



VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Lingxiao Huang

FULL-STACK E-COMMERCE ONLINE STORE

A Web Application for Purchasing and Uploading Products for Every User

Technology
2022

ACKNOWLEDGEMENTS

I would like to show my gratitude towards Professor Ghodrat Moghadampour, my supervisor for giving support during my thesis project and thesis writing.

I would like to give my appreciation to VAMK, University of Applied Sciences where I have studied for five years. All the knowledge and skills I learned there are always valuable.

In addition, I am grateful for Mr. Umer, Mr Phi, Mr Sangam and other Integrify team members for giving me the opportunities to gain experience of all the skills for this project in Integrify Academy. I believe what I learn here will always be an asset in my future career.

Finally, I would like to send my love to my family and my friends who have supported me throughout the university studying period. All the love and support I have received will not be forgotten.

Vaasa 30/04/2022

Lingxiao Huang

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Lingxiao Huang
Title	Full-stack E-commerce Online Store
Year	2022
Language	English
Pages	59
Name of Supervisor	Ghodrat Moghadampour

This thesis focused on creating a full stack web application for people to buy products, upload and sell items and place orders. The application in this project was built to give users an online platform to sell and but second-hand items.

The technologies used to develop the application were React, TypeScript, SASS and Cloudinary in frontend and Node, Express, MongoDB and Nodemailer in backend.

The project resulted in a full-stack application, which allows users to create accounts, upload items to sell, add items to the shopping cart and place orders. Moreover, the administrator can login to the management console page to view and perform different actions on users, products and orders.

Both the client and server applications have been deployed in Heroku (a cloud services provider that can host dynamic web applications) successfully.

Keywords	E-commerce, full-stack, React, MongoDB, and cloud services
----------	--

CONTENTS

ABSTRACT

INTRODUCTION	1
1.1 Background	1
1.2 Motivation.....	1
2 TECHNOLOGIES AND TOOLS.....	3
2.1 TypeScript Programming Language	4
2.2 React Framework	4
2.3 Redux	6
2.4 SASS.....	7
2.5 NodeJS.....	7
2.6 Express Framework.....	8
2.7 MongoDB & MongoDB Atlas.....	8
3 APPLICATION DESCRIPTION.....	11
3.1 Application Overview and Constraints	11
3.2 Application Requirements Analysis	12
3.3 Use Case Diagram	14
3.4 Sequence Diagrams.....	14
3.4.1 Add Items to Cart Sequence Diagram	14
3.4.2 Add New Listing Sequence Diagram	15
3.5 Class Diagram.....	16
4 USER INTERFACE DESIGN.....	19
4.1 User Interface for General Users	19
4.2 User Interface for Admin Management Console.....	27
5 IMPLEMENTATION.....	32
5.1 Integrated Development Environment.....	32
5.2 Backend Server	33
5.2.1 Project Environment	33
5.2.2 Directory Structure.....	33

5.2.3	Application Source Code	35
5.3	Frontend Client	42
5.3.1	Project Environment	42
5.3.2	Directory Structure.....	43
5.3.3	Application Source Code	45
6	APPLICATION TESTING.....	54
7	CONCLUSIONS	58
	REFERENCE	60

LIST OF FIGURES AND TABLES

Figure 1. Full stack development (Singh, 2019)	3
Figure 2. npx create-react-app	5
Figure 3. Redux workflow	7
Figure 4. MongoDB Compass	9
Figure 5. MongoDB Atlas User Interface	10
Figure 6. Software application architecture	12
Figure 7. The application main functions	14
Figure 8. Add Items to Cart Sequence diagram	15
Figure 9. Add New Listing Sequence diagram	16
Figure 10. User, product and CartItem class diagram	17
Figure 11. Order and billingInfo class diagram	18
Figure 12. Home page with user not logged in	19
Figure 13. ItemDetails page UI	20
Figure 14. Login and Register page	21
Figure 15. Email to be confirmed page	21
Figure 16. Email Confirm Message Page	22
Figure 17. Home page with user logged in	22
Figure 18. Cart page	23
Figure 19. Order page	24
Figure 20. Check out success page	25
Figure 21. Profile page	25
Figure 22. Deletion confirmation modal	26
Figure 23. New listing modal	26
Figure 24. Order detail page	27
Figure 25. Admin login	28
Figure 26. Admin home page	28
Figure 27. Admin user management page	29
Figure 28. Admin products page	30

Figure 29. Admin product detail page	30
Figure 30. Admin orders page	31
Figure 31. Visual Studio Code user interface	33
Figure 32. Directory Structure backend	34
Figure 33. Directory Structure frontend	44
Figure 34. Components directory tree	47
Figure 35. Pages directory tree	48
Figure 36. Jest test result in the terminal window	57
Table 1. Application Requirements	13

LIST OF CODE SNIPPETS

Code Snippet 1. TypeScript Installation	4
Code Snippet 2. Create React App	5
Code Snippet 3. Start React App	5
Code Snippet 4. server.ts	35
Code Snippet 5. app.ts	36
Code Snippet 6. auth router	37
Code Snippet 7. auth controller	37
Code Snippet 8. order controller createOrder method	38
Code Snippet 9. order service	40
Code Snippet 10. CartItem model	41
Code Snippet 11. Order model	42
Code Snippet 12. package.json frontend sample	43
Code Snippet 13. ReactDOM with Redux and Router setup in index.tsx	46
Code Snippet 14. React Routes defined in app.tsx	46
Code Snippet 15. The home.tsx function and state definitions	49
Code Snippet 16. The home.tsx file first usage of useEffect	50
Code Snippet 17. The home.tsx file second usage of useEffect	51
Code Snippet 18. The home.tsx file return statement	52
Code Snippet 19. Create a listing and product test case	55
Code Snippet 20. Get back an existing product test case	56
Code Snippet 21. Not get back a non-existing product	56
Code Snippet 22. Test script configuration in package.json file	57

LIST OF ABBREVIATIONS

API – Application Programming Interface

CSS - Cascading Style Sheets

CORS – Cross-Origin Resource Sharing

DOM – Document Object Model

GUI – Graphical User Interface

PC – Personal Computer

SDK – Software Development Kit

HTML – Hypertext Markup Language

IDE – Integrated Development Environment

SASS – Syntactically Awesome Stylesheets

URL – Uniform Resource Locator

UI – User Interface

INTRODUCTION

1.1 Background

With the increasing number of digital devices in our daily lives, it now has become a common practice for people to use their computing devices to perform many tasks and use different digital services. For instance, online shopping services, online hotel reservation services and online plane tickets purchasing services have been booming in the past decades. For most PC users nowadays, a web browser is installed on their computers. Therefore, many digital services are provided through World Wide Web in the browser.

A full-stack application is a software application that has both a frontend application and a backend application which combines to a large application with client-side and server-side. Full-stack application development is beneficial for developers and customers in different ways, such as cost-effectiveness, faster development and troubleshooting, flexibility, and maintenance.

1.2 Motivation

The main general motivation of this thesis is to create a web-based full-stack application with a React frontend and a Node backend for users to purchase and sell products. The application has a clear UI design and is easy to use for any computer users.

Users should be able to register new accounts. After emails are verified, users can login with their accounts in the application. The application has implemented authentication and authorization methods for better security. Users can add items to their carts and view the cart items after logout and login again. Then users can make orders with their billing information and the system will store order details in the database. In addition, users can upload their own items to the application

or remove items. For system administrator, actions such as user suspension and products removal can be conducted.

2 TECHNOLOGIES AND TOOLS

This section discusses the tools and technologies used in developing the web application. A full-stack web application development generally consists of two parts: frontend development and backend development. Frontend web development is the development of the graphical user interface of a website, using HTML, CSS, and JavaScript, so that users can view and interact with that website (Wikipedia, 2022). In this case, ReactJS was used as a JavaScript and TypeScript framework and SASS as a CSS framework. Besides, Redux was used as a React state management tool. Backend development, also known as server-side development, is about the things that users cannot see but happen behind the scenes. It focuses on databases, backend logic, APIs, and Servers (InterviewBit, 2021). In this project NodeJS and Express were used to build the backend server and APIs. MongoDB was used as a database connected to the backend server.

FULL-STACK DEVELOPMENT

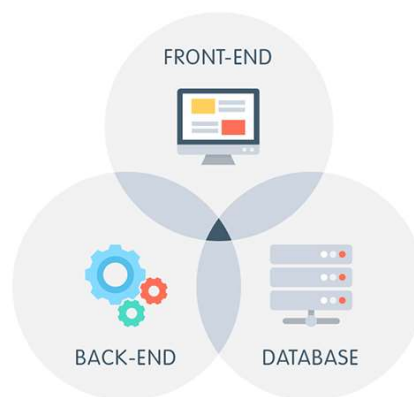


Figure 1. Full stack development (Singh, 2019)

2.1 TypeScript Programming Language

TypeScript was developed by Microsoft in 2012 and it is a superset of JavaScript. TypeScript is designed for the development of large applications (Wikipedia, 2022). It adds strict typing to JavaScript, which helps to reduce bugs and errors in the code. In addition, TypeScript adds Object-Oriented Programming features such as class and interface that are not supported in the original JavaScript. To run a TypeScript file, TypeScript has to be installed and its compiler have to be installed via NPM or yarn:

```
npm install -g typescript
yarn add -g typescript
```

Code Snippet 1. TypeScript Installation

After that run **tsc** can be run followed by a typescript filename to compile the TS file to JS file. Then the compiled JS file can be run through the browser or in the Node environment.

2.2 React Framework

React is a JavaScript library (also known as a JavaScript framework) for building front end user interfaces. When using React to build webpage, React creates a Virtual DOM in memory and React only changes what needs to be changed. React enables programmers to use JSX to write HTML elements in JavaScript code.

To create a React application, Node must be installed. Then in the terminal, the following commands are run:

```
npx create-react-app [your_app_name] for JavaScript React
```

```
npx create-react-app [your_app_name] --template  
typescript for TypeScript React
```

Code Snippet 2. Create React App

```
Huang@Jason-Asus MINGW64 ~/React-Projects  
$ npx create-react-app my-demo-react  
  
Creating a new React app in C:\Users\Huang\React-Projects\my-demo-react.  
  
Installing packages. This might take a couple of minutes.  
Installing react, react-dom, and react-scripts with cra-template...  
  
added 1371 packages in 40s  
  
169 packages are looking for funding  
  run `npm fund` for details  
  
Initialized a git repository.  
  
Installing template dependencies using npm...  
npm WARN deprecated source-map-resolve@0.6.0: See https://github.com/lydell/source-map-resolve#deprecated  
added 38 packages in 4s  
  
169 packages are looking for funding  
  run `npm fund` for details  
Removing template package using npm...  
  
removed 1 package, and audited 1409 packages in 3s
```

Figure 2. npx create-react-app

To start the React application, the following commands are processed (see Code Snippet 3):

```
npm start
```

```
yarn start
```

Code Snippet 3. Start React App

Some important concepts of React are components, props, and states. React components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML (W3Schools, n.d.). React props stand for properties. They are like function

arguments that can be passed via from and to components. React states are like variables in JavaScript that are mutable. They can be mutated through state updating functions.

There are two ways of writing React components: one is writing functional components which is used in this project; another is writing class components. Before React 16.8, class components were the only way to track state and lifecycle on a React component. Function components were considered "state-less". With the addition of Hooks, Function components are now almost equivalent to Class components.

2.3 Redux

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across the entire application, with rules ensuring that the state can only be updated in a predictable fashion (Redux, 2022). Redux is a great tool for managing applications with large number of states and these states need to be accessed across the entire application. However, not all applications need to use Redux. For example, some applications with small number of states and the states do not update frequently.

Generally Redux is more useful when:

- There are large amounts of application states that are needed in many places in the application
- The application state is updated frequently over time
- The logic to update that state may be complex
- The application has a medium or large-sized codebase, and might be worked on by many people

(Redux, 2022)

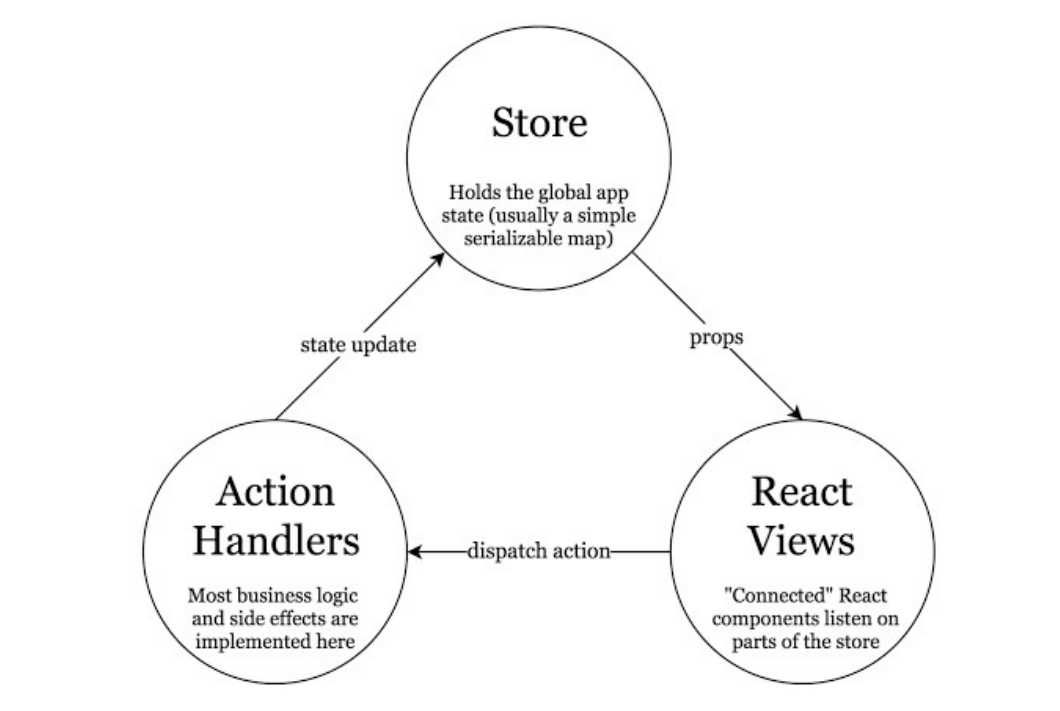


Figure 3. Redux workflow

2.4 SASS

SASS stands for Syntactically Awesome Stylesheet, and it is an extension to CSS. It allows users to use variables, nested rules, mixins, functions and more with CSS-compatible syntax. SASS helps to keep large stylesheets well-organized and makes it easy to share design within and across projects (SASS, n.d.).

2.5 NodeJS

NodeJS is a JavaScript runtime environment built on Chrome's V8 JavaScript engine and it enables JavaScript code executed outside a web browser. Node.js brings event-driven programming to web servers, enabling development of fast

web servers in JavaScript. Developers can create scalable servers without using threading, by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task (Sons, 2012).

2.6 Express Framework

Express is a fast, unopinionated, minimalist web framework for Node.js. Express provides a robust set of features for web and mobile applications (Express, 2017). It provides many mechanisms to write HTTP handlers with different URL paths named routes and offers integration with built-in or third-party middleware functions which process tasks in the middle of request and response. Programmers can add additional libraries to work with cookies, encryption, CORS, web token, authentication & authorization, database connectivity, logging, error handling and a lot more.

2.7 MongoDB & MongoDB Atlas

MongoDB is a cross-platform, document-oriented non-relational database that provides high performance, high availability, and easy scalability. MongoDB works on a concept of collection and document; a collection in MongoDB is a group of documents which is the equivalent of a relational database table. The difference is collections do not enforce a database schema. A document is a set of key-value pairs and has dynamic schema, which implies documents in the same collection do not need to have the same set of fields or structure (Tutorialspoint, n.d.).

There are many ways to work with MongoDB. During the development period in this project, MongoDB Shell is used which is a command-line interface to work with MongoDB. It allows access to MongoDB Shell through Windows command prompt. In addition, MongoDB Compass is used which is an intuitive and flexible GUI tool for MongoDB data management.

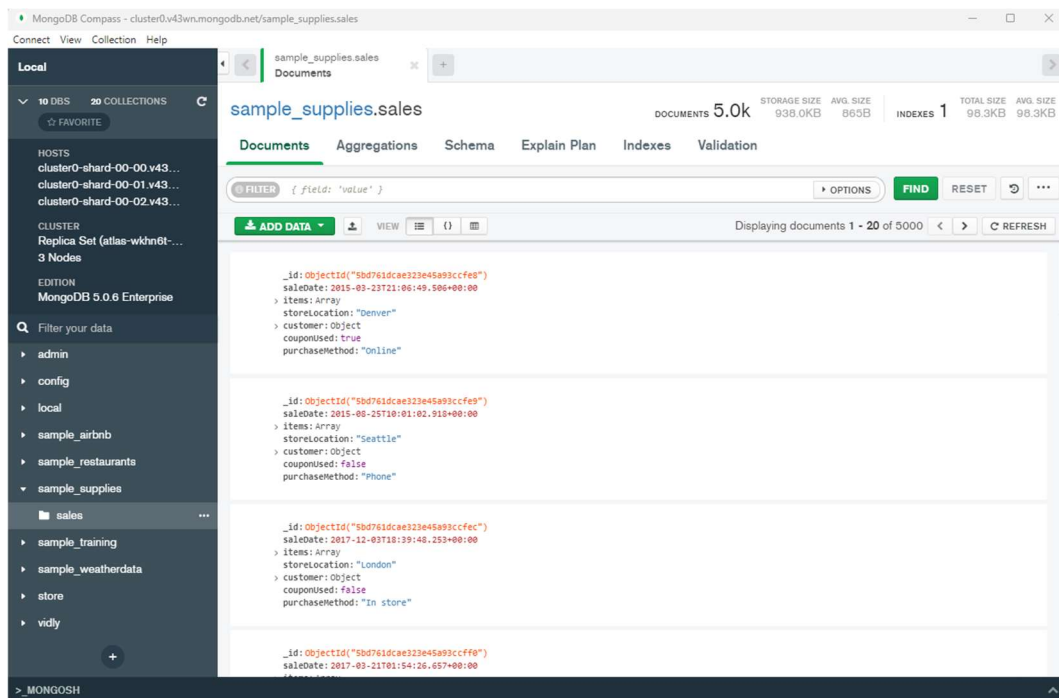


Figure 4. MongoDB Compass

MongoDB Atlas was used in the project deployment. MongoDB Atlas is a multi-cloud database service that simplifies the deployment and management of databases while offering the versatility needed to build resilient and performant global applications on the cloud providers. (MongoDB, n.d.). It also offers SDK tools for different programming languages (in this case Node.js) to access MongoDB Atlas programmatically.

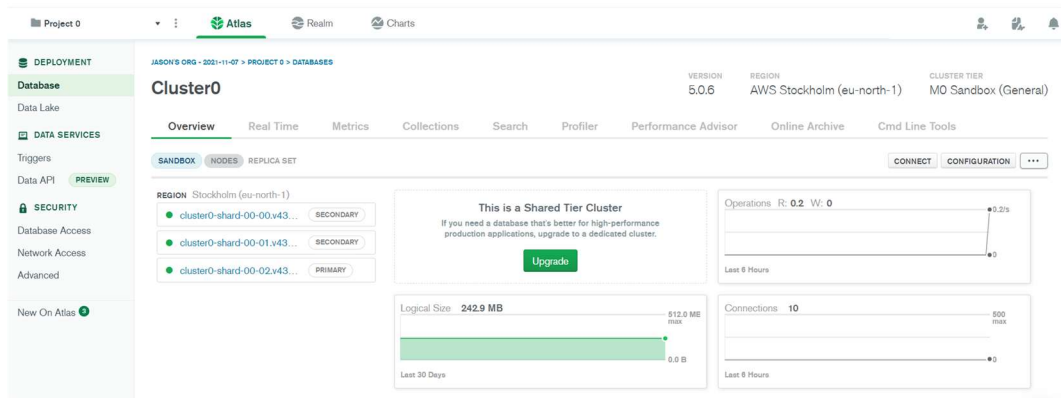


Figure 5. MongoDB Atlas User Interface

3 APPLICATION DESCRIPTION

In this chapter, the application project, requirements, objectives and constraints in details are described. First, we will look at the application structure overview and constraints. Then project requirements are analyzed with a requirement table. Finally, different UML diagrams will be presented to demonstrate project functionalities and objectives in great details.

3.1 Application Overview and Constraints

The main objectives of this project are to create an online second-hand store for people to trade their own items in an acceptable price with no middleman charging commissions. The online store or platform is completely user-oriented which means all actions and services are centered around users.

To create such application meeting the objectives, a full-stack application was developed which consists of three essential components:

- Frontend Client
- Backend Server
- Database

The web frontend developed in React communicates with Node server via HTTP requests. The Node server exchanges data with MongoDB with MongoDB data querying language.

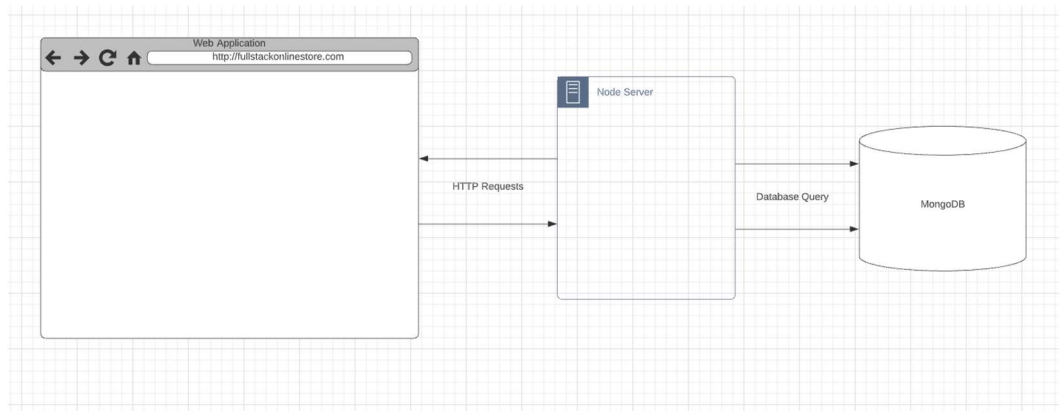


Figure 6. Software application architecture

The project has satisfied the basic objectives and requirements. However, there are certain constraints of this application due to time shortage and technical difficulties and they are also mentioned in requirements analysis. The constraints include:

- No real payment solutions
- User account data is not comprehensive
- Data exchange over http

3.2 Application Requirements Analysis

The application requirements display required functions ordered by different priority levels. Table 1 shows the implemented functions in this project and also some improvement suggestions.

Table 1. Application Requirements

References	Description	Priority (1. Must have 2. Should have 3. Good to have)
F1	User account registration and login	1
F2	Adding, removing and modifying products in cart	1
F3	New listing upload	1
F4	Order creation from check-out	1
F5	Administration management console	2
F6	Email Verification during registration	2
F7	Data exchange over HTTPS	3
F8	Real payment solutions like PayPal or Stripe	3

3.3 Use Case Diagram

The use case diagram includes all functions in the application. The primary use cases for this application are viewing all products, adding items to cart and adding new listing.

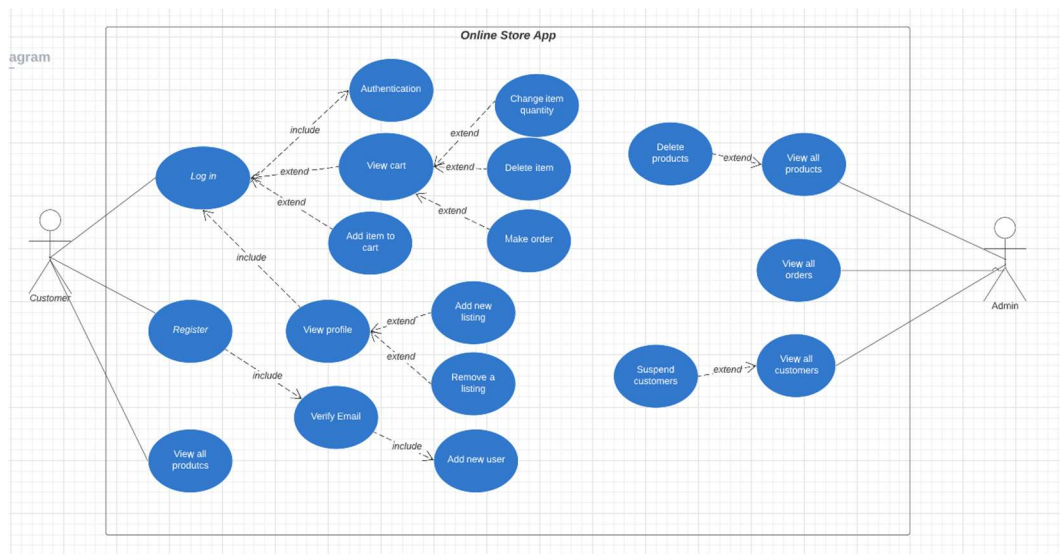


Figure 7. The application main functions

3.4 Sequence Diagrams

In this thesis, only two sequence diagrams for program operations are discussed: **add items to cart** and **add new listing**. There are other operations in this application and their sequence diagrams is similar to these two.

3.4.1 Add Items to Cart Sequence Diagram

One of the main functions of this application is the ability for users to add items to their carts. First, users access the application and log in with credentials and client sends requests to the server. If provided credentials are invalid, server responds an error message. After a successful login, users perform *add items to cart* action

and then server will update a user cart document in MongoDB database. Finally, server responds with a 200 -success code if everything is correct.

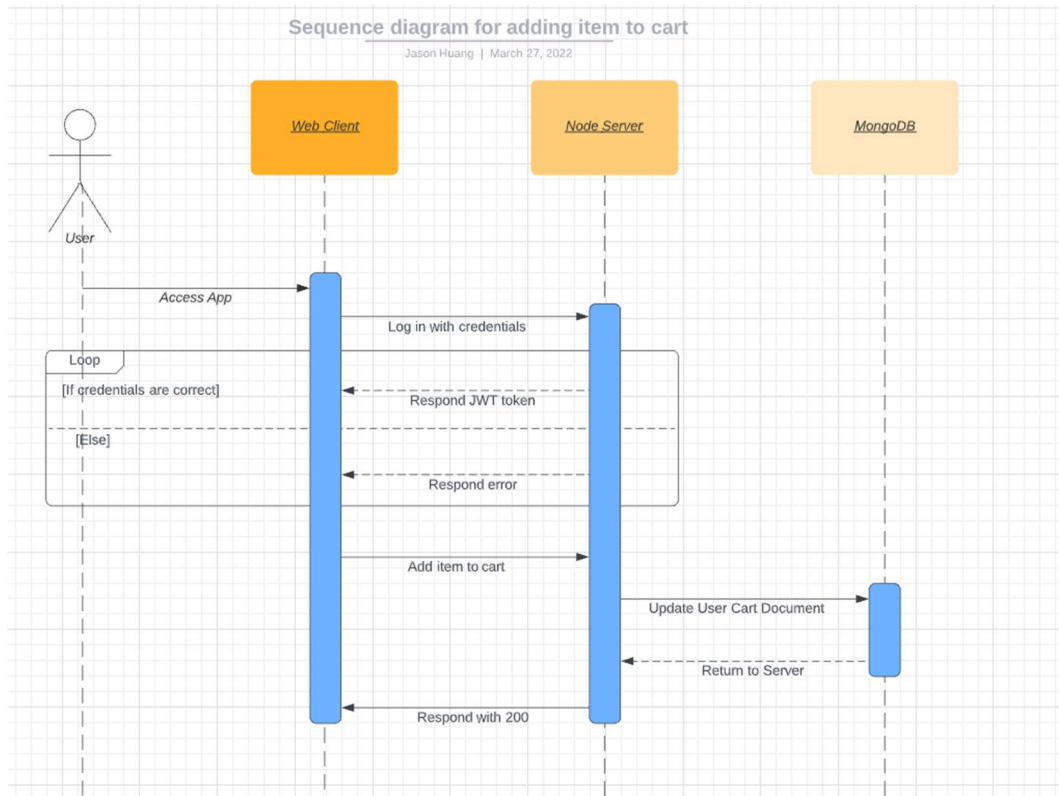


Figure 8. Add Items to Cart Sequence diagram

3.4.2 Add New Listing Sequence Diagram

Another main function of this application is for users to add new listing to the listing document. Again, users access the application and log in with credentials and client sends requests to the server. If provided credentials are invalid, the server responds with an error message. After login, users perform *add new listing* action and then the server will update user listing document in the MongoDB database. Finally, the server responds with a 200 success code if everything is correct.

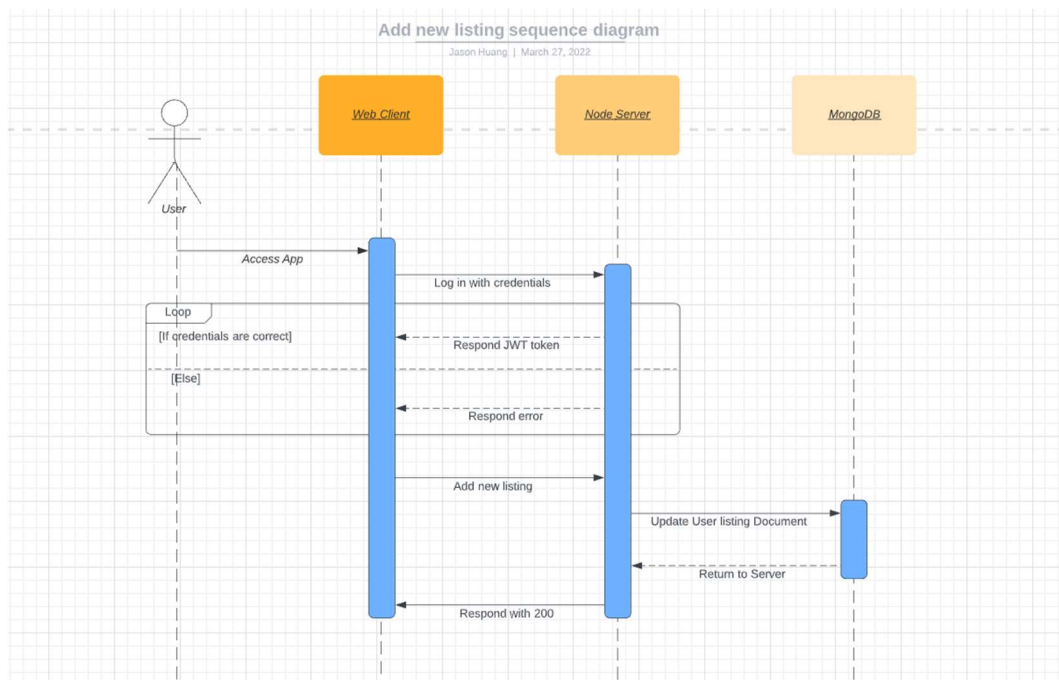


Figure 9. Add New Listing Sequence diagram

3.5 Class Diagram

The class diagram contains all database models defined in the backend Node application. For example, the User class has name, email, password as string, cart, listings and orders as array of custom types. The billingInfo class is a subclass within Order class. The relationships of classes are all composition, which indicates that Product, Order and CartItem class does not exist without User class.

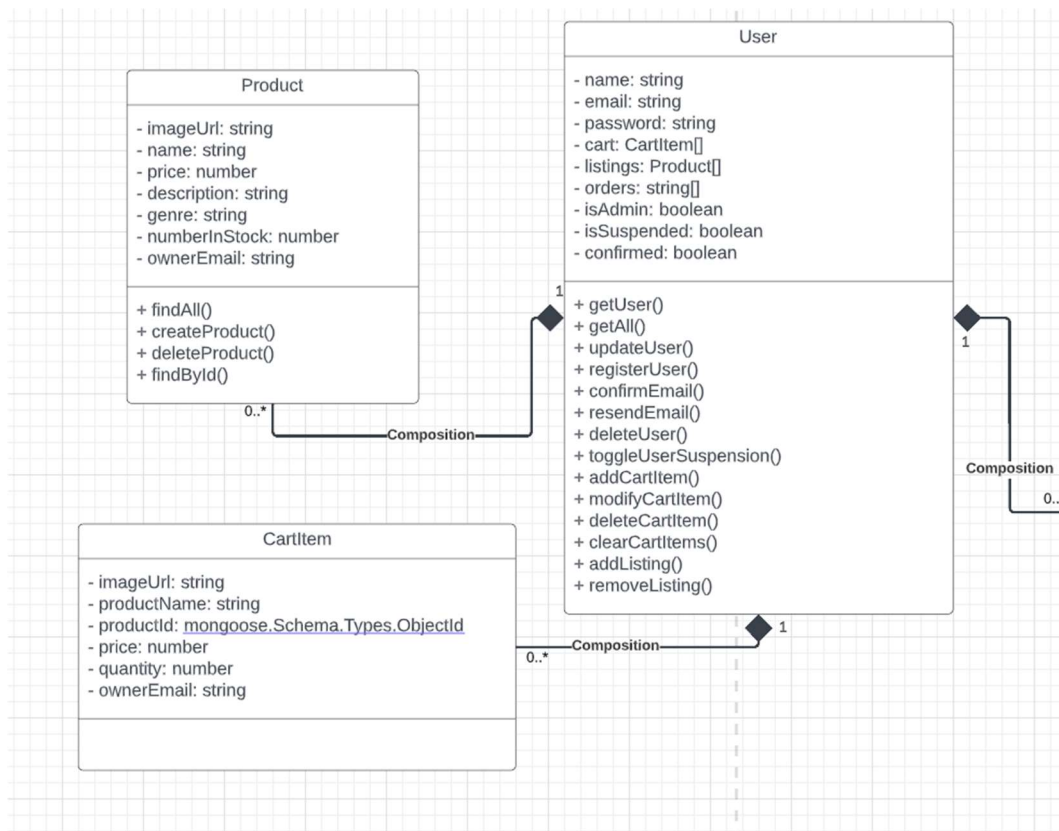


Figure 10. User, product and CartItem class diagram

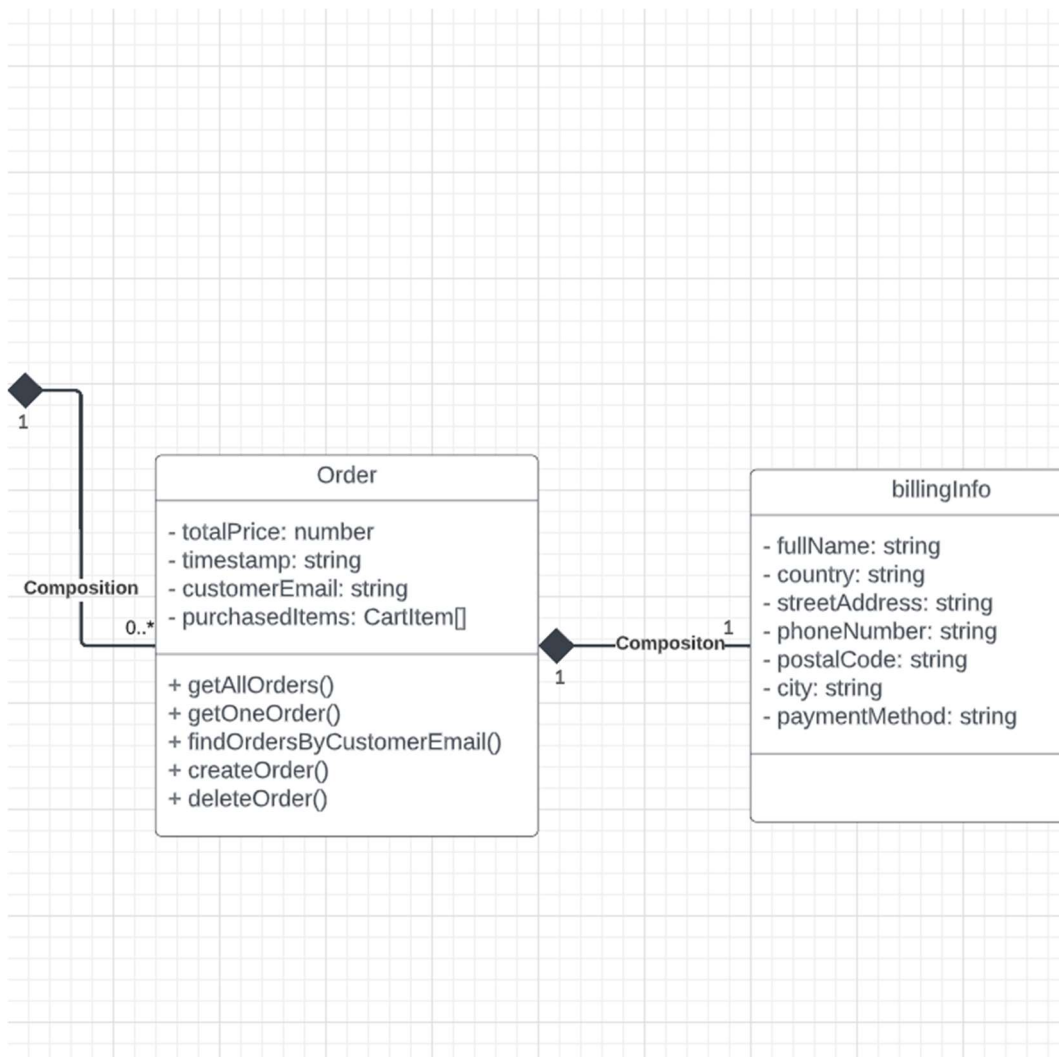


Figure 11. Order and billingInfo class diagram

4 USER INTERFACE DESIGN

This section explains how the graphical user interface is designed in this application. The user interface has been designed for general users and admin management console.

4.1 User Interface for General Users

The design principles of this application are reactivity, user-friendliness, and minimalism. Every action in the application will give users some feedback for notification purpose. Its minimalist design also makes the application very simple and easy to use.

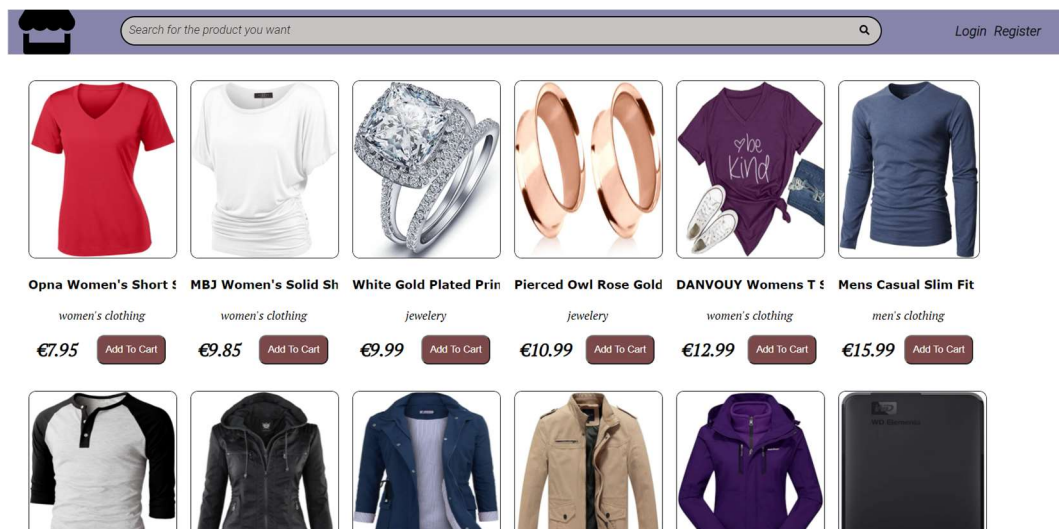


Figure 12. Home page with a user not logged in

On the Home page, there is a search bar, navigation bar in the header and a list of products in the body of the website. On the left top corner there is a store icon from Font Awesome, which gives the users basic understanding of the website. Then there is a search bar on the right of store icon with a search icon at the end

of search bar. The search bar has a product searching functionality. Then in the navigation bar there are two navigation buttons: login and register, both of which will lead to different pages. In the website body, a list of all products is shown in the form of image, product name, product type, price and the “Add to Cart” button. The “Add to Cart” button will only work when users are logged in and the logged in user is not the seller of the product. If the user clicks on the product image or product name, the application will route to the ItemDetails page which is shown in Figure 13.

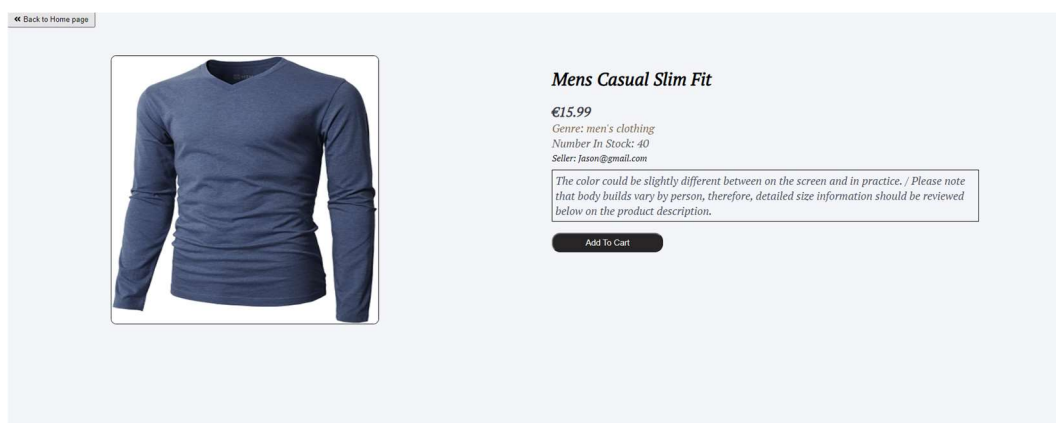
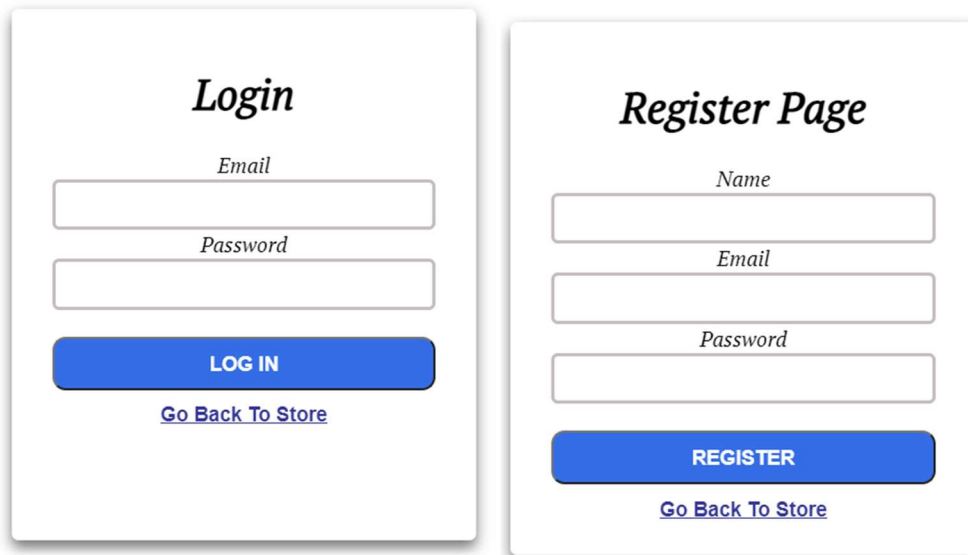


Figure 13. ItemDetails page UI

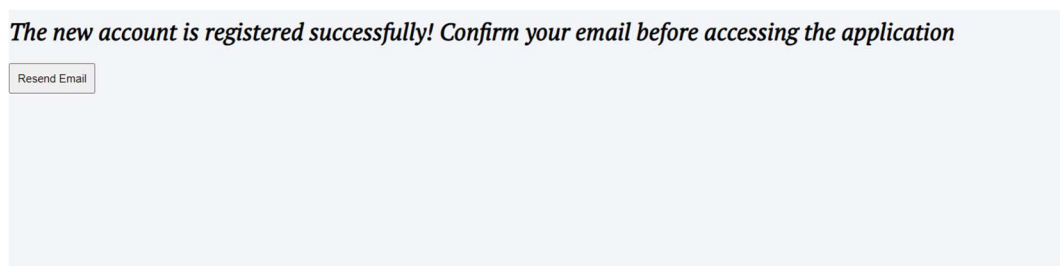
When the user clicks on the product image or product name in Home Page, they will be directed to the ItemDetails Page. It has a large product image on the left side and more product information on the right side, for example: name, price, genre, number in stock, seller email and description. An item can be added to the cart on the detail page. On the left top corner there is the “Back to Home Page” button which will redirect users back to the Home Page.



The image shows two side-by-side UI cards. The left card is titled "Login" and contains two input fields labeled "Email" and "Password". Below the fields is a blue button labeled "LOG IN" and a link labeled "Go Back To Store". The right card is titled "Register Page" and contains three input fields labeled "Name", "Email", and "Password". Below the fields is a blue button labeled "REGISTER" and a link labeled "Go Back To Store".

Figure 14. Login and Register page

The Login page and the Register page are implemented similarly in a simple UI design. They are contained in a card component with a header tag, form and two buttons in the bottom. On the Register page, if a new user registers successfully, the application will direct the user to the EmailToBeConfirmed page for email verification shown in Figure 15.



The image shows a light blue background with a header message: "The new account is registered successfully! Confirm your email before accessing the application". Below the message is a button labeled "Resend Email".

Figure 15. Email to be confirmed page

On the EmailToBeConfirmed page, a text of header message is displayed and below that there is a "Resend Email" button in case the user does not receive the

verification email, or some users try to login without emails being confirmed (see Figure 16).

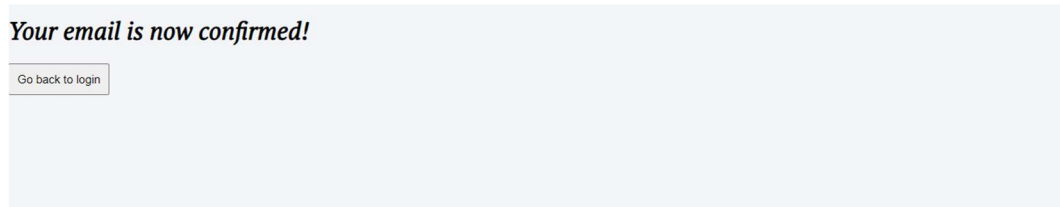


Figure 16. Email Confirm Message Page

After the user opens the link in the confirmation email, a dynamic confirm page will be displayed based on the user id in the database. On the page a confirmation message is shown and the “Go back to login” button can direct the user to the login page.

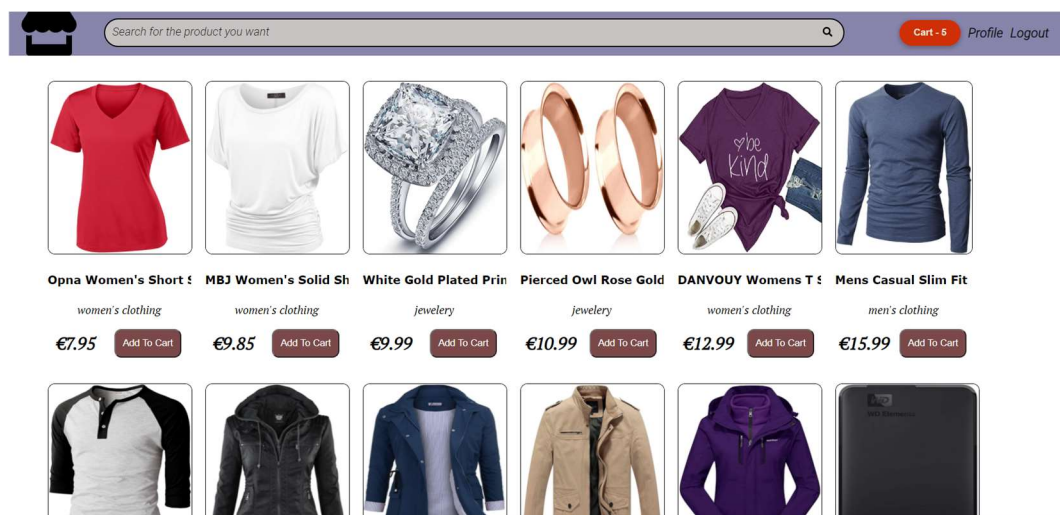


Figure 17. Home page with the user logged in






When the user logs in successfully through the Login Page, the Home Page with users logged in is shown (Figure 17). The only difference with the Home Page not logged in is the changes in the navigation bar. On the logged in Home Page there

are two navigation buttons: Cart, Profile and Logout buttons. The Cart and Profile buttons will lead to different pages and the Logout button will just log the user out and the application will update the UI to the not logged in Home Page.

If the user is on the Home Page logged in, they can add selected items to their cart by clicking the “Add to Cart” button on each item. This action will not only add an item to the cart but also updates the items count in the navigation bar Cart button. The system will check if the item already exists in the cart or not and decide whether to increment the items count or not.

« Back to Home page

5 items in basket

	DANVOUY Womens T Shirt Casual Cotton Short	1	Total: €12.99	✕
	BIYLACLESEN Women's 3-in-1 Snowboard Jacket Winter Coats	2	Total: €113.98	✕
	WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive	1	Total: €114.00	✕
	Dell Alienware	1	Total: €1500.00	✕
	Mercedes-Benz E350	1	Total: €80000.00	✕

Total Price: €81740.97

[Check Out](#)

Copyright © 2022 LingxiaoHuang. All rights reserved.

Figure 18. Cart page

On the Cart Page (Figure 18), the user will find all the items added to cart. The user can change the quantity of items which will change the total price as well and delete an item. All the actions will update the UI and updates will be stored on the backend server. In the right bottom corner, the total price of the cart items are shown and as well as the check out action which leads users to the Order Page (Figure 19).

« Back to Home page

Your Cart Overview:

Product	Price	Quantity	Total
Mens Casual Premium Slim Fit T-Shirts	22.30	1	22.30

Enter your billing information:

Full Name Country Street Address

Phone number Postal Code City

Payment Options: PayPal Apple Pay Google Pay Alipay Visa Card

Total Price: €22.30

Pay Now

Figure 19. Order page

On the Order page, the user will see a summary of ordered items in the cart with information about the single price, quantity, and total price. Figure 19 shows a form for user billing information, and it asks general shipping details, including name, country, address, phone number and payment methods.

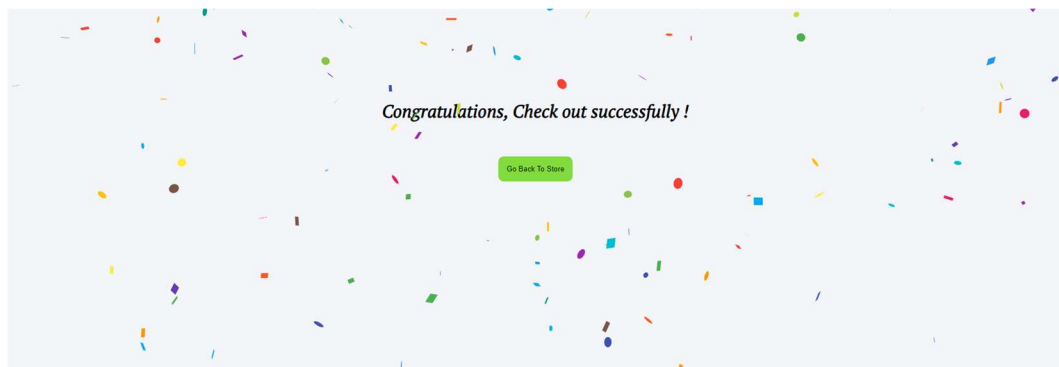


Figure 20. Check out success page

After the user clicks the “Pay Now” button with all billing information entered properly, it will go to the CheckoutSuccess Page that displays a check out success message (Figure 20). To make the operation more interesting, a confetti animation effect is added. The user can then go back to the store using the button below the message.

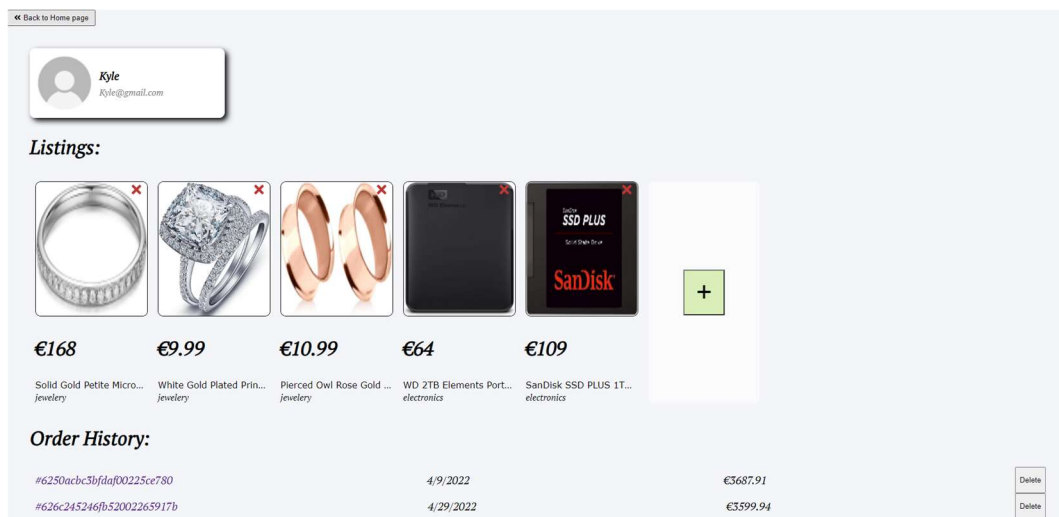


Figure 21. Profile page

If the user goes to the Profile Page with the “Profile” link in the home navigation bar, they will find their user information and all the listings that they have posted in the application. Below the listings they can also find the order history that

contains all the orders they have completed. The user information section shows username and email address. In the Listings section, the user can add a new listing by clicking the green add icon at the end of listings and delete a listing.

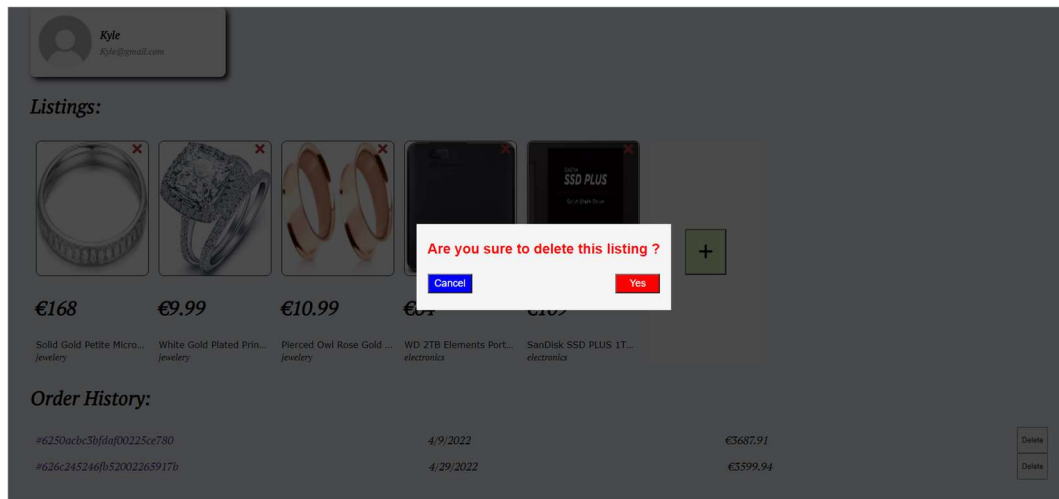


Figure 22. Deletion confirmation modal

When the user clicks the red arrow in the listing, a modal (also called popup) will show up for deletion confirmation (see Figure 22).

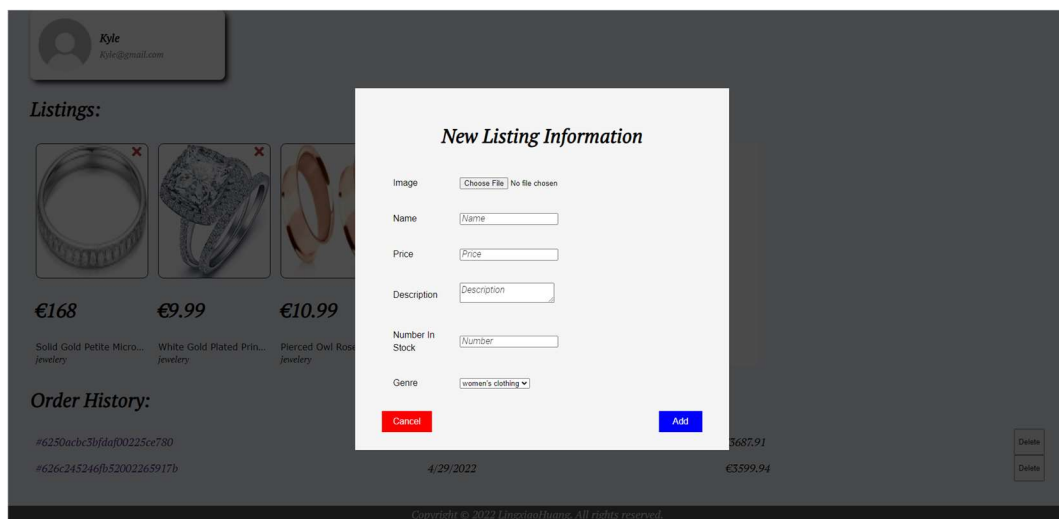
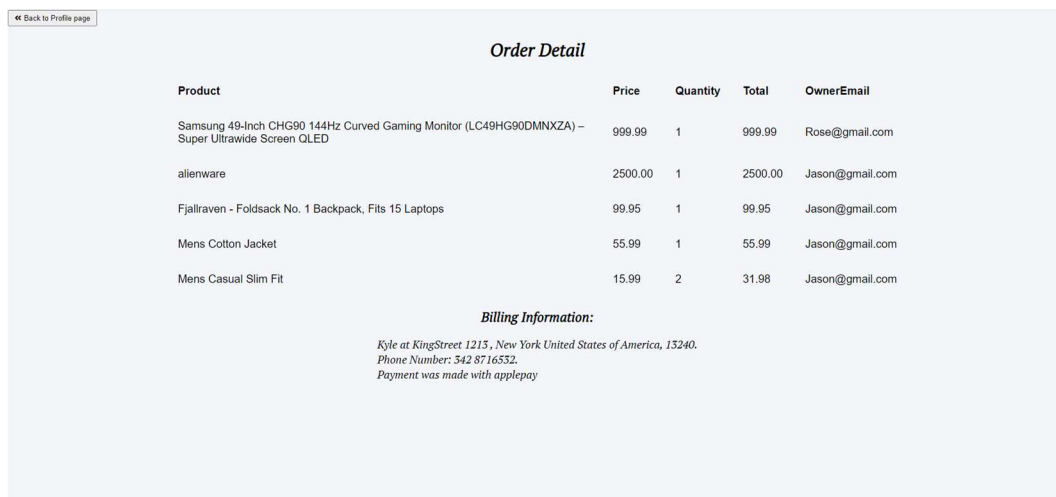


Figure 23. New listing modal

A New Listing form modal (Figure 23) is displayed when the user wants to add a new listing item. There the user uploads an image file, enters name, price,

description, number in stock and genre information. If all information is entered correctly, a new product (also listing) will be created and displayed on both the Profile page and the Home page.



Order Detail

Product	Price	Quantity	Total	OwnerEmail
Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED	999.99	1	999.99	Rose@gmail.com
alienware	2500.00	1	2500.00	Jason@gmail.com
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops	99.95	1	99.95	Jason@gmail.com
Mens Cotton Jacket	55.99	1	55.99	Jason@gmail.com
Mens Casual Slim Fit	15.99	2	31.98	Jason@gmail.com

Billing Information:
 Kyle at KingStreet 1215, New York United States of America, 13240.
 Phone Number: 342 8716532.
 Payment was made with applepay

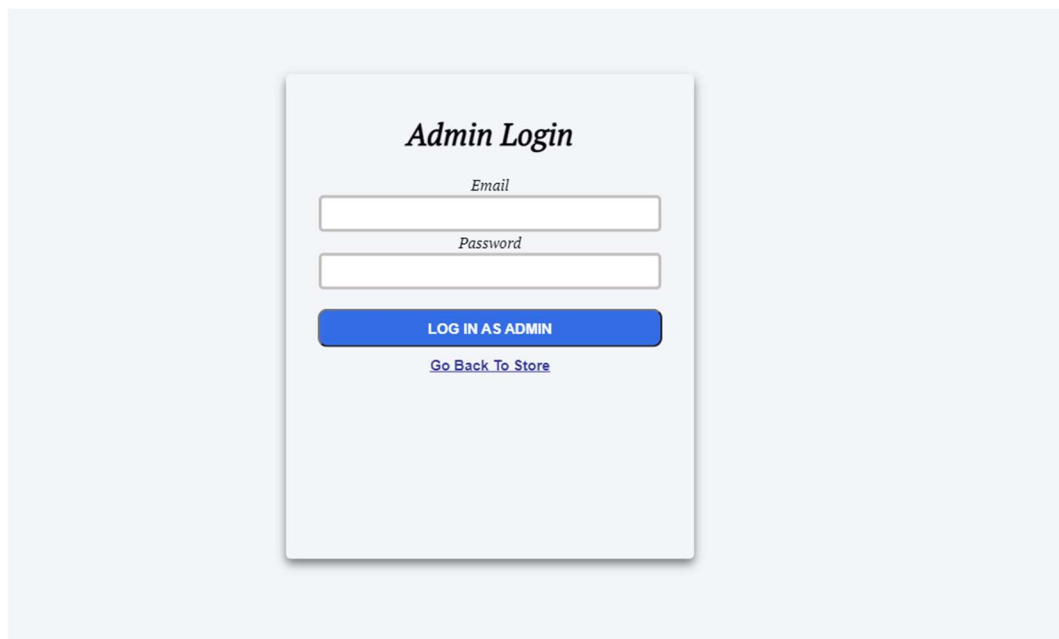
Figure 24. Order detail page

If the user clicks on the order Id on the Profile Page in the Order History section, it will direct to the OrderDetail Page. The user can view purchased items and billing information.

4.2 User Interface for Admin Management Console

Apart from user interface for general users, there is also an admin management console for admin user to manage suspension of user accounts, deletion of products and etc.

The admin login page, shown in Figure 25, can only be accessed through the correct URL. The log in function will not work for general user accounts.



The image shows a centered login form titled "Admin Login". It features two input fields: the first is labeled "Email" and the second is labeled "Password". Below the input fields is a blue button with the text "LOG IN AS ADMIN". Underneath the button is a link that says "Go Back To Store". The entire form is set against a light blue background.

Figure 25. Admin login

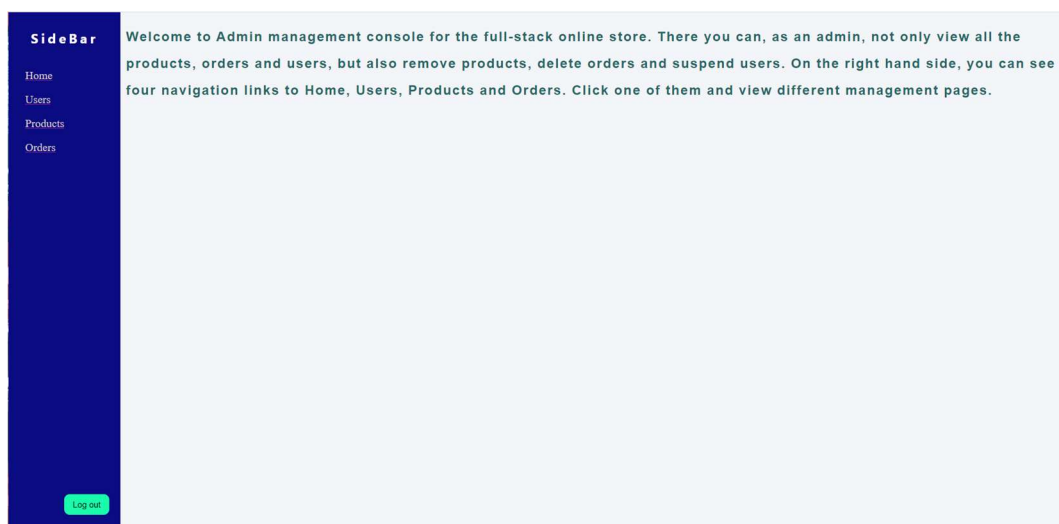


Figure 26. Admin home page

After the admin successfully logs in, the application shows the Admin Home Page with a side navigation bar on the left side and a welcome message. The side bar has four navigation links: Home, Users, Products and Orders. Each one of them correspond to one page. The default page is the home page.

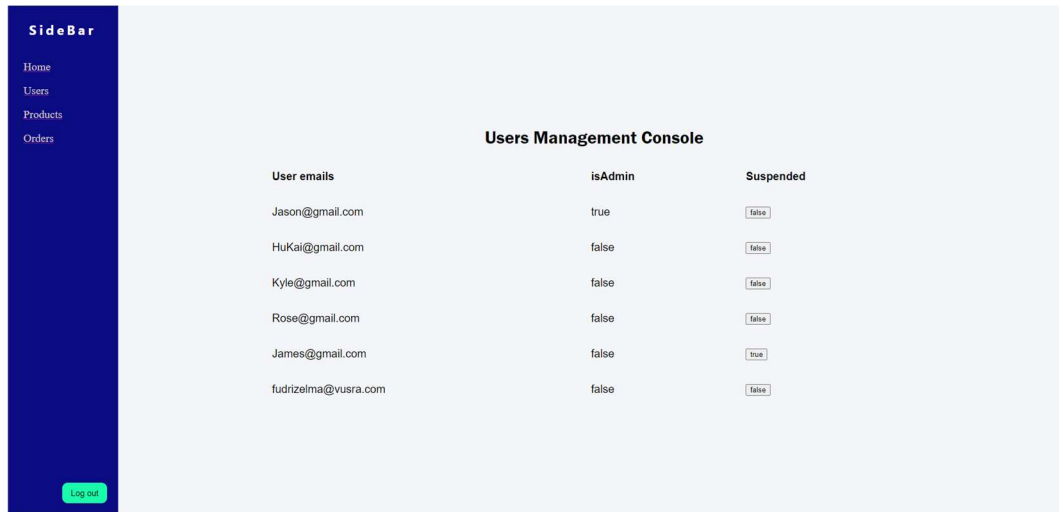


Figure 27. Admin user management page

On the admin/users page (Figure 27), the admin can view all users including himself. The users and information are displayed in table format. There are user emails, isAdmin and Suspended fields. The admin is able to toggle the suspended property between false and true. When a user's suspended property is true, the user will not be able to login to the application.

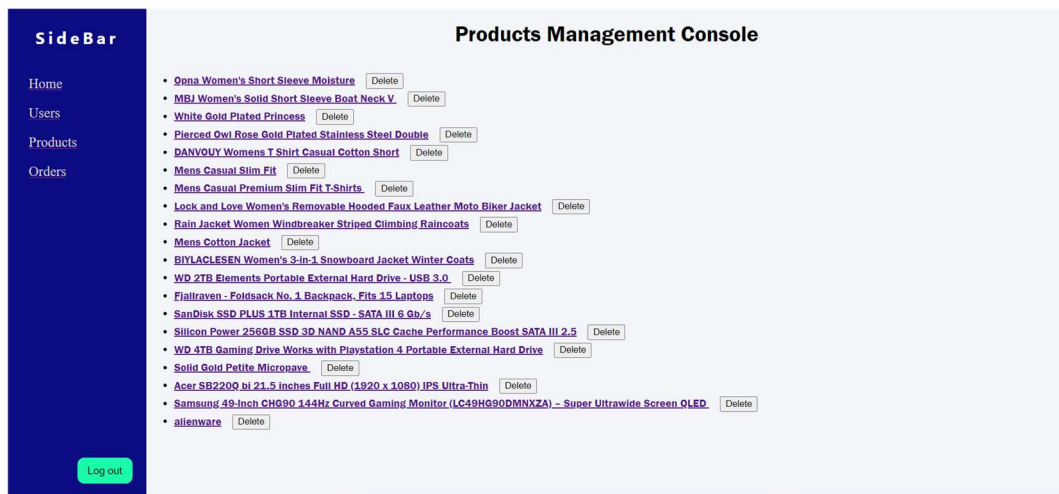


Figure 28. Admin products page

Unlike the Products view on the Home page of the application, the products management console displays only the product names and a delete button which will trigger product deletion. The admin can also click the product names to view the product details at admin/products/productDetail page.



Figure 29. Admin product detail page

The Admin Product Detail Page is very similar to the Product detail page except it does not provide the add to cart button.

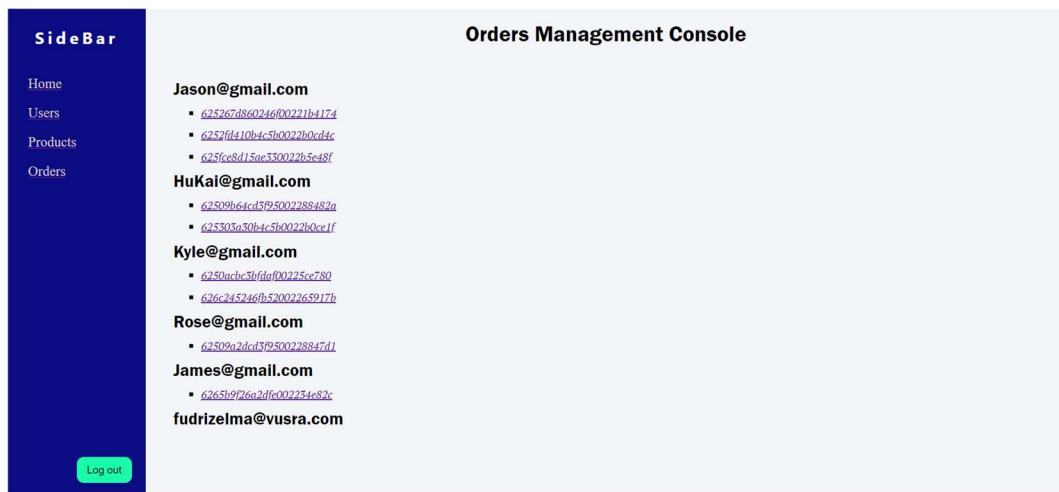


Figure 30. Admin orders page

The Admin orders page displays all the orders grouped with different users. The admin is able to view order details by clicking the order name which will direct to admin order detail page, which is the same as the order detail page in the user profile.

After the operations are complete, the admin can get back to the application main page by logging out in right bottom corner of the side bar.

5 IMPLEMENTATION

This section will describe how the functionalities and graphical user interfaces were implemented by code in detail. As it was mentioned in the Introduction section that this project combines two applications: backend server and frontend client. In each subsection, these topics will be discussed:

- Project Environment
- Directory Structure
- Application Source Code

5.1 Integrated Development Environment

There are many different IDEs in the market, for example, Eclipse, NetBeans, Visual Studio, WebStorm, and Atom. The one chosen for this project is Visual Studio Code, a source code editor (can also be classified as IDE) developed by Microsoft. It is a lightweight but powerful source code editor which runs on the desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes (Microsoft, 2022).

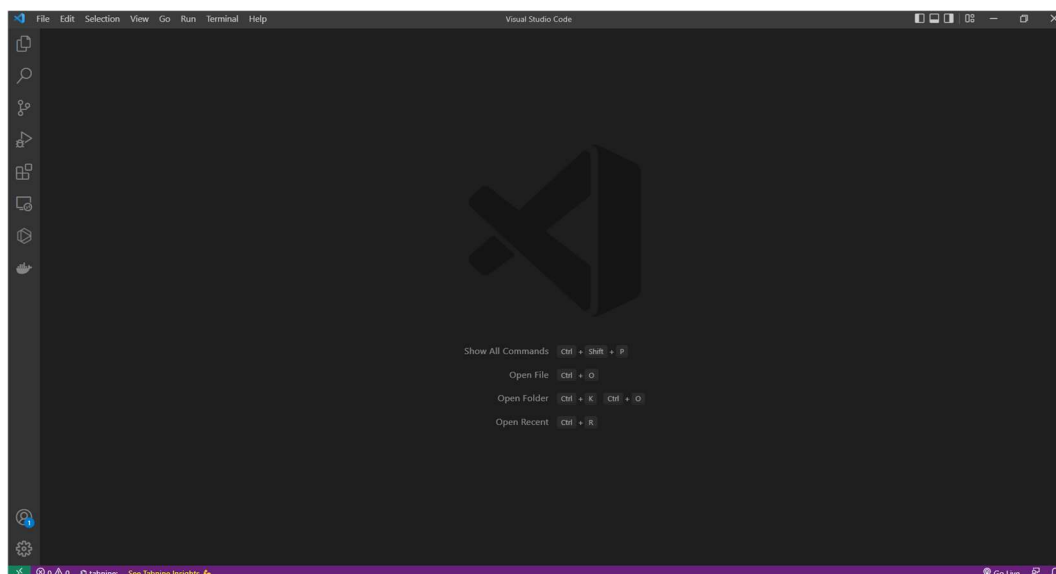


Figure 31. Visual Studio Code user interface

5.2 Backend Server

The Node backend server is developed to create REST API endpoints for frontend and execute project logics and then modify data in the database.

5.2.1 Project Environment

The Project environment is the basic setup of a project. It consists of different external dependencies, config file and environment variables. On the backend server, a Node Typescript environment is set up with yarn, which is a package manager similar to NPM. It creates a package.json file in the project root directory.

The package.json file configures dependencies, version, engines, license, running scripts, development dependencies and so on.

5.2.2 Directory Structure

In Figure 32 shown below, the first directory is *coverage*. It is a testing report automatically generated by Jest, which will be discussed later in the chapter. It basically shows how many percentages the test cases have covered the

application. The second one, *dist*, is the output directory for JavaScript files which are compiled TypeScript files. The third *node_modules* are installed packages modules. The *test* is for testing code which will be covered in the chapter on testing.

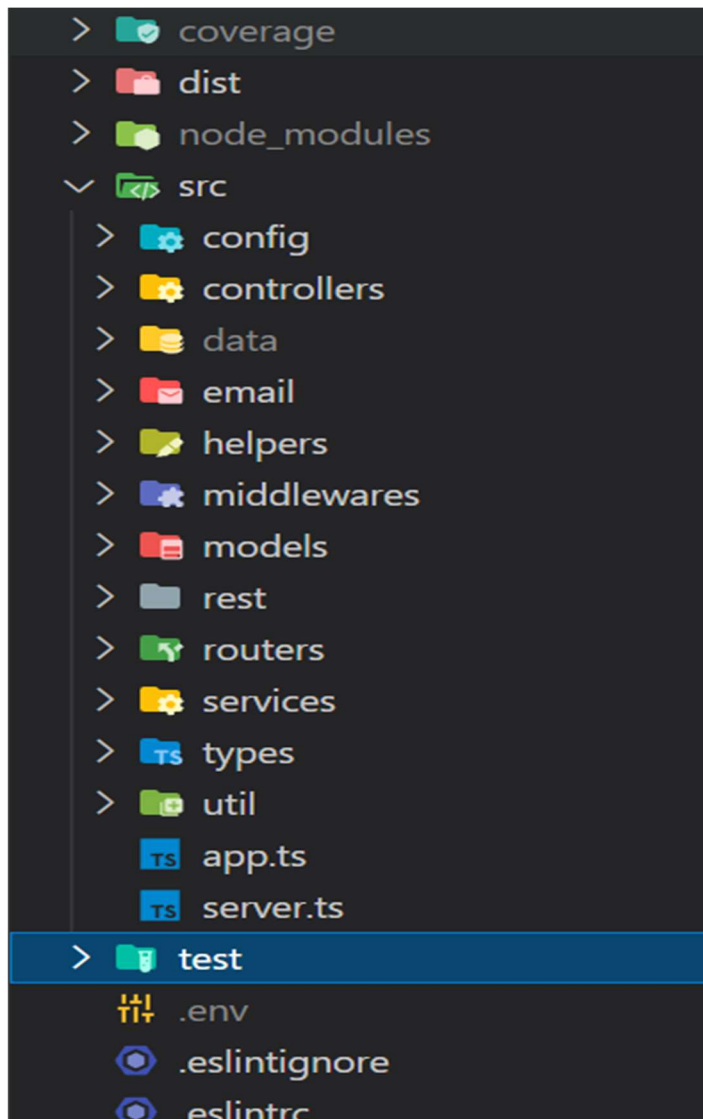


Figure 32. Directory Structure backend

The main directory is *src* folder, which contains *config*, *controllers*, *middlewares*, *models*, *routers*, *services*, *types* and *utils*. Each of them has corresponding files inside.

5.2.3 Application Source Code

In the root directory of the project, *aptotes* and *server.ts* are used as base files. In the *server.ts* MongoDB connection and server boot are handled. First *dotenv* (an environment variable loader) is used to load environment variables defined in *.env* file. Then *Mongoose* and *server* are running on the port defined as environment variables (see Code Snippet 4).

```
mongoose.connect(host, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useFindAndModify: false,
  useCreateIndex: true,
}).then(() => {
  app.listen(app.get('port'), () => {
    console.log(
      ' App is running in %s mode',
      app.get('port'),
      app.get('env')
    )
  })
})
```

Code Snippet 4. server.ts

The *app* function is imported from the *app.ts* module. In *app.ts*, we import built-in and 3rd-party middlewares: *express*, *dotenv*, *lusca*, *compression* and *cors*. Then *app* is defined using *express* and call *app.use* to allow *cors* access from all url

addresses. `JWT_SECRET` will also be checked. After that *app* will utilize all the 3rd-party middlewares and add route handlers and error handlers (Code Snippet 5).

```
if (!process.env.JWT_SECRET) {  
  console.log('FATAL ERROR: JWT_SECRET is not defined.')  
  process.exit(1)  
}
```

```
app.set('port', process.env.PORT || 3000)  
app.use(express.json())  
app.use('/api/products', productRouter)  
app.use('/api/user', userRouter)  
app.use('/api/orders', orderRouter)  
app.use('/api/auth', auth)
```

Code Snippet 5. `app.ts`

The route handlers used by *app* are defined in the *routers* directory as mentioned. In *auth.ts*, we import two controllers, `authenticate User` and `is Admin`, from the *controllers* directory

The *router* function is achieved with the `express Router()` method. Then we will call `router.post()` to define post method for each controller. Then in *order.ts*, we have similar methods and route handlings (Code Snippet 6).

```
import express from 'express'
```

```
import { authenticateUser, isAdmin } from
'../controllers/auth'

const router = express.Router()

router.post('/', authenticateUser)

router.post('/isAdmin', isAdmin)

export default router
```

Code Snippet 6. auth router

In *controllers* directory, it provides necessary controllers for the router handlers. First *auth.ts* controller has *authenticateUser* and *isAdmin* controllers (Code Snippet 7).

```
const { email, password } = req.body

const user = await User.findOne({ email: email })

const passwordIsValid = bcrypt.compareSync(password,
user.password)

const token = jwt.sign({ _id: user._id }, jwtKey)

res.status(200).json(token)
```

Code Snippet 7. auth controller

In *authenticateUser* method, we will get a request from user in request body. After checking if the user exists, a 3rd-part library *bcrypt* is used to compare the hashed password in the user database. If the password matches with the email and the user is not suspended by admin, a jwt token is sent back with a status 200, saying the user is logged in. In the *isAdmin* method, the logics are similar except we will check if the *isAdmin* property is true.

In the *order.ts* controller, we define several methods:

- getAllOrders
- getOneOrder
- findOrdersByCustomerEmail
- createOrder
- deleteOrder

All the methods follow the basic principles of REST api CRUD operations. One example of code is displayed in Code Snippet 8.

```
const newOrder = new Order({
  totalPrice,
  timestamp,
  customerEmail,
  purchasedItems,
  billingInfo,
})

const user = await User.findOne({ email: customerEmail })
user.cart = []
user.orders.push(newOrder._id)
await UserService.saveUser(user)
await OrderService.createOrder(newOrder)
```

Code Snippet 8. order controller createOrder method

The main logic is straightforward: Create, Read, Update, Delete (CRUD) operations. Notice that when creating an order, the cart items must be cleared so that the user will have an empty cart after making an order. There *UserService* and *OrderService* are used which are both defined in the *services* directory. In the

deleteOrder method, the *splice()* method is called to remove one from the index. *Splice()* is a native JavaScript array method. The *product* controller and *user* controller are similar to the above *order* controller.

In the *services* directory, there are three services which are created to manipulate application data and run some queries. The first file is *order.ts* which creates order service. It has six methods which are self-explanatory as the names suggest (see Code Snippet 9):

- *createOrder*
- *getAllOrders*
- *getOneOrder*
- *findOrdersByEmail*
- *findOrderById*
- *deleteOrderById*

```
const createOrder = async (order: OrderType) => {  
  return order.save()  
}
```



```

}

const getOneOrder = async (orderId: string):
Promise<OrderType> => {

  const foundOrder = await Order.findById(orderId)

  return foundOrder
}

const findOrdersByEmail = async (customerEmail: string)
=> {

  const foundOrders = await Order.find({ customerEmail:
customerEmail })

  return foundOrders
}

```

Code Snippet 9. order service

Those methods are called by controllers and the *Order* type and class are imported from *models* directory. The methods for queries are based on mongoose query building.

In *models* directory. We have created four models:

- CartItem
- Order
- Product
- User

Each model has different schemas defined using mongoose schema method because mongodb is a NoSQL database.

The first model *CartItem* is not shown in database. It is included as an embedded model in user model. We first declare the *CartItemType* with imageUrl as string,

productName as string, productId as a mongodb object id, price as number, quantity as number and ownerEmail as string. The the schema and the corresponding model are created as name *CartItem* (Code Snippet 10).

```
export type CartItemType = Document & {
  imageUrl: string
  productName: string
  productId: mongoose.Schema.Types.ObjectId
  price: number
  quantity: number
  ownerEmail: string
}
```

Code Snippet 10. CartItem model

Then we have *Order* model which has similar methods and definition like CartItem model. The *OrderType* is defined with *totalPrice* as number, *timestamp* as string, *customerEmail* as string, *purchasedItems* as array of *CartItemType* and *billingInfo* as an object which has its own types: *fullName*, *country*, *streetAddress*, *phoneNumber*, *postalCode*, *city* and *paymethodMethod* all as string. When we declare *purchasedItems*, we use a common document-oriented relationship pattern called **embedded document** (Tutorialspoint, n.d.). It has the CartItem as an embedded document in the Orders documents. All the properties in CartItem will also be displayed in Orders documents (Code Snippet 11).

```
export type OrderType = Document & {
  totalPrice: number
  timestamp: string
  customerEmail: string
  purchasedItems: CartItemType[]
}
```

```
billingInfo: {  
  fullName: string  
  country: string  
  streetAddress: string  
  phoneNumber: string  
  postalCode: string  
  city: string  
  paymentMethod: string  
}  
}
```

Code Snippet 11. Order model

5.3 Frontend Client

The TypeScript React frontend is developed to give a clear and easy-to-use User Interface which will render the content automatically and re-render it when states are changed. Same as backend IDE, Visual Studio Code is used for developing the frontend application.

5.3.1 Project Environment

Similar to the backend, in frontend NPM is also used. A *package.json* file is initialized (Code Snippet 12).

```
"name": "online-store-frontend",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {
```

```
    "@fortawesome/fontawesome-svg-core": "^1.2.36",  
    "@fortawesome/free-solid-svg-icons": "^5.15.4",  
  }  
}
```

Code Snippet 12. package.json frontend sample

5.3.2 Directory Structure

In the base directory, there are four main directories:

- build
- node_modules
- public
- src

The *build* directory contains compiled TypeScript files from the *src* folder, which are simply JavaScript files. The *node_modules*, same with backend, has dependencies installed for this project. The *public* directory includes a *index.html*, a *favicon.ico* and logo *png* file which are auto-generated files when we set up a React project. The main file in *public* is the *index.html* file.

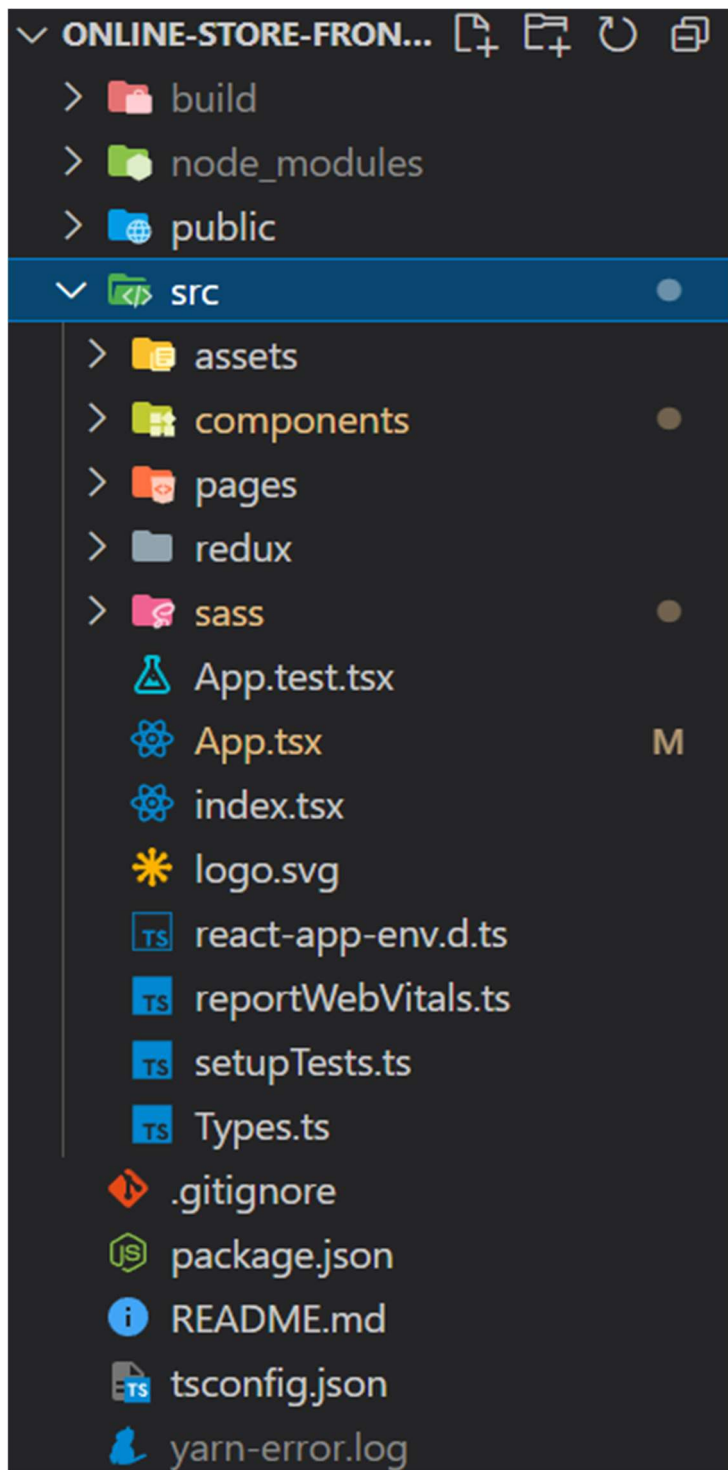


Figure 33. Directory Structure frontend

The directory that has our source code is the *src* directory. Inside *src* folder, we can see there are five embedded directories:

- assets
- components
- pages
- redux
- sass

The first *assets* directory has one image file which will be used as profile picture for every user in the application. The *components* directory holds component code like *home*, *profile*, *cart* etc. The *pages* directory contains code defining React Router pages. The *redux* directory contains source code handling Redux operations.

5.3.3 Application Source Code

The source code of this application is all included in *src* directory. In the root directory of *src*, two important files are *App.tsx* and *index.tsx*. The *index.tsx* file renders ReactDOM with **BrowserRouter** and **redux Provider** (Code Snippet 13).

```

ReactDOM.render (
  <React.StrictMode>
    <Provider store={store}>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);

```

Code Snippet 13. ReactDOM with Redux and Router setup in index.tsx

In *app.tsx* file, a main **Routes** are defined and inside the main Routes many *Routes* for this app are included (Code Snippet 14).

```

<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/profile" element={<ProfilePage />} />
  <Route path="/orderDetail" element={<OrderDetail />} />
  <Route path="/cart" element={<CartPage />} />
  <Route path="/detail" element={<ItemDetail />} />
  ...
</Routes>

```

Code Snippet 14. React Routes defined in app.tsx

In the *components* directory, we have a nested directory tree like this:

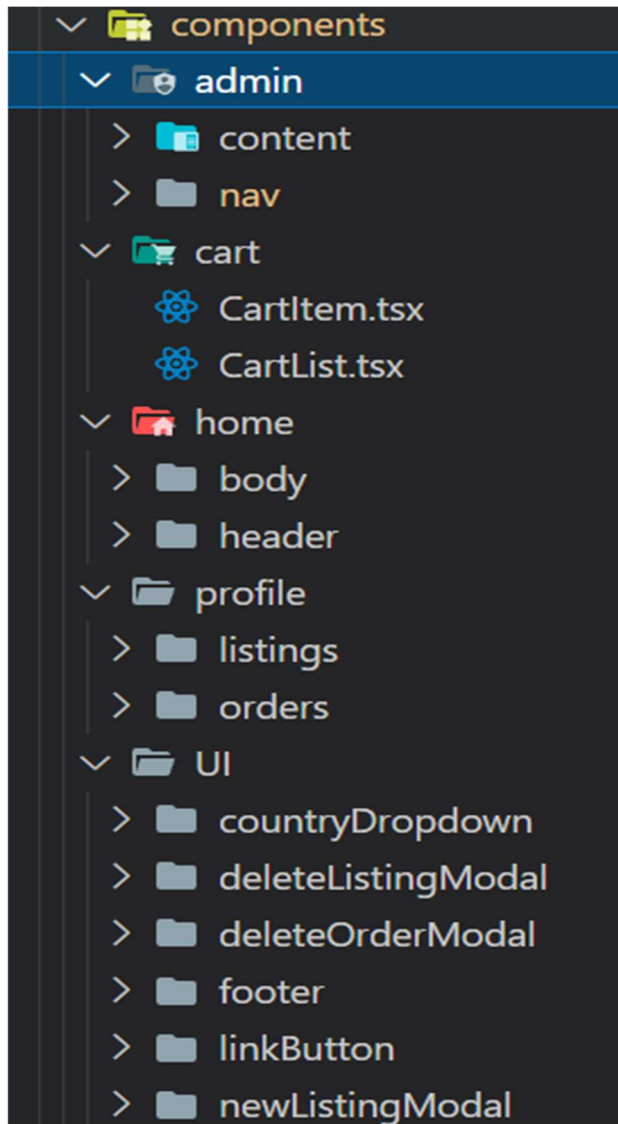


Figure 34. Components directory tree

And in each directory *admin*, *cart*, *home*, *profile*, the source code inside is related to the different components and pages.

The *pages* directory includes all the pages or **Routes** defined in *App.tsx* file. Each file in the *pages* has their own urls in the browser.

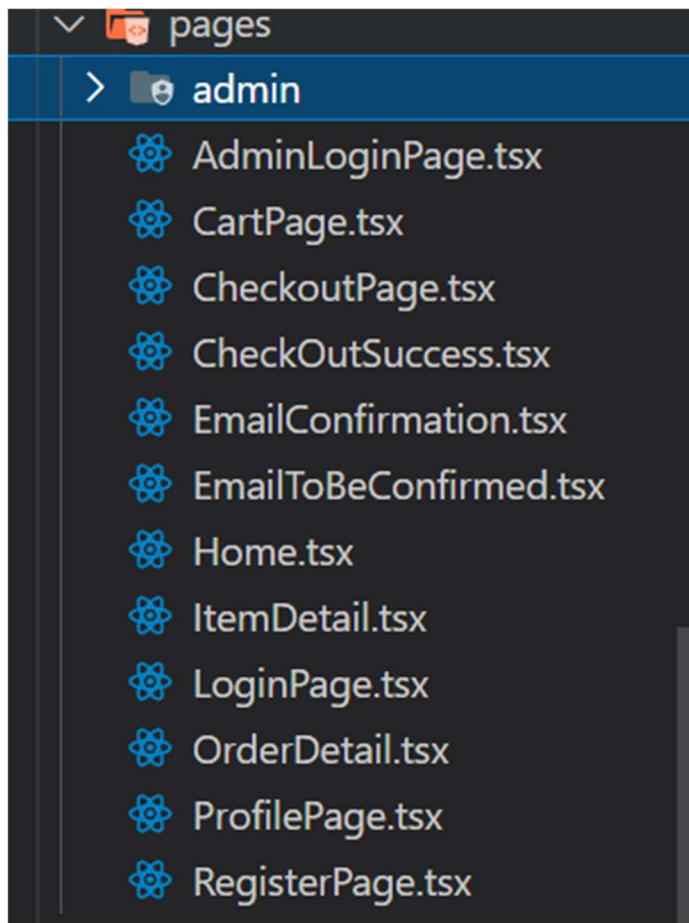


Figure 35. Pages directory tree

Inside *Home.tsx* file which defines the content in home page, I have defined different **states** like *items* for all product items, *isLoading* for checking if the page is waiting for response, *hasError* for checking error, *searchText* for text typed in search field, *cartItems* for items added to the cart and *cartItemQuantity*. The *isLoggedIn* value is given by redux root state. The *navigate* is used to navigating custom url and *dispatch* is used for dispatching redux action (Code Snippet 15).

```
const [items, setItems] = useState([]);  
const [isLoading, setIsLoading]=useState<boolean>(true);  
const [hasError, setHasError] = useState<boolean>(false);  
  
const [searchText, setSearchText] = useState<String>("");  
const [cartItems, setCartItems] = useState([]);  
  
const [cartItemsQuantity, setCartItemsQuantity] =  
useState(0);  
  
const isLoggedIn = useSelector((state: RootState) =>  
state.isLoggedIn);  
  
const navigate = useNavigate();  
const dispatch = useDispatch();
```

Code Snippet 15. The home.tsx function and state definitions

Then a **useEffect(callback, dependency)** hook is used to get products from backend. The **useEffect** hook is used to perform side effects in application and different usage of **useEffect** will correspond to different **lifecycle methods** in **class-based component**. It uses **axios** to fetch data and handle the Promise with **then** method and handle error with **catch** method. The related states are changed and finally we will clean up the operation after the component unmounts (Code Snippet 16).

```
useEffect(() => {  
  
    axios("https://fierce-spring-store-  
backend.herokuapp.com/api/products")  
  
    .then((res) => {  
  
        setHasError(false);  
  
        setIsLoading(false);  
  
        setItems(res.data);  
  
    })  
  
}, []);
```

Code Snippet 16. The home.tsx file first usage of useEffect

After the first **useEffect()** hook runs, a second `useEffect()` will run in parallel. The second `useEffect` check JWT in Cookies with three conditions (Code Snippet 17):

- No JWT stored in Cookies, the user is not logged in
- A JWT token longer than 149 characters stored in Cookies, it is the token from admin login, and it will be removed
- A JWT token with 149 characters long stored in Cookies, get the user information with the token

```
useEffect(() => {  
  const jwt = Cookies.get("jwt");  
  if (jwt) {  
    if (jwt.length > 149) {  
      Cookies.remove("jwt");  
    } else {  
      axios("https://fierce-spring-store-backend.herokuapp.com/api/user/me", {  
        headers: { "x-auth-token": jwt },  
      })  
    }  
  }  
}, [isLoggedIn, cartItems.length, dispatch]);
```

Code Snippet 17. The home.tsx file second usage of useEffect

And finally in the return statement, the Home component returns *Header*, *ItemList* and *Footer* component with related props (Code Snippet 18).

```
return (  
  <>  
    <div className="body-container">  
      <Header  
        setSearchText={setSearchText}  
        cartItemsQuantity={cartItemsQuantity}  
        setCartItemsQuantity={setCartItemsQuantity}  
      />  
      <ItemList  
        searchText={searchText}  
        items={items}  
        isLoading={isLoading}  
        hasError={hasError}  
        setCartItems={setCartItems}  
        cartItems={cartItems}  
      />  
    </div>  
    <Footer />  
  </>  
);
```

Code Snippet 18. The home.tsx file return statement

One distinct aspect in return statement is the angle bracket <> inside return statement which is named **React Fragment**. When multiple React components are returned in a component, React Fragment is needed (The alternative would be div

container). Fragment could help you group child components without extra nodes in the DOM. Angle bracket symbol is a short way of writing React Fragment. The long format is `<React.Fragment>` and `</React.Fragment>`.

6 APPLICATION TESTING

In this section, testing types and tools used for the application are introduced. When it comes with software testing, it always refers to automatic testing. Due to time limit and heavy workload for this thesis project, automatic testing was only done partly in the backend. For the frontend application, every component was evaluated manually to make sure everything is working correctly on the client side.

As for backend server testing, there are two testing types that are implemented: unit testing and integration testing. Unit testing tests a unit of an application without its external dependencies; integration testing tests the application with its external dependencies (Hamedani, n.d.). The testing tool used was Jest, which is a delightful JavaScript testing framework with a focus on simplicity (Jest, n.d.).

For testing product api endpoints, most of testings are integration testings because the product APIs have database server connection and need requests from client. For this reason, request was imported from the *supertest* library which mocks a client request and *MongoMemoryServer* from *mongodb-memory-server* to create a MongoDB database. These methods were used to create some data in database and test it automatically.

To write automatic tests, a test suite needed to be created with some setups and different test cases. The *describe* method creates a test suite. In the callback function, there were three methods to setup the database environment:

- *beforeeach* – runs before each test case
- *aftereach* – runs after each test case
- *afterAll* – runs after all test cases are complete

The first test case was to create a listing and product if all required parameters are satisfied. A user was created first and a product was created using defined function *createProduct()*. Then the http status code was expected to be 201, meaning

created and it also should have an id field. The name, genre and price were all statically defined in `createProduct()` function (Code Snippet 19).

```
it('should create a listing and product', async () => {
  await createUser()
  const res2 = await createProduct()
  expect(res2.status).toBe(201)
  expect(res2.body).toHaveProperty('_id')
  expect(res2.body.name).toBe('Asus VivoBook S16')
  expect(res2.body.genre).toBe('electronics')
  expect(res2.body.price).toBe(1600)
})
```

Code Snippet 19. Create a listing and product test case

In the second test case, we expected to get back an existing product. First, the process of creating a user was repeated as well as a product. Then the id field was extracted from the response body and the GET method was called on url `/api/products/productid`. The id from the **createProduct()** method is expected to be the same as the id from GET method (Code Snippet 20).


```

It('should get back an existing product', async () => {
  await createUser()
  let res = await createProduct()
  expect(res.status).toBe(201)
  const productId = res.body._id
  res = await
  request(app).get(`/api/products/${productId}`)
  expect(res.body._id).toEqual(productId)
})

```

Code Snippet 20. Get back an existing product test case

The last test case in this application is to not get back a non-existing product test case. This test is fairly straightforward and short. The GET method was called on the url with a non-existing product id and the response status code should be 404 not found (Code Snippet 21).

```

it('should not get back a non-existing product', async
() => {
  const res = await
  request(app).get(`/api/products/${nonExistingProductId}`)
  expect(res.status).toBe(404)
})

```

Code Snippet 21. Not get back a non-existing product

In the terminal, to run the test files in *test* directory, the following command was executed:

```
yarn test
```

which is defined as *test* script in the package.json file (Code Snippet 22).

```
"test": "jest --forceExit --detectOpenHandles --coverage  
--verbose false"
```

Code Snippet 22. Test script configuration in package.json file

Then in the terminal window, testing operations were conducted. After the test was finished, a test result was displayed (Figure 36) along with a detailed test report in *coverage* directory as a HTML file.



```
Test Suites: 2 passed, 2 total  
Tests:      6 passed, 6 total  
Snapshots:  0 total  
Time:       11.253 s  
Ran all test suites.
```

Figure 36. Jest test result in the terminal window

7 CONCLUSIONS

The initial goal of this thesis project was to build an online e-commerce second-hand store for users to purchase, sell their own items or place orders and for administrator to easily manage the application data. To achieve the goal and meet the requirements, a full-stack web application was developed with TypeScript, React, Node, Express and MongoDB. After the development and testing phases, the project was then successfully deployed to Heroku for both frontend client and backend server for production.

During the development phase, numerous challenges and failures were encountered. For instance, some commas or semicolons missing in JSON file in the backend API testing took two days to understand what the problem is. Similarly, environment variables undefined in production phase also greatly delayed the deployment process. One of the greatest challenges during the development was to setup an email confirmation service in backend using Nodemailer. To integrate Nodemailer with the server, it requires many configurations such as credentials, transport, email template etc. In addition, the Gmail sender account must be configured as *less secure apps access* for Nodemailer to access the Gmail account.

In summary, the project meets its requirements and the initial goal. It was completed and accepted by the supervisor after some improvement on functionalities of the application.

This project still has much room for improvement although it achieves the initial goal. For instance, an online payment solution such as Stripe or PayPal API could be added to manage the online payment. The operations of item quantity in stock can also be implemented so that the actual inventory stock operations can be achieved. In addition, user information can be more complete with common shipping address and profile picture. In addition, the project is currently using HTTP as the internet protocol for data exchange because it can be more easily

implemented than HTTPS which is a more secure version of HTTP. In the future improvement, HTTP could be replaced by HTTPS for better security.

REFERENCE

Express, 2017. *Express official site.* [Online]
Available at: <https://expressjs.com/>
[Accessed 25 March 2022].

Hamedani, M., n.d. *Udemy.* [Online]
Available at: <https://www.udemy.com/course/nodejs-master-class/>
[Accessed 8 May 2022].

InterviewBit, 2021. *InterviewBit.* [Online]
Available at: <https://www.interviewbit.com/blog/backend-developer-skills/>
[Accessed 24 March 2022].

Jest, n.d. *Jest Official Docs.* [Online]
Available at: <https://jestjs.io/>
[Accessed 8 May 2022].

Microsoft, 2022. *Visual Studio Code official docs.* [Online]
Available at: <https://code.visualstudio.com/docs>
[Accessed 2 May 2022].

MongDB, n.d. *MongoDB Documentation.* [Online]
Available at: https://www.mongodb.com/docs/atlas/?_ga=2.32041213.2145040344.1648165715-579287739.1648165715
[Accessed 25 March 2022].

Redux, 2022. *Redux.* [Online]
Available at: <https://redux.js.org/tutorials/essentials/part-1-overview-concepts>
[Accessed 24 March 2022].

SASS, n.d. *SASS official document.* [Online]
Available at: <https://sass-lang.com/documentation>
[Accessed 24 March 2022].

Singh, A., 2019. *Medium.* [Online]
Available at: <https://medium.com/@aj.ankitsingh/stairways-to-become-a-full-stack-developer-in-2019-189f954fff16>
[Accessed 24 March 2022].

Sons, J. W. &, 2012. *Professional Node.js: Building JavaScript Based Scalable Software.* 1st edition ed. s.l.:Wrox.

Tutorialspoint, n.d. *tutorialspoint mongodb overview.* [Online]
Available at: https://www.tutorialspoint.com/mongodb/mongodb_overview.htm
[Accessed 25 March 2022].

Tutorialspoint, n.d. *tutorialspoint mongodb relationships.* [Online]
Available at: https://www.tutorialspoint.com/mongodb/mongodb_relationships.htm
[Accessed 3 May 2022].

W3Schools, n.d. *W3Schools.* [Online]
Available at: https://www.w3schools.com/REACT/react_components.asp
[Accessed 24 March 2022].

Wikipedia, 2022. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Front-end_web_development

Wikipedia, 2022. *Wikipedia.* [Online]
Available at: <https://en.wikipedia.org/wiki/TypeScript>
[Accessed 24 March 2022].