Nhu Quynh Bui

E-commerce mobile application

Bachelor's thesis

Information Technology

Bachelor of Engineering

2021

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree title	Time
Nhu Quynh Bui	Bachelor of Engineering	August 2021
Thesis title E-commerce mobile application		46 pages 0 pages of appendices
Commissioned by		
Supervisor Timo Hynninen		

Abstract

The thesis goal is to create a mobile application that allows users to trade products online. The application's target user is everyone. The theoretical background was composed to provide the study concepts and definition of React Native, Nodejs, MongoDB, REST API, mobile application development, and software development lifecycle models. The theoretical background provides a solid basis for the practical implementation.
The practical implementation part provides the planning for server-side implementation and the deloping process of server-side and client-side. In this part, the application development was demonstrated. This part also provides images of the application which help demonstrate how to use the application.
The final result is the functional mobile application that allows users to trade products online. The application was developed using methods and technologies that were introduced in the theoretical background.

Keywords

mobile application development, React Native, Nodejs

CONTENTS

1 INTRODUCTION

Nowadays, we have a trend called "mobile first", which means designing user interface for mobile first before desktop applications and mobile applications. Smartphones play a crucial role in our life. More and more mobile applications are developed in order to improve our life. In software development, we have various options, for example, desktop applications, web applications, mobile applications and others. However, the reason why mobile is leading is that almost everyone now owns a smartphone so it would be the easiest way for people to approach an application.

E-commerce is a model that allow firms and individuals to trade things online. It was first introduced in 1979 when Michael Aldrich invented electronic shopping by connecting a customized TV to a transaction-processing computer through a telephone line. The significant development of E-commerce happened in 1995 when Amazon and eBay were introduced to the market. Amazon was started by Jeff Bezos, while Pierre Omidyar launched eBay. Due to the pandemic situation, the demand for online shopping has increased rapidly because social distancing is required. Moreover, convenience is one reason why online shopping demand has developed quickly.

The application that I tend to do is an E-commerce mobile application. The app includes two roles, which are admin and user. Users can view and buy the products that are available in the application while admins are allowed to manage those products. The main function of this application is that users can search, view and select the products that they want to buy. An admin can manage products by adding or deleting an item, as well as upload pictures and descriptions for products.

The application target users are everyone who want to trade products online. The application provides a platform for people who want to do ecommerce. E-commerce is so popular that many online buying websites are developed. However, the number of people that are using mobile are increasing day by day. It is easier to carry a smart phone outside instead of a laptop so mobile

application would be a great choice to develop. It helps bussinesses to get in touch with their customers easier. That is the reason why I decided to develop a mobile application.

In order to achieve the goals, the thesis will be divided into different parts, described as follows:

- **Part 1** (Introduction) provides general information about the thesis topic which includes the objectives of the thesis
- **Part 2** (Theoretical Background) describes the definition of the method that is used in the thesis to create the solution for the topic
- **Part 3** (Implementation) explains the process of development and implemantation to provide the final result.
- **Part 4** (Conclusion) summarizes the outcome of the topic

During the development process, there were several issues and limitations. The most significant problem is bug. I have met several bugs during the implementation. To resolve these bugs, I have spent many times to investigate and fix them. Moreover, the platform that I used to deploy the server is free to use so it is quite slow. Also, there is a problem that still cannot find a solution for it is some images that upload to the server are not display in the application. This problem might be caused by the storage of images. However, I still cannot find a solution to replace it.

2 THEORETICAL BACKGROUND

This section covers the study concepts of technologies that are used in the project. It will provide both basic and detailed knowlegde of methods that are used to develop the mobile application. Following that, the requirements for the application and a brief introduction to mobile development are also described.

2.1 Mobile application development

Mobile applications can be divided into three types, which are native application, hybrid application and cross-platform application. Native applications are applications that are developed for a specific operating system. It can be Java or Kotlin for Android and Objective-C for iOS (Sergey 2019). The most important advantage that native applications have is that they are high performing applications, which create an impressive user experience. However, it is expensive to create a native application because it requires a different code base for different operating systems. One more way which we can approach mobile application development is hybrid application. Hybrid application can be described as a webview object that is deployed in a native container. Developing a hybrid application is simple and take less time, however, a hybrid application doesn't provide a brilliant user experience. Another approach is cross-platform application. Cross-platform application is closest to a native application. The code base is written in JavaScript and connects to native components using bridges (Sergey 2019). It allows sharing code between different platforms, therefore, with one code base we can develop applications for both Android and iOS. Some cross-platform tools which allow creating cross-platform applications are Xamarin and React Native. The application in this project is considered as an cross-platform application.

2.2 Project requirements

As a result, the application should be working with the provided features. The application will have two roles, which are users and admins. For users, they can use all the features as follows:

- Sign in and sign out
- Register
- Search and select products
- Checkout process: Fill in shipping form, choose payment method, confirm order

There is one tab for admin to manage the products. Admins are allowd to do these actions as follows:

- Add and update products
- Delete products
- Upload products' image
- Update orders' status
- Add categories

However, the payment is not implemented in the application. Therefore, the application flow is stopped when user confirm their orders

2.3 React Native

React Native is a platform that allows developers to create mobile applications for both Android and iOS. It was first released in 2015 by Facebook. React Native combines the best parts of native development with React and a best-in-class JavaScript library for building user interfaces. (React Native 2021.) React Native is built by JavaScript, and therefore both JavaScript and TypeScript can be used in development. It will use native view instead of webview so the result application will be an native application.

There are two important threads in React Native, which are Main Thread and JS Thread. JS Thread is responsible for the application logic, which means the code execution, API fetch and other processes. Main Thread is responsible for rendering native views. JS Thread is also used to decide what will be displayed in the screen and it will inform the Main Thread. However, Main Thread and JS Thread will never communicate directly. These two threads will communicate by using React Native bridge. This bridge allows two threads that are written in two different languages to communicate, which makes "bridge" a core aspect of React Native structure.

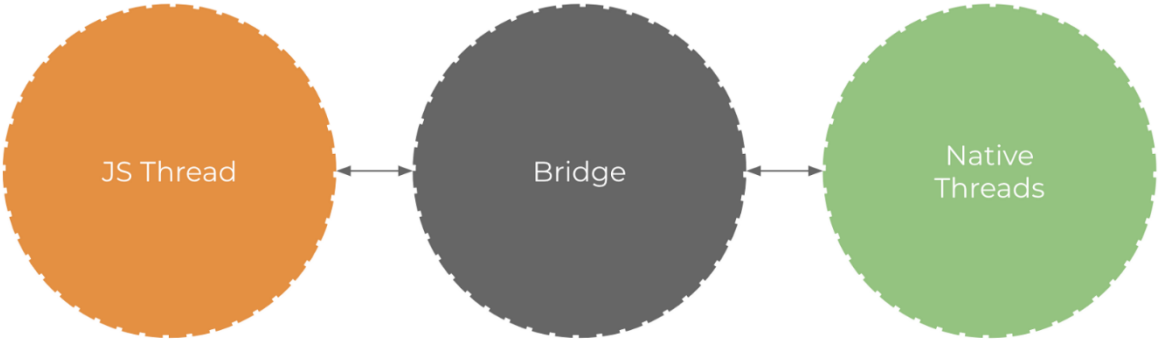


Figure 1. JS Thread communicates with Native Thread

2.4 Nodejs

Nodejs is a open-soure and cross-platform JavaScript runtime environment. It was first introduced in 2009. Nodejs was built on Google Chrome's V8 JavaScript engine which ensures rapid code execution. Nodejs runs in a single process, without creating a new thread for every request (OpenJS Foundation). It is an asynchronous, event-driven and non-block I/O platform, which means it won't block the thread and waste CPU cycles. "Asynchronous" means Nodejs doesn't need to wait to execute all the statements in the right order, it can execute different statement while waiting. As a result, these will help Nodejs improve its performance. Nowadays, Nodejs is an popular option in the developer community all over the world. Moreover, it is widely used by many large corporations. Acording to Jignesh (2021), Nodejs is considered to be the most used tool in early 2021. Besides, there is also Node Package Manager or NPM. NPM is now the world's largest software registry. NPM is use for installing packages of code to your application, sharing code with npm users and so on. It helps ease the development processes.

2.5 MongoDB

MongoDB is a document database which is free to use. MongoDB was first released in 2009. It is scalable and flexible, which provides many capabilities. MongoDB drivers are available in more than ten languages. With each language, it provides an official document on its website which make mongoDB simple to use. MongoDB is available in two options, which are cloud and server. MongoDB

cloud is known as MongoDB Atlas. MongoDb server is an application that we can download and install it to use locally. Although MongoDB is available in two options but both of them provides same features. MongoDB store data in a JSON-like document, which means fields can vary from document to document and data structure can be changed over time (MongoDB Inc 2021).

2.6 REST API

API or application programming interface is a set of rules define how application can connect and communicate with each other (IBM Cloud Education, 2021). REST API or RESTful API is an API that follow the REST principle. REST is known as representational state transfer. It is an architectural style and was first introduced in 2000. To develop REST APIs, any programming languages can be used. However, to be considered a REST API, it must follow the REST principles which are uniform interface, client-server decoupling, statelessness, cacheability, layered system architecture, code on demand. REST APIs using HTTP requests to perform function. It provides creating, reading, updating and deleting function to manage records. GET request is used to read the record, which means we can retrieve the data through GET request. POST request is to create record while PUT and DELETE request are to update and delete records. REST API is a flexible and lightweight solution to conform applications. It is now one of the most common methods for connecting applications.

2.7 Software development life cycle models

Software development life cycle is a process to design and develop software. Different project will have a different suitable model. There are more than 50 recognized software development life cycle models (SDLC) in use (Boris, S. 2019) . However, there are some models that are more popular compare to others, for example, Waterfall, V-model, Incremental and Iterative model, Spiral model, The Rational Unified Process and The Agile group. In this report, Agile methodology will be introduced. Agile is one of the most popular methodology, it is chosen and implemented in IT projetcs of about 70% of organization (Boris, S. 2019). In Agile, the development process will be divided into iterations. The team

will collaborate in planning, processing and evaluating each iteration to get the best result. In the Agile group, it includes variable models, for example, Scrum, Kanban, Extreme Programming and Adaptive Project Framework. Scrum model is now one of the most popular model of Agile. Scrum seperate the development process into small iteration call sprint. Each sprint usually lasts for two to four weeks. Before each sprint starts, there will be a sprint planning meeting to decide which issues should be handled in this sprint. At the end of each sprint, there will be a retrospective meeting to clarify what the team has achieved and what should be improve. Scrum model will help organize and speed up the development process.

3 PRACTICAL IMPLEMENTATION

This section is divided into three parts, which are application overview, server implementation ,and mobile application implementation. The first part is about which features are included in the application. The second part will cover the details about what is the plan for the server and how it was implemented. The last part will be about the implementation of the mobile application.

To provide a clearer picture of the implementation process, I will use the following typography in this section:

- API paths will be in **bold**
- Code snippets are color-coded
- Code quotes are in *italic*

The syntax makes it easier to follow the implementation part.

3.1 Application overview

The mobile application provides a platform that contains two roles which are user and admin. For users, they can search, select and order products that they demand. For admins, the application is used to manage all the available products. Moreover, they can also add new products to the platform.

The following list will describe the features and funtions that are available in the mobile application to make it simpler to understand how it works:

- Product list includes all the available products so that users can search for products and view their details. They can search products by their name using the search bar or by categories. Users can also add the wanted items to their shopping cart.
- The shopping cart displays all the items that are selected by a user. Users can do the checkout process in the cart. The checkout process will include a form for shipping information, payment method ,and confirmation before placing an order.
- User management allows users to sign in and sign out the application. Moreover, users can also see their profile.
- Product management is only available for accounts that have admin rights. The admin rights will include product management, category management ,and order management. Admins can add a new product with all the required information. They can also manage orders by updating their status.

In order to fulfill the requirements, a database and server are needed to implement the mobile application. The database is used to store all the data that is used in the mobile application. The server is used to serving the requests to retrieve and manage data from the mobile side.

3.2 Server-side implementation

3.2.1 API planning

An application programming interface or API is used as a tool for communication between a mobile application and a server. In order to understand how APIs work, it is necessary to know their paths, inputs and funtionality. 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/register	POST	name, phone, email, password	To register new user
/users/getProfile/:id	GET	id	To get user profile by id
/products/getProductList	GET	-	To get list of all products
/products/getProduct/:id	GET	id	To get product details by its id
/products/searchProduct	GET	search (search text)	To search products by keyword
/products/createProducts	POST	name, brand, price, category, countInStock, description, image	To add new product to database
/products/updateProduct	PUT	name, brand, price, category, countInStock, description, image	To update existing products' information
/products/deleteProduct/:id	DELETE	id	To delete product by id
/categories/getListCategory	GET	-	To get list of all categories
/categories/addCategory	POST	name	To add new category
/orders/createOrder	POST	orderItems, shippingAdress1, city, totalPrice, zip, country, phone	To create a new order
/orders/getOrderList	GET	-	To get a list of all orders
/orders/updateStatus	PUT	Status	To update an order's status

In the implementation, this table helps organize all the APIs that are used during the development. All the APIs are created base on the mobile application requirements.

3.2.2 Server implementation

Before implementing the server, a database is needed to store all the data. In this development, I chose MongoDB as the database to implement. The database includes five collections which are categories, orders, products, users, and orderItems. These collections usage is listed below:

- Users collection is for storing all users' data include username, password, and profile information.
- Orders collection are used to store information about customers' orders, which includes the shipping information and order items.
- Orderitems collection stores the detail about the items that are included in an order. From the products' id, the products' details can be retrieved.
- Categories collection has information about the existing categories
- Products collection include information on all products

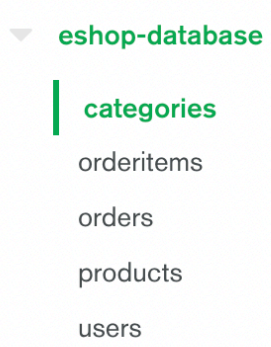


Figure 2. Database collections

As mentioned above, there are two options for MongoDB which are cloud and server. In this project, I chose to use the cloud version of MongoDB called Atlas. For MongoDB, there are three ways that we can connect the database to the

server: connect with MongoDB Shell, connect your application, and connect using MongoDB Compass. I chose to connect to my application which requires a connection string that is provided by MongoDB. Figure 3 displays how to connect to MongoDB in your application.

```
mongoose
  .connect(process.env.CONNECTION_STRING, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    dbName: "eshop-database",
  })
  .then(() => {
    console.log("connection is ready");
  })
  .catch((err) => {
    console.log(err);
  });
```

Figure 3. MongoDB connection

To create a collection, a schema must be defined first. Each schema maps to a MongoDB collection and define the shape of documents within that collection (Mongoose). Each field of a schema will be the collection property. Each field also has different options to be defined, for example, type, default, required, etc. . Figure 4 describes all the collections in the database and their fields.

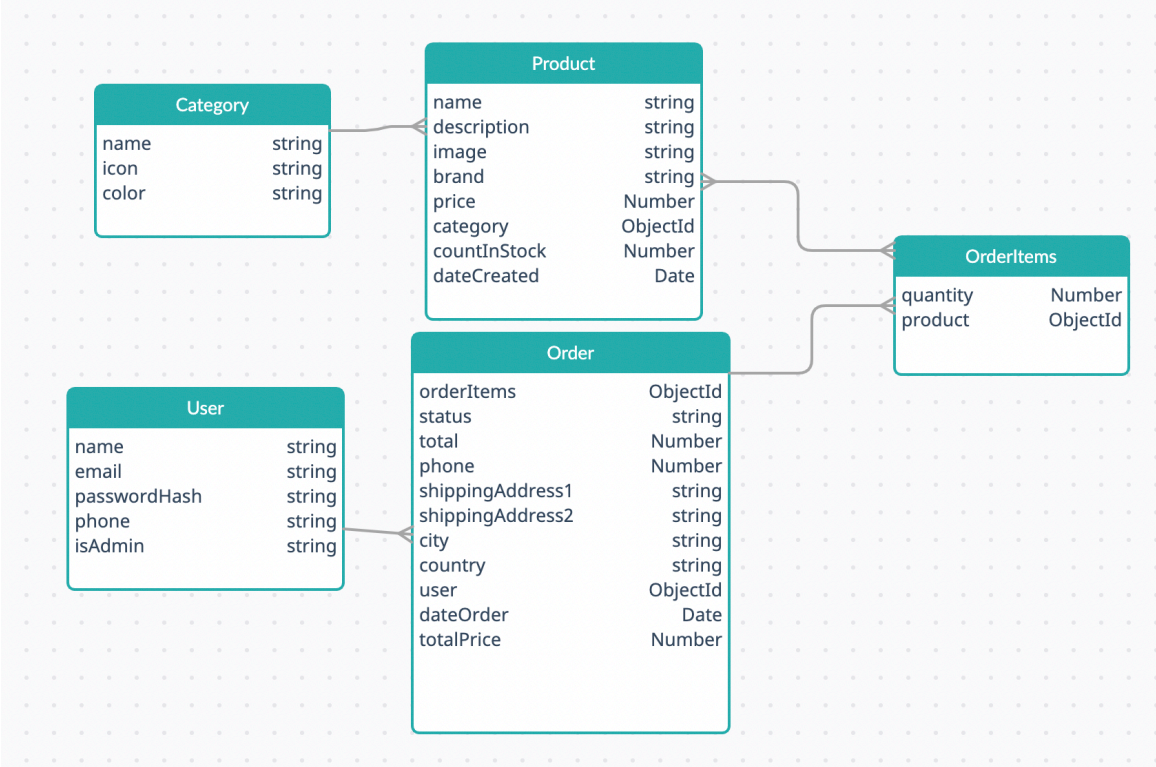


Figure 4. Database collections

However, to use the schema in the collection, it must be converted into Model. Figure 5 shows how to convert schema to model

```
exports.Order = mongoose.model('Order', orderSchema);
```

Figure 5. Convert schema to model

In the database, there are five collections, each of them should have a model and schema to shape the documents within the collection. After defining all the models for each collection, I created a route or URL path for each request that was described in Table 1. Each API path will be a route. To make it easier to understand, I will divide the API paths into small groups of paths that have a similar function. First, to create a route, I had to choose a method for each route. Four methods were used in this project: GET, POST, PUT, and DELETE. Second, based on each path description, I defined a prefix and a function for the route.

```
//get list category
router.get(`/getListCategory`, async (req, res) =>{
    const categoryList = await Category.find();

    if(!categoryList) {
        res.status(500).json({success: false})
    }
    res.json({success: true, categoryList})
    ;
})
```

Figure 6. API get list category

Figure 6 displays how to create a GET route to get a list of all categories. *router.get()* was used to create a GET method. */getListCategory* is the prefix for this route. The description of this API path is to get all the categories that are existing in the application so from the Category model, I used *find()* to retrieve all the data that is available. If the path was able to get data from the collection, the API path will return a response including data and a success status, which is true. If there are any problems that prevent getting data, the route will send an error message with a false success status. According to the API path description, **/products/getProductList**, **/categories/getListCategory**, and **/orders/getOrderList** have similar functions so all of them will have a similar codebase but with different prefixes.

The next part explains how to create a GET route that requires input to get sufficient data. These **/users/getProfile/:id** and **/products/getProduct/:id**. In these two paths, there is *:id* after the prefixes. This means a parameter called id is needed in this path. Based on the id, the route can get the exact information for each product or user in the collections. In order to get the information, *findById()* method is used to get data by id in these API paths. Figure 7 displays how to create these routes.

```
router.get(`/getProduct/:id`, async (req, res) => {
  const product = await Product.findById(req.params.id).populate("category");
  if (!product) {
    res.status(500).json({
      success: false,
    });
  }

res.status(200).json({success: true, product})
});
```

Figure 7. API get product by id

In the mobile application, there is a feature that allows users to search for the products. The **/products/searchProduct** path is used for this feature. This path allows getting products whose names have the keyword that users are searching. For this route, the search keyword is the required input. This route returns all the products whose names have the keyword inside regardless of position and case sensitive. This API path code is displayed in Figure 8.

```
router.get(`/searchProduct`, async (req, res) => {

    let filter = req.query.search ? req.query.search :'';

    const productList = await Product.find({name: new RegExp('.*' + filter + '.*','i')});
    if (!productList) {
      res.status(500).json({
        success: false,
      });
    }
  //   res.send(productList);
  res.status(200).json({
      success: true,
      productList
    });
  });
```

Figure 8. API get product by keyword

Next path covers how to construct routes for **/users/register**, **/categories/addCategory**, and **/orders/createOrder**. Each route requires different inputs however, their functions are similar. The method used in these routes is POST. These routes are used to collect data that is sent from the mobile application and store it in the database. Each path input was defined in Table 1. I will take the **/users/register** path as an example. If the function can save the user to the collection, it will return the user information as a response. If it is

unable to save, the response will be an error message, which is “User cannot be created”.

```
router.post(`/register`, async (req, res)=>{

    let user = new User({
        name: req.body.name,
        email: req.body.email,
        passwordHash: bcrypt.hashSync(req.body.password,10),
        phone: req.body.phone,
        apartment: req.body.apartment,
        isAdmin: req.body.isAdmin,

    })

    user = await user.save()

    if (!user){
        return res.status(400).send({
            message: 'User cannot be created'
        })
    } else {
        return res.send(user)
    }

})
```

Figure 9. API register new user

To secure the application, a token is essential whenever a request is done. The token is generating after the user login to the application. In this project, I used JSON Web Token for authorization. The process of generating the token happens in the **/users/login** route. After the route receives an email and password from the mobile application, it 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 “User not found” message. *jwt.sign()* function combines the payloads, which are userId and isAdmin fields, with the secret key to 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 (Auth0). After creating the token, the path returns the

user's email and token in the response. Figure 10 displays how to implement login API.

```
//login
router.post('/login', async(req, res)=>{
    const user = await User.findOne({email: req.body.email})
    const secret = process.env.secret

    if (!user){
        return res.status(400).json({message:'User not found', success: false})
    }
    if (user && bcrypt.compareSync(req.body.password, user.passwordHash)){
        const token = jwt.sign(
            {
            userId: user.id,
            isAdmin: user.isAdmin
        },
        secret,
        {
            expiresIn: '1d'
        }
        )
        res.status(200).send({
            user: user.email,
            token: token,
            success: true
        })
    } else {
        res.status(400).json({message:'Incorrect email or password', success: false})
    }
})
```

Figure 9. Login API

When creating a product, information about the name, brand, price, image, etc is required. This API path also has quite a similar function with **/users/register** path, however, the difference is that this path has a function to upload images. To store images, I use multer library. The destination to store images is /public/images folder. There is also the validation for file types. In this application, I configured it to allow only jpg, png, and jpeg files. In case of the wrong type, the route sends an error message. By using this library, I can upload images from the mobile application to the server. Figure 10 shows to configure the storage for images.

```
const FILE_TYPE = {
  "image/jpg": "jpg",
  "image/png": "png",
  "image/jpeg": "jpeg",
};

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    const isValid = FILE_TYPE[file.mimetype];
    let error = new Error("Invalid type");
    if (isValid) {
      error = null;
    }
    cb(error, "./public/images");
  },
  filename: function (req, file, cb) {
    const fileName = file.originalname.split(" ").join("-");
    const extension = FILE_TYPE[file.mimetype];

    cb(null, `${fileName}-${Date.now()}.${extension}`);
  },
});
```

Figure 10. Multer configuration

The **/products/createProducts** path configures the path for images that are uploaded to the database. The image path will be used to render the image in the mobile application. Except for the uploading function, this route saves the product to the database. If the process is successful, it returns the updated products. Otherwise, it returns an error message. The code for this path is described below.

```
router.post(`/createProducts`, upload.single("image"), async (req, res) => {
  const category = await Category.findById(req.body.category);
  if (!category) {
    return res.status(400).json({
      message: "Invalid category",
      success: false
    });
  }

  const file = req.file;
  if (!file) {
    return res.status(400).json({
      message: "Invalid image",
      success: false
    });
  }
  const fileName = req.file.filename;
  const imagePath = `${req.protocol}://${req.get("host")}/public/images/`;
  let product = new Product({
```

```
    name: req.body.name,
    description: req.body.description,
    richDescription: req.body.richDescription,
    image: `${imagePath}${fileName}`,
    brand: req.body.brand,
    price: req.body.price,
    category: req.body.category,
    countInStock: req.body.countInStock,
    rating: req.body.rating,
    numReviews: req.body.numReviews,
    isFeatured: req.body.isFeatured,
  });

  product = await product.save();
  if (!product) {
    return res.status(400).json({
        success: false,
      message: "Product cannot be created",
    });
  } else {
    return  res.status(200).json({
        success: true,

    product
    });
  }
});
```

Besides adding new products, there is a feature to delete products. The API path is **/products/deleteProduct/:id**. It uses the product id to retrieve that product and to remove that from the collection.

```
router.delete("/deleteProduct/:id", (req, res) => {
  Product.findByIdAndRemove(req.params.id)
    .then((product) => {
      if (product) {
        return res.status(200).json({
          success: true,
          message: "The product is deleted",
        });
      } else {
        return res.status(404).json({
          success: false,
          message: "Product not found",
        });
      }
    })
    .catch((err) => {
      return res.status(400).json({
        success: false,
        error: err,
      });
    });
});
```

Figure 11. API to delete a product

This part is to demonstrate how to product **/products/updateProduct** and **/orders/updateStatus** path. To update the path, I used PUT method. The fields that need to be updated are the inputs for these paths. It also requires the id to find products or orders in the collections. The route uses *findByIdAndUpdate()* method to update the fields. Each path has separate required inputs that were defined in Table 1. Figure 12 illustrates the code base for **/orders/updateStatus** path.

```
router.put('/updateStatus/:id', async (req, res)=>{
    const order = await Order.findByIdAndUpdate(
        req.params.id,
        {
            status: req.body.status
        },
        {new: true}
        )
        if (!order){
            return res.status(400).send('The status cannot be updated')
        } else {
            res.status(200).json({success: true, order})

        }
})
```

Figure 12. Update status API

To improve security for the application, the token is required in all the API paths. However, there are some paths that do not need to apply the token. The token is only generated after login. However, users are allowed to view the products, place orders, and register a new account while not signed in. Figure 13 specifies paths that do not need the token: login, register, all the GET requests for products, GET requests for categories, GET requests for images, and POST requests for orders.

```
function authJwt() {
    const secret = process.env.secret //JWT_SECRET
    const api = process.env.API_URL
    return expressJwt({
        secret,
        algorithms: ['HS256'],
        isRevoked: isRevoked
    })
    .unless({
        path:[
            `${api}/users/login`,
            `${api}/users/register`,
            {url: /\/api\/v1\/products(.*)/ , methods: ['GET', 'OPTIONS'] },
            {url: /\/api\/v1\/categories(.*)/ , methods: ['GET', 'OPTIONS'] },
            {url: /\/images(.*)/ , methods: ['GET', 'OPTIONS'] },
            {url: /\/api\/v1\/orders(.*)/ , methods: ['POST', 'OPTIONS'] },


        ]
    })
}
```

Figure 13.Token exception

The server runs locally using localhost. To run the server globally, I need to deploy the server to a deployment platform. I chose Heroku as the deployment platform to deploy my server. To do the deployment, we can use the command line to choose to deploy on Heroku website. In this project, I chose to deploy on Heroku website by connecting to my repository on GitHub. On Heroku, I created an application called nhu-eshop-server. After that, I selected GitHub as the deployment method. From that, I can choose the branch that I want to deploy to Heroku and deploy it.

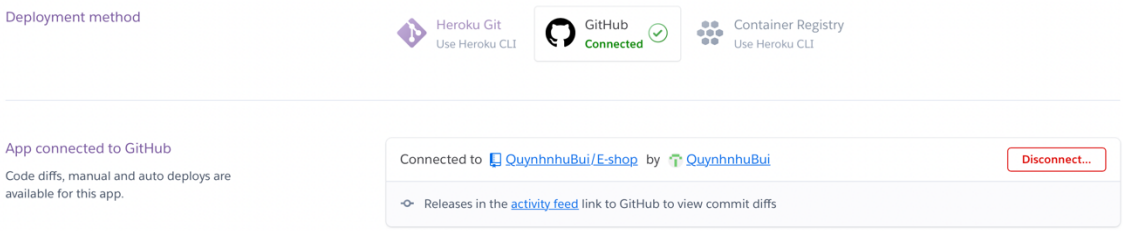


Figure 14. Connect GitHub to Heroku

3.3 Mobile application implementation

This section covers the process of mobile application implementation. It includes how to create the application and how the mobile application looks like. To create this application, I chose React Native as the development platform. By using React Native, I can create an application for both Android and iOS using only one codebase. First to create a React Native project, setting up the development environment is required. React Native provides a comprehensive document on their website about how to set up a coding environment. By following the instruction, the environment can be set up successfully. Next is to initialize a new project, I used *npx react-native init Eshop* command to start a completely new project. After creating the project, I organized it into various folders to make the project easier to understand.

For every application, navigation plays a crucial role, therefore, I created the navigation for the application first. To create the navigation, I used a react-navigation library. For every library that was used in this application, they were installed by using the *npm install* command. The main navigation is the bottom tab navigation that includes four tabs, which are Home, Cart, User, and Admin. Each tab will be a navigation stack. Each screen must be defined in its navigation. Figure 15 displays the mobile application's navigation.

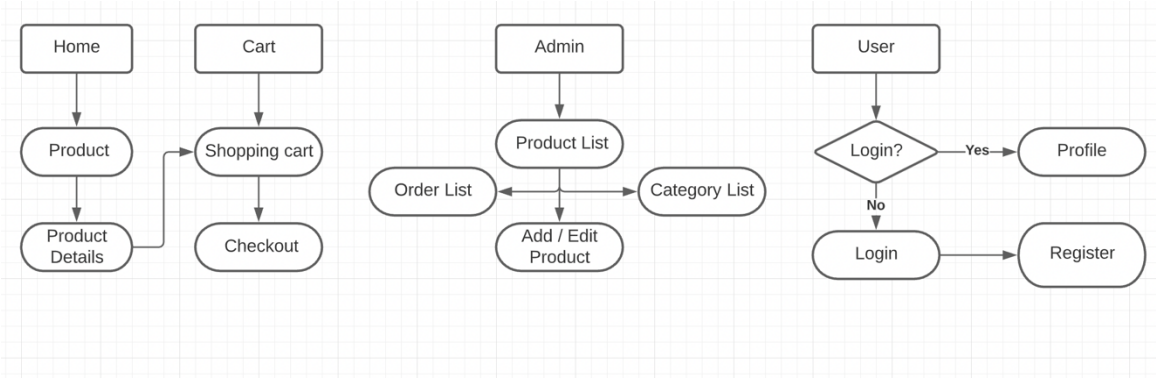


Figure 15. Navigation

For tab navigation, each screen is a tab and a tab icon is required. In each tab, the component must be defined. As mentioned above, each tab also contains a navigation stack so that in Figure 16 the component for the Home tab is a stack instead of a screen.

```
const Tab = createBottomTabNavigator();

const TabNavigation = () => {
  const context = useContext(AuthGlobal);

  return (
    <Tab.Navigator
      initialRouteName="Home"
      screenOptions={{
        keyboardHidesTabBar: true,
        showLabel: false,
        tabBarActiveTintColor: '#ff6600',
      }}>
      <Tab.Screen
        name="Home"
        component={ProductNavigation}
        options={{
          tabBarIcon: ({color}) => <Icon name="home" color={color} size={30} />,
          headerShown: false,
        }}
      />
```

Figure 16. Bottom tab navigation

Another type of navigation in the mobile application is stack navigation. Stack navigation works similarly to tab navigation. There are five navigation stacks in the project which are Admin Navigation, Product Navigation, User Navigation, and Cart Navigation. Screens inside the stack must be declared. In the configuration, I also defined a custom header so every screen has the same header. For Checkout Navigation, I also used tab navigation but it is a top tap navigator. Figure 17 is an example of a navigation stack. Figure 18 displays the configuration for top tab navigation.

```
const ProductNavigation = () => {
    return (
        <Stack.Navigator >
          <Stack.Screen
            name="productList"
            options={({route}) => ({
              title: 'Product',
              headerTitleStyle: {
                fontWeight: 'bold',
              },
              headerStyle: {
                backgroundColor: '#ffffff',
                shadowColor: 'transparent',
                shadowRadius: 0,
                shadowOffset: {
                  height: 0,
                  width: 0,
                },
                elevation: 0,
              },
              headerBackTitle: ' ',
              headerTitleAlign: 'center',
            })}
            component={ProductList}></Stack.Screen>
```

Figure 17. Stack navigation

```
const CheckoutNavigation = () => {
  return (
    <Tab.Navigator
      screenOptions={{
        swipeEnabled: false,
      }}>
      <Tab.Screen name="shipping" component={CheckoutContainer} />
      <Tab.Screen name="payment" component={Payment} />
      <Tab.Screen name="confirm" component={ConfirmContainer} />
    </Tab.Navigator>
  );
};

export default CheckoutNavigation;
```

Figure 18. Top tab navigation

All the available screens are in the Screen folder. This would help organize the project making it easier to find files. In this part, I will describe the screens by their navigation stack. The first stack is the Product navigation stack. It includes two screens, which are the Product screen and Product detail screen. In the Product screen, a list of all products is shown and there is also a search bar for

users to search for any products. Moreover, users can also choose to filter products by category. By clicking on the category name, only products that belong to that category are shown. In the Product detail screen, users are allowed to add products to the shopping cart. Figure 19 includes a picture of product and product detail screens.

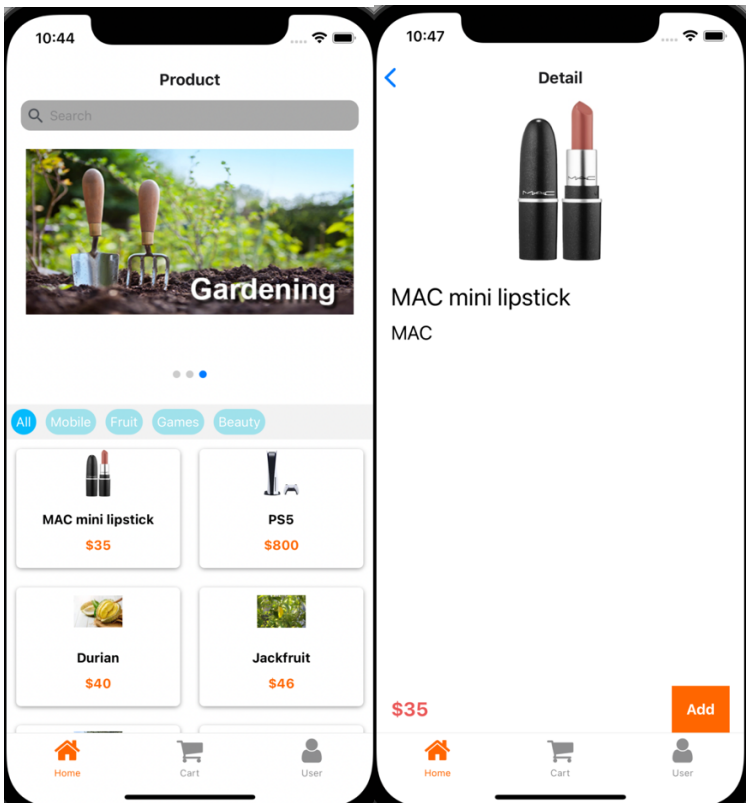


Figure 19. Product screens

The list of items is shown in a card item form. This card item is a custom component. When users click on an item, it navigates to the product detail screen to display more information about the product. To create the custom card item, I use the Touchable Opacity component which allows users to click on it. To render it as a list, I used the Flatlist component. Flatlist allows rendering a list of data under a custom view. Figure 20 displays the implementation process.

```
const CardItem = props => {
  const {name, image, price, countInStock, marginRight} = props;
  return (
    <TouchableOpacity
      style={{...styles.card, marginRight: marginRight}}
      onPress={props.onPress}>
      <View>
        <Image
          style={styles.image}
          resizeMode="contain"
          source={{uri: image ? image : null}}
        />
        <View style={{padding: Sizes.s20, alignItems: 'center'}}>
          <Text style={styles.name}>{name}</Text>
          <Text style={styles.price}>${price}</Text>

        </View>
      </View>
    </TouchableOpacity>
  );
};
```

```
  <FlatList
    showsVerticalScrollIndicator={false}
    key={'grid'}
    numColumns={2}
    data={productList}
    keyExtractor={item => item.id}
    renderItem={({item, index}) => (
      <CardItem
        {...item}
        marginRight={
          index === productList.length - 1 &&
          productList.length % 2 != 0
            ? Sizes.s50
            : null
        }
        onPress={() => {
          navigation.navigate('detail', {id: item.id});
        }}
        // onPressAdd={() => {
        //   props.addToCart(item);
        // }}
      />
    )}
  />
```

Figure 20. Card item implementation

To have the product data, I can get it from the API path. The **/products/getProductList** path is used to get the product list while **/products/getProduct/:id** is to get product details. **/products/searchProduct** is used when the search bar text changes to get products that their name includes that keyword. 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.

```
const getProductList = (id) => {
  axios
    .get(`${url}products/getProductList?categories=${id}`)
    .then(res => {
      if (res.data.success == true) {
        setList(res.data.productList);
        setLoading(false);
      }
    })
    .catch(error => {
      console.log('Fetch api error');
    });
};
```

Figure 21. Get list product request

When the users click Add button in the product detail screen, the product is added to the cart. Users can add as many products as they want. When the Add button is clicked, the product id is stored to use that id to get the product on the cart screen. The product id is stored in reducer. When users click the button, it dispatches the a predefined action to store data in reducer. Reducer can be used everywhere in the project, therefore, storing data in reducer allows that data can be accessed on different screens.

```
export const ADD_TO_CART = 'ADD_TO_CART';
export const DELETE_FROM_CART = 'DELETE_FROM_CART';
export const EMPTY_CART = 'EMPTY_CART';

export const addToCart = product => {
  return {
    type: ADD_TO_CART,
    product,
  };
};
export const deleteFromCart = data => {
  return {
    type: DELETE_FROM_CART,
    data,
  };
};
export const emptyCart = () => {
  return {
    type: EMPTY_CART,
  };
};
```

```
import {ADD_TO_CART, DELETE_FROM_CART, EMPTY_CART} from '../action/cartAction';


const cartReducers = (state =[], action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return [...state, action.product]
    case DELETE_FROM_CART:
      return state.filter(item => item !== action.data);
    case EMPTY_CART:
      return (state = []);
    default:
      return state;
  }
};

export default cartReducers;
```

Figure 22. Actions and reducers.

To render the selected products, I used the selected ids that are stored in the reducer and the **/products/getProduct/:id** path to get the product information. At the bottom of the screen, there are Clear and Checkout buttons. The Clear button is used to delete all items in the cart. The Checkout button navigates to the checkout process. Users are also able to delete a single item by swiping left on that item. The Clear and Delete buttons dispatch EMPTY_CART and DELETE_FROM_CART actions that are shown in Figure 23.

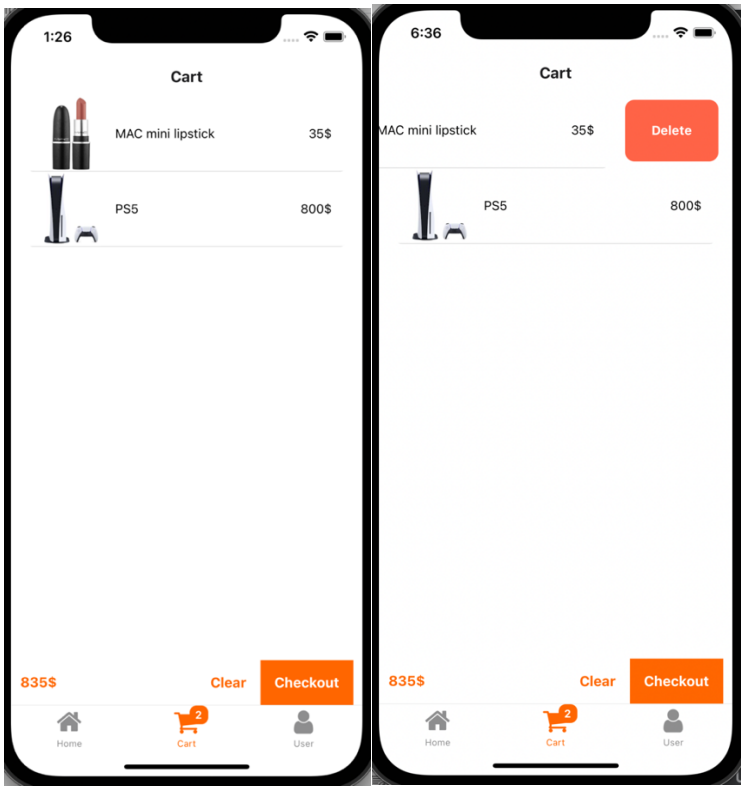


Figure 23. Cart Screen

After clicking Checkout, the application navigates to the checkout process. This process has three steps: shipping, payment, and confirmation. In this process, I used a top tab bar that includes three screens: shipping, payment and confirm. For the shipping tab, users are required to fill in all the shipping information. If any information is missing, there is an error message under each missing input.

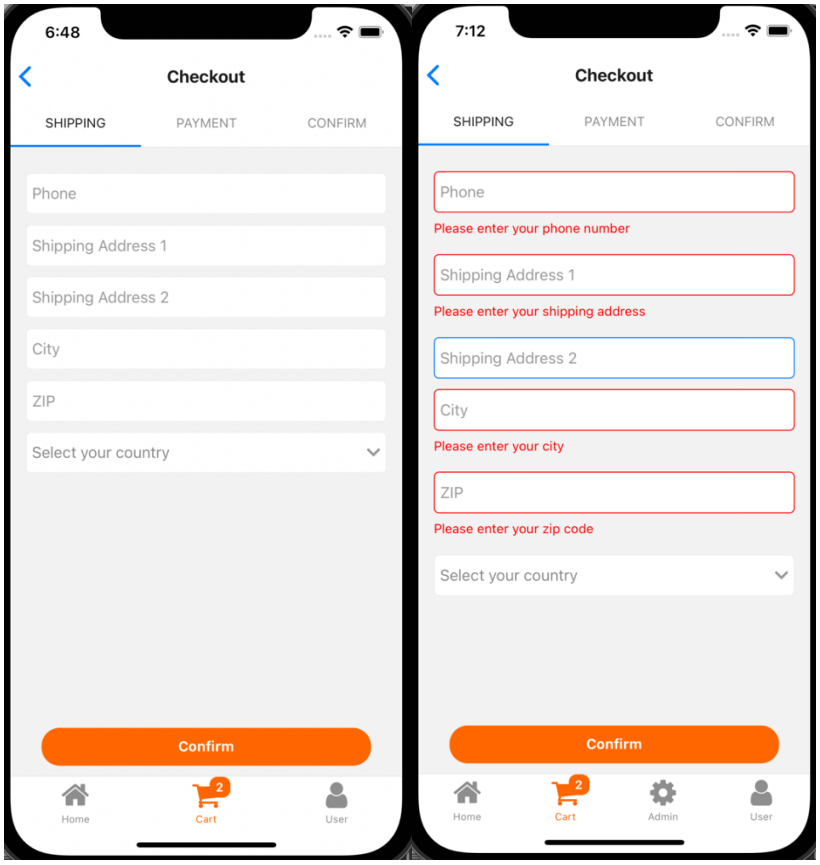


Figure 24.Shipping information screen

For showing the purpose of error message, I created a custom input called TextField that displays a message when users click out of the input without filling it. When the input is focused, the message is clear. By using refs, the application can access the appropriate component and execute the code for that component. This custom input is created based on the React Native TextInput component. Figure 25 shows how to implement the TextField.

```
<TextField
  id={'phone'}
  onBlur={text => {
    if(text == ""){
      phoneNumber.current.error('phone','Please enter your phone number')
    }
  }}
  onFocus={() => {
    phoneNumber.current.clearError('phone')

  }}
  editable={true}
  ref={phoneNumber}
  isRequired={true}
  onChangeText={text => {
    setPhone(text.trim())
  }}
  placeholder="Phone"

/>
```

Figure 25. Custom input

The next step is to choose a payment method. There are three methods which are cash on delivery, bank transfer, and card payment. However, in this application, the payment gateway is not implemented yet. In the Confirm tab, there is a summarization for shipping information and selected products, therefore, users can check all information before placing an order. When users press the Place order button, the order detail is sent to the server using **/orders/createOrder** path. When an order is placed successfully, there is a pop-up message to indicate that the order has been made and the cart is clear.

```
const onPress = () => {
  axios
    .post(`${url}orders/createOrder`, confirm.order.order)
    .then(res => {
      if (res.data.success == true) {
        navigation.navigate('cart');
        props.emptyCart();
        Toast.showWithGravity('Order successfully', Toast.LONG, Toast.TOP);
      }
    })
    .catch(error => {
      console.log('Fetch api error');
    });
};
```

Figure 26. Place order function

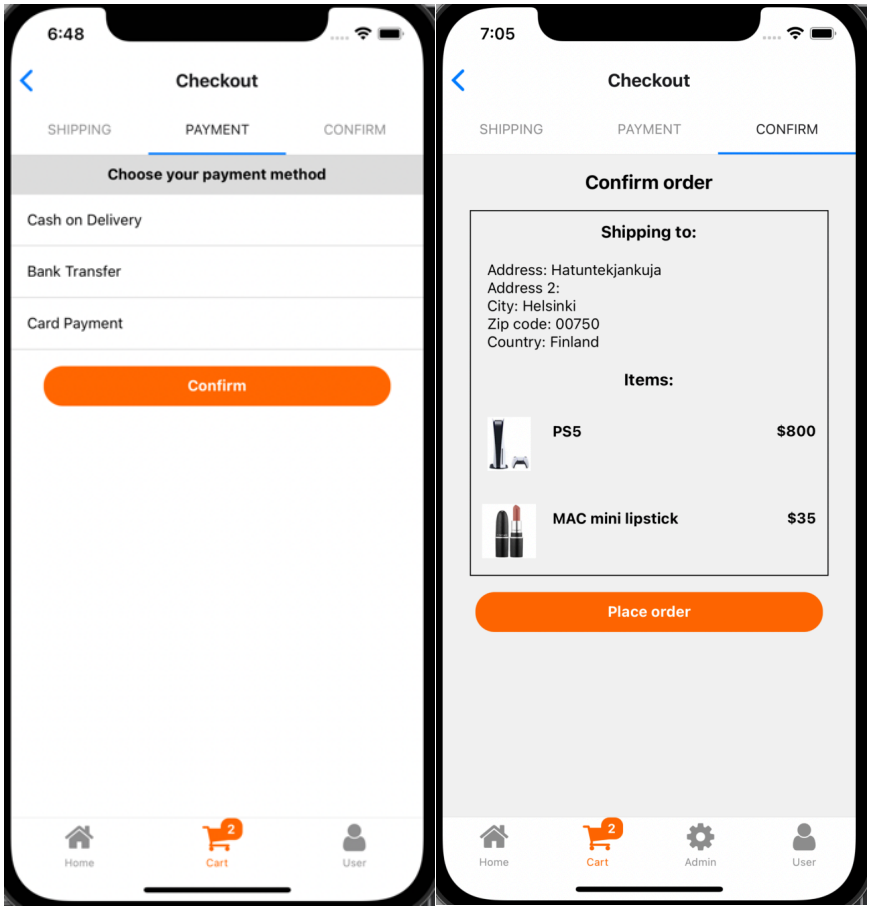


Figure 27. Payment and Confirm screens

The next part was to implement the User tab. If users are not signed in, when they press the User tab, it shows the login screen. On the login screen, the user can log in with an existing account or register for a new account. The **/users/login** path is used to login user and returns a token. A token is required for making other requests. The token is stored in local storage called async storage. The token value is stored in the storage until I removed it or the application is uninstalled. Previously, when I implemented the server, I generated the token that included the value of the isAdmin field, consequently, in the mobile application, I can decode it to get the isAdmin value. This value is used to verify if the login account is an admin or not. Moreover, when logging in successfully, the user's information is saved in a reducer similar to the cart process. From there, the user's information can be accessed on every other screen throughout the application. The application uses the token to check whether the account has admin rights or not. If the user is admin, in the bottom tab it will show the Admin

tab. Figure 28 shows the implementation of the login process and saving the information to the reducer.

```
export const loginUser = (user, dispatch) => {
  fetch(`${url}users/login`, {
    method: 'POST',
    body: JSON.stringify(user),
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
  })
    .then(res => res.json())
    .then(data => {
      console.log(data);
      if (data) {
        if (data.success == true) {
          const token = data.token;
          const decoded = jwt_decode(token);
          AsyncStorage.setItem('token', token).then(res => {
            console.log(222);
          });
          dispatch(setUserData(decoded, user));
        } else {
          Toast.showWithGravity(data.message, Toast.LONG, Toast.TOP);
        }
      } else {
        logoutUser(dispatch);
      }
    })
    .catch(err => {
      console.log(err);
      Toast.showWithGravity(
        'Something went wrong. Please try again',
        Toast.LONG,
        Toast.TOP,
      );
      logoutUser(dispatch);
    });
};
```

Figure 28. Login request

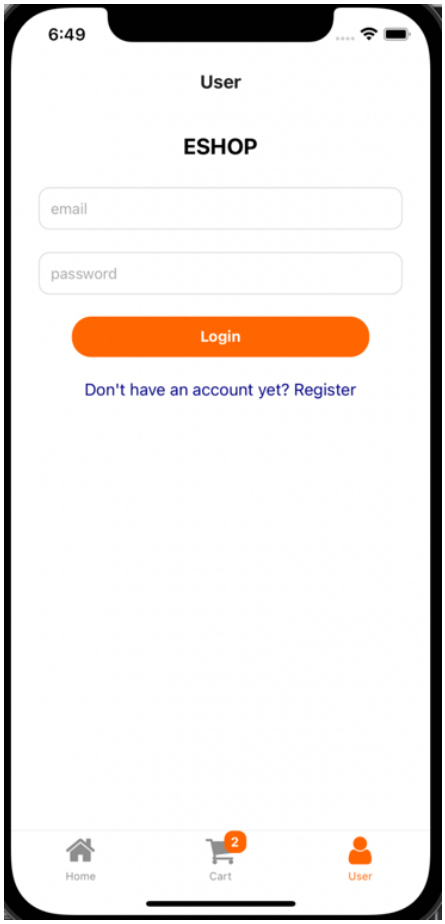


Figure 29. Login screen

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 is name, email, phone number, and password. After registering successfully, the application displays login for users to log in. If users are signed in, when they press on the User tab, it shows the Profile screen, which displays the user's personal information. Users can sign out in the Profile screen also. When users press the Sign out button, all the information that is stored in store and async storage will be removed.

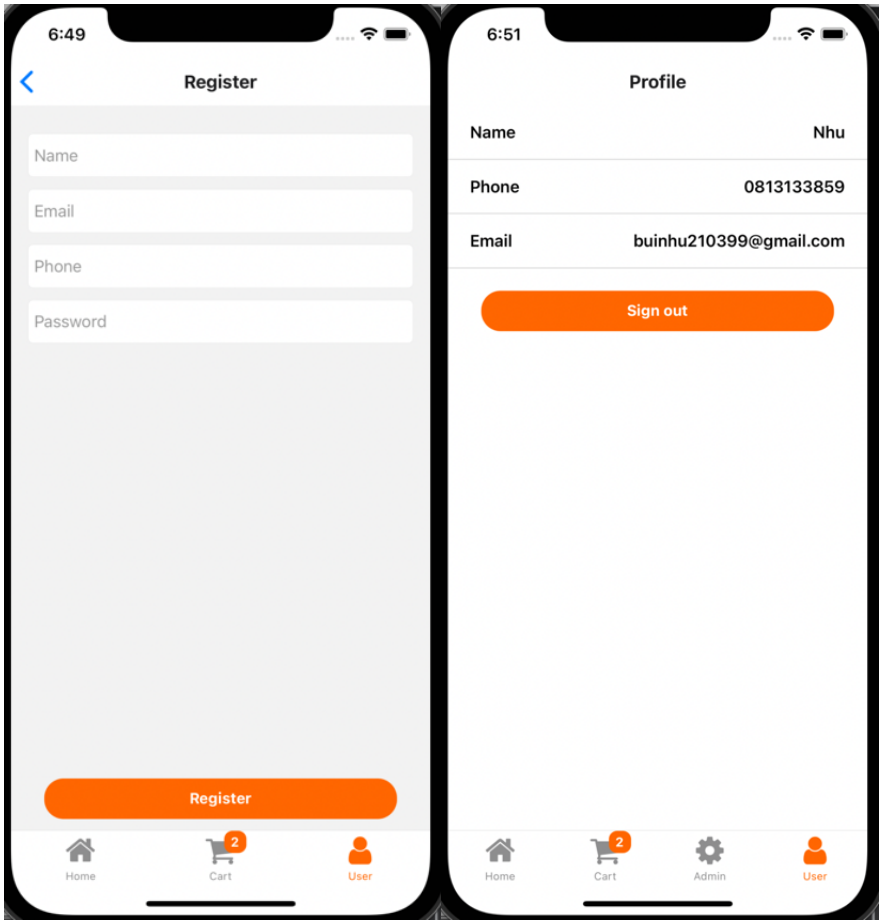
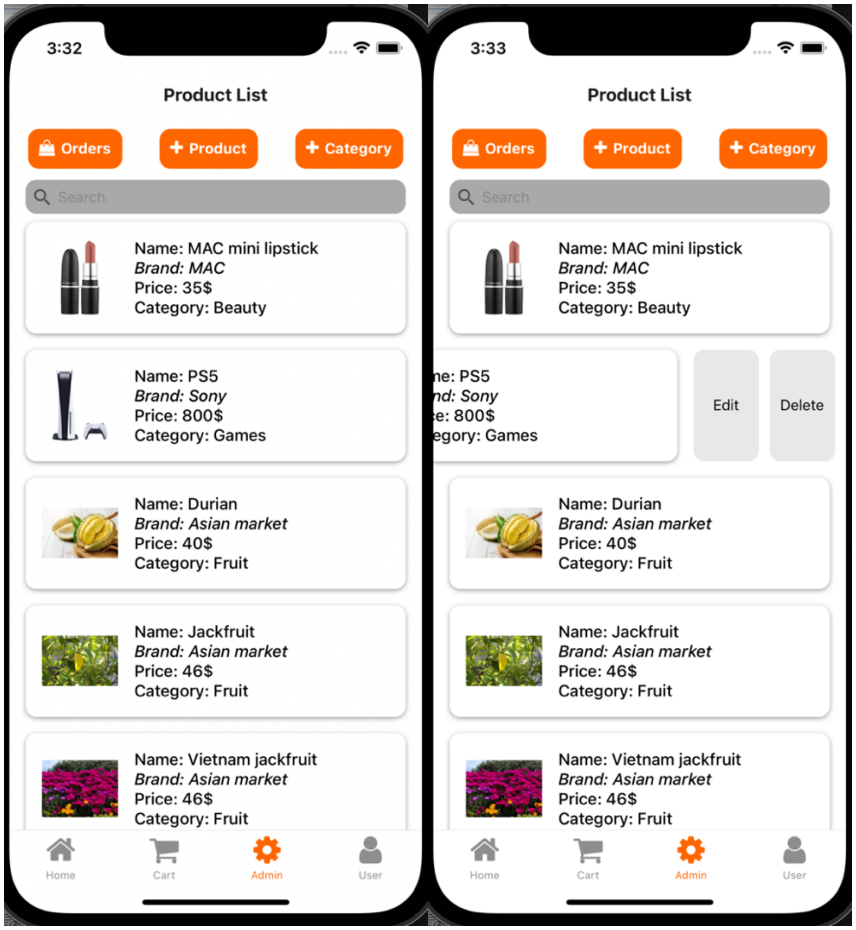


Figure 30. Register and Profile screen

For admins, there is an Admin tab that allows admins to do management. Admins can add new products and categories, update products and orders, and delete products. When pressing the Admin tab, it shows the list of all existing products. An admin can edit or delete the product by swiping left on that product. There are two buttons, Edit and Delete. There is also a search bar for admins to search for products. This search bar has a similar function to the one on the Product screen. Name, brand, price, and category are shown on each card. At the top of the screen, there are three buttons which are Orders, Product, and Category. Pressing the Orders button, the application navigates to the Orders screen where admins can manage all orders. The Product and Category buttons are used to navigate the application to the Product and Category screens. In these two screens, admins can add new products and categories.


3:32
Product List
Orders
+ Product
+ Category
Search
Name: MAC mini lipstick
Brand: MAC
Price: 35$
Category: Beauty
Name: PS5
Brand: Sony
Price: 800$
Category: Games
Name: Durian
Brand: Asian market
Price: 40$
Category: Fruit
Name: Jackfruit
Brand: Asian market
Price: 46$
Category: Fruit
Name: Vietnam jackfruit
Brand: Asian market
Price: 46$
Category: Fruit
Home
Cart
Admin
User
3:33
Product List
Orders
+ Product
+ Category
Search
Name: MAC mini lipstick
Brand: MAC
Price: 35$
Category: Beauty
Edit
Delete
Name: Durian
Brand: Asian market
Price: 40$
Category: Fruit
Name: Jackfruit
Brand: Asian market
Price: 46$
Category: Fruit
Name: Vietnam jackfruit
Brand: Asian market
Price: 46$
Category: Fruit
Home
Cart
Admin
User


6:54
Category
Mobile
Fruit
Games
Beauty
Add category
Add
2
Home
Cart
Admin
User

Figure 31. Product List and Category screen

On the Orders screen, there is a list of orders that includes information about the orders. In this screen, admins are allowed to update the orders' status. When admins press the Update button, it displays a bottom sheet that contains all statuses. From there, admins can choose which one to update. After updating success, it navigates to the Product List screen. To custom the bottom sheet, I used the Modal component. To render the list of status, I used Flatlist and each item is a button. When admins press one button, it shows the check icon to mark the selected options. When touching outside the bottom sheet, it closes the sheet. Figure 32 displays the Orders screen and the bottom sheet.

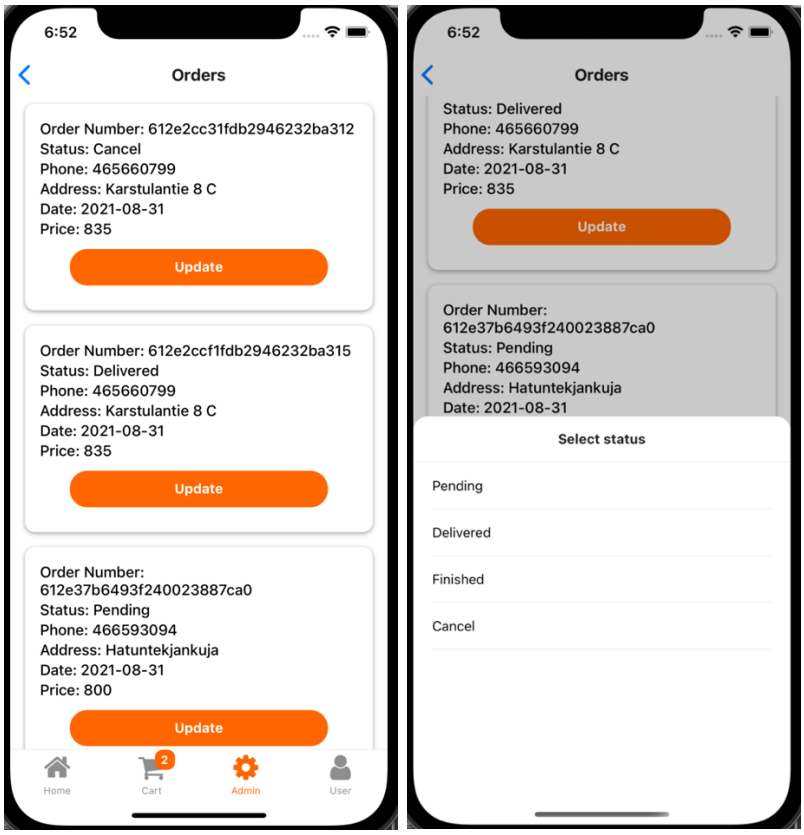


Figure 32. Orders screen

To add new products, admins need to fill in all the information for the product. The information includes brand, name, price, count in stock, description, category, and picture. Other fields use the custom TextField that was created to use in the Register screen except for picture and category. For the category field, it displays the bottom sheet similar to one in the Orders screen. There is a new function in this screen which is adding an image. To add images, I used a library

called react-native-image-picker. This library does not require additional settings for permission. By using this library, the application can access the device's photos. The result contains the name, type, URI, and so on. From that information, I can render the image and also use it for sending the image to the server.

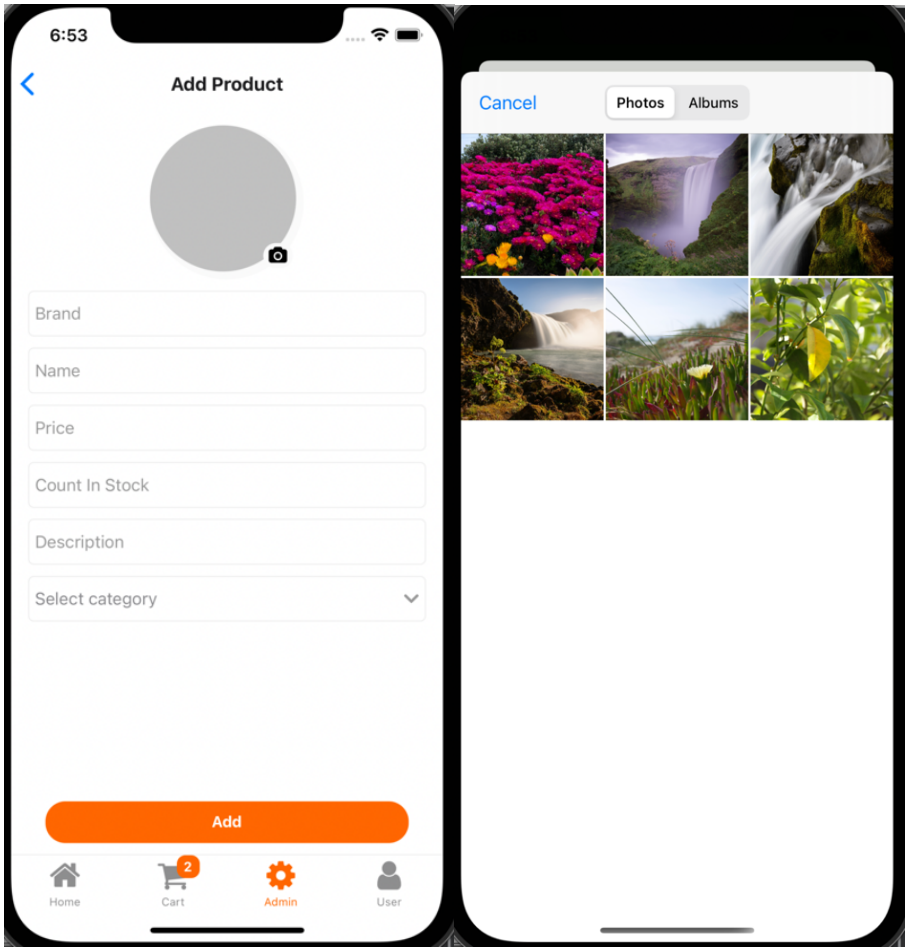


Figure 33. Add product screen

```
const selectImage = () => {
  const options = {};
  launchImageLibrary(options, res => {
    if (res.assets) {
      setImage(res.assets[0].uri);
      setRes(res.assets[0]);
    }
  });
};
```

Figure 34. Select image function

Unlike other requests, this request is made differently. Usually, the data sent to the server is in JSON format. However, in this request, it is in FormData. I created a new FormData and appended all the required fields to the FormData. For the image, the different operating system has a different URI. Because of that, in iOS, I had to change the image's URI to make the request. After creating a new product successfully, the application navigates to the Product List screen and displays a success message.

```
let body = new FormData();

body.append('name', name);
body.append('price', price);
body.append('description', description);
body.append('countInStock', count);
body.append('brand', brand);
if (imageRes) {
  body.append('image', {
    name: imageRes.fileName,
    type: imageRes.type,
    uri:
      Platform.OS === 'android'
        ? imageRes.uri
        : imageRes.uri.replace('file://', ''),
  });
}
body.append('category', category);
```

Figure 34. Create new FormData

4 CONCLUSION

The goal of this thesis is to create an online shopping mobile application and to do research to put it into action. The application is for everyone who wants to do shopping online. The theoretical background section has covered the study

concepts and definitions of software development, methods and technologies to implement the thesis. This section explains the concept of database and server, the definition of different platforms and tools that are used in the implementation section. This theoretical background part makes it straightforward to understand the implementation section.

The practical implementation part describes the process of making the server and mobile application. It also provides the planning on how to implement the application. The pictures of all the screens in the application is displayed in this section. By applying the methods and technologies, the application was developed successfully. With the provided tools, the application fulfills the predefined goals in the beginning. The application still needs to be developed more in the future. More features should be included to provide the best user experience. Possible further development ideas are sending an email confirmation, changing email and password, adding comments and so on.

To ensure that the application is working, I have done some testing on the application. I tested it based on my experience when using other applications. I tried to ensure that there are no crashes in the application. The server side of this thesis was deployed to Heroku so that the API can be used in public. However, the application has not been deployed to any platforms. The requirement for publishing the application to App Store ang Play Store is quite complex so I have not done it in this thesis.

REFERENCES

Auth0. No date. Introduction to JSON Web Tokens. WWW document. Available at: https://jwt.io/introduction [Accessed 2 Sep 2021]

Boris, S. 2019. 8 Software Development Models: Sliced, Diced and Organized in Charts. WWW document. Available at: https://www.scnsoft.com/blog/software-development-models [Accessed 30 August 2021]

Facebook Inc. 2021. React Native. WWW document. Available at: https://reactnative.dev/ [Accessed 28 August 2021]

IBM Cloud Education. 2021. REST APIs. WWW document. Available at: https://www.ibm.com/cloud/learn/rest-apis [Accessed 28 August 2021]

Jignesh, S. 2021. Comparing Nodejs vs Java: Your Backend Tech Stacks Explained. WWW document. Available at: https://www.simform.com/blog/nodejs-vs-java/ [Accessed 30 August 2021]

Louise, M. 2018. E-commerce: The Past, Present and Future. WWW document. Available at: https://www.spiralytics.com/blog/past-present-future-ecommerce/ [Accessed 5 August 2021]

MongoDB, Inc. 2021. What is MongoDB ?. WWW document. Available at: https://www.mongodb.com/what-is-mongodb [Accessed 28 August 2021]

Mongoose. No date. Schema. WWW document. Available at: https://mongoosejs.com/docs/guide.html [Accessed 1 Sep 2021]

Npm. No date. About npm. WWW document. Available at: https://docs.npmjs.com/about-npm [Accessed 30 August 2021]

OpenJS Foundation. No date. Introduction to Nodejs. WWW document. Available at: https://nodejs.dev/learn [Accessed 28 August]

Saket, K. 2018. How React Native Works ?. WWW document. Available at: https://www.codementor.io/@saketkumar95/how-react-native-works-mhjo4k6f3 [Accessed 28 August 2021]

Sergey, K. 2019. Mobile App Development Approaches Explained. WWW document. Available at: https://railsware.com/blog/native-vs-hybrid-vs-cross-platform/#Native_app_development [Accessed 28 August 2021]

Taha, S. 2021. What is Nodejs: A Comprehensive Guide. WWW document. Available at: https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs [Accessed 30 August 2021]