Full-stack e-commerce application

Contents

List of Abbreviations

DBMS:	Database management system. Software for maintaining, querying and updating data and metadata in a database.
DB:	Database. An organized collection of structured information, or data, typically stored electronically in a computer system.
ORM:	Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.
UI:	User interface refers to the screens, buttons, toggles, icons, and other visual elements that the user interacts with when using a website or an app.
API:	API is a mechanism that enables two software components to communicate with each other using a set of definitions and protocols.
DB:	Database. An organized collection of structured information, or data, typically stored electronically in a computer system.
XXS:	Cross-Site Scripting is a security flaw found in some Web applications that enables unauthorized parties to cause client-side scripts to be executed by other users of the Web application.
SQL:	Structured Query Language is a domain-specific language used in programming and designed for managing data held in a relational database management system.
CSRF:	Cross-Site Request Forgery is an attack that forces authenticated users to submit a request to a Web application against which they are currently authenticated.

1 Introduction

In the past decades, humans have used several means for trading and purchasing goods starting from modest shops to big malls and open markets. And as a result of the booming population increase rate, people have shown a growing demand for both national and international services and goods, where clients are facing a lack of resources and local offers or high-cost goods due to competition between big vendors, and here the role of e-commerce websites can be noticed as a solution for both consumers and manufacturers. At the same time, there have been some social media platforms integrating the same idea in their business model by providing a seamless experience and benefits. However, due to the original nature of social media, the user is always losing attention when shopping due to entertainment subjects broadcasted on platforms and the lack of an online store shopping experience.

The goal of this thesis is to discover the great potential of the MERN technology for building a complete full-stack E-commerce web application using Node JS, Express JS, MongoDB, Firebase and React JS, as they have gained wide popularity amongst web and mobile developers as they provide a highly scalable and powerful solution for building full-stack applications, at the same time providing a dynamic usage across different platforms.

The primary objective of this full-stack application is to deliver a hassle-free online shopping experience to its users, with an extensive product range, uncomplicated payment options, and a user-friendly interface. The application targets a consumer audience of diverse ages and backgrounds who prioritize convenience and prefer to shop from the comfort of their homes. Additionally, it caters to individuals with busy lifestyles who may find it challenging to visit physical stores in person.

As this study will go through the detailed process of developing a full-stack e-commerce web application, by simplifying and splitting the different stages starting from where and how the data is stored in the backend, the implementation of the front-end part which will interact with the client actions and the communication with the backend to execute the desired operations and queries.

1.1 Process and project structure

To build an e-commerce full-stack application the developer has to go through a strict process starting by defining the technology stacks that going to be used throughout the back-end and frontend next to other necessary services and software tools to develop the application in a good shape and support various features.

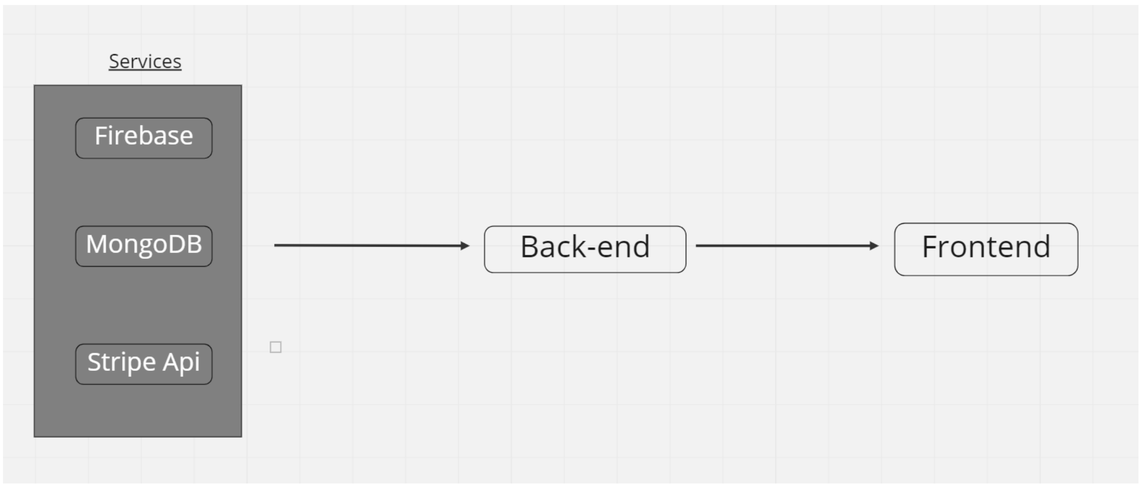


Figure 1. Project process structure

The above figure represents the process in which this project has been carried on. Firstly, services were researched to understand which options to integrate in the current e-commerce application. Lastly, when services are defined, the project implementation can be divided into two parts frontend and back-end to a well-structured and clear implementation process.

2 Full-stack web development

Full-stack web development is a software development approach that has grown in popularity in the last decade, as it permits developers to build and maintain websites and web platforms, as it combines both the frontend part that interacts with the client and back-end side which stores and takes care of the logical layer for handling the data of the application [1.], which puts full-stack web developers in a high necessity to have knowledge and experience in both front-end and back-end technologies. They must also understand how to integrate both layers to craft a seamless web application. This requires knowledge of various web development tools and technologies, including web servers, application servers, databases, and APIs

In addition to technical skills, full-stack web developers also need to have strong problem-solving skills and a team player spirit, as they need to communicate effectively with all the project partners to ensure that the final product is delivered on time, meets the exact project features and delivers useful needs to users.

On the other hand, a full-stack developer has an advantage to work on a project from beginning to end, which allows for greater flexibility and efficiency. Additionally, full-stack developers can make changes to both the front-end and back-end components of a web application, which can enhance the overall functionality and user experience. Another benefit of being a full-stack developer is the ability to work with various web development technologies, including programming languages, frameworks, and databases, which allows for greater adaptability and versatility. Due to these advantages, full-stack web development is in high demand in the tech industry.

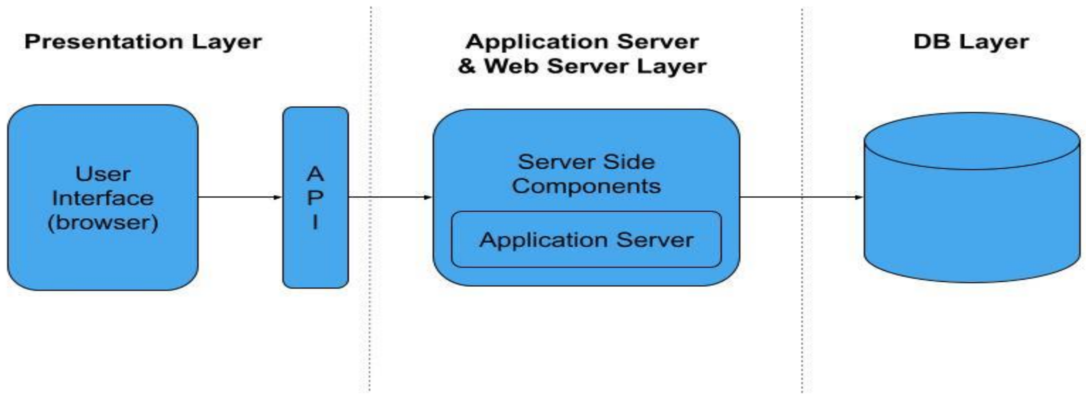


Figure 2. Structure of a full-stack application

As shown in the figure above, the presentation layer, or as known as (the front end) is most of the application that the user interacts with, and it is responsible part for displaying all the information and data in a more readable and user-friendly way. In our case, we will be using React JS.

The second part in the figure indicates the application server-side layer, which handles the logic of connecting the Database part with the front-end part, as it also specifies routes that the front end can use for any specific query, in which we will be using Node JS and Express JS.

The last part shown in the picture demonstrates the Database layer, which holds and stores all the information and data that the application is using. And for our case, we will be using Firebase and MongoDB.

2.1 Security

Full-stack web applications are vulnerable to security threats, as organisations always thrive to provide new features to their customer base or audience on a wide range over a shared network, as a result all the shared data is subject to an unauthorized access, data theft, and system compromise. [2, p.4].

There are various types of attacks that a full-stack application can be threatened by, such as XSS attacks, that can highly happen when the attacker injects malicious codebase script into a web page, which can execute scripts and access sensitive data, and to mitigate this threat, developers should sanitize user input and use encoding techniques to prevent the execution of malicious code, also (WAFs) or Web application firewalls shall be used as a security measure by detecting and blocking XSS attacks. [3.]

On the other hand, SQL injection attacks involve the insertion of malicious SQL statements into an application's input fields, which can allow attackers to view, modify or delete data, and in order to prevent SQL injection attacks, developers should consider parameterized queries or stored procedures. Parameterized queries make use of placeholders for user input, which are validated and sanitized before execution. [4.]

Also, CSRF attacks is one of the threats that occur when an attacker sends a request from a user's browser to a web application without the user's knowledge or consent, which can result in unwanted actions such as unauthorized transactions. [4.] Developers can use CSRF tokens or nonce values to ensure that each request is valid and initiated by an authorized user. Tokens are generated by the server and included in each form or link, which the user must submit along with the request to prove authenticity.

In addition to these specific threats, developers should also follow general security best practices to protect their full-stack web applications. Passwords should be securely stored using hashing and salting techniques to prevent unauthorized access to user accounts. HTTPS can be used to encrypt all data transmitted between the client and server.

To improve the security of full-stack web applications, developers can use various tools and techniques such as static code analysis, penetration testing, and monitoring. Static code analysis tools can identify potential vulnerabilities in the codebase before deployment. Penetration testing has the ability to replicate genuine attacks and detect areas of weakness that were not detected during development. Monitoring and logging can be used to detect and respond to security incidents quickly, allowing developers to take appropriate action to prevent further damage.

3 Mern stack

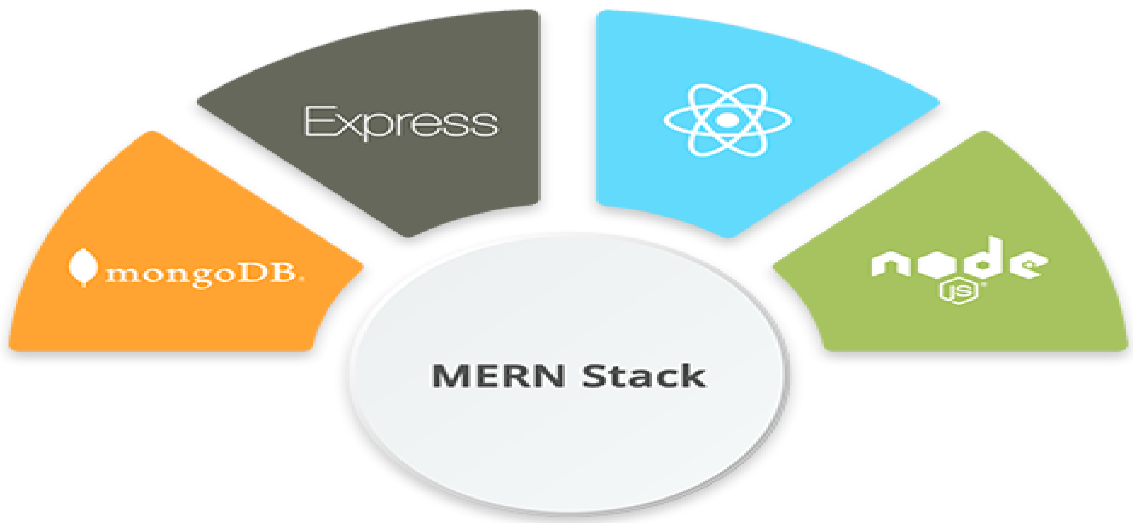


Figure 3. MERN stack components

The MERN Stack, a popular full-stack development technology stack, is composed of four primary technologies. The M in MERN stands for MongoDB, which is a NoSQL (Non-Structured Query Language) database system primarily utilized for document database preparation. The E in MERN stands for Express, a web development framework for Node.js. The R in MERN stands for React.js, which is mainly employed for client-side JavaScript application development. Lastly, the N in MERN represents Node.js, which is used as the foundation for JavaScript web server development. See figure 3 above.

All four of these technologies are JavaScript based and can be contributed to as they are open source, and due to their independent state on each other, the developer needs to familiarize himself with them to understand their basic usage and work combination from both the front-end and the back end. [5, pp. 9-10.]

3.1 Node Js

Node.js is an environment in which JavaScript codebases can be compiled and run. It's open-source and works on multiple platforms. The Node.js application functions within a single process, and the libraries it uses employ non-blocking paradigms. This means that blocking behaviours are not common. [6.]

Furthermore, when the Node application is running, and to ensure the efficiency of the asynchronous running processes, all files should be loaded into memory.

3.2 Express Js

Express is the most widely used Node web framework, and many other popular Node web frameworks use it as their foundation. It offers a range of capabilities, such as the ability to write request handlers for different HTTP verbs at various URL paths (routes). [7.]

Additionally, Express can work together with "view" rendering engines to create responses by injecting data into templates. Express also provides options for customizing common web application settings, such as the port used for connections and the location of templates used for rendering responses. [7].

Lastly, Express allows for the addition of extra request processing "middleware" at any stage within the request handling pipeline. [7.]

3.3 MongoDB

MongoDB is a database that is based on a distributed architecture, which enables it to have inherent features for high availability, horizontal scaling, and geographic distribution without any additional configuration. This distributed system ensures that data can be easily replicated across multiple servers, enabling the efficient storage and retrieval of large amounts of data. Moreover, MongoDB has built-in capabilities that make it user-friendly and easy to manage, providing a flexible and powerful database solution. Consequently, it has gained popularity among businesses and organizations that require a reliable and scalable database system. [8, p.7].

In MongoDB, a record is referred to as a document, which consists of pairs of fields and values within a data structure. MongoDB documents are similar to JSON objects and can include other documents, arrays, and arrays of documents as values for fields. Documents in MongoDB are stored in collections, which are similar to tables in traditional relational databases, with each document serving as a row within a collection.

3.4 React Js

React.js is a freely available JavaScript library that is utilized to develop UIs for single-page applications. It is particularly useful in handling the view layer for web and mobile applications. Also, React.js offers the capability to create reusable UI elements. Jordan Walke, who was employed as a software engineer by Facebook, is the original creator of React.js, and the first usage of React.js was on Facebook's newsfeed in 2011, while in 2012 it was utilized on Instagram.com.

Additionally, React.js is a library that is purpose-built for creating user interfaces, which is the fundamental requirement in many cases. This description is commendable because it doesn't attempt to include every conceivable feature. React.js isn't an extensive framework that addresses every aspect of a full-stack solution, from the database to real-time updates over WebSocket connections. [9, p.10.]

3.5 Firebase

Firebase has established itself as a leading player in the BaaS (Backend as a Service) realm, consistently enhancing the cloud experience through the introduction of innovative features and functionalities. Among its peers, Firebase is the sole provider of auto-syncing database functionality, enabling the creation of exceptional applications, the expansion of consumer bases, and the generation of increased monetary value. Each feature of Firebase operates independently, and when used together, they perform exceptionally well. Consequently, Firebase has generated significant excitement within the developer community. Unlike many traditional backend services that necessitate substantial implementation efforts to launch a product, Firebase is quite straightforward to set up and deploy. It is the perfect choice for time-constrained development projects that require real-time data and scalability. [10, p.10.]
Moreover, developers can effortlessly mix and match Firebase products to tackle daily app development challenges.

4 Client side

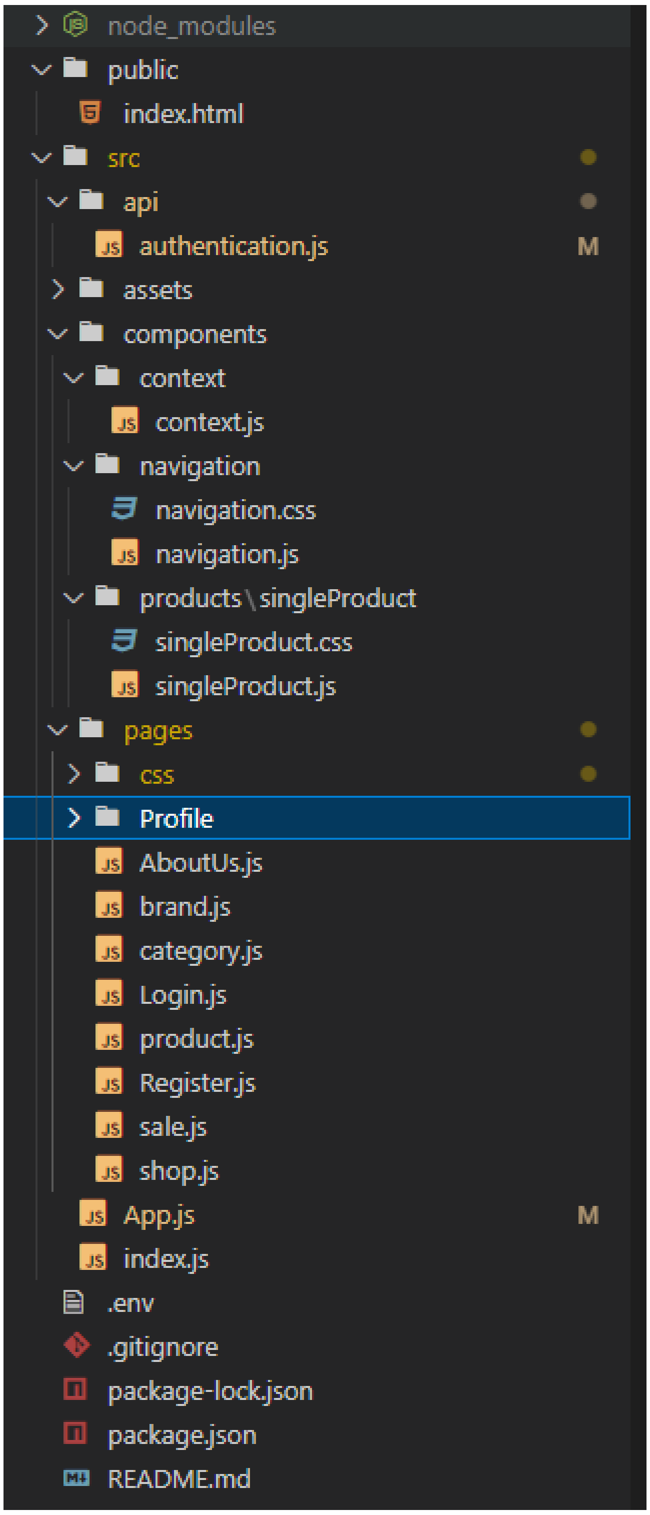
The significance of client-side technologies in full-stack development has expanded beyond using JavaScript for client-side validations to create fully-fledged, single-page applications employing client-side MVVM frameworks. However, the frameworks and toolchains employed have become increasingly intricate and daunting for inexperienced developers who are new to client-side development. [11, p.25.]

In the next chapters, we will go through the front-end part implementation of the application for a deeper understanding of the detailed steps in the creation process.

The client can only interact with the UI from the front end which then executes the actions described based on our view pages. In Total, we have 6 different pages from the concept point of view (Authentication, Home, Sale, Designers, Categories, and Profile)

4.1 Project structure

Making a good readable project structure is vital for the developer when it comes to organizing the work and implementing the logic between various files and folders.



```
> node_modules
v public
    index.html
v src                              ●
  v api                            ●
      authentication.js            M
  > assets
  v components
    v context
        context.js
    v navigation
        navigation.css
        navigation.js
    v products \ singleProduct
        singleProduct.css
        singleProduct.js
  v pages                          ●
    > css                          ●
    > Profile
      AboutUs.js
      brand.js
      category.js
      Login.js
      product.js
      Register.js
      sale.js
      shop.js
    App.js                         M
    index.js
  .env
  .gitignore
  package-lock.json
  package.json
  README.md
```

Figure 4. Project structure

As the above figure shows, we have structured our front-end application to include 5 separate folders in the ‘/src’ folder, as we have:

- ‘api’ folder contains ‘authentication.js’ that is responsible for direct authentication with the server side in the case of login and register.

- ‘assets’ folder, contains all the images that we show on our application as logos and other product brand images.

- ‘components’ folder contains 2 child folders. First, we have the context folder, which has ‘context.js’ file, which handles passing the data from the child component to the parent and vice-versa, without the need to pass a prop through all the application tree levels, which will be consuming as our application is using React navigation. Second, we have the navigation folder which holds the navigation upper bar component that helps the user navigate through our application hierarchy to access all the view pages.

- ‘pages’ folder contains all our view pages that the user can access and interact with by either inserting, fetching the data, as we will see in the next chapter.

4.2 Pages and components

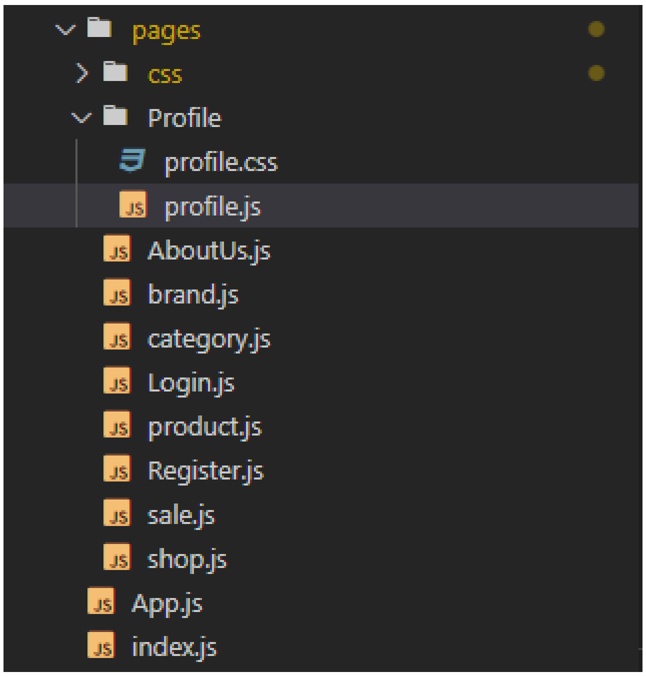

```
pages
  css
  Profile
    profile.css
    profile.js
  AboutUs.js
  brand.js
  category.js
  Login.js
  product.js
  Register.js
  sale.js
  shop.js
App.js
index.js
```

Figure 5. Project page’s structure

As shown in the figure above, our front-end application has a total of 8 public pages, that the client can access. Starting from the ‘profile’ page, where the user can see his personal information and orders history.

‘brand’ and ‘Category’ pages are more complex pages that give a unique shopping experience and wide variety of selection for the client to filter among product different brands and categories.

‘shop’ page is the home view of the entire application as it is the first page the client gets to interact with after the authentication.

‘sale’ page is containing all the discounted products, that are currently on sale.

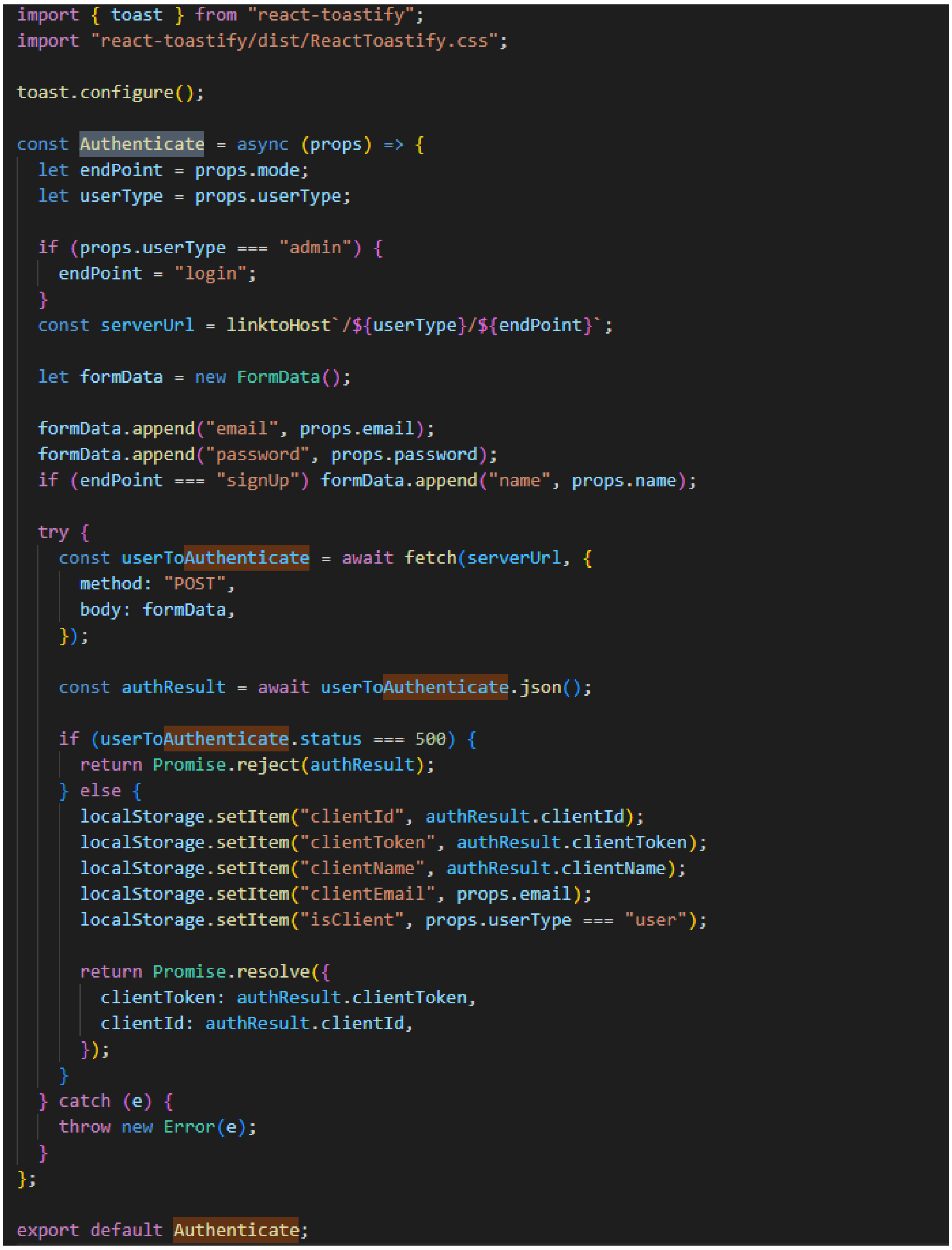
‘login’ and ‘register’ are clearly intended to authenticate the user before accessing the main view pages.

‘AboutUs’ is a profile component that contains more information and business goals about the store, also providing instructions for clients to contact the store owner.

‘product’ component is the most reusable component throughout the application as it provides same structure for displaying the product card containing all the requested data, which we will get to see later in the coming chapters.

4.2.1 Authentication

The authentication is required in our application as it gives a security layer to access our databases data and information, as it is not possible to navigate and interact with other view pages if the client has not logged-in or registered to become a member of our clients group.



```
import { toast } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

toast.configure();

const Authenticate = async (props) => {
  let endPoint = props.mode;
  let userType = props.userType;

  if (props.userType === "admin") {
    endPoint = "login";
  }
  const serverUrl = linktoHost`/${userType}/${endPoint}`;

  let formData = new FormData();

  formData.append("email", props.email);
  formData.append("password", props.password);
  if (endPoint === "signUp") formData.append("name", props.name);

  try {
    const userToAuthenticate = await fetch(serverUrl, {
      method: "POST",
      body: formData,
    });

    const authResult = await userToAuthenticate.json();

    if (userToAuthenticate.status === 500) {
      return Promise.reject(authResult);
    } else {
      localStorage.setItem("clientId", authResult.clientId);
      localStorage.setItem("clientToken", authResult.clientToken);
      localStorage.setItem("clientName", authResult.clientName);
      localStorage.setItem("clientEmail", props.email);
      localStorage.setItem("isClient", props.userType === "user");

      return Promise.resolve({
        clientToken: authResult.clientToken,
        clientId: authResult.clientId,
      });
    }
  } catch (e) {
    throw new Error(e);
  }
};

export default Authenticate;
```

Figure 6. Authentication functionality

As shown in the above image, the `Autheticate()` function is called from ‘Login’ & ‘Register’ pages that passes an action property the user want to perform either ‘login’ or ‘register’, and the ‘userType’ property which will define the profile page content restrictions for normal users.

In case of a failed authentication, a toast will be visible containing the failure reason for better usability and user experience.

After a successful success, the returned user/admin data will be stored in local storage by calling `localStorage.setItem`. and the data will be passed to the global context in order to bypass the security checks in the App.js.

4.2.2 Shop

After a succesful authentication the user will be redirected to the shop page, as it is the first view page the user interacts with, and contains several card sections for product categories, brands, and discounts to select and chose from.

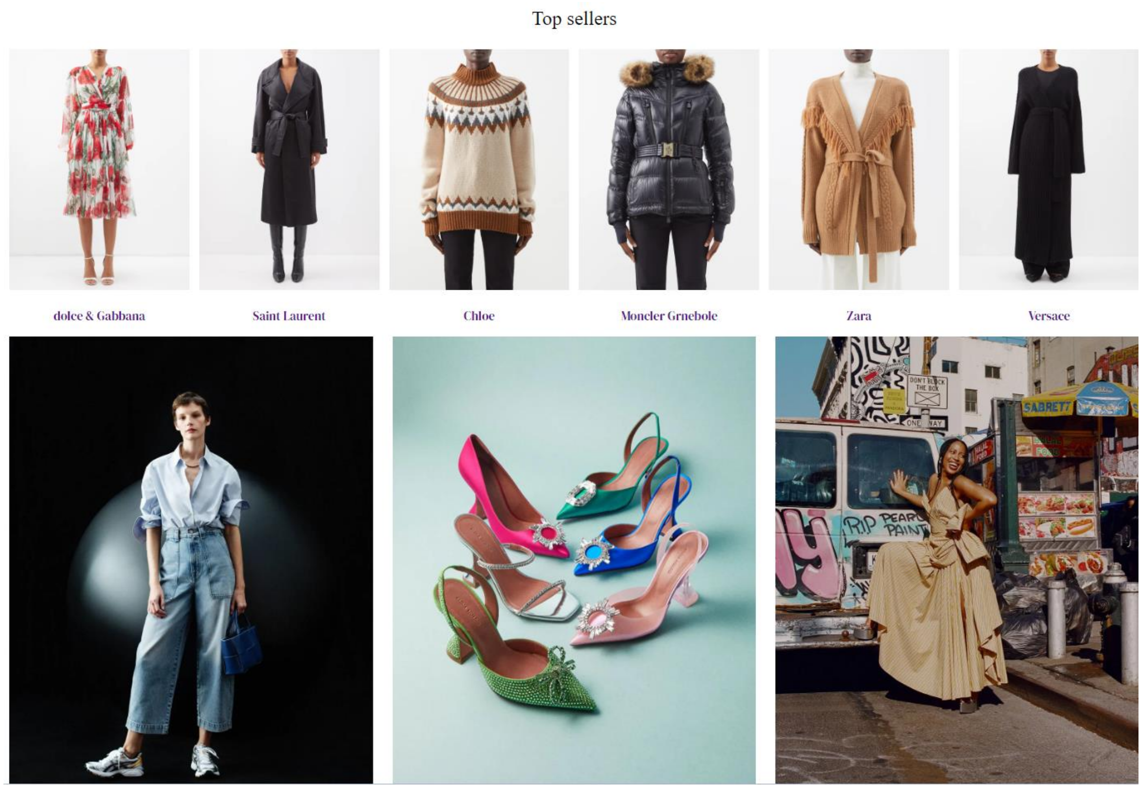


Figure 7. Shop view page

4.2.3 Sale

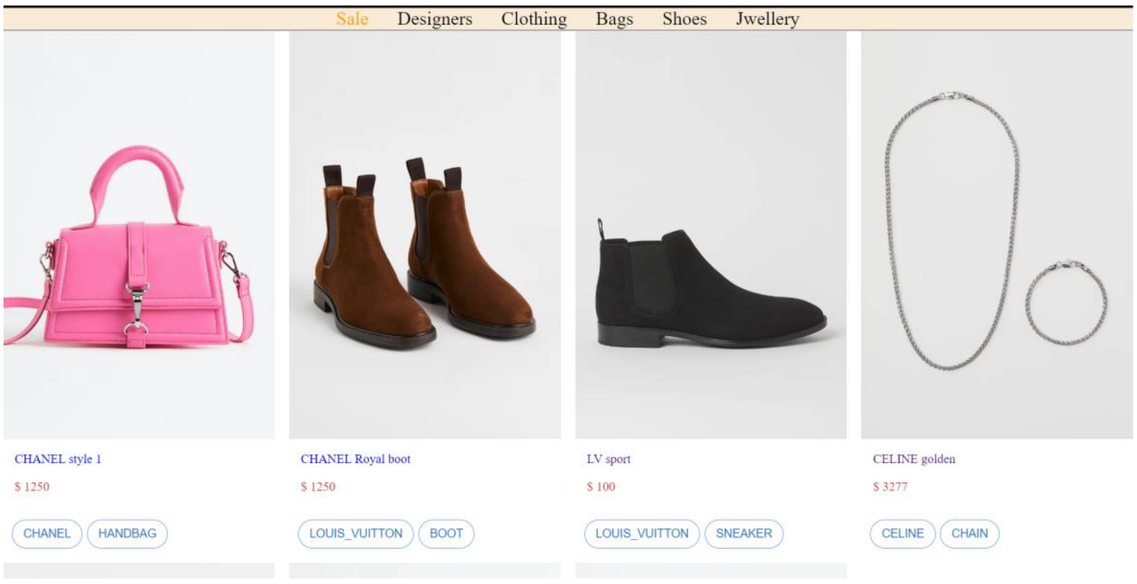
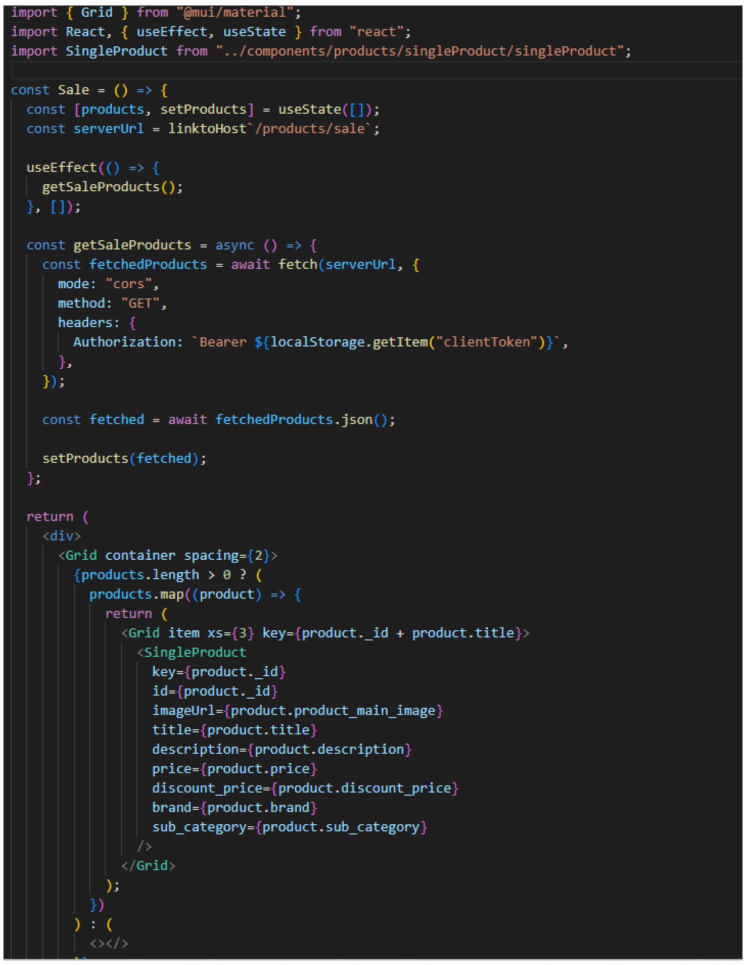


Figure 8. Sale page



```
import { Grid } from "@mui/material";
import React, { useEffect, useState } from "react";
import SingleProduct from "../components/products/singleProduct/singleProduct";

const Sale = () => {
  const [products, setProducts] = useState([]);
  const serverUrl = linktoHost`/products/sale`;

  useEffect(() => {
    getSaleProducts();
  }, []);

  const getSaleProducts = async () => {
    const fetchedProducts = await fetch(serverUrl, {
      mode: "cors",
      method: "GET",
      headers: {
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
    });

    const fetched = await fetchedProducts.json();

    setProducts(fetched);
  };

  return (
    <div>
      <Grid container spacing={2}>
        {products.length > 0 ? (
          products.map((product) => {
            return (
              <Grid item xs={3} key={product._id + product.title}>
                <SingleProduct
                  key={product._id}
                  id={product._id}
                  imageUrl={product.product_main_image}
                  title={product.title}
                  description={product.description}
                  price={product.price}
                  discount_price={product.discount_price}
                  brand={product.brand}
                  sub_category={product.sub_category}
                />
              </Grid>
            );
          })
        ) : (
          <></>
```

Figure 9. Sale page code base

When the ‘Sale’ view page is accessed, getSaleProducts() function is executed, to fetch discounted products from the endpoint ‘/products/sale’, the returned data is assigned to the stateful variable ‘products’, which will be used to visualize a singleProduct grid. ‘SingleProduct’ is a reusable component that represents a product layout with all the desired data to show, as represented in the code block below.

```
products.map((product) => {
            return (
              <Grid item xs={3} key={product._id + product.title}>
                <SingleProduct
                  key={product._id}
                  id={product._id}
                  imageUrl={product.product_main_image}
                  title={product.title}
                  description={product.description}
                  price={product.price}
                  discount_price={product.discount_price}
                  brand={product.brand}
                  sub_category={product.sub_category}
                />
              </Grid>
            );
          })
```

4.2.4 Designers

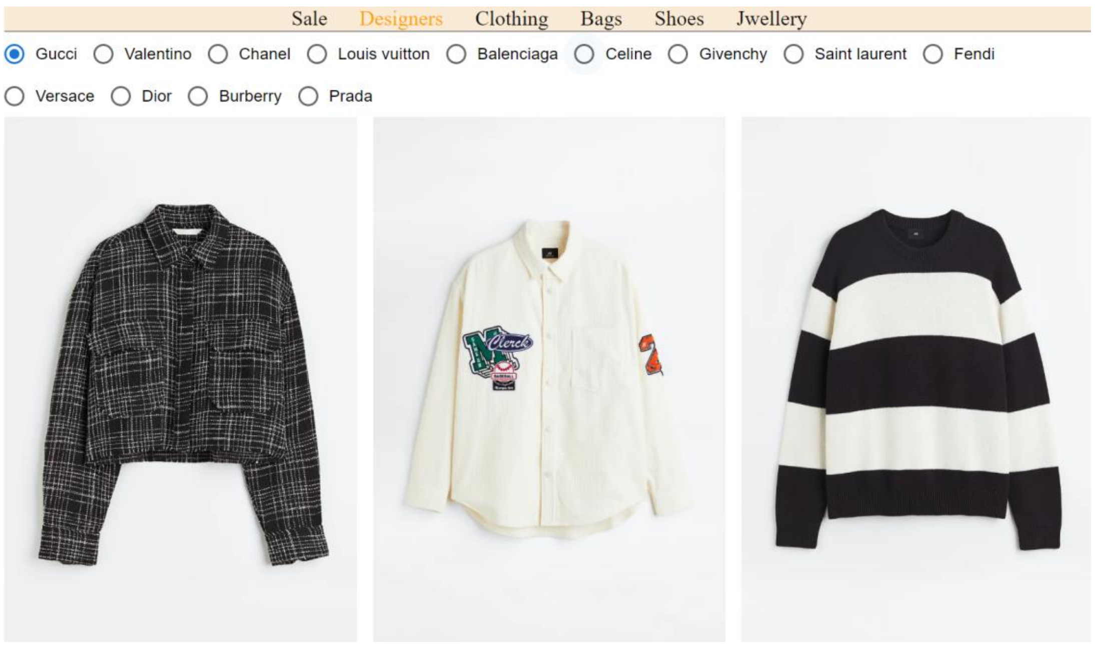
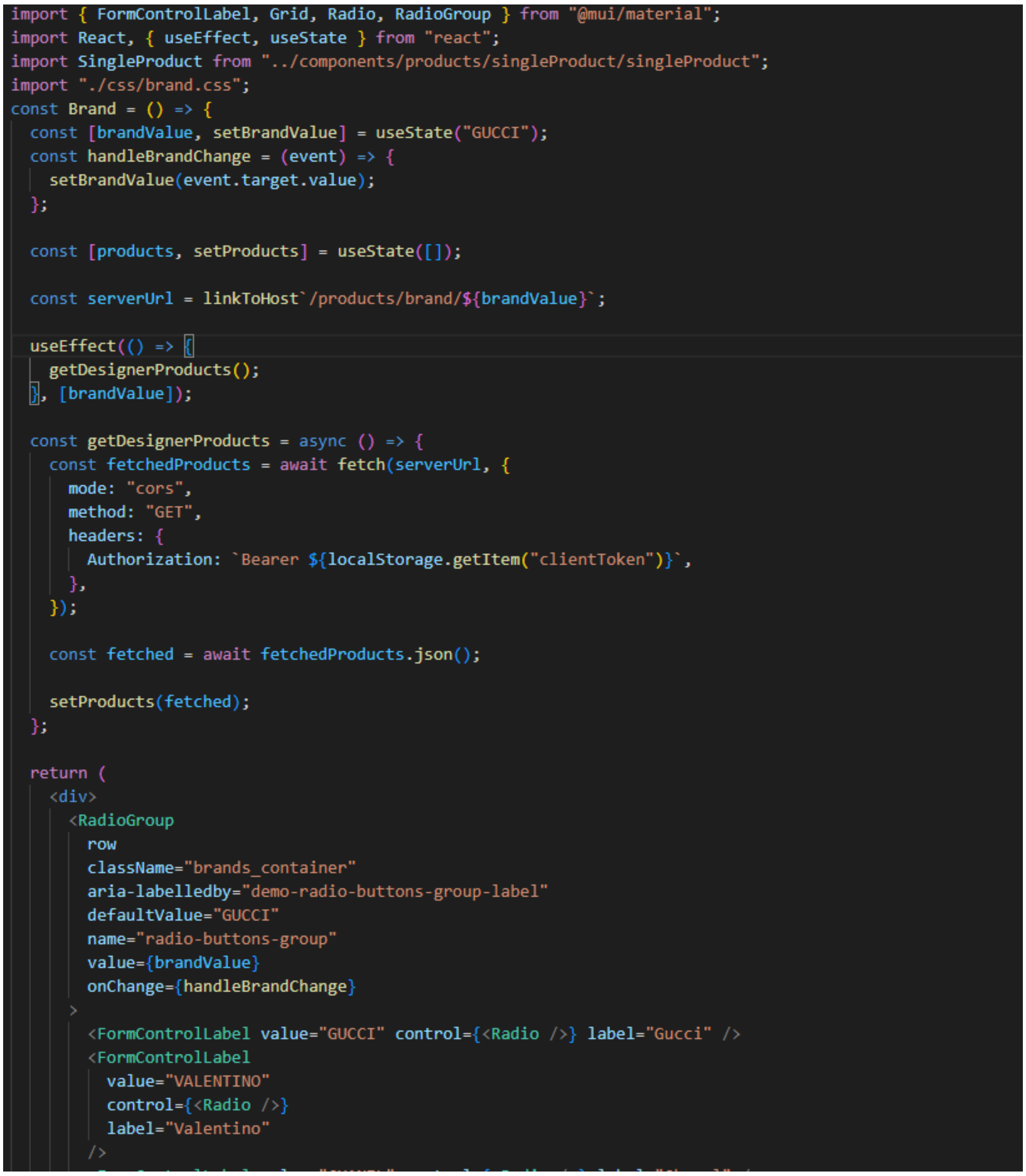

Sale
Designers
Clothing
Bags
Shoes
Jwellery
[✓ Gucci] Valentino Chanel Louis vuitton Balenciaga Celine Givenchy Saint laurent Fendi
Versace Dior Burberry Prada

Figure 10. Designer’s page

The designers page contains all the products brands supported by the store, as the client can select the desired designer from the top of the page, and the content of product list will be changed accordingly.

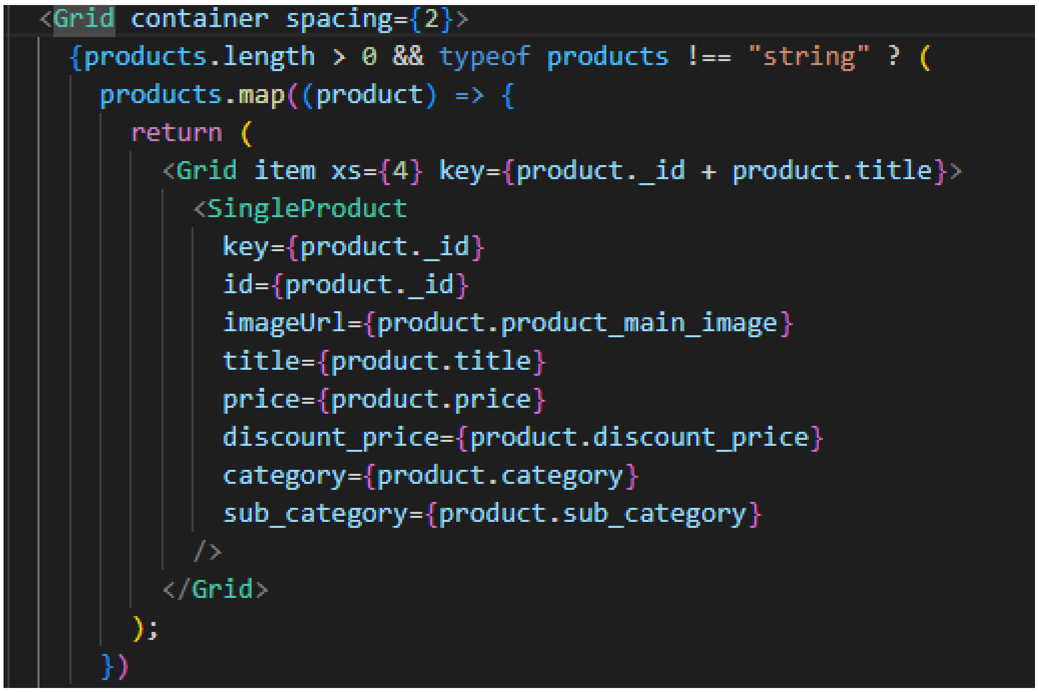


```
import { FormControlLabel, Grid, Radio, RadioGroup } from "@mui/material";
import React, { useEffect, useState } from "react";
import SingleProduct from "../components/products/singleProduct/singleProduct";
import "./css/brand.css";
const Brand = () => {
  const [brandValue, setBrandValue] = useState("GUCCI");
  const handleBrandChange = (event) => {
    setBrandValue(event.target.value);
  };

  const [products, setProducts] = useState([]);

  const serverUrl = linkToHost`/products/brand/${brandValue}`;

  useEffect(() => {
    getDesignerProducts();
  }, [brandValue]);

  const getDesignerProducts = async () => {
    const fetchedProducts = await fetch(serverUrl, {
      mode: "cors",
      method: "GET",
      headers: {
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
    });

    const fetched = await fetchedProducts.json();

    setProducts(fetched);
  };

  return (
    <div>
      <RadioGroup
        row
        className="brands_container"
        aria-labelledby="demo-radio-buttons-group-label"
        defaultValue="GUCCI"
        name="radio-buttons-group"
        value={brandValue}
        onChange={handleBrandChange}
      >
        <FormControlLabel value="GUCCI" control={<Radio />} label="Gucci" />
        <FormControlLabel
          value="VALENTINO"
          control={<Radio />}
          label="Valentino"
        />
```

Figure 11. Designer page code base

As shown in the figure above, ‘getDesignerProducts()’ function is responsible for fetching the selected product brands by calling the GET method to the endpoint ‘/products/brand/${brandValue}’, where the brandValue instance is the selected brand by the client.

After a successful api call, the fetched products are assigned to the local variable ‘products’ which will be passed to ‘SingleProduct’ component in order to display the product data for the client on the UI.

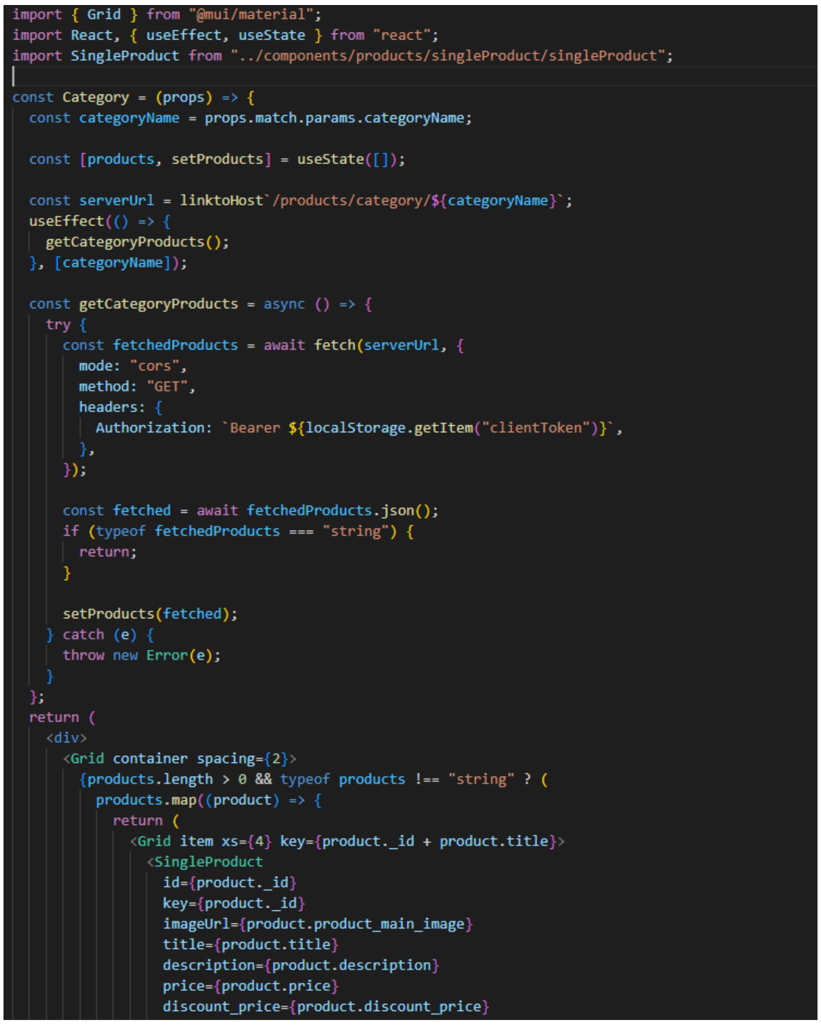


```
<Grid container spacing={2}>
  {products.length > 0 && typeof products !== "string" ? (
    products.map((product) => {
      return (
        <Grid item xs={4} key={product._id + product.title}>
          <SingleProduct
            key={product._id}
            id={product._id}
            imageUrl={product.product_main_image}
            title={product.title}
            price={product.price}
            discount_price={product.discount_price}
            category={product.category}
            sub_category={product.sub_category}
          />
        </Grid>
      );
    })
```

Figure 12. Usage of SingleProduct component inside designer page

4.2.5 Categories

Categories pages can be visualized as separate pages of clothes, bags, shoes, and jewellery containing different types of products in order to give the client a better user experience on an online store, instead of mixing every type and category in one page, which will lead to very low interest and lack of decision making when purchasing [12].

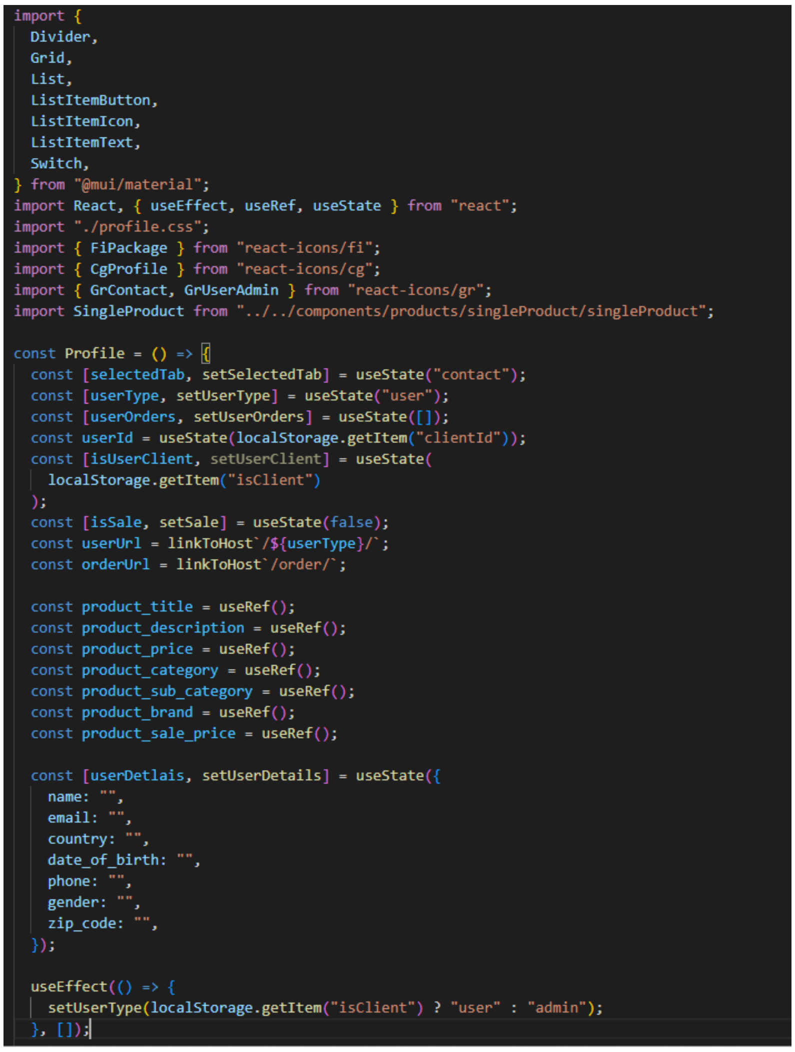


```
import { Grid } from "@mui/material";
import React, { useEffect, useState } from "react";
import SingleProduct from "../components/products/singleProduct/singleProduct";

const Category = (props) => {
  const categoryName = props.match.params.categoryName;

  const [products, setProducts] = useState([]);

  const serverUrl = linktoHost`/products/category/${categoryName}`;
  useEffect(() => {
    getCategoryProducts();
  }, [categoryName]);

  const getCategoryProducts = async () => {
    try {
      const fetchedProducts = await fetch(serverUrl, {
        mode: "cors",
        method: "GET",
        headers: {
          Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
        },
      });

      const fetched = await fetchedProducts.json();
      if (typeof fetchedProducts === "string") {
        return;
      }

      setProducts(fetched);
    } catch (e) {
      throw new Error(e);
    }
  };
  return (
    <div>
      <Grid container spacing={2}>
        {products.length > 0 && typeof products !== "string" ? (
          products.map((product) => {
            return (
              <Grid item xs={4} key={product._id + product.title}>
                <SingleProduct
                  id={product._id}
                  key={product._id}
                  imageUrl={product.product_main_image}
                  title={product.title}
                  description={product.description}
                  price={product.price}
                  discount_price={product.discount_price}
```

Figure 13.Category page code base

All the four different categories (Clothes, Shoes, Bags and Jewellery are implementing the same code base, in which ‘getCategoryProducts()’ function is responsible for calling a get Method for the endpoint ‘/category/categoryName’, where the ‘categoryName’ is the parameter passed from the navigation by calling ‘props.match.params.categoryName’, which can be either ‘clothing’, ‘shoes’, ‘bags’ or ‘jewellery’. The entity ‘products will hold all the fetched products and will be passed for the common component ‘SingleProducts’.

4.2.6 Profile



```
import {
  Divider,
  Grid,
  List,
  ListItemButton,
  ListItemIcon,
  ListItemText,
  Switch,
} from "@mui/material";
import React, { useEffect, useRef, useState } from "react";
import "./profile.css";
import { FiPackage } from "react-icons/fi";
import { CgProfile } from "react-icons/cg";
import { GrContact, GrUserAdmin } from "react-icons/gr";
import SingleProduct from "../../components/products/singleProduct/singleProduct";

const Profile = () => {
  const [selectedTab, setSelectedTab] = useState("contact");
  const [userType, setUserType] = useState("user");
  const [userOrders, setUserOrders] = useState([]);
  const userId = useState(localStorage.getItem("clientId"));
  const [isUserClient, setUserClient] = useState(
    localStorage.getItem("isClient")
  );
  const [isSale, setSale] = useState(false);
  const userUrl = linkToHost`/${userType}/`;
  const orderUrl = linkToHost`/order/`;

  const product_title = useRef();
  const product_description = useRef();
  const product_price = useRef();
  const product_category = useRef();
  const product_sub_category = useRef();
  const product_brand = useRef();
  const product_sale_price = useRef();

  const [userDetlais, setUserDetails] = useState({
    name: "",
    email: "",
    country: "",
    date_of_birth: "",
    phone: "",
    gender: "",
    zip_code: "",
  });

  useEffect(() => {
    setUserType(localStorage.getItem("isClient") ? "user" : "admin");
  }, []);
```

Figure 14. Variables declaration in the Profile page

As shown in the figure above, multiple variables have been declared in the beginning of the ‘Profile’ function.

The Profile page will be customized based on the type of ‘isClient’ storage entity that has been initialized after the authentication, meaning that if the user has successfully authenticated as an admin, the ‘isClient’ storage entity will hold a ‘false’ value otherwise it will be ‘true’, and this value will be assigned to the local variable ‘isUserClient’ in the ‘Profile’ page by calling ‘localStorage.getItem(‘isClient’)’.

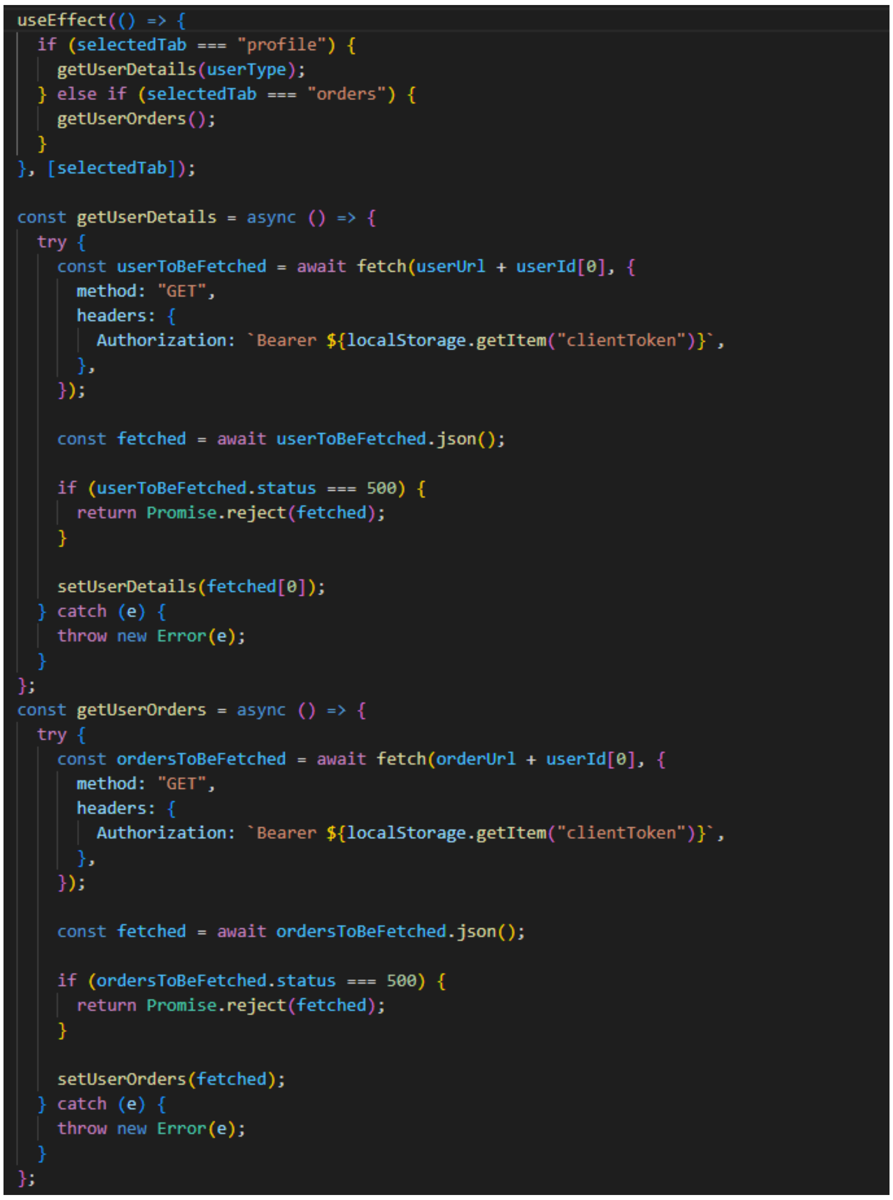
‘selectedTab’ will hold the value of the current selected tab that can be either:

- ‘admin’ tab, which will be visible only if ‘isClient’ is false, and when selected, the admin can add new products and other admins too.

- ‘contact’ tab, which will be holding the general information about the online store and how to get in touch with admins.

- ‘profile’ tab, which will be visible only if ‘isClient’ is true, meaning that the profile tab is showing the general information of the current client but not admin’s information.

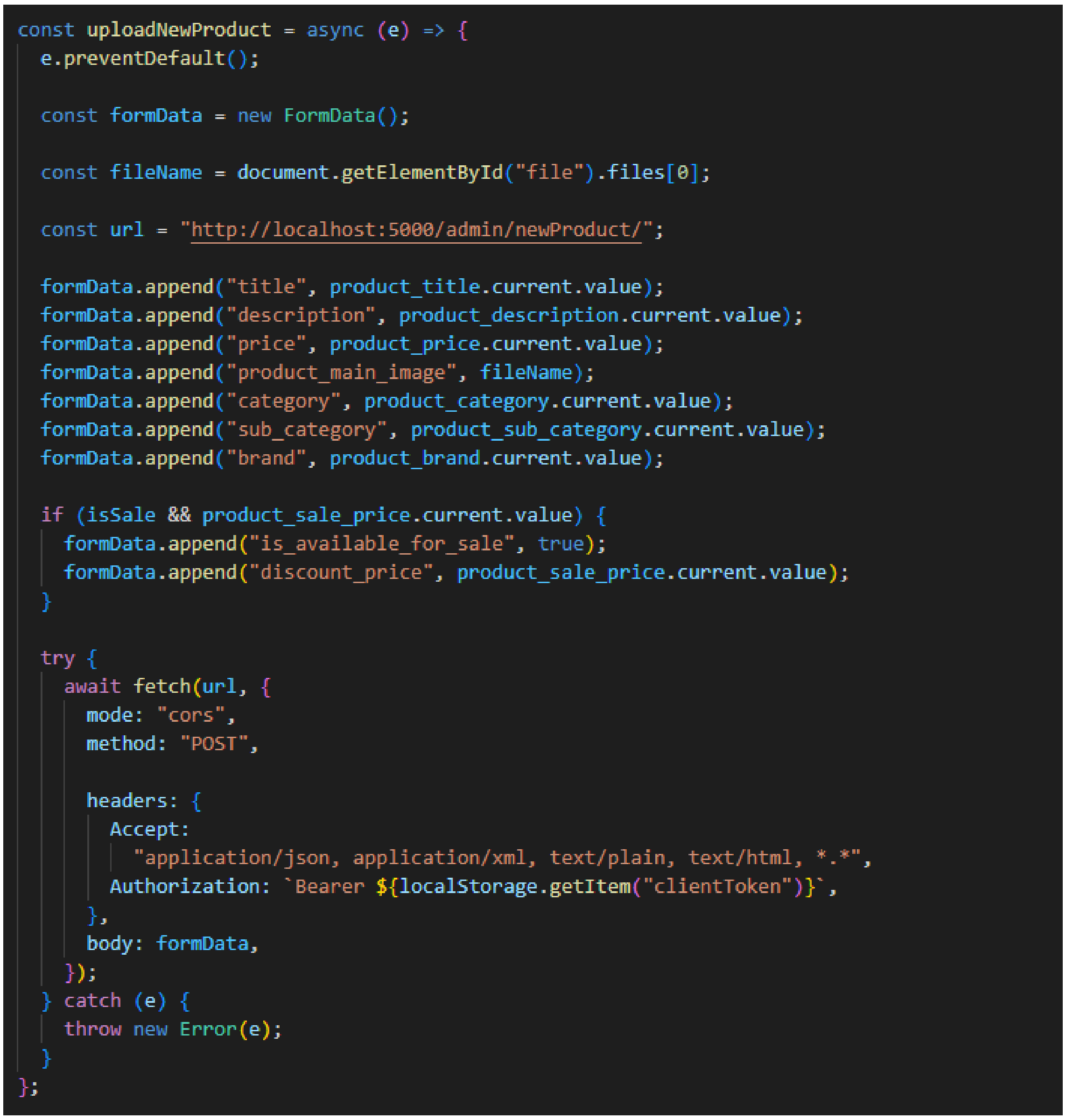
- ‘order’ tab, which will be visible only if ‘isClient’ is true, where the client can visualize his orders history.



```
useEffect(() => {
  if (selectedTab === "profile") {
    getUserDetails(userType);
  } else if (selectedTab === "orders") {
    getUserOrders();
  }
}, [selectedTab]);

const getUserDetails = async () => {
  try {
    const userToBeFetched = await fetch(userUrl + userId[0], {
      method: "GET",
      headers: {
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
    });

    const fetched = await userToBeFetched.json();

    if (userToBeFetched.status === 500) {
      return Promise.reject(fetched);
    }

    setUserDetails(fetched[0]);
  } catch (e) {
    throw new Error(e);
  }
};
const getUserOrders = async () => {
  try {
    const ordersToBeFetched = await fetch(orderUrl + userId[0], {
      method: "GET",
      headers: {
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
    });

    const fetched = await ordersToBeFetched.json();

    if (ordersToBeFetched.status === 500) {
      return Promise.reject(fetched);
    }

    setUserOrders(fetched);
  } catch (e) {
    throw new Error(e);
  }
};
```

Figure 15. Implementation of getting orders and user information

As shown in the figure above, the useEffect hook will be listening to ‘selectedTab’ value throughout the lifecycle of the ‘Profile’ page. One case is when the ‘selectedTab’ has a value of ‘orders’, the ‘getUserOrders()’ function will be called, which fetches all the order history of the current user and assigns the returned data to ‘useOrders’ state variable that will be rendered later and passed to ‘SingleProduct’ component. The other case is when the ‘selectedTab’ has a value of ‘profile’, the ‘getUserDetails()’ function will be called, and the fetched client data will be assigned to ‘userDetails’ state variable.



```javascript
const uploadNewProduct = async (e) => {
  e.preventDefault();

  const formData = new FormData();

  const fileName = document.getElementById("file").files[0];

  const url = "http://localhost:5000/admin/newProduct/";

  formData.append("title", product_title.current.value);
  formData.append("description", product_description.current.value);
  formData.append("price", product_price.current.value);
  formData.append("product_main_image", fileName);
  formData.append("category", product_category.current.value);
  formData.append("sub_category", product_sub_category.current.value);
  formData.append("brand", product_brand.current.value);

  if (isSale && product_sale_price.current.value) {
    formData.append("is_available_for_sale", true);
    formData.append("discount_price", product_sale_price.current.value);
  }

  try {
    await fetch(url, {
      mode: "cors",
      method: "POST",

      headers: {
        Accept:
          "application/json, application/xml, text/plain, text/html, *.*",
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
      body: formData,
    });
  } catch (e) {
    throw new Error(e);
  }
};
```

Figure 16. Upload new Product function.

As shown in the above figure, the upload product function will be called in the product panel for admins only, where the endpoint ‘/newProduct’ is used with POST method.

4.3 Payment gateway

As the e-commerce represents an online version of the traditional shop, there should be a way to process and accept payments from clients. And in this application, we will be using Stripe checkout.

Stripe is a payment service company providing payment solutions to merchants to accept payments on behalf of the sellers, by minimizing the burden of settling official agreement with banks to process payment internationally, Also Stripe made it easy to integrate their checkout process system on custom websites, easily by adding the stripe component to the html. [13, pp. 61-62.]

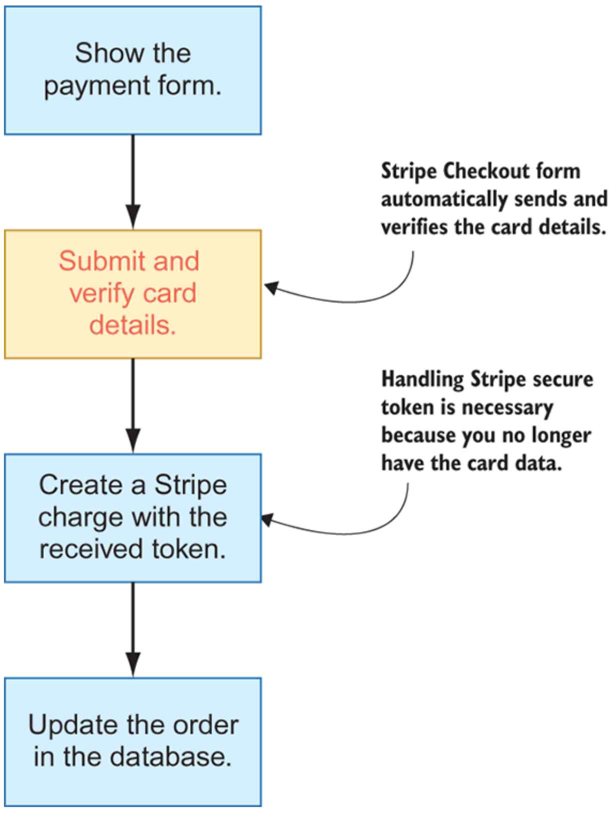
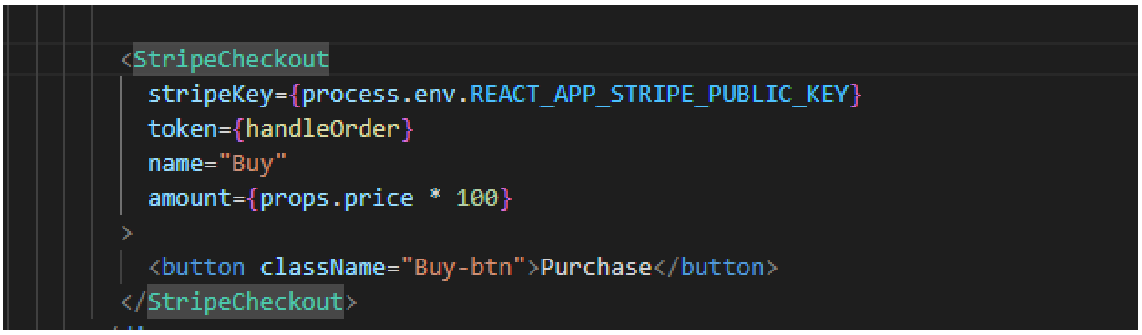

Show the payment form.
Submit and verify card details.
Create a Stripe charge with the received token.
Update the order in the database.
Stripe Checkout form automatically sends and verifies the card details.
Handling Stripe secure token is necessary because you no longer have the card data.

Figure 17. stripe checkout process

As shown in the figure above, instead of collecting client’s data locally, the stripe checkout component will handle all the sensitive data, as we only need to show the stripe form where the user enters his personal name and credit card.

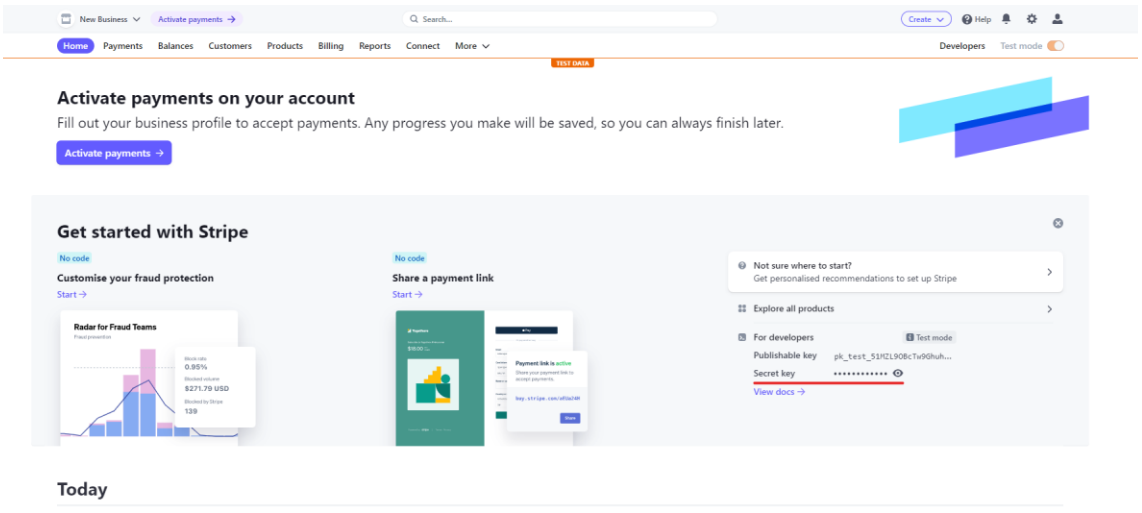


```
<StripeCheckout
  stripeKey={process.env.REACT_APP_STRIPE_PUBLIC_KEY}
  token={handleOrder}
  name="Buy"
  amount={props.price * 100}
>
  <button className="Buy-btn">Purchase</button>
</StripeCheckout>
```

Figure 18. Stripe checkout react component implementation.

As shown in the figure above, the ‘stripeCheckout’ component needs two mandatory properties:

- stripeKey: represents the stripe public key, which is a unique string that can be found on the merchant dashboard on stripe.com as shown in the below figure.
- token: is a function type property that can handle the result of the payment checkout, and in this case we are calling handleOrder() function.



Activate payments on your account

Fill out your business profile to accept payments. Any progress you make will be saved, so you can always finish later.

Get started with Stripe

Not sure where to start?
Get personalised recommendations to set up Stripe

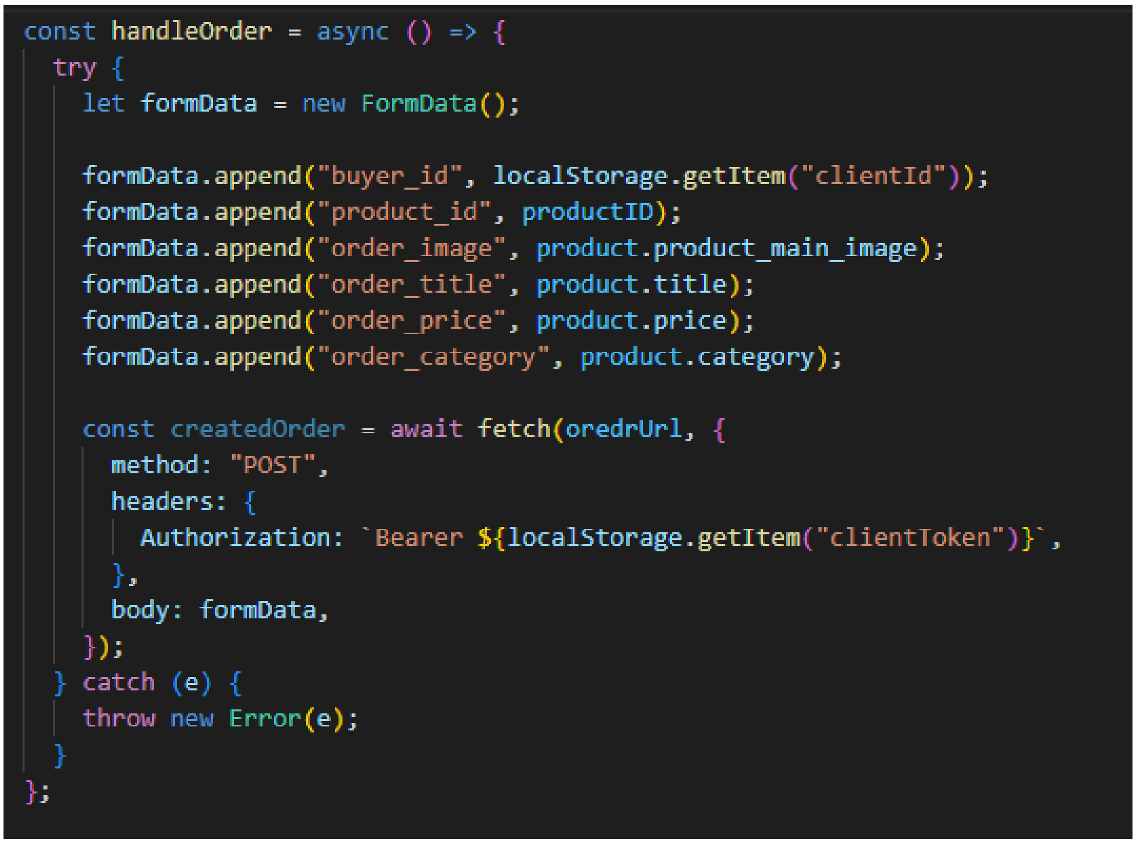
Explore all products

For developers — Test mode

Publishable key pk_test_51MZL90BcTw9Ghuh...

Secret key ••••••••••••

Figure 19. Merchant stripe dashboard.



```
const handleOrder = async () => {
  try {
    let formData = new FormData();

    formData.append("buyer_id", localStorage.getItem("clientId"));
    formData.append("product_id", productID);
    formData.append("order_image", product.product_main_image);
    formData.append("order_title", product.title);
    formData.append("order_price", product.price);
    formData.append("order_category", product.category);

    const createdOrder = await fetch(oredrUrl, {
      method: "POST",
      headers: {
        Authorization: `Bearer ${localStorage.getItem("clientToken")}`,
      },
      body: formData,
    });
  } catch (e) {
    throw new Error(e);
  }
};
```

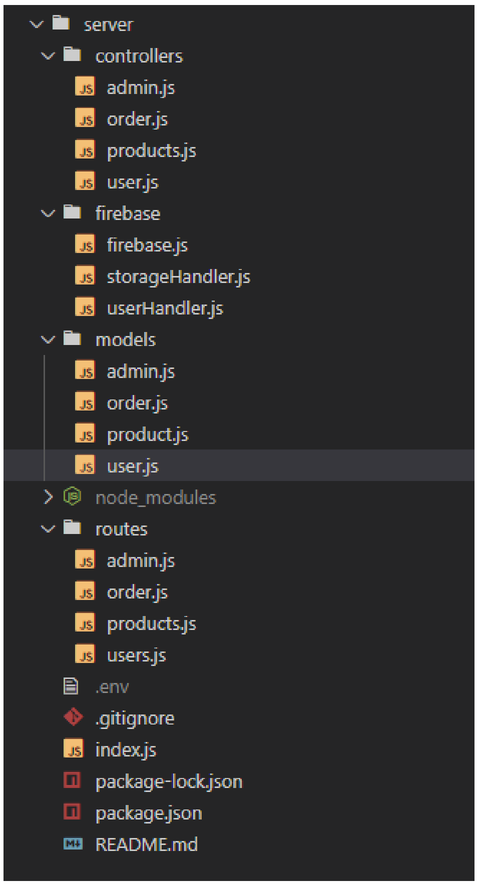
Figure 20. Add new order function.

The above figure represents the handleOrder function which is called as a result of a successful payment processing, the '/ order/create' endpoint with POST method accepts multiple parameters related to the buyer and the bought product as buyer_id, product_id, order_image, order_title, order_price and order_category.

5 Backend side

In general, the backend in software development is the hidden part that the application user cannot see or interact with, as it is the data access layer that the frontend connects to, in order to make API calls in a form of sending or receiving the data to be displayed for the user on the UI.

5.1 Project structure



```
server
  controllers
    admin.js
    order.js
    products.js
    user.js
  firebase
    firebase.js
    storageHandler.js
    userHandler.js
  models
    admin.js
    order.js
    product.js
    user.js
  node_modules
  routes
    admin.js
    order.js
    products.js
    users.js
  .env
  .gitignore
  index.js
  package-lock.json
  package.json
  README.md
```

Figure 21. Backend project files structure

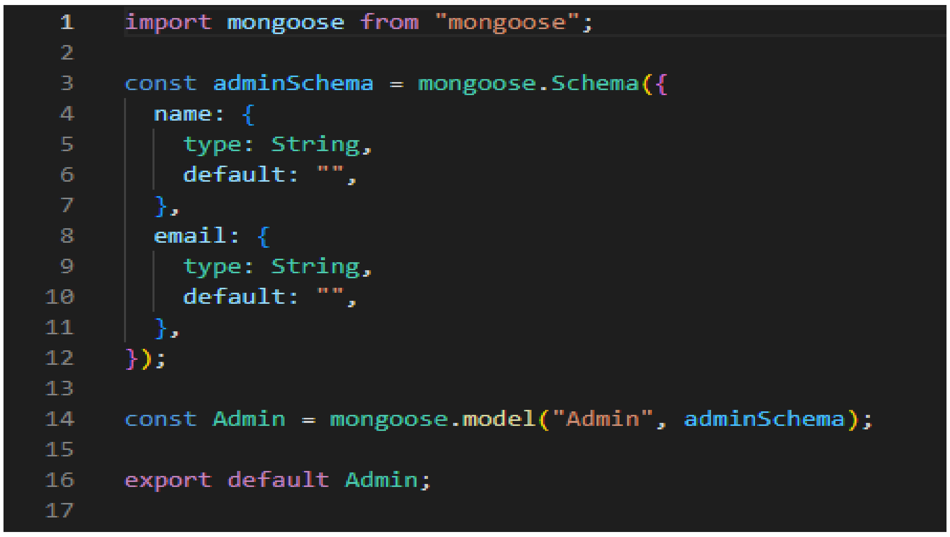
As figure 21 represents the root folder “server” containing a package.json which holds the project metadata and handles the project’s dependencies. Package-lock.json, that is automatically generated when installing npm dependencies from the terminal. .gitignore that specifies what should be ignored when committing the project changes to git. .env is the file that controls the application environment constants and stores sensitive code that cannot be exposed on git. Index.js is the entry file that will be accessed to run our application. Then we do have routes folder which will hold our defined routes related to products, user, admin and orders endpoints that can be accessed from our frontend. Also, the model’s folder that includes mongoose schema files for defining the structure of the documents on the Mongodb database. Next, we have firebase folder, which will contain all related files that have access and interaction with the firebase console, and lastly we have the controller’s folder that are responsible for manipulating the mongoose models, write and read the data received from the routes.

5.2 Mongoose models

Mongoose models allow to access data by providing an interface to MongoDB. The first step to creating a model is defining the schema for it. Afterwards, we'll need to register the model with Mongoose and export it so that we can use it throughout our application.

5.2.1 Admin model

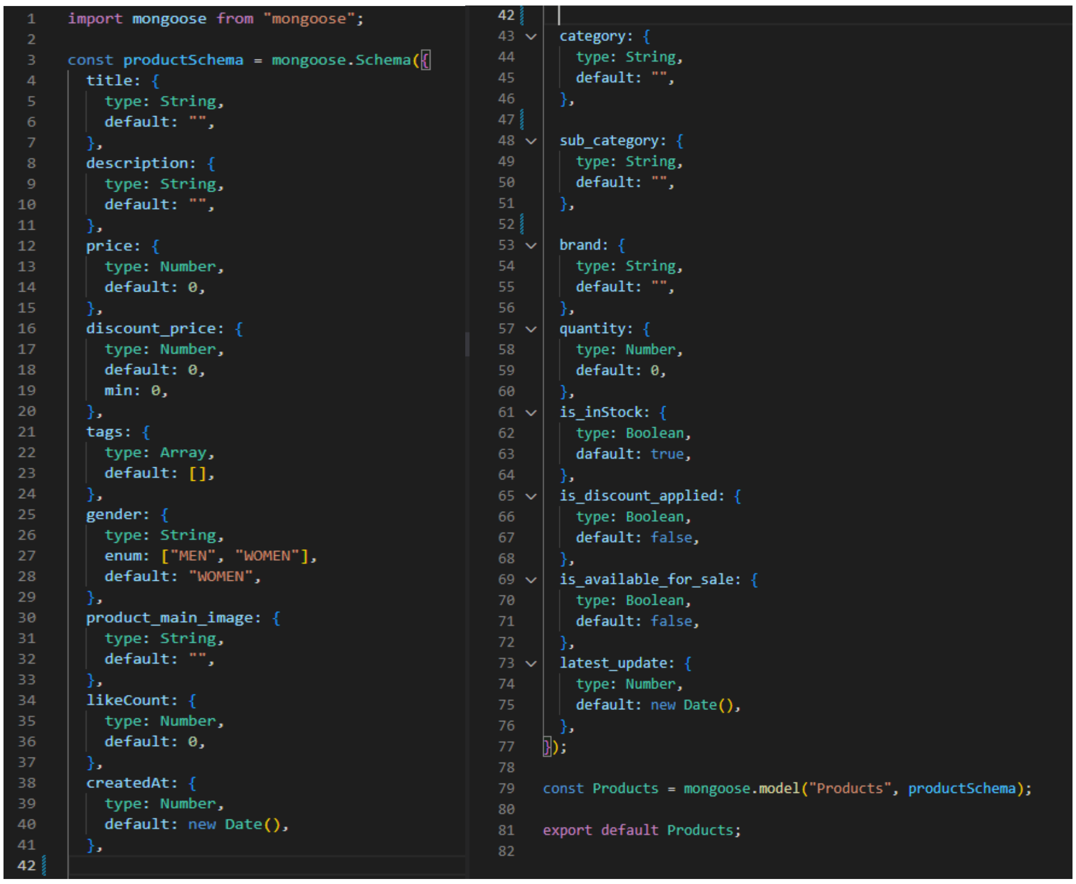
Figure 22 as shown below represents the Admin model, whereas schema defines the structure of the document and by calling Mongoose.schema we will create properties ‘name’ and ‘email’ both of type String and default value “”.



```
import mongoose from "mongoose";

const adminSchema = mongoose.Schema({
  name: {
    type: String,
    default: "",
  },
  email: {
    type: String,
    default: "",
  },
});

const Admin = mongoose.model("Admin", adminSchema);

export default Admin;

```

Figure 22. Admin model

5.2.2 Product model

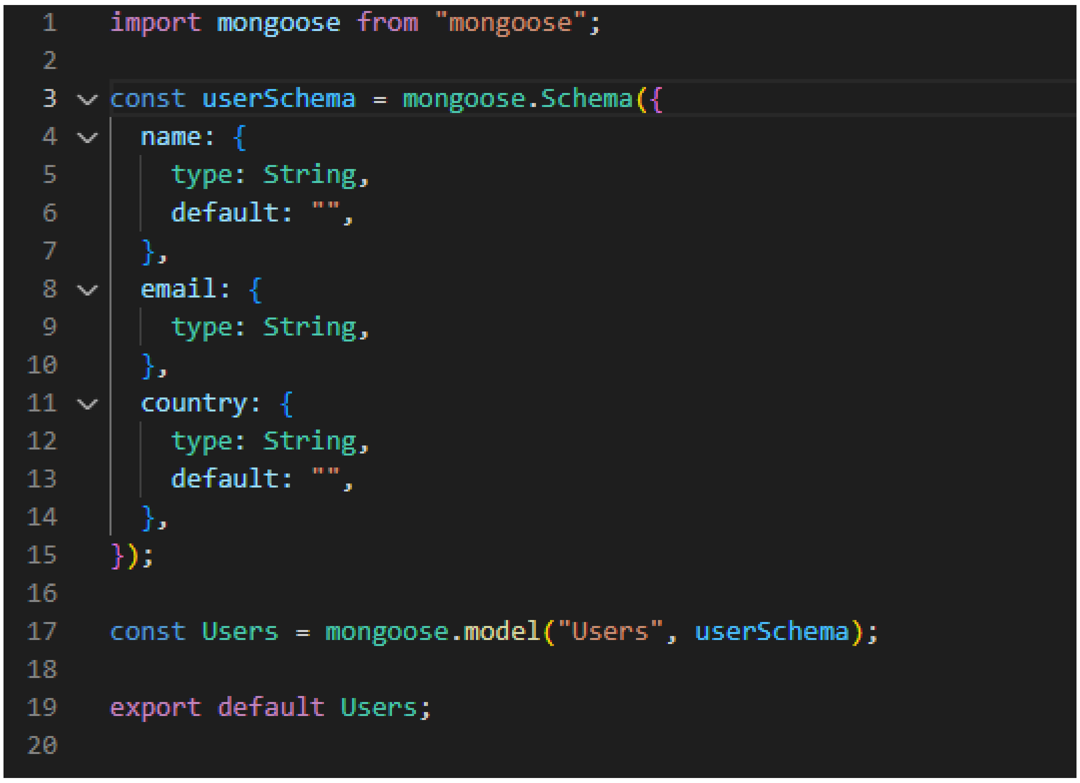


```javascript
import mongoose from "mongoose";

const productSchema = mongoose.Schema({
  title: {
    type: String,
    default: "",
  },
  description: {
    type: String,
    default: "",
  },
  price: {
    type: Number,
    default: 0,
  },
  discount_price: {
    type: Number,
    default: 0,
    min: 0,
  },
  tags: {
    type: Array,
    default: [],
  },
  gender: {
    type: String,
    enum: ["MEN", "WOMEN"],
    default: "WOMEN",
  },
  product_main_image: {
    type: String,
    default: "",
  },
  likeCount: {
    type: Number,
    default: 0,
  },
  createdAt: {
    type: Number,
    default: new Date(),
  },

  category: {
    type: String,
    default: "",
  },

  sub_category: {
    type: String,
    default: "",
  },

  brand: {
    type: String,
    default: "",
  },
  quantity: {
    type: Number,
    default: 0,
  },
  is_inStock: {
    type: Boolean,
    dafault: true,
  },
  is_discount_applied: {
    type: Boolean,
    default: false,
  },
  is_available_for_sale: {
    type: Boolean,
    default: false,
  },
  latest_update: {
    type: Number,
    default: new Date(),
  },
});

const Products = mongoose.model("Products", productSchema);

export default Products;
```

Figure 23. Product model

Figure 23 represents the largest mongoose model in the project, as it holds multiple vital properties (title, description, price, discounted_price, quantity and product_main_image), then we find other optional values that will be used mainly for customizing search and filtering from the client side as tags, gender, category, sub_category, brand, is_available_for_sale. And lastly other properties that will be shown as a meta data of the product and holding extra details for the it as is_inStock, is_discount_applied, latest_update and createdAt.

5.2.3 User model

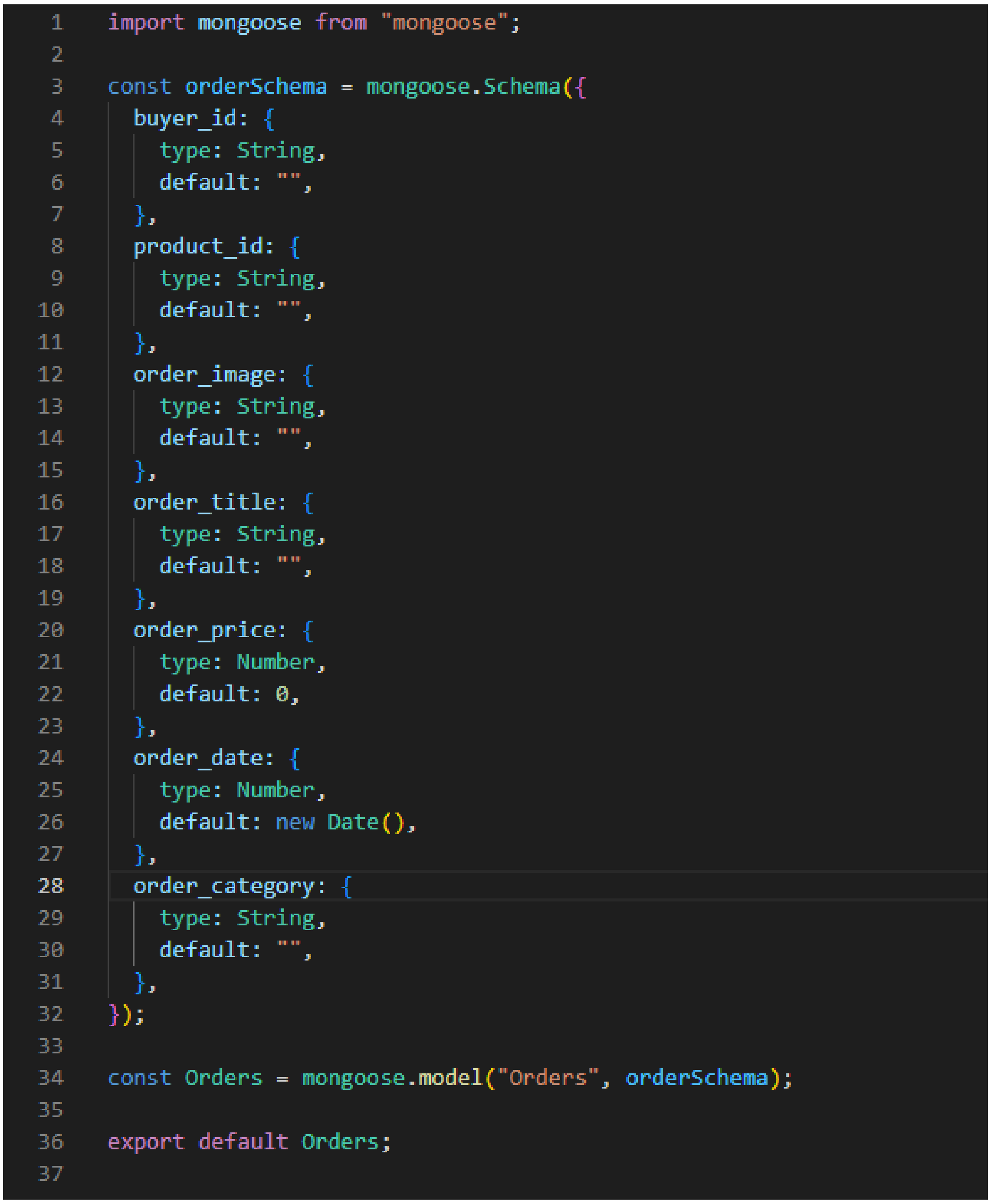


```javascript
import mongoose from "mongoose";

const userSchema = mongoose.Schema({
  name: {
    type: String,
    default: "",
  },
  email: {
    type: String,
  },
  country: {
    type: String,
    default: "",
  },
});

const Users = mongoose.model("Users", userSchema);

export default Users;

```

Figure 24. User model

As shown in figure 24 above, we are defining only three properties (name, email and country) then we export it as Users to be used in the user routes page. We have eliminated the password entity from the Users Schema for security reasons with mongoDb and only store it in Firebase that will handle the authentication for us.

5.2.4 Order model



```
import mongoose from "mongoose";

const orderSchema = mongoose.Schema({
  buyer_id: {
    type: String,
    default: "",
  },
  product_id: {
    type: String,
    default: "",
  },
  order_image: {
    type: String,
    default: "",
  },
  order_title: {
    type: String,
    default: "",
  },
  order_price: {
    type: Number,
    default: 0,
  },
  order_date: {
    type: Number,
    default: new Date(),
  },
  order_category: {
    type: String,
    default: "",
  },
});

const Orders = mongoose.model("Orders", orderSchema);

export default Orders;
```

Figure 25. Order model

Figure 25 represents the mongoose order schema which will hold various entities as the ’buyer_id’ that represents the user who made this order, also we find ‘product_id’, ‘order_image’, ‘order_title’, ‘order_price’, ‘order_date’ and ‘order_category’ all of the later entities that starts with the key-name ‘product’ are fetched from the bought product at the time of creating this order, as we will see in the order controller.

5.3 Controllers

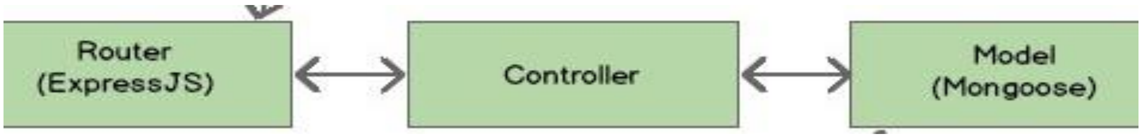
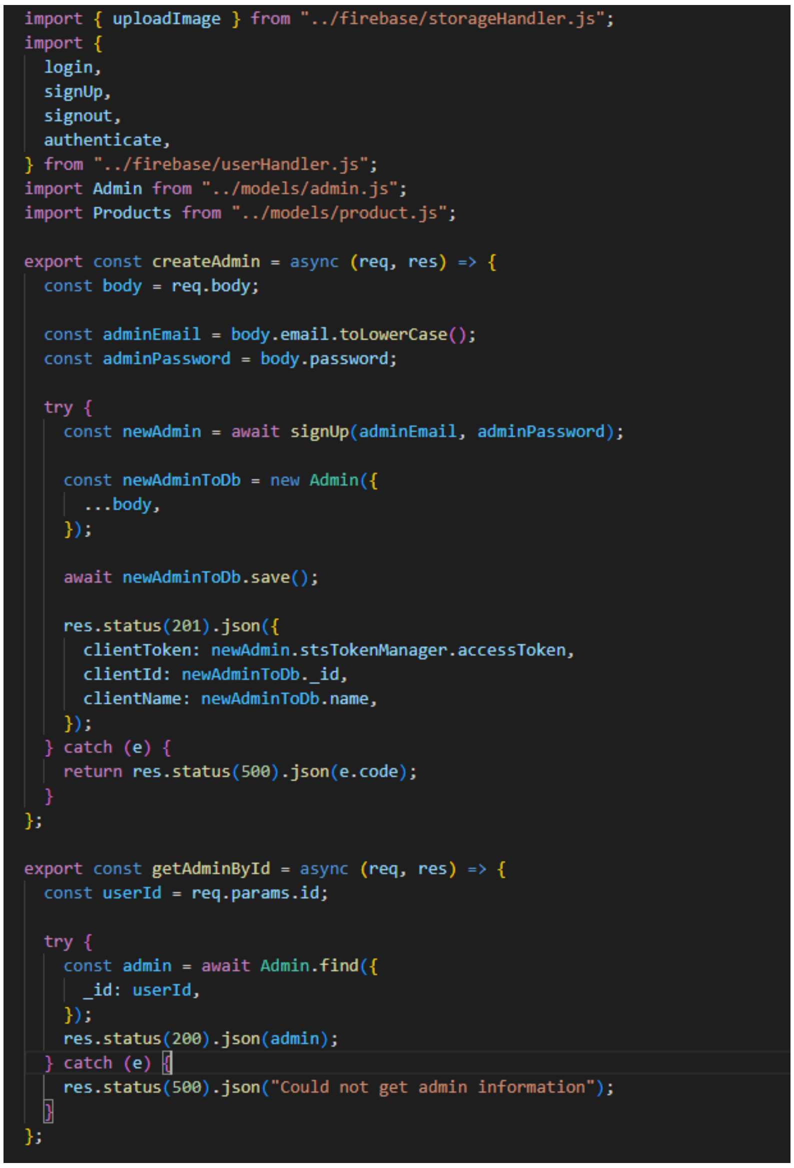


Figure 26. Relationship between model, router and controller

In mongoose, controller functions are called from routes to get the requested data from the corresponding model which starts the view renderer and manipulates the subjected model [14, p 91.]

5.3.1 Admin controller

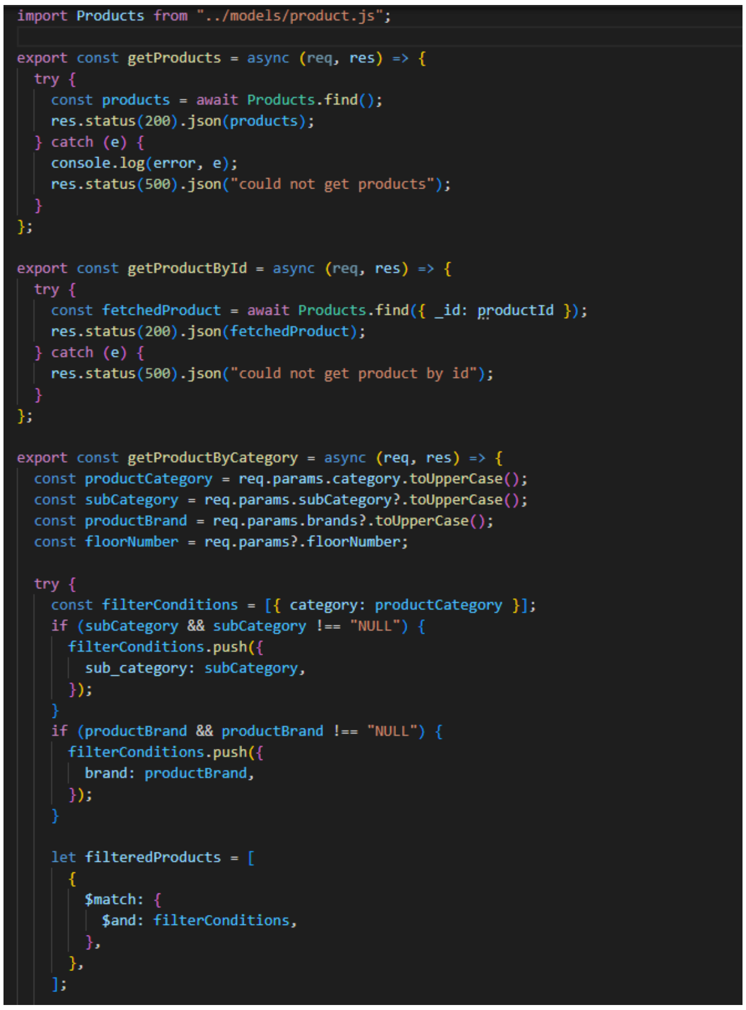


```
import { uploadImage } from "../firebase/storageHandler.js";
import {
  login,
  signUp,
  signout,
  authenticate,
} from "../firebase/userHandler.js";
import Admin from "../models/admin.js";
import Products from "../models/product.js";

export const createAdmin = async (req, res) => {
  const body = req.body;

  const adminEmail = body.email.toLowerCase();
  const adminPassword = body.password;

  try {
    const newAdmin = await signUp(adminEmail, adminPassword);

    const newAdminToDb = new Admin({
      ...body,
    });

    await newAdminToDb.save();

    res.status(201).json({
      clientToken: newAdmin.stsTokenManager.accessToken,
      clientId: newAdminToDb._id,
      clientName: newAdminToDb.name,
    });
  } catch (e) {
    return res.status(500).json(e.code);
  }
};

export const getAdminById = async (req, res) => {
  const userId = req.params.id;

  try {
    const admin = await Admin.find({
      _id: userId,
    });
    res.status(200).json(admin);
  } catch (e) {
    res.status(500).json("Could not get admin information");
  }
};
```

Figure 27. Admin controller

In the admin controller file, we imported multiple functions that will have a direct interaction with firebase functions as (login, signUp and authenticate). Then, we defined 4 methods to be called in the admin routes, as *createAdmin()* function, which is responsible for registering admins and saving admins data into Firebase then saves the forwarded data to mongoDb by calling *await newAdminToDb.save()*. Then we have loginAdmin function that does two-layer authentication by skimming through the admin table from Mongodb and find if the admin email is existing then tries to authenticate from firebase by passing the admin email and the password, as a success response we are returning access-token to be used later for secured API calls from the client side. We also have a *signOut()* function that logout the admin from the platform. And since products are only added by admins, we have *createProduct()* function that can be split into two parts, the first part is responsible for uploading the image file to firebase storage by using '*uploadImage()'* function imported from '*storageHandler.js'* file, which returns the hosted URL of the image which can be easily stored in mongoDB product model as a string into the '*product_main_image*' entity. Furthermore, we have '*updateProduct()'* function by using the integrated *findByIdAndUpdate* functionality from Mongoose.

5.3.2 Product controller

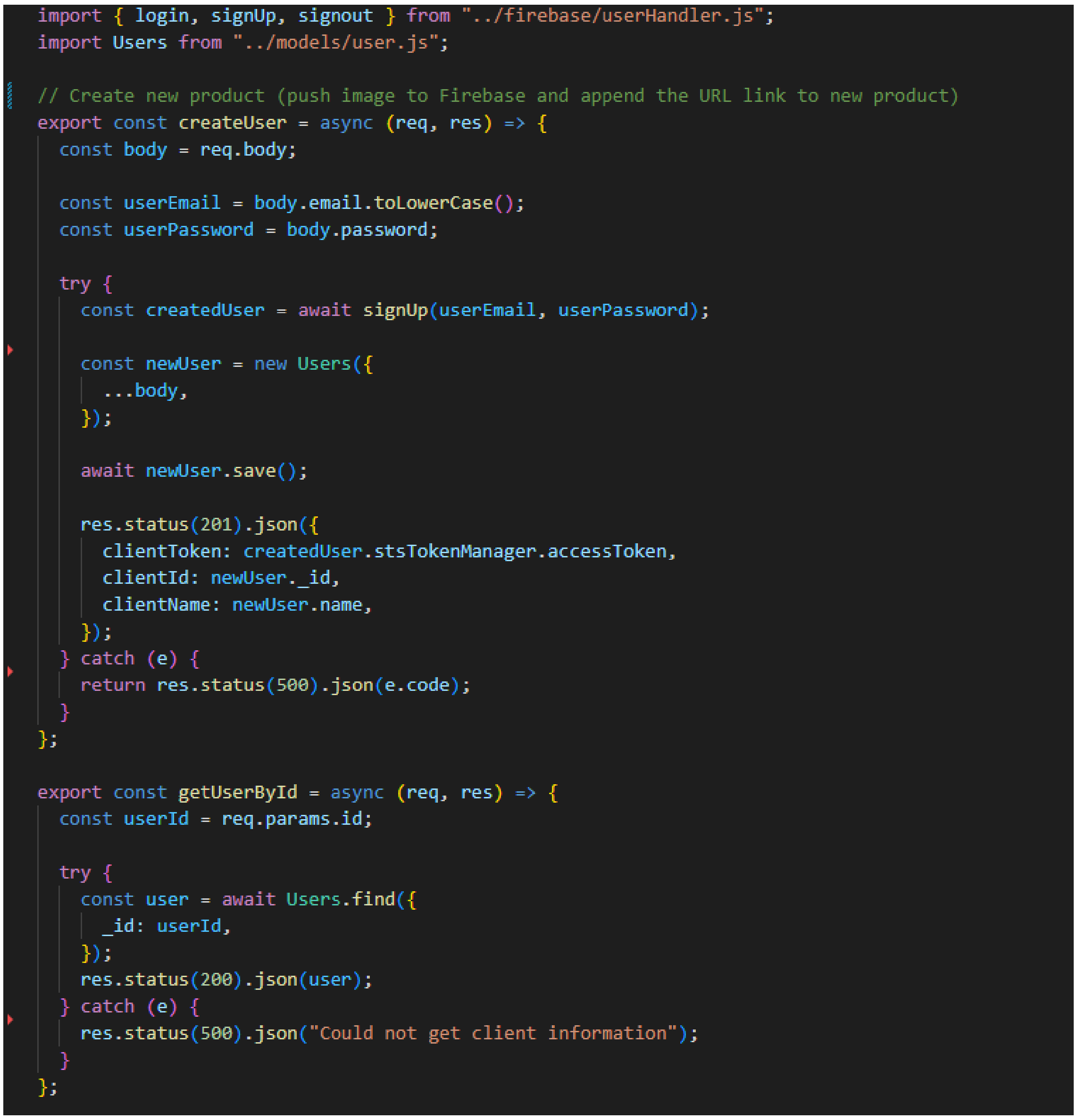


```
import Products from "../models/product.js";

export const getProducts = async (req, res) => {
  try {
    const products = await Products.find();
    res.status(200).json(products);
  } catch (e) {
    console.log(error, e);
    res.status(500).json("could not get products");
  }
};

export const getProductById = async (req, res) => {
  try {
    const fetchedProduct = await Products.find({ _id: productId });
    res.status(200).json(fetchedProduct);
  } catch (e) {
    res.status(500).json("could not get product by id");
  }
};

export const getProductByCategory = async (req, res) => {
  const productCategory = req.params.category.toUpperCase();
  const subCategory = req.params.subCategory?.toUpperCase();
  const productBrand = req.params.brands?.toUpperCase();
  const floorNumber = req.params?.floorNumber;

  try {
    const filterConditions = [{ category: productCategory }];
    if (subCategory && subCategory !== "NULL") {
      filterConditions.push({
        sub_category: subCategory,
      });
    }
    if (productBrand && productBrand !== "NULL") {
      filterConditions.push({
        brand: productBrand,
      });
    }

    let filteredProducts = [
      {
        $match: {
          $and: filterConditions,
        },
      },
    ];
```

Figure 28. Product controller

The product controller will be managing all the product related manipulations starting by getting all products and by id, then getting products by category by calling *getProductByCategory()* function which will have two optional fields for subcategory and product brand which will be useful for filtering. we also have other filtering functions as *getProductsByBrand()*, *getProductsBySubcategory()*, and get *productByGender()*.

5.3.3 User controller



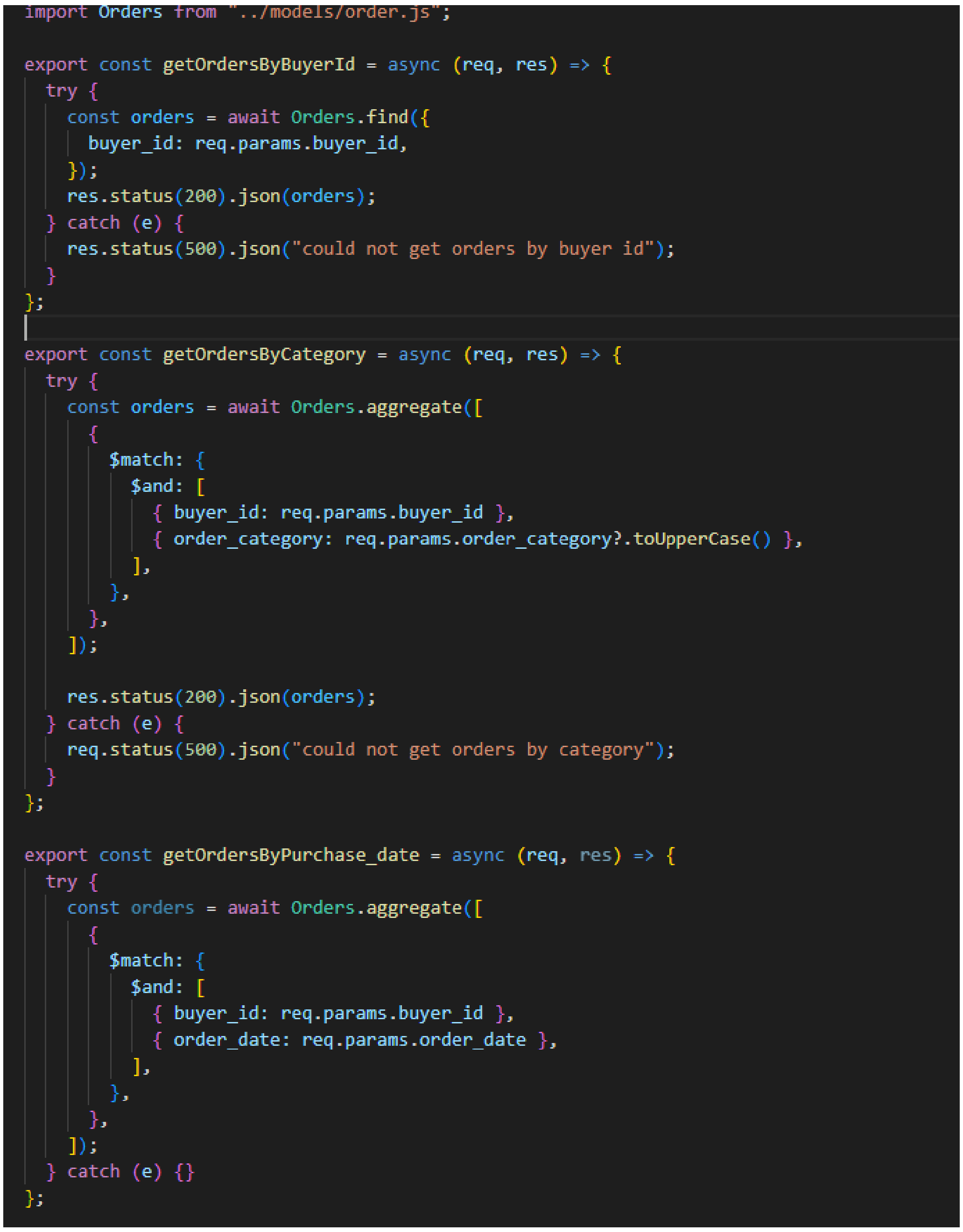
```
import { login, signUp, signout } from "../firebase/userHandler.js";
import Users from "../models/user.js";

// Create new product (push image to Firebase and append the URL link to new product)
export const createUser = async (req, res) => {
  const body = req.body;

  const userEmail = body.email.toLowerCase();
  const userPassword = body.password;

  try {
    const createdUser = await signUp(userEmail, userPassword);

    const newUser = new Users({
      ...body,
    });

    await newUser.save();

    res.status(201).json({
      clientToken: createdUser.stsTokenManager.accessToken,
      clientId: newUser._id,
      clientName: newUser.name,
    });
  } catch (e) {
    return res.status(500).json(e.code);
  }
};

export const getUserById = async (req, res) => {
  const userId = req.params.id;

  try {
    const user = await Users.find({
      _id: userId,
    });
    res.status(200).json(user);
  } catch (e) {
    res.status(500).json("Could not get client information");
  }
};
```

Figure 29. User controller

The user controller file has only three basic functions(*createUser*, *loginUser* and *signOut*) that will be used for the normal buyer as a way of separation from admins.

In the case of *Login* and *Signup* , the Firebase integrated functionalities such as *login* and *register* are used to first authenticate the user on the Firebase layer which return a token that will be used to authenticate the current user API calls throughout the lifecycle of the application, then authenticating the user on the mongoDB layer which return all information regarding the user to be used in the profile page.

5.3.4 Order controller



```
import Orders from "../models/order.js";

export const getOrdersByBuyerId = async (req, res) => {
  try {
    const orders = await Orders.find({
      buyer_id: req.params.buyer_id,
    });
    res.status(200).json(orders);
  } catch (e) {
    res.status(500).json("could not get orders by buyer id");
  }
};

export const getOrdersByCategory = async (req, res) => {
  try {
    const orders = await Orders.aggregate([
      {
        $match: {
          $and: [
            { buyer_id: req.params.buyer_id },
            { order_category: req.params.order_category?.toUpperCase() },
          ],
        },
      },
    ]);

    res.status(200).json(orders);
  } catch (e) {
    req.status(500).json("could not get orders by category");
  }
};

export const getOrdersByPurchase_date = async (req, res) => {
  try {
    const orders = await Orders.aggregate([
      {
        $match: {
          $and: [
            { buyer_id: req.params.buyer_id },
            { order_date: req.params.order_date },
          ],
        },
      },
    ]);
  } catch (e) {}
};
```

Figure 30. Order controller

Figure 30 represents the order controller which has a *createOrder()* function that populates the order array with the data received from the request and then saves it into the order table in MongoDB by calling *newOrder.save()*. Then we find a function called *getOrdersByBuyerID* which is responsible of finding all the orders in the order table in MongoDB by the matching *buyerID* passed in the get request. Furthermore, we can find to filtering order functions as *getOrdersByCategory* and *getOrdersByPurchaseDate*.

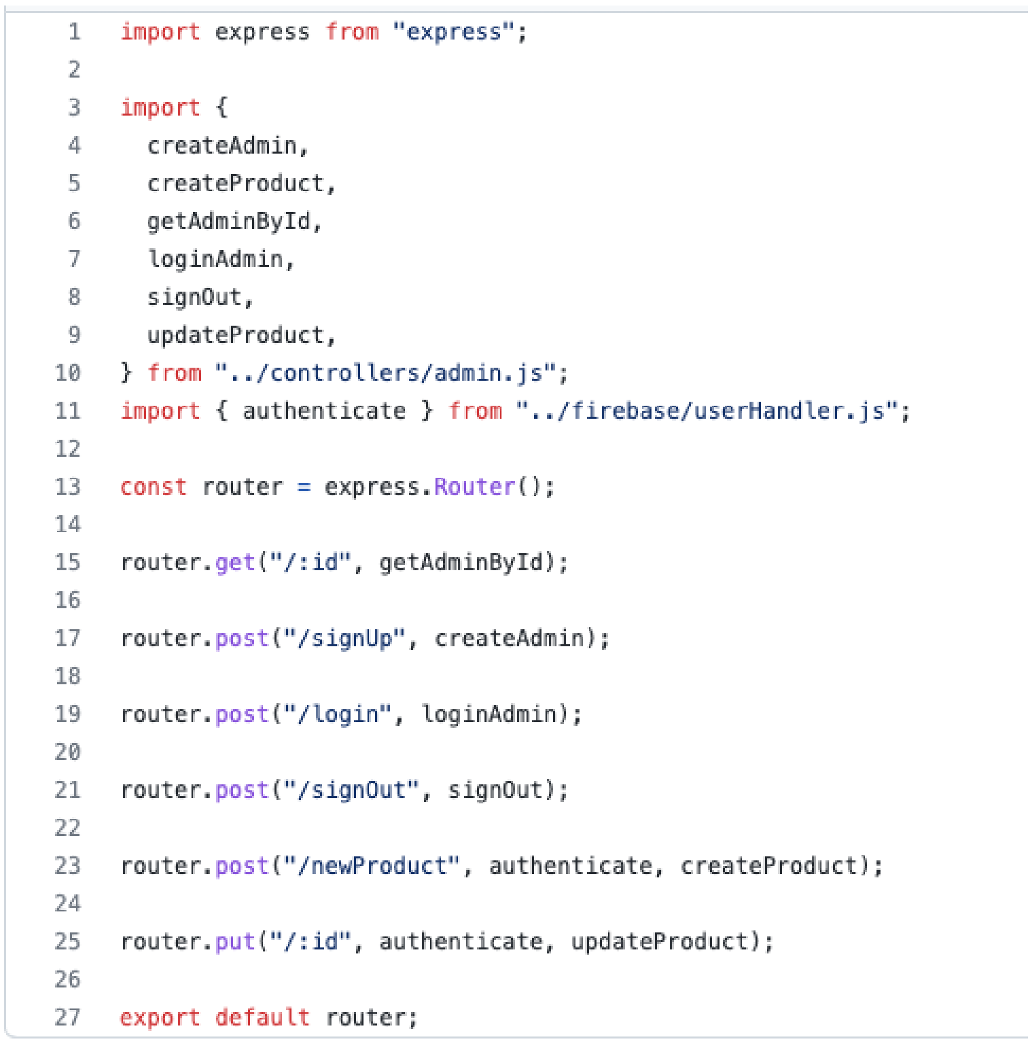
5.4 Routes

In this Full stack application we will use Express JS to create and handle routes from the server side that can be addressed next from the front end for specific queries and methods as (‘GET’, ‘POST‘, ‘PUT’, ‘DELETE’) and for different endpoints as (‘USER‘, ‘ADMIN’, ‘PRODUCT’, ‘ORDER’).

For that reason, we should add express.json() as a middleware, that takes care of parsing the incoming JSON requests and structures the data in the req.body. And as we will upload product image as a file type, we need to add ‘*express.urlencoded({ extended: true })’* as a middleware.

Furthermore, we will be adding Cors middleware to bypassing the Access-Control-Allow-Origin headers in the requests, that indicates which origins is permitted to access our Api.

5.4.1 Admin routes



```
import express from "express";

import {
  createAdmin,
  createProduct,
  getAdminById,
  loginAdmin,
  signOut,
  updateProduct,
} from "../controllers/admin.js";
import { authenticate } from "../firebase/userHandler.js";

const router = express.Router();

router.get("/:id", getAdminById);

router.post("/signUp", createAdmin);

router.post("/login", loginAdmin);

router.post("/signOut", signOut);

router.post("/newProduct", authenticate, createProduct);

router.put("/:id", authenticate, updateProduct);

export default router;
```

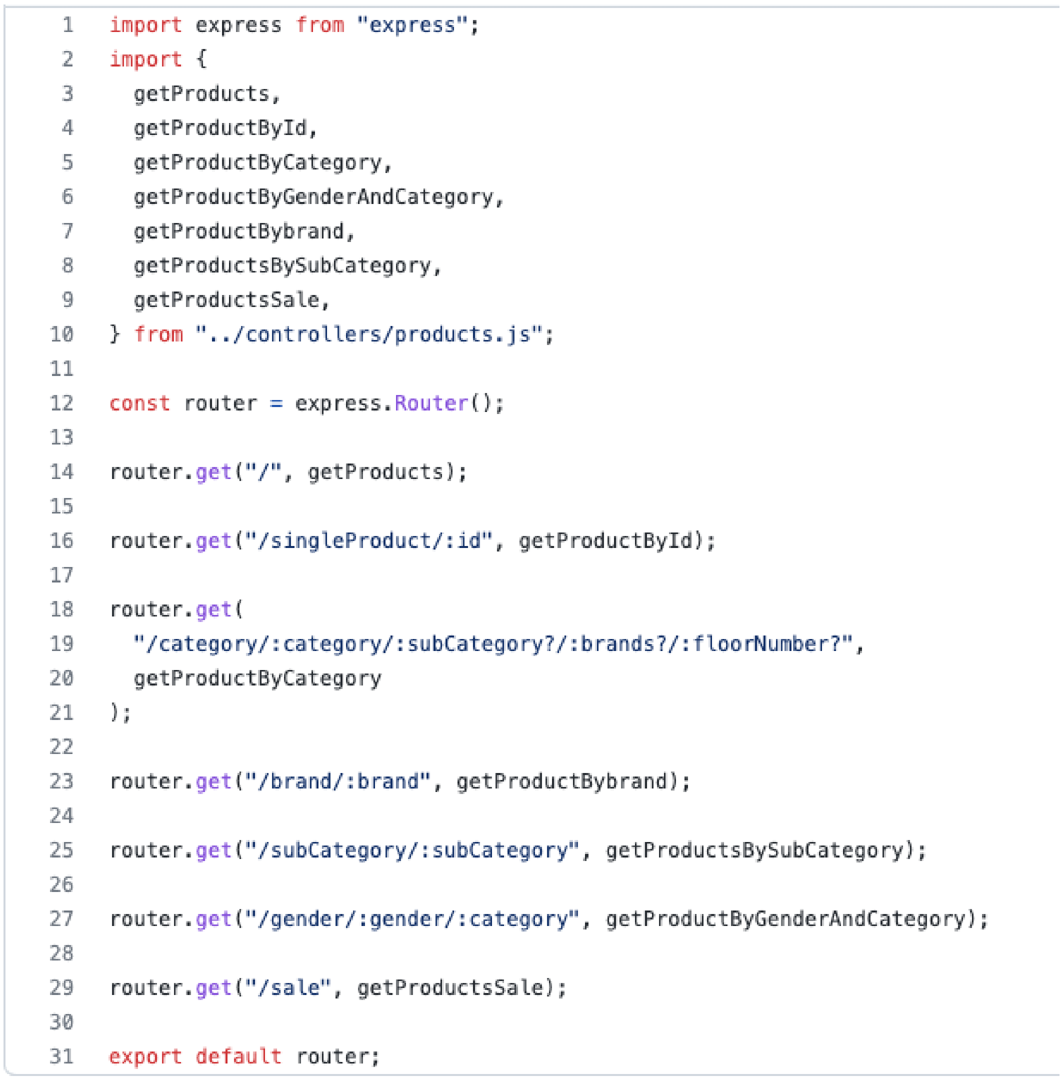
Figure 31. Admin routes supported by the backend.

As seen in the Mongoose models earlier, we have defined admins to be different from users, as they are responsible for handling all products within the store, and for that purpose we need to define separate routes for admins such as   *Login*, *Register*, *createProduct*, and *getAdminById*.

Furthermore, by separating the admin's routes from the client's routes, we are securing sensitive endpoints as creating products and adding new admins from any potential data leak and bridge hacking, as we keep clients routes to only have more read than write access throughout the application.

5.4.2 Product routes

Our product routes will be consisting just of reading data from the database as we added earlier the create product route to the admin route for security reasons as explained.



```javascript
import express from "express";
import {
  getProducts,
  getProductById,
  getProductByCategory,
  getProductByGenderAndCategory,
  getProductBybrand,
  getProductsBySubCategory,
  getProductsSale,
} from "../controllers/products.js";

const router = express.Router();

router.get("/", getProducts);

router.get("/singleProduct/:id", getProductById);

router.get(
  "/category/:category/:subCategory?/:brands?/:floorNumber?",
  getProductByCategory
);

router.get("/brand/:brand", getProductBybrand);

router.get("/subCategory/:subCategory", getProductsBySubCategory);

router.get("/gender/:gender/:category", getProductByGenderAndCategory);

router.get("/sale", getProductsSale);

export default router;
```

Figure 32. Product routes supported by the backend.

As the figure above shows, we have defined a route for getting single product by the id entity with the endpoint *'/singleProduct/:id'*, getting all available products with the plain ‘/’ route that calls *getProduct()* function from the product controller folder.

Then we can add filtering for our product queries, such as:

- Filtering by brand, with the endpoint ‘/brand/:brand’ that calls *getProductByBrand()* function.

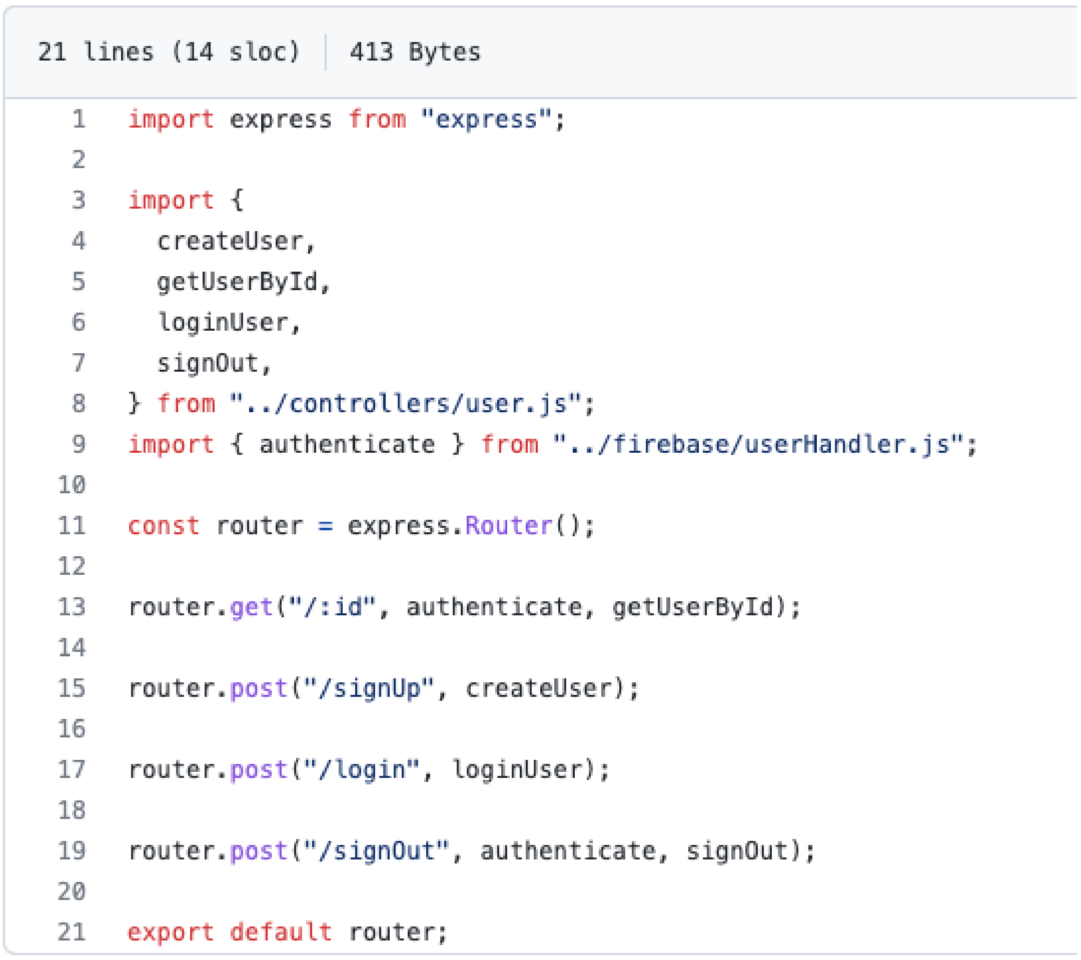
- Filtering by subCategory, with the endpoint ‘/subCategory/:subCategory’, that calls *getProductsBySubCategory()* function.

- Filtering products by gender and category, with the endpoint ‘gender/:gender/:category’ that calls *getProductByGenderAndCategory()* function.

- Filtering product by the Sale option, that indicates if there are some discounts offer on some products, with the endpoint ‘/sale’, that calls *getProductSale()* function.

- Filtering with many options simultaneously, as (‘*category’*, ‘*subCategory’*, ‘*brand’* and ‘*floorNumber’*), with the endpoint *‘/category/:category/:subCategory?/:brands?/:floorNumber?*’, that calls *getProductByCategory()* function, in which ‘*floorNumber’* entity will be used in the product controller as a maximum number of products to return with a random order.

5.4.3 User routes


```
21 lines (14 sloc) | 413 Bytes
```

```
import express from "express";

import {
  createUser,
  getUserById,
  loginUser,
  signOut,
} from "../controllers/user.js";
import { authenticate } from "../firebase/userHandler.js";

const router = express.Router();

router.get("/:id", authenticate, getUserById);

router.post("/signUp", createUser);

router.post("/login", loginUser);

router.post("/signOut", authenticate, signOut);

export default router;
```

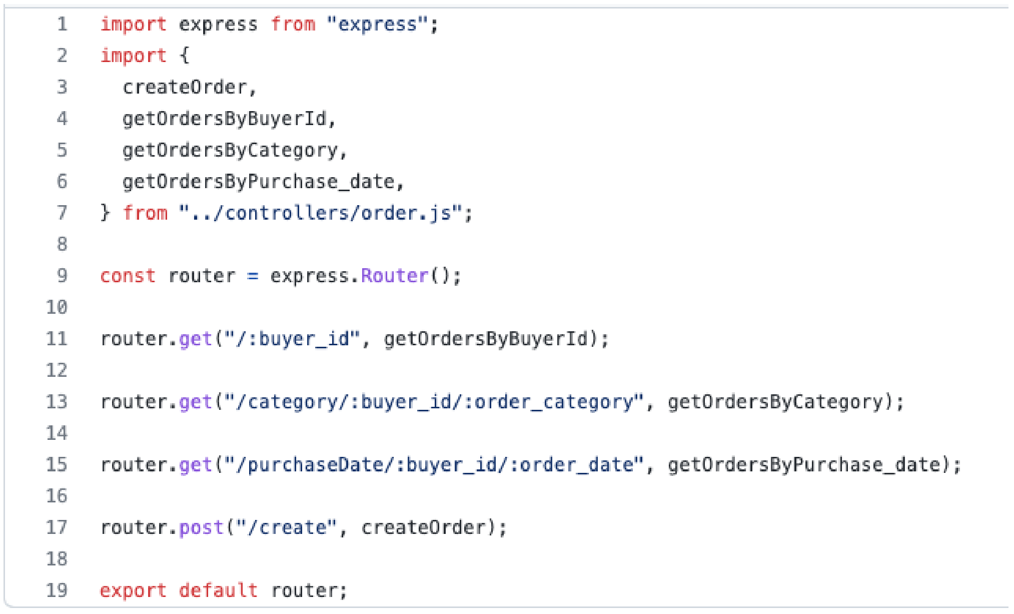
Figure 33. User routes supported by the backend.

As the above figure shows multiple user routes endpoints that consists of getting the user information with the endpoint ‘/:id’ which calls *getUserById*() function.

Then, the authentication part responsible for login, with the endpoint ‘/login’ and register, with the endpoint ‘/*signUp’*. Both of the *createUser*() and *loginUser*() functions have more similarities from the codebase point of view, as seen in the user controller file.

And finally the *signOut* route with the endpoint ‘/*signOut’* which calls *signOut*() function from the user controller file.

5.4.4 Order route



```
import express from "express";
import {
  createOrder,
  getOrdersByBuyerId,
  getOrdersByCategory,
  getOrdersByPurchase_date,
} from "../controllers/order.js";

const router = express.Router();

router.get("/:buyer_id", getOrdersByBuyerId);

router.get("/category/:buyer_id/:order_category", getOrdersByCategory);

router.get("/purchaseDate/:buyer_id/:order_date", getOrdersByPurchase_date);

router.post("/create", createOrder);

export default router;
```

Figure 34. Order routes supported by the backend.

As shown in the figure above, we can fetch orders based on the buyer id entity, with the endpoint '/:*buyer_id*' which is a unique entity for each client.

Also, the client have the possibility to create orders from the front end, with the endpoint '*/create*' using the POST method that calls *createOrder*() function from the order controller file.

Furthermore, other order routes can be represented as endpoints for getting the purchased goods orders information with filtering such as:

- Getting orders by category, with the endpoint *"/category/:buyer_id/:order_category*".

- Getting orders by the purchase date, with the endpoint *"/purchaseDate/:buyer_id/:order_date*".

6 Discussion

In this part we will go through the outcomes of this thesis and possibilities for improvement that can be made for the developed full-stack e-commerce application.

6.1 Evaluation

Ultimately, by leveraging the four core technologies that comprise the MERN stack and employing numerous Node modules, a prototype version of an electronic commerce application that mirrors an internet-based shop was successfully developed. This particular application is designed to be efficient, user-friendly, and operate seamlessly.
The clients of this platform have access to several features such as authentication, surfing the available products, and making and reviewing their order histories. Additionally, this application provides various different product filtering options that rely on the product's category, sale, brand, and sub-category.

The administrative users of this platform enjoy the same capabilities and functions as regular users and have access to additional features such as creating and managing new products. Furthermore, the payment gateways for this application have already undergone rigorous testing in a sandbox account and have demonstrated reliable functionality.

In terms of the operational effectiveness of a comprehensive e-commerce application designed with small businesses in mind, particularly an online shop, that was conceptualized from the outset, the output product successfully satisfied all the criteria for functionality.

6.2 Improvements

During the development of this application, several improvements ideas have been noticed such as:

Firstly, the application's security can be enhanced. Security is a critical aspect of any e-commerce application since it involves sensitive information such as personal details and payment information. Implementing features such as two-factor authentication, SSL/TLS encryption, and secure password storage can help bolster the application's security and increase user trust. [15].

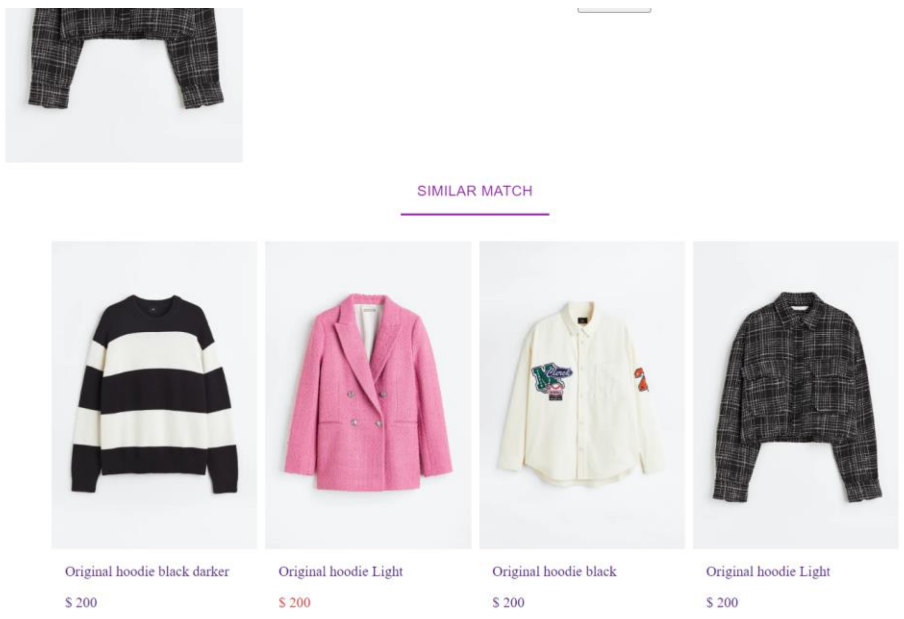


Figure 35. Similar product match feature in the product page.

Secondly, although the application has a similar product match feature on the product page, It is beneficial to incorporate an artificial intelligence smart product recommendation functionality into the platform. This feature can suggest products that are relevant to the client's past orders or browsing history, thereby improving the probability of conversion and elevating the user experience, resulting in higher customer loyalty and retention rates. [16.]

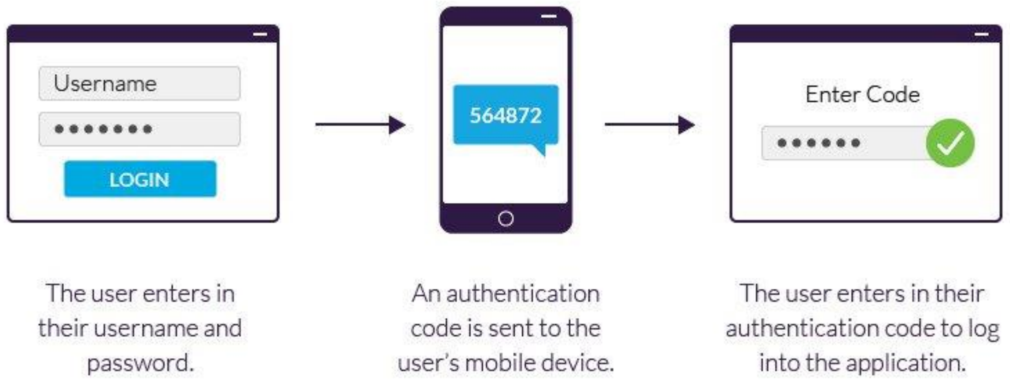

Username
•••••••
LOGIN
564872
Enter Code
••••••
The user enters in their username and password.
An authentication code is sent to the user's mobile device.
The user enters in their authentication code to log into the application.

Figure 36. Two-factor authentication workflow.

Thirdly, incorporating a chatbot feature into the application can provide prompt responses to customer inquiries, resulting in higher levels of satisfaction and loyalty. Additionally, it can ease the burden on customer support representatives, freeing up their time to tackle more intricate issues. [17.]

7 Conclusion

The goal of this dissertation was to investigate the unique features of every technology in the MERN stack and build a complete e-commerce web application utilizing it. The author dedicated a significant amount of time to research and thoroughly study each contemporary technology in order to facilitate the development of the web application. The dissertation presented a detailed examination of all the technologies that make up the MERN stack, including fundamental principles and advanced features, and their use in the e-commerce domain to ensure the readers' comprehension. Additionally, the dissertation provided a detailed guide on the necessary steps for developing an e-commerce application.

Also, the author accomplished the creation and publication of a comprehensive e-commerce application that included an online store, with the intention of generating a functional e-shop that could be assimilated into any small business. The primary objective of this e-shop was to provide clients and customers with access to its services, while also increasing its popularity by embracing the virtual world in conjunction with the physical world.

To summarize, this thesis can be utilized as a guide or point of reference for individuals interested in the MERN stack or full-stack web development as a whole. The author has acquired invaluable knowledge by conducting extensive research and studying this manuscript, resulting in a deeper comprehension of the reasons behind the recent surge in popularity and leadership role of the MERN stack in web development. Despite the application's existing limitations, such as stylistic issues and the need for new features, it nevertheless represents an amalgamation of the most commonly used web stack technology with one of the most burgeoning business concepts of our time - e-commerce.