COMP 6251
WEB & CLOUD APPLICATIONS DEVELOPMENT

Managest: A web-based application for connecting Restaurant Owners, Drivers, and Customers

Created by: Team 22

Paskal Stepien: pbs1e21@soton.ac.uk

Yijiang Wang: yw20u21@soton.ac.uk

Liusong He: lh2u21@soton.ac.uk

University of Southampton
School of Electronics and Computer Science

COMP6251 Coursework - Proposed Marks Distribution

To be submitted as a part of your report submission
Record here your proposed distribution of the total number of marks awarded to your team.

Please enter all requested details and the percentage of the **total** team effort contributed by each team member. The contribution percentages **must** total 100%.

Each team member **must** sign and date the form before submission to confirm that they **agree** with the **proposed** distribution.
Only one fully completed form per team is necessary (but see below when this is not possible), to be submitted as the first page of your team report.
Your proposal will normally be followed but other evidence such as *logbooks* may be considered. **The course leader's decision is final.**

Group:	22

Student ID	Email ID	Percentage	Signature	Date
33356904	yw20u21@soton.ac.uk	33.33%	[illegible]	17/05/2022
32776853	lh2u21@soton.ac.uk	33.33%	Liusong He	17/05/2022
33374864	pbs1e21@soton.ac.uk	33.33%	[illegible]	17/05/2022

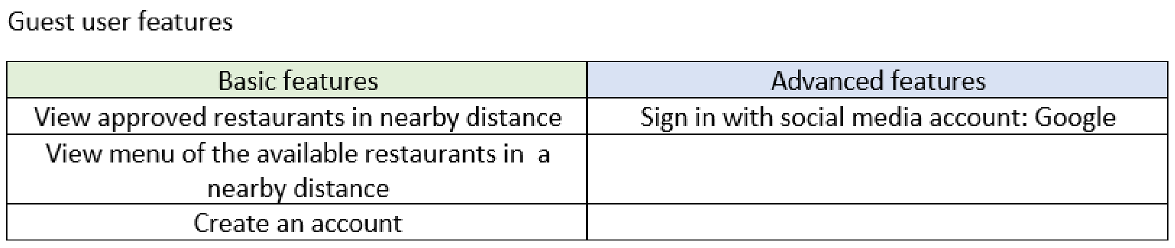
If your team is unable to agree on a proposed distribution, please talk to the module leader before submitting your form. Your team should **avoid this,** if at all possible, because it will be interpreted as demonstrating a general lack of competence.

Word count: 1936

1. Overview

This web application was developed using the React framework for the front end, and Firebase, Node.js, and express for the backend, more of which will be explained further in this report, following the specification provided for this assignment.

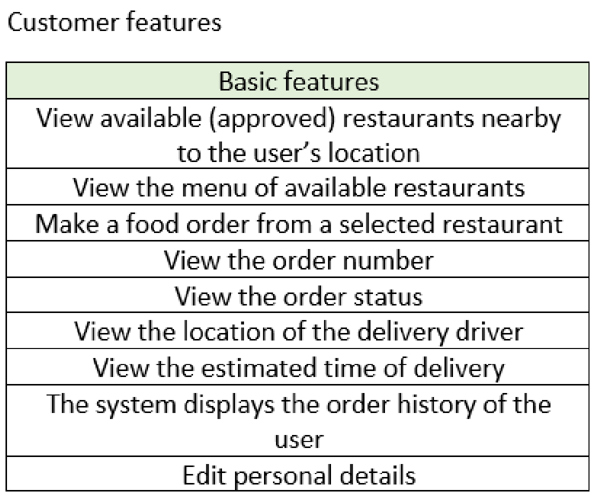
We started development of this application with requirements gathering, and sorting of the requirements into the appropriate stakeholder/role. Below is a figure representing the tables of the features that have been completed in this assignment, sorted by the corresponding role.



Guest user features

Basic features	Advanced features
View approved restaurants in nearby distance	Sign in with social media account: Google
View menu of the available restaurants in a nearby distance	
Create an account	

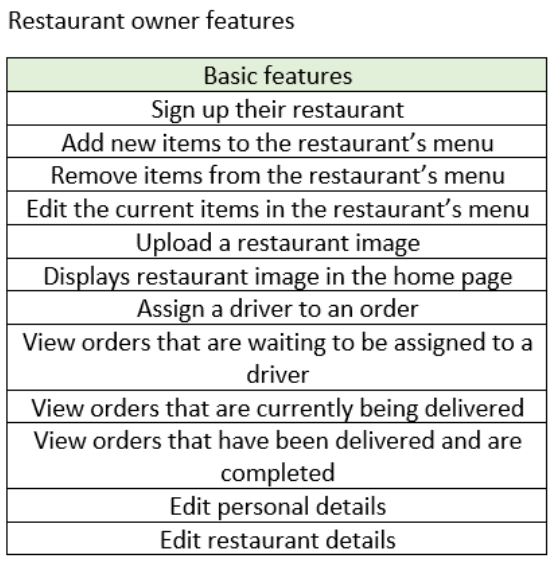
Figure 1 A screenshot of the Guest User Features table, listing the basic features and the advanced features that were implemented



Customer features

Basic features
View available (approved) restaurants nearby to the user's location
View the menu of available restaurants
Make a food order from a selected restaurant
View the order number
View the order status
View the location of the delivery driver
View the estimated time of delivery
The system displays the order history of the user
Edit personal details

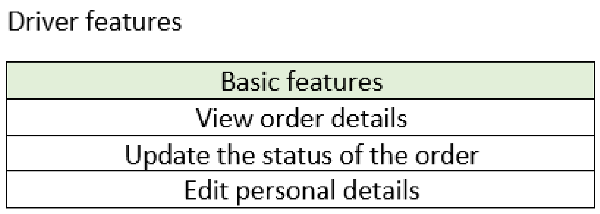
Figure 2 A screenshot of the Customer User Features table, listing the basic features that were implemented



Restaurant owner features

Basic features
Sign up their restaurant
Add new items to the restaurant's menu
Remove items from the restaurant's menu
Edit the current items in the restaurant's menu
Upload a restaurant image
Displays restaurant image in the home page
Assign a driver to an order
View orders that are waiting to be assigned to a driver
View orders that are currently being delivered
View orders that have been delivered and are completed
Edit personal details
Edit restaurant details

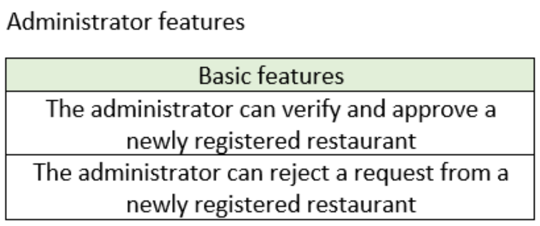
Figure 3 A screenshot of the Restaurant Owner User Features table, listing the basic features that were implemented



Driver features

Basic features
View order details
Update the status of the order
Edit personal details

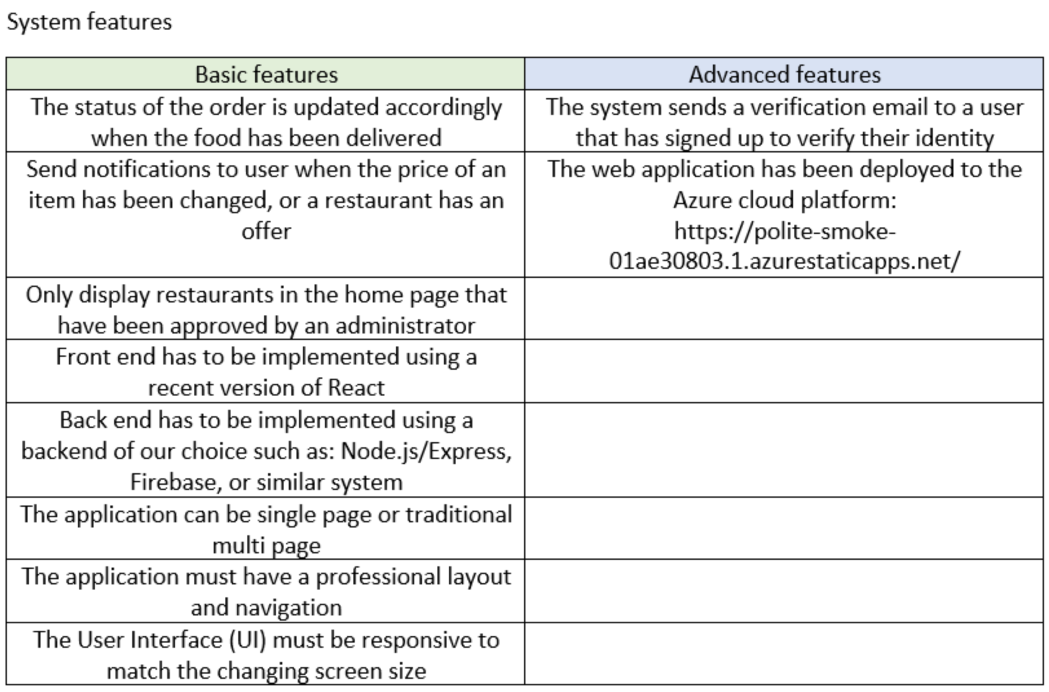
Figure 4 A screenshot of the Driver User Features table, listing the basic features that were implemented



Administrator features

Basic features
The administrator can verify and approve a newly registered restaurant
The administrator can reject a request from a newly registered restaurant

Figure 5 A screenshot of the Administrator User Features table, listing the basic features that were implemented



System features

Basic features	Advanced features
The status of the order is updated accordingly when the food has been delivered	The system sends a verification email to a user that has signed up to verify their identity
Send notifications to user when the price of an item has been changed, or a restaurant has an offer	The web application has been deployed to the Azure cloud platform: https://polite-smoke-01ae30803.1.azurestaticapps.net/
Only display restaurants in the home page that have been approved by an administrator	
Front end has to be implemented using a recent version of React	
Back end has to be implemented using a backend of our choice such as: Node.js/Express, Firebase, or similar system	
The application can be single page or traditional multi page	
The application must have a professional layout and navigation	
The User Interface (UI) must be responsive to match the changing screen size	

Figure 6 A screenshot of the System Features table, listing the basic features and the advanced features that were implemented

2. Design

2.1 Overall Business Logic

This project is aimed to build a website that connects restaurants and customers. The figure below shows an activity diagram of the restaurant owner's business logic in this project.

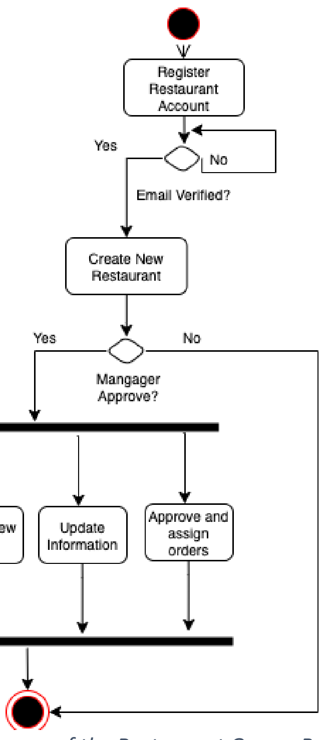


Figure 7 Activity Diagram of the Restaurant Owner Business Logic

As we can see, the restaurant owner will firstly register an account and they will receive a verification email. Once the restaurant owner finished verifying their email, they can create a new restaurant with restaurant information and image. Then the manager will approve or decline this new restaurant creation request. If the restaurant is approved, then the restaurant owner can create new food with detail, update their food information, approve(decline) an order, and assign a driver to this order.

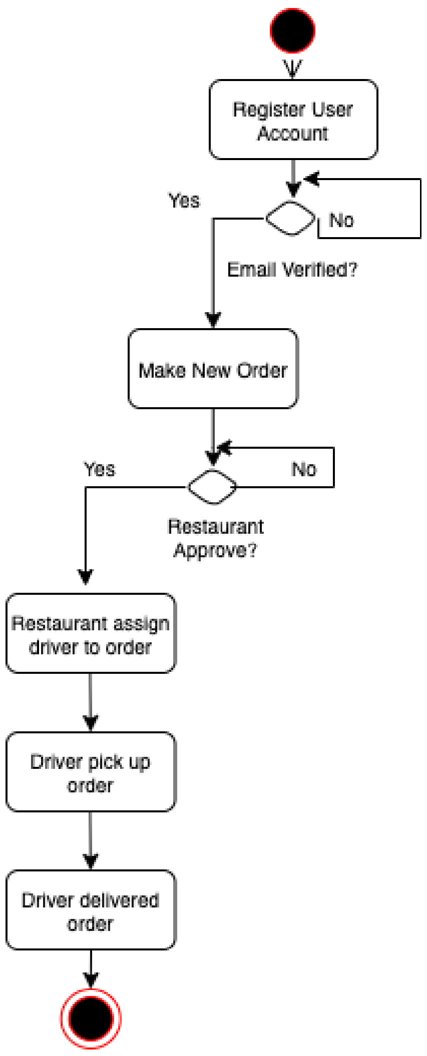


Figure 8 Activity Diagram of User Making a New Order

For normal users and drivers, first, they can register a new account by selecting a different role and they also need to verify their email by clicking the link sent to their email. For each order, users can only order from a single restaurant, and they can only make an order when they are logged in. When a restaurant received an order from a user, they can approve it and assign a driver that is not in the delivering status to it. When the driver is assigned, the status of the driver will change from “waiting” to “assigned”, and the order status will change from “placed” to “delivering”. If the driver clicks the delivered button on their page, the order status will change to “delivered”. During the order delivery process, the user could see an estimated time of delivery and a map of the position of the driver.

2.2 Architecture

This project was separated into two parts, the frontend written in React and the backend written in Node.js with express. In the backend, the NoSQL database is stored in the Firebase Firestore. The fronted React application communicates with the backend by Restful API. This architecture guarantees that the project is highly maintainable. All the authentication is based on the Firebase Firestore and the 2-factor authentication is written by ourselves.

2.3 User Interface Design

As with any software engineering project, we started off with a very important stage, which was the design of the application. Specifically, we started off with the User Interface (UI) design. This application was intended as a traditional multi-page application, therefore we started with designing of the various pages that would be implemented in this application. This also meant that we would have to focus on designing an appropriate navigation menu for the user.

All the pages were designed with certain key design principles in mind. Most commonly applied principles were those of Jakob Nielsen and Steve Krug. Jakob Nielsen's 10 usability heuristics for UI design[1], were implemented as they are very important in designing user interfaces, as they provide key aspects that users desire.

In addition, we followed teachings from Steve Krug's book titled 'Don't make me think'[2], which is a book specifically written about good user interface design principles for web applications. This book, although focuses primarily on design principles for web applications, we believe could be applicable to other platforms.

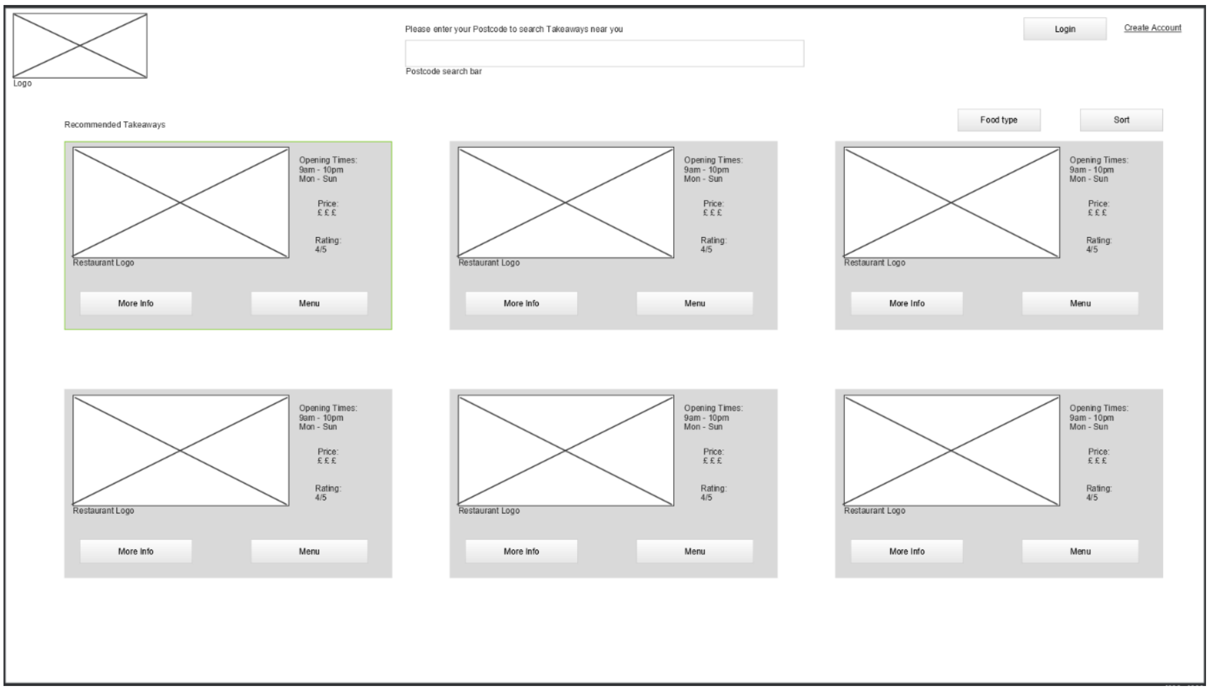


Figure 9 Wireframe design of the home page, demonstrating the various elements used in the homepage, and the appropriate layout for ease of use

As it can be seen in figure 9, many of the principles from Jakob Nielsen and Steve Krug were applied. Firstly, we focused on implementing a minimalistic and aesthetic design, which was even further improved in the development stage, which will be discussed in more depth. This combined with the clear layout of the clickable components, and easily recognisable components such as the buttons, as suggested by Steve Krug, resulting in a very effective user interface for the users.

We have also focused on making sure the majority of the interactive components are in the centre of the page, and there's not too much information to prevent the user from being overwhelmed. Furthermore, we designed this application to have a traditional top bar navigation, for easy recognition of the page that many people are familiar with.

3. Implementation

3.1 Front end - React

This application was developed using a recent version of React, as specified in the coursework specification. This therefore meant that we also implement this application with commonly used principles of developing web applications using React, such as re-usability of components. Because of this we tried to make components re-usable that utilise the use of properties (props) and state variables.

As described in the previous design section, it was explained that this application was designed as a traditional multi-page application, for which we designed an appropriate top bar navigation element. This was made into a re-usable component with properties and state, which was implemented across all the pages.

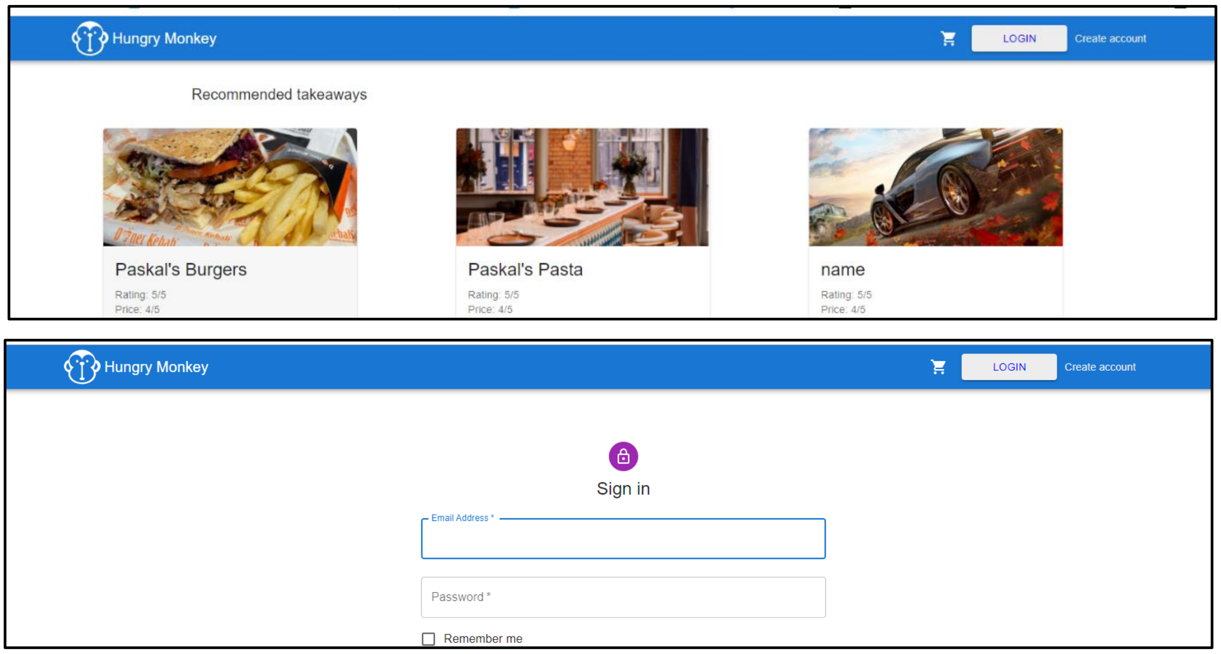
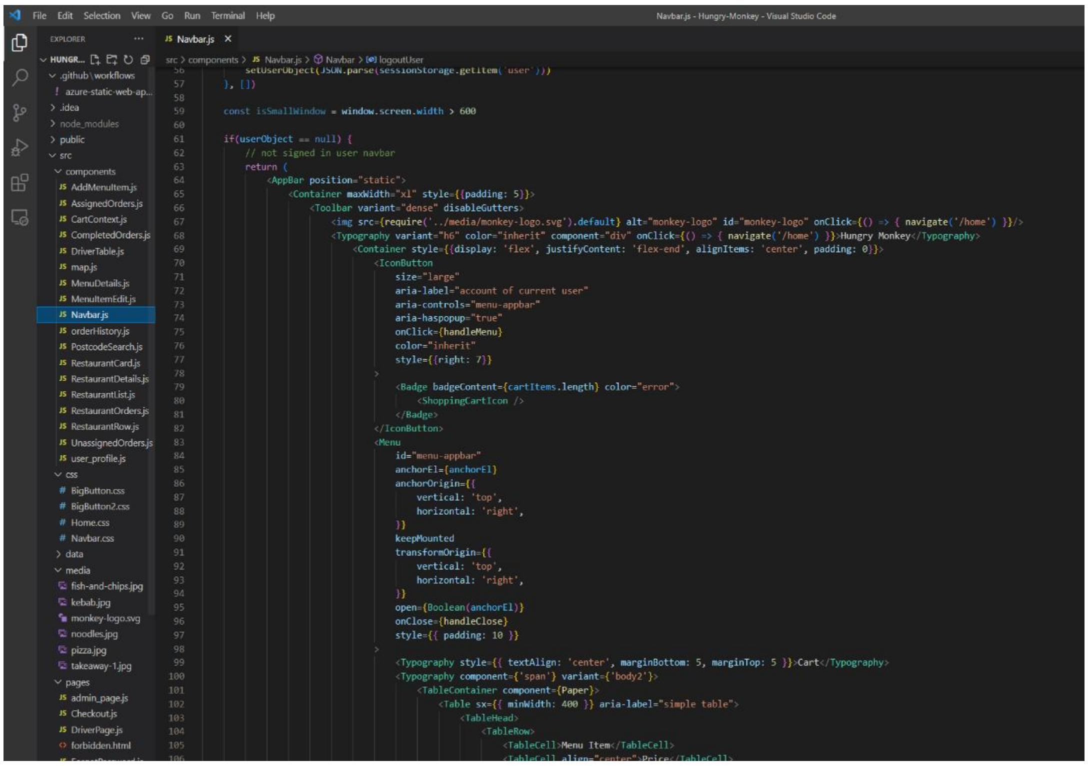
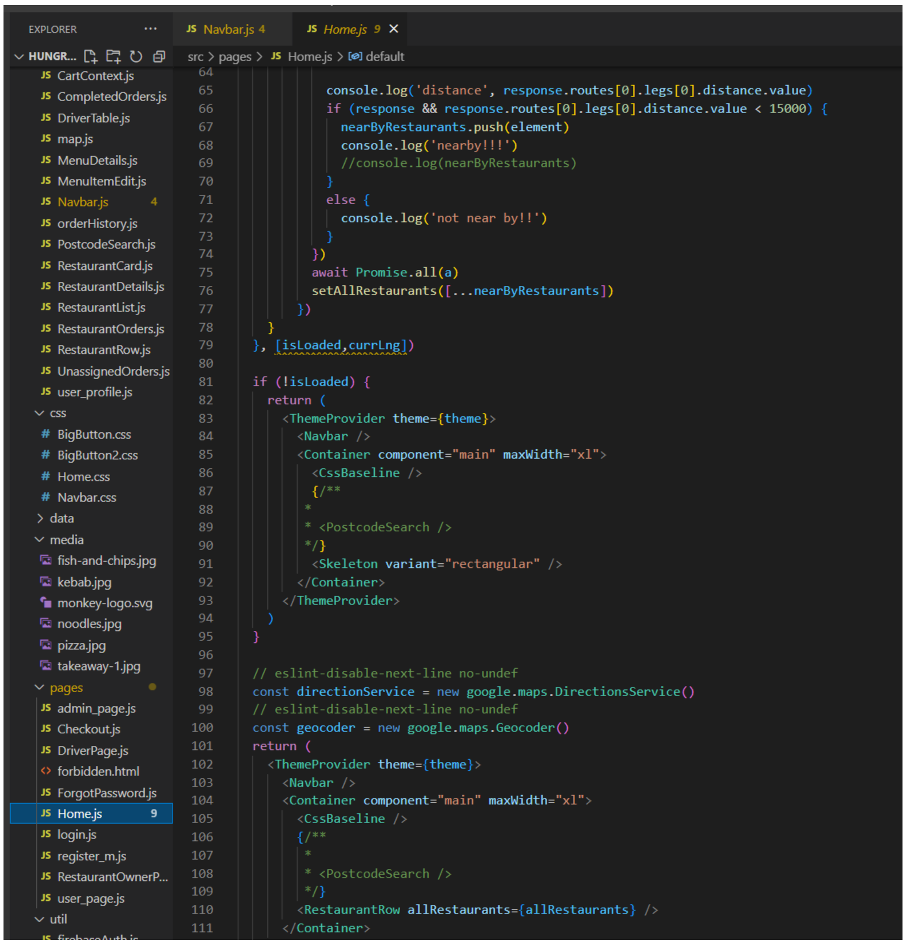


Figure 10 Screenshots of the implemented top bar navigation across variosu pages, such as the home page and the log in page


Navbar.js - Hungry-Monkey - Visual Studio Code
Navbar.js
src > components > Navbar.js > Navbar > logoutUser

```
        setUserObject(JSON.parse(sessionStorage.getItem('user')))
    }, [])

    const isSmallWindow = window.screen.width > 600

    if(userObject == null) {
        // not signed in user navbar
        return (
            <AppBar position="static">
                <Container maxWidth="xl" style={{padding: 5}}>
                    <Toolbar variant="dense" disableGutters>
                        <img src={require('../media/monkey-logo.svg').default} alt="monkey-logo" id="monkey-logo" onClick={() => { navigate('/home') }}/>
                        <Typography variant="h6" color="inherit" component="div" onClick={() => { navigate('/home') }}>Hungry Monkey</Typography>
                            <Container style={{display: 'flex', justifyContent: 'flex-end', alignItems: 'center', padding: 0}}>
                                <IconButton
                                    size="large"
                                    aria-label="account of current user"
                                    aria-controls="menu-appbar"
                                    aria-haspopup="true"
                                    onClick={handleMenu}
                                    color="inherit"
                                    style={{right: 7}}
                                >
                                    <Badge badgeContent={cartItems.length} color="error">
                                        <ShoppingCartIcon />
                                    </Badge>
                                </IconButton>
                                <Menu
                                    id="menu-appbar"
                                    anchorEl={anchorEl}
                                    anchorOrigin={{
                                        vertical: 'top',
                                        horizontal: 'right',
                                    }}
                                    keepMounted
                                    transformOrigin={{
                                        vertical: 'top',
                                        horizontal: 'right',
                                    }}
                                    open={Boolean(anchorEl)}
                                    onClose={handleClose}
                                    style={{ padding: 10 }}
                                >
                                    <Typography style={{ textAlign: 'center', marginBottom: 5, marginTop: 5 }}>Cart</Typography>
                                    <Typography component={'span'} variant={'body2'}>
                                        <TableContainer component={Paper}>
                                            <Table sx={{ minWidth: 400 }} aria-label="simple table">
                                                <TableHead>
                                                    <TableRow>
                                                        <TableCell>Menu Item</TableCell>
                                                        <TableCell align="center">Price</TableCell>
```

Figure 11 Screenshot of Visual Studio Code IDE, demonstrating the Navbar.js file which is the navbar re-usable component, and the contents of the navbar code returning the components inside the navbar element



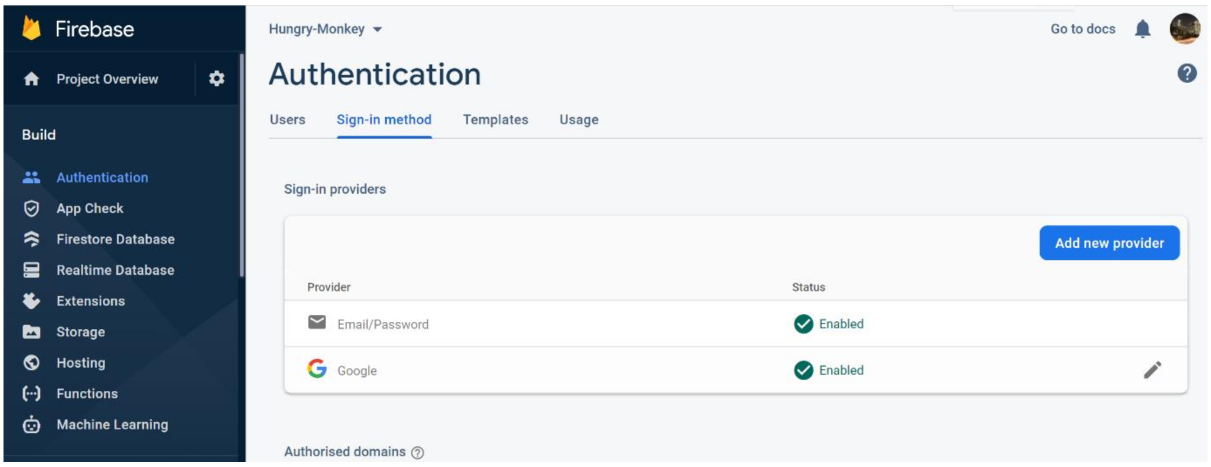
```
              console.log('distance', response.routes[0].legs[0].distance.value)
              if (response && response.routes[0].legs[0].distance.value < 15000) {
                nearByRestaurants.push(element)
                console.log('nearby!!!')
                //console.log(nearByRestaurants)
              }
              else {
                console.log('not near by!!')
              }
            })
            await Promise.all(a)
            setAllRestaurants([...nearByRestaurants])
          })
      }
    }, [isLoaded,currLng])

    if (!isLoaded) {
      return (
        <ThemeProvider theme={theme}>
          <Navbar />
          <Container component="main" maxWidth="xl">
            <CssBaseline />
            {/**
           *
           * <PostcodeSearch />
           */}
            <Skeleton variant="rectangular" />
          </Container>
        </ThemeProvider>
      )
    }

    // eslint-disable-next-line no-undef
    const directionService = new google.maps.DirectionsService()
    // eslint-disable-next-line no-undef
    const geocoder = new google.maps.Geocoder()
    return (
      <ThemeProvider theme={theme}>
        <Navbar />
        <Container component="main" maxWidth="xl">
          <CssBaseline />
          {/**
           *
           * <PostcodeSearch />
           */}
          <RestaurantRow allRestaurants={allRestaurants} />
        </Container>
```

Figure 12 Screenshot of Visual Studio Code IDE, demonstrating the Home.js page in which it can be seen the Navbar custom element is being included in the return statement

We have also used this concept of re-usable components in most of this application, such as menu cards, and most importantly user input forms.

3.2 Advanced Feature implementation – Social media login Google

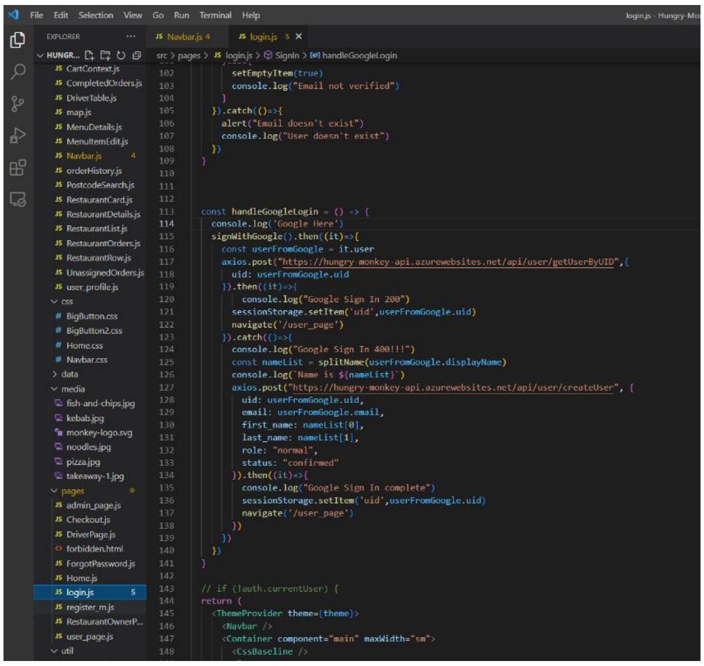
For the implementation of advanced feature of allowing a user to sign in using social media such as Google, we implemented the solution provided from Firebase. Firebase provides. Firstly, we had to select sign-in provider method in the firebase, after which initially we set the authorise domain to be localhost during the development stage. The key change occurred after the application was deployed to the cloud, as we had to update the authorised domain for the google sign in method to the URL of the deployed application.



Provider	Status
Email/Password	Enabled
Google	Enabled

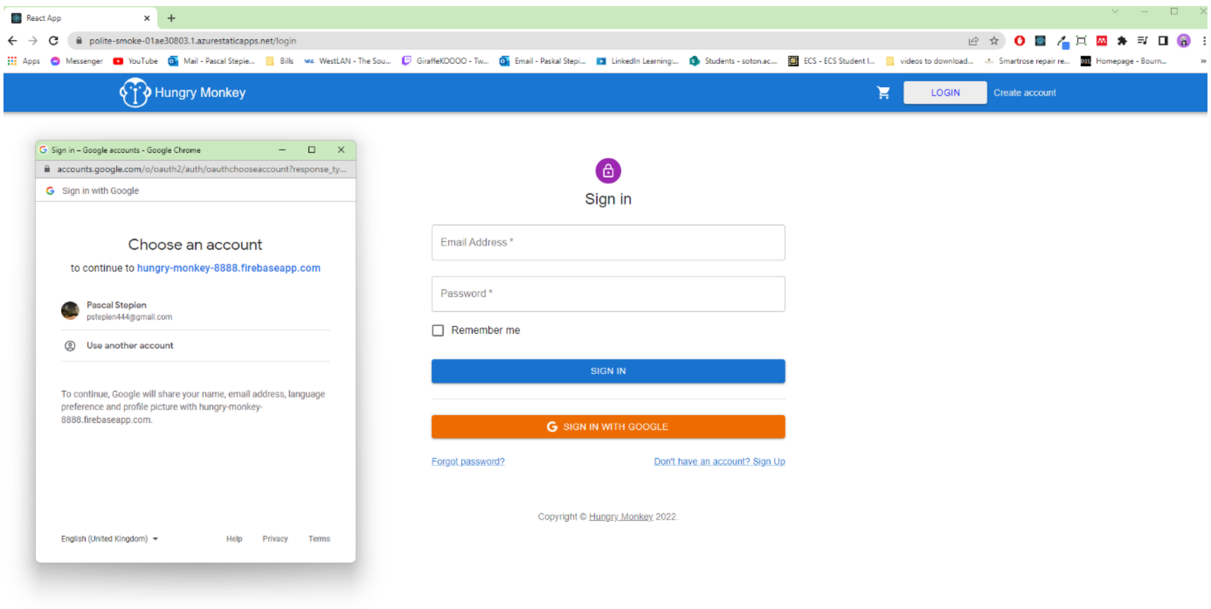
Figure 13 Screenshot of the Firebase user interface for addition of sign-in methods available for a web application

After this sign-in method was added to the firebase, it was then implemented in the code base. We implemented this in the Login.js file, which was the login page, and that's where the function for handling different types of login were located.


File Edit Selection View Go Run Terminal Help
login.js - Hungry-Mo
EXPLORER
Navbar.js 4
login.js 5
src > pages > login.js > SignIn > handleGoogleLogin

```
        setEmptyItem(true)
        console.log("Email not verified")
      }
    }).catch(()=>{
      alert("Email doesn't exist")
      console.log("User doesn't exist")
    })
  }


  const handleGoogleLogin = () => {
    console.log('Google Here')
    signWithGoogle().then((it)=>{
      const userFromGoogle = it.user
      axios.post("https://hungry-monkey-api.azurewebsites.net/api/user/getUserByUID",{
        uid: userFromGoogle.uid
      }).then((it)=>{
          console.log("Google Sign In 200")
        sessionStorage.setItem('uid',userFromGoogle.uid)
        navigate('/user_page')
      }).catch(()=>{
        console.log("Google Sign In 400!!!")
        const nameList = splitName(userFromGoogle.displayName)
        console.log(`Name is ${nameList}`)
        axios.post("https://hungry-monkey-api.azurewebsites.net/api/user/createUser", {
          uid: userFromGoogle.uid,
          email: userFromGoogle.email,
          first_name: nameList[0],
          last_name: nameList[1],
          role: "normal",
          status: "confirmed"
        }).then((it)=>{
          console.log("Google Sign In complete")
          sessionStorage.setItem('uid',userFromGoogle.uid)
          navigate('/user_page')
        })
      })
    })
  }

  // if (!auth.currentUser) {
  return (
    <ThemeProvider theme={theme}>
      <Navbar />
      <Container component="main" maxWidth="sm">
        <CssBaseline />
```

Figure 14 Screenshot of Visual Studio Code IDE, demonstrating the Login.js file, which is the login page, and contains all the functions for handling the various login in methods


React App
polite-smoke-01ae30803.1.azurestaticapps.net/login
Hungry Monkey
LOGIN
Create account
Sign in – Google accounts - Google Chrome
accounts.google.com/o/oauth2/auth/oauthchooseaccount?response_ty...
Sign in with Google
Choose an account
to continue to hungry-monkey-8888.firebaseapp.com
Pascal Stepien
pstepien444@gmail.com
Use another account
English (United Kingdom)
Help
Privacy
Terms
Sign in
Email Address *
Password *
Remember me
SIGN IN
SIGN IN WITH GOOGLE
Forgot password?
Don't have an account? Sign Up
Copyright © Hungry Monkey 2022.

To continue, Google will share your name, email address, language preference and profile picture with hungry-monkey-8888.firebaseapp.com.

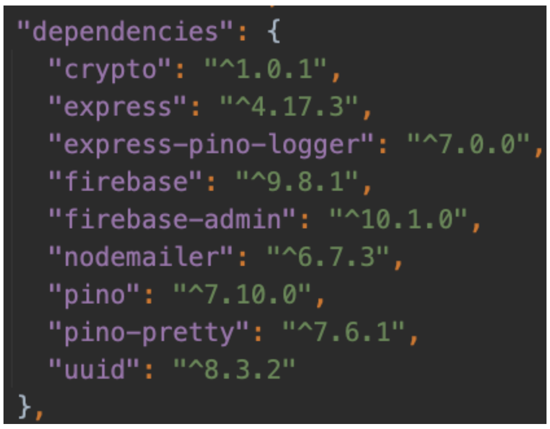
Figure 15 Screenshot of the web application login page, showing the possibility of logging in using social media accounts such as Google

The final view of the login screen can be seen in the above figure, which shows the implemented login screen, with the various login in methods, and a user attempting to sign in with a social media account such as Google.

3.3 Backend Server

3.3.1 Disclaimer

The backend is based on Node.js and Express framework, and all the code is written in JavaScript by Yijiang Wang. No code copy from GitHub or any other place. The figure below shows the third-part libraries implemented in this project.



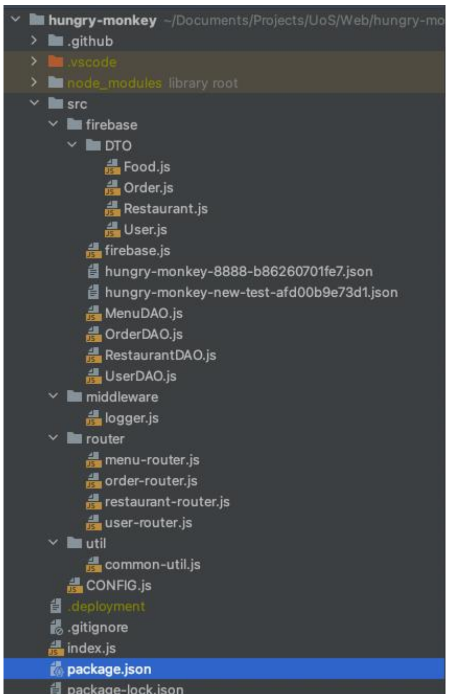
```
"dependencies": {
  "crypto": "^1.0.1",
  "express": "^4.17.3",
  "express-pino-logger": "^7.0.0",
  "firebase": "^9.8.1",
  "firebase-admin": "^10.1.0",
  "nodemailer": "^6.7.3",
  "pino": "^7.10.0",
  "pino-pretty": "^7.6.1",
  "uuid": "^8.3.2"
},
```

Figure 16 A screenshot of the code snippet listing the Third-party libraries implemented in the backend code

3.3.2 Overview

The figure above shows the project directory and the structure of the whole backend project. The firebase package contains the DTO (Data Transfer Object), Firebase configuration, and DAO (Data Access Object). The middleware package contains the logger middleware which is used in Express middleware to log all kinds of information. The router package includes the user, menu, order, and restaurant API routers. The util package has all the util functions used for the backend. The file “CONFIG.js” is all the settings of the backend server (port, notification parameters, etc). At last, the “index.js” is the entry point of the whole Node.js project.

This backend server is highly available and can handle multiple requests at the same time. It will also handle all kinds of errors itself (leaves a log file when the error happens) and will not crash.



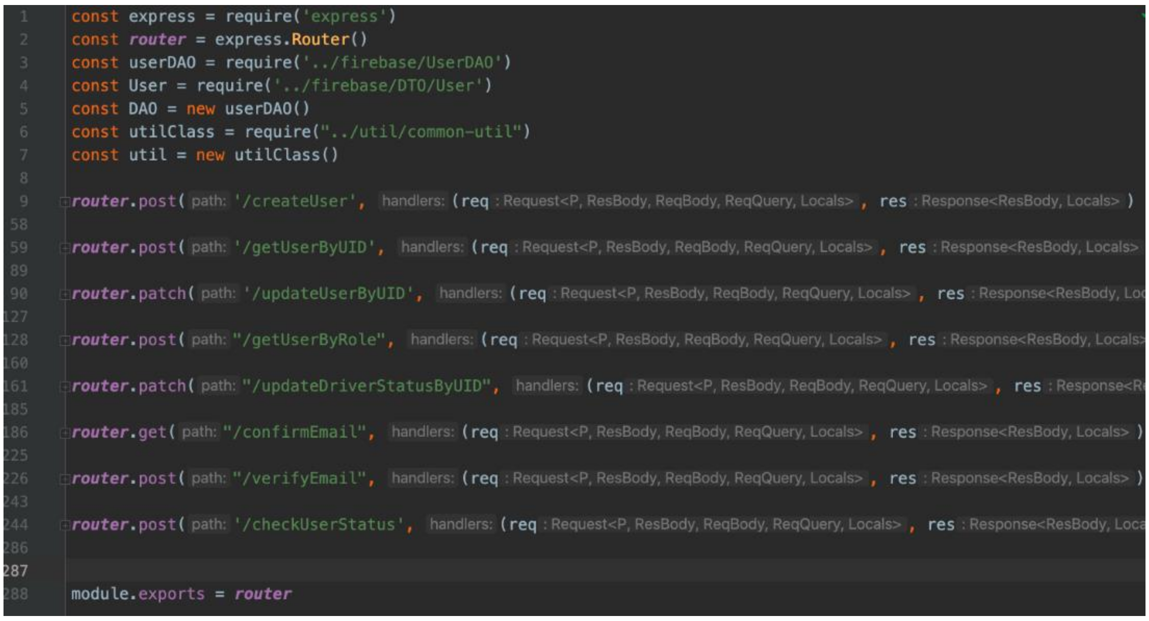
```
hungry-monkey ~/Documents/Projects/UoS/Web/hungry-mo
  .github
  .vscode
  node_modules library root
  src
    firebase
      DTO
        Food.js
        Order.js
        Restaurant.js
        User.js
      firebase.js
      hungry-monkey-8888-b86260701fe7.json
      hungry-monkey-new-test-afd00b9e73d1.json
      MenuDAO.js
      OrderDAO.js
      RestaurantDAO.js
      UserDAO.js
    middleware
      logger.js
    router
      menu-router.js
      order-router.js
      restaurant-router.js
      user-router.js
    util
      common-util.js
    CONFIG.js
  .deployment
  .gitignore
  index.js
  package.json
  package-lock.json
```

Figure 17 A screenshot of WebStorm showing the project directory and the structure of backend

3.3.3 User API

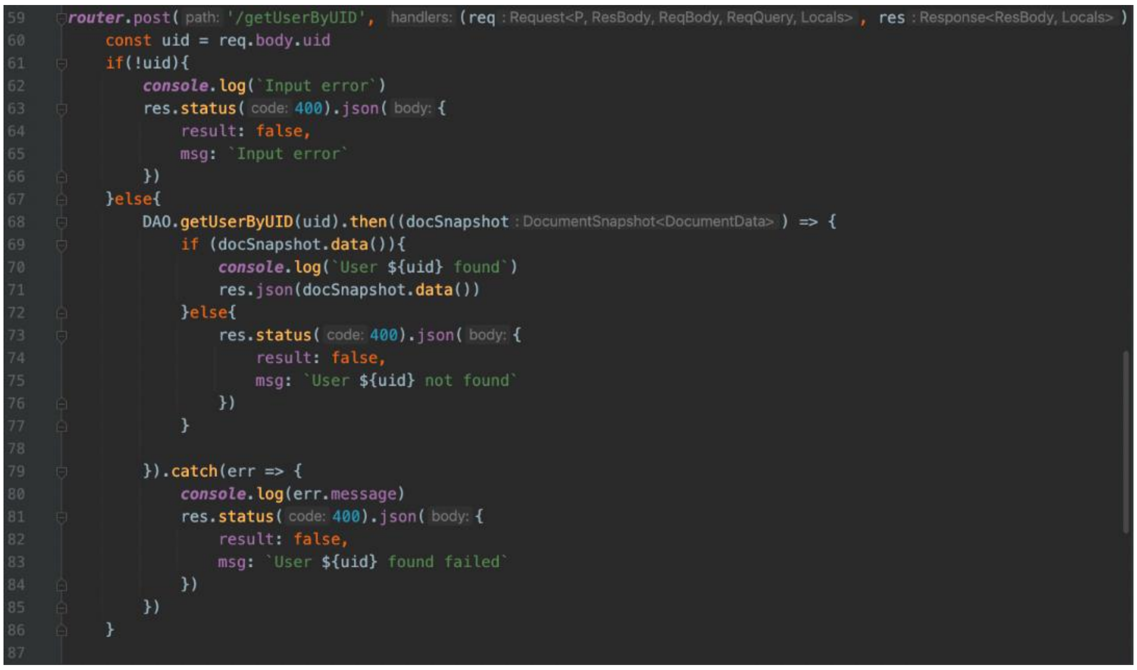
For user API, 8 APIs were implemented. These APIs can create a new user in Firebase Firestore, get user information by user UID or user role, update user information by UID, and update driver deliver status by UID and 2-factor authentication.

For example, as we can see, the figure below is a code snippet of API “getUserByUID”. This API will first check whether the post body is valid or not, this will make the backend server more robust. Then it will call the user DAO function and get the information from the Firestore asynchronously (this will not block the Node.js I/O), which gives the backend server the ability to handle multiple requests at the same time. When it encounters some error, it will send an error message to respond and log. All the CRUD API is the same process as this one.



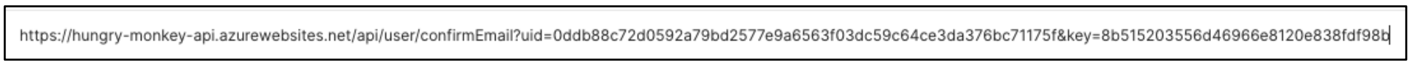
```
const express = require('express')
const router = express.Router()
const userDAO = require('../firebase/UserDAO')
const User = require('../firebase/DTO/User')
const DAO = new userDAO()
const utilClass = require("../util/common-util")
const util = new utilClass()

router.post( path: '/createUser',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals> )

router.post( path: '/getUserByUID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals>

router.patch( path: '/updateUserByUID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Lo

router.post( path: "/getUserByRole",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals>

router.patch( path: "/updateDriverStatusByUID",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<R

router.get( path: "/confirmEmail",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals> )

router.post( path: "/verifyEmail",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals> )

router.post( path: '/checkUserStatus',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Loca


module.exports = router
```

Figure 18 A screenshot of the code snippet responsible for User API functions



```
router.post( path: '/getUserByUID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> ,  res : Response<ResBody, Locals> )
    const uid = req.body.uid
    if(!uid){
        console.log(`Input error`)
        res.status( code: 400).json( body: {
            result: false,
            msg: `Input error`
        })
    }else{
        DAO.getUserByUID(uid).then((docSnapshot : DocumentSnapshot<DocumentData> ) => {
            if (docSnapshot.data()){
                console.log(`User ${uid} found`)
                res.json(docSnapshot.data())
            }else{
                res.status( code: 400).json( body: {
                    result: false,
                    msg: `User ${uid} not found`
                })
            }

        }).catch(err => {
            console.log(err.message)
            res.status( code: 400).json( body: {
                result: false,
                msg: `User ${uid} found failed`
            })
        })
    }
```

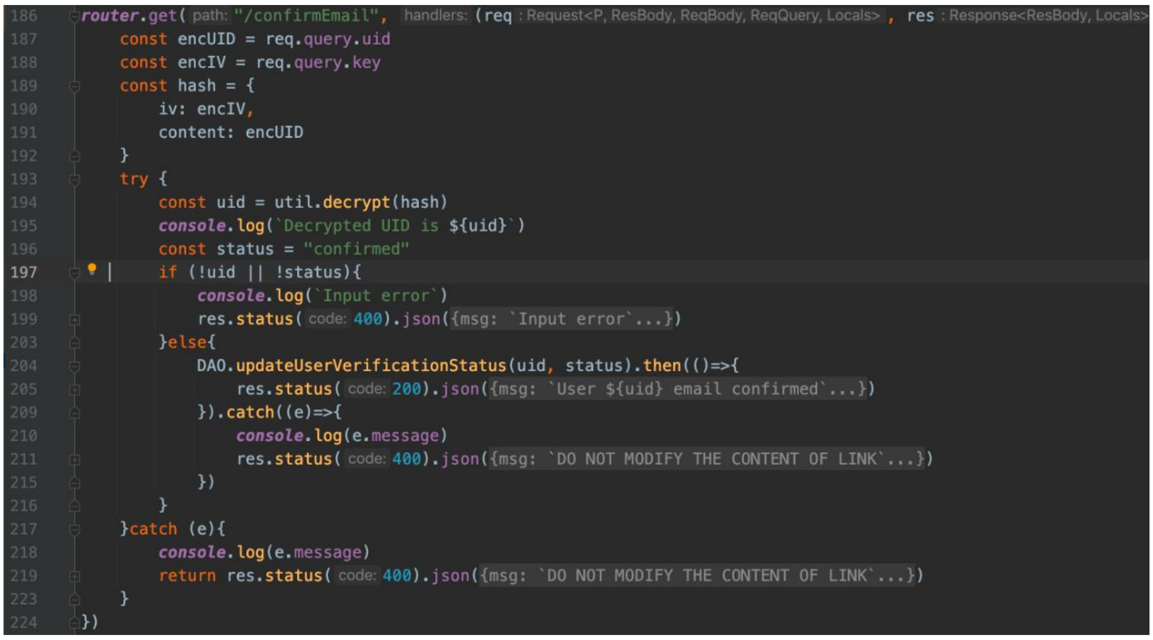
Figure 19 A screenshot of the code snippet showing the getUserByUID API function



https://hungry-monkey-api.azurewebsites.net/api/user/confirmEmail?uid=0ddb88c72d0592a79bd2577e9a6563f03dc59c64ce3da376bc71175f&key=8b515203556d46966e8120e838fdf98b

Figure 20 A screenshot of an example of the encrypted link sent in the confirmation email of a newly registered user

For the 2-factor authentication, every time a new user was created, the frontend will request the API “verifyEmail” and it will send the user a verification email with a special link which requests the API “confirmEmail” as the figure shows above. If the user clicks the link, their status will be changed from “unconfirmed” to “confirmed” and they will be able to log into the website. Because every time they try to log in, the frontend will call API “checkUserStatus” to check the email of the user is verified or not. To prevent a user from modifying the confirmation link sent to their email, the parameter of the link was encrypted by AES with random IV.



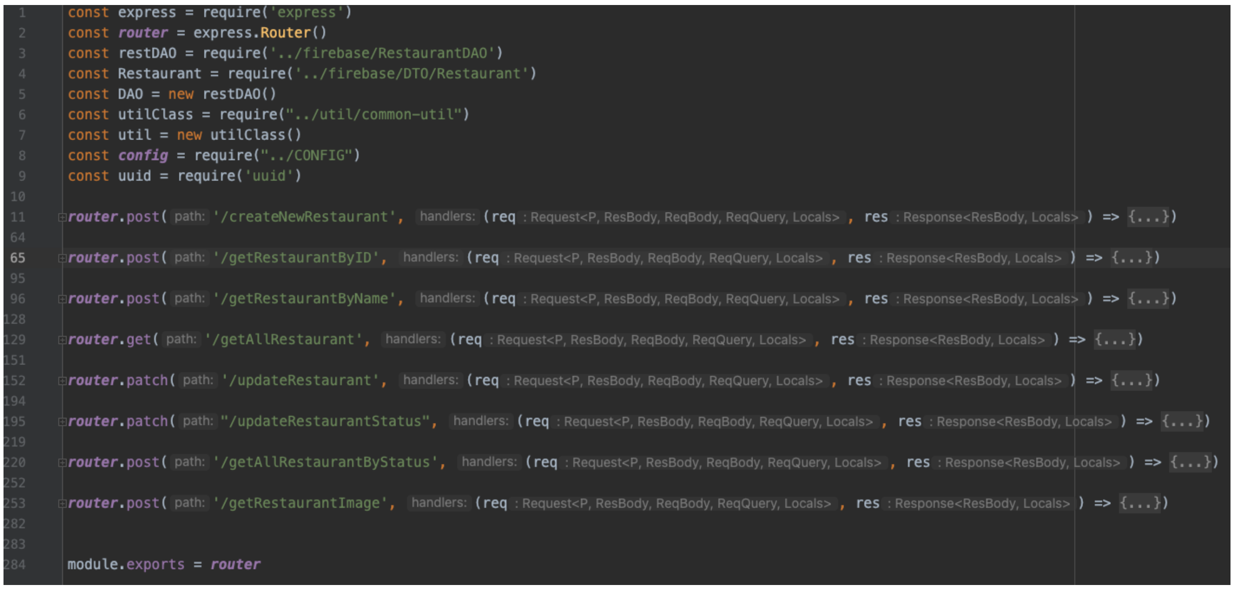
```
router.get( path: "/confirmEmail",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals>
    const encUID = req.query.uid
    const encIV = req.query.key
    const hash = {
        iv: encIV,
        content: encUID
    }
    try {
        const uid = util.decrypt(hash)
        console.log(`Decrypted UID is ${uid}`)
        const status = "confirmed"
        if (!uid || !status){
            console.log(`Input error`)
            res.status( code: 400).json({msg: `Input error`...})
        }else{
            DAO.updateUserVerificationStatus(uid, status).then(()=>{
                res.status( code: 200).json({msg: `User ${uid} email confirmed`...})
            }).catch((e)=>{
                console.log(e.message)
                res.status( code: 400).json({msg: `DO NOT MODIFY THE CONTENT OF LINK`...})
            })
        }
    }catch (e){
        console.log(e.message)
        return res.status( code: 400).json({msg: `DO NOT MODIFY THE CONTENT OF LINK`...})
    }
})
```

Figure 21 A screenshot showing the 2-Factor Authentication confirmEmail API function

If the user modifies the link sent to them, the response of this link will show a message to give them a warning that does not modify the link. This keeps our application safe and robust.

3.3.4 Restaurant API

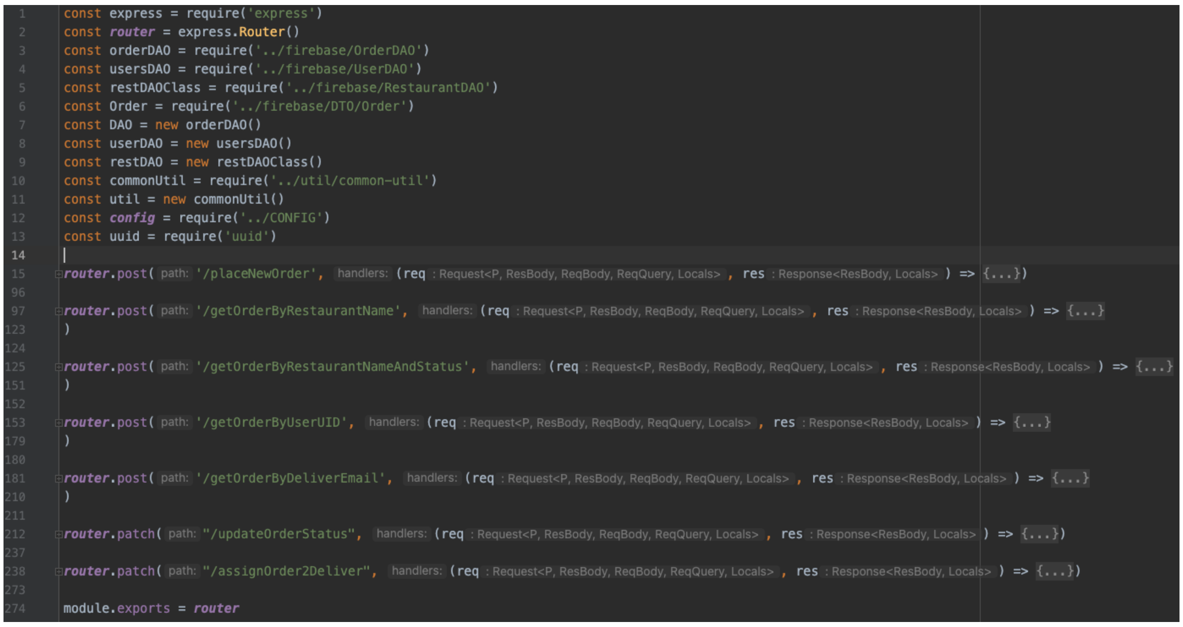
The restaurant router contains 8 APIs, which are API that CRUD restaurant information and get the access link image of the restaurant from Firebase Storage. The basic structure of each API is the same as the user router.



```
const express = require('express')
const router = express.Router()
const restDAO = require('../firebase/RestaurantDAO')
const Restaurant = require('../firebase/DTO/Restaurant')
const DAO = new restDAO()
const utilClass = require("../util/common-util")
const util = new utilClass()
const config = require("../CONFIG")
const uuid = require('uuid')

router.post( path: '/createNewRestaurant',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getRestaurantByID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getRestaurantByName',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.get( path: '/getAllRestaurant',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.patch( path: '/updateRestaurant',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.patch( path: "/updateRestaurantStatus",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getAllRestaurantByStatus',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getRestaurantImage',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})


module.exports = router
```

Figure 22 A screenshot of the code snippet for the Restaurant API functions

3.3.5 Order API

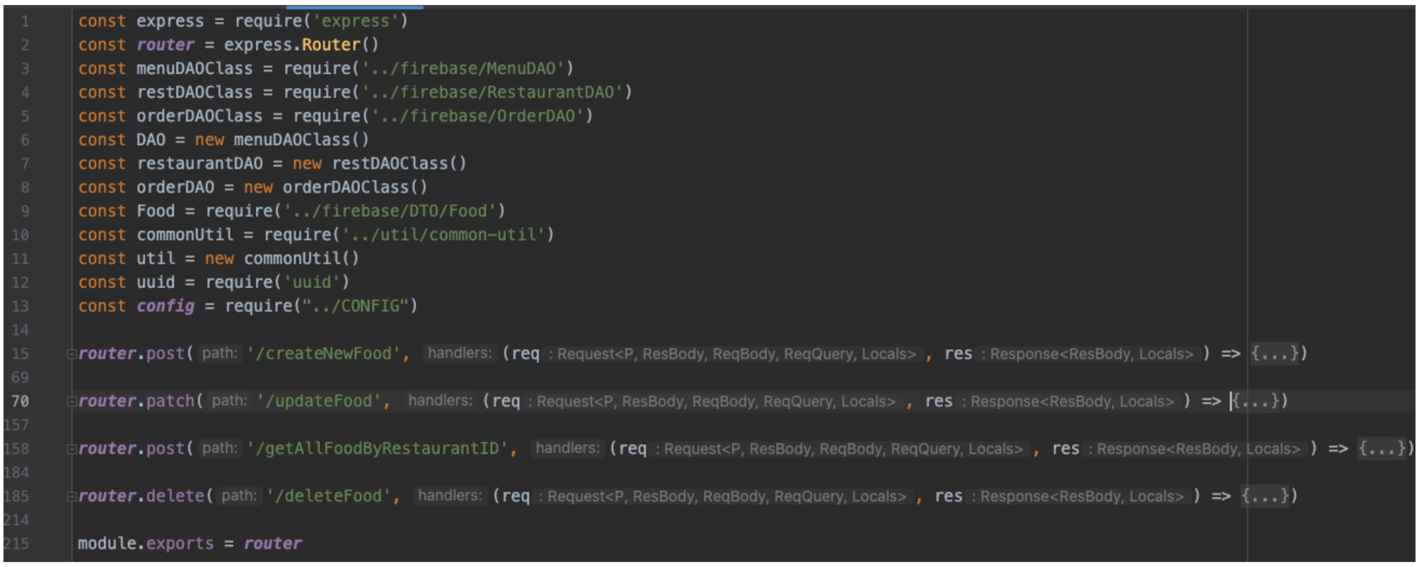


```
const express = require('express')
const router = express.Router()
const orderDAO = require('../firebase/OrderDAO')
const usersDAO = require('../firebase/UserDAO')
const restDAOClass = require('../firebase/RestaurantDAO')
const Order = require('../firebase/DTO/Order')
const DAO = new orderDAO()
const userDAO = new usersDAO()
const restDAO = new restDAOClass()
const commonUtil = require('../util/common-util')
const util = new commonUtil()
const config = require('../CONFIG')
const uuid = require('uuid')

router.post( path: '/placeNewOrder',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getOrderByRestaurantName',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...}
)

router.post( path: '/getOrderByRestaurantNameAndStatus',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...}
)

router.post( path: '/getOrderByUserUID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...}
)

router.post( path: '/getOrderByDeliverEmail',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...}
)

router.patch( path: "/updateOrderStatus",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.patch( path: "/assignOrder2Deliver",  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

module.exports = router
```

Figure 23 A screenshot of the code snippet for the Order API functions

This router for the order has 7 different API, including order CRUD in firebase and assigning order to the driver. The structure of the code of each API is almost the same as the user router.

3.3.6 Menu API



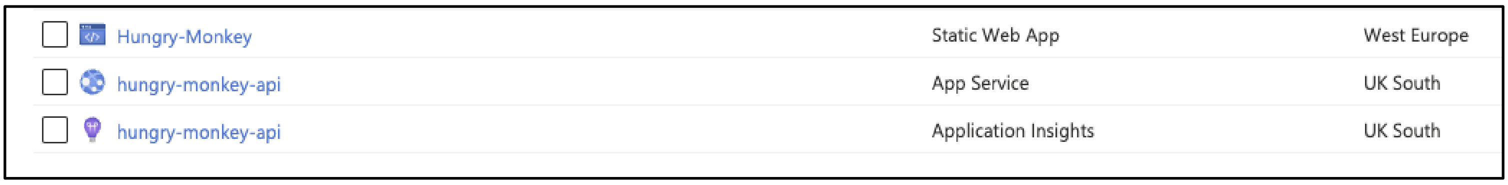
```
const express = require('express')
const router = express.Router()
const menuDAOClass = require('../firebase/MenuDAO')
const restDAOClass = require('../firebase/RestaurantDAO')
const orderDAOClass = require('../firebase/OrderDAO')
const DAO = new menuDAOClass()
const restaurantDAO = new restDAOClass()
const orderDAO = new orderDAOClass()
const Food = require('../firebase/DTO/Food')
const commonUtil = require('../util/common-util')
const util = new commonUtil()
const uuid = require('uuid')
const config = require("../CONFIG")

router.post( path: '/createNewFood',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.patch( path: '/updateFood',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.post( path: '/getAllFoodByRestaurantID',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

router.delete( path: '/deleteFood',  handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {...})

module.exports = router
```

Figure 24 A screenshot of the code snippet for the Menu API functions

The last router is the menu router, which only contains 4 APIs about the CRUD of food information. When the front end calls the “updateFood” API, it will automatically decide that the restaurant owner lower the price and make an offer notification to all customers that made an order in this restaurant before. The code structure is almost the same as the other router.

3.4 Deploy to Cloud

The backend API server was deployed to Azure using Azure App Service as a hosting server. App Service is a very convenient Linux or Windows server service for Node.js, Java, or Python, and its supports CI/CD DevOps. The frontend React application was deployed to Azure Static Web App service, which is a host for a static web application, supporting React, Angular, Vue etc. Static Web App also supports CI/CD. So, during the whole development of this application, we are collaborating through Github and using Github Action CI/CD YAML to automatically deploy the master branch to Azure App Service and Static Web App.



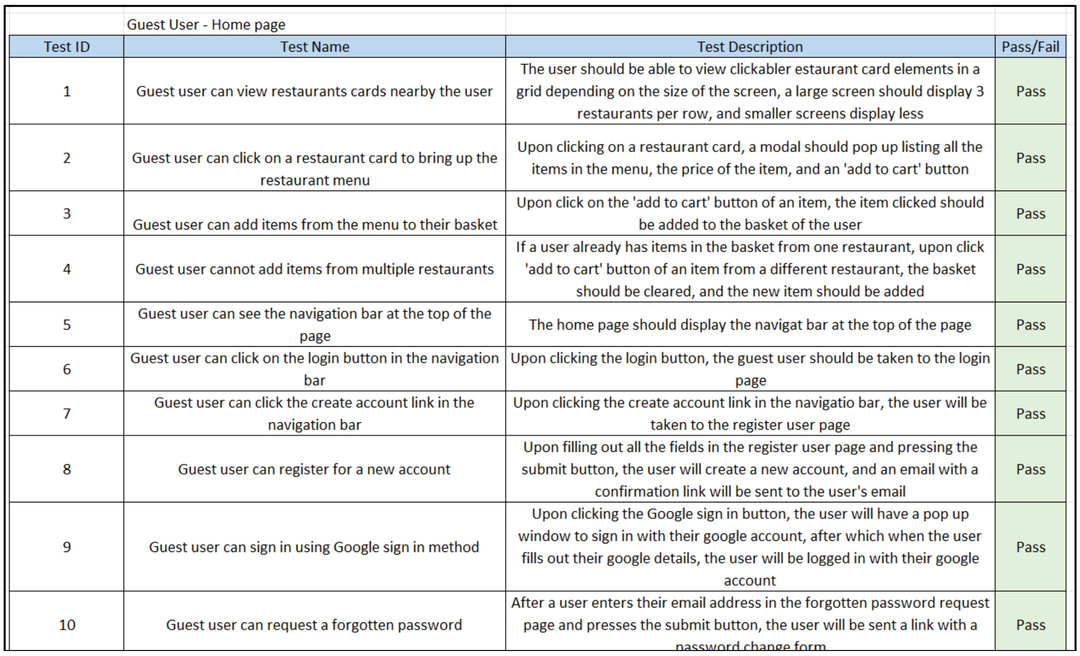
Hungry-Monkey	Static Web App	West Europe
hungry-monkey-api	App Service	UK South
hungry-monkey-api	Application Insights	UK South

Figure 25 A screenshot of the Azure Deployment Resource Group interface showing the deployed web application

4. Testing

4.1 Manual Testing

For the testing of the application, we have used a combination of testing techniques to achieve the most effective rate of finding and fixing bugs. For the front end of this web application, we chose to do manual testing. This is because we believe this was the most efficient method of testing of the graphical user interface. For the backend, the testing of the API calls and the data being returned was carried out with tests in the Postman application, which is explained in the further section. In addition, due to

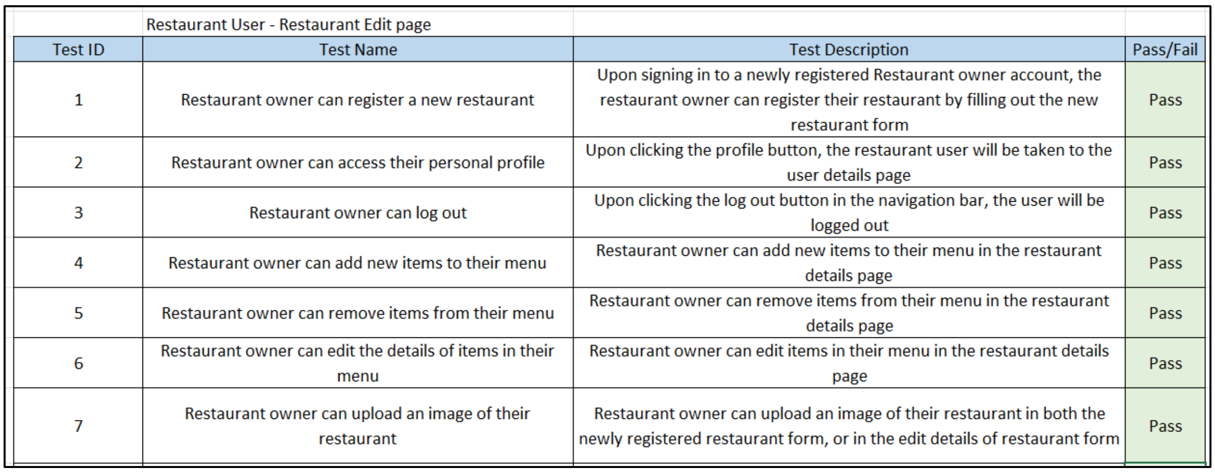


	Guest User - Home page		
Test ID	Test Name	Test Description	Pass/Fail
1	Guest user can view restaurants cards nearby the user	The user should be able to view clickabler estaurant card elements in a grid depending on the size of the screen, a large screen should display 3 restaurants per row, and smaller screens display less	Pass
2	Guest user can click on a restaurant card to bring up the restaurant menu	Upon clicking on a restaurant card, a modal should pop up listing all the items in the menu, the price of the item, and an 'add to cart' button	Pass
3	Guest user can add items from the menu to their basket	Upon click on the 'add to cart' button of an item, the item clicked should be added to the basket of the user	Pass
4	Guest user cannot add items from multiple restaurants	If a user already has items in the basket from one restaurant, upon click 'add to cart' button of an item from a different restaurant, the basket should be cleared, and the new item should be added	Pass
5	Guest user can see the navigation bar at the top of the page	The home page should display the navigat bar at the top of the page	Pass
6	Guest user can click on the login button in the navigation bar	Upon clicking the login button, the guest user should be taken to the login page	Pass
7	Guest user can click the create account link in the navigation bar	Upon clicking the create account link in the navigatio bar, the user will be taken to the register user page	Pass
8	Guest user can register for a new account	Upon filling out all the fields in the register user page and pressing the submit button, the user will create a new account, and an email with a confirmation link will be sent to the user's email	Pass
9	Guest user can sign in using Google sign in method	Upon clicking the Google sign in button, the user will have a pop up window to sign in with their google account, after which when the user fills out their google details, the user will be logged in with their google account	Pass
10	Guest user can request a forgotten password	After a user enters their email address in the forgotten password request page and presses the submit button, the user will be sent a link with a password change form	Pass

Figure 26 Screenshot of Microsoft Excel displaying the manual test cases for the Guest User category of the Home page, including test case description, and whether the test passed or failed

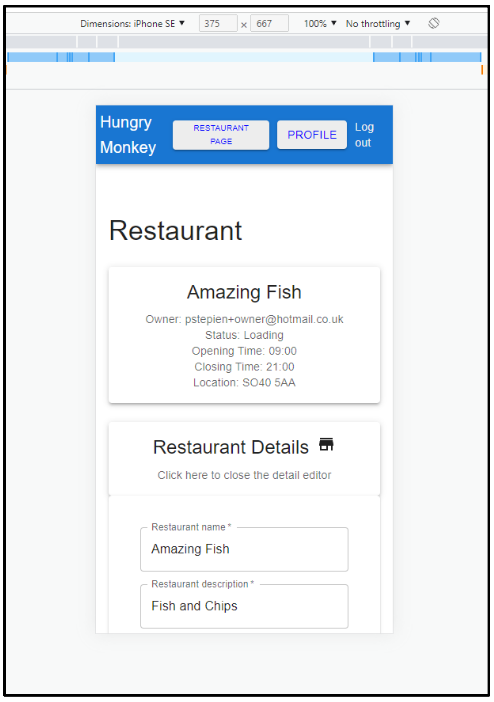
time constrains, we also regularly tested the elements as they were being created, to ensure the least number of possible bugs during the testing stage.

The manual tests that were carried out, were recorded using Microsoft Excel, as this is one of the most commonly used software applications for manual testing. We chose to test various aspects of this application, ranging from making sure the components display the correct data, making sure the navigation buttons take the user to the correct page, and overall business logic is implemented correctly, based on the previous activity diagrams discussed in the previous sections.



	Restaurant User - Restaurant Edit page		
Test ID	Test Name	Test Description	Pass/Fail
1	Restaurant owner can register a new restaurant	Upon signing in to a newly registered Restaurant owner account, the restaurant owner can register their restaurant by filling out the new restaurant form	Pass
2	Restaurant owner can access their personal profile	Upon clicking the profile button, the restaurant user will be taken to the user details page	Pass
3	Restaurant owner can log out	Upon clicking the log out button in the navigation bar, the user will be logged out	Pass
4	Restaurant owner can add new items to their menu	Restaurant owner can add new items to their menu in the restaurant details page	Pass
5	Restaurant owner can remove items from their menu	Restaurant owner can remove items from their menu in the restaurant details page	Pass
6	Restaurant owner can edit the details of items in their menu	Restaurant owner can edit items in their menu in the restaurant details page	Pass
7	Restaurant owner can upload an image of their restaurant	Restaurant owner can upload an image of their restaurant in both the newly registered restaurant form, or in the edit details of restaurant form	Pass

Figure 27 Screenshot of Microsoft Excel displaying the manual test cases for the Restaurant Owner User category of the Edit Restaurant page, including test case description, and whether the test passed or failed



Dimensions: iPhone SE ▾ 375 x 667 100% ▾ No throttling ▾

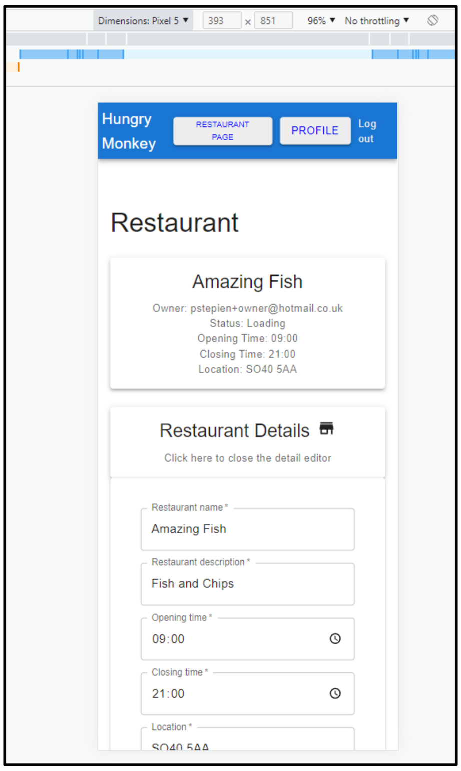
Restaurant

Amazing Fish
Owner: pstepien+owner@hotmail.co.uk
Status: Loading
Opening Time: 09:00
Closing Time: 21:00
Location: SO40 5AA

Restaurant Details
Click here to close the detail editor

Restaurant name *
Amazing Fish

Restaurant description *
Fish and Chips



Dimensions: Pixel 5 ▾ 393 x 851 96% ▾ No throttling ▾

Restaurant

Amazing Fish
Owner: pstepien+owner@hotmail.co.uk
Status: Loading
Opening Time: 09:00
Closing Time: 21:00
Location: SO40 5AA

Restaurant Details
Click here to close the detail editor

Restaurant name *
Amazing Fish

Restaurant description *
Fish and Chips

Opening time *
09:00

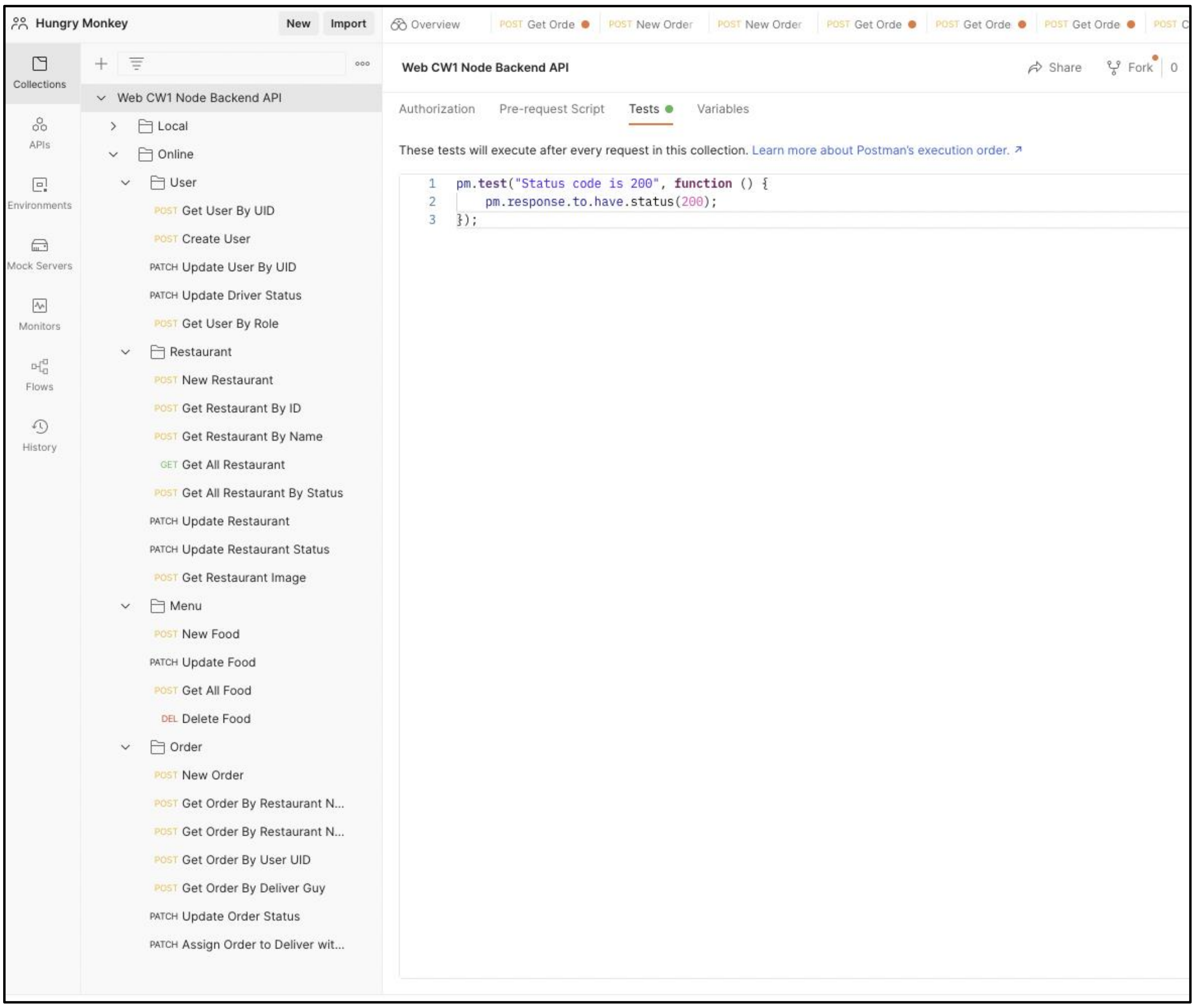
Closing time *
21:00

Location *
SO40 5AA

Figure 28 Screenshot of Google Chrome Inspection tools set to different view sizes of different phones demonstrating the responsiveness of the User Interface during the testing stage

4.2 Backend API Testing

All potential errors in the backend API were caught and logged. And the backend server API was tested through Postman.



- Web CW1 Node Backend API
  - Local
  - Online
    - User
      - POST Get User By UID
      - POST Create User
      - PATCH Update User By UID
      - PATCH Update Driver Status
      - POST Get User By Role
    - Restaurant
      - POST New Restaurant
      - POST Get Restaurant By ID
      - POST Get Restaurant By Name
      - GET Get All Restaurant
      - POST Get All Restaurant By Status
      - PATCH Update Restaurant
      - PATCH Update Restaurant Status
      - POST Get Restaurant Image
    - Menu
      - POST New Food
      - PATCH Update Food
      - POST Get All Food
      - DEL Delete Food
    - Order
      - POST New Order
      - POST Get Order By Restaurant N...
      - POST Get Order By Restaurant N...
      - POST Get Order By User UID
      - POST Get Order By Deliver Guy
      - PATCH Update Order Status
      - PATCH Assign Order to Deliver wit...

These tests will execute after every request in this collection. Learn more about Postman's execution order. ↗

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

Figure 29 A screenshot of the Postman user interface displaying the Backend API Tests

5. Bibliography

[1] Jakob Nielsen, “10 Usability Heuristics for User Interface Design,” *Nielsen Norman Group*, Apr. 24, 1994. https://www.nngroup.com/articles/ten-usability-heuristics/ (accessed May 23, 2021).

[2] S. Krug, *Don’t make me think, revisited : a common sense approach to Web usability*, Third. Berkeley, California: New Riders, 2014. Accessed: May 18, 2022. [Online]. Available: https://www-lib.soton.ac.uk/uhtbin/cgisirsi/?ps=DlNlACGc2s/HARTLEY/X/9