CSC4001 Report:
Campus Food Ordering System

Group Members:

Huang Tianjian 117010099

Zeng Lewei 117010366

Wu Runzhong 117010289

Zhang Pinfang 117010383

Introduction

Campus canteens are a major food source for students and employees in the Chinese University of Hong Kong, Shenzhen (CUHKSZ). The food from campus canteens is guaranteed to be safe, and it is relatively cheap comparing with the food from outside restaurants. However, there is growing inconvenience about dining in canteens. First, the number of students in CUHKSZ is rising each year, but the number of canteens and the number of seats in canteens are limited. Second, students' time for dining is relatively concentrated, which results in surges of people in canteens during dining time. Third, our canteens are not only open to students and employees, but also available to tourists and bunches of visitors outside. The above factors lead to insufficient seats and long queues during lunch and dinner time.

One way to solve the problem is to create an on-campus "Mei Tuan". In this project, our team is going to design and implement a campus food ordering system. Students and employees can order food without going to the canteen in person, and the system also enables canteens to provide food delivery service within the campus.

Statement of Requirements

Our campus food ordering system is a web-based PC-end application. There are three kinds of users (roles) who will be involved in the working flow of application: customer, shop owner, and administrator. Each role has its own requirements about the system.

a) *Customer*

Customer is the basic role in this system. When a non-admin user registers an account and logs in, the user is a customer by default. A customer may need the following functions:

1. Registration, login/logout and reset forgotten passwords;
2. User information management, such as changing username and changing passwords;
3. Searching and surfing shops and dishes (products), and getting recommended popular dishes;
4. Order management, such as creating an order, cancelling an order, confirming a receipt, etc.

b) *Shop owner*

Shop owner is a role based on customer. A user needs to complete an additional application process to become a shop owner, and such process is inspected by the administrator. A shop owner has the following functions which are not available for ordinary customers:

1. Sending applications for shop opening and shop cancelling;
2. Shop information management, such as changing name, address and pictures;
3. Shop status management, which includes opening a shop, closing a shop, etc.;
4. Manage dishes (products) in a shop, such as adding/deleting dishes, changing prices, etc.;
5. Order management, which is the complementary side of customer's functions. For example, shop owner can accept and process an order of a customer.

c) *Administrator*

Administrator is a supplementary role in this system. Currently, an administrator can view public information of all users, process user applications and block/unblock shops.

Structure of the system

a) *Use case diagrams*

Base on the roles described above, we draw the use case diagrams as follows:

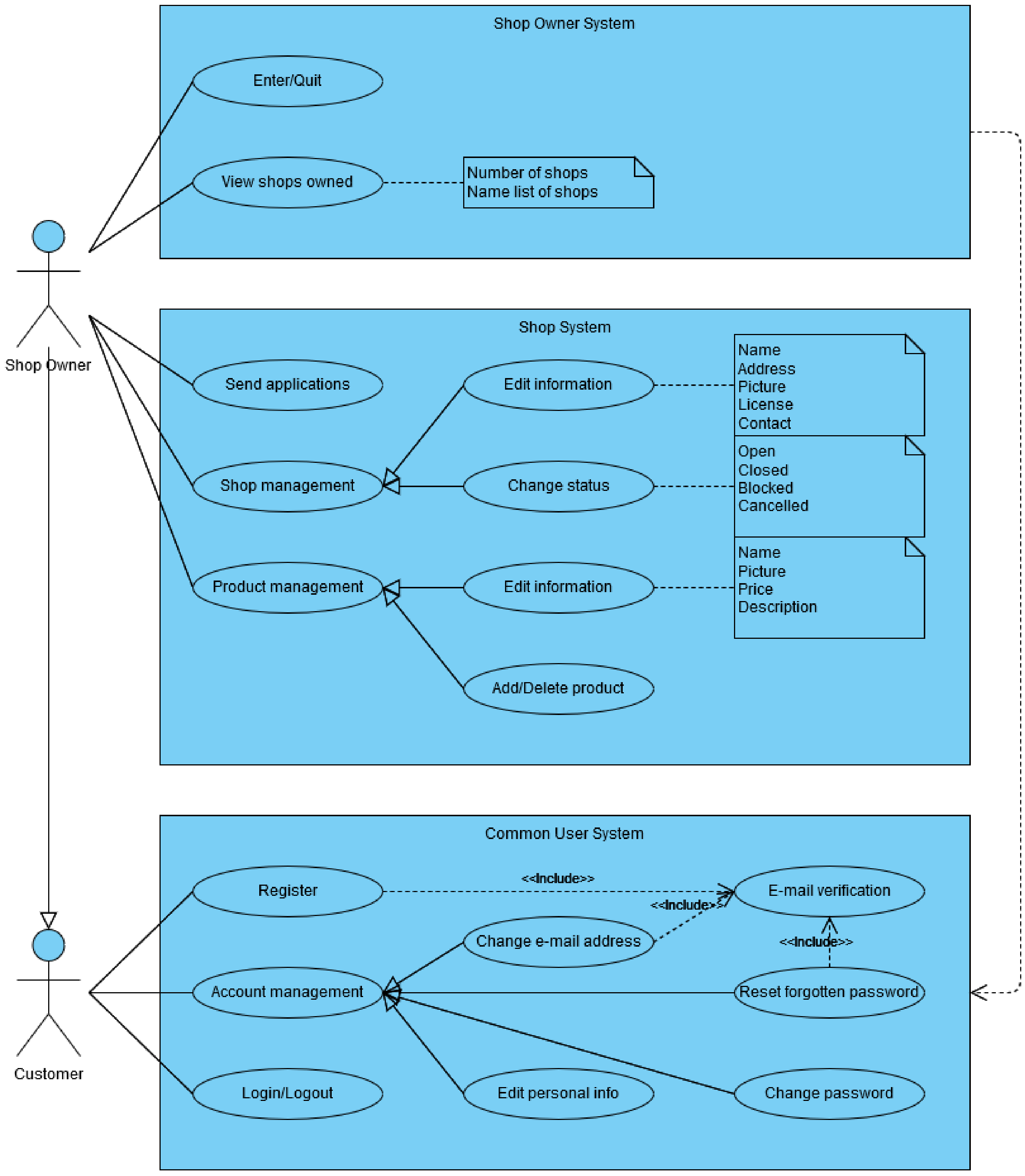


Figure 1. User system of customer, user system of shop owner and shop system

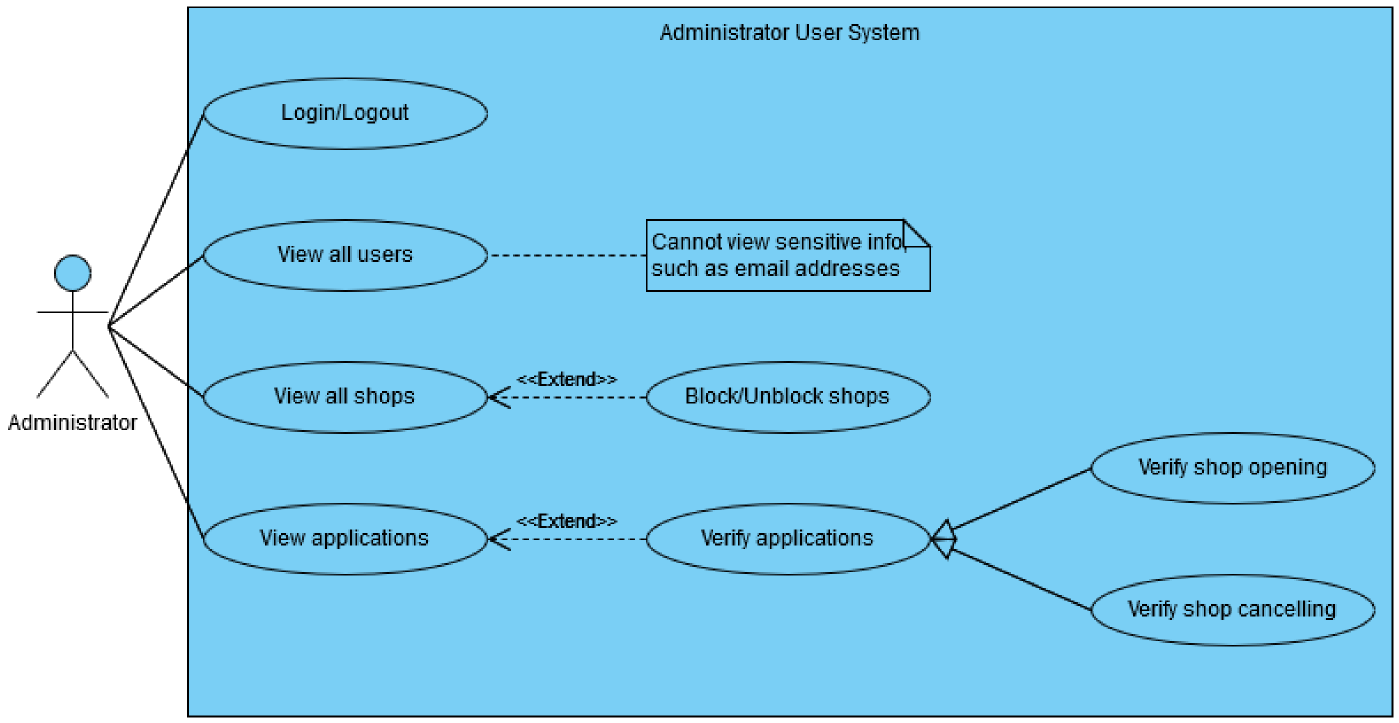


Figure 2. Administrator system

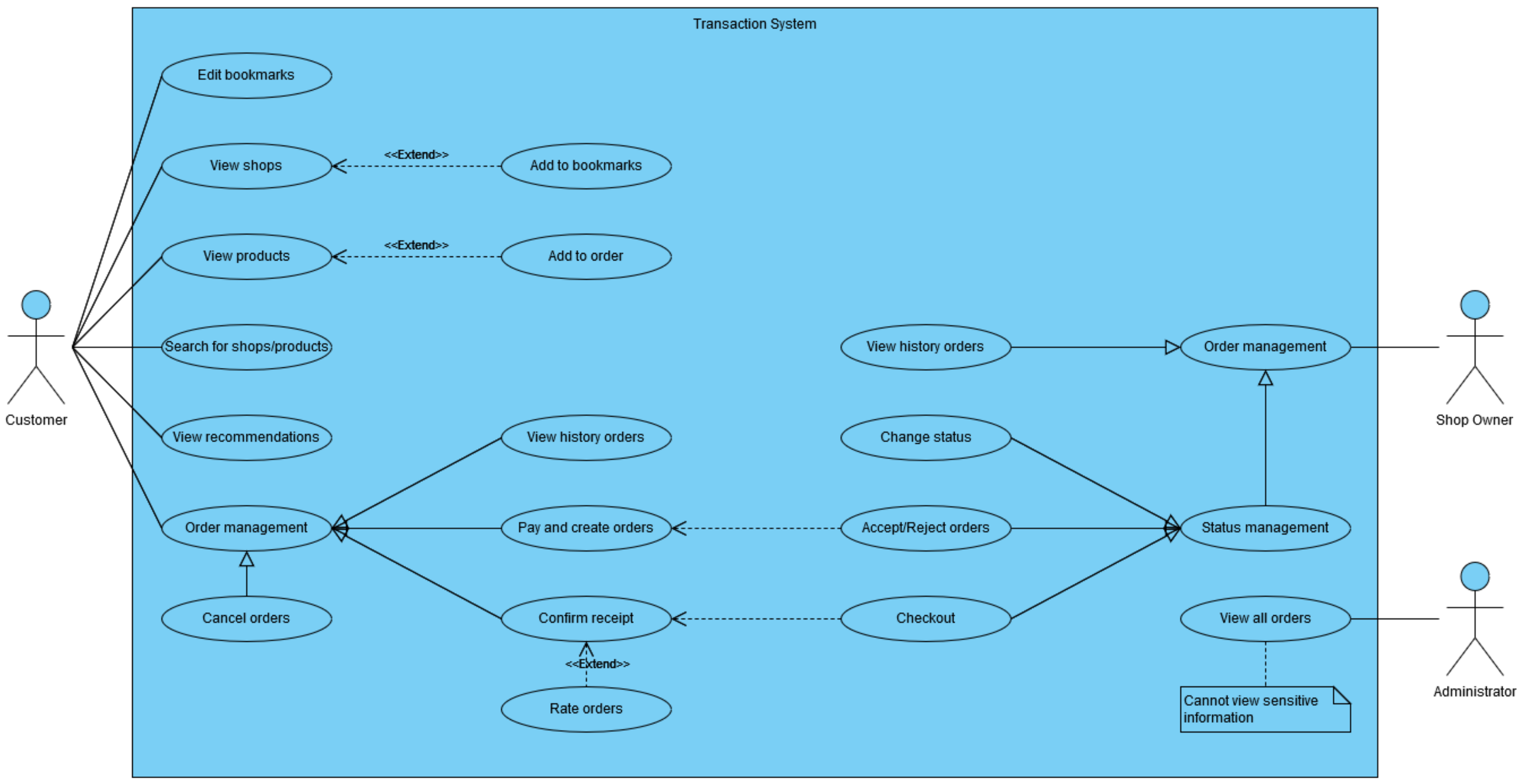


Figure 3. Transaction system

We divide the whole application into the following systems:

1. User systems, including common user system and shop owner system. In our system, to be a shop owner, a user must first register as a customer and then send an application for opening a shop. As a result, both roles need a common user system, and shop owner needs an additional shop owner system.

2. Shop system, which is affiliated to shop owners. The shop system includes all functions that only involve one role, the shop owner. Those functions include shop information management and dishes (products) management in a shop.
3. Administration system, includes all functions of an administrator.
4. Transaction system, which is the core and the most complicated system in the whole application. This system includes all three roles and all functions related to food searching, ordering, delivering and financial transactions.

In general, shop owner is the role who provides information about dishes and shops, and customer is the role who gains that information to fulfill his or her needs. Transaction system provides a link which allows that information to flow from shop owners to customers. In addition, transaction system provides interfaces of creating orders, tracking and processing orders, completing orders, and enables deals of money to realize transactions between customers and shops.

b) Database design

The database `ordering_system` contains all 8 tables which are necessary for the project. Their names are `"administrators", "users", "shops", "products", "orders", "products_in_orders", "applications", "bookmarks"` respectively. Detailed information of those tables is listed below:

`administrators(administrator_name, administrator_password)`
The table stores the information of all administrators.
Primary Key:
(`administrator_name`)

`users(user_id, email, user_password, user_name, user_contact, available_balance, frozen_balance, user_status)`
The table stores the basic information and the balance of all users.
Primary Key:
(`user_id`)

`shops(shop_id, user_id, shop_name, shop_info, shop_delivery_fee, shop_rate_total, shop_rate_number, shop_balance, shop_contact, shop_location, shop_location_detail, shop_opening_time, shop_closing_time, shop_license_number, shop_status)`
The table stores the basic information, owner's ID, the balance and the rate of all shops.
Primary Key:
(`shop_id`)
Foreign Key:
(`user_id`) references the table "`users`", indicating the owner of a shop.

`products(product_id, shop_id, product_name, product_info, product_price, month_sell)`
The table stores the basic information and the month sell of all products.
Primary Key:
(`product_id`)
Foreign Key:
(`shop_id`) references the table "`shops`", indicating the shop that a product belongs to.

`orders(order_id, shop_id, user_id, user_contact, user_location,` `delivery_fee, create_time, order_status)`

The table stores some information of orders, including the seller's ID and the buyer's ID, but it does not store the products in any order.

Primary Key:

(`order_id`)

Foreign Key:

(`shop_id`) references the table "`shops`", indicating the shop in an order.

(`user_id`) references the table "`users`", indicating the customer in an order.

`purchased_product(product_id, order_id, product_name, product_price, quantity)`

The table stores the basic information and the quantity of the products in orders, which may be different from the products in shops if the shops edit their products after selling them.

Primary Key:

(`product_id`)

Foreign Key:

(`order_id`) references the table "`orders`", indicating the order that a product belongs to.

`applications(application_id, user_id, application_type, shop_name, shop_info, shop_contact, shop_location, shop_location_detail, shop_opening_time, shop_closing_time, shop_license_number, application_status)`

The table stores the information of all applications and shops which may open in the future.

Primary Key:

(`application_id`)

Foreign Key:

(`user_id`) references the table "`users`", indicating the user who submits an application.

`bookmarks(user_id, shop_id)`

The table stores the bookmarks of all users.

Primary Key:

(`user_id, shop_id`)

Foreign Key:

(`user_id`) references the table "`users`".

(`shop_id`) references the table "`shops`".

c) *Frontend design*

The frontend design is based on the three roles. There are three main pages for customer system, shop system, and administrator system respectively. On these pages, the roles can perform their basic functions by interacting with web pages. For example, an administrator can check the order information he or she needs. The web page sends appropriate request to fetch the corresponding information from the database, and presents the information in a user-friendly way. The transaction system, a complex system involving all roles, is primarily represented by the interaction between customers and shops, and the information flow from customers to the shop owners.

The following are design drawings:

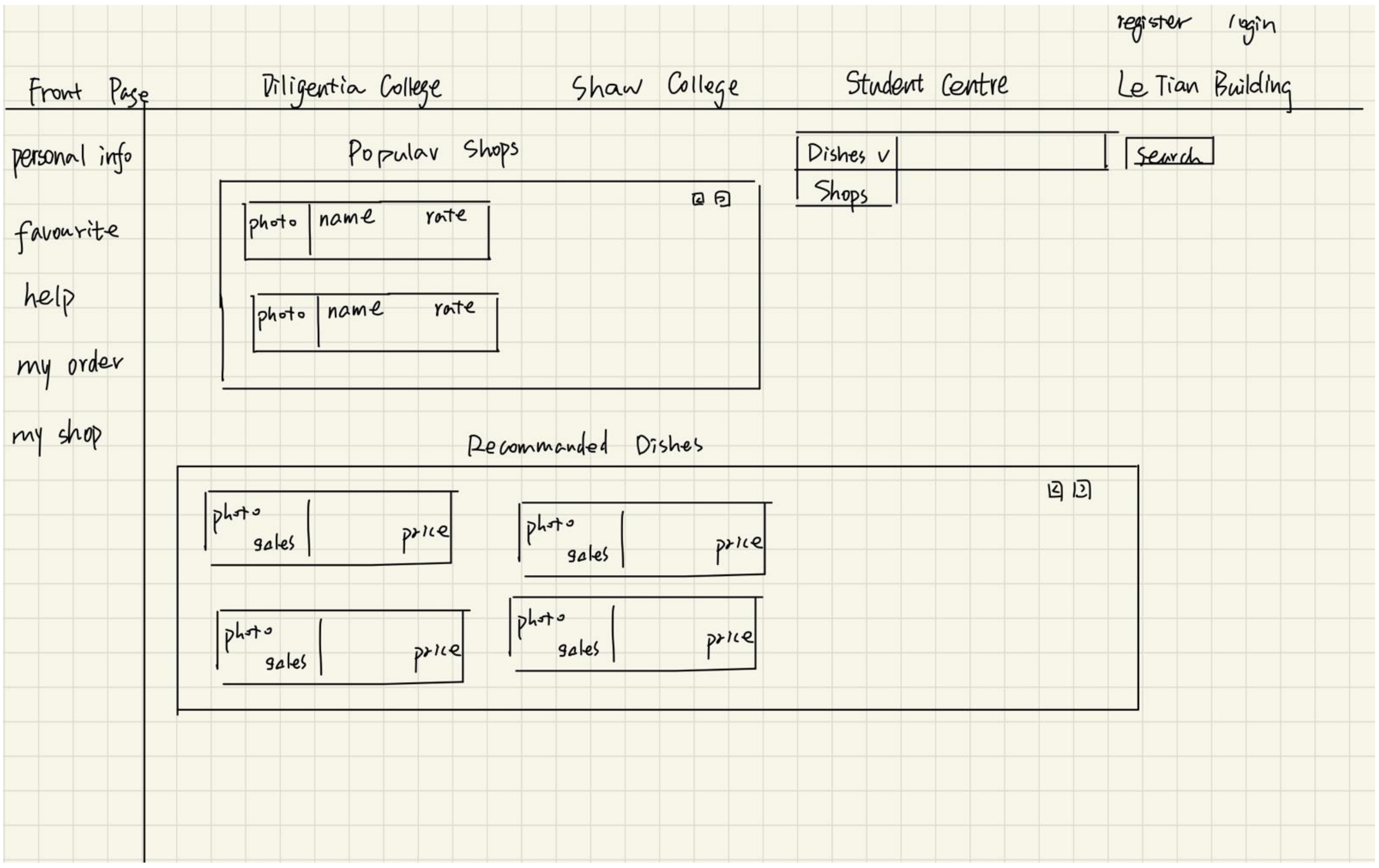


Figure 4. Design of index page

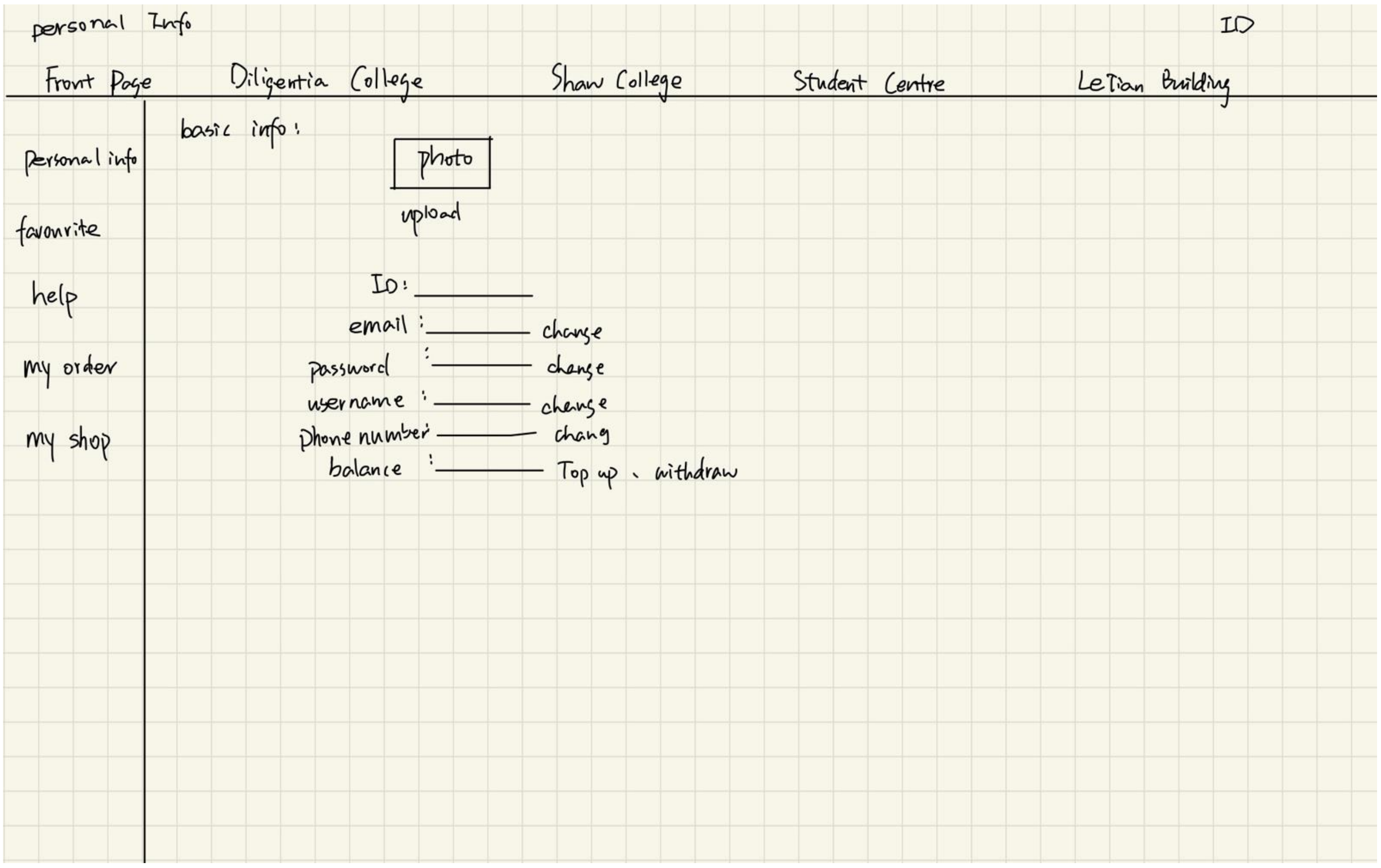


Figure 5. Design of customer system

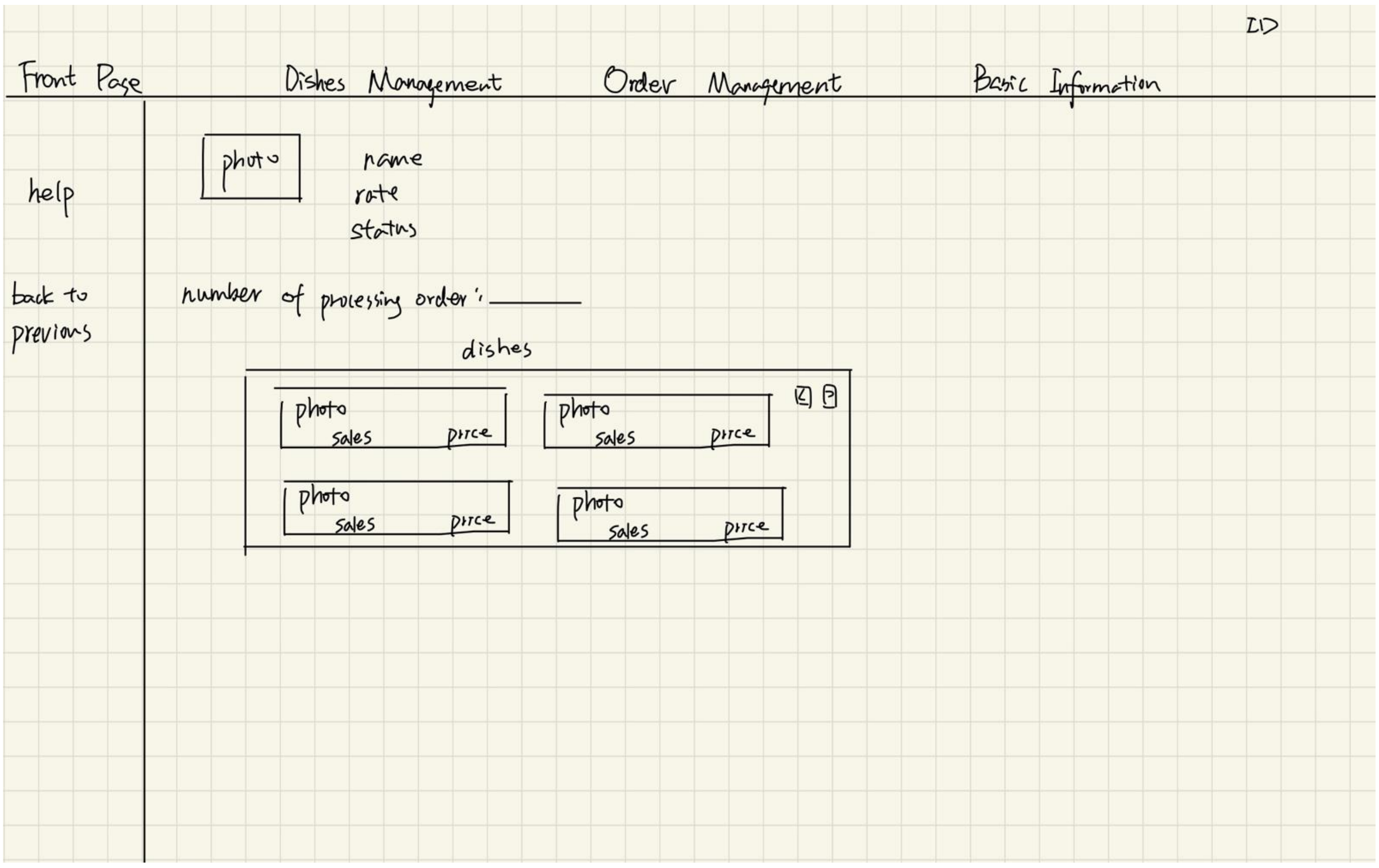


Figure 6. Design of shop system

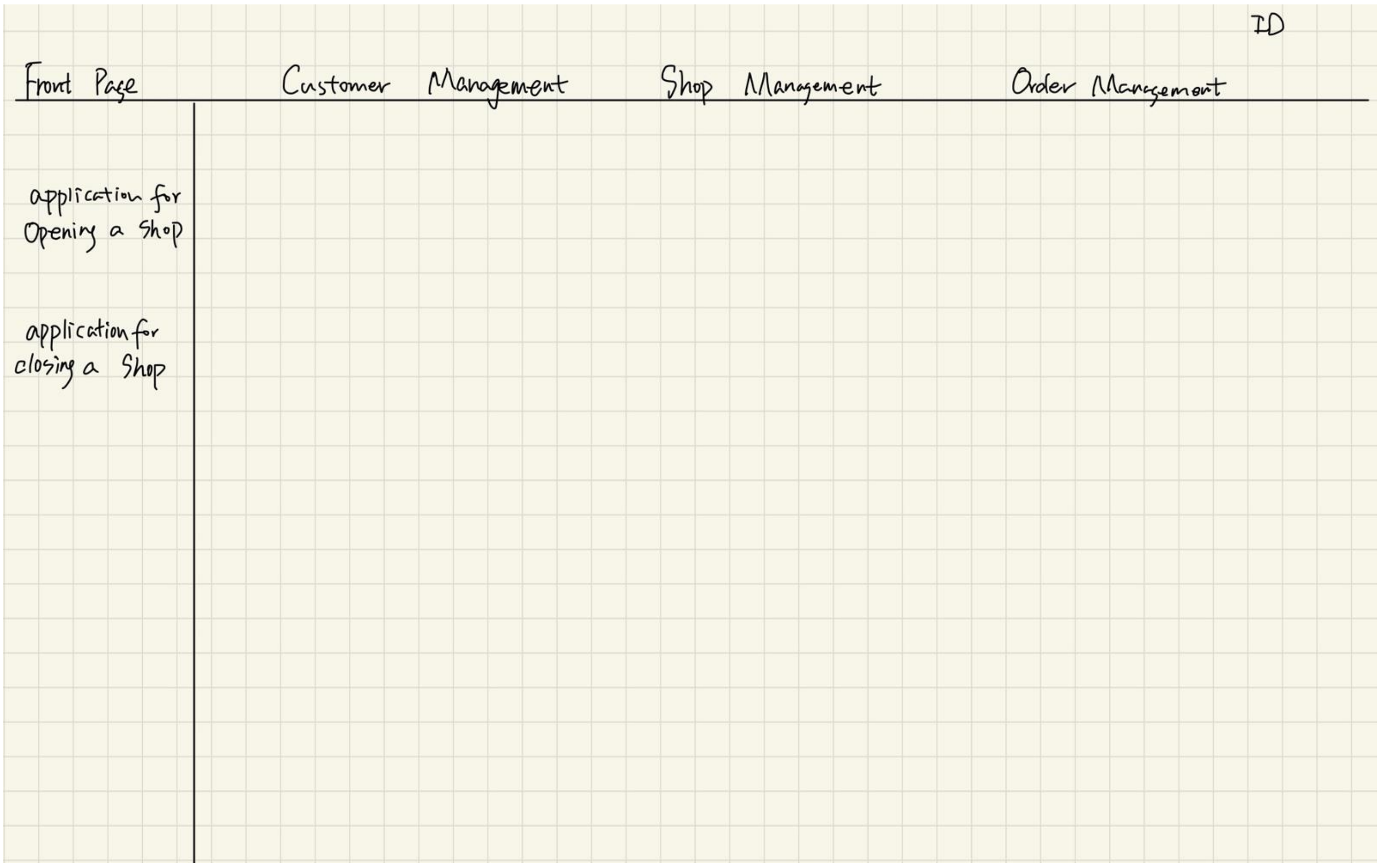


Figure 7. Design of administrator system

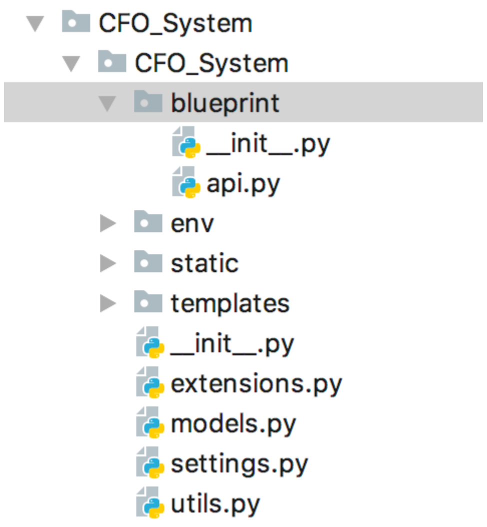
Implementation

Our group uses common light-weighted framework to complete this project. To make the developing process more structural and easier, we choose Python as programming language and "Flask + Vue" solution.

The first reason of using this solution is convenience. Flask is a light-weighted web application framework, and Vue.js is a frontend JavaScript framework which is compatible with Python and Flask. Both of them are supported by active communities and various extension packages. Therefore, we can obtain help easier when we encounter bugs or other problems. Another reason of using the two frameworks is agility. Light-weighted frameworks consist of the core and extensions. This kind of structure provides functional supports by installing extensions, and we can efficiently decouple the modules of the program. Subsystems can be developed concurrently without influencing each other. Hence testing and debugging workloads are decreased.

a) Backend Implementation

The backend code structure is like this:



```
▼ CFO_System
  ▼ CFO_System
    ▼ blueprint
        __init__.py
        api.py
    ▶ env
    ▶ static
    ▶ templates
      __init__.py
      extensions.py
      models.py
      settings.py
      utils.py
```

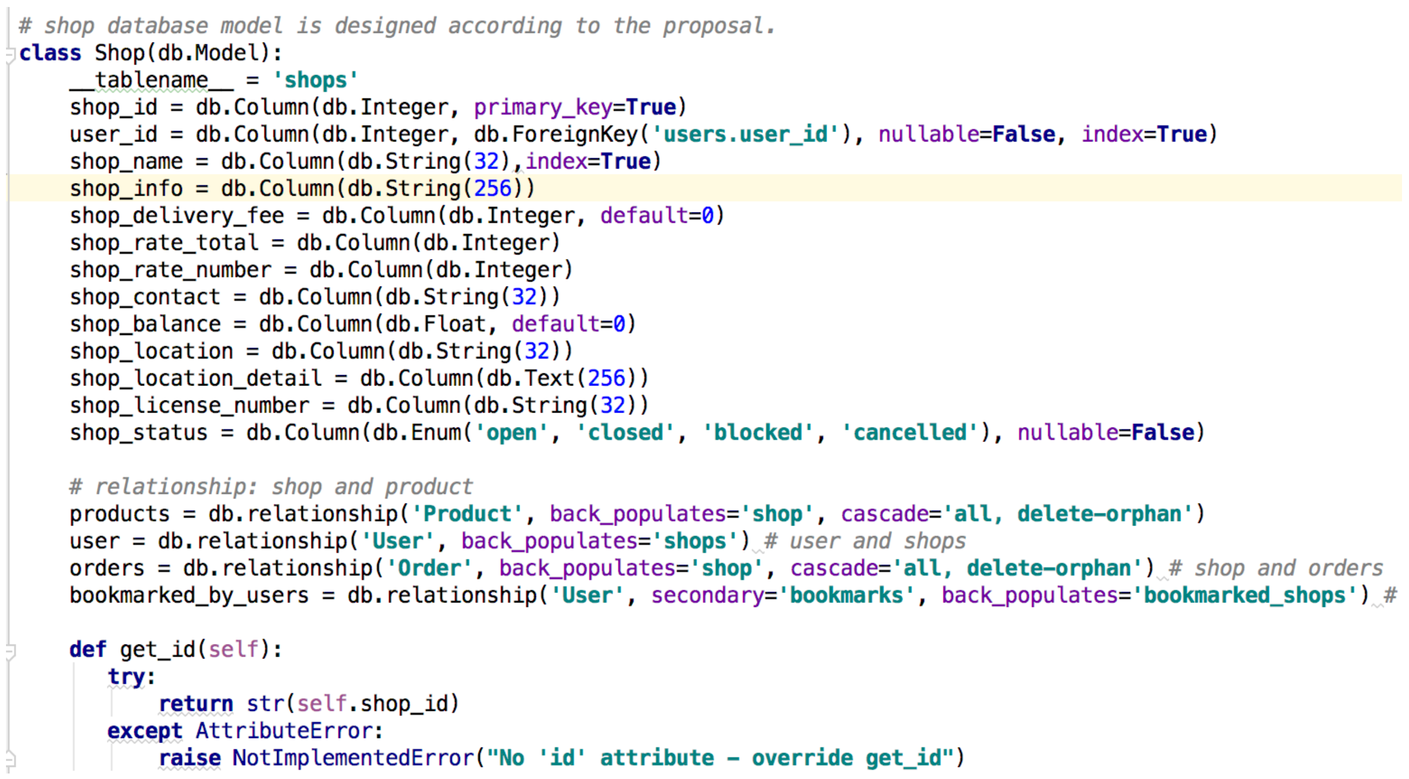
Figure 8. Backend code structure

The main modules corresponding to different functions are as follows:

1. `__init__.py`: initializes the web app, such as loading database objects, holding context, registering blueprints, etc.
2. `blueprint/api.py`: defines API blueprints that process database transactions and business logics.
3. `extensions.py`: initializes and stores extensions.
4. `models.py`: defines database models in Python classes.
5. `settings.py`: stores global variables and global contexts.
6. `utils.py`: stores auxiliary methods which are constantly used.
7. `env`: stores environmental variables.

We construct backend code according to the suggestions in Li Hui's book. First, we implement database models. Second, we define APIs about data transactions. Once the APIs are specified, the frontend can connect them with the UI interface and complete the web application.

To implement our database design with Python, we use the Flask extension package Flask-SQLAlchemy. It provides APIs allowing users to manipulate database file by Python ORM objects. An example of our database implementation is shown in Figure 9. Database tables can be represented by classes. Attributes such as `shop_id`, `user_id` are defined with `db.Column()` method. Relationships between tables are defined with `db.relationship()` method. Self-defined methods can also be attached to the classes. Just like the `Shop` class shows, `get_id()` method is rewritten in order to provide correct primary key values to login modules.

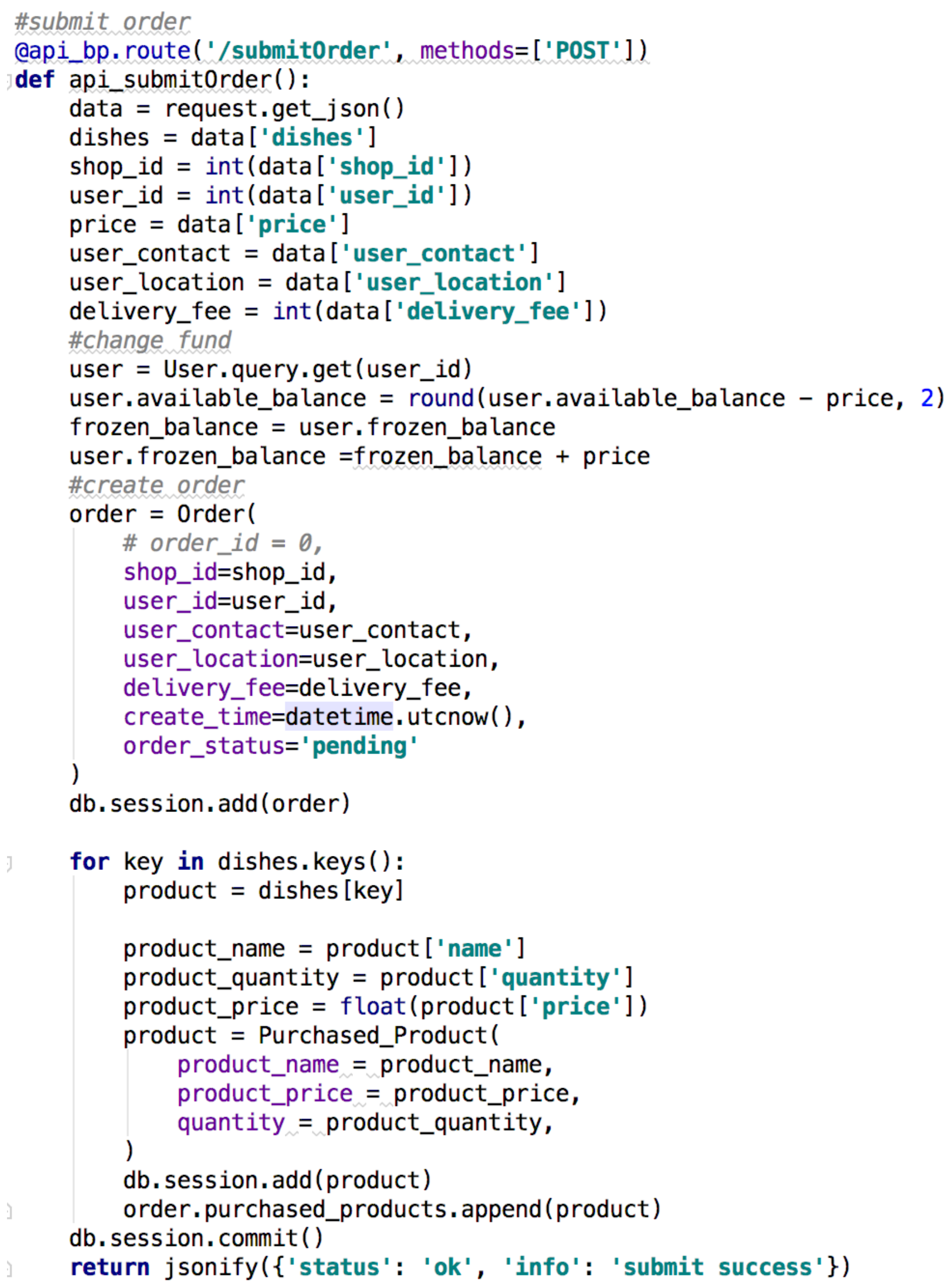


```
# shop database model is designed according to the proposal.
class Shop(db.Model):
    __tablename__ = 'shops'
    shop_id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.user_id'), nullable=False, index=True)
    shop_name = db.Column(db.String(32),index=True)
    shop_info = db.Column(db.String(256))
    shop_delivery_fee = db.Column(db.Integer, default=0)
    shop_rate_total = db.Column(db.Integer)
    shop_rate_number = db.Column(db.Integer)
    shop_contact = db.Column(db.String(32))
    shop_balance = db.Column(db.Float, default=0)
    shop_location = db.Column(db.String(32))
    shop_location_detail = db.Column(db.Text(256))
    shop_license_number = db.Column(db.String(32))
    shop_status = db.Column(db.Enum('open', 'closed', 'blocked', 'cancelled'), nullable=False)

    # relationship: shop and product
    products = db.relationship('Product', back_populates='shop', cascade='all, delete-orphan')
    user = db.relationship('User', back_populates='shops') # user and shops
    orders = db.relationship('Order', back_populates='shop', cascade='all, delete-orphan') # shop and orders
    bookmarked_by_users = db.relationship('User', secondary='bookmarks', back_populates='bookmarked_shops') #

    def get_id(self):
        try:
            return str(self.shop_id)
        except AttributeError:
            raise NotImplementedError("No 'id' attribute - override get_id")
```

Figure 9. Implementation of the `shops` table

Following the implementation of database models is the API realization. APIs are functions registered to specific URLs, which receive certain HTTP requests, process data and return responses to the frontend. Besides request/response processing, backend business logic is all about data transactions, including creation, retrieval, update and deletion (CRUD). We allocate each use case to an API method. Each method handles unique jobs. When an HTTP request is received by the web application, it will resolve the URL and invoke corresponding API method to modify data and complete the desired functionality.

Take the functionality of order transaction as an example. To complete a normal order transaction, first the customer creates an order, and then, the shop owner is able to see and accept the new order in the shop dashboard. The shop owner can change the status of the order (e.g. from "accepted" to "delivering"). Finally, the customer can confirm the receipt. Meanwhile, money is transferred from the customer to the shop. Down to API points of view, several API methods are involved: `api_submitOrder()`, `shop_orders()`, `shop_order_detail()`, `change_order_status()` and `pay_order()`. We may look closer into the `api_submitOrder()` method as Figure 10 shows. When a POST request comes, the method will read jsonified request data and get variables. This process is called "deserialization". Then, the method will change the account balance of the customer who creates the order. The amount of cost will be transferred from the `available_balance` to the `frozen_balance` when creating the order, and the `frozen_balance` will be transferred to the shop account after the order is finished. Next, a new order is created and added to the database, and the products ordered are added to the `purchased_products` relationship of the order. Finally, the method commits database changes and returns a response.

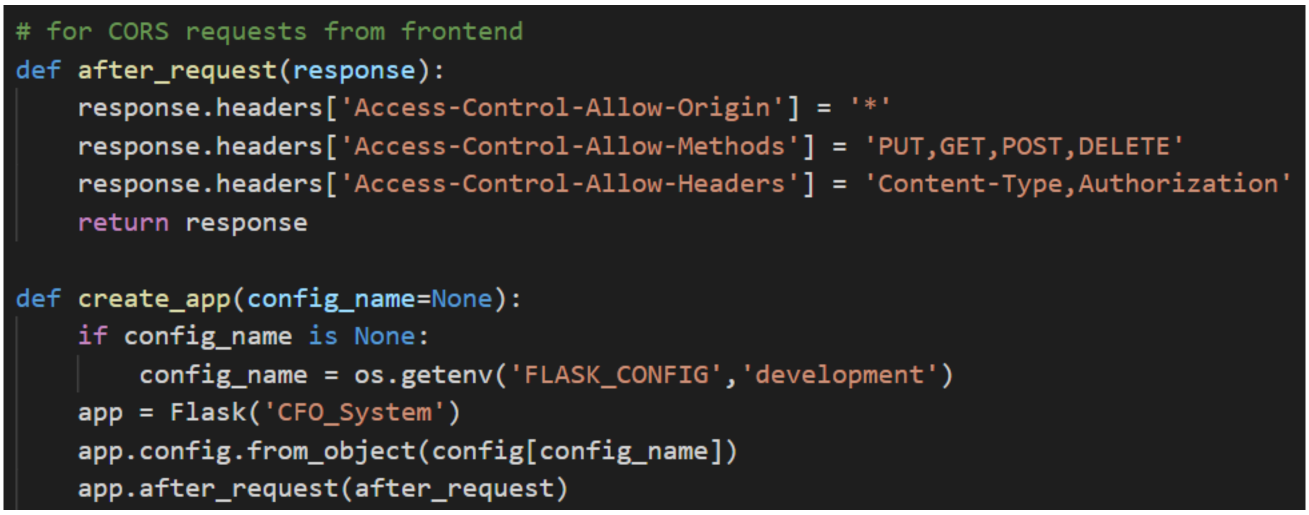


```python
#submit order
@api_bp.route('/submitOrder', methods=['POST'])
def api_submitOrder():
    data = request.get_json()
    dishes = data['dishes']
    shop_id = int(data['shop_id'])
    user_id = int(data['user_id'])
    price = data['price']
    user_contact = data['user_contact']
    user_location = data['user_location']
    delivery_fee = int(data['delivery_fee'])
    #change fund
    user = User.query.get(user_id)
    user.available_balance = round(user.available_balance - price, 2)
    frozen_balance = user.frozen_balance
    user.frozen_balance =frozen_balance + price
    #create order
    order = Order(
        # order_id = 0,
        shop_id=shop_id,
        user_id=user_id,
        user_contact=user_contact,
        user_location=user_location,
        delivery_fee=delivery_fee,
        create_time=datetime.utcnow(),
        order_status='pending'
    )
    db.session.add(order)

    for key in dishes.keys():
        product = dishes[key]

        product_name = product['name']
        product_quantity = product['quantity']
        product_price = float(product['price'])
        product = Purchased_Product(
            product_name = product_name,
            product_price = product_price,
            quantity = product_quantity,
        )
        db.session.add(product)
        order.purchased_products.append(product)
    db.session.commit()
    return jsonify({'status': 'ok', 'info': 'submit success'})
```

Figure 10. Code of `api_submitOrder()` method

Although different API methods perform different operations, but the general steps are just like those in `api_submitOrder()`. One important advantage of organizing functionalities with APIs is that APIs are modularized and decoupled. Functionalities are independent to the greatest extent, which is convenient for the development, debugging, and maintenance of the system. Also, APIs only provide business logics and data manipulations. Representational logics such as routing, template rendering, form validation and visualization are completed by the frontend. This provides more flexibility to develop interactive web applications.

Because the backend and frontend work on two different ports, the communication between them requires cross-origin resource sharing (CORS). The backend supports CORS by importing Flask-Cors.



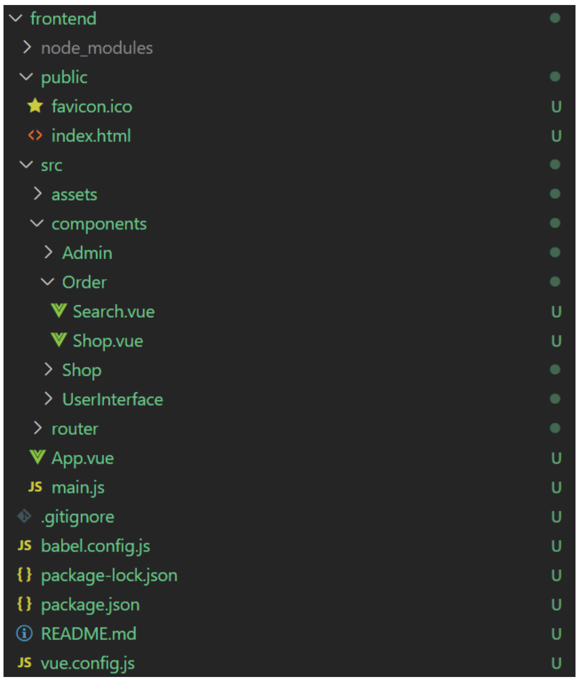
```python
# for CORS requests from frontend
def after_request(response):
    response.headers['Access-Control-Allow-Origin'] = '*'
    response.headers['Access-Control-Allow-Methods'] = 'PUT,GET,POST,DELETE'
    response.headers['Access-Control-Allow-Headers'] = 'Content-Type,Authorization'
    return response

def create_app(config_name=None):
    if config_name is None:
        config_name = os.getenv('FLASK_CONFIG','development')
    app = Flask('CFO_System')
    app.config.from_object(config[config_name])
    app.after_request(after_request)
```

Figure 11. CORS implementation in backend

b) *Frontend Implementation*

We construct the frontend node with Vue-CLI. This is a full system for rapid Vue.js development, providing interactive project scaffolding and a runtime dependency, which is upgradeable, configurable via in-project config file and extensible via plugins. Another important plugin is Vue Router, which is the official router for Vue.js. It deeply integrates with Vue.js core to make building Single Page Applications with Vue.js a breeze.

The frontend code structure is like this:



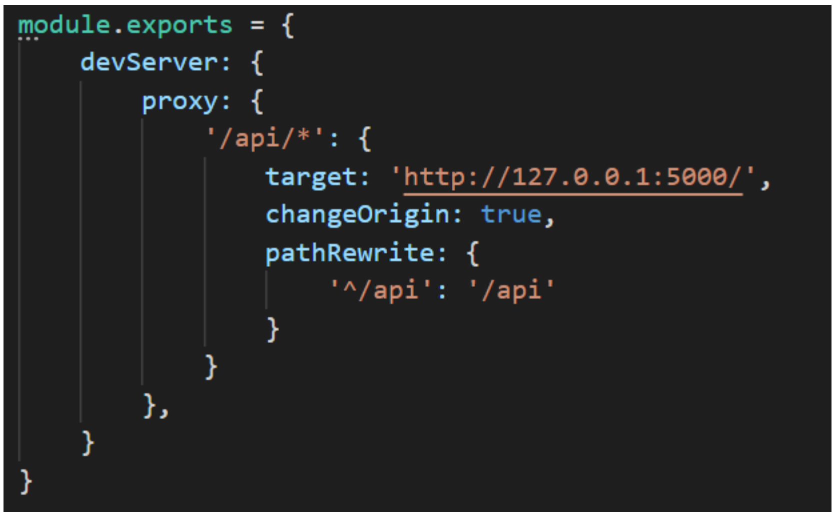
```
frontend
  node_modules
  public
    favicon.ico           U
    index.html            U
  src
    assets
    components
      Admin
      Order
        Search.vue        U
        Shop.vue          U
      Shop
      UserInterface
    router
    App.vue               U
    main.js               U
  .gitignore              U
  babel.config.js         U
  package-lock.json       U
  package.json            U
  README.md               U
  vue.config.js           U
```

Figure 12. Frontend code structure

The main modules corresponding to different functions are as follows:

1. `src/main.js`. This is the main page and the entrance of the project. It will initialize the Vue instance and insert required plugins.
2. `src/App.vue`. This will be initialized by main.js, and all the pages are switched under `App.vue`. All the pages are child components of `App.vue`.
3. `public/index.html`. This is the mount point of `App.vue`, and is showed to users. Since all pages are child components of `App.vue`, the switching of pages exactly happens in this `index.html`.

4. `vue.config.js`. This is the config file of Vue-CLI project. In this project, it controls the proxy request from the frontend to the backend.

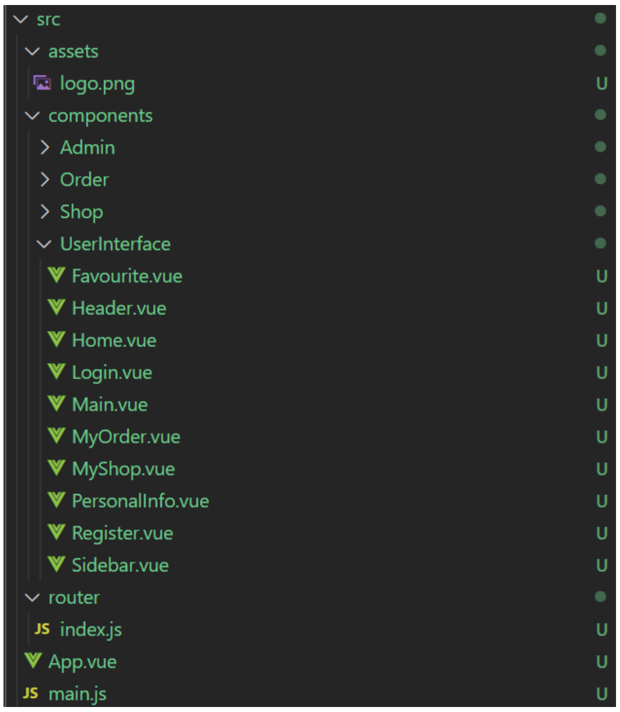


```
module.exports = {
    devServer: {
        proxy: {
            '/api/*': {
                target: 'http://127.0.0.1:5000/',
                changeOrigin: true,
                pathRewrite: {
                    '^/api': '/api'
                }
            }
        },
    }
}
```

Figure 13. Configuration in `vue.config.js`.

Flask backend opens at port 5000 so that the proxy requires cross-origin resource sharing. CORS is implemented in backend.

5. `package.json` and `package-lock.json`. They include the dependency of the frontend.
In the `src` folder:



```
src
  assets
    logo.png                U
  components
    > Admin
    > Order
    > Shop
    UserInterface
      Favourite.vue         U
      Header.vue            U
      Home.vue              U
      Login.vue             U
      Main.vue              U
      MyOrder.vue           U
      MyShop.vue            U
      PersonalInfo.vue      U
      Register.vue          U
      Sidebar.vue           U
  router
    index.js                U
  App.vue                   U
  main.js                   U
```

Figure 14. Src folder structure

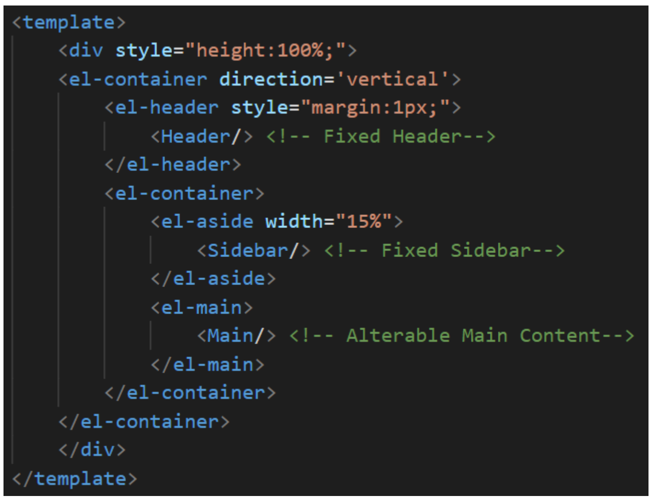
1. `./router/index.js`. This is a router that can control the mapping relationship between link path and component files in components folder.
2. All `.vue` files in `./components`. They are distributed into different folders according to their functions, which is convenient for organization.

With the MVC structure, the implementation of frontend can be divided into view and controller.

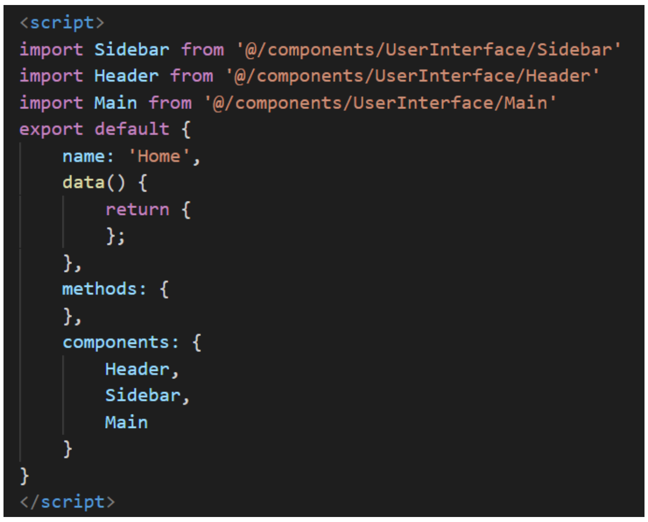
View

The implementation of view part is deeply dependent on Ant Design Vue and Element-UI, which also provide some interfaces between controller and view.

First, Element-UI provides global layout component, which is called container. It provides scaffolding basic structure of the page - header, side section and main section. Almost all pages follow the layout structure.



```
<template>
    <div style="height:100%;">
    <el-container direction='vertical'>
        <el-header style="margin:1px;">
            <Header/> <!-- Fixed Header-->
        </el-header>
        <el-container>
            <el-aside width="15%">
                <Sidebar/> <!-- Fixed Sidebar-->
            </el-aside>
            <el-main>
                <Main/> <!-- Alterable Main Content-->
            </el-main>
        </el-container>
    </el-container>
    </div>
</template>
```



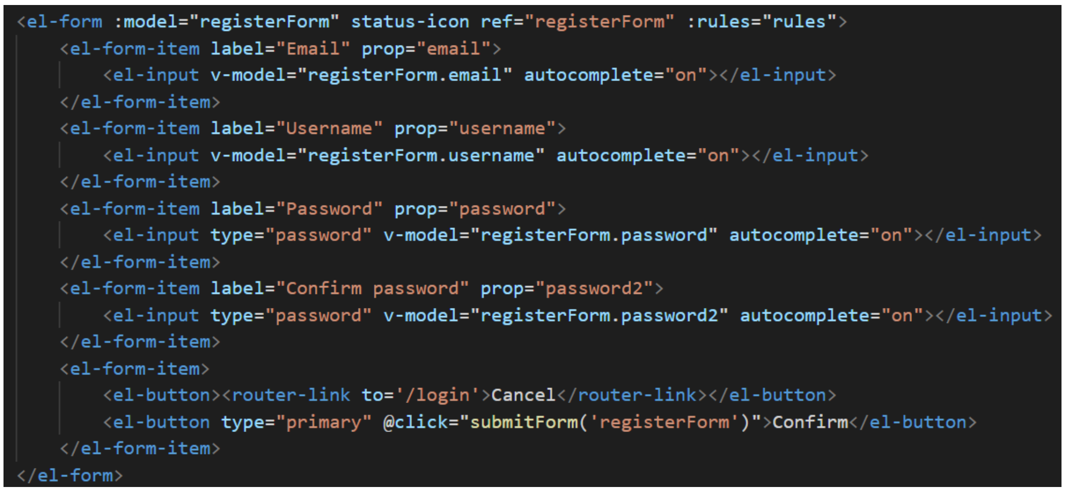
```
<script>
import Sidebar from '@/components/UserInterface/Sidebar'
import Header from '@/components/UserInterface/Header'
import Main from '@/components/UserInterface/Main'
export default {
    name: 'Home',
    data() {
        return {
        };
    },
    methods: {
    },
    components: {
        Header,
        Sidebar,
        Main
    }
}
</script>
```

Figure 15 & 16. Structure of `.vue` file

This is why we develop with Vue-CLI and Vue Router. Our target is to develop a multi-page application, but instead of constructing several .html file, we can use reusable Vue components by constructing the frontend with Vue-CLI, and organize them with Vue Router. Besides, plugins like Element-UI, Ant Design Vue and Axios can be easily used in the frontend node.

Back to the Vue component. The HTML elements are implemented between `<template>` labels. We use Element-UI and Ant Design Vue in Vue.js by adding labels similar to HTML labels.Several controls used in the implementation:

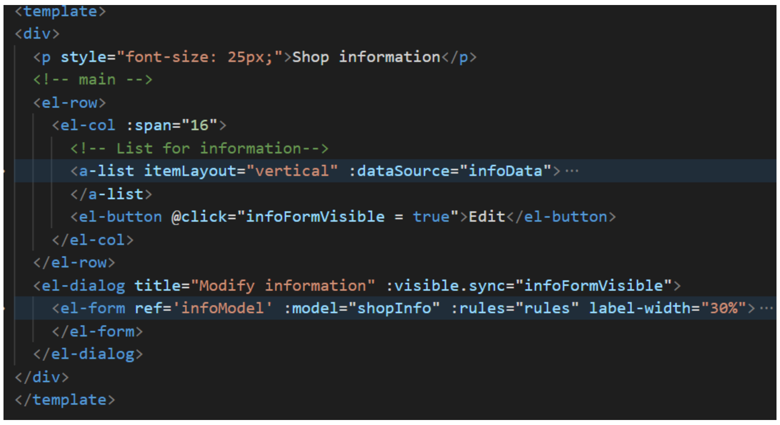
1. `<el-form>`. This is a packaging of input boxes, with unified form validating and submitting control.



```
<el-form :model="registerForm" status-icon ref="registerForm" :rules="rules">
    <el-form-item label="Email" prop="email">
        <el-input v-model="registerForm.email" autocomplete="on"></el-input>
    </el-form-item>
    <el-form-item label="Username" prop="username">
        <el-input v-model="registerForm.username" autocomplete="on"></el-input>
    </el-form-item>
    <el-form-item label="Password" prop="password">
        <el-input type="password" v-model="registerForm.password" autocomplete="on"></el-input>
    </el-form-item>
    <el-form-item label="Confirm password" prop="password2">
        <el-input type="password" v-model="registerForm.password2" autocomplete="on"></el-input>
    </el-form-item>
    <el-form-item>
        <el-button><router-link to='/login'>Cancel</router-link></el-button>
        <el-button type="primary" @click="submitForm('registerForm')">Confirm</el-button>
    </el-form-item>
</el-form>
```

Figure 17. Using `el-form` in `Login.vue`

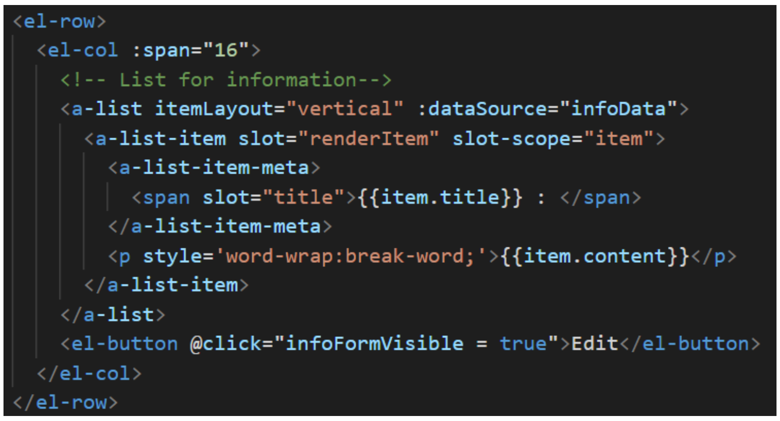
2. `<el-dialog>`. It provides the ability of informing users while preserving the current page state.
3. `<el-button>`. This is a packaging of buttons, with convenient combination of callback function.



```
<template>
<div>
  <p style="font-size: 25px;">Shop information</p>
  <!-- main -->
  <el-row>
    <el-col :span="16">
      <!-- List for information-->
      <a-list itemLayout="vertical" :dataSource="infoData">···
      </a-list>
      <el-button @click="infoFormVisible = true">Edit</el-button>
    </el-col>
  </el-row>
  <el-dialog title="Modify information" :visible.sync="infoFormVisible">
    <el-form ref='infoModel' :model="shopInfo" :rules="rules" label-width="30%">···
    </el-form>
  </el-dialog>
</div>
</template>
```

Figure 18. Using `el-button` and `el-dialog` in `InformationMain.vue`

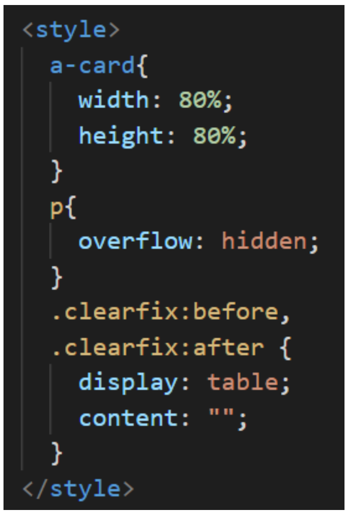
4. `<a-list>`. This is a packaging of `v-for` method, which can show the elements in a list circularly. It is used to handle the rendering of elements in a variable length list, which are dynamic loading.
5. `<el-row>` and `<el-col>`. They control the layout of the whole page.



```
<el-row>
  <el-col :span="16">
    <!-- List for information-->
    <a-list itemLayout="vertical" :dataSource="infoData">
      <a-list-item slot="renderItem" slot-scope="item">
        <a-list-item-meta>
          <span slot="title">{{item.title}} : </span>
        </a-list-item-meta>
        <p style='word-wrap:break-word;'>{{item.content}}</p>
      </a-list-item>
    </a-list>
    <el-button @click="infoFormVisible = true">Edit</el-button>
  </el-col>
</el-row>
```

Figure 19. Using `el-row`, `el-col` and `a-list` in `InformationMain.vue`

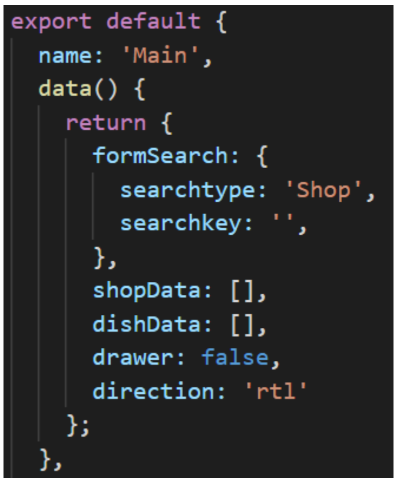
The controls above is widely used in the implementation of view part. In addition, Vue.js supports using CSS statements.



```
<style>
  a-card{
    width: 80%;
    height: 80%;
  }
  p{
    overflow: hidden;
  }
  .clearfix:before,
  .clearfix:after {
    display: table;
    content: "";
  }
</style>
```

Figure 20. Using CSS in Vue

The data required by the current page will be stored in `data()`. They can be dynamically bound to DOM elements, including showing, synchronizing with controls and receiving input.



```
export default {
  name: 'Main',
  data() {
    return {
      formSearch: {
        searchtype: 'Shop',
        searchkey: '',
      },
      shopData: [],
      dishData: [],
      drawer: false,
      direction: 'rtl'
    };
  },
```

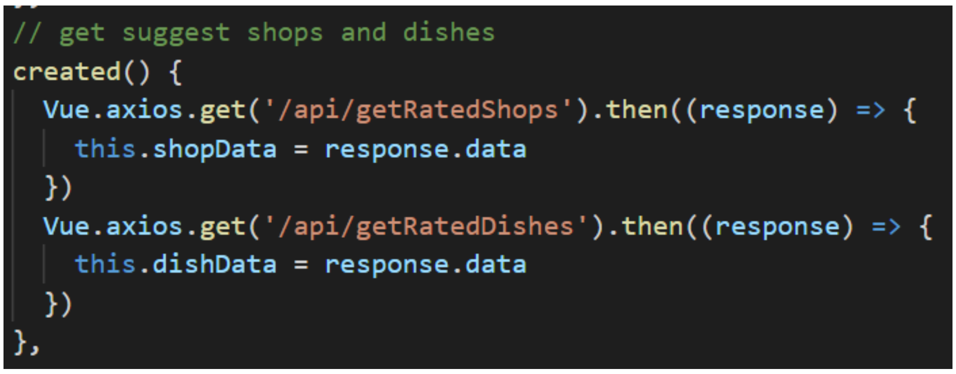
Figure 21. Using `data()` in Vue

Controller

API in the backend implements a part of controller. The rest part of controller is implemented in the frontend as communicating with the backend.

The way to realize the communication with the backend is using Axios. Axios is an HTTP client for the browser and Node.js, which can easily make HTTP requests from Node.js, including GET requests and POST requests.

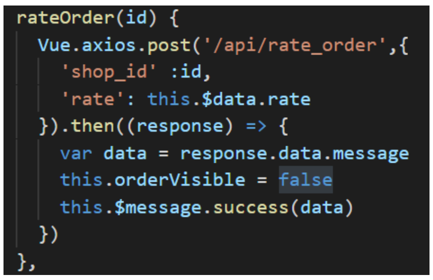
Most of the functions are implemented in `methods` part. Other of them are implemented as functions called lifecycle hooks, including “created” hook, “mounted” hook and “before created” hook. These functions run while the Vue instance is initializing so that the data can be dynamically injected from the backend to the frontend pages.



```
// get suggest shops and dishes
created() {
  Vue.axios.get('/api/getRatedShops').then((response) => {
    this.shopData = response.data
  })
  Vue.axios.get('/api/getRatedDishes').then((response) => {
    this.dishData = response.data
  })
},
```

Figure 22. Created hook

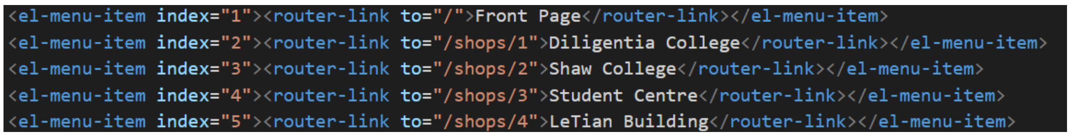
The functions in `methods` are asynchronous functions, such as callback functions of click or close actions. For example, when a rating of a shop is submitted, it will trigger a callback function that makes a POST request, including data of the rating in the body of HTTP request. API handles the submission, adds them into the database and returns a response. Axios handles the response asynchronously, and gets success or error signal.



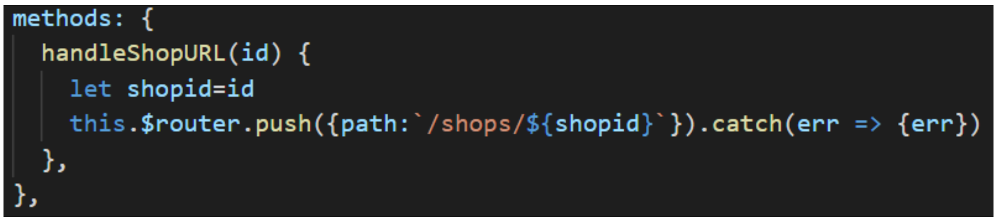
```
rateOrder(id) {
  Vue.axios.post('/api/rate_order',{
    'shop_id' :id,
    'rate': this.$data.rate
  }).then((response) => {
    var data = response.data.message
    this.orderVisible = false
    this.$message.success(data)
  })
},
```

Figure 23. Using Axios in Vue

Another kind of callback functions handles the jumps between pages. We use a built-in function of Vue.js called `this.$router.push`, or use label `<router-link>` on HTML elements.



```
<el-menu-item index="1"><router-link to="/">Front Page</router-link></el-menu-item>
<el-menu-item index="2"><router-link to="/shops/1">Diligentia College</router-link></el-menu-item>
<el-menu-item index="3"><router-link to="/shops/2">Shaw College</router-link></el-menu-item>
<el-menu-item index="4"><router-link to="/shops/3">Student Centre</router-link></el-menu-item>
<el-menu-item index="5"><router-link to="/shops/4">LeTian Building</router-link></el-menu-item>
```



```
methods: {
  handleShopURL(id) {
    let shopid=id
    this.$router.push({path:`/shops/${shopid}`}).catch(err => {err})
  },
},
```

Figure 24 & 25. Using `this.$router.push` and `router-link` in Vue

Conclusion & Further improvement

The implementation of our campus food ordering system covers most of our requirements. Flask and Vue.js are both simple for construction and maintenance. They can also be easily implemented on server with other network control software like Nginx. However, we still made some mistakes during development, and there is something that need further improvements.

At the beginning, we decided to use Flask and Vue.js to develop the project, but we have no experience of front-back separation. The backend part was developed with Jinja2 at first, which also provides a way to dynamically bind data with HTML DOM. It was able to use Vue in static HTML file, but the frontend part had constructed static web pages by Vue-CLI. They are hard to bind. Therefore, we have to rewrite the backend part and communicate the data by JSON. Fortunately, the development is successful in this way.

Some feathers that need further improvements:

1. Image support. It can be supported by sending image file with base64 encoding, but was not implemented due to the shortness of time and the hardness of debugging.
2. The security of login and register can be improved. The status of login is simply stored in session storage. To improve security, the backend should generate complex access token. For each time of jumping between pages, there must be a communication between the server and the client in order to ensure the status of user. This can be easily implemented by using Flask itself, but with using Vue.js and Flask and communication by JSON, it is more complex.
3. It has not been tested with large rate of flow. We do not have effective methods to simulate the test of large flows so that our test is just for the logic of operation.
4. The frontend is not unified in style. The components are constructed like "header + sidebar + main". However, some of them are implemented by importing header and sidebar, and some by importing header, sidebar and main, which means that the main part is an individual file.

Group Members

Huang Tianjian 117010099
Zeng Lewei 117010366
Wu Runzhong 117010289
Zhang Pinfang 117010383

Job Distribution

Job	**Person/People in charge**
Backend (shop owner & shop system, administrator system, transaction system)	Huang Tianjian
Structure design, Backend (customer system, transaction system)	Wu Runzhong
Frontend (design, administrator system, customer system, transaction system)	Zhang Pinfang
Backend APIs, Frontend (shop owner & shop system)	Zeng Lewei
Testing	All members

References

Project GitHub page: https://github.com/CSC4001/Campus-Food-Ordering-System/
Li Hui. (2018). Python web development with Flask. China Machine Press.
HTML Zhong Wen Wang: https://www.html.cn/
Vue.js: https://vuejs.org/index.html
Vue-CLI: https://cli.vuejs.org/
Vue Router: https://router.vuejs.org/
Axios.js: http://www.axios-js.com/zh-cn/
Ant Design. Ant Design of Vue. Retrieved from https://www.antdv.com/docs/vue/introduce-cn/
Element. Element UI of Vue. Retrieved from https://element.eleme.cn/#/zh-CN