

Bach Le

# DEVELOPING AN E-COMMERCE WEBSITE WITH REACT

Bachelor's thesis

Information Technology

Bachelor of Engineering

2022



South-Eastern Finland  
University of Applied Sciences

Degree title	<u>Bachelor of Engineering</u>
Author(s)	Bach Le
Thesis title	Developing an e-commerce website with React
Commissioned by	-
Year	2022
Pages	48 pages, 1 appendix page
Supervisor(s)	Timo Mynttinen

## ABSTRACT

The thesis goal was to create a website allowing users to trade products online.

The application's target user is everyone who has internet access. The theoretical background provides React.js, Node.js, MongoDB, JavaScript, Express.js, and REST API definitions and functions. The theories from the theoretical background part will provide solid knowledge for the implementation part.

The thesis's implementation part is about planning and coding for the website, including server-side and client-side implementation. Also, the website will be demonstrated by images in this part.

The result of this thesis is a functional website that allows users to buy/ trade products online.

**Keywords:** web development, full-stack, MERN Stack, React

# CONTENTS

1	INTRODUCTION .....	5
2	THEORETICAL BACKGROUND .....	6
2.1	E-commerce .....	6
2.1.1	Definition.....	6
2.1.2	Types.....	6
2.1.3	Advantages.....	7
2.1.4	Challenges.....	7
2.2	JavaScript.....	8
2.3	REACT .....	9
2.3.1	Virtual DOM .....	9
2.3.2	Components and Props .....	10
2.3.3	JSX.....	11
2.3.4	State .....	11
2.4	NODE.JS .....	12
2.5	EXPRESS.JS.....	12
2.6	MONGODB.....	13
2.7	REST API .....	13
3	IMPLEMENTATION.....	15
3.1	Application Overview .....	15
3.2	Back-end implementation .....	16
3.2.1	API Planning.....	16
3.2.2	Server implementation.....	17
3.3	Web application implementation .....	27
3.3.1	Structure .....	27
3.3.2	Styling.....	28

3.3.3	Screens and Components .....	29
3.3.4	Redux .....	32
3.4	My website application.....	35
3.4.1	Home Screen.....	35
3.4.2	Single Product Screen .....	37
3.4.3	Cart Screen.....	38
3.4.4	Shipping Screen .....	39
3.4.5	Payment Screen .....	40
3.4.6	Place Order Screen .....	40
3.4.7	Order Screen .....	41
3.4.8	Register Screen .....	42
3.4.9	Log In Screen .....	43
3.4.10	Profile Screen .....	45
4	CONCLUSION.....	46
	REFERENCES .....	47

## 1 INTRODUCTION

Nowadays, with the fast-paced development of technologies, online shopping becomes a must for companies to provide/ sell their products to customers. More and more companies have their own websites and many online platforms that sell products online and appear to meet customers' needs. E-commerce plays a very important role in the rise of those websites.

E-commerce is a model that allows companies and customers to trade things online. It first appeared in the 1960s when companies used an electronic system called the Electronic Data Interchange to facilitate the transfer of documents. Later, with the introduction of the internet, more and more e-commerce sites started showing up: Amazon, eBay, Etsy, etc. The convenience of online shopping, and the development of transporting and shipping systems increase the demand for e-commerce quickly.

The project that has been made for this thesis is an E-commerce website. The site basically has 2 roles: admin and users/customers. Admin manages the products by adding new products, deleting products, and making changes to product information. Users can view and search for the product that they want, also they can buy them by adding the products to the site's shopping cart.

In order to achieve the goals, I have divided my thesis into 4 chapters, described as follows:

- **Chapter 1 (Introduction):** basic information about the thesis topic, also what I will do in this thesis.
- **Chapter 2 (Theoretical Background):** theories that are needed to complete the thesis project.
- **Chapter 3 (Implementation):** the process of developing the thesis project.
- **Chapter 4 (Conclusion):** summary of the results and outcome of the thesis topic.

## 2 THEORETICAL BACKGROUND

### 2.1 E-commerce

#### 2.1.1 Definition

E-commerce, EC for short (electronic commerce) is a process referring to transactions, purchases, and goods being sold over the internet. Customers come to the website or online marketplace and purchase products using electronic payments. Upon receiving the payment, the merchant ships the goods or provides the service.

E-commerce is closely intertwined with the history of the internet. The first e-commerce site was Book Stacks Unlimited, an online bookstore created by Charles M. Stack in 1992, just one year after the internet was introduced (1991).

After years of development, as mobile devices became popular, social media increasingly affirmed the power and the boom of e-commerce.

#### 2.1.2 Types

Currently, there are many forms of e-commerce, including the following basic forms:

**B2B (Business to Business):** When a business sells a good or service to another business (e.g. A business sells software-as-a-service for other businesses to use).

**B2C (Business to Consumer):** When a business sells a good or service to an individual consumer (e.g. You buy a pair of shoes from an online retailer).

**C2B (Consumer to Business):** When a consumer sells their products or services to a business or organization (e.g. An influencer offers exposure to their online audience in exchange for a fee, or a photographer licenses their photo for a business to use).

**C2C (Consumer to Consumer):** When a consumer sells a good or service to another consumer (e.g. You sell your old furniture on eBay to another consumer).

### 2.1.3 Advantages

**Availability:** E-commerce sites are open 24/7 and 365 days (except for outages and maintenance), enabling customers to come and shop at any time. For businesses, it's a great opportunity to increase sales opportunities all the time.

**International sales:** Compared to opening a physical store where you will only be able to have access to the customers that live near/ in your area, an e-commerce store can sell to anyone in the world and isn't limited by physical geography. That will help reach out to more customers, and increase sales.

**Budget savings:** Compared with traditional forms of commercial business, all costs when e-commerce business are reduced: the cost of renting booths, salespeople, and management is much more economical. Naturally, when sellers save operating costs, they can offer more incentives and better discounts for their customers.

**Increased selection:** Many stores offer a wider array of products online than they carry in their brick-and-mortar counterparts. Many stores that solely exist online may offer consumers exclusive inventory that is unavailable elsewhere.

**Fast-accessing:** Instead of waiting for lines of people to buy/pay for the goods at physical stores, we just need to go on the website, put the goods that we want into the cart, pay for them and wait until the goods are delivered to us. That would be really convenient.

### 2.1.4 Challenges

**Internet access required:** To be able to buy and sell on e-commerce sites, you need devices that are connected to the internet. For now, there are more and

more places that have internet connections, but still, there are places that haven't got the internet yet.

**Trusting issues:** Products and services that cannot be seen, touched, held, or felt directly before we spend money to buy them. There's always a high chance that customers might be scammed by the "sellers".

**Limited customer service:** If customers have a question or issue in a physical store, they can ask a clerk, cashier, or store manager for help. In an e-commerce store, customer service can be limited: The site may only provide support during certain hours, and its online service options may be difficult to navigate or not answer a specific question.

**Security:** Skilled hackers can create authentic-looking websites that claim to sell well-known products. Instead, the site sends customers fake or imitation versions of those products -- or simply steals credit card information.

## 2.2 JavaScript

**JavaScript** is a scripting or programming language that allows you to implement complex features on web pages which HTML and CSS cannot do. Any time we clicked on the drop-down menu, write something on the site, or the transitions of the elements on the site, we are seeing the effects of JavaScript. It is used for web development, web applications, game development, and lots more.

In Web development, JavaScript is implemented on 2 sides of a website: client-side and server-side, following its features:

- **Client-side JavaScript:** JavaScript is developed to enable the enhancement and manipulation of web pages and client browsers. In a browser environment, your code will have access to things provided only by the browser, like the document object for the current page, the window, functions like an alert that pops up a message, etc. The main tasks of client-side JavaScript are validating input, animation, manipulating UI



elements, and append lying styles, some calculations are done when you don't want the page to refresh so often.

- **Server-side JavaScript:** JavaScript is developed to enable back-end access to databases, file systems, and servers. Server-side JavaScript, is JavaScript code running over a server's local resources. Also, with server-side code, you can still send JavaScript to the client-side.

## 2.3 REACT

**React**, also known as ReactJS or React.js is a JavaScript library focused on creating declarative user interfaces (UIs) using a component-based concept. It's used for handling the view layer and can be used for web and mobile apps. React's main goal is to be extensive, fast, declarative, flexible, and simple.

### 2.3.1 Virtual DOM

**Document Object Model**, or DOM, is an Application Programming Interface (API), it represents the UI of your application. Every time there is a change in the state of your application UI, the DOM gets updated to represent that change.

A virtual DOM is a representation of a real DOM that is built/manipulated by browsers. React generate a tree of elements in memory equivalent to the real DOM, which forms the virtual DOM in a declarative way. Because the virtual DOM will never be presented to the user, it will only exist in memory, rendering it is quicker.

In React, every UI piece is a component, and each component has a state. As you can see from Figure 1, the red circles represent the nodes or states that have changed. When the state of a component changes, React updates the virtual DOM tree. Once the virtual DOM has been updated, React then compares the current version of the virtual DOM with the previous version of the virtual DOM.

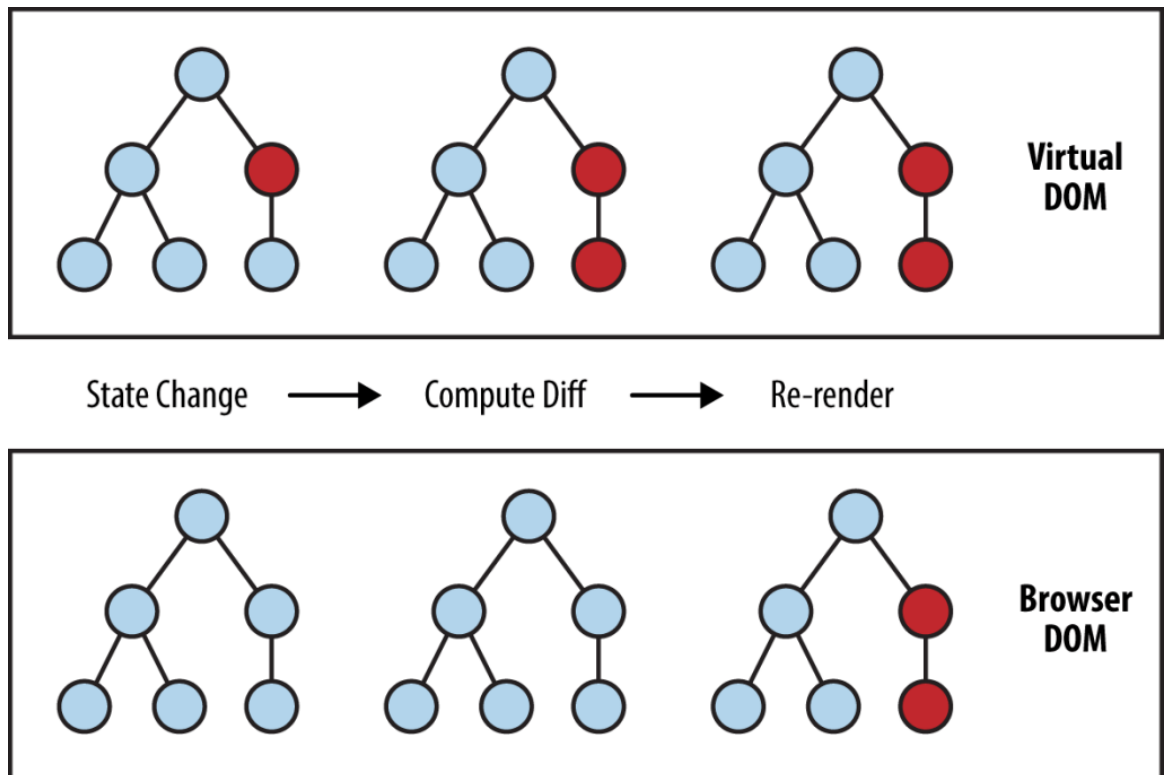


Figure 1. DOM process

Once React knows which virtual DOM objects have changed, then React updates only those objects, in the real DOM. This makes the performance far better when compared to manipulating the real DOM directly.

### 2.3.2 Components and Props

**ReactJS** is a component-based library where components make our code reusable and split our UI into different pieces. Components are divided into two types: Class components and Function components.

- **Function components:** they are basically JavaScript functions that may or may not receive data as parameters. They only contain a render method and don't have their own state. We can create a function that takes props(properties) as input and returns what should be rendered.
- **Class components:** they are more complex than functional components. It requires you to extend from React component and create a render function which returns a React element. You can pass data from one class

to other class components. You can create a class by defining a class that extends `Component` and has a `render` function.

We use props in React to pass data from one component to another (from a parent component to a child component).

### 2.3.3 JSX

React uses a syntax extension to JavaScript called JSX. JSX stands for JavaScript XML. JSX uses Babel preprocessors to convert HTML-like text in JavaScript files into JavaScript objects to be parsed. The layout of React components is mostly written using JSX. JSX returned by React components is compiled into JavaScript.

### 2.3.4 State

**State** is a built-in React object that is used to contain data or information about the component. A component's state can change over time, whenever it changes, the component re-renders. The change in state can happen as a response to events or actions that occurred and these changes determine the behavior of the component and how it will render.

- A state can be modified based on user action or network changes.
- Every time the state of an object changes, React re-renders the component to the browser.
- The state object is initialized in the constructor.
- The state object can store multiple properties.

State can be updated in response to event handlers, server responses, or prop changes. This is done using the `setState()` method.

## 2.4 NODE.JS

**Node.js** is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.

Some basic features of Node.js:

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, non-blocking. Basically, a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **Single-Threaded but Highly Scalable** – Node.js uses a single-threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
- **No Buffering** – Node.js applications never buffer any data. These applications simply output the data in chunks.
- **License** – Node.js is released under the MIT license

**Node.js** has a unique advantage since millions of front-end developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without learning another language.

## 2.5 EXPRESS.JS

**Express.js** is an open-source web application framework for Node.js. It is used for designing and building web applications quickly and easily. Since Express.js only requires JavaScript, it becomes easier for programmers and developers to build web applications and API without any effort.

Express.js is a framework of Node.js which means that most of the code is already written for programmers to work with. You can build a single-page, multi-page or hybrid web application using Express.js. Express is lightweight and helps to organize web applications on the server-side into a more MVC architecture.

## 2.6 MONGODB

**MongoDB** is a document-oriented NoSQL database used for high-volume data storage. Instead of using tables and rows like traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and functions which is the equivalent of relational database tables.

MongoDB features:

- **Schema-less Database:** one collection can hold different types of documents. These documents may consist of the different numbers of fields, content, and size.
- **Data model:** allows you to represent hierarchical relationships, store arrays, and other more complex structures more easily.
- **Scalability:** MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database.
- **Indexing:** every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool of the data.
- **Replication:** MongoDB creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data can be retrieved from another server.

## 2.7 REST API

**Representational State Transfer (REST)** is an architectural style that defines a set of constraints to be used for creating web services.

**API** stands for application programming interface, which is a set of definitions and protocols for building and integrating application software.

**REST API** is used to fetch or give some information from a web service. All communication done via REST API uses only HTTP requests.

How **REST API** works: as you can see from Figure 2, a request is sent from client to server in the form of a web URL as a HTTP request. After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON.



Figure 2. API

In HTTP there are five methods that are commonly used in a REST-based Architecture i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively. Other methods are less frequently used like OPTIONS and HEAD.

**GET:** The HTTP GET method is used to read (or get) the data of a resource. If nothing went wrong, it returns the data you need and an HTTP response code of 200 (OK). If there are any errors, it returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

**POST:** The POST method is used to create new resources. In a successful operation, it returns a Location header with a link to the resource that you have created and a 201 HTTP status.

**PUT:** It is used for updating the resources. PUT can also be used to create a resource in case where the resource ID have not existed in the database in the

first place. If successfully updated, return 200 (or 204 if not returning any content in the body). If using PUT for create, return HTTP status 201 on successful operation.

**PATCH:** It is used to modify the resources. The PATCH request contains the changes of the resource, not the complete resource. This basically the same as PUT, but the body contains a set of instructions describing how a resource should be modified.

**DELETE:** It is used to delete a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body.

### 3 IMPLEMENTATION

This section will be divided into three parts: application overview, back-end implementation, and front-end implementation. The first part describes all the features and functions of the application. The second part is about how did I implement the back-end server and how it was planned. The third part will be about the implementation of my web application.

#### 3.1 Application Overview

My application contains two basic roles, including user and admin. Admins, they can manage all the products on the site, for example create, update or delete products. Users can search, select and order products, or they can even manage all the orders that they have purchased before.

My application is called **Phone Shop**, basically it's a phone shop that sells phones. The main functions are the following:

- **Sign up and log in:** to have better experience with the website, the site requires all the users to register an account using their e-mails. Besides, users can also see and update their profiles.

- **Shopping cart:** whenever users want to buy something, they can add products to the shopping cart, and users can come back to check (or buy) the products later.
- **Search:** users can search for the products they want to buy by using a search box provided by the website.
- **Buy and pay:** The site supports a variety of payment options for the users when they purchasing the products.

## 3.2 Back-end implementation

### 3.2.1 API Planning

First of all, to create back-end server for the application, we need to plan API. API is used for communication between an application and a back-end server. So if we want to get data from back-end to our front-end, we need API to call the data that we need. The description of each API is in the table below.

Table 1. API paths

Path	Method	Input	Description
/users/login	POST	email, password	To sign in user
/users/	POST	name, email, password	To register a new user
/users/profile	GET	-	To get profile of current user
/users/:id	GET	id	Get data of user by id
/users/profile	PUT	name, email, password	To update current user
/products/	GET		Get all products
/products/:id	GET	id	Get product by id
/products/	POST	Name, price, description, image, countInStock	Add new product



/products/:id/review	POST	id, rating, comment	Add product review by id
/products/:id	DELETE	id	Delete product by id
/products/:id	PUT	:id, name, price, description, image, countInStock	Update product by id
/orders/all	GET	-	Get all orders
/orders/	GET	-	Get orders from logged in user
/orders/	POST	orderItems, shippingAddress, paymentMethod, itemsPrice, taxPrice, shippingPrice, totalPrice	Add order
/orders/:id	GET	id	Get order by id
/orders/:id/pay	PUT	id	Pay order by id
/orders/:id/delivered	PUT	id	Deliver order by id

### 3.2.2 Server implementation

To implement the server, first, we need to create a database to store all the data. I chose MongoDB as the database to work on this project. The database has three collections, which are: orders, products, and users. Figure 3 shows all collections I used for this project.

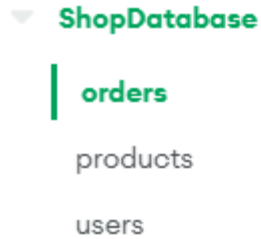


Figure 3. MongoDB Database

- Users: stores all data of users, including username, email, and password.
- Products: stores all the details of all products in the e-commerce store.
- Orders: information about orders that users have ordered.

In this project, I will connect to MongoDB by using its cloud server which is MongoDB Atlas. There are three ways to connect the database to the server: connect with MongoDB Shell, connect to your application, and connect using MongoDB Compass. I will connect using “connect to my application” way which requires a connection string that is provided by MongoDB.

```
console.log(process.env.MONGODB_URI)

const connectDatabase = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI, {
      useUnifiedTopology: true,
      useNewUrlParser: true,
    })
    console.log(`MongoDB Connected`)
  } catch (error) {
    console.error(`Error: ${error.message}`)
    process.exit(1)
  }
}
```

Figure 4. Connect to MongoDB

Figure 4 shows the way I connect to MongoDB, using the string provided by MongoDB: MONGODB\_URI, and connect to the database using mongoose.

Then, I defined the schema for this project. A schema is a JSON object that defines the structure and contents of your collection. Each field of a schema will be the collection property. Schemas represent types of data rather than specific values. There are many different types to be defined, for example type, required, string, number, etc. Figure 5 shows the schema used in this project:

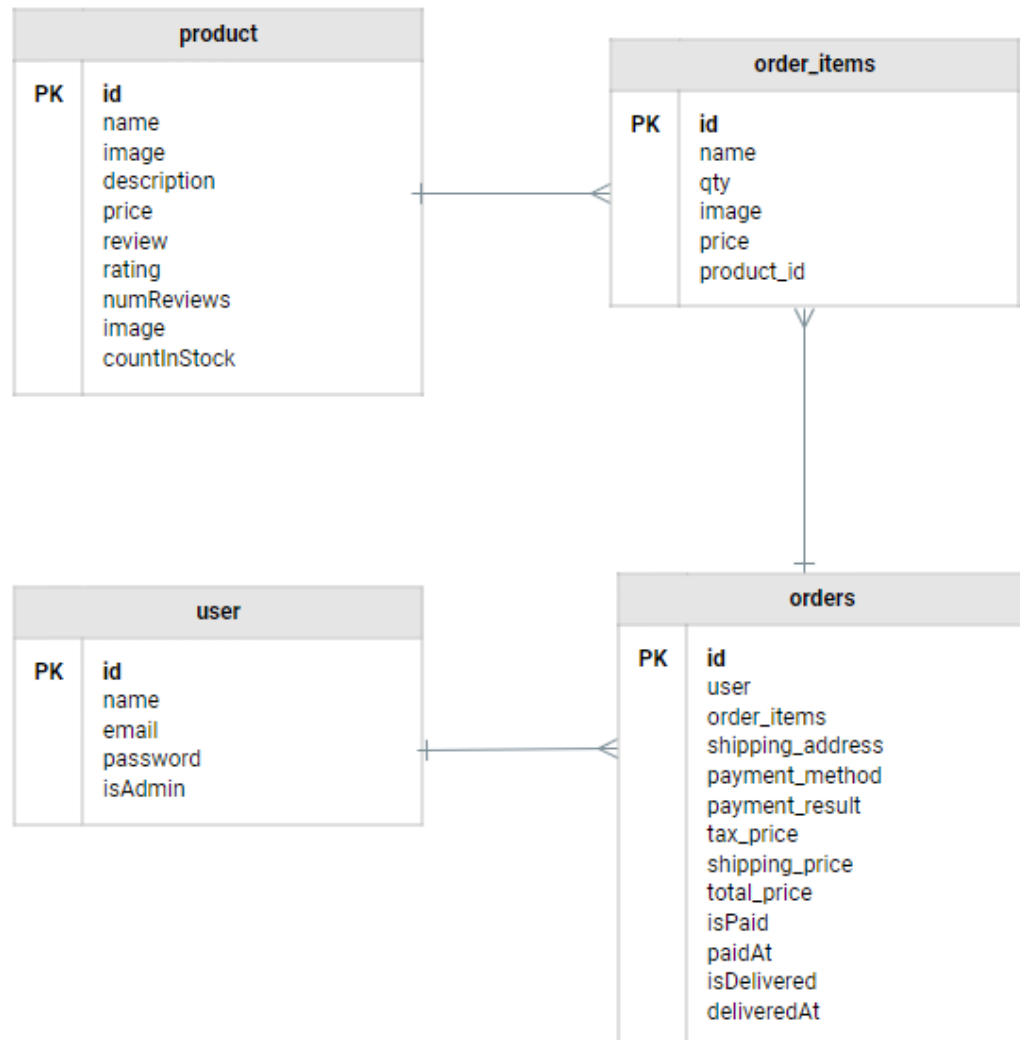


Figure 5. Schemas for this project

To use Schema in the collection, we have to convert it into Model. Figure 6 displays how I convert the *userSchema* to *User* model.

```
const User = mongoose.model("User", userSchema)
```

Figure 6. Database Model

After defining all the models for each collection, I created routes for each request from the table that I have made above (Table 1). Each API path will be a route for the corresponding request. In this project, I only use 4 basic API methods: GET, POST, PUT and DELETE. Based on each request's requirements, I create all the routes by using prefixes and functions.

```
orderRouter.get(
  "/all",
  protect,
  asyncHandler(async (req, res) => {
    const orders = await Order.find({})
      .sort({ _id: -1 })
      .populate("user", "id name email")
    res.json(orders)
  })
)
```

Figure 7. GET all orders method

Figure 7 shows how to create a route using GET method to get the list of all orders. *orderRouter* is the router of Order model, *orderRouter.get()* is to create a GET method. */all* is the prefix of this route. This API path is to get all the orders from the *Order* model by using *find()* to get all the available data. I also used *sort* and *populate* the result with user's information. If the operation is success, the API path will return the data and a successful status. If there are any errors while trying to get the data, a response with an error message will be sent.

Figure 8 displays how to create a route using GET method requires input to get the data. As you can see from Table 1: */users/:id*, */products/:id*, in these paths, there's *":id"* after the prefixes. This *":id"* thing will be the keyword for the route to get the information that the user/admin needs.

```

userRouter.get("/:id", asyncHandler(async (req, res) => {
  const product = await User.findById(req.params.id)
  if (product) {
    res.json(product)
  } else {
    res.status(404)
    throw new Error("User not Found")
  }
}))

```

Figure 8. GET User by id

In API, in order to create data to the database, we use POST method. Figure 9 displays how to create a new product. First, add all the needed information about the product on the request body, then the data will be sent to the database using the POST method. If the API call is success, it will return the product's info as a response. If there are any products that have the same name, it will return "product is already exists". If there are any errors, it will send an error message: "Invalid product data".

```

productRouter.post("/", protect, asyncHandler(async (req, res) => {
  const { name, price, description, image, countInStock } = req.body
  const productExist = await Product.findOne({ name })

  if (productExist) {
    res.status(400)
    throw new Error("Product name already exist")
  } else {
    const product = new Product({
      name,
      price,
      description,
      image,
      countInStock,
      user: req.user._id,
    })
    if (product) {
      const createdproduct = await product.save()
      res.status(201).json(createdproduct)
    } else {
      res.status(400)
      throw new Error("Invalid product data")
    }
  }
}))
)

```

Figure 9. POST method

If you can create, it means that you can also delete. For example, in this DELETE method on product route as you can see from Figure 10, I will input the id of the product that I want to delete, then use function *findById* to find the exact product, and finally remove it from the database.

```
productRouter.delete("/:id", protect, asyncHandler(async (req, res) => {
  const product = await Product.findById(req.params.id)

  if (product) {
    await product.remove()
    res.json({ message: "Product deleted" })
  } else {
    res.status(404)
    throw new Error("Product not Found")
  }
}))
)
```

Figure 10. DELETE method

To update the data in API, we use PUT method. Figure 11 displays how to update product's data using provided id. Input the information that you want to change and also the id of the product. Then save the product using *save()* function. If anything wrong, it will send you an error message.

```
productRouter.put("/:id", protect, asyncHandler(async (req, res) => {
  const { name, price, description, image, countInStock } = req.body
  const product = await Product.findById(req.params.id)

  if (product) {
    product.name = name || product.name
    product.price = price || product.price
    product.description = description || product.description
    product.image = image || product.image
    product.countInStock = countInStock || product.countInStock

    const updatedProduct = await product.save()
    res.json(updatedProduct)
  } else {
    res.status(404)
    throw new Error("Product not found")
  }
}))
)
```

Figure 11. PUT method

I created a route to search for the products, Figure 12 displays how. Here, I combined the search function with pagination. For this route, we will get the keyword from the API params and use *find()* to find all the products that have the exact keyword.

```
productRouter.get(
  "/",
  asyncHandler(async (req, res) => {
    const pageSize = 12;
    const page = Number(req.query.pageNumber) || 1;
    const keyword = req.query.keyword
      ? {
        name: {
          $regex: req.query.keyword,
          $options: "i",
        },
      }
      : {};
    const count = await Product.countDocuments({ ...keyword });
    const products = await Product.find({ ...keyword })
      .limit(pageSize)
      .skip(pageSize * (page - 1))
      .sort({ _id: -1 });
    res.json({ products, page, pages: Math.ceil(count / pageSize) });
  })
);
```

Figure 12. Search route

Next part covers how to register and login users. Register user basically is creating a new user, which means we need to post a new user into Users collection. So we do the same way as what we did with create a new product, using POST method. Figure 13 shows how I created a new user.

```
userRouter.post("/", asyncHandler(async (req, res) => {
  const { name, email, password } = req.body

  const userExists = await User.findOne({ email })

  if (userExists) {
    res.status(400)
    throw new Error("User already exists")
  }

  const user = await User.create({
    name,
    email,
    password,
  })

  if (user) {
    res.status(201).json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      token: generateToken(user._id),
    })
  } else {
    res.status(400)
    throw new Error("Invalid User Data")
  }
})
})
```

Figure 13. Create new User

But with only posting the email and password, it is not good for security of the accounts. So, to secure the application, we will need a token whenever a request that was made by users/admins is done. In this project, I use JSON Web Token (jwt) for authentication. Whenever we create a new user, a token will be generated for that user too.



```
import jwt from "jsonwebtoken";

const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: "30d",
  });
};

export default generateToken;
```

Figure 14. Generate token

Figure 14 shows how I generate token: *jwt.sign()* function combines the payloads, which are *id* (the id of the user) and the secret key. This *sign()* function will generate the token. The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML. After that, it returns user's email and token in the response.

```
import jwt from "jsonwebtoken"
import asyncHandler from "express-async-handler"
import User from "../models/User.js"

const protect = asyncHandler(
  async( req, res, next) => {
    let token

    if (req.headers.authorization && req.headers.authorization.startsWith("Bearer")) {
      try {
        token = req.headers.authorization.split(" ")[1];

        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        req.user = await User.findById(decoded.id).select("-password");
        next();
      } catch (error) {
        console.error(error);
        res.status(401);
        throw new Error("Not authorized, token failed");
      }
    }

    if (!token) {
      res.status(401)
      throw new Error("not authorized")
    }
  }
)

export default protect
```

Figure 15. jwt middleware

To authorize the request, I use a middleware called **protect**. After the route receives the request, it will get the current user's email and password, then finds a user that has the same email in the collection. When the route can find a user with the provided email, it compares the given password with the one that is stored in the collection. If the password matches, it starts generating the token. If not, the route returns an error message. If there are any problems with finding a user, it returns a "Not authorized" message. You can see the **protect** middleware in Figure 15.

Figure 16 displays how we log in into the site:

```

userRouter.post("/login", asyncHandler(async (req, res) => {
  const { email, password } = req.body
  const user = await User.findOne({ email })

  if (user && (await user.matchPassword(password))) {
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      token: generateToken(user._id),
      createdAt: user.createdAt,
    })
  } else {
    res.status(401)
    throw new Error("Invalid Email or Password")
  }
})
)

```

Figure 16. Login route

We have the back-end, but currently it only works on the local server. So if we want to use it on the internet, we must deploy it to a server online. In this project, I use Heroku to deploy the back-end application. There are multiple ways to deploy using Heroku, but I choose the easiest way for me: using GitHub repo. On Heroku, I created a new application called *phonestopapi*, and then connect to

GitHub to get my repo, then deploy it. Figure 17 shows the successfully deployment.

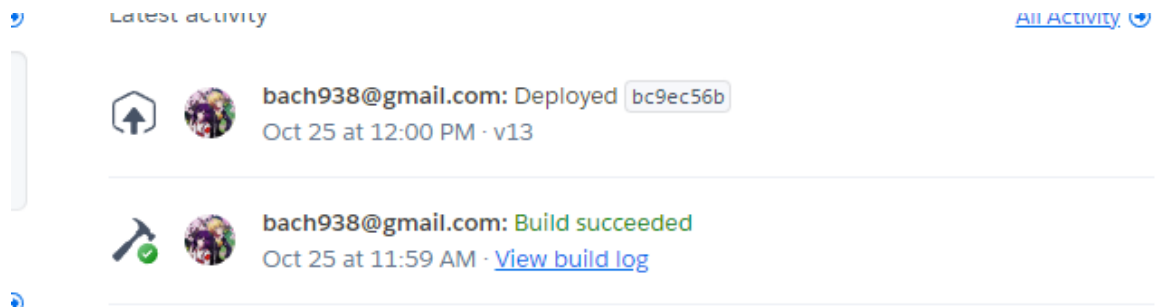
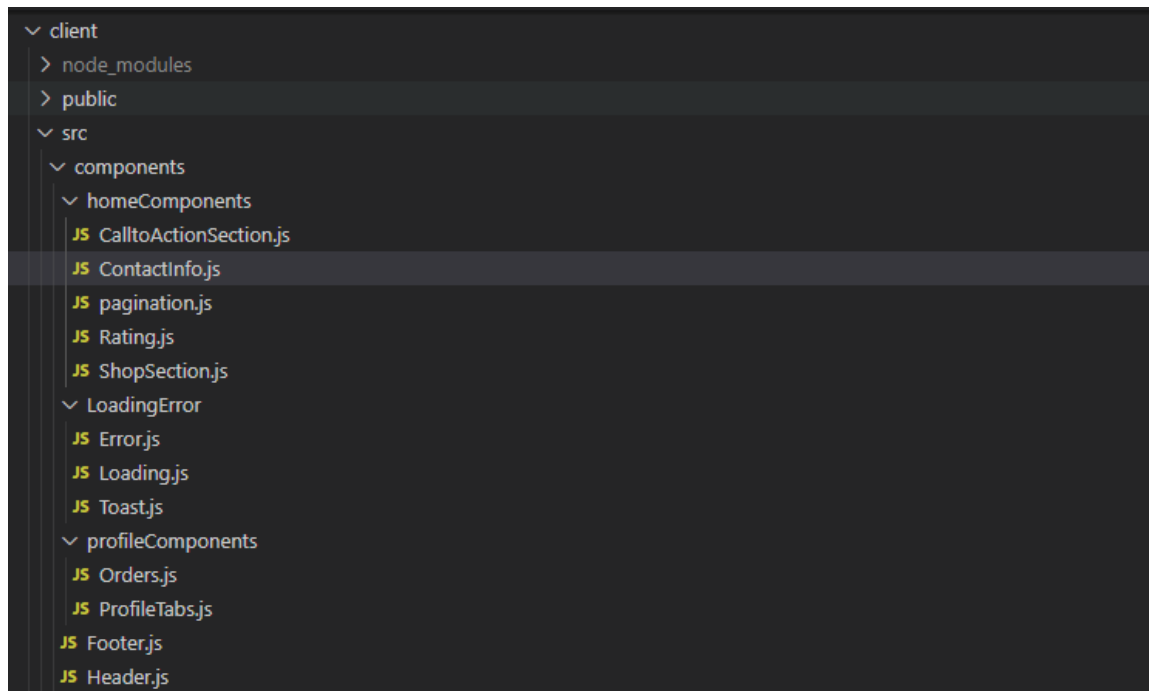


Figure 17. Heroku deploy

### 3.3 Web application implementation

#### 3.3.1 Structure

Figure 18 shows how my project is structured:



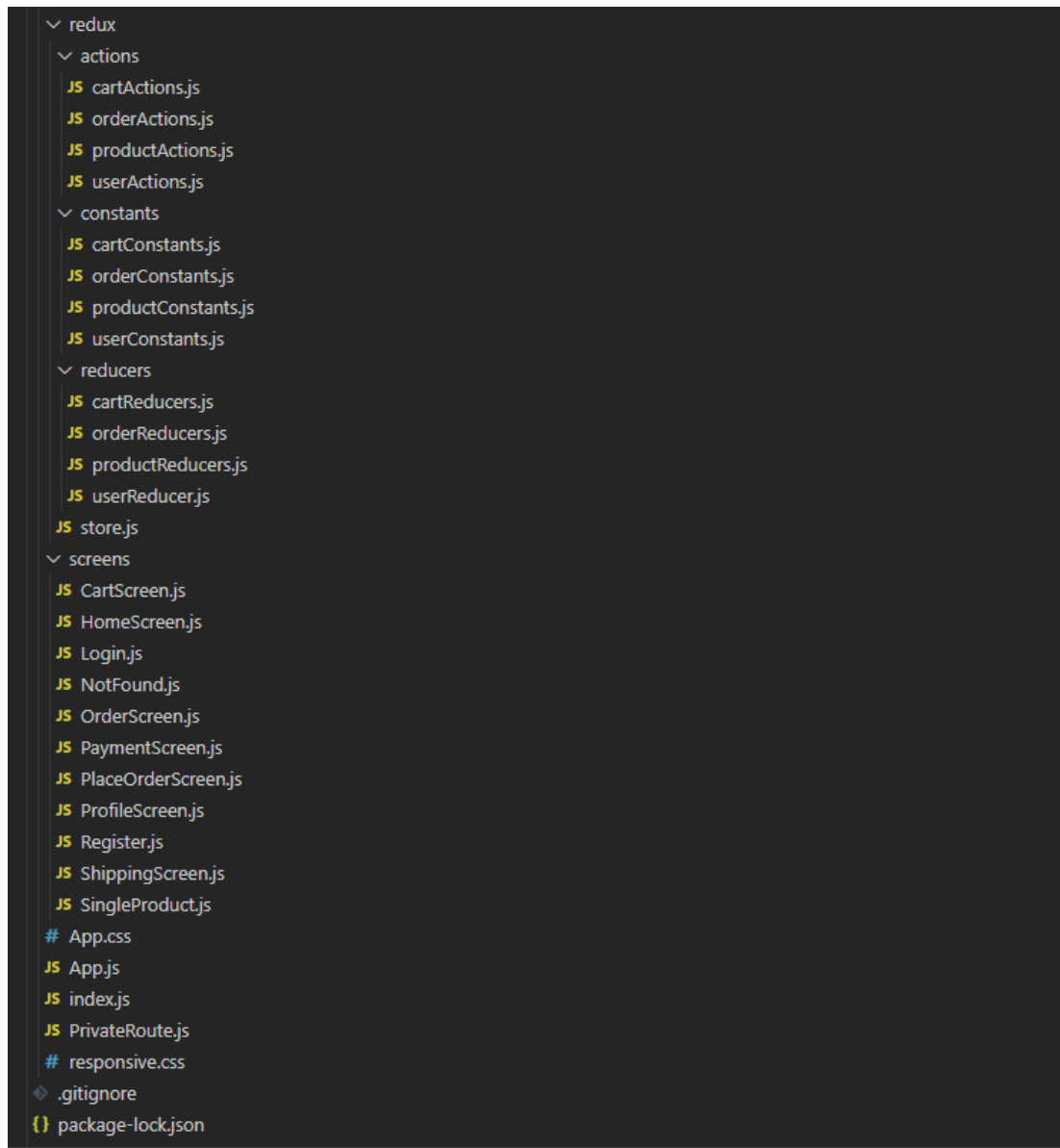


Figure 18. Front-end structure

### 3.3.2 Styling

CSS or Cascading Style Sheet is a style sheet language used for describing the presentation of a document written in a markup language such as HTML or XML. CSS handles the look and feel part of a web page.

Using CSS, you can control the color of the text, the font styles, the spacing between paragraphs, how columns are sized and laid out, what background images or colors are used, layout designs, variations in display for different devices, and screen sizes as well as a variety of other effects.

In this project, I put all my CSS codes to a file called “App.css” as you can see in Figure 19 below.

```
# App.css X
client > src > # App.css > .subscribe-section
1  body {
2    margin: 0;
3    font-family: "Josefin Sans", sans-serif;
4    /* font-family: 'Open Sans', sans-serif; */
5    -webkit-font-smoothing: antialiased;
6    -moz-osx-font-smoothing: grayscale;
7    font-size: 17px;
8    padding: 0;
9    box-sizing: border-box;
10 }
11
12 a {
13   text-decoration: none;
14   color:  black;
15 }
16
```

Figure 19. App.css file

### 3.3.3 Screens and Components

As stated before in the theory chapter, components make our code reusable and split our UI into different pieces. Figure 20, 21 shows all the components that are used in this project.

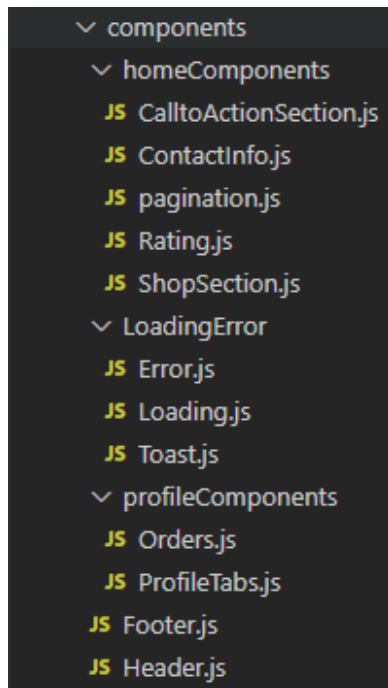


Figure 20. Components

Figure 20 displays the list of additional components in my project. As you can see there are Header and Footer and they will be used on every screen. There are other small components that work with other screens. For example, *homeComponents* that are used on *HomeScreen*.

In my project, there are screens with different functions, for example, cart screen, order screen, payment screen, etc. The list of screens that are used in this project is also shown in the Figure 21.

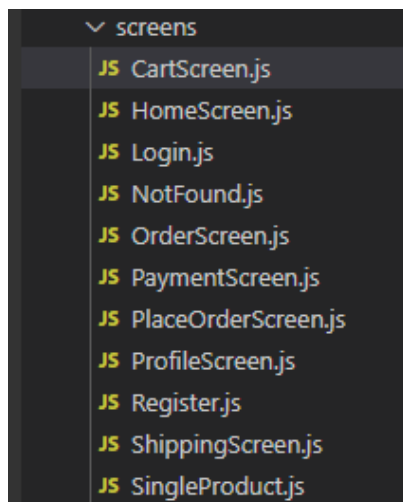


Figure 21. Screens

**CartScreen** component provides the list of ready-to-checkout products.

**HomeScreen** is the landing screen of the site, it provides basic information about the company and its products. **OrderScreen** and **PlaceOrderScreen** is the screen where users can view their placed orders. By instance, the **SingleProduct** component showed users detailed of the product, including its name, category, and price. **Login** is the page for users using their email and password to sign in, requiring a valid email and password. **Users** can redirect to **Register** page to create a new account by providing an email valid address, password.

To navigate to all the screens, I use React Router. Figure 22 displays how I use react router in this project.

```

<Router>
  <Switch>
    <Route path="/" component={HomeScreen} exact />
    <Route path="/search/:keyword" component={HomeScreen} exact />
    <Route path="/page/:pagenumber" component={HomeScreen} exact />
    <Route
      path="/search/:keyword/page/:pageNumber"
      component={HomeScreen}
      exact
    />
    <Route path="/products/:id" component={SingleProduct} />
    <Route path="/login" component={Login} />
    <Route path="/register" component={Register} />
    <PrivateRoute path="/profile" component={ProfileScreen} />
    <Route path="/cart/:id?" component={CartScreen} />
    <PrivateRoute path="/shipping" component={ShippingScreen} />
    <PrivateRoute path="/payment" component={PaymentScreen} />
    <PrivateRoute path="/placeorder" component={PlaceOrderScreen} />
    <PrivateRoute path="/order/:id" component={OrderScreen} />
    <Route path="*" component={NotFound} />
  </Switch>
</Router>

```

Figure 22. React router

### 3.3.4 Redux

Redux is a predictable state container designed to help you write JavaScript apps that behave consistently across client, server, and native environments, and are easy to test. Redux is used to maintain and update data across your applications for multiple components to share, all while remaining independent of the components.

The working way of Redux is pretty simple. There is a central "store" that keeps the state of the application. Each component can access any state that it needs from this store. There are three core components in Redux: actions, store, and reducers.

In this project, Redux is mainly used to manage the loading, adding and deleting state of the products in store and cart. Figure 23 displays the Redux store of this application.

```
const initialState = {
  cart: {
    cartItems: cartItemsFromLocalStorage,
    shippingAddress: shippingAddressFromLocalStorage,
  },
  userLogin: {
    userInfo: userInfoFromLocalStorage,
  },
}

const middleware = [thunk]

const store = createStore(
  reducer,
  initialState,
  composeWithDevTools(applyMiddleware(...middleware))
)

export default store
```

Figure 23. Redux store



The store is a “container” that holds the application state, and the only way the state can change is through actions dispatched to the store. Redux allows individual components connect to the store and apply changes to it by dispatching actions.

Figure 24 shows an example how Redux was used in this one particular function: createOrder.

```
//get order create state
const orderCreate = useSelector((state) => state.orderCreate)
const { order, success, error } = orderCreate
```

Figure 24. get orderCreate state

As you can see from Figure 25, we have the information from the cart, and now we will dispatch them to create a new order.

```
const placeOrderHandler = () => {
  dispatch(
    createOrder({
      orderItems: cart.cartItems,
      shippingAddress: cart.shippingAddress,
      paymentMethod: cart.paymentMethod,
      itemsPrice: cart.itemsPrice,
      shippingPrice: cart.shippingPrice,
      taxPrice: cart.taxPrice,
      totalPrice: cart.totalPrice,
    })
  )
}
```

Figure 25. createOrder

This information that we have dispatched from the handler will be handled by the createOrder actions in Figure 26.

```

export const createOrder = (order) => async (dispatch, getState) => {
  try {
    dispatch({ type: ORDER_CREATE_REQUEST })

    const {
      userLogin: { userInfo },
    } = getState()

    const config = {
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${userInfo.token}`,
      },
    }

    const { data } = await axios.post(`${link}/api/orders`, order, config)
    dispatch({ type: ORDER_CREATE_SUCCESS, payload: data })
    dispatch({ type: CART_CLEAR_ITEMS, payload: data })

    localStorage.removeItem("cartItems")
  } catch (error) {
    const message =
      error.response && error.response.data.message
      ? error.response.data.message
      : error.message
    if (message === "Not authorized, token failed") {
      dispatch(Logout())
    }
    dispatch({
      type: ORDER_CREATE_FAIL,
      payload: message,
    })
  }
}

```

Figure 26. createOrder actions

Simply put, Redux actions are events. They are the only way you can send data from your application to your Redux store. We just dispatch them to our store instance whenever we want to update the state of our application. The rest is handled by the reducers.

```
export const orderCreateReducer = (state = {}, action) => {
  switch (action.type) {
    case ORDER_CREATE_REQUEST:
      return { loading: true }
    case ORDER_CREATE_SUCCESS:
      return { loading: false, success: true, order: action.payload }
    case ORDER_CREATE_FAIL:
      return { loading: false, error: action.payload }
    case ORDER_CREATE_RESET:
      return {}
    default:
      return state
  }
}
```

Figure 27. createOrder reducer

Whenever we dispatch an action to our store, the action gets passed to the reducer. Reducers take the previous state of the app and return a new state based on the action passed to it. As you can see from the Figure 27, the reducer will return state based on which case the action did.

### 3.4 My website application

#### 3.4.1 Home Screen

The home screen is the landing screen of the website. It includes Header, the list of products, a search bar, profile menu button, and cart button. As shown in Figure 28, this is the home screen of my application.

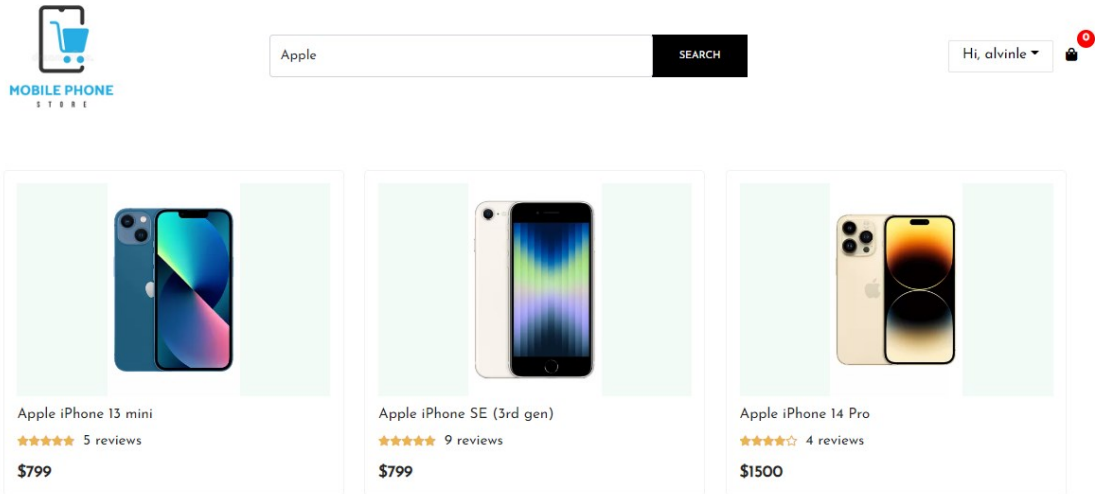


Figure 26. HomeScreen

The footer of the website includes email subscriptions for the users, contact information of the company, and finally types of payment that accepted on this site.

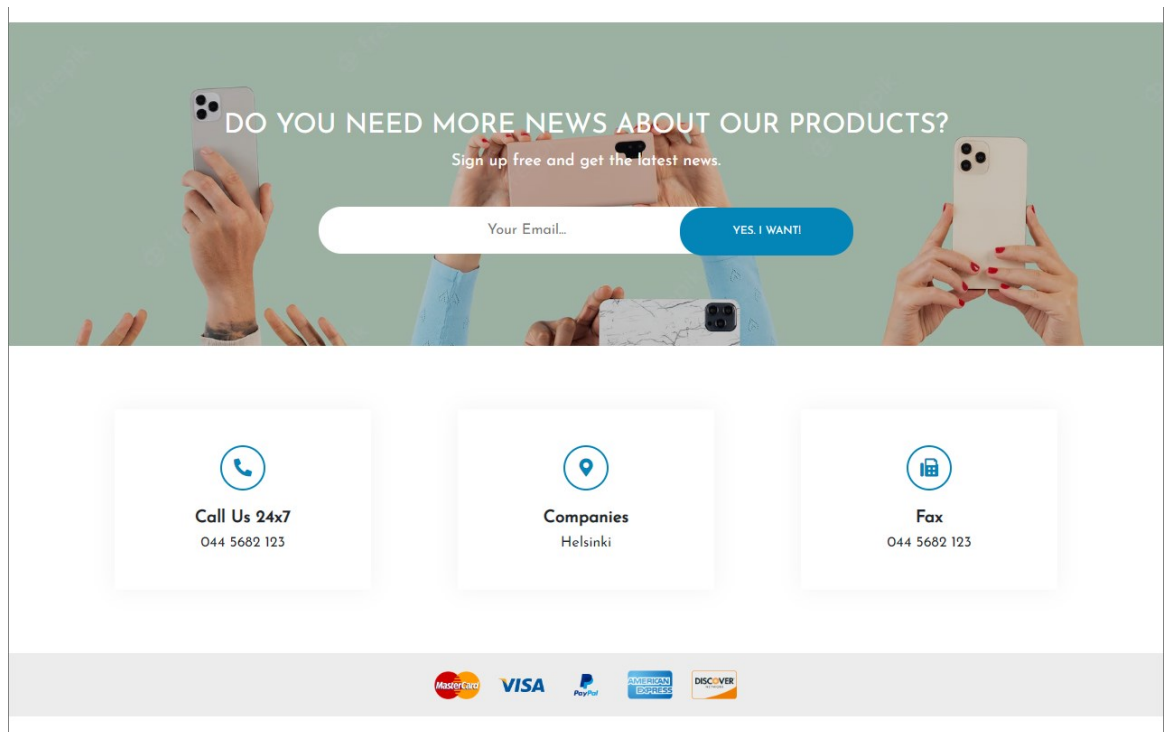


Figure 27. Footer

The product data can be get from the API path. As shown in Figure 30, I combined pagination and search functions into this API, that's why you can see there are *keyword* and *pageNumber* params on the GET request. To make the request, I used the axios library. It allows you to make a request from the application to get data from the server.

```
export const listProduct =
(keyword = " ", pageNumber = " ") =>
  async (dispatch) => {
    try {
      dispatch({ type: PRODUCT_LIST_REQUEST });
      const { data } = await axios.get(
        `/api/products?keyword=${keyword}&pageNumber=${pageNumber}`
      )
      dispatch({ type: PRODUCT_LIST_SUCCESS, payload: data });
    } catch (error) {
      dispatch({
        type: PRODUCT_LIST_FAIL,
        payload:
          error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
      });
    }
  };
};
```

Figure 28. Get product list request

### 3.4.2 Single Product Screen

When you click any product on Home screen, it will automatically guide you to Single Product screen, as you can see from figure 31 below. There will be all the information of the products: name, pictures, price, ... that will help users choosing the products.

Hi, alvinle



### Nokia X100 5G

A Big full HD+ display, immersive OZO Audio technology, and super-fast streaming with 5G1 make the Nokia X100 your ideal entertainment wingman. All with facial recognition and a fingerprint sensor to quickly access everything on your phone. Safely kept, easily accessed.

Price	<b>\$399</b>
Status	<b>In Stock</b>
Reviews	★★★★☆ <b>9 reviews</b>
Quantity	1 <input type="button" value="v"/>
<input type="button" value="ADD TO CART"/>	

REVIEWS

WRITE A CUSTOMER REVIEW

Rating

Comment



Figure 29. Single Product Screen

On this screen, users can write reviews of the products, rate them, and have some comments about the products. This screen also has “Add to cart” button, choosing how many products you want to buy then click the button, it will guide you to the Cart screen.

### 3.4.3 Cart Screen

After clicking the “Add to cart” button, or accessing to the cart using the button on the Home screen, the Cart screen will appear. As seen from Figure 32, you can see all the products that have been added to the cart and the total price.

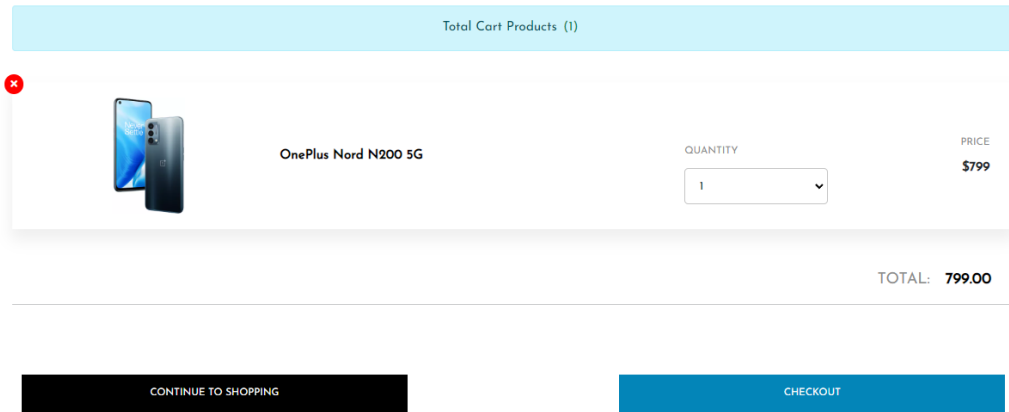


Figure 30. Cart Screen

You have 2 options: “Continue to Shopping”, and “Check out”. If you choose the first one, you will be redirected to the Home screen, to continue looking for products. If you want to check out, just click the “Check out” button.

#### 3.4.4 Shipping Screen

After clicking the checkout button, the shipping screen will appear. You will input your address so that the store can ship the product directly to your home. Click “Continue” and it will get to the payment screen. Figure 33 displays the shipping screen.

DELIVERY ADDRESS

36 Hoang Thuc Tram

Da Nang, Vietnam

550000

Vietnam

CONTINUE

Figure 31. Shipping Screen

### 3.4.5 Payment Screen

Onto the payment screen, now I have only implemented PayPal/ Credit Card so there's only one choice for now as seen from Figure 34.

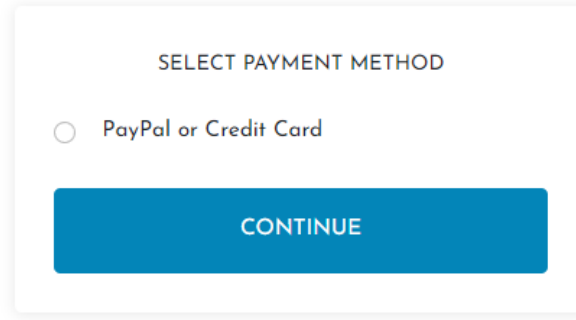


Figure 32. Payment Screen

### 3.4.6 Place Order Screen

After done choosing payment method, you will be lead to the Place order screen. Figure 35 displays the Place Order screen. You can see all the information about your order, price include tax and shipping fee here. Click the Place order button to get to the Order screen.

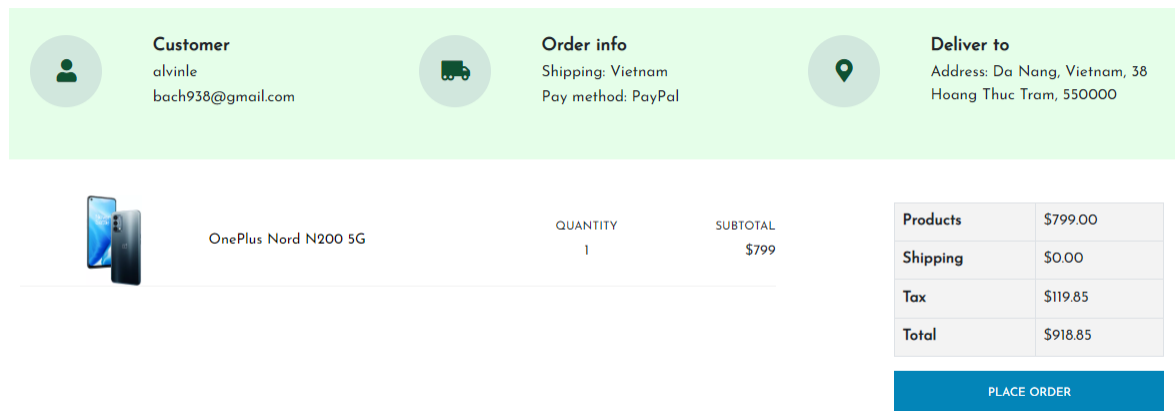


Figure 33. Place Order Screen



### 3.4.7 Order Screen

At the order screen, you can see your order again and do the payment. Figure 36 shows the Order screen.

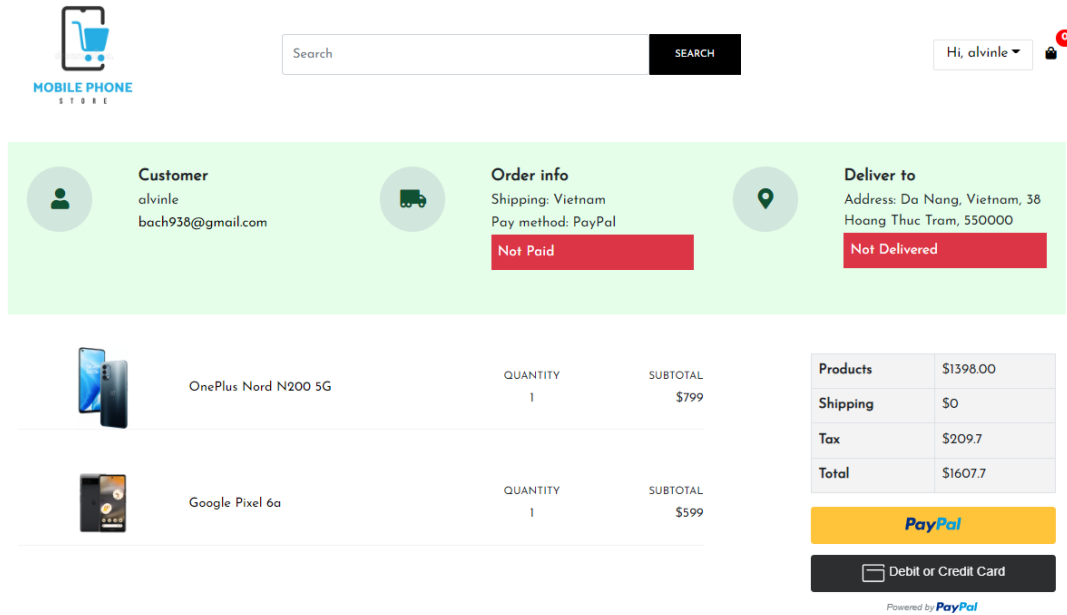


Figure 34. Order Screen

In this project, I use PayPal API for payment. Figure 37 shows how I set up the PayPal environment for my project.

```
const addPayPalScript = async () => {
  const { data: clientId } = await axios.get("/api/config/paypal")
  const script = document.createElement("script")
  script.type = "text/javascript"
  script.src = `https://www.paypal.com/sdk/js?client-id=${clientId}`
  script.async = true
  script.onload = () => {
    setSdkReady(true)
  }
  document.body.appendChild(script)
}
if (!order || successPay) {
  dispatch({ type: ORDER_PAY_RESET })
  dispatch(getOrderDetails(orderId))
} else if (!order.isPaid) {
  if (!window.paypal) {
    addPayPalScript()
  } else {
    setSdkReady(true)
  }
}
}, [dispatch, orderId, successPay, order])
```

Figure 35. PayPal setup

As you can see from Figure 38, we login with our PayPal account and pay.

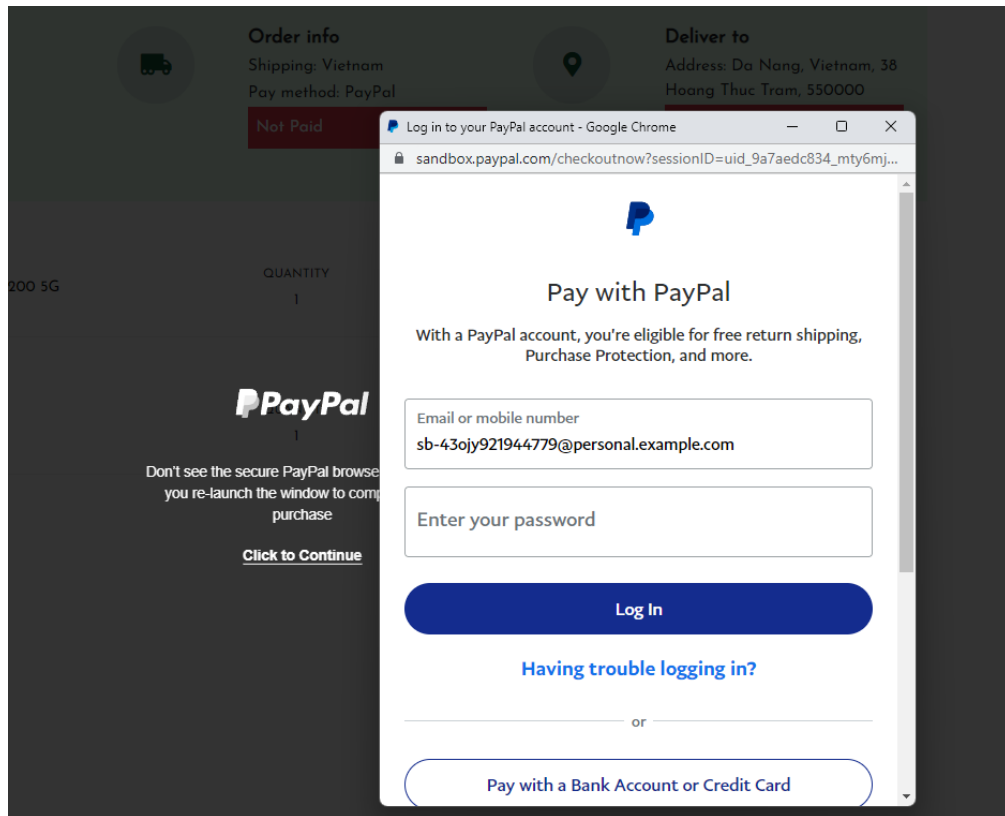


Figure 36. Do the payment

### 3.4.8 Register Screen

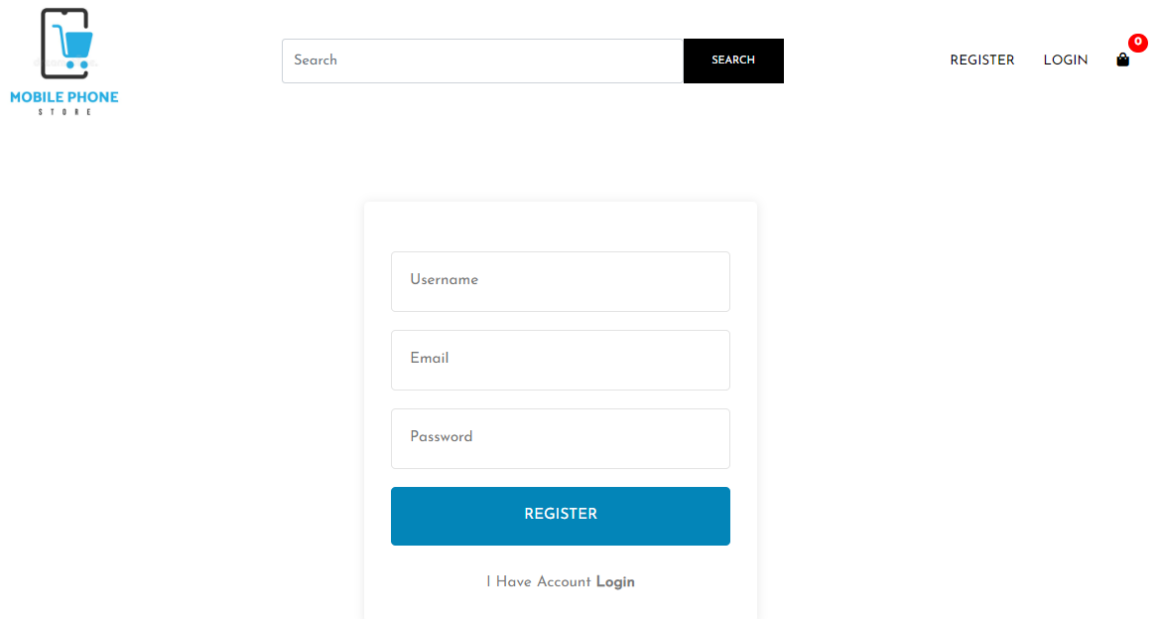
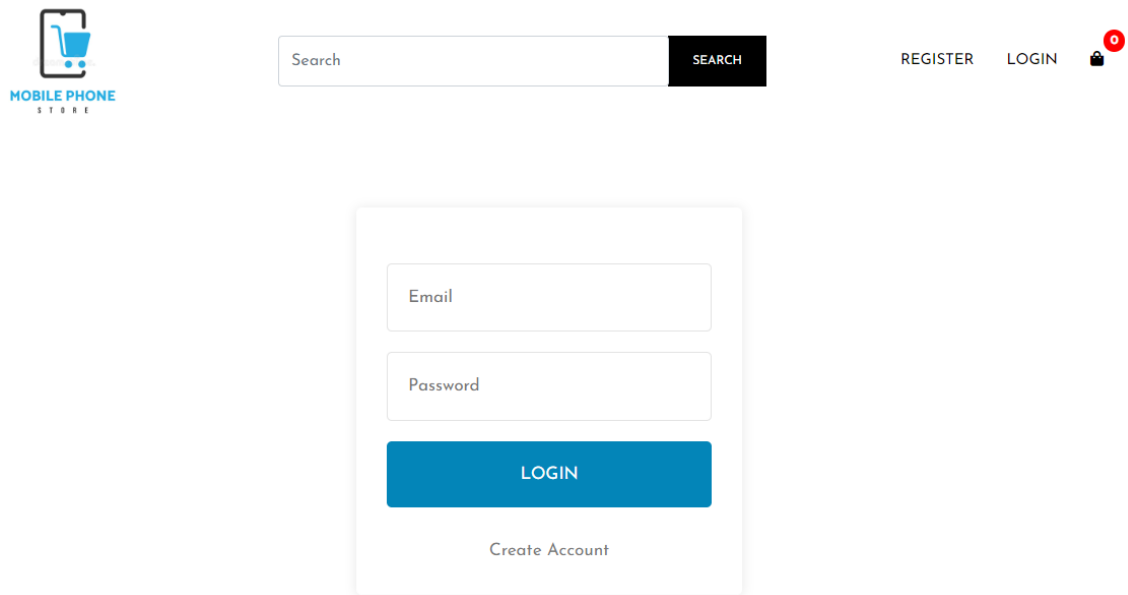


Figure 37. Register Screen

Figure 39 displays the Register screen. If users do not have accounts, they can register for a new account. When pressing the Register button, the application navigates to the Register screen. There is a form to fill in personal information. The required information includes name, email, phone number, and password. After registering successfully, the site will automatically sign you in.

### 3.4.9 Log In Screen



The screenshot shows the top navigation bar of the 'MOBILE PHONE STORE' website. On the left is the store's logo. In the center is a search bar with the placeholder text 'Search' and a black 'SEARCH' button. On the right are links for 'REGISTER', 'LOGIN', and a shopping cart icon with a red notification badge. Below the navigation bar is a white login form with a light gray border. The form contains two input fields: 'Email' and 'Password'. Below these fields is a prominent blue 'LOGIN' button. At the bottom of the form is a link that says 'Create Account'.

Figure 38. Log In Screen

If users already have accounts, they can log in with their existing accounts on the login screen. Figure 41 shows the implementation of the login process and saving the information to the reducer.

```

export const LogIn = (email, password) => async (dispatch) => {
  try {
    dispatch({ type: USER_LOGIN_REQUEST })
    const { data } = await axios.post(
      `/api/users/login`,
      { email, password },
      {
        headers: {
          "Content-Type": "application/json",
        },
      }
    )
    dispatch({ type: USER_LOGIN_SUCCESS, payload: data })

    localStorage.setItem("userInfo", JSON.stringify(data))
  } catch (error) {
    dispatch({
      type: USER_LOGIN_FAIL,
      payload:
        error.response && error.response.data.message
        ? error.response.data.message
        : error.message,
    })
  }
}

```

Figure 39. Log In request

When you have logged in, on the right of the header, there will a menu for account as you can see from Figure 42.

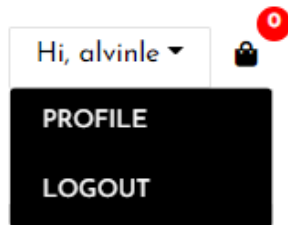


Figure 40. Profile menu

You can choose to log out or choose profile to access to your profile.

### 3.4.10 Profile Screen

After clicking the Profile button, you can see all the information of the current user, they can also make changes to their profile by using the update button. Figure 43 displays Profile screen.

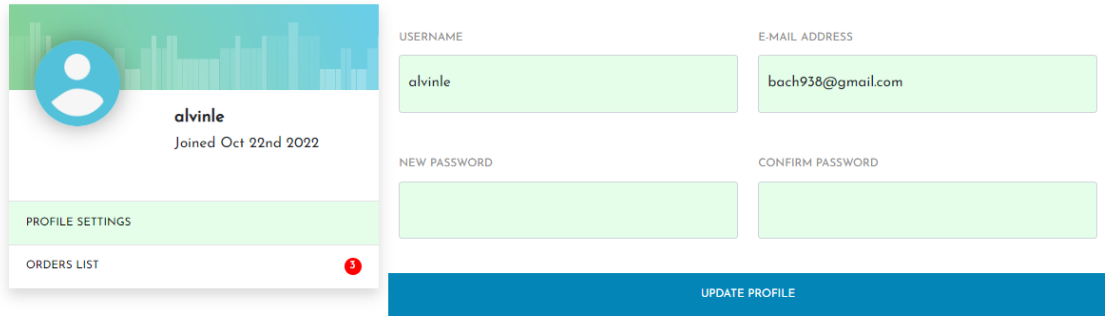


Figure 41. Profile Screen

There will be a list of orders that user have made before as you can see from Figure 44.



Figure 42. Profile Orders list

## 4 CONCLUSION

The goal of this thesis was to create an e-commerce web application and research the basics of MERN Stack: MongoDB, Express.js framework, React.js library and Node.js platform. The theoretical background section has covered the study of e-commerce concept, also methods and technologies to implement the thesis. This chapter helps us to understand how this thesis project was implemented.

The practical implementation part described the process of building this e-commerce application, on both front-end and back-end parts. It provides how to planning and deploying the server to save all the data of the store. It also shows all the steps to create a front-end application and connect the front-end part to the server that we have created. The screenshots of the application are also displayed in this section. By applying the methods and technologies that had been discussed in the theory part, the application was developed successfully. With the provided tools, the application fulfills the goals. The application still needs to be developed more in the future, for example, create an admin dashboard for easier management, include google authentication and so on.

## REFERENCES

Alesia Sirotko. Feb 18, 2022. What is React? WWW document. Available at: <https://flatlogic.com/blog/what-is-react/> [Accessed 29 Oct 2022]

Auth0. No date. Introduction to JSON Web Tokens. WWW document. Available at: <https://jwt.io/introduction> [Accessed 29 Oct 2022]

Besant. No date. What is Express.js? WWW document. Available at: <https://www.besanttechnologies.com/what-is-expressjs> [Accessed 30 Oct 2022]

Donna Fuscaldo. Oct 20, 2021. The History of Ecommerce: How Did It All Begin? WWW document. Available at: <https://www.businessnewsdaily.com/15858-what-is-e-commerce.html> [Accessed 28 Oct 2022]

MDN. No date. JavaScript — Dynamic client-side scripting. WWW document. Available at: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript> [Accessed 29 Oct 2022]

Miva. November 23, 2020. The History of Ecommerce: How Did It All Begin? WWW document. Available at: <https://blog.miva.com/the-history-of-ecommerce-how-did-it-all-begin> [Accessed 28 Oct 2022]

Mosh Hamedani. December 3, 2018. React Virtual DOM Explained. WWW document. Available at: <https://programmingwithmosh.com/react/react-virtual-dom-explained/> [Accessed 29 Oct 2022]

Neo Ighodar. October 3, 2022. Understanding Redux: A tutorial with examples. WWW document. Available at: <https://blog.logrocket.com/understanding-redux-tutorial-examples/> [Accessed 4 Nov 2022]

TutorialsTeacher. No date. What is MongoDB? WWW document. Available at: <https://www.tutorialsteacher.com/mongodb/what-is-mongodb> [Accessed 2 Nov 2022]