that technologies and tools operate effectively together as well as assist in the optimization of the development flow.

3.2.1 Visual Studio Code

The code editor is essential for developers to begin constructing an application. Therefore, Visual Studio Code (VSCode) was selected as the text editor for developing this thesis application. VSCode is now one of the best code editors available as it is free and open-source and integrates built-in support for various programming languages, including JavaScript, NodeJS, Java, Python, PHP,

C/C++, C#, and Go. Thanks to the enormous support community, developers can have multiple options to customize the theme, write code, install extensions to provide more capabilities, and manage their projects with version control systems such as Git or GitHub. VSCode is developed and managed by Microsoft, and it is compatible with Windows, Linux, and macOS. (Microsoft, 2022).

3.2.2 Backend initialization

The first thing to start the server-side is creating the package.json file to maintain required dependencies and devDependencies and perform different tasks by running commands. Before generating the command in Figure 9 below, NodeJS version 16.13.0 was installed.

Figure 9. NPM command for server initialization.

By running the command in Figure 9 above from the server directory, the package.json would automatically be generated the default settings without going through an interactive process.

The index.js and app.js files are then added to the root of the server application folder, including the installation of “dotenv” and ”express” packages as dependencies. The logger.js and config.js files are also comprised inside the utils folder.

The configuration of environment variables is stored in the utils/config.js file, as Figure 10 shows below:

```
1 require('dotenv').config()
2
3 let PORT = process.env.PORT || 3003
4 let MONGODB_URI = process.env.MONGODB_URL
5
6 module.exports = { MONGODB_URI, PORT }
```

Figure 10. Content of config.js file.

On line 1, the package ‘dotenv’ is imported to load the environment variables stored in the .env file and then concerning not getting involved in git history by including it in the “.gitignore” file. Therefore, the environment variables can be separated from the code base and excluded while running the production application.

The app.js file is created to be the actual application, which includes all the tasks as well as some of the node packages to get the server to work properly. Figure 11 below depicts the content of the initial app.js file created at the start of the backend configuration.

```
1  const config = require('./utils/config')
2  const express = require('express')
3  const logger = require('./utils/logger')
4  const mongoose = require('mongoose')
5  const cors = require('cors')
6  const cookieParser = require('cookie-parser')
7
8  const app = express()
9  app.use(express.json())
10 app.use(cookieParser())
11 app.use(cors())
12
13 // db connection
14 mongoose
15   .connect(config.MONGODB_URI, {
16     useCreateIndex: true,
17     useFindAndModify: false,
18     useNewUrlParser: true,
19     useUnifiedTopology: true,
20   })
21   .then(() => {
22     logger.info('Connected to MongoDB')
23   })
24   .catch((error) => {
25     logger.error('error connection to MongoDB:', error.message)
26   })
27
28 app.get('/', (req, res) => {
29   return res.json({ message: 'Running...' })
30 })
31
32 module.exports = app
```

Figure 11. Content of app.js file.

As depicted in Figure 11, some of the node packages and middlewares included in app.js can be demonstrated as below:

- mongoose stimulates the connection between the MongoDB database to the server, as depicted from line 14 to line 12.

- cookie-parser, on line 6, populates all the cookies in the request (or req) property with an object keyed by cookie names after decoding the Cookie header.

In regard to establishing the communication between the MongoDB database and the server, the MONGO_URI is indispensable. After following the essential process on the MongoDB Atlas website, a cluster configured with the cloud provider and location setting is generated. The connection string is then provided and added to the .env file as MONGO_URL. Finally, the MONGO_URL is specified in the config.js file as MONGO_URI, which is illustrated in Figure 11 above. On line 22, the terminal logs the 'Connect to MongoDB' message when the connection is successful. Alternatively, the message demonstrating connection error is compiled, as indicated on line 24 in Figure 11 above.

The index.js file merely imports the actual application from the app.js file and later launches the application, as depicted in Figure 12 below.

```
1  const app = require('./app')
2  const http = require('http')
3  const config = require('./utils/config')
4  const logger = require('./utils/logger')
5
6  const server = http.createServer(app)
7
8  server.listen(config.PORT, () => {
9    logger.info('Server is running on port', config.PORT)
10 })
```

Figure 12. Content of index.js file.

On line 4, the logger is separated into its own so that all the log messages can be managed in one place for writing logs to a file or sending them to external log management services such as Graylog and Papertrail (FullStackOpen,

2022). The logger module's function info is utilized for the console printout, indicating that the application is currently executing on port 3003 from the config.PORT of config-module. The response 'Running…' from the application came from the root URL loaded from app.js.

After successfully simulating a functional server, the complete version of the server for the application is taken into consideration. As a result, all of the files are created and organized into distinct folders for several purposes, as indicated in Figure 13 below.

Figure 13. The updated folder structure of the server.

As Figure 13 depicts, the server folder contains six sub-folders, whereas the main ones are controllers, middleware, models, and routes. The models folder consists of all required mongoose schemas for the application. Meanwhile, all

the logical event handlers are stored in the “controllers” folder to execute incoming requests that match the corresponding API routes defined in the routes folder.

3.3.3 Frontend initialization

Setting up the React application

Compared to the server-side handling of logical functionalities and databases, the client-side generates the UI interface provided for the users to communicate with the server directly. By running the command ”npx create-react-app” in the root folder, the React application template is generated. Instead of setting up the Babel packages manually to interpret JSX into JavaScript and Webpack configuration to bundle all the modules in the application, developers can save time thanks to the development of the ”create-react-app” Node module with pre-installed packages mentioned above. Figure 14 below is the folder structure of the frontend application:

Figure 14. The folder structure of the frontend application.

After setting the client, the connection between the server and client needs to be implemented. By adding the proxy field as the following declaration in package.json of the client folder, as shown in Figure 15 below, the server can recognize the fetching on the client-side and then proxy the request to the corresponding API request of the server in the development mode.

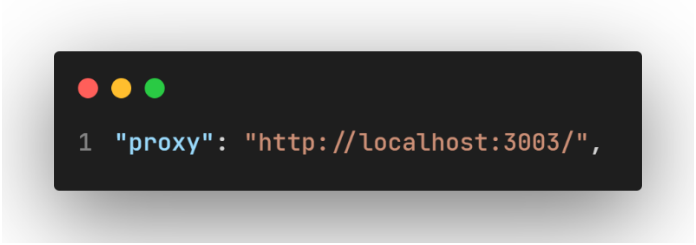


Figure 15. Proxy declaration in the package.json of the client folder.

Besides the proxy, Cross-Origin Resource Sharing (CORS) needs to be configured to relax the same-origin policy. The same-origin policy restricts interaction between documents, scripts, and media files from different origins. Therefore, it can isolate the malicious websites from unknown access from loading resources, such as user information, and relaying them to the attacker (MDN Web Docs. 2022a).

Since the client-side and server-side of the application are hosted on different domains, the same-policy error will occur when fetching APIs from the client to the server.

In order to mitigate the risks of cross-origin HTTP requests, the cors package is installed. Then, by running the command ”npm install cors” in the server directory and importing the module in the app.js file, as illustrated in Figure 16 below, the middleware is provided to enable CORS and customizable succinct settings regarding the MERN stack application particularly.

Figure 16. Usage of CORS npm packages in the server directory.

TailwindCSS integration

For this thesis application, TailwindCSS was chosen to design the web application's layout. According to TailwindCSS's official website, using the PostCSS plugin while installing Tailwind CSS is selected as the most frictionless method to integrate with Create React App. Since the application uses TailwindCSS combined with SASS preprocessor to add custom style

components in the future, the process is quite different from regular CSS integration. Instead of adding”@tailwind” directives for each of Tailwind’s layers to CSS, it will be included in the global.scss file in the styles folder created in the client directory, which indicates in Figure 17 below.

Figure 17. The global.scss file.

4 Project implementation

This chapter explains the implementation process of the web application. There are two stages of the process: backend and frontend. The backend implementation will demonstrate the Mongoose models, authentication and authorization, routes and APIs construction, and the testing API process. On the other hand, the frontend implementation will illustrate the process of handling routes and implementing different pages to accomplish the desired user behaviors specified in the project requirements in chapter 3.1.

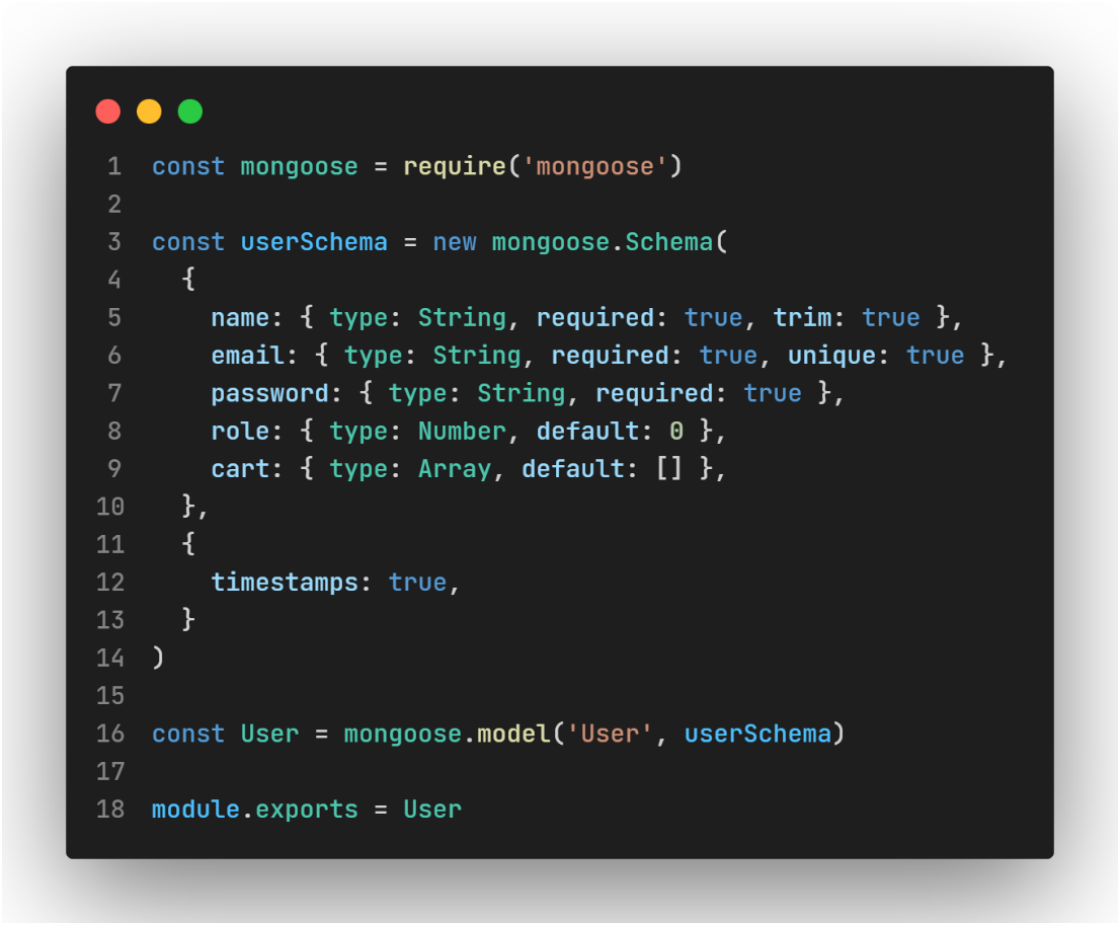
4.1 Backend implementation

4.2.1 Mongoose models

A Mongoose model is constructed by wrapping on the Mongoose schema interface representing the document frame inside the MongoDB collection. The server consists of four models: user, product, category, and payment.

User model

The structure of the user schema in detail is illustrated in Figure 18 below.



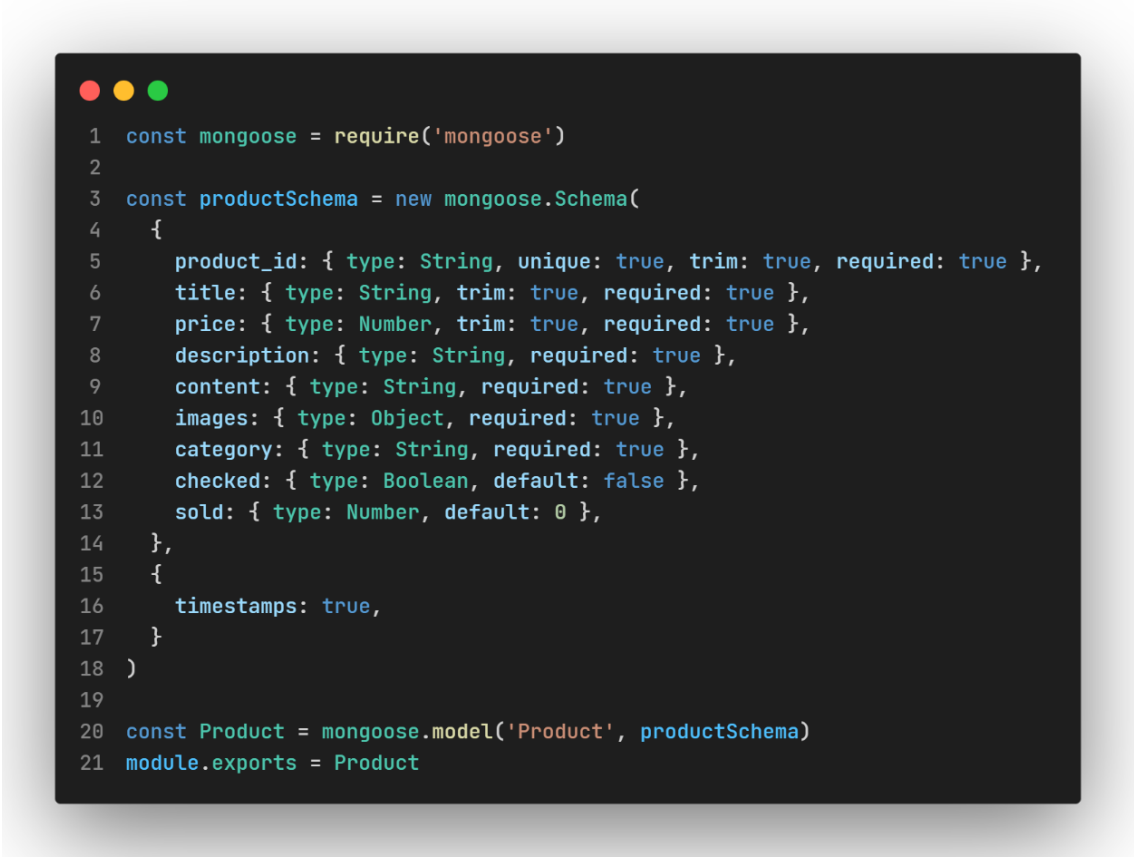
```
const mongoose = require('mongoose')

const userSchema = new mongoose.Schema(
  {
    name: { type: String, required: true, trim: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    role: { type: Number, default: 0 },
    cart: { type: Array, default: [] },
  },
  {
    timestamps: true,
  }
)

const User = mongoose.model('User', userSchema)

module.exports = User
```

Figure 18. User model.

As depicted in Figure 18, there are three required fields: name, email, and password, while the email is also set to unique because no two emails simultaneously exist. The default value for the role field is 0, indicating the typical user. By altering the value to 1 manually, the user becomes an administrator. The cart field displays an array of added product items, whilst the property ”timestamps” is enabled by setting to true to monitor the time when updating the data by assigning createAt and updateAt properties to the schema.

Product model

Figure 19 below illustrates the specifics of the product schema.



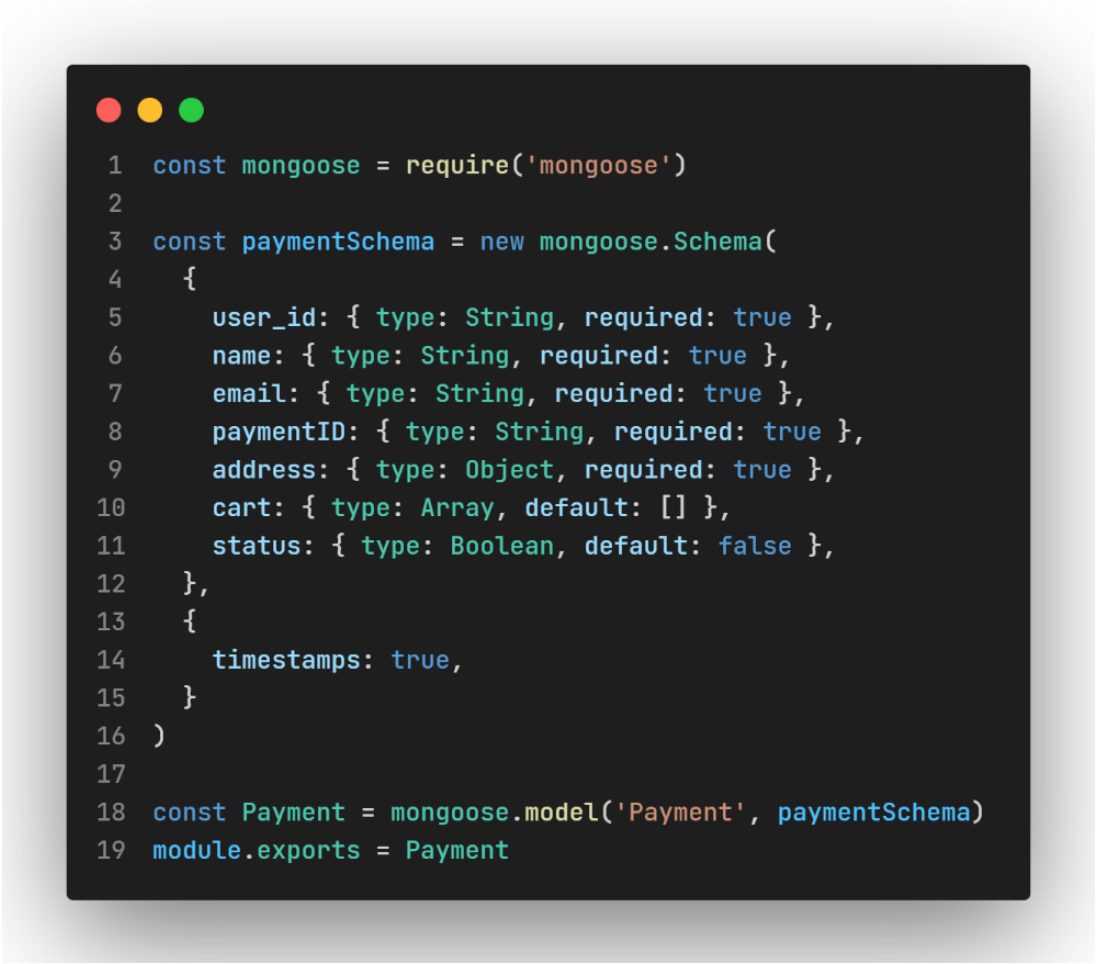
```
1  const mongoose = require('mongoose')
2
3  const productSchema = new mongoose.Schema(
4    {
5      product_id: { type: String, unique: true, trim: true, required: true },
6      title: { type: String, trim: true, required: true },
7      price: { type: Number, trim: true, required: true },
8      description: { type: String, required: true },
9      content: { type: String, required: true },
10     images: { type: Object, required: true },
11     category: { type: String, required: true },
12     checked: { type: Boolean, default: false },
13     sold: { type: Number, default: 0 },
14   },
15   {
16     timestamps: true,
17   }
18 )
19
20 const Product = mongoose.model('Product', productSchema)
21 module.exports = Product
```

Figure 19. Product model.

As exhibited in Figure 19, the product model contains all of the fields, such as title, price, description, content, images and category, that most e-commerce products require. In addition, the product_id field allows the administrator to customize the product's id without affecting the generated _id in MongoDB. The checked field, on line 12, is set for deleting multiple products function. The sold field has 0 as the default value, and it is increased after each successful purchase from a user.

Payment model

Figure 20 below indicates the detail of fundamental information of payment related to the user.



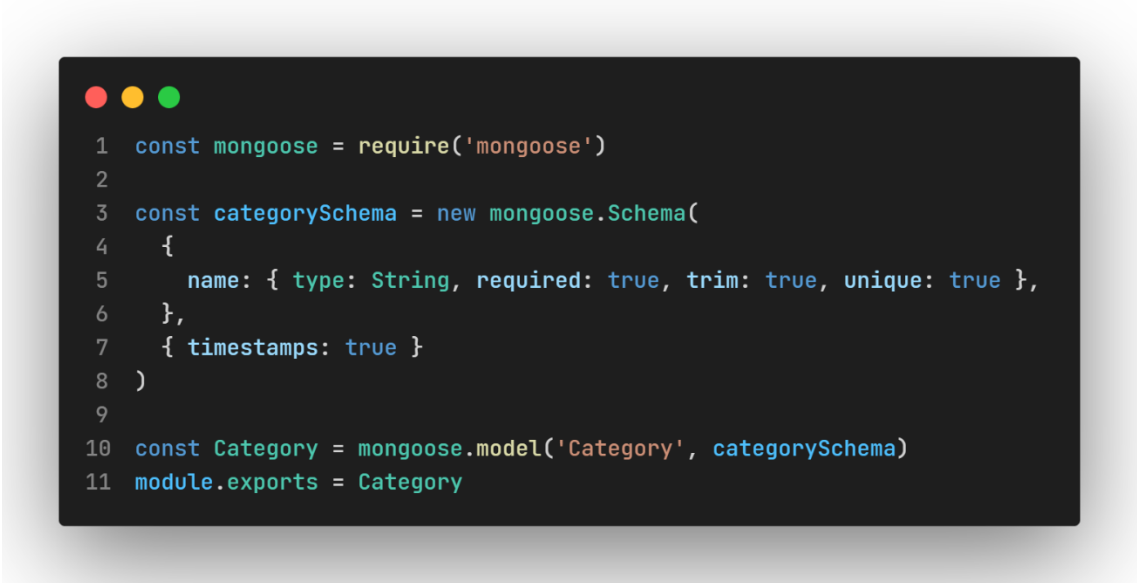
```
1  const mongoose = require('mongoose')
2
3  const paymentSchema = new mongoose.Schema(
4    {
5      user_id: { type: String, required: true },
6      name: { type: String, required: true },
7      email: { type: String, required: true },
8      paymentID: { type: String, required: true },
9      address: { type: Object, required: true },
10     cart: { type: Array, default: [] },
11     status: { type: Boolean, default: false },
12   },
13   {
14     timestamps: true,
15   }
16 )
17
18 const Payment = mongoose.model('Payment', paymentSchema)
19 module.exports = Payment
```

Figure 20. Payment model.

As seen in Figure 20, the payment model has the fields of user_id, name, email, paymentID, address and cart. The payment schema also has the status field, on line 11, indicating whether the cart is paid.

Category model

While all the aforementioned models seem complicated, the category model is clear and precise, with simply name and timestamps fields, as indicated in Figure 21.



```
const mongoose = require('mongoose')

const categorySchema = new mongoose.Schema(
  {
    name: { type: String, required: true, trim: true, unique: true },
  },
  { timestamps: true }
)

const Category = mongoose.model('Category', categorySchema)
module.exports = Category
```

Figure 21. Category model.

4.2.2 Authentication and authorization

The authentication and authorization in the backend can be generated as middleware. Authentication middleware is compulsory for routes requiring a logged-in user, while admin authorization verifies some routes that only users with role admin (value 1) can access. These middlewares will activate when the user register or login into the system.

User authentication

The code implementation of middleware for user authentication is illustrated in Figure 22 below.

```
1  const jwt = require('jsonwebtoken')
2
3  const auth = (req, res, next) => {
4    try {
5      const token = req.header('Authorization')
6      if (!token) return res.status(401).json({ msg: 'Invalid Authentication' })
7
8      jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (error, user) => {
9        if (error) return res.status(401).json({ msg: 'Invalid Authentication' })
10
11       req.user = user
12       next()
13     })
14   } catch (error) {
15     return res.status(500).json({ msg: error.message })
16   }
17 }
18
19 module.exports = auth
```

Figure 22. Authentication middleware.

As seen on line 5 in Figure 22, the JWT token is extracted from the Authorization request header and assigned as token variable. If no token is present or the token is invalid, an error message along with status 401 will be thrown. If yes, the verify() from jsonwebtoken package will be loaded. The method takes the token and ACCESS_TOKEN secret key stored in the .env file, and then the callback function is called to decode the token. When the token is valid, the decoded payload user is assigned to the user request, and the control is sent to the next() middleware function, or else the error is thrown, and the request will be left hanging.

Admin authorization

The admin authorization middleware comes after authentication middleware. It will be generated when the user successfully logs in. Figure 23 below shows the code implementation of authorization middleware for the administration user.

```
const User = require('../models/user')

const adminAuth = async (req, res, next) => {
  try {
    // Get user information by id
    const user = await User.findOne({ _id: req.user.id })
    if (user.role === 0) return res.status(401).json({ msg: 'Admin resources access denied' })

    next()
  } catch (error) {
    return res.status(500).json({ error: error.message })
  }
}

module.exports = adminAuth
```

Figure 23. Admin authorization.

Figure 23 above indicates the process of authorizing the admin user in detail. Firstly, the user information is retrieved based on the id of the user model. Then, the user role is evaluated whether it is admin or not. If the test passes, the user is an admin and can access the routes.

Storing JWT tokens

A JWT token can be stored in two main methods: localStorage and cookies. LocalStorage is the most used thanks to its convenience and access to third-party APIs without setting a backend. However, it is vulnerable to XSS attacks. By running any Javascript script inside the page, an attacker can have permission to take the access token stored in localStorage. Therefore, to secure the JWT token from XSS, storing it in the httpOnly cookie is recommended. httpOnly cookie is a unique type of cookie that is only transmitted to the server in HTTP requests, so it is impossible to access via JavaScript and is not vulnerable to XSS attack as localStorage. Even though the cookie can still be vulnerable to CSRF attacks, it can simply be mitigated by setting the ”sameSite=true” flag in the cookie and an anti-CSRF token (Michelle Wirantono, 2020).

Therefore, the best option to securely store JWT is to store the refresh token in the httpOnly cookie and the secure and sameSite flags to prevent XSS and CSRF attacks. The process is included in the register and login controllers of the user model. Figure 24 below shows the logic of token implementation in the register controller:

```
const User = require('../models/user')
const bcrypt = require('bcrypt')
const jwt = require('jsonwebtoken')

const createAccessToken = (user) => {
  return jwt.sign(user, process.env.ACCESS_TOKEN, { expiresIn: '1d' })
}

const createRefreshToken = (user) => {
  return jwt.sign(user, process.env.REFRESH_TOKEN, { expiresIn: '5d' })
}

const userController = {
  register: async (req, res) => {
    try {
      // save new user to MongoDB database
      ...

      const accessToken = createAccessToken({ id: newUser.id })
      const refreshToken = createRefreshToken({ id: newUser.id })

      res.cookie('refreshToken', refreshToken, {
        httpOnly: true,
        path: '/user/refresh_token',
        secure: true,
        sameSite: 'strict',
        maxAge: 5 * 24 * 60 * 60 * 1000,
      })

      res.json({ accessToken })
    }
```

Figure 24. Usage of httpOnly cookie in register controller.

As Figure 24 depicts, on line 5 and line 9, the createAccessToken and createRefreshToken functions are constructed using the JWT sign() method to return the JWT tokens from the found user payload. After saving the created user to the MongoDB database, the aforementioned functions are generated and stored as accessToken and refreshToken. The access token is the temporary JWT token stored in the response body and is hidden from the user, while the refresh token is the long-live one saved in the database and is available in the cookie session. Subsequently, the refresh token cookie, on line 22, is set up with the following flags:

- httpOnly prevents JavaScript from reading it (MDN Web Docs. 2022c).
- secure is set to true to allow the token to be only sent over HTTPS (MDN Web Docs. 2022c).
- sameSite is set to strict so as to prevent possible CSRF attacks (MDN Web Docs. 2022c). However, it can only be utilized when Authorization Server has the same site as the client.
- path is defined as the specific path that is included in the cookie (MDN Web Docs. 2022c).
- maxAge sets the expiration of the cookie to the current time in milliseconds (MDN Web Docs. 2022c).

The access token is then obtained and sent to the Response object as a JSON string.

After that, the access token should be stored as a variable in the client site to remain invisible when switching tabs or reloading the website.

Finally, the /refresh_token endpoint should be retrieved after expiration to obtain the new access token. Figure 25 below indicates the logic of the token refresh function in the user controller.

```
getRefreshToken: (req, res) => {
    try {
      const rfToken = req.cookies.refreshToken
      if (!rfToken) {
        return res.status(400).json({ msg: 'Please Login or Register' })
      }

      jwt.verify(rfToken, process.env.REFRESH_TOKEN, (error, user) => {
        if (error) {
          return res.status(400).json({ msg: 'Please Login or Register' })
        }
        const accessToken = createAccessToken({ id: user.id })

        res.json({ accessToken })
      })
    } catch (error) {
      return res.status(500).json({ msg: error.message })
    }
  },
```

Figure 25. Get refresh token logic controller for user model.

As Figure 25 depicts above, the refresh token is obtained from req.cookie when logging or registering successfully and saved as rfToken. After verifying the refresh token with verify() method from jwt and REFRESH_TOKEN secret key, a unique access token is generated and sent to the Response object as JSON.

4.2.3 Routes and APIs implementation

Distribution and exchange of data across two or more systems have long been vital elements of software development, so REST API is considered. API, abbreviated for Application Programming Interface, is a collection of rules that enable the interaction of two applications or systems, whereas REST (Representational State Transfer) dictates the appearance of the API. REST is a set of architectural constraints that enables two computer systems to interact through HTTP, similar to the communication between web browsers and servers (Gupta, 2022). Therefore, it is a set of guidelines that developers must

adhere to while developing APIs. Figure 26 below illustrates all of the product-related API routes with their associated method and actions as well as the usage of router-level middlewares.

```
const router = require('express').Router()
const productController = require('../controllers/products')
const auth = require('../middleware/auth')
const adminAuth = require('../middleware/adminAuth')

router
  .route('/products')
  .get(productController.getProducts)
  .post(auth, adminAuth, productController.createProduct)

router
  .route('/products/:id')
  .delete(auth, adminAuth, productController.deleteProduct)
  .put(auth, adminAuth, productController.updateProduct)

module.exports = router
```

Figure 26. The usage of REST API in product route.

As depicted, only administrators, including “auth” and “adminAuth” middlewares, can create, delete, and update the products, corresponding to GET, DELETE, and PUT requests. Meanwhile, other public routes are accessible from any location and by any user.

4.2.4 API Testing with REST Client

During the process of server development, testing is also taken into account to assure the logic of handling the requests works appropriately. Therefore, a third-party tool or extension is preferable to writing test code in the server folder. Therefore, Postman is a popular choice for this situation. However, it has plenty of drawbacks. Firstly, it is a separate application, and thus it needs to be downloaded and installed. Therefore, it requires switching context since it separates from IDE used for development, which can cause RAM consumption and slow down the computer. Secondly, the version control is not well-supported in the free tier, and it depends strongly on the Postman GUI, which is different from Git for the version control on the project. (Baron, 2021)

REST Client – a VSCode extension for sending HTTP requests and viewing the response directly in VSCode – is considered to solve these problems. By creating a directory named requests at the root of the server application after installing the plugin, all the REST client requests can be stored in the directory as files with the ”.rest” suffix. Figures 27 and 28 indicate the product deletion testing using REST Client.

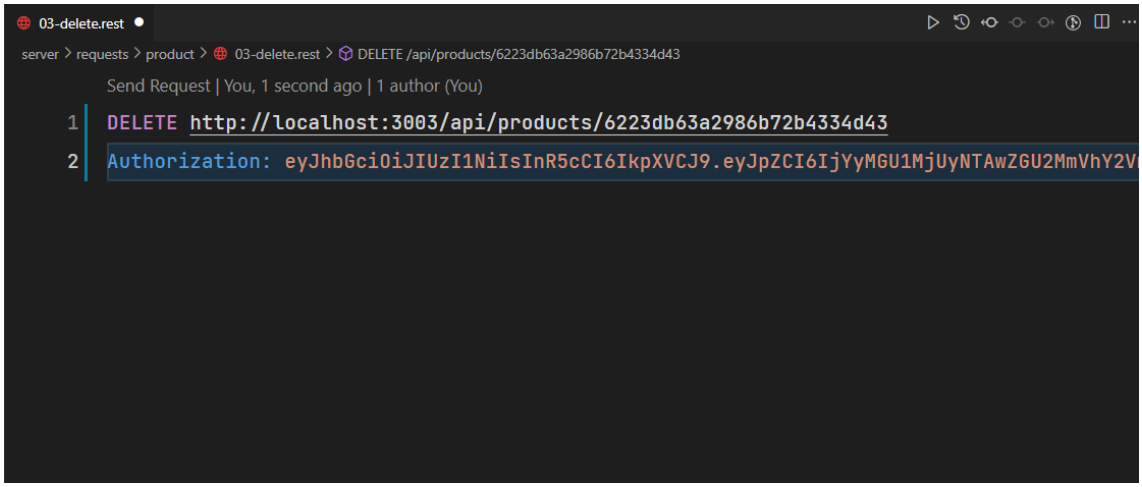


Figure 27. Screenshot of product deletion testing with REST Client.

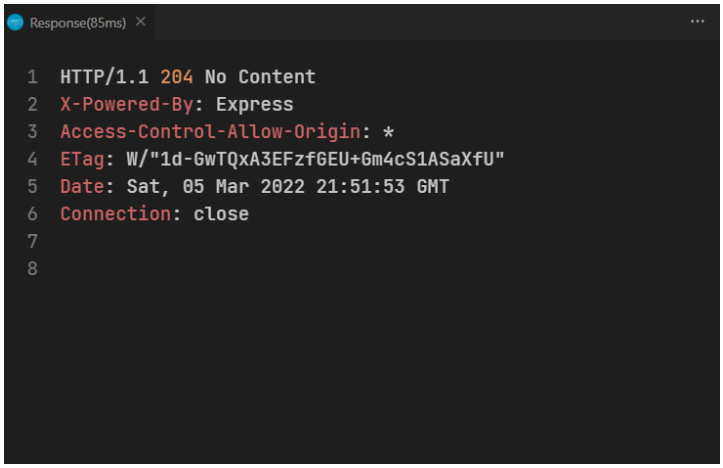


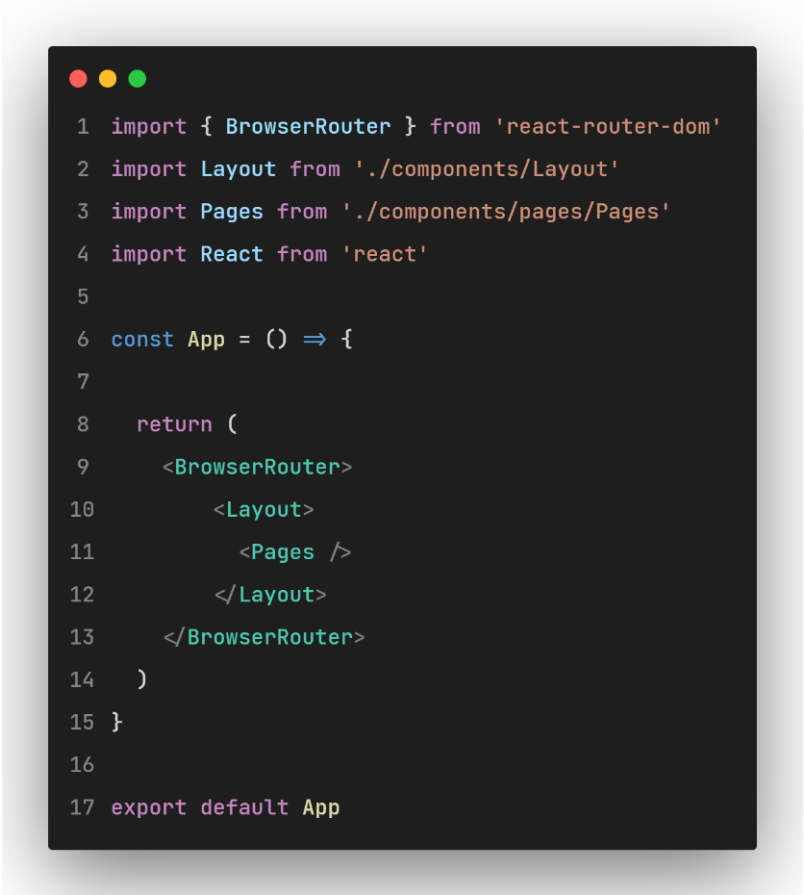
Figure 28. The result of product deletion testing with REST Client.

In Figure 27, the HTTP DELETE request fetches the product with the relevant id defined on the left window. The Authorization header is also included with the token verifying the user based on role. After clicking the ”Send request” text, the HTTP request is executed, and the server response in detail is opened as a new tab, which indicates in Figure 28. The result throws the status ”204 No Content”, indicating that the product is no longer in the database and the deletion is successful.

4.2 Frontend implementation

4.2.1 Routing

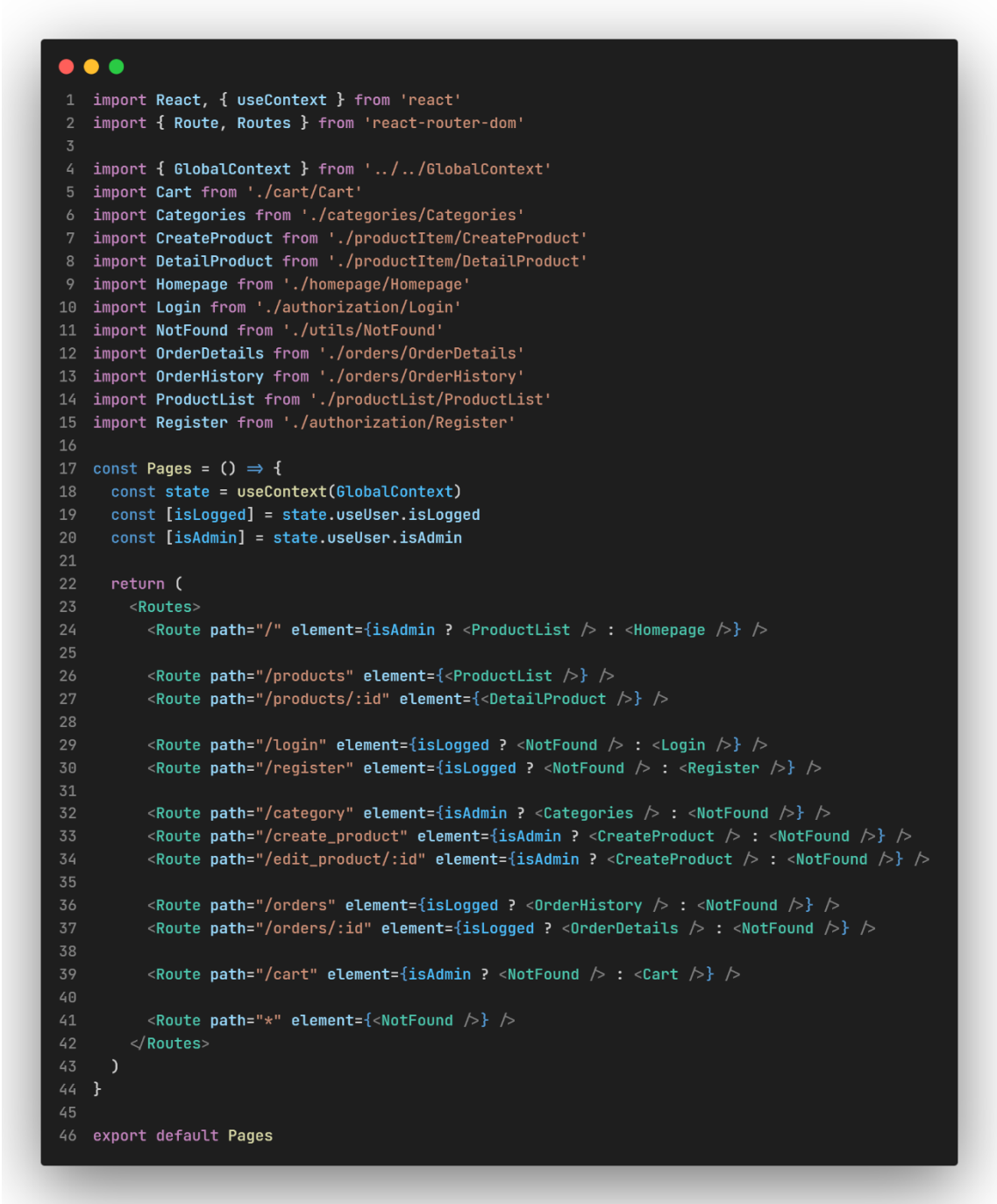
It is critical to enable navigation between pages without reloading the page in the React application. Therefore, React Router is utilized owing to its reputation for efficiently managing routing and navigation throughout the frontend. The Pages.jxs file is generated using the React Router package to comprise all the route paths and their corresponding components. The file is then imported to App.jxs file as a component and is wrapped inside the <BrowserRouter> component, which indicates in Figure 29 below.



```
import { BrowserRouter } from 'react-router-dom'
import Layout from './components/Layout'
import Pages from './components/pages/Pages'
import React from 'react'

const App = () => {

  return (
    <BrowserRouter>
        <Layout>
          <Pages />
        </Layout>
    </BrowserRouter>
  )
}

export default App
```

Figure 29. The App.jxs file.

The <Routes> and <Route> are the primary methods to render in React Router, depending on the current location. <Routes> searches through its children <Route> components for the best match and displays that branch of the UI when the location changes. Figure 30 below indicates all of the route paths that render based on the different types of users:



```jsx
import React, { useContext } from 'react'
import { Route, Routes } from 'react-router-dom'

import { GlobalContext } from '../../GlobalContext'
import Cart from './cart/Cart'
import Categories from './categories/Categories'
import CreateProduct from './productItem/CreateProduct'
import DetailProduct from './productItem/DetailProduct'
import Homepage from './homepage/Homepage'
import Login from './authorization/Login'
import NotFound from './utils/NotFound'
import OrderDetails from './orders/OrderDetails'
import OrderHistory from './orders/OrderHistory'
import ProductList from './productList/ProductList'
import Register from './authorization/Register'

const Pages = () => {
  const state = useContext(GlobalContext)
  const [isLogged] = state.useUser.isLogged
  const [isAdmin] = state.useUser.isAdmin

  return (
    <Routes>
      <Route path="/" element={isAdmin ? <ProductList /> : <Homepage />} />

      <Route path="/products" element={<ProductList />} />
      <Route path="/products/:id" element={<DetailProduct />} />

      <Route path="/login" element={isLogged ? <NotFound /> : <Login />} />
      <Route path="/register" element={isLogged ? <NotFound /> : <Register />} />

      <Route path="/category" element={isAdmin ? <Categories /> : <NotFound />} />
      <Route path="/create_product" element={isAdmin ? <CreateProduct /> : <NotFound />} />
      <Route path="/edit_product/:id" element={isAdmin ? <CreateProduct /> : <NotFound />} />

      <Route path="/orders" element={isLogged ? <OrderHistory /> : <NotFound />} />
      <Route path="/orders/:id" element={isLogged ? <OrderDetails /> : <NotFound />} />

      <Route path="/cart" element={isAdmin ? <NotFound /> : <Cart />} />

      <Route path="*" element={<NotFound />} />
    </Routes>
  )
}

export default Pages
```

Figure 30. Screenshot of Pages.jsx file.

As depicted above, <Routes> will search for the route corresponding to the URL; if the route "/products" is found, it will cease searching and present just the ProductList component. Alternatively, if <Routes> is not intended to be utilized, as seen in Figure 30 above, it will continue to check for matching routes until the end. As a result, it will display all the imported components mentioned

from line 5 to line 15 in Figure 30, as all routes beginning with "/" will match the URL.

4.2.2 Homepage

The Homepage is set as default when a visitor or a user first navigates to the web application. The user interface of the Homepage is shown in Figure 31 below.

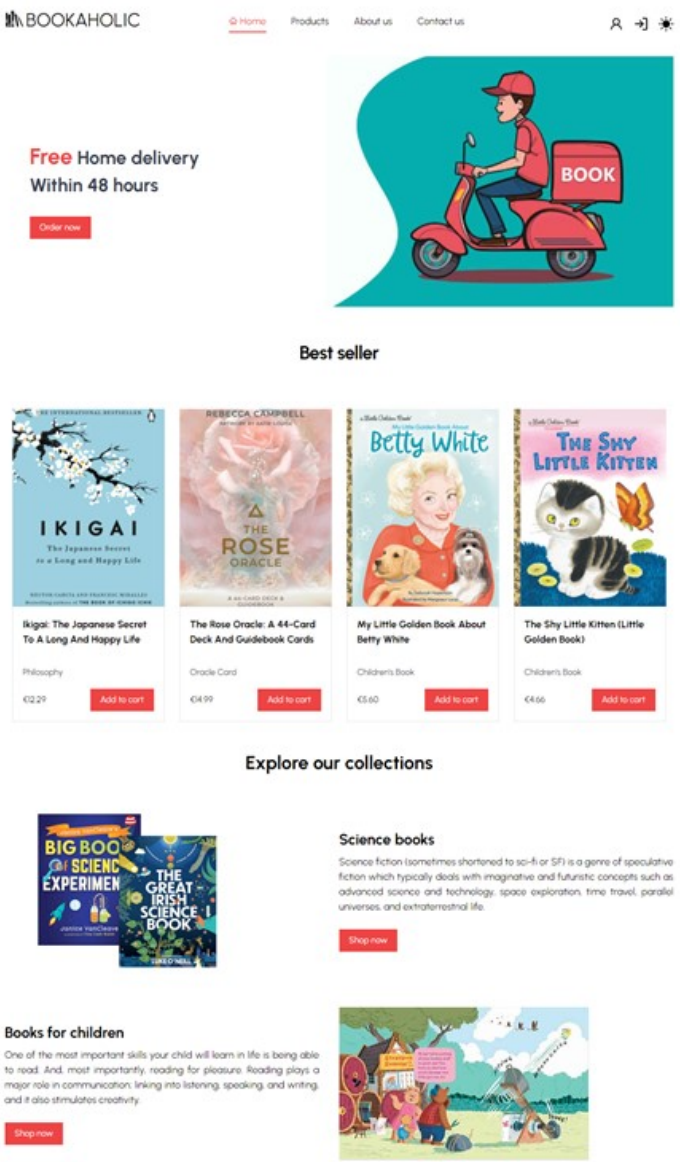


Figure 31. Homepage of the web application.

As Figure 31 depicts above, the Homepage features three components: banner, best seller product list and product collections based on category. The banner and product collections display images and text descriptions related to the components with buttons navigating to the login page, while the best seller product list shows the items with the four most sold quantities. By clicking the logo or the Home section of the navigation bar, users can revisit the Homepage instantly. Figure 32 indicates the logic of the ProductList component of the Homepage.

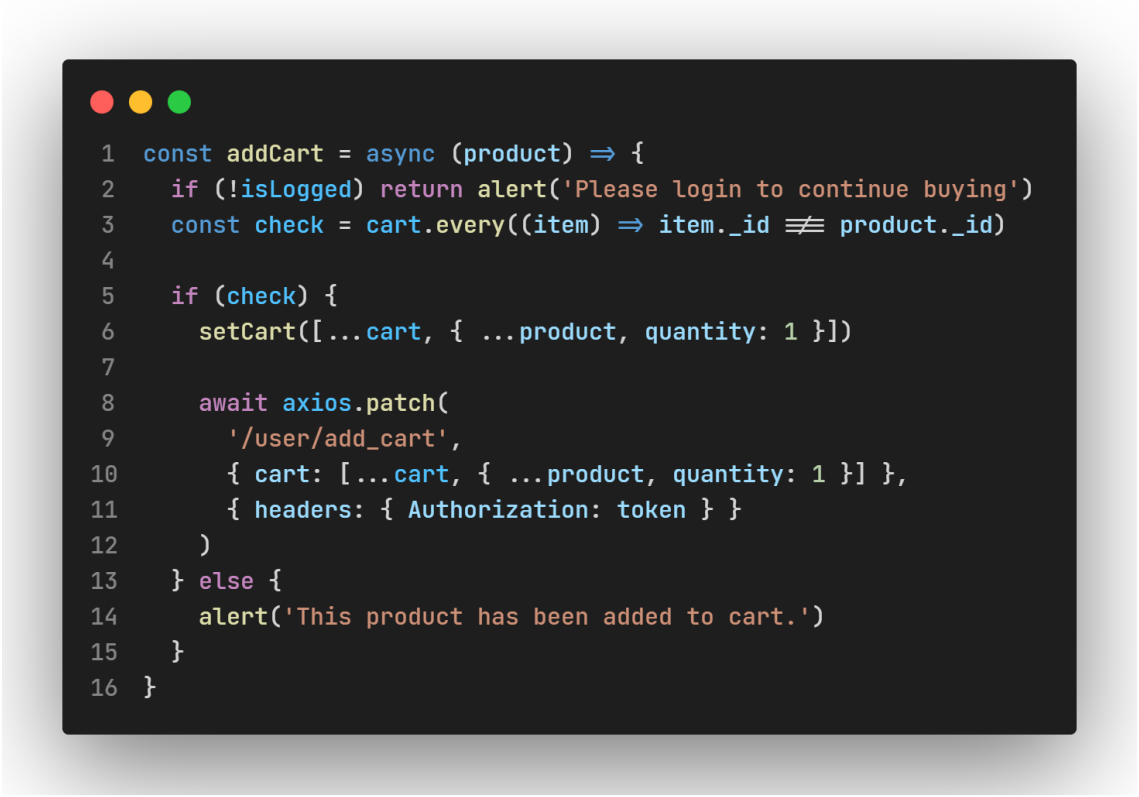
```
const ProductList = ({ title }) => {
  const state = useContext(GlobalContext)
  const [bestSellers] = state.useProducts.bestSellers

  return (
    <div className="container">
      <h2 className="text-2xl md:text-3xl text-center mt-8 lg:mt-14 pb-6 md:pb-10 tracking-tight font-bold">
        {title}
      </h2>
      <div className="grid grid-cols-4 ">
        {bestSellers.slice(0, 4).map((product) => (
          <ProductItem key={product._id} product={product} />
        ))}
      </div>
    </div>
  )
}
```

Figure 32. ProductList component from Homepage.

As Figure 32 shows above, the best-selling products will be fetched from the server to the client using the Axios package inside the useProducts hook and saved as bestSellers, and the product list is formed by using the map and slice methods from JavaScript to get the four highest selling items and then construct the ProductItem components. The ProductItem contains the title, category and “Add to cart” button with the addCart function imported from the useUser hook.

When clicking Add to cart button on a particular product, the error message will be shown if the user does not sign in. This function will check to see whether the product has already been added to the cart. If not, the product is added to

the cart and subsequently sent to the database using the PATCH method. Figure 33 below illustrates the functionality of adding the product to the shopping cart.



```
const addCart = async (product) => {
  if (!isLogged) return alert('Please login to continue buying')
  const check = cart.every((item) => item._id !== product._id)

  if (check) {
    setCart([...cart, { ...product, quantity: 1 }])

    await axios.patch(
      '/user/add_cart',
      { cart: [...cart, { ...product, quantity: 1 }] },
      { headers: { Authorization: token } }
    )
  } else {
    alert('This product has been added to cart.')
  }
}
```

Figure 33. A screenshot of addCart functionality.

4.2.3 User Authentication

The user authentication of the frontend of the application is composed of the Register and Login pages, followed by the registration, login, and logout functionalities.

Register Page

The Register page allows visitors to sign up for a user account and log in so as to purchase products on the site. Regarding the user experience enhancement, it is sufficient to access certain pages of the application as a visitor, such as

Homepage and Product page, without requiring logging into the platform. Figure 34 below illustrates the user interface of the Register page:

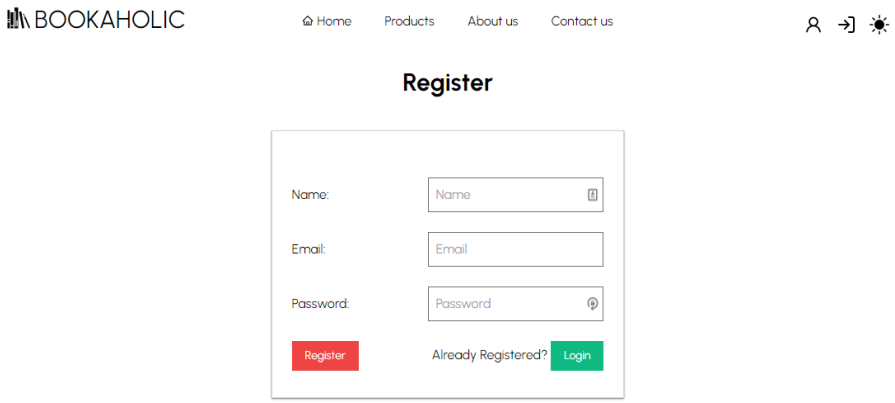


Figure 34. Register page.

The Register page contains the share navigation bar with the Homepage based on the layout. It also includes a form right below the navigation bar with input fields such as name, email, and password. After entering valid data and pressing the Register button to confirm the account creation process, the POST request will be fetched to the “user/register” route from the backend with user data under JSON format. If the input email already exists or the input password is shorter than six characters, an error asserting the following issues will be sent back from the backend and displayed on the page as an alert window. Conversely, a new model user instance is generated and saved into the database. Finally, the page will sign the user in as the registration process is successful.

Login page

The Login page is similar to the Register page, except that only email and password fields are displayed. In addition, the Login page allows users to perform tasks such as checking the order history or purchasing products. Figure 35 below is the code implementation of logic for the login functionality.

```
const [user, setUser] = useState({ name: '', email: '', password: '' })

const onChangeInput = (event) => {
  const { name, value } = event.target
  setUser({ ...user, [name]: value })
}

const registerSubmit = async (event) => {
  event.preventDefault()
  try {
    await axios.post('/user/register', { ...user })

    localStorage.setItem('userLogin', true)

    window.location.href = '/'
  } catch (err) {
    alert(err.response.data.msg)
  }
}
```

Figure 35. Login functionality.

As Figure 35 depicts, the user variable is initially declared with an empty object containing email and password by using the useState hook. When the email and password are entered, the onChangeInput function is triggered to immediately update the user variable's state. After the user info is submitted by clicking the Login button, the loginSubmit function is generated by fetching the “user/login” using the POST method and then sending the user to the server.

Following that, the accessToken and refreshToken are generated as declared in 3.2.3, along with the refreshToken being saved in the Cookie header, as

illustrated in Figure 36 below. In addition, the user login status in Local Storage is updated to true to confirm the user successfully login to the application.

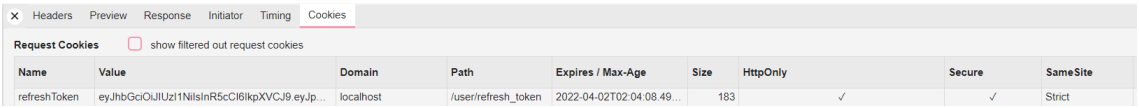


Figure 36. Screenshot of the user refresh token in Cookie header.

Logout functionality

After users log in, the navigation bar will be modified to include the logout icon. When the user selects the “Logout” icon, the logout function, indicated in Figure 37 below, is generated by fetching the ‘/user/logout’ route using the GET method. The login status is then removed from the Local Storage, and the user will be redirected to the Homepage as a successful confirmation.

```
1  const logoutUser = async () => {
2      await axios.get('/user/logout')
3      localStorage.removeItem('userLogin')
4      window.location.href = '/'
5    }
```

Figure 37. Logout functionality.

4.2.4 Products Page

The Products page is one of the essential pages of the application, which demonstrates all the products of the shop. Figure 38 below demonstrates the user interface of the Products page:

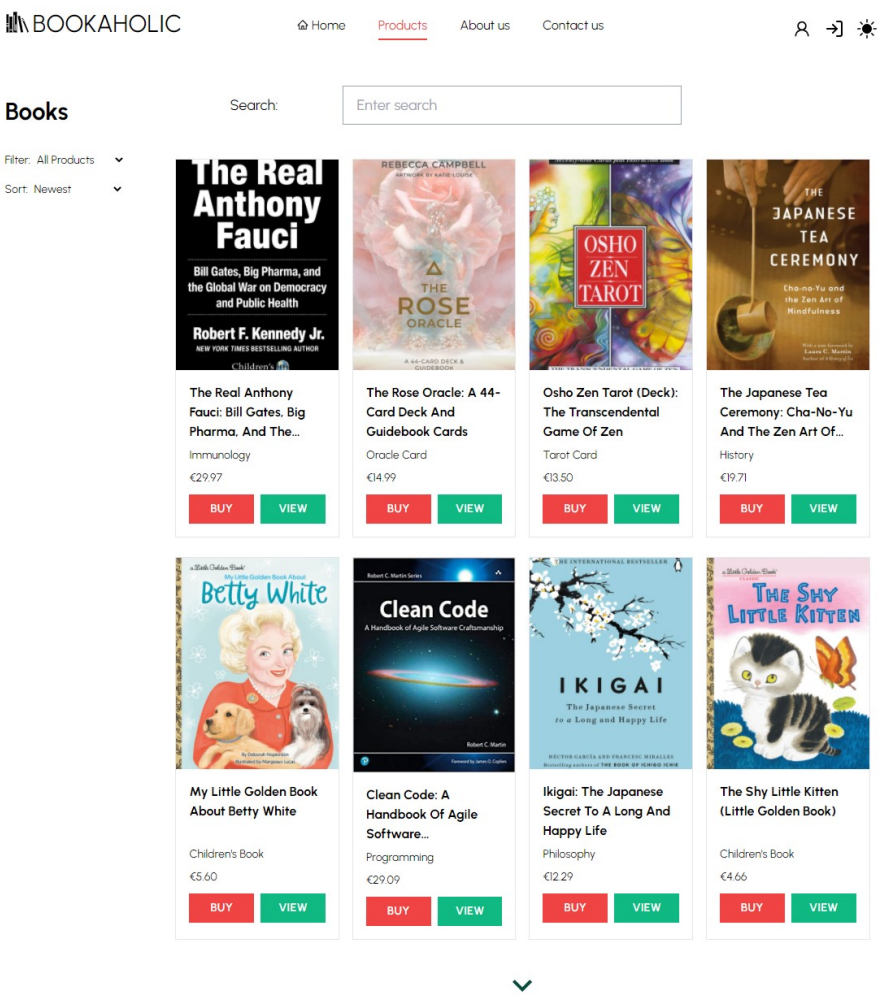


Figure 38. Products page.

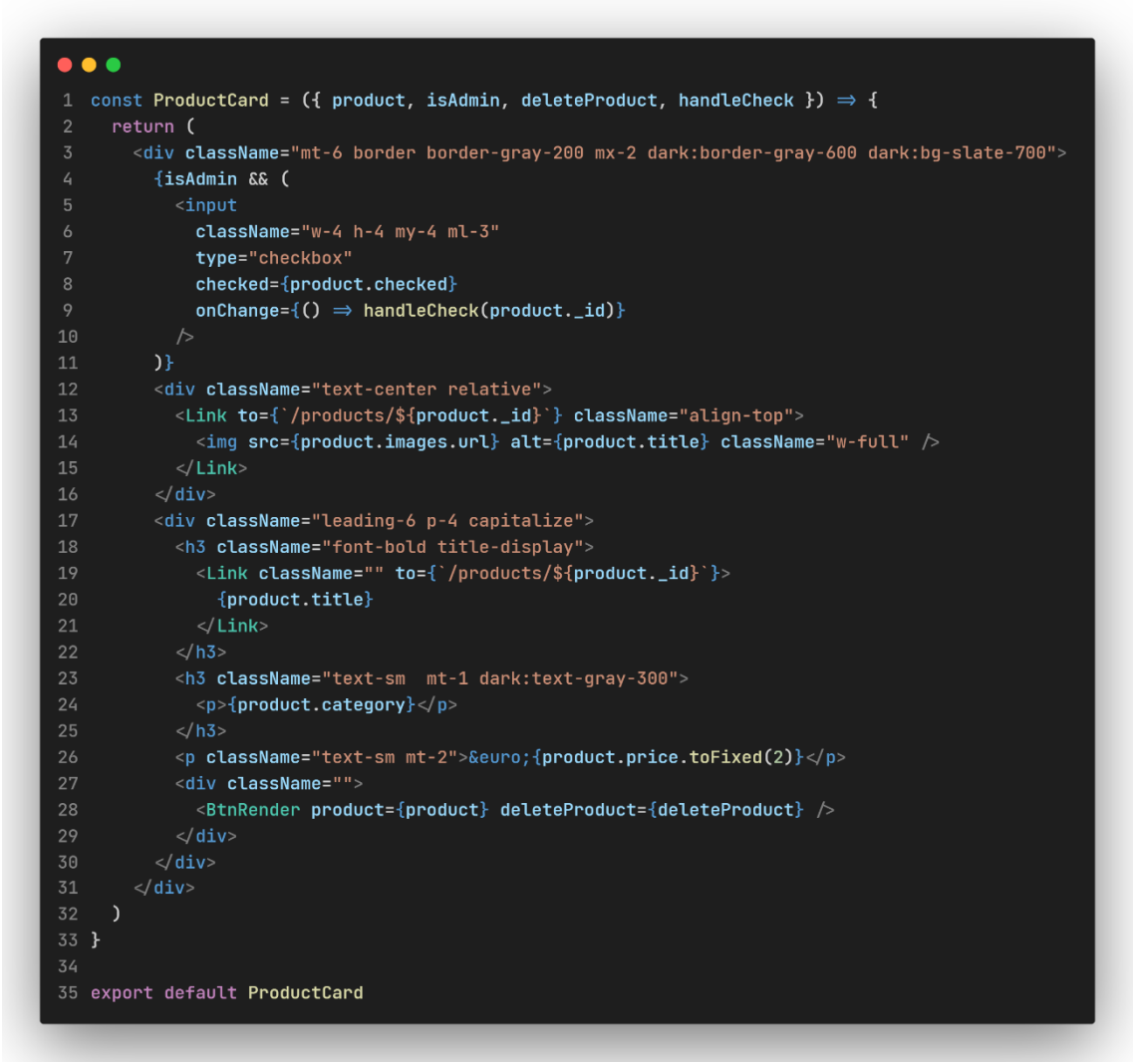
As shown in Figure 38, there are four components portraited on this page: filter, search, product list and product card. The filter component is on the left side, containing two sorting features. The ‘Filter’ sorts product list based on category, while the ‘Sort’ one is based on product price, highest sales, and time, which depends on createAt property. The Search component renders the search engine on top of the page and below the shared navigation bar. The product list component renders eight products each time, along with an arrow down button for loading more products. Figure 39 below is the data fetching for the product list using the GET method:

```
1  useEffect(() => {
2    const getProducts = async () => {
3      const res = await axios.get(
4        `/api/products?limit=${
5          page * 8
6        }&${category}&${sort}&title[regex]=${search}`
7      )
8
9      setProducts(res.data.products)
10     setResult(res.data.result)
11   }
12
13   getProducts()
14 }, [callback, category, sort, search, page])
```

Figure 39. Data fetching of product list.

As demonstated in Figure 39, the useEffect hook is invoked to perform the asynchronous method specified inside the hook to handle the data fetching side effect. The Effect hook operates just during the first render and then updates in accordance with the dependency value changes displayed on line 14. The getProducts function is then performed with three distinct sorting methods depicted on line 6 as parameters while restricting the number of display items on lines 4 and 5 in order to submit a GET request to the backend API. The server then provides a JSON object to the client, including all best sellers and new products.

The codebase of the ProductCard component is shown in Figure 40 below.



```jsx
const ProductCard = ({ product, isAdmin, deleteProduct, handleCheck }) => {
  return (
    <div className="mt-6 border border-gray-200 mx-2 dark:border-gray-600 dark:bg-slate-700">
      {isAdmin && (
        <input
          className="w-4 h-4 my-4 ml-3"
          type="checkbox"
          checked={product.checked}
          onChange={() => handleCheck(product._id)}
        />
      )}
      <div className="text-center relative">
        <Link to={`/products/${product._id}`} className="align-top">
          <img src={product.images.url} alt={product.title} className="w-full" />
        </Link>
      </div>
      <div className="leading-6 p-4 capitalize">
        <h3 className="font-bold title-display">
          <Link className="" to={`/products/${product._id}`}>
            {product.title}
          </Link>
        </h3>
        <h3 className="text-sm  mt-1 dark:text-gray-300">
          <p>{product.category}</p>
        </h3>
        <p className="text-sm mt-2">&amp;euro;{product.price.toFixed(2)}</p>
        <div className="">
          <BtnRender product={product} deleteProduct={deleteProduct} />
        </div>
      </div>
    </div>
  )
}

export default ProductCard
```

Figure 40. ProductCard component.

Similar to the ProductItem component displayed on the Homepage, the ProductCard component presents each product information such as product name, image, category, and price.

4.2.5 Product Detail Page

The Product Detail page allows users to learn more about a product in greater depth. Figure 41 denotes the codebase of the DetailProduct component.

```
1   return (
2     <div className="container ">
3       <div className="flex flex-row gap-8">
4         <div className="w-1/2">
5           <img src={detailProduct.images.url} alt={detailProduct.title} />
6         </div>
7         <div className="w-1/2">
8           <div>
9             <h1 className="text-4xl font-bold uppercase">{detailProduct.title}</h1>
10          </div>
11          <p className="mt-4 capitalize">Category: {detailProduct.category}</p>
12          <p className="mt-4 capitalize">Description:</p>
13          <p className="mt-4 whitespace-pre-line">{detailProduct.description}</p>
14          <div className="flex flex-row justify-between mt-5">
15            <p className="text-2xl">Sold: {detailProduct.sold}</p>
16            <span className="text-2xl text-right">Price: € {detailProduct.price.toFixed(2)}</span>
17          </div>
18          <p className="text-2xl flex flex-row justify-between mt-4">
19            <button className="button-1">
20              <Link to="/products">BACK</Link>
21            </button>
22            {!isAdmin ? (
23              <Link to="/cart" className="button" onClick={() => addCart(detailProduct)}>
24                BUY NOW
25              </Link>
26            ) : (
27              <Link to={`/edit_product/${detailProduct._id}`} className="button">
28                EDIT
29              </Link>
30            )}
31          </p>
32        </div>
33      </div>
34
35      <div>
36        {allBooks.some(
37          (product) => product.category === detailProduct.category && product._id !== detailProduct._id
38        ) === true ? (
39          <h2 className="text-3xl font-bold mt-8">Related products</h2>
40        ) : null}
41
42        <div className="grid grid-cols-4">
43          {allBooks.map(
44            (product) =>
45              product.category === detailProduct.category &&
46              product._id !== detailProduct._id && (
47                <div key={product._id}>
48                  <ProductCard key={product._id} product={product} />
49                </div>
50              )
51          )}
52        </div>
53      </div>
54    </div>
55  )
56  }
```

Figure 41. DetailProduct component.

As depicted in Figure 41, from line 4 to line 17, the product name, image, category, description, price, and the number of product that has been sold are rendered on the Product Detail page. Below the product information, two buttons with equivalent functionality dependent on user role are then performed.

Additionally, the products in the same category as the current item are presented below. The ProductCard component for rendering products is once again deployed. In addition, the logged-in users are able to add the product to the cart by clicking the “Buy Now” button, which will send the product to the user information.

4.2.6 Cart page

The Cart page is regarded as the final page on which users can complete their purchase. It displays the items that have been added to the cart. By clicking the Cart icon in the navigation bar, the user will be redirected to the Cart page. Below is the UI demonstration of the Cart page:

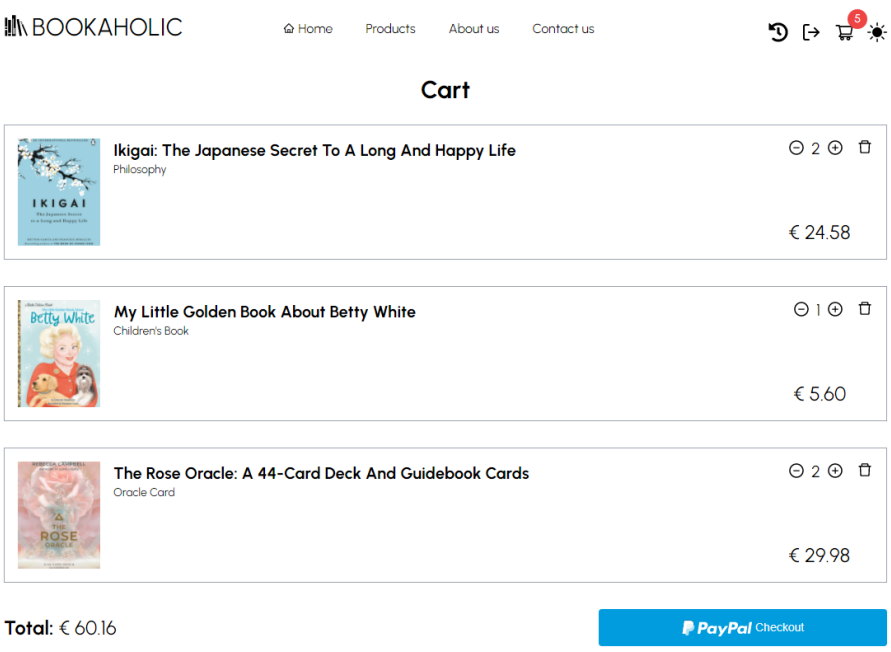


Figure 42. Cart page.

In Figure 42, the order items are displayed, and the user can update the quantity of each product or remove it from the shopping cart while examining the total price of each product depending on its quantity. Although the

application currently supports Paypal as the only payment option, other payment options can be considered in the future to accelerate the payment process. The total price and the Paypal payment button are located at the bottom of the page. The Paypal payment button is generated by utilizing the 'react-paypal-express-checkout' package.

After selecting the Paypal button, the 'react-paypal-express-checkout' package is activated, and the page will generate the payment modal. Then, the user can complete the transaction by filling out the form to login into Paypal. Following that, users can adjust the shipping address inside the modal to ensure it is correct. Moreover, Paypal offers developers the sandbox accounts for testing Paypal integration and stimulating actual transactions. Finally, if the transaction is successful, the alert window will appear with the message indicating that the transaction was completed, and the cart will be empty.

4.2.7 Administrator pages

All administrator pages are password-protected, and users must have the administrator role to access them. After signing in as an administrator, the navigation bar will be changed to correspond with the administrators' actions, with the Administrator Homepage, Create Product page, and Categories page being titled accordingly.

Administrator Homepage

The administrator homepage is essentially the Product page with a few modifications, whereas the logo is replaced with the Administration icon, the buttons are modified with corresponding functionalities, and "Select all" is added to choose multiple products to delete. Figure 43 below demonstrates the functionalities of administrators, such as selecting multiple products and removing a product from the database:

```
const handleCheck = (id) => {
  products.forEach((product) => {
    if (product._id === id) product.checked = !product.checked
  })
  setProducts([...products])
}

const deleteProduct = async (id, public_id) => {
  try {
    const config = { headers: { Authorization: token } }
    if (window.confirm('Are you sure you want to delete this product?')) {
      setLoading(true)
      const destroyImg = axios.post('/api/destroy', { public_id }, config)
      const deleteProduct = axios.delete(`/api/products/${id}`, config)
      await destroyImg
      await deleteProduct
      setCallback(!callback)
      setLoading(false)
    }
  } catch (err) {
    alert(err.response.data.msg)
  }
}

const checkAll = () => {
  products.filter((product) => (product.checked = !isCheck))
  setProducts([...products])
  setIsCheck(!isCheck)
}

const deleteAll = () => {
  if (window.confirm('Are you sure you want to delete all products?')) {
    products.map(
      (product) =>
        product.checked &&
        deleteProduct(product._id, product.images.public_id)
    )
  }
}
```

Figure 43. The administrator functionalities in the ProductList component.

Create Product Page

The Create Product page contains a form to fill out the product information and the image uploader to provide the visualization of the product. To establish a new product, an administrator must complete all of the required fields in the form and provide a proper image. Then, the new product will be generated and added to the database. Figure 44 below illustrates the user interface of the Create Product page.

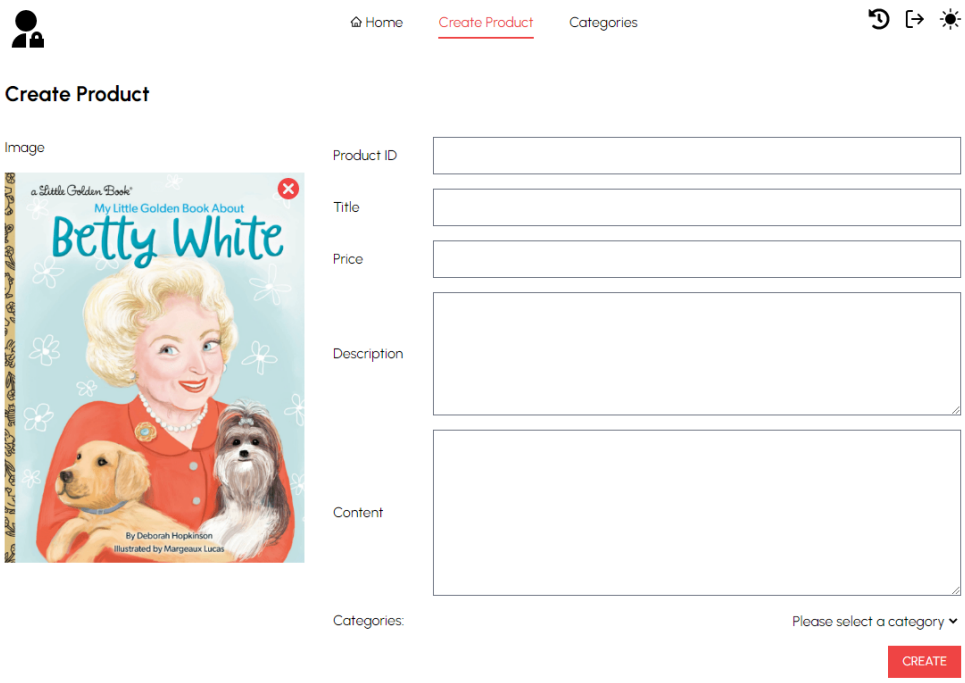


Figure 44. Create Product Page.

Categories Page

The Categories page is where administrators manage the categories of the products. Administrators have the ability to manage the category list by adding, updating, and removing methods. The category list will then be fetched to the Create Product Page, where administrators can modify the product details, and the Products Page, where it can be used for sorting the product list. Figure 45 below is the user interface of the Category page:

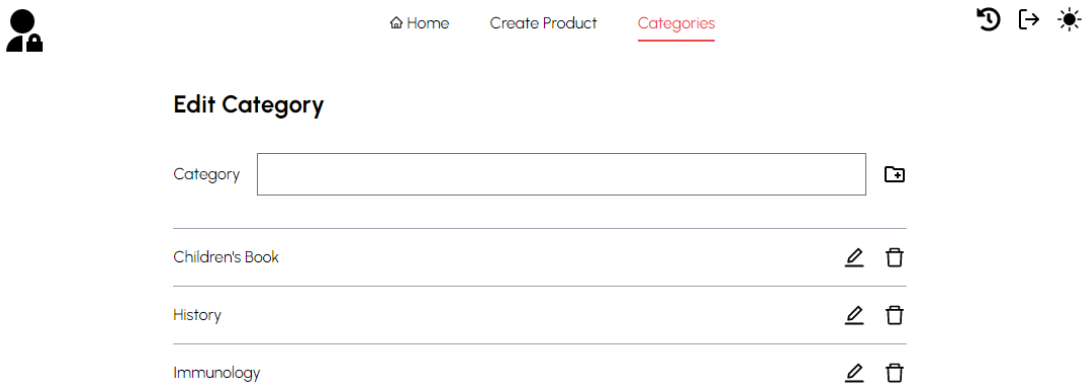


Figure 45. Categories Page.

5 Deployment

After building the application on the local machine is complete, the deployment stage should be considered to bring the application to the users as well as assure a better user experience. Therefore, Heroku cloud hosting was utilized to host this thesis application. Heroku is a supplier of cloud-based application hosting services, and it is well-known due to its straightforwardness, accessibility, and profitability. Initially, the “client/build” folder is generated by running the command “npm run build” to create the production build for the frontend, and the build directory containing static files is then copied to the root of the server repository. Additionally, the middleware declaration demonstrated in Figure 46 below is added in the “server/app.js” file in order to make the server display the files with static content.

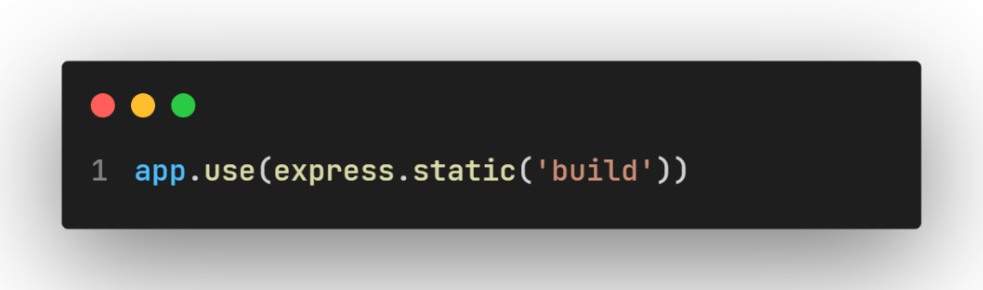
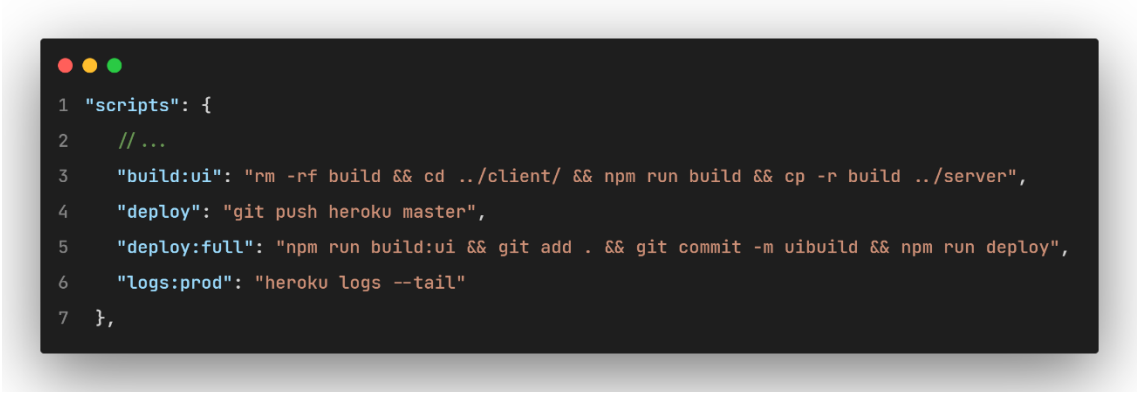


Figure 46. Express middleware for serving static files.

The application is then published to Heroku, which is a cloud-based application deployment platform. Next, Heroku CLI is installed, and the login to Heroku is then implemented through the terminal. After that, the command “heroku create” is run in the terminal to re-create a new application hosted on Heroku. Following that, the Procfile file is created to the root of the server directory with the script “web: npm start” to set up the instruction for Heroku to start the project, and the commands included in package.json illustrated in Figure 47 below are occupied to reduce the manual work for renovating a new production build to Heroku.



```
1 "scripts": {
2   // ...
3   "build:ui": "rm -rf build && cd ../client/ && npm run build && cp -r build ../server",
4   "deploy": "git push heroku master",
5   "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && npm run deploy",
6   "logs:prod": "heroku logs --tail"
7 },
```

Figure 47. npm-scripts to create new production build to Heroku.

Finally, environment variables are added to the Heroku platform, and the application is successfully built and deployed instantly. By running the command on line 5 in Figure 47 above, the application can be rebuilt and redeployed whenever an updated version is released, and the “logs:prod” command on line 6 demonstrates the Heroku logs with the real-time session.

6 Testing

Testing is essential for quality assurance since it verifies that the application is functional and compatible with the majority of contemporary browsers such as Chrome, Microsoft Edge, Safari, and Firefox. Therefore, the browser compatibility test conducted in the aforementioned browsers was successful.

Additionally, the web content is proofread for grammatical errors to provide a better user experience.

The functionality test is taken into account, including verifying all the links, observing products with images, sorting the product list ascendingly and descendingly based on added date and price, and filtering the products by category and best seller. Furthermore, the add cart functionality modifying the products in the shopping cart, completing the payment using Paypal, and checking their own order history were examined for the logged-in user. On the other hand, the product information was updated for the administrator, adding new products, modifying the category list, and all users' order reviews were analyzed. As a result, all functions operate correctly. Therefore, as per the criteria, the test was successful.

The process of purchasing products on the deployed web application is described starting from Figure 48 to Figure 54 below, with each figure illustrating each step of the process.

Bookaholic shopping process

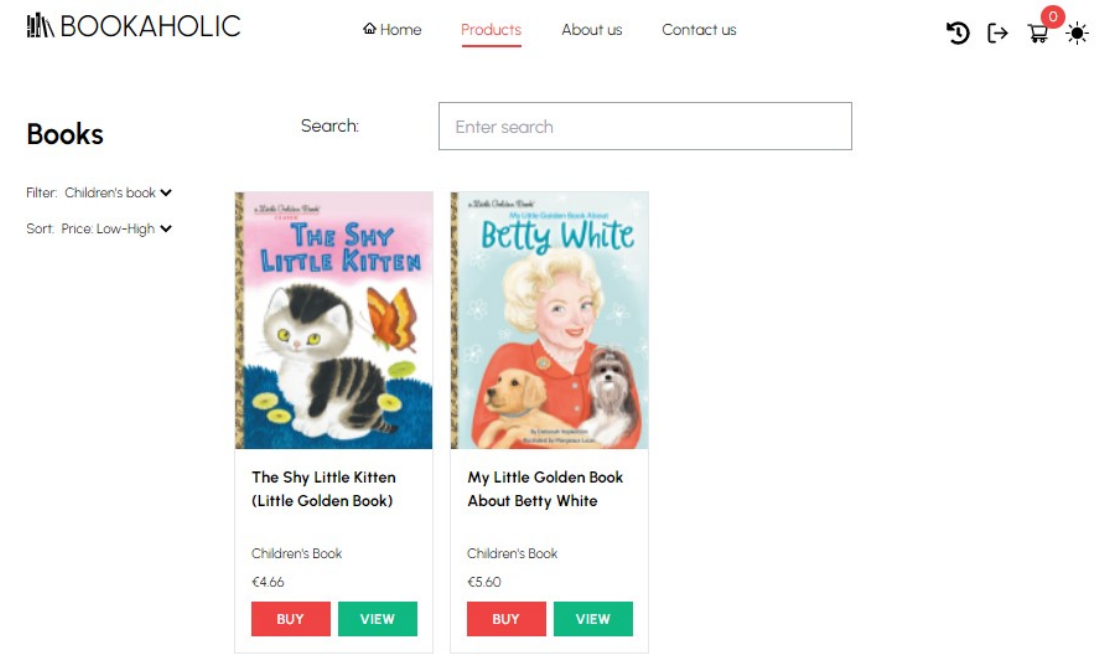


Figure 48. Products are sorted based on category and ascending price.

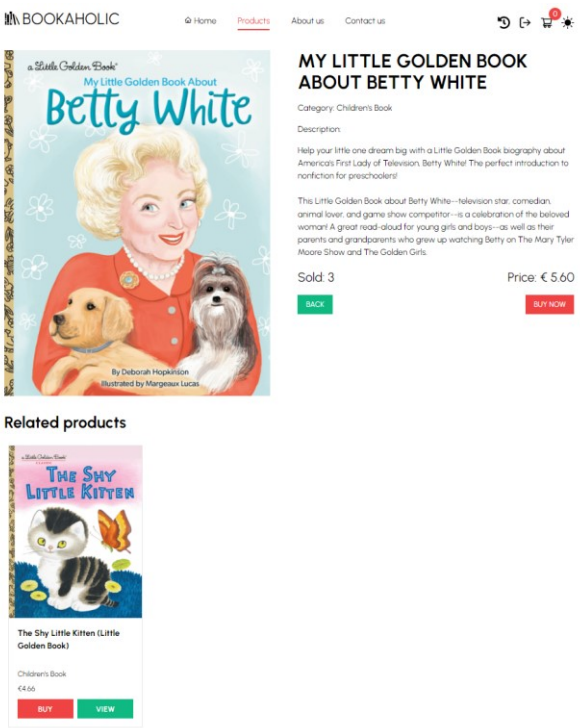


Figure 49. Product Detail Page with Related Products based on category.

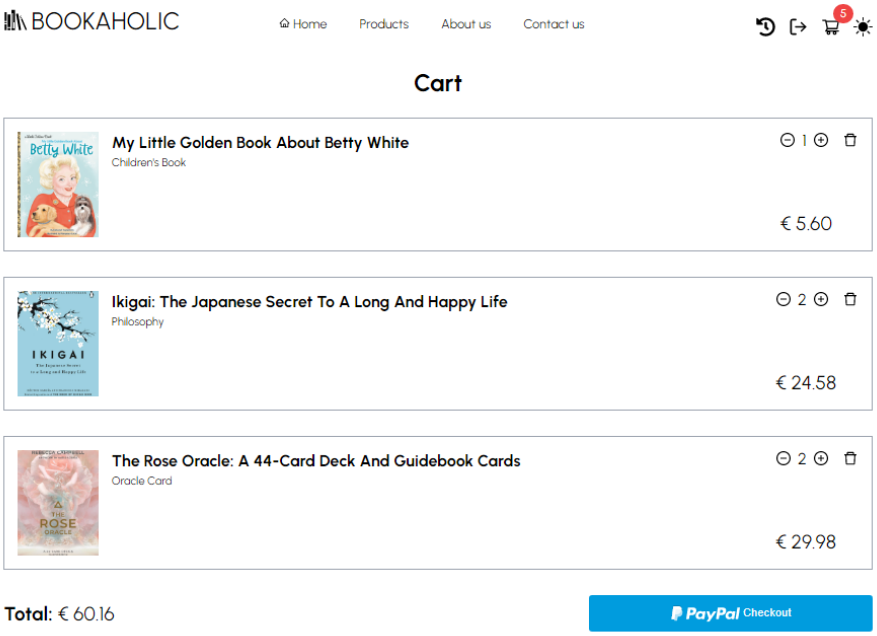


Figure 50. Add products to shopping cart success.

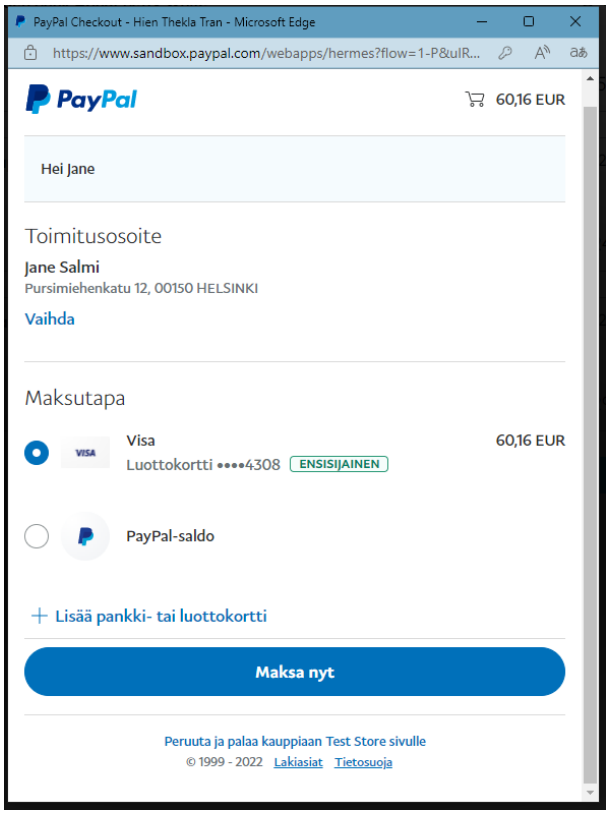


Figure 51. Redirect to the Paypal checkout window.

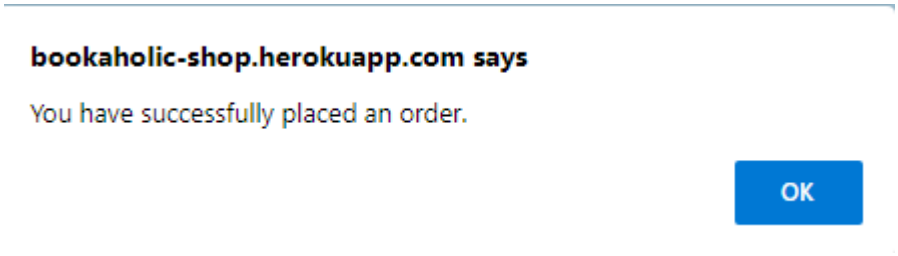


Figure 52. The alert window after the transaction is completed.

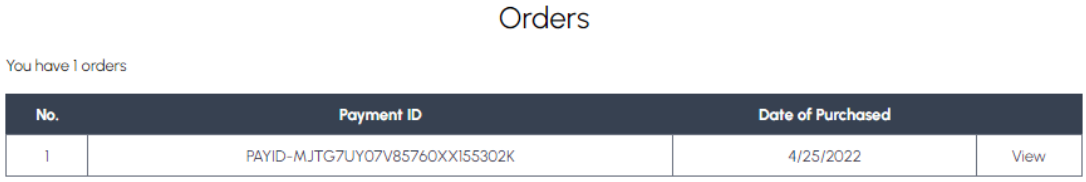


Figure 53. View order on the Order History page.

Figure 54. The order details include the customer name, delivery address, and order info.

7 Discussion

This chapter demonstrates the outcome of the E-commerce web application. In addition, recommendations regarding potential improvements are discussed for enhancing the application in the future.

7.1 Results

In the end, the E-commerce web application was successfully constructed and deployed by utilizing the MERN stack as the primary fullstack technology, demonstrating that the MERN stack has the ability to develop the complicated fullstack application. The final version of the application was built using MongoDB, Express, React, and NodeJS in combination with various tools and libraries, and the deployed web application can be accessed at https://bookaholic-shop.herokuapp.com/.

Throughout the thesis, the technologies and techniques utilized were thoroughly discussed. Consequently, the completed application achieves all the criteria specified in the project requirements. Any visitor can browse the products and select an individual item to access its details. Users can log in to the system, add products to the shopping cart, manage the cart, complete the transaction and review their purchase history. Users with the administrative role, on the other hand, can add products, update product information and check the detail of all users' orders. The application is approachable, straightforward, and practical since it only requires a few steps for any visitor to transition from being an anonymous user to completing the transaction on the platform.

7.2 Further improvements

Even though this thesis application satisfies all of the requirements as anticipated, there are still several areas regarding capabilities that can be extended. The web application currently supports only the desktop computer, so

the responsive design device for accessing various devices, including tablets and mobile phones, can be implemented in the future to provide a better user experience. Furthermore, since Paypal is the sole payment mechanism of the application at the moment, additional payment options, such as Stripe, Visa, and MasterCard, can be installed. Additionally, registration and logging functionality using third-party mailing services or social media platforms such as Gmail, Outlook, and Facebook can be constructed to accelerate the sign-in process as well as enhance the user experience. Moreover, email notifications can be considered to verify the sign-in accounts and confirm successful payment. The order status is also considered to monitor the shipping or delivery process, allowing administrators to control the process and users to track the orders. Finally, the product rating and review feature can be implemented to enable users to evaluate products and compare product ratings and reviews before making a purchase.

On the other hand, other sophisticated methodologies and concepts can be utilized to enhance various aspects of the application. Server-side rendering (SSR) can be added to boost search engines' access to the web’s content, which optimizes Search Engine Optimization (SEO) to improve the web page's visibility in search engines like Google or Bing. Automated testing can be implemented using different popular tools, such as react-testing-library, Jest, and Cypress, to execute different testing methods, such as unit testing, integration testing, and end-to-end (E2E) testing, which diminishes the errors and bugs of the application.

8 Conclusion

The aim of the thesis was to study and understand the fundamental concepts of each technology in the MERN stack and, as a result, build an E-commerce web application for the online shop based on it. The thesis conducted an in-depth examination of the MERN stack fundamental principles as well as libraries and tools that supplement the MERN stack utilized to construct the application.

Finally, a functional and production-ready online store application was constructed and successfully deployed. With a few easy steps, any visitor can register, add products to their shopping cart, complete the transaction using Paypal and review their order history. In addition, users with administrator privileges can manage product information, add new products and check the detail of all users' orders. Generally, the application satisfied all the initial requirements from the beginning.

The thesis can be utilized to reference the MERN stack, assisting people interested in learning more about the MERN stack development. As a result, the MERN stack demonstrated its ability to construct complicated full-stack applications. However, the final application might be further enhanced by including new features such as responsive web design for various devices, other payment options, third-party mailing services and social network login, sign-in email notifications, and product rating and review capabilities. Furthermore, additional sophisticated principles, such as server-side rendering, and testing, might be included to enhance various elements of the application.

References

Aggarwal, S., 2018. Modern Web-Development using ReactJS. International Journal of Recent Research Aspects, 5(1), p. 133.

Baron, D., 2021. A VS Code Alternative to Postman. [online] Daniela Baron. Available at: https://danielabaron.me/blog/postman-alternative-vscode/ [Accessed 9 March 2022].

DeBill, E., 2022. Module Counts. [online] Available at: http://www.modulecounts.com/ [Accessed 30 March 2022].

Facebook Inc., 2022a. Components and Props. [online] Available at: https://reactjs.org/docs/components-and-props.html [Accessed 09 April 2022].

Facebook Inc., 2022b. Introducing Hooks – React. [online] Available at: https://reactjs.org/docs/hooks-intro.html [Accessed 05 May 2022].

Facebook Inc., 2022c. Introducing JSX. [online] Available at: https://reactjs.org/docs/introducing-jsx.html [Accessed 09 April 2022].

Facebook Inc., 2022d. React – A JavaScript library for building user interfaces. [online] Available at: https://reactjs.org/ [Accessed 24 April 2022].

Facebook Inc., 2022e. Reconciliation. [online] Available at: https://reactjs.org/docs/reconciliation.html [Accessed 24 April 2022].

Facebook Inc., 2022f. Using the Effect Hook – React. [online] Available at: https://reactjs.org/docs/hooks-effect.html [Accessed 12 May 2022].

Facebook Inc., 2022g. Using the State Hook – React. [online] Available at: https://reactjs.org/docs/hooks-state.html [Accessed 12 May 2022].

Facebook Inc., 2022h. Virtual DOM and Internals. [online] Available at: https://reactjs.org/docs/faq-internals.html [Accessed 09 April 2022].

FullStackOpen, 2022. Fullstack part4. [online] FullStackOpen. Available at: https://fullstackopen.com/en/part4 [Accessed 12 March 2022].

GeeksforGeeks, 2021. What is MongoDB - Working and Features. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/ [Accessed 10 April 2022].

Gupta, L., 2022. What is REST. [online] REST API Tutorial. Available at: https://restfulapi.net/ [Accessed 24 April 2022].

Hamedani, M., 2018. React Virtual DOM Explained in Simple English - Programming with Mosh. [online] Programming with Mosh. Available at: https://programmingwithmosh.com/react/react-virtual-dom-explained/ [Accessed 24 April 2022].

Heroku. n.d. Cloud Application Platform | Heroku. [online] Available at: https://www.heroku.com/ [Accessed 24 April 2022].

Ho, C., 2016. Understanding the Middleware Pattern in Express.js. [online] DZone. Available at: https://dzone.com/articles/understanding-middleware-pattern-in-expressjs [Accessed 29 April 2022].

Karnik, N., 2018. Introduction to Mongoose for MongoDB | Codementor. [online] Codementor. Available at: https://www.codementor.io/@theoutlander/introduction-to-mongoose-for-mongodb-gw9xw34el [Accessed 24 April 2022].

Keshishyan, A., 2019. Node.js event loop architecture. [online] Medium. Available at: https://medium.com/preezma/node-js-event-loop-architecture-go-deeper-node-core-c96b4cec7aa4 [Accessed 29 April 2022].

MDN Web Docs. 2022a. Cross-Origin Resource Sharing (CORS) - HTTP | MDN. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS [Accessed 10 March 2022].

MDN Web Docs. 2022b. Introduction to the DOM - Web APIs | MDN. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction [Accessed 24 April 2022].

MDN Web Docs. 2022c. Using HTTP cookies - HTTP | MDN. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies [Accessed 09 February 2022].

MongoDB Inc., 2013. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js | MongoDB Blog. [online] MongoDB. Available at: https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and [Accessed 29 April 2022].

MongoDB Inc. n.d. What Is The MERN Stack? Introduction & Examples. [online] Available at: https://www.mongodb.com/mern-stack [Accessed 24 April 2022].

Mongoose. n.d (a). Mongoose: Models. [online] Mongoose. Available at: https://mongoosejs.com/docs/models.html [Accessed 29 April 2022].

Mongoose. n.d (b). Mongoose: Schemas. [online] Available at: https://mongoosejs.com/docs/guide.html [Accessed 29 April 2022].

OpenJS Foundation. 2022. Express - Node.js web application framework. [online] Available at: https://expressjs.com/ [Accessed 24 April 2022].

OpenJS Foundation. n.d. Introduction to Node.js. [online] Available at: https://nodejs.dev/learn/introduction-to-nodejs/ [Accessed 24 April 2022].

Singh, S., 2020. What Is NPM?—A Simple English Guide to Truly Understanding the Node Package Manager. [online] Medium. Available at: https://medium.com/swlh/what-is-npm-a-simple-english-guide-to-truly-understanding-the-node-package-manager-41e82f6c5515 [Accessed 24 April 2022].

Stack Overflow. 2022. Stack Overflow Developer Survey 2021. [online] Available at: https://insights.stackoverflow.com/survey/2021#technology-web-frameworks-all-respondents2 [Accessed 24 April 2022].

Tailwind Labs Inc., 2022. Install Tailwind CSS with Create React App. [online] Tailwindcss. Available at: https://tailwindcss.com/docs/guides/create-react-app [Accessed 15 March 2022].

UNCTAD, 2021. How COVID-19 triggered the digital and e-commerce turning point. [online] UNCTAD. Available at: https://unctad.org/news/how-covid-19-triggered-digital-and-e-commerce-turning-point [Accessed 29 April 2022].

Visual Studio Code. n.d. Documentation for Visual Studio Code. [online] Available at: https://code.visualstudio.com/docs [Accessed 24 April 2022].

W3schools. n.d. JavaScript HTML DOM. [online] Available at: https://www.w3schools.com/js/js_htmldom.asp [Accessed 29 April 2022].

Wikipedia. 2022a. Express.js - Wikipedia. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Express.js [Accessed 29 April 2022].

Wikipedia. 2022b. MongoDB - Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/MongoDB [Accessed 24 April 2022].

Wikipedia. 2022c. npm (software). [online] Available at: https://en.wikipedia.org/wiki/Npm_(software) [Accessed 05 May 2022].

Wirantono, M., 2020. LocalStorage vs. Cookies: All You Need to Know About Storing JWT Tokens Securely in the Frontend - JavaScript inDepth. [online] inDepthDev. Available at: https://indepth.dev/posts/1382/localstorage-vs-cookies [Accessed 10 March 2022].