Local market delivery service

John Fernandez 60090012
Passawit Umrod 60090024
Pollakit Ngamampornitthi 60090031

Bachelor of Engineering in Software Engineering
Department of Computer Engineering,
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Academic Year 2020

ISBN 13016291

Project Report - Academic Year 2020

Bachelor of Engineering in Software Engineering
Department of Computer Engineering, Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang

Title: Local market delivery service

Authors:

1. John Fernandez Student ID: 60090012
2. Passawit Umrod Student ID: 60090024
3. Pollakit Ngamampornitthi Student ID: 60090031

Approved for Submission

Isara Anantavrasilp
.....

(Asst.Prof.Dr. Isara Anantavrasilp)
Advisor

Date ...1.../...7.../...21...

Acknowledgments

This project is not possible without the help of Assistant Prof.Dr. Isara Anantavrasilp, our advisor, who gives advice, comments and provides us with many resources. Secondly, our fellow colleagues who have been criticizing us which ultimately contribute to the progress of this project.

Abstract

NearMe is a localised online food delivery service for small businesses, where customers are given a platform to order food from food stalls or shops in their local area. On the other hand, shop owners can use our platform to advertise, publish menus, and track incoming orders for delivering purposes. The system also allow shop owners to keep track of their sales for individual menu with sales report.

NearMe consists of 3 main components: the shop/marketplace management, online ordering, and sales tracking. Shop/market management is a front-end level system, where shop owners can register or remove their shop from our system after a verifying process, then customize their item list available for ordering and delivery. This customization includes item price, category, and product photo.

Online ordering is another front-end level system, where customer can find shops or specific dishes that he wish to order. The customer can select a market near his location and browse list of shops and products or search for specific product names.

Lastly, the Sales tracking system for the shop owners, where the list of items of a shop will be kept in the database. This system will keep track of the quantities of item sold/delivered, when a transaction is made. Shop owners can then request for a sales report from the system at a later date.

NearMe is mainly developed using Python web framework Django and JavaScript. On the client-side, the shop/marketplace management system and the online ordering system are developed using React native; which is a JavaScript library emphasising mobile application development. On the server-side, the application employs Postgres as a database.

Table of contents

List of figures

Chapter 1
Objectives and Problem Description

1.1 Problem description

In this modern era, it is becoming more apparent that a business without any digital outlet to interact with customers are limiting themselves from reaching potential buyers browsing through the internet. Most businesses in various fields are aware of this fact. Therefore, digital transformation, the process of bringing people, data, and digital processes together to create value for customers and maintain a competitive advantage in a digital-first world, is one aspect that every business is taking interested in. However, digital transformation in small businesses such as stalls in a market or stores in a cafeteria are not easily achieved, due to the restriction of low affording budget. To solve this problem, one can think about the process which can make low-budget stores easily bind themselves into the digital world.

Therefore, our aim is to create a application-based local-area ordering platform allowing all existing markets to register and bring all stores in their market to work with us similarly to food delivery applications. Instead of aiming to get more customers that may reside anywhere throughout the city, we focus on the customer residing near the shop's physical location. Customers residing near the shop's location means that the cost and time of traveling could be reduced. Therefore, the shop owner can just send their existing employee to deliver products instead of relying on a middle man for delivery.

1.2 Objective

The object of this project is to develop an online platform for existing shops in the physical market, in order to make their products easier to discover and approach by potential customers.

1.3 Scope of work

The scope of this project can be listed as follows:

- To develop an application where a market store owner can promote and advertise themselves and their products with a custom display, giving potential customers essential information such as price, opening and closing times, etc.
- To create a system where a customer can browse for nearby markets, stores within that market, buy products of their choosing, then have it delivered to them.
- With each delivery, aiming to utilize a verification system that makes sure each transaction is verified for completion.
- To develop a system where product sales of each store can be tracked, recorded, and displayed as a sales report, for further analytic usage.
- To eliminate the potential need for a middle delivery man, due to the fact that the system solely focuses on a single local area, which means that a store can use their own employee to deliver the product during hours where there are little to no customer.

1.4 Report structure

We will cover the motivations and objectives of this project leading to explaining related works and background knowledge. We will then present our proposed methods, implementations, and experimentation's closing off with a conclusion.

- Chapter 1 Introduction - introduces the concept of the project, brief background analysis, motivation, and most importantly provides overview of project aim.
- Chapter 2 Related work – proposes the Literature survey, a various existing software that are relevant to this project, and comparison.
- Chapter 3 Background knowledge - explains the knowledge and technology being adopted or related to this project
- Chapter 4 Requirement analysis/ Design/ Architecture – presents the requirement of the system, the use case diagram and relevant system architecture diagram
- Chapter 5 Development - show the concepts, tools and techniques used in developing the project

- Chapter 6 Preliminary result - shows and explains the current phase results of the project
- Chapter 7 Conclusion - explain what we have done, problem that we face, and the possible improvement

Chapter 2
Related Works

2.1 Food delivering business model

In today's world, many businesses are moving towards an online platform for the benefit of reaching a wider range of potential customers and better advertisement, while customers could also enjoy the benefit of browsing more variety of products in some cases even from overseas. Food Industry is not an exception to the digital transformation of businesses, in fact there are many service providers in food delivering business these days such as grab, foodpanda, Ubereat etc. In this section we will explore different business models of different food delivery services[11].

EXAMPLES	ORDERING	COOKING	DELIVERY
Just Eat, GrubHub, FoodPanda	Taken care of by the company	Partnered with Restaurants	Restaurants have their own delivery or use third party
Munchery, SpoonRocket, Sprig, Maple	Taken care of by the company	Taken care of by the company	Taken care of by the company
Postmates, Doordash, Caviar, Deliveroo	Taken care of by the company	Partnered with Restaurants	Taken care of by the company
RocketFood, Extra Plate	Taken care of by the company	Partnered with Home Chefs (Crowdsourced)	Taken care of by the company

Figure 2.1: Comparing food delivery platforms

2.1.1 The order only model

This model was used by the first generation of restaurant delivery services like JustEat and Grubhub in the early days are focused on acting as a pure software layer that aggregates a fragmented offering of independent restaurants, which may have their own fleet of couriers.

These software-only marketplaces' main selling point to restaurants is to bring a lot of new orders and replace their antiquated phone-ordering system with an optimised Web or mobile platform, that integrates with their kitchen workflow.

The limitation of this model is that prices, speed and quality of delivery wholly rely on the ability of the restaurant to manage their fleet of delivery men.

The advantage is the service provider is not required to hire and maintain any 3rd party delivery services, therefore the cost could be cheaper. Thus reducing the customer charging price[11].

2.1.2 The order and delivery model

This model similarly to the last model, provided restaurants with a software layer but additionally, the service provider also managed the delivery on restaurant's behalf, through a fleet of independent couriers connected by an Uber-like mobile app.

Like "order only model", ordering occurs on the app or website where the menu of a restaurant is uploaded. But the prices of items on the menu are usually marked up by the restaurants to cover the commission free of service provider's independent couriers. The customers are also charged a flat fee for every restaurant where you place an order. When an order comes in, it is sent to the couriers closest to the pick-up location.

The limitation of this model is that service providers themselves need to hire, train and supply and maintain courier equipment which is a significantly larger operation to do.

This model benefits from the ability to charge a higher amount of commission and the existence of independent courier offers a wider range of restaurants throughout the city without the limitation of restaurant location[11].

2.1.3 The fully integrated model

The last model, which includes Sprig, Maple and SpoonRocket has opted for full integration of the process, where the service provider has their own kitchen for their own meals preparation, an app through which consumers can order a limited range of meals, and their own fleet of courier. The food will either be reheated or fridge in their cars as orders come in, and delivered as fast as possible. The limitation of meals available means that service providers trade choices for convenience and a highly curated experience[11].

2.2 Services and business

2.2.1 Just Eat Takeaway

As a result of the merging between U.K. meal delivery app Just Eat and Dutch rival Takeaway, Just Eat Takeaway emerged as the lead food delivery service provider in Europe.[19].

In 2020, Just Eat Takeaway bought the American food delivery firm, Gruhhub, Allowing it to expanded its market into the U.S. and also being the biggest online food delivery operator outside China.[19].

Figure 2.2: How does Just Eat Takeaway work

Hybrid operation

Just Eat Takeaway offer shops' owner a choice to either use shops' own couriers or Just Eat couriers.

Shops with their own delivery capabilities may use their own fleet of couriers to delivery shop's product while using Just Eat application or website to only receive an order from customers.

Meanwhile, shops without the necessary resources for delivery operation may use Just Eat Takeaway couriers to delivery products to customers similarly to Grab, Uber or Panda.

Shops with own delivery capabilities may benefit from this system because they can control their own delivery price.

Commission price

Just Eat Takeaway charge 14 percent commission on the order value from the shop meaning the restaurant gains about 86 percent of the value.

Customers need to pay additional delivery fee which depends on the location difference between the shop and customer location.

2.2.2 Grabfood

Currently, grab is one of the lead service providers in the food delivering business in South East Asia. Grab provided an online platform for restaurants across the city to register with, the menu of restaurants' products will be uploaded to the application for customers to browse.

The application will show restaurants starting from the one closest to the customer location. The customer can choose their favourite restaurant, select a meal and place their order.

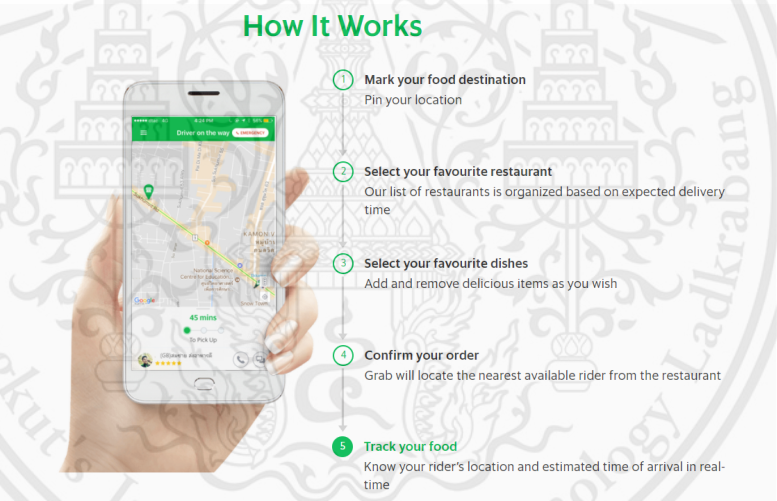

Figure 2.3: How does grab work

Courier operation

When the order is place, grab will locate and send the order to the closest courier. Couriers nearby restaurant location will be notified with a list of order from a customer, which they can either accept or reject. If a courier accepts the order then he will be on duty. The courier would be the one to order the food for the customer. While the courier is on duty, the customer can track and communicate with the courier through both of their applications which assures that products will be delivered [5].

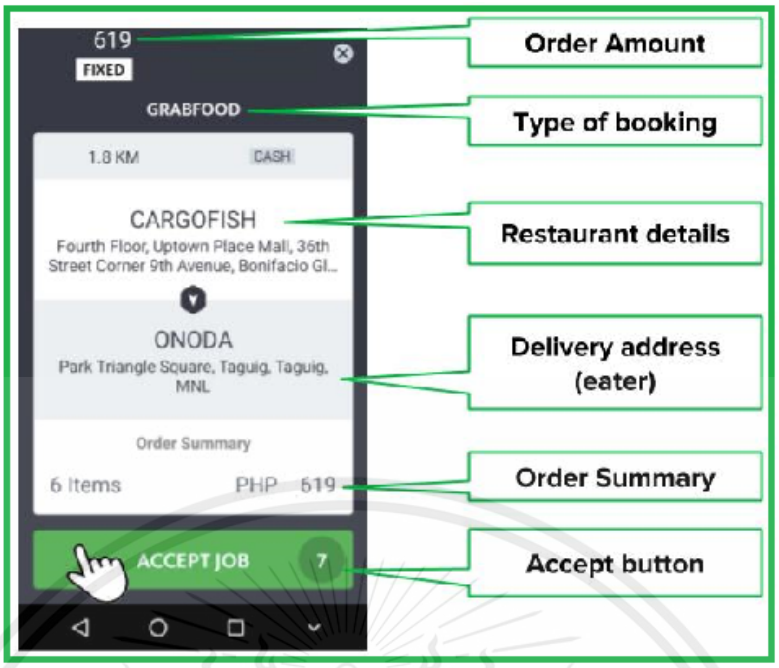

Figure 2.4: rider accept job

Shop application

When customers placed orders and courier accepted an order, the order will be automatically accepted and notify to the shop. Shop owner can view order detail from the pop-up notification of the order[5].

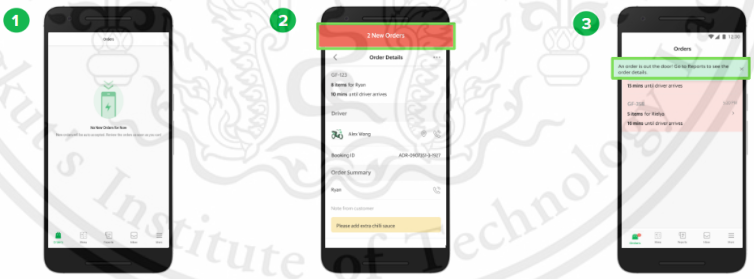

Figure 2.5: Shop recieve an order

Shop's owner can set their menu items and its prices in the application, if an item is unavailable or run out of an ingredient, shop's owner can set the state of item to be out of stock so the customer can not place an order[5].

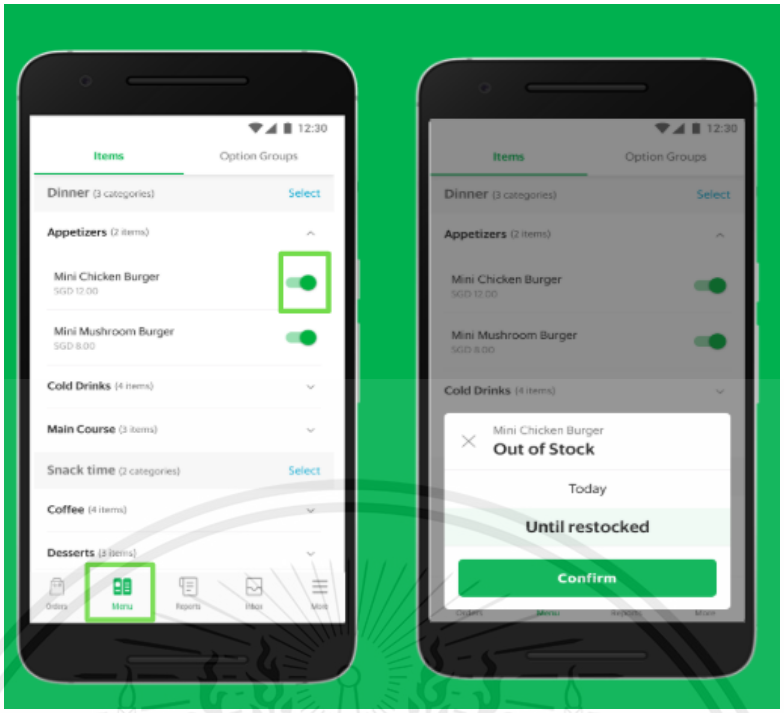

Figure 2.6: Out of stock

Shop's owner can set an opening and closing hour through the application, or if need be, shop can be temporary close at anytime[5].

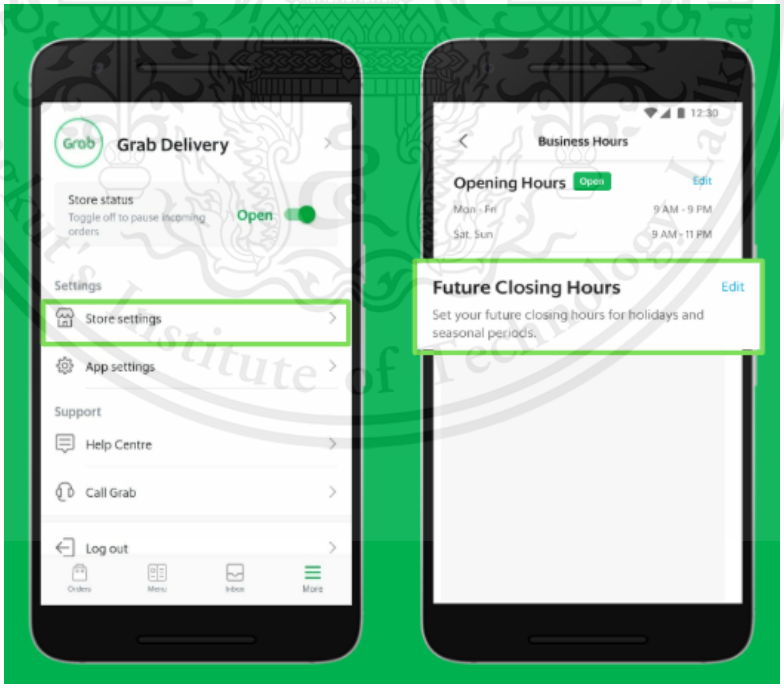

Figure 2.7: Opening and closing status

Commission price

Because Grab needs to train, supply, maintain and pay their fleet of independent couriers. They charge up to 25 to 30 percent commission on the order value from the shop meaning the shop gains about 70 percent of the value.

Customers need to pay additional delivery fee which depends on the location difference between the shop and customer location.[4].

Commission price calculation example

(numbers used may vary based on factors shared, using the higher end of commission at 30 percent)

- Total order: 18 USD
- Delivery fee: 3.40 USD (assuming 2.5 to 3 km delivery distance; delivery fee is customarily based on distance, supply-demand and may vary)
- Platform fee: 0.20 USD

Total paid by a consumer: 21.60 USD. Where does the money go?

- To the restaurant: 12.60 USD (70 percent of order)
- To the delivery partner: Approximately 8.00 USD (delivery fee 3.40 USD and Grab's top up of delivery fee + incentive to ensure they earn enough 4.60 USD)
- To Grab: 1.00 USD (30 percent commission 5.40 USD to partner incentive 4.60 USD + platform fee 0.20 USD)

In this example, out of the 18 USD order, the effective breakdown:

Effective % breakdown	**Total**	Order $18	Delivery fee $3.40	Platform fee $0.20
Restaurant keeps 70%	**$12.60**	12.60		
Delivery partner keeps 25%	**$8.00**	4.60	3.40	
Grab keeps 5% *(It's +/- a few percent for different size orders)*	**$1.00**	0.80		0.20

Figure 2.8: Grab fee breakdown table

Customer payment

Grab accept various method of payment such as cash, PayPal, credit or debit card and GrabPay.

Chapter 3
Requirement Analysis / Design / Architecture

3.1 Functional requirement

The project is made with objectives of local area ordering platform as a core feature and onsite purchase feature to enhance shop owner business productivity as the secondary objective. Therefore the function requirement will revolve around features and how to make the application to operate properly.

3.1.1 System

- The system can locate the nearby market to the customer favourite location.
- The system can generate sales reports for the shop owner.
- The system can make an onsite purchase.

3.1.2 Market admin

- Market admin can add, edit, and remove markets in their market chain.
- Market admin can add, edit, and remove shops in their markets.
- Market admin can add, edit, and remove delivery location in their market chain and assign added delivery locations to each of their market.
- Market admin can view the summarized sales report of each shop in their market chain.

3.1.3 Shop owner

- The shop owner can edit the shop owner profile.
- The shop owner can edit the shop profile.
- The shop owner can add, edit, or remove menu categories in their shop.
- The shop owner can add, edit, or remove products in each of their menu categories.
- The shop owner can add, edit, or remove decorators in their shop and assign decorators to each of their product.
- The shop owner can get active order or order that is ordered by the customer but is not completed or canceled yet.
- The shop owner can receive or reject the order.
- The shop owner can mark an order as having been delivered and remove it from the list of active orders.
- The shop owner can set the order as paid.
- The shop owner can perform onsite purchase.
- The shop owner could set open and close the shop.
- The shop owner can set any specific menu category, product, and decorator's state to "run out" if they are already run out at the shop.
- The shop owner can see the sales report.
- The shop owner can print the receipt that contains pricing information and customer information before handing the product(s) and receipt to his underling for delivery.

3.1.4 Customer

- The customer can register, and edit his user profile.
- The registered and non-registered customer can search for the registered market chain in the system that is still active.
- The registered and non-registered customer can search for the markets in each market chain that is still active.

- The registered and non-registered customer can search for the shops in each market that is still active.
- The registered and non-registered customer can look for menu categories and products in each shop that is still active.
- The registered and non-registered customer can see market chains, markets, shops, and products detail.
- The registered customer can add, and remove their favourite delivery location.
- The registered customer can add, and remove their favourite shop.
- The customer search for shops that can deliver product to their favourite location.
- The customer get their favourite shops.
- The customer can add products to the cart.
- The customer can review the current order.
- The customer can remove an item/remove all items from their cart.
- The customer can place an order and receive confirmation.
- The customer can cancel the order.
- The customer see their order history and preform re-purchase.

3.2 Non-functional requirement

3.2.1 Usability

- Well-designed and intuitive GUI.
- Menu should be able to correctly show which items run out and customers should not have the ability to order the run out item.
- Shop, market, and market chain list should show only stores that are currently active or open.

3.2.2 Reliability

- The summation price of all products customer orders should always be correctly shown to the customer.
- The shop owner must receive an order within one minute starting from the customer placing the order.
- The application should always correctly show all products the customer has ordered in the ordering list for the shop owner to see and should not miss any item.

3.2.3 Supportability

- The system must be able to support mobile application access.

3.3 Use Cases

Customer

- Login : After registration, Customers can log-in to the system by opening our application, then fill in email and password in each respective field. Click on the button “Log-in”, if both email and password are correct, the log-in process will be successful and customers will be redirected to the landing page.
- Reset password : On the login page under password field, click on “forgot password” a pop up will appear to let the customer type in their email. The reset instruction will be sent to their email.
- View list of market : After a successful login, customers will see a list of canteens/market in their local area, for example, in KMITL area, users may see Canteen A, Canteen L, Billion Park, Phra thep canteen etc.
- View list of shops : Clicking on individual canteens/market will show a list of shops in the canteen, for example, if customers click on canteen A, they can see a list of shops in that canteen.
- Find shops : Alternatively, if customers already know which shop they want to order from and don’t want to navigate through a list of canteens and a list of shops. Customers may type the name of the shop directly in the text field above, which then, the page will show an individual searched shop instead of a list.

- Favourite shops : If customers have shops where they usually purchase from, they can click on a grey star in the corner of shops banner. The star will turn orange after clicking on, indicating that the shop is already favourite.
- View Favourite shops : To easily view favourite shop without either typing in the text field or search for the shop through list of many shops, customer can navigate to Favourite shops page by clicking on “Favourite shops” down below in the navigation bar, the page will show a list of any shops that user already favourite.
- shop menu : When customers find their preferred shop, they can click on the shop to see its shop menu, the menu will show all available products and its prices.
- Order products : To order foods from the shop, customers must click on a product they want to order from the shop menu, a single number of that item will be added to the basket. Customers can then change each product quantity then confirm the order.
- Confirm order : To finalise the order, customers are required to enter a delivery location and their phone number. Customers can also send a note to the shop which might just be a request such as “not spicy”. Pressing on “confirm” will now place an order to the shop. The pop-up indicating the success will appear ending the ordering steps.
- View list of order : To see currently process order and pass order, customers can navigate to the order history page. The page will show a list of orders, which will have a date, shop name, total sum and order state of each order on the list.
- View order information : To view more information of each order, customers can click on the order they want to look at. The page will show additional menu information namely, delivery person name and phone number, list of ordered items’ and their quantity, and the sum.
- Cancel order : If the customers want to cancel their order, click on the order in the list. Click on “cancel” will remove that order from the list.
- Edit customer profile : Clicking on “Profile” in the navigation bar will lead to the profile page. From the page, customers can change their name, phone number, email address and set a default delivery address.
- Log out : Clicking on “Profile” in the navigation bar will lead to the profile page. Click on “Logout” to log out of the system.

Shop owner

- Login : After registration, Shop owners can log-in to the system by opening our application, then fill in email and password in each respective field. Click on the button “Log-in”, if both email and password are correct, the log-in process will be successful and Shop owners will be redirected to the landing page.
- Reset password : On the login page under password field, click on “forgot password” a pop up will appear to let shop owners type in their email. The reset instruction will be sent to their email.
- View shop menu : Shop owners can view their own shop menu by clicking on “menu” in the navigation bar below, the menu page will show every shop products and its price and category.
- Add new category to the menu : On the menu page, clicking on “add category/product” will redirected shop owners to a page where they can add a category by enter the category name then confirm.
- Add new product to the menu : On the menu page, clicking on the “add category/product” button redirects shop owners to a page where they can add an product by choose a category from a drop down menu, add product image, name and price then confirm.
- Edit product information : Product information can be edited by long clicking an individual Product. Shop owners will be redirected to another page where they can change item name, price, picture or category.
- Delete product : Product can be deleted by long clicking an individual product. Shop owners will be redirected to another page with a “delete” button, which when clicked will delete the item from the menu.
- Edit category information : Category information can be edited by long clicking an category item. Shop owners will be redirected to another page where they can change category name.
- Delete category : Categories can be deleted by long clicking an individual category. Shop owners will be redirected to another page with a “delete” button, which when clicked will delete the category and entirety of its product.

- Change product state : Each product will have a toggle button behind its name. If a product runs out or unavailable at the moment, shop owners can click on the button to hide the product from customers.
- Shops Food Ordering : When a customer comes to the storefront to order anything from the shop, shop owners perform onsite selling to record the transaction to our system. To order foods from the shop, Shop owners must click on a product they want to order from the shop menu, a single number of that item will be added to the basket. Shop owners can then change each product quantity then confirm the order. Confirming an order will also include the product in the sales report.
- View list of order : To see currently processed order and pass order, Shop owners can navigate to the order page by clicking on "order". The page will show a list of orders, which will have a date, order ID, total sum and order state of each order on the list.
- View order information : To view more information of each order, shop owners can click on the order they want to look at. The page will show additional order's information namely, list of ordered product and their quantity, the sum, delivery location, customer name and customer phone number.
- Cancel order : If the shop owner wants to decline an order, click on the order in the list. Click on "cancel order" will change the state of order to "canceled".
- Accept order : If the shop owner wants to accept an order, click on the order in the list. Click on "accept order" will change the state of order to "received".
- End transaction : When customers receive their meal, and shop owners receive their cash and the transaction is still on, shop owners can click on the "delivered" button in the order information page to end the transaction, changing the state of the order to "done".
- View sales report : Clicking on "Report" in the navigation will lead to a page showing list of total items sold out, and the sum of the items ordered both directly from the shop and online order. The report can be chosen to show either in daily, weekly or monthly by choosing from the collapse menu above.
- Edit shop profile : Clicking on "Profile" in the navigation bar will lead to the profile page. From the page, shop owners can change shop category, shop description, shop location, set a shop picture and set shop state to either "close" or "open". Closed shops will not be shown on customers' shop list.

- Log out : Clicking on “Profile” in the navigation bar will lead to the profile page. The shop owner will be able to see a log-out button on the profile page, clicking on it will log out the shop owner from the system.

Market owner

- Login : After registration, market owners can log-in to the system by opening our application, then fill in email and password in each respective field. Click on the button “Log-in”, if both email and password are correct, the log-in process will be successful and market owners will be redirected to the landing page.
- Reset password : On the login page under password field, click on “forgot password” a pop up will appear to let Market owner type in their email. The reset instruction will be sent to their email.
- View list of shops : On the market page, market owners are able to see a list of shops in their market.
- View shop information : On the market page, clicking on any individual shop banner will redirect the market admin to a page with the shop information namely, shop category, shop name, shop description, location, shop owner name and phone number, and shop image.
- Find shops : Alternatively, market owners can search for any specific shop by typing in shops name in the text field above the page. Which will instead show only the matched result of the searched shop.
- Add Shops to market : On the market page, clicking on the “add shop” button down below will redirect shop owners to a page where they can add a shop to the market by entering the shop category, shop name, shop description, location, shop owner name and phone number, and shop image then click on “confirm”.
- Remove Shops from market : On the market page, clicking on any individual shop banner will redirect the market admin to a page with the shop information, there will be a “delete” button down below. Clicking on it will delete the shop and its entire information.
- View sales report : Clicking on “Report” in the navigation bar below will lead to a page showing a list of total income of each shop and sum of it. The report can be chosen to show either in daily, weekly or monthly by choosing from the collapse menu above.

- Edit market profile : Clicking on “Profile” in the navigation bar below will lead to the profile page. From the page, market owners can change market name, market location , and set a canteen picture.

Figure 3.1 illustrates use case diagram corresponding to use cases mentioned above.

3.4 System architecture

Our system architecture is a client-server architecture and a monolithic architecture. The diagram is shown in the figure 3.3.The architecture can be divided into two main parts, client-side and server-side. The client-side can communicate to the server-side via HTTP protocol. The request and response of HTTP protocol are sent in RESTful API structure. On the client-side, we provide a mobile application for each type of user, which in total, there are three mobile applications. They were developed using the React Native framework with Javascript as the primary programming language.

On the server-side, we have a web server deployed on the Heroku cloud platform and an additional media storage. The web server has three main components: web server gateway interface (WSGI) server, web application, and database. We selected Gunicorn as our WSGI server. The WSGI will be the interface that handles the communication between the client and the web application. Next, the web application was developed using Django and Django REST Framework with Python as the primary programming language. As for the data, it is divided into two parts: text and image. Text will be stored in the database which we choose PostgreSQL as our database since it is well compatible with the Django framework. PostgreSQL database is host in the same Heroku dyno as the webserver. For the image, unfortunately, Heroku will shut down the dyno that has no request within 30 minutes. This means that the stored images in the static file storage will lose after no request getting to the server. Therefore, we decided to store images in another cloud service. Cloudinary is a media cloud storage that runs cloud service at the service as a service (SaaS) level.

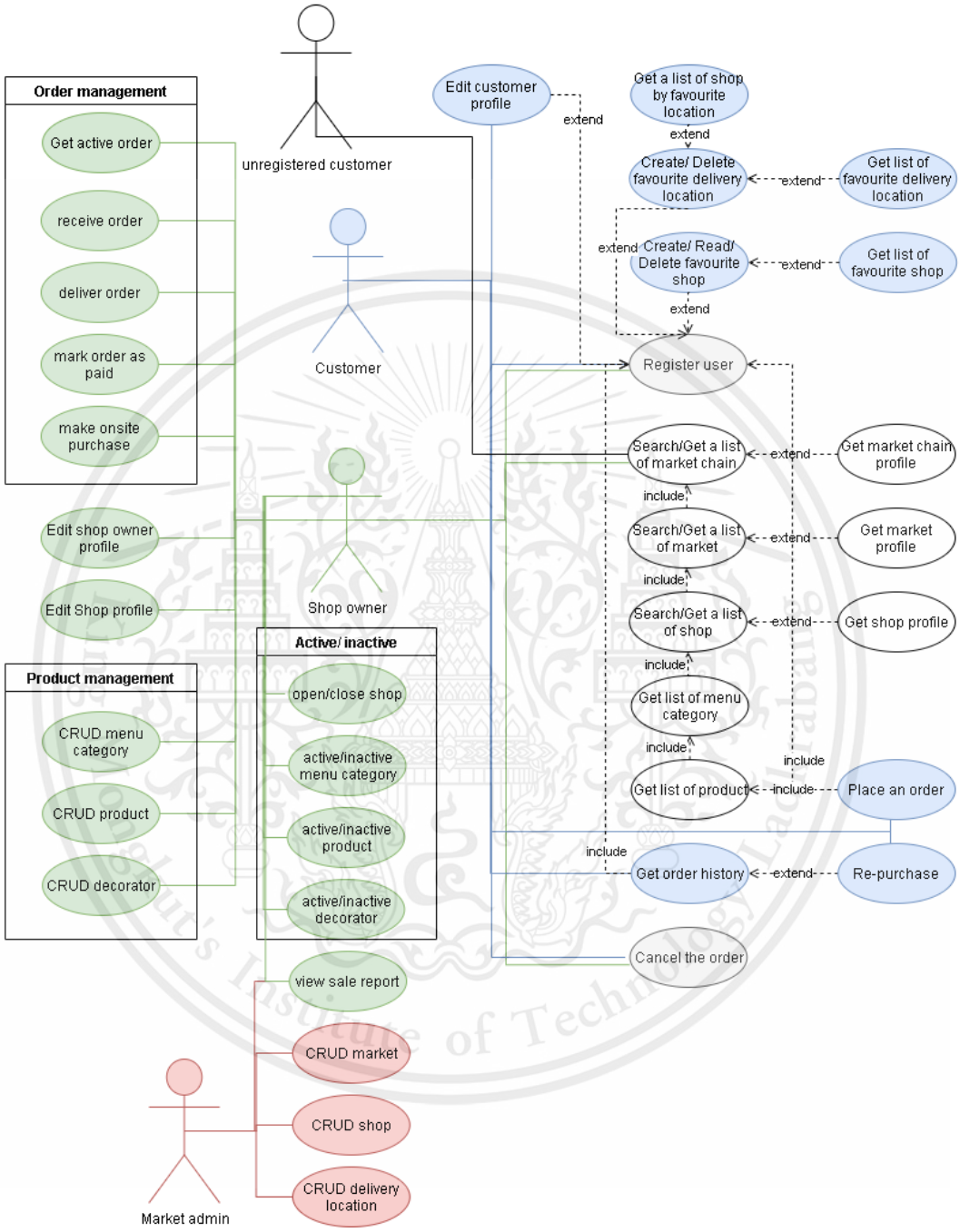

Figure 3.1: Use case diagram

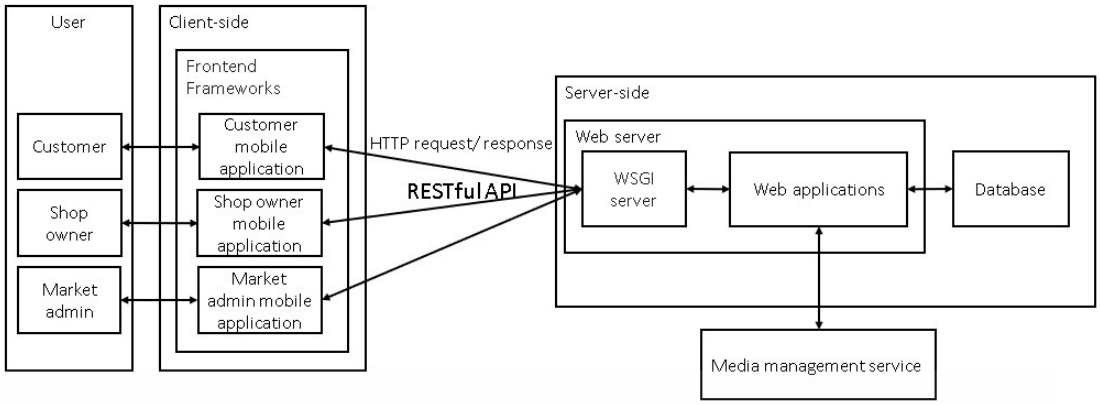

Figure 3.2: Architecture of the system

Client-side component

- Three applications for three user roles.
- Front-end framework developed using React native and JavaScript

Server-side component

- Web server gateway interface (WSGI): Gunicorn
- Web application is developed using Django and Django REST framework.
- Web server is running on Heroku cloud service.
- Database: PostgreSQL server running on AWS cloud service.
- Media management service: Cloudinary.

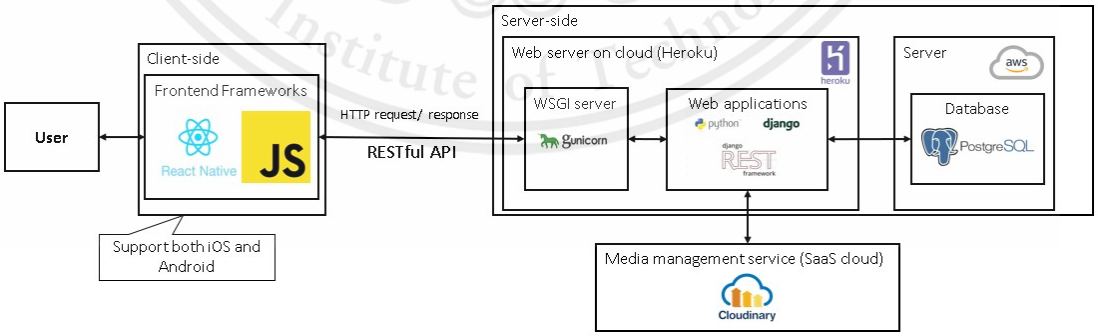

Figure 3.3: Architecture of the system with frameworks and tools

3.5 Software architecture

The system is developed using Django to provide API service for the client-side, which is mobile applications. The system composes of 4 main components as displayed in the figure 3.4: URL Pattern, View, Serializer, and Model. The URL pattern is the part that checks the incoming HTTP request and routes that request to the related view. The view is the part that prepares the response according to the request. Serializer is the part that converts JSON data format receiving from the view into ORM that models can use, and also converts ORM back to JSON data format for the view. Lastly, the model is an object that can communicate to the database.

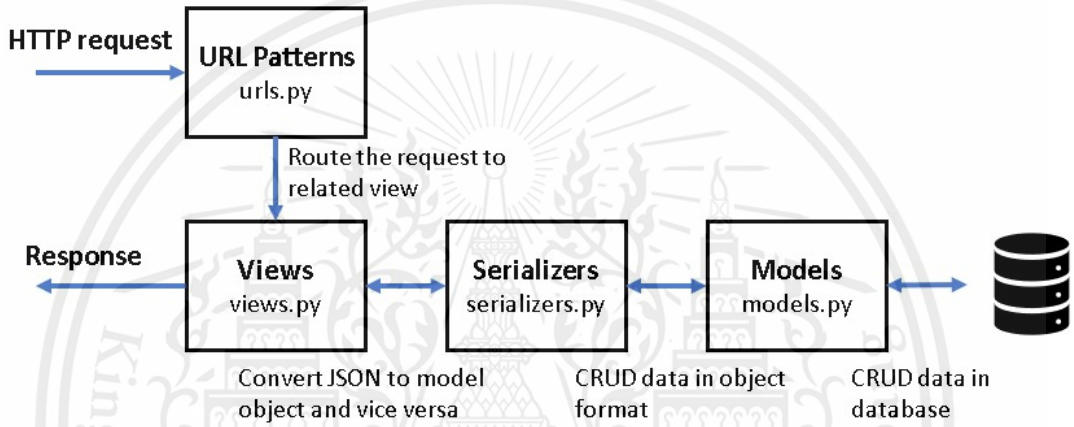

Figure 3.4: Django REST framework software architecture

3.6 Model diagram

The following figure 3.5 refers to the UML diagram of this project models. There are 3 user roles: Customer, Shop owner, and Market admin. The action of each type of user will be explained in the API part. The system is divided into four main services:

- accounts
- markets
- shops
- orders

The accounts service is the service that mainly manages user accounts. According to the diagram in figure 3.6 it composes of 4 classes:

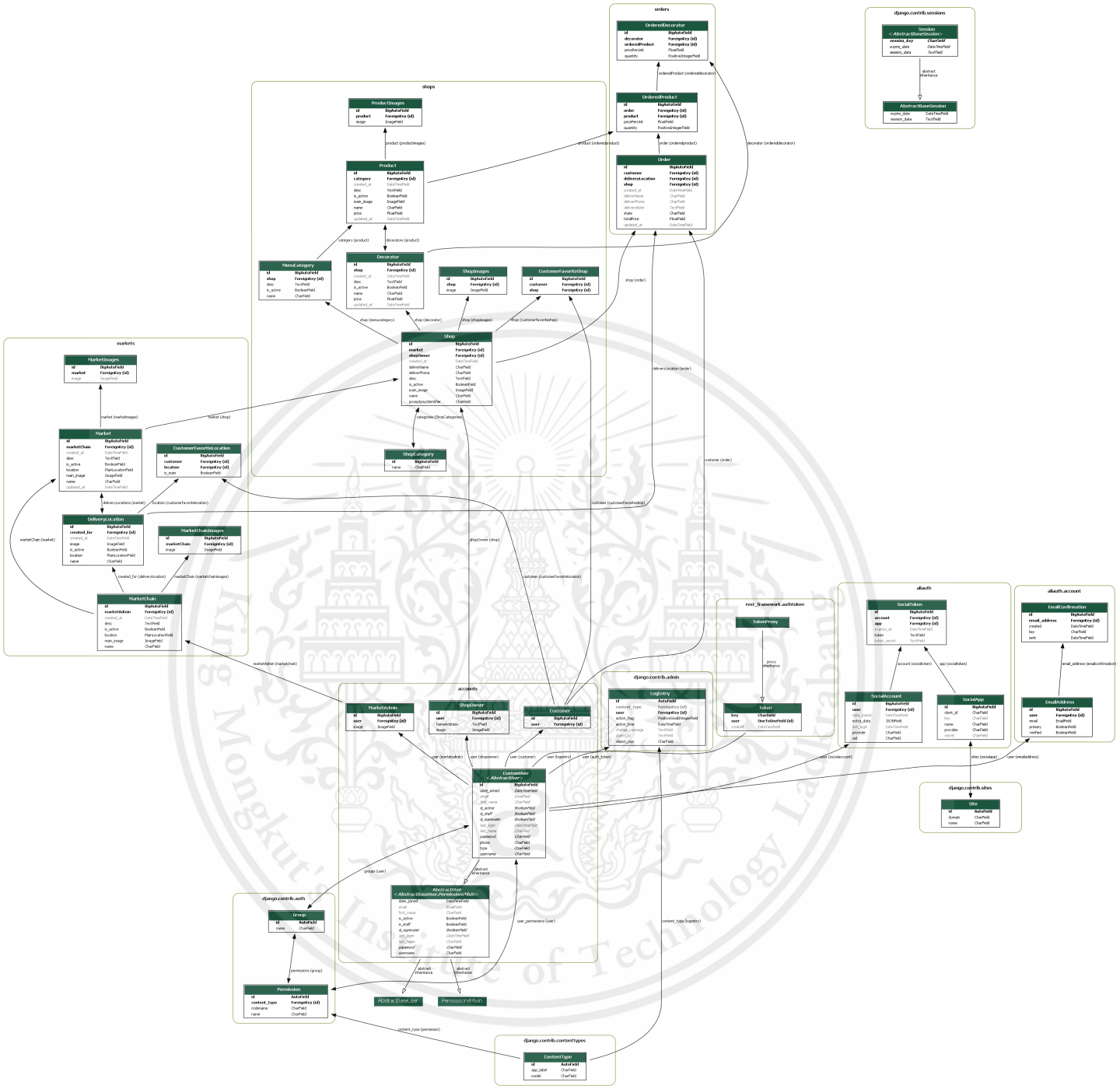

Figure 3.5: Django models diagram

- CustomUser which is the base class of all user role.
- Customer
- ShopOwner which is the account of the owner of the shop.
- MarketAdmin which is the account of the staff assigned by the organization such as university that can manage the market chain and market.

The markets service is the service that manages market chains and market. The following diagram in figure3.7 compose of 6 classes.

- Market chain which simulate the entire university that has many canteens scattered around the university.
- Market which simulate the canteen that has many shops inside.
- MarketChainImages which is the additional class that stores additional images of market chain. Due to the fact this application use SQL database as its database, storing objects or images as an array in the other model is impossible.
- MarketImages similar to the MarketChainImages, it stores additional images of market.
- DeliveryLocation is a class that stores a registered location for delivery that will be created by market admin and assigned the delivery to each market. For example the A canteen is a market of KMITL market chain, A canteen has 3 delivery locations: building A, B, and C. Any building excluded is considered as outside the service area.
- CustomerFavouriteLocation is a delivery location that is registered by customer as their favourite location.

Next, Shops, it is the service that manages shops and everything being sold by that shop. According to the figure 3.8 there are 8 classes:

- Shop, the most important class in this service, it is the model of the shop in the market.
- Shop category, is the category of that shop, each shop can have many categories. Example of the shop categories, western cuisine, Chinese cuisine, steak, and et cetera. If a shop sell steak it could be labeled as both western cuisine, and steak.

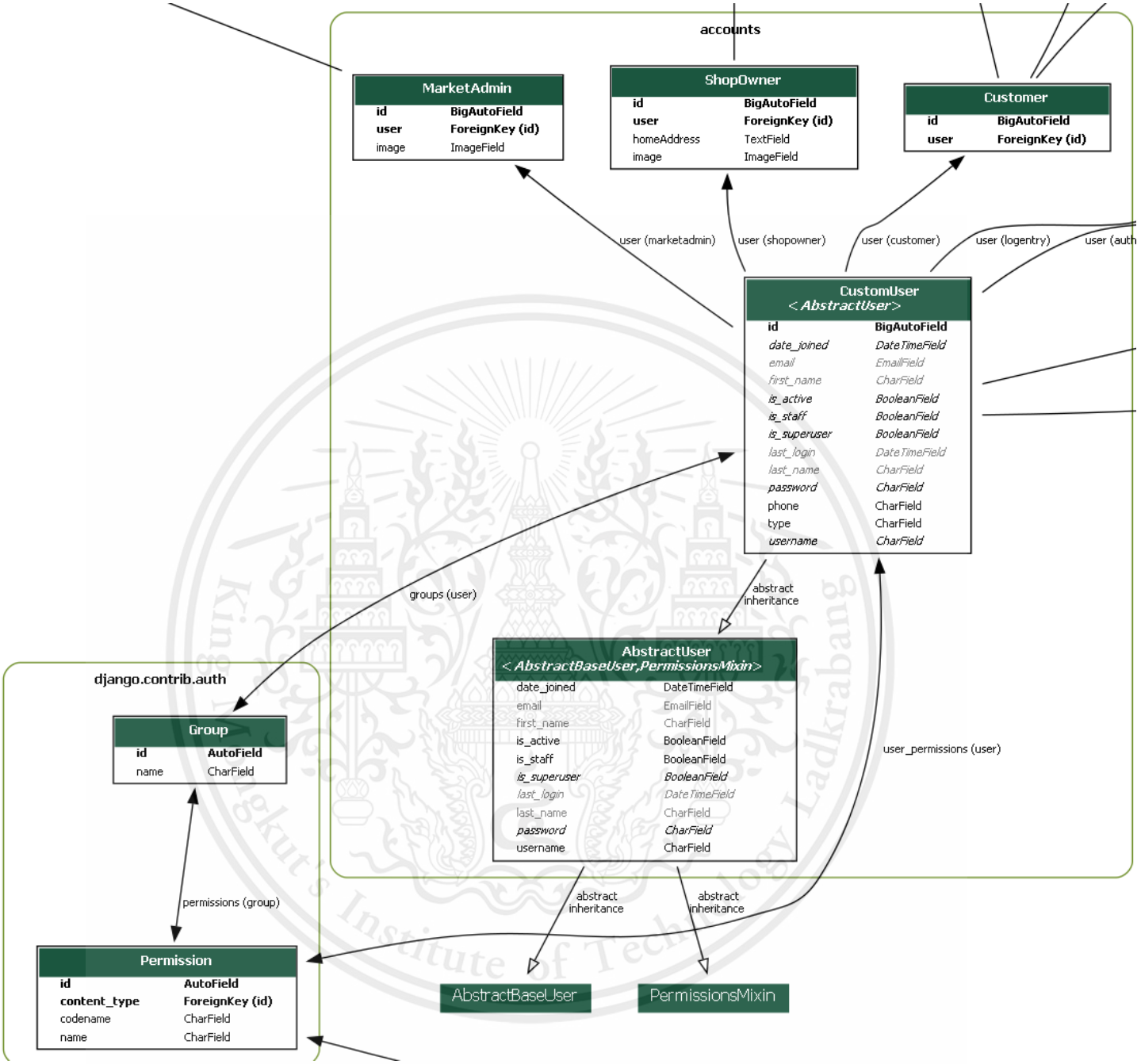

Figure 3.6: Model diagram of accounts service

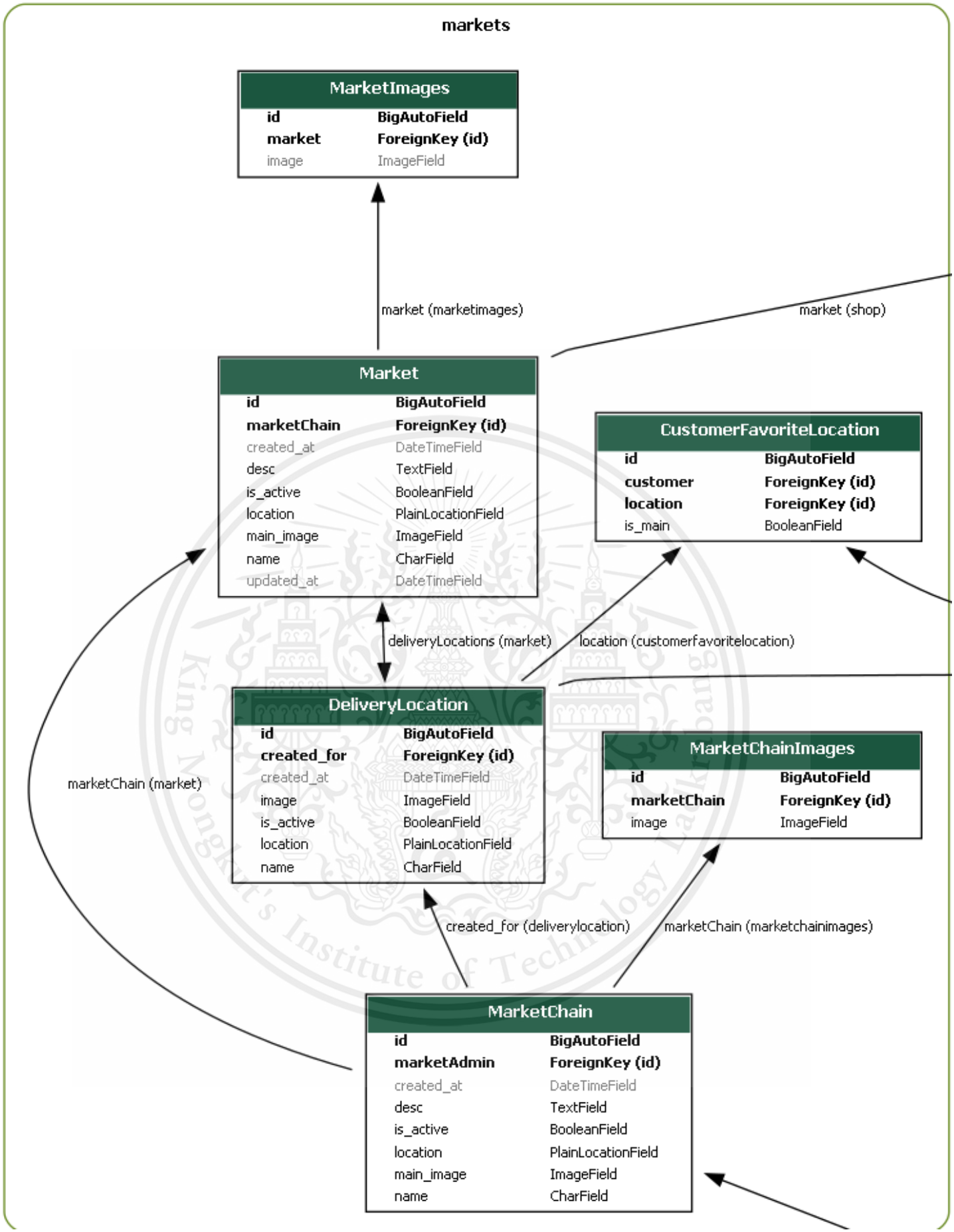

Figure 3.7: Model diagram of markets service

- ShopImages, similar to the MarketChainImages and MarketImages, it stores the additional images of the shop.
- MenuCategory, is the category of product sell in that shop. For example, main dish, appetizer, and snack.
- Product, is the product being sold in the shop. It needs to be registered to a Menu-Category.
- ProductImages, additional images of the product.
- Decorator, is the decorator that can be added to the product. For example add more spicy decorator can be added to a noodle or add more cheese decorator can be added to the burger. Which each shop has their own decorators.
- CustomerFavouriteShop, the customer can register a shop as their favourite shop.

The last service, orders, which is the service that manage the ordered orders. According to the figure3.9 there are 3 classes.

- Order, is the order that the customer order. The order has 4 status: cancel, ordered, received, delivered, and paid.
- OrderedProduct, the product that is ordered.
- OrderedDecorator, the decorator that is ordered.

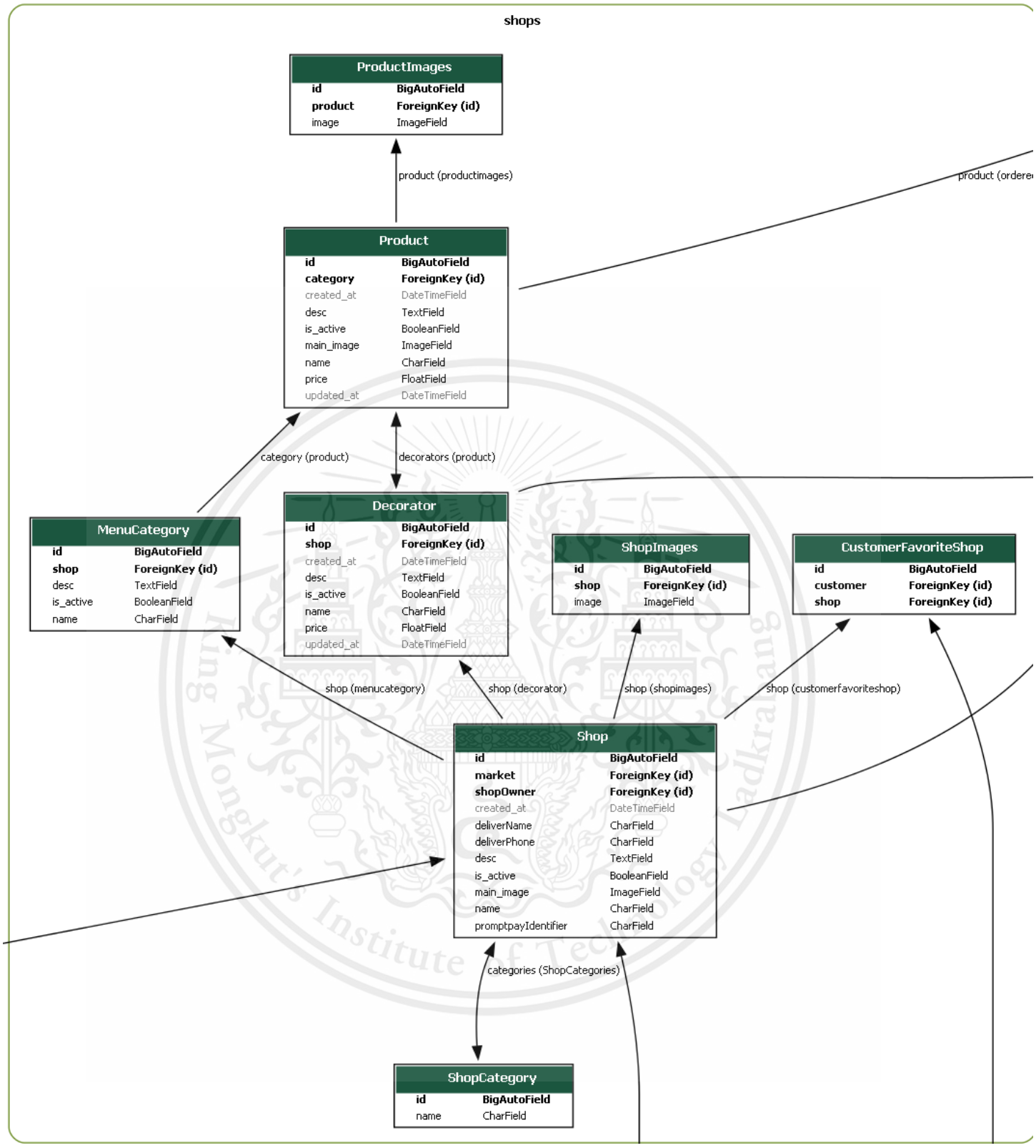

Figure 3.8: Model diagram of shops service

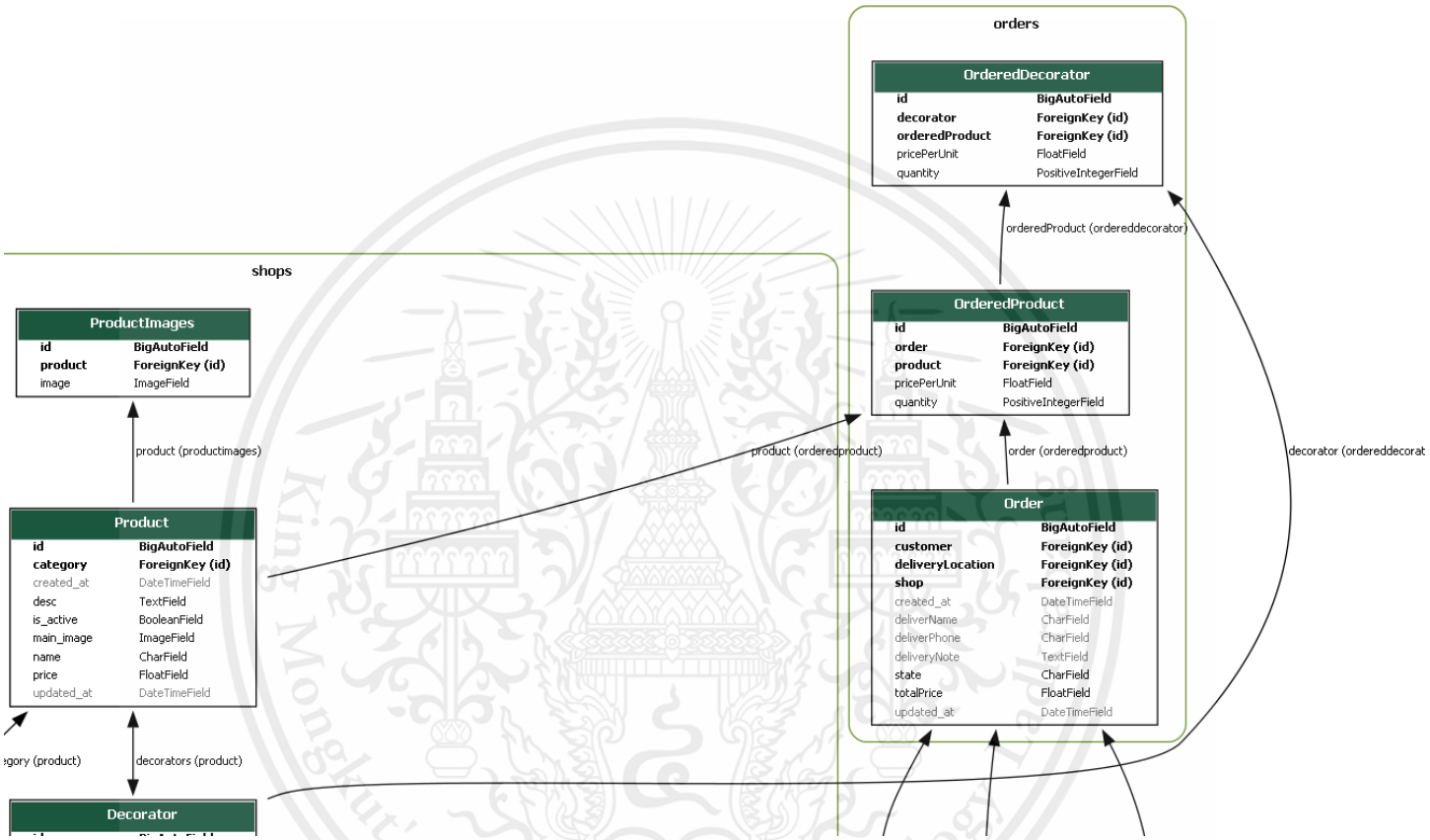

Figure 3.9: Model diagram of orders service

Chapter 4
Development

4.1 Development tools

The tools being used to develop this project which composes of code editor, database, computer languages and frameworks.

4.1.1 Code editor

Visual Studio Code

Visual studio code is a text editor program that is used in both our front-end and back-end project development. It has support for Javascript which could be used for web development in React and greatly support Python which is used to create a server application.

4.1.2 Database

PostgreSQL

PostgreSQL[8] is an ideal relational database that is needed to store data. PostgreSQL[8] uses a SQL database which is compatible with Cesium.JS allowing seamless supportability and flexibility when receiving and storing data. PostgreSQL[8] is also ideal as it could operate and process databases in real time. It also is very stable and could operate continuously at all time reducing crash and unnecessary error.

PostgreSQL[8] is a general purpose and object-relational database management system, the most advanced open source database system. PostgreSQL[8] was designed to run on UNIX-like platforms. However, PostgreSQL[8] was then also designed to be portable so that it could run on various platforms such as Mac OS X, Solaris, and Windows.

PostgreSQL[8] is the first database management system that implements multi-version concurrency control (MVCC) feature, even before Oracle. The MVCC feature is known as snapshot isolation in Oracle.

PostgreSQL[8] is a general-purpose object-relational database management system. It allows you to add custom functions developed using different programming languages such as C/C++, Java, etc.

PostgreSQL[8] is designed to be extensible. In PostgreSQL[8], you can define your own data types, index types, functional languages, etc. If you don't like any part of the system, you can always develop a custom plugin to enhance it to meet your requirements e.g., adding a new optimizer.

Full Text Searching and Elasticsearch

Full Text Searching, commonly known as text search, is an operator that existed in databases such as PostgreSQL that we are utilizing. It allow users the capability to identify language documents that satisfy a query, and sort them by relevance to the query if need be. One of the most common use of Full Text Searching is to find any and all documents that contain a given query terms, then return them in order of similarity to the query.

Elasticsearch on the other hand, is a search and analytic engine for all kinds of data such as textual, numerical, geospatial, structured, and unstructured. It has the advantages of being simple to utilize its REST APIs, relatively good speed, its distributed nature, and its scalability.

With this knowledge, these solutions could be useful in our system, utilizing them in manner such as implementing Elasticsearch for application search, logging data and performing log analysis, and business analytic, if need be.

4.1.3 Computer language

React (JavaScript Library) and Javascript

First appearing in 1995, Javascript[6] is as of today, the most popular programming language for web development. (As evidenced by the fact that over 94 percent of all websites are utilizing it). JavaScript is a client-side programming language that allows web developers to do Web Application Development and make dynamic and interactive web pages by implementing custom client-side scripts.

Developers can also utilize cross platform runtime engines like Node.js to write server-side code in JavaScript[6]. Developers can also create web pages which work

well across various browsers, platforms, and devices by combining JavaScript, HTML5, and CSS3. Javascript[6] also supports multiple Frameworks such as AngularJS, ReactJS, and NodeJS. These frameworks can help reduce the amount of time and effort required for developing JS based sites and apps.

Our framework choice is React[14], a JavaScript Library that we use to build a fronted web page. React allows fast updating and rendering of ideal components when data changes. React is also ideal since it could render on the server using Node.

Python

Python[15] is a multi-paradigm programming language supporting object-oriented programming and structured programming. It uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management.

In this project Python3[15] will be used to create a server software together with Django framework.

4.1.4 Framework

Django framework

The Django framework is used as a backend of our web development. We used the Django framework to communicate between React and postgreSQL.

During web development, it is crucial to have a similar set of components. Such as different ways to deal with user authentication, a management panel for your website, forms and a way to upload files. Frameworks is ideal for saving you from having to reinvent the wheel and to help alleviate some of the overhead when you're building a new site. Moreover, Django provides Django REST Framework[1] which supporting RESTful API[10].

React Native

React Native[16] or RN is a Javascript-based mobile app framework developed and released by Facebook. RN allows developer to built a mobile application for both ios and android with the same codebase, hugely reducing both resources and time use in the development process.

Moreover, because RN was built based on React – a JavaScript library, which was already hugely popular when the mobile framework was released. Many developer with a prior experience working with react could easily used the framework without the need to learn a new language or code.

4.2 Development process

This section describes the development process of our project. The software development will divide into 2 parts: client-side and server-side.

4.2.1 Client-side development

User interface design changes

From previous semester, there have been changes made to the design of the application itself, based on our advisor and other professor's criticism. Some of these include adding the menu decorator function (for menu customisation) and removing point of sale system, which was deemed redundant by some of the professor committee.

We visualised the changes by using Figma collaborative design tool to edit the designs of customer and shop owner's user interface accordingly. Then, we continue by translating them into a proper React Native (Javascript) interfaces.

React Native and dependencies utilized

```
"dependencies": {
  "@react-native-community/masked-view": "0.1.10",
  "@react-navigation/material-bottom-tabs": "^5.3.10",
  "@react-navigation/native": "^5.8.10",
  "FormData": "^0.10.1",
  "axios": "^0.21.1",
  "expo": "~40.0.0",
  "expo-status-bar": "~1.0.3",
  "js-cookie": "^2.2.1",
  "react": "16.13.1",
  "react-dom": "16.13.1",
  "react-native": "https://github.com/expo/react-native/archive/sdk-40.0.1.tar.gz",
  "react-native-gesture-handler": "~1.8.0",
  "react-native-paper": "^4.5.0",
  "react-native-reanimated": "~1.13.0",
  "react-native-safe-area-context": "3.1.9",
  "react-native-screens": "~2.15.0",
  "react-native-web": "~0.13.12",
  "react-navigation": "^4.4.4",
  "react-navigation-material-bottom-tabs": "^2.3.3",
  "react-navigation-stack": "^2.10.2",
  "react-navigation-tabs": "^2.10.1",
  "react-router-dom": "^5.2.0"
},
```

Figure 4.1: List of dependencies used by client-side

With a framework such as react native, there are many selection of tools to be utilized, from both first and third parties. These tools can perform many different tasks

such as assisting in navigation between pages, allow fetching of data from different API, or creating a nice looking bottom navigation tab. Overall, different dependencies allow developers to have more options and flexibility to do what is required.

The figure above shows the dependencies that is used in our application. Some of which will be discussed subsequently.

React Native fetch API request service, useState, useEffect

```
141  const LoginScreen = ({navigation}) => {
142
143    const [password, setPassword] = useState("");
144    const [username, setUsername] = useState("");
145
146    const handleResponse = res => {
147      if(res.ok) {
148        navigation.navigate('Canteen')
149      }
150      throw new Error('Network response was not ok.')
151    }
152
153    const apicall = (username, password) => {
154
155      const apiUrl = 'https://nearme-kmitl.herokuapp.com/api/dj-rest-auth/login/';
156
157      fetch(apiUrl, {
158        method: 'post',
159        headers: {
160          'Accept': 'application/json',
161          'Content-Type': 'application/json',
162        },
163        body: JSON.stringify(
164          {
165            username: username,
166            password: password,
167          }
168        )
169      }).then(handleResponse)
170      .then(data => console.log(data))
171      .catch(error => console.log("Error detected: " + error))
172    }
```

Figure 4.2: Demonstration of fetching

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides a way to fetch resources from network. (from methods such as GET, POST, PUT, DELETE).

Utilizing this with effect hook (useState, useEffect) and react state, we can interact with the postgresql database to complete CRUD tasks related to our application

in each screen.

The figure above demonstrates using fetching API to handle a login screen, where it can confirm if the username and password entered matches each other, and are valid or not.

React Native RefreshControl

```
14  const CanteenScreen = ({navigation}) => {
15
16    const apiUrl = window.apiurl + 'api/markets/markets/marketchain/1/';
17
18    const [Canteens, setCanteens] = useState([]);
19    const [Search, setSearch] = useState([]);
20    const [refreshing, setRefreshing] = useState(false);
21
22    const onRefresh = React.useCallback(async () => {
23      setRefreshing(true);
24      loadData();
25      setRefreshing(false)
26    }, [refreshing]);
27
28    useEffect(() => {
29      loadData();
30    }, [Search]);
31
32    const loadData = async () => {
33      if (Search.length == 0) {
34        const response = await fetch(apiUrl);
35        const data = await response.json();
36        setCanteens(data);
37        console.log(data);
38      } else {
39        const response = await fetch(window.apiurl + 'api/markets/markets/search/?search=' + Search);
40        const data = await response.json();
41        setCanteens(data);
42        console.log(data);
43      }
44    }
```

```
111           showsVerticalScrollIndicator={false}
112           numColumns={1}
113           data={Canteens}
114           renderItem={({item}) => <MenuDetail canteen={item} />}
115           refreshControl={<RefreshControl refreshing={refreshing} onRefresh={onRefresh} />}
116         />
```

Figure 4.3: Demonstration of RefreshControl

The Refresh control function is used so that a user can implement a Pull to refresh functionality for their React Native apps. When this happens, new data is loaded and replaces the old, yet to be updated data.

The figure above demonstrates the use of refresh control, implemented to a market/canteen selection page, where if a pull-gesture is done to the screen, it will activate

the refresh control, re-rendering the screen and displays new data if something new is posted to the list of available canteen.

4.2.2 Server-side development

The server-side development, is separated into 2 part: deployment and web application which is Django application.

Deployment

Only the server-side web application is deployed and run on the server since the applications running on the client-side are mobile applications that are not required to be deployed. The server is deployed and running on the Heroku cloud platform. Inside the web server, there are 2 major components: web application which will be explained in the following subsection, and web server gateway interface (WSGI)

WSGI is a part that connect the web application to the internet. The configuration of the WSGI is in the figure 4.4.

nearme > nearme > wsgi.py

```
"""
WSGI config for nearme project.

It exposes the WSGI callable as a module-level variable named ``application``

For more information on this file, see
https://docs.djangoproject.com/en/3.2/howto/deployment/wsgi/
"""

import os
from dj_static import Cling
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'nearme.settings')

application = Cling(get_wsgi_application())
```

Figure 4.4: Web server gateway interface configuration

After the web application is deployed on the Heroku platform, to make it know how to run the web application on the server, Procfile is required. Procfile is a file that records necessary command scripts to run the application. According to figure 4.5 the web application will run through WSGI, and before release the application, it needs to migrate the application.

nearme > Procfile

```
1   web: gunicorn nearme.wsgi --log-file -
2   release: python manage.py migrate
```

Figure 4.5: Procfile

The next thing to be concerned about is the database. Running web application on the cloud need to have a database that connects to that application. In our project, we choose PostgreSQL as our database, fortunately, Herkou provides the Heroku Postgres as an add-on. After studying more about it, we found out that it is actually hosted on AWS. To config the database in the Django setting. We let the Django web application look up the database information in the Heroku config vars which is designed to be safe for storing sensitive information as the figure 4.6.

```
186   # Heroku Dabase Config
187
188   DATABASES = {
189       'default': dj_database_url.config(conn_max_age=600, ssl_require=True)
190   }
```

Figure 4.6: Database configuration in Django setting

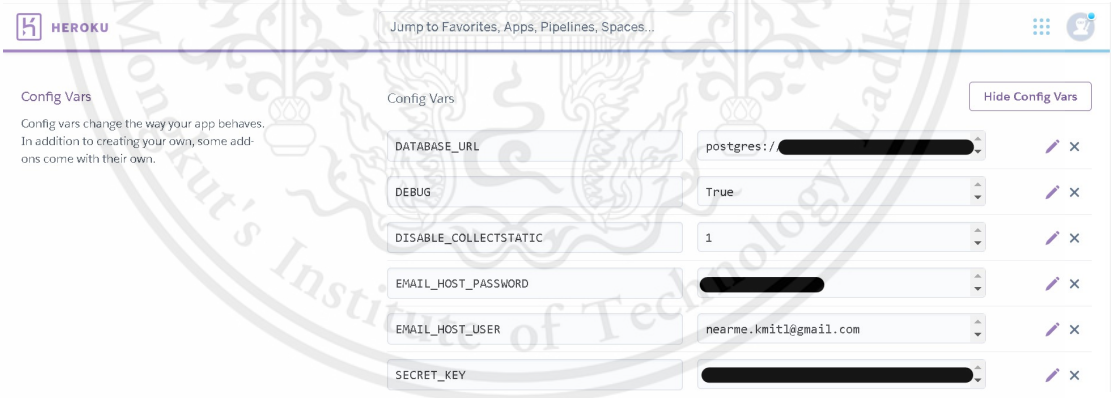

Figure 4.7: Heroku config vars

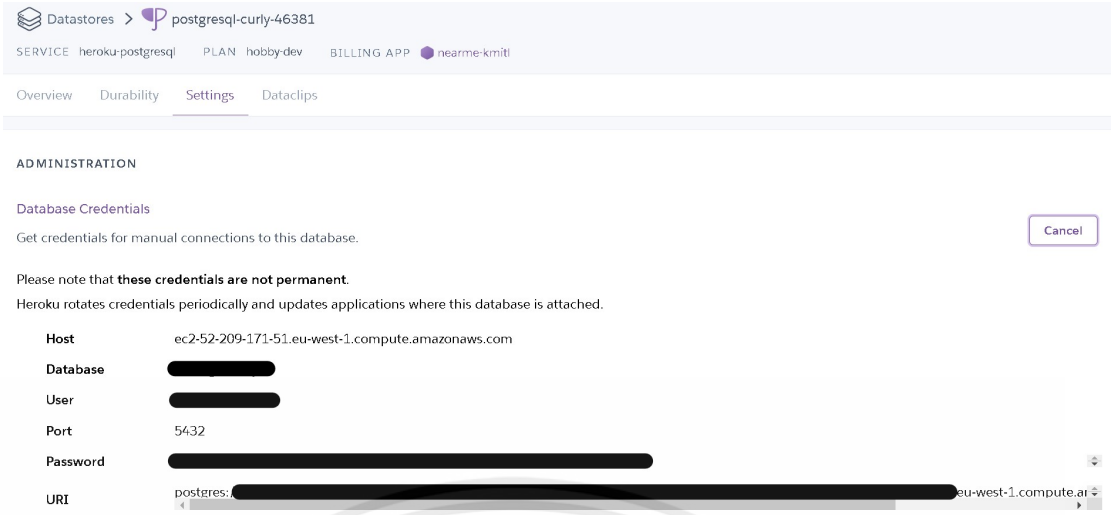

Figure 4.8: Database

After dealing with the data, the next thing to be concern about is the media file. In our case, it is images. In normal practice, the server will store media in static storage, however as its name suggests, it is static; once something wrong with the server, all static files will be gone. Therefore, we decided to move all images getting into our server to another place that is well managed. We choose Cloudinary since it is well known, supports all types of media files, provides API service, provides a library that supports integration, and lastly, it is free of charge.

```
246  # Static files (CSS, JavaScript, Images)
247  # https://docs.djangoproject.com/en/3.2/howto/static-files/
248
249  STATIC_URL = '/static/'
250  STATIC_ROOT = os.path.join(BASE_DIR, 'static')
251  STATICFILES_DIRS = (os.path.join(BASE_DIR, 'nearme/static'),)
252
253  MEDIA_URL = '/media/'
254  MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
255
256  CLOUDINARY_STORAGE = {
257      'CLOUD_NAME': 'nearme-kmitl',
258      'API_KEY': '264357333432383',
259      'API_SECRET': 'KfGrnYRZgXEBxn0MvXzrHvn5K3Q',
260  }
261  DEFAULT_FILE_STORAGE = 'cloudinary_storage.storage.MediaCloudinaryStorage'
```

Figure 4.9: Cloudinary configuration in Django setting

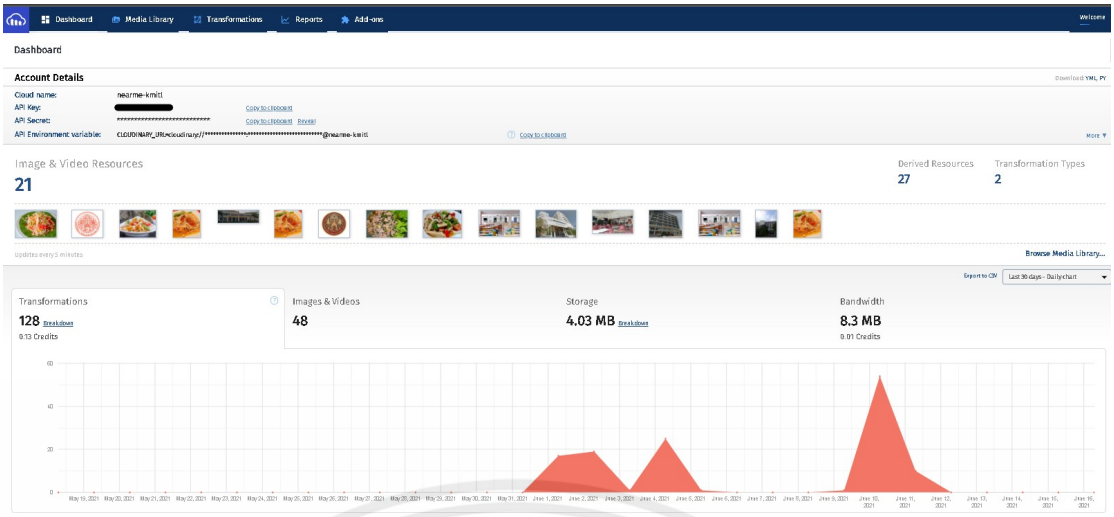

Figure 4.10: Cloudinary configuration in Django setting

Next, the mail service, to enable the forgot password feature that the server will send a link with user id and token to the user email as the figure 4.12. Mail service is required. In our project, we decided to use Gmail as our web site email. We connect the API that provided by Gmail to our Django as it appears in the figure 4.11. Then storing email and password in the Heroku config var for the security purpose in the figure 4.7.

```
84    # EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
85
86    EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
87    EMAIL_HOST = 'smtp.gmail.com'
88    EMAIL_USE_TLS = True
89    EMAIL_PORT = 587
90    EMAIL_HOST_USER = config('EMAIL_HOST_USER')
91    EMAIL_HOST_PASSWORD = config('EMAIL_HOST_PASSWORD')
92
93    DEFAULT_FROM_EMAIL = 'Nearme service Team <noreply@nearme.com>'
94    SITE_ID = 1
```

Figure 4.11: Email configuration in Django setting

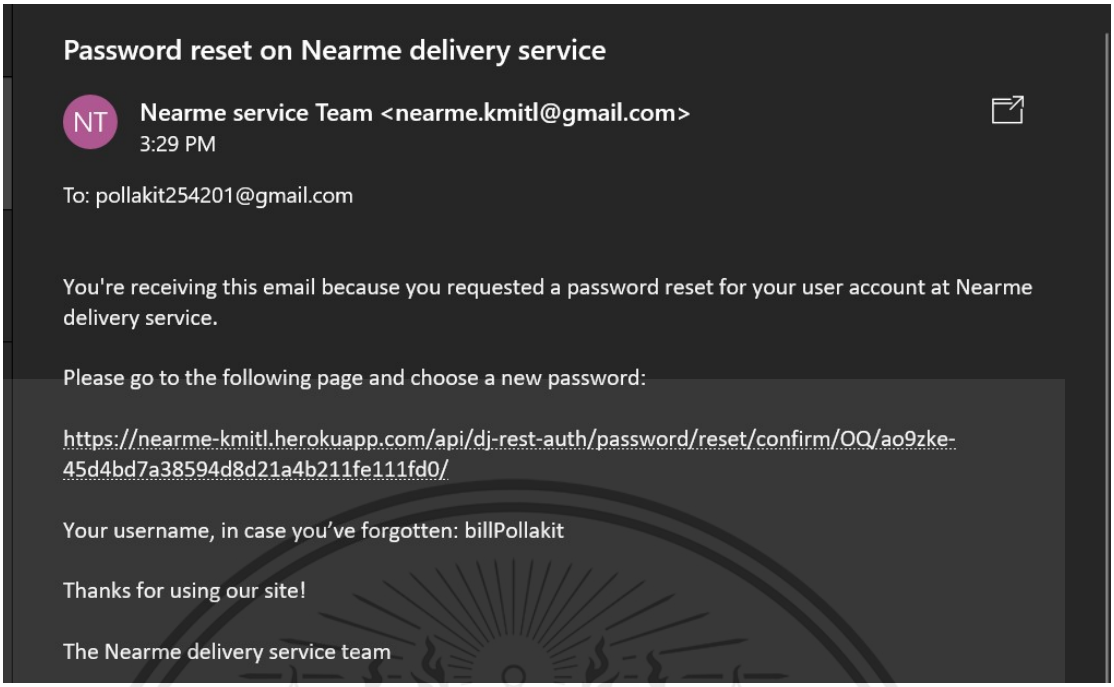

Figure 4.12: Password reset email

Server-side web application

For the web application, there are 2 main parts: models and API framework. The models were explained in the previous chapter. In this section, we will discuss on API framework that sends API to the client-side.

Before going to the API framework, since our web application runs on the Django framework, to make it function with REST, external libraries and frameworks are required, as it appears in figure4.13. For example, the Django REST framework enables REST API, Django-rest-auth, and allauth to facilitate authentication, et cetera.

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',

    # 3rd party
    'django_extensions',  # For diagram
    'cloudinary_storage',  # Cloudinary ( For images )
    'requests',
    'location_field.apps.DefaultConfig',
    'rest_framework',
    'django_filters',
    'corsheaders',
    'crc16',  # for promptpay

    'rest_framework.authtoken',  # For Authen

    'allauth',
    'allauth.account',
    'allauth.socialaccount',

    'dj_rest_auth',
    'dj_rest_auth.registration',

    # Heroku
    'whitenoise.runserver_nostatic',
    # Cloudinary ( For images )
    'cloudinary',

    # Local
    'accounts.apps.AccountsConfig',
    'markets.apps.MarketsConfig',
    'orders.apps.OrdersConfig',
    'shops.apps.ShopsConfig',
]
```

Figure 4.13: Installed apps and third-party libraries

After the libraries and dependencies are installed. The following things we need is URLs. URLs is a Python code that creates API endpoints to route the request to the

correct view. However, since we separate our web application into 4 services: accounts, markets, orders, and shops, we have 4 ”urls.py” for each service plus one extra ”urls.py” for the project.

Firstly, the main URLs consists of the Django administration terminal, user authentication, APIs documentation, and the rest 4 services URLs as it appears in figure 4.14.

nearme > nearme > urls.py > ...

```
27  urlpatterns = [
28      path('admin/', admin.site.urls),
29      path('accounts/', include('allauth.urls')),
30      # path('api-auth/', include('rest_framework.urls')),
31      path('api/dj-rest-auth/', include('dj_rest_auth.urls')),
32      path('api/dj-rest-auth/registration/',
33           include('dj_rest_auth.registration.urls')),
34      path(
35          'api/dj-rest-auth/registration/account-confirm-email/<str:key>/', confirm_email),
36      path('api/dj-rest-auth/password/reset/confirm/<slug:uidb64>/<slug:token>/',
37           PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
38
39      path('docs/', include_docs_urls(title='My API title')),
40
41      path('api/accounts/', include('accounts.urls')),
42      path('api/markets/', include('markets.urls')),
43      path('api/orders/', include('orders.urls')),
44      path('api/shops/', include('shops.urls')),
45
46  ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Figure 4.14: The main urls.py

Next, URLs of the account service can be separated into 3 main categories: Customer, Shop owner, and market admin.

nearme > accounts > urls.py > ...

```
21
22  urlpatterns = [
23      path('customers/', CustomerList.as_view()),  # GET POST
24      path('customers/<int:pk>/', CustomerDetail.as_view()),  # GET PATCH
25      path('customers/user/<int:pk>/', UserViewCustomerDetail.as_view()),  # GET
26
27      path('shopowners/', ShopOwnerList.as_view()),  # GET POST
28      path('shopowners/<int:pk>/', ShopOwnerDetail.as_view()),  # GET PATCH
29      path('shopowners/user/<int:pk>/', UserViewShopOwnerDetail.as_view()),  # GET
30
31      path('marketadmins/', MarketAdminList.as_view()),  # GET
32      path('marketadmins/<int:pk>/', MarketAdminDetail.as_view()),  # GET
33      path('marketadmins/user/<int:pk>/',
34           UserViewMarketAdminDetail.as_view()),  # GET
35
36  ]
```

Figure 4.15: URLs of the account service

Next, URLs of the markets service can be separated into 4 main categories: market chain which for example KMITL, market, favourite location of a customer, and delivery location.

nearme > markets > urls.py > ...

```python
urlpatterns = [
    path('marketchains/', MarketChainList.as_view()),                       # GET
    path('marketchains/search/', MarketChainSearch.as_view()),              # GET
    path('marketchains/<int:pk>/', MarketChainDetail.as_view()),            # GET
    path('openclosemarketchain/<int:pk>/', OpenCloseMarketChain.as_view()),# GET PATCH

    path('markets/', MarketList.as_view()),                                 # GET POST
    path('markets/search/', MarketSearch.as_view()),                        # GET
    path('markets/deliveryLocation/<int:pk>/',
        MarketListByDeliveryLocation.as_view()),                            # GET
    path('markets/marketchain/<int:pk>/', MarketListByMarketChain.as_view()),# GET
    path('markets/<int:pk>/', MarketDetail.as_view()),                      # GET PATCH DELETE
    path('openclosemarket/<int:pk>/', OpenCloseMarket.as_view()),           # GET PATCH


    path('favoriteLocation/', FavouriteLocationList.as_view()),             # GET POST
    path('favoriteLocation/<int:pk>/',
        FavouriteLocationDetail.as_view()),                                 # GET PATCH DELETE

    path('favoriteLocation/customer/<int:pk>/',
        CustomerFavouriteLocationDetail.as_view()),                         # GET

    path('deliverylocations/', DeliveryLocationList.as_view()),             # GET
    path('deliverylocations/marketchain/<int:pk>/',
        DeliveryLocationListByMarketChain.as_view()),                       # GET
    path('deliverylocations/<int:pk>/', DeliveryLocationDetail.as_view()), # GET PATCH DELETE
]
```

Figure 4.16: URLs of the markets service

Next, URLs of the order service can be separated into 3 main categories: the order, product in the order, and status of the order. The order has 4 status. First when the customer create an order, that order will have the status ”ordered”. Later on, the status can be changed into ”cancel”, ”received”, ”delivered”, and ”paid”.

nearme > orders > urls.py > ...

```python
urlpatterns = [
    path('orders/', OrderList.as_view()),                           # GET POST
    path('products/', OrderedProductList.as_view()),                # GET POST

    path('orders/<int:pk>/', OrderDetail.as_view()),                # GET
    path('state/<int:pk>/', UpdateOrderStateView.as_view()),        # PATCH

    # Record
    path('orders/customer/<int:pk>/', CustomerOrderHistory.as_view()),  # GET
    path('orders/shop/<int:pk>/', ShopSellHistory.as_view()),       # GET

    path('uncompleteOrder/shop/<int:pk>/', UncompletedOrder.as_view()),  # GET
]
```

Figure 4.17: URLs of the orders service

Lastly, URLs of the order service can be separated into 6 main categories: shop which contains shop profile, shop categories which is the category of the shop for example, a western cuisine restaurant, menu category which is the category of the menu possess by a shop for example main dish and appetizer, product, decorator which is the decorator of a product for example adding cheese on a burger, and lastly favourite shop.

nearme > shops > urls.py > ...

```python
urlpatterns = [
    # Shop
    path('shops/', ShopList.as_view()),                                  # GET POST
    path('shops/market/<int:pk>/', ShopListByMarket.as_view()),  # GET
    path('shops/<int:pk>/', ShopDetail.as_view()),                       # GET PATCH DELETE
    path('shops/market/<int:pk>/search/', ShopSearch.as_view()), # GET

    # ShopCategory
    path('shopcategories/', ShopCategoryList.as_view()),                 # GET POST
    path('shopcategories/<int:pk>/', ShopCategoryDetail.as_view()),# GET PATCH DELETE
    path('shopcategories/search/', ShopCategorySearch.as_view()),# GET


    # MenuCategory
    path('menucategories/', MenuCategoryList.as_view()),                 # GET POST
    path('menucategories/shop/<int:pk>/', MenuCategoryListByShop.as_view()), # GET
    path('menucategories/<int:pk>/', MenuCategoryDetail.as_view()),# GET PATCH DELETE
    path('menucategories/customer/shop/<int:pk>/',
        CustomerViewMenuCategoryListByShop.as_view()),                   # GET

    # Product
    path('products/', ProductList.as_view()),                            # GET POST
    path('products/menucategory/<int:pk>/',                              # GET
        ProductListByMenuCategory.as_view()),
    path('products/customer/menucategory/<int:pk>/',                     # GET
        CustomerViewProductListByMenuCategory.as_view()),

    path('products/market/<int:pk>/search/', ProductSearch.as_view()),# GET
    path('products/<int:pk>/', ProductDetail.as_view()),                 # GET PATCH DELETE

    # Decorator
    path('decorators/', DecoratorList.as_view()),                        # GET POST
    path('decorators/shop/<int:pk>/', DecoratorListByShop.as_view()),# GET POST
    path('decorators/product/<int:pk>/', DecoratorListByProduct.as_view()),# GET POST
    path('decorators/<int:pk>/', DecoratorDetail.as_view()),             # GET PATCH DELETE

    # Customer Favourite Shop
    path('favouriteShop/',                                               # GET POST
        CustomerFavoriteShopList.as_view()),
    path('favouriteShop/customer/<int:pk>/',                             # GET
        CustomerFavoriteShopByCustomer.as_view()),
    path('favouriteShop/<int:pk>/',                                      # GET DELETE
        CustomerFavoriteShopDelete.as_view()),
]
```

Figure 4.18: URLs of the shops service

After we have URLs, we need views that connect to URLs. However since there are too many views in the entire project. We will pick an example to show how does it work. The following view in the figure 4.19 is to retrieve customer profile from the given user id.

```
35  class UserViewCustomerDetail(generics.ListAPIView):
36      serializer_class = CustomerSerializer
37
38      def get_queryset(self):
39          userID = self.kwargs['pk']
40          return Customer.objects.filter(user__id=userID)
```

Figure 4.19: Customer view by user id

Since view requires the serializer to convert JSON format to ORM and vice versa. The serializer in the figure 4.20 is used to handle API request which is JSON format for the Customer view which is mentioned previously.

```
85  class CustomerSerializer(serializers.ModelSerializer):
86      delverylocation = serializers.SerializerMethodField(
87          "getMainDelveryLocation")
88
89      user = CustomUserDetailsSerializer(many=False)
90
91      class Meta:
92          model = Customer
93          fields = ['id', 'user', 'delverylocation']
94
95      def getMainDelveryLocation(self, obj):
96          return RoughFavoriteLocationSerializer(obj.customerfavoritelocation_set.filter(is_main=True), many=True).data
97
98      def update(self, instance, validated_data):
99
100         user_data = validated_data.pop('user')
101         if('type' in user_data.keys()):
102             user_data.pop('type')
103
104         if('password' in user_data.keys()):
105             user_data.pop('password')
106
107         user_data['type'] = "CUSTOMER"
108         user = instance.user
109         instance.save()
110
111         for attr, value in user_data.items():
112             if attr == 'password':
113                 user.set_password(value)
114             else:
115                 setattr(user, attr, value)
116
```

Figure 4.20: Customer serializer

For the API endpoints, they are separated into 5 parts: main, accounts, markets, orders, and shops. The list of API endpoints that is provided by the system is in the following tables.

Path	HTTP Method	Action
/admin/		Go to admin terminal
/docs/		Go to API documentation
/api/dj-rest-auth/login/	POST	Login to the system
/api/dj-rest-auth/logout/	POST	Logout and destroy the token of that user
/api/dj-rest-auth/password/change/	POST	Change the password
/api/dj-rest-auth/password/reset/	POST	Request reset the password and the reset password form will send to the given email.
/api/dj-rest-auth/password/reset/confirm/{uidb64}/{token}/	POST	Password reset e-mail link is confirmed, therefore this resets the user's password.
/api/dj-rest-auth/password/reset/confirm/	POST	Password reset e-mail link is confirmed, therefore this resets the user's password.
/api/dj-rest-auth/registration/	POST	Register a user to the system.

Figure 4.21: APIs of the main project

Path	HTTP Method	Action
/api/accounts/customers/	POST	Create a customer from registered user.
/api/accounts/customers/{id}/	PATCH	Update customer profile given customer id
/api/accounts/customers/user/{id}/	GET	Get customer profile from given user id
/api/accounts/shopowners/	POST	Create a shop owner from registered user.
/api/accounts/shopowners/{id}/	GET	Get shop owner profile from shop owner id
/api/accounts/shopowners/{id}/	PATCH	Update shop owner profile
/api/accounts/shopowners/user/{id}/	GET	Get shop owner profile from user id
/api/accounts/marketadmins/user/{id}/	GET	Get market admin profile from user id

Figure 4.22: APIs of the accounts service

Path	HTTP Method	Action
/api/markets/marketchains/	GET	List all market chain that is open in the system.
/api/markets/marketchains/{id}/	GET	Get market chain detail from given market chain id
/api/markets/marketchains/search/	GET	Search market chain by name and description
/api/markets/markets/	POST	Create a market
/api/markets/markets/marketchain/{id}/	GET	List all market that is open in the given market chain
/api/markets/markets/{id}/	GET	Get market detail
/api/markets/markets/{id}/	PATCH	Update market
/api/markets/markets/search/	GET	Search market by name and description
/api/markets/markets/{id}/	DELETE	Delete a market
/api/markets/deliverylocations/	POST	Create a delivery location
/api/markets/deliverylocations/{id}/	GET	Get delivery location detail
/api/markets/deliverylocations/{id}/	PATCH	Update delivery location detail
/api/markets/deliverylocations/{id}/	DELETE	Delete a delivery location
/api/markets/deliverylocations/marketchain/{id}/	GET	Get delivery location belong to the market chain of given id
/api/markets/favoriteLocation/	POST	Create a favorite location for a
/api/markets/favoriteLocation/{id}/	GET	Get favorite location detail
/api/markets/favoriteLocation/{id}/	PATCH	Update favorite location
/api/markets/favoriteLocation/{id}/	DELETE	Update a favorite location
/api/markets/favoriteLocation/customer/{id}/	GET	List all favorite location of the given customer
/api/markets/openclosemarketchain/{id}/	PATCH	Set open or close to the market chain. Once the market chain close all market and shop in the market chain will also close
/api/markets/openclosemarket/{id}/	PATCH	Set open or close to the market. Once the market close all shop in the market will also close

Figure 4.23: APIs of the markets service

Path	HTTP Method	Action
/api/orders/orders/	POST	Create an order
/api/orders/orders/{id}/	GET	Get a order detail from given order id
/api/orders/orders/customer/{id}/	GET	List all orders that related to the given customer
/api/orders/orders/shop/{id}/	GET	List all orders that related to the given shop
/api/orders/products/	POST	Create an ordered product with decorator of an order
/api/orders/state/{id}/	PATCH	Update state of the order
/api/orders/uncompleteOrder/shop/{id}/	GET	List all uncomplete or active order of the given shop

Figure 4.24: APIs of the orders service

Path	HTTP Method	Action
/api/shops/shopcategories/search/	GET	Search shop category by name
/api/shops/shopcategories/	POST	Create a shop category
/api/shops/shops/	POST	Create a shop
/api/shops/shops/{id}/	GET	Get shop profile
/api/shops/shops/{id}/	PATCH	Update shop profile
/api/shops/shops/{id}/	DELETE	Delete a shop
/api/shops/shops/market/{id}/	GET	List all shop that is open inside the given market
/api/shops/shops/market/{id}/search/	GET	Search shop by name that abelong to the given market
/api/shops/menucategories/	POST	Create a menu category in a shop
/api/shops/menucategories/{id}/	GET	Get menu category detail
/api/shops/menucategories/{id}/	PATCH	Update menu category detail
/api/shops/menucategories/{id}/	DELETE	Delete a menu category including all products belong to it
/api/shops/menucategories/customer/shop/{id}/	GET	List all active menu category that belong to the given shop (customer
/api/shops/menucategories/shop/{id}/	GET	List all menu category belong to the given shop (shop owner view)
/api/shops/products/{id}/	GET	Get product detail
/api/shops/products/{id}/	PATCH	Update product detail
/api/shops/products/{id}/	DELETE	Delete a product
/api/shops/products/customer/menucategory/{id}/	GET	List all active product that belong to given menu category (customer view)
/api/shops/products/menucategory/{id}/	GET	List all product that belong to given menu category without concern of
/api/shops/decorators/	POST	Create a decorator
/api/shops/decorators/{id}/	GET	Get decorator detail
/api/shops/decorators/{id}/	PATCH	Update decorator detail
/api/shops/decorators/{id}/	DELETE	Delete a decorator
/api/shops/decorators/product/{id}/	GET	List all decorator available for given product
/api/shops/decorators/shop/{id}/	GET	List all decorator that belong to the given shop
/api/shops/favouriteShop/	POST	Create a favourite shop
/api/shops/favouriteShop/{id}/	PATCH	Update the favourite shop
/api/shops/favouriteShop/{id}/	DELETE	Delete the favourite shop
/api/shops/favouriteShop/customer/{id}/	GET	List all favourite shops of the given customer

Figure 4.25: APIs of the shops service

Chapter 5
Result

5.1 Mobile user interface

The user interface for the customers have been divided into four main sections, which is the Search section (where we no longer associate the search function with user's current location), the Order History section, the Favorite section, and the Account section.

On the other hand, the user interface for the shop owner have been divided into four main sections, which is the Menu section, the Order section, the Report section, and the Account section.

5.1.1 Customer interface

Search section

Figure 5.1: Canteen searching result

Being the landing page of the mobile application, the customer can search for the food that they want to order. This process begins with selecting a particular canteen available within KMITL. The user can also manually search for a canteen using text input.

Figure 5.2: Canteen detail screen

When a canteen is selected, a customer will be able to look at a more detailed description of that particular canteen, including it's operating time, and a brief location of the canteen before deciding to select from the shops within.

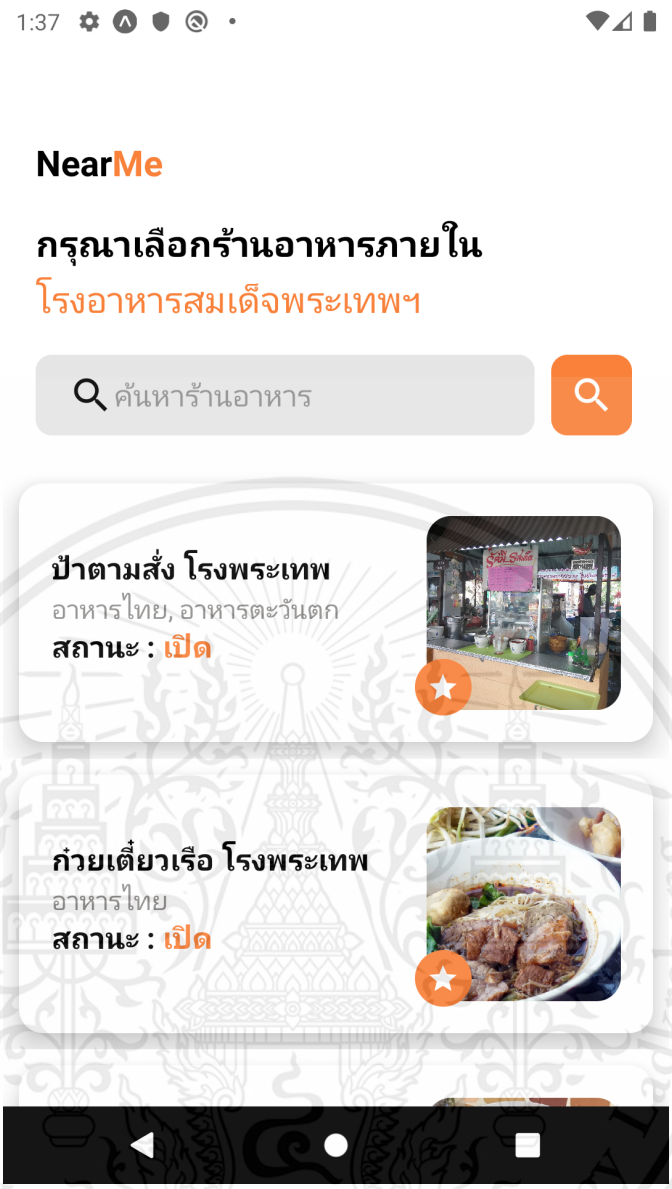

Figure 5.3: Shop display screen

After the Canteen detail screen, a customer will be presented with a list of shops, where they can select from all the currently operating shops within that canteen to view menus inside. In addition, they can use the favorite button next to a shop's portrait to mark a shop as favorite, making it easier to access at later time.

Figure 5.4: Menu display screen

Figure 5.5: Decorator screen

After the Shop display screen, a customer will be presented with a list of menus from the shop that they have selected. Information such as name and price of the menu will be displayed, along with the available decorator to customise the menu. A customer can use the text input search in case there are too many menus to browse.

If a customer selected a menu, a decorator screen will be brought up so that they can decide which customisation they prefer for that particular menu. When the decorators are decided, they can choose to go to the Cart screen to confirm their order.

Figure 5.6: Cart Screen

When menu decorators have been decided, customers can go to their cart to view the selected menu, and adjust the quantity of food that they have ordered. When this is complete, they can confirm the order so that it is sent to the respective shop owner.

Order History section

Figure 5.7: Order history

Figure 5.8: Order detail

This section displays the list of previous orders made by the customer.These orders will be labeled as a completed transaction, or an on going delivery. In addition, the customer can view a specific order's detail, or verify if an ongoing delivery is completed manually by selecting that order, then selecting order completion. The details being displayed include the shop's information, the type and quantity of ordered menu, and the delivery date/time.

Favorite section

Figure 5.9: Favorite Screen

This section displays the list of shops that has been marked as favorite by a customer. To mark a shop as a favorite, a customer use the favorite button back in the search shop selection screen. This serves as an easy way to access frequently visited shops without having to go through the steps of canteen search for the customers.

Account section

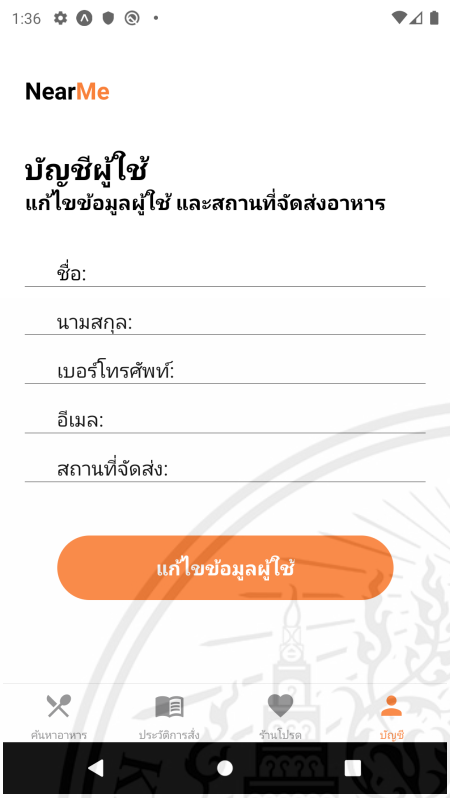

Figure 5.10: Customer profile

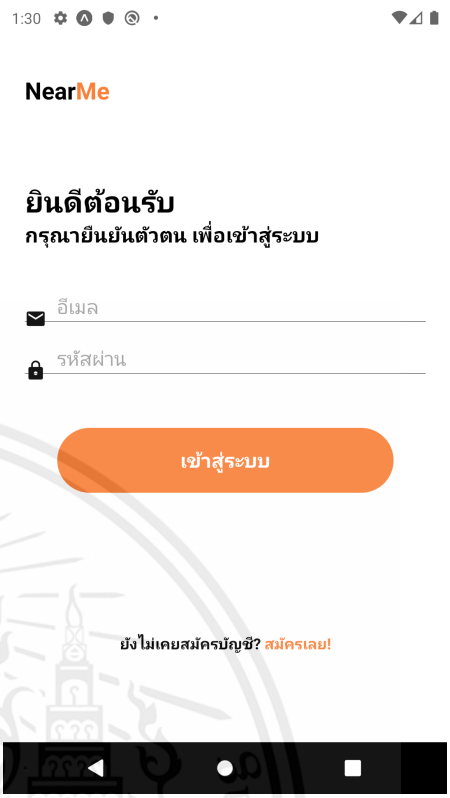

Figure 5.11: Login screen

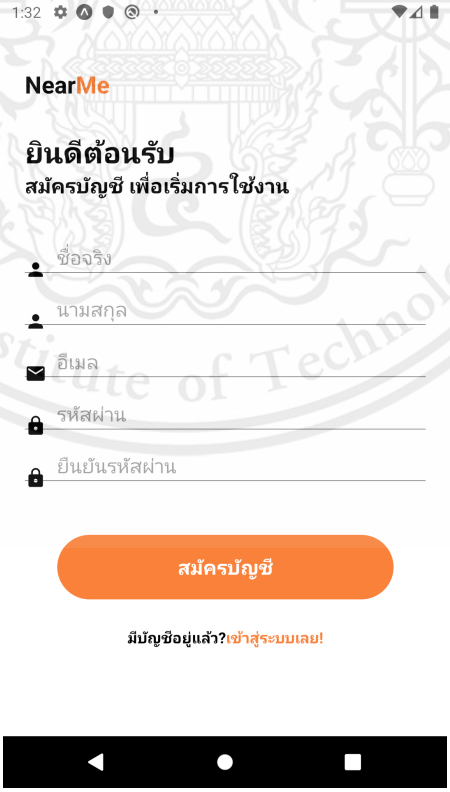

Figure 5.12: Registration screen

This section allow the customers to view their profile details, edit profile, and create an account. The displayed details in a customer profile such as default address can be edited, so that a customer can choose their delivery location for each orders.

5.1.2 Shop Owner Interface

Menu section

Figure 5.13: Menu list screen

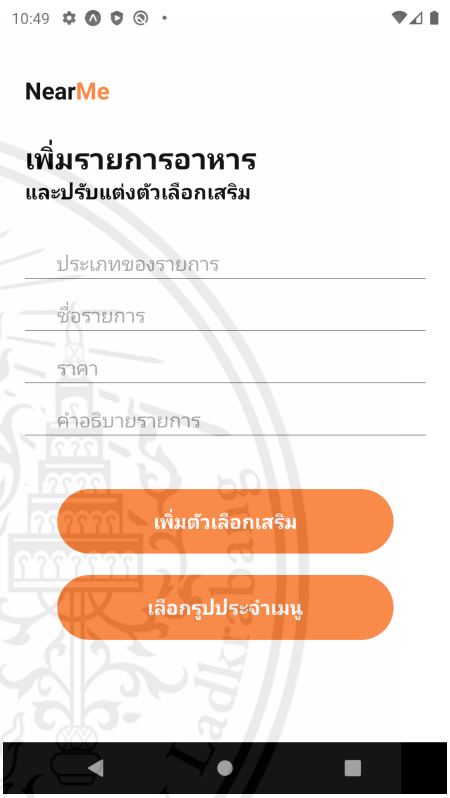

Figure 5.14: Menu customisation

The menu section allows the shop owner to manage the available menu list that they can currently provide to the customers for delivery. The shop owner can customize their menu detail such as photo representation, menu price, menu decorator, menu category, and menu name.

Order section

Figure 5.15: Order list screen

Figure 5.16: Order detail screen

The order section allows the shop owner to view incoming orders from the customers, then have the option to reject or accept that particular order. In addition, similar to customer side, the shop owner can also select completed orders to view details such as which customer was it sent to, where, what type and amount of menu, and at what time.

After accepting an order, the state of the order is considered to be currently processing to the customer.

Report section

Figure 5.17: Report screen

The report section allows the shop owner to keep track of the quantities of product delivered, and record sales revenue generated by an individual menu.

Account section

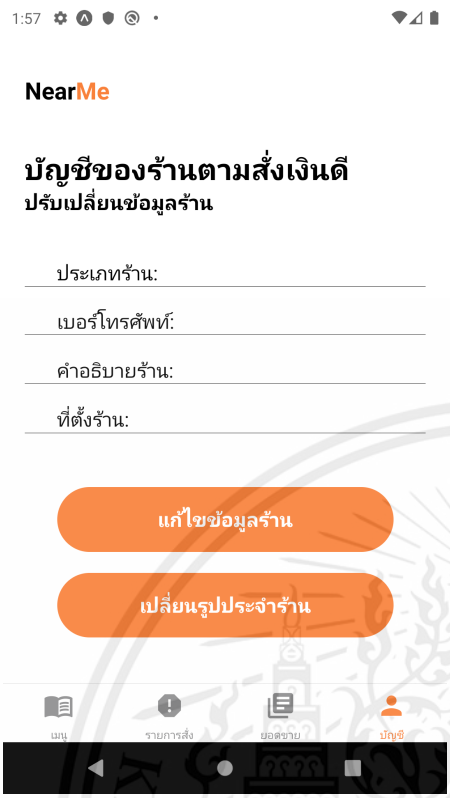

Figure 5.18: Shop owner profile

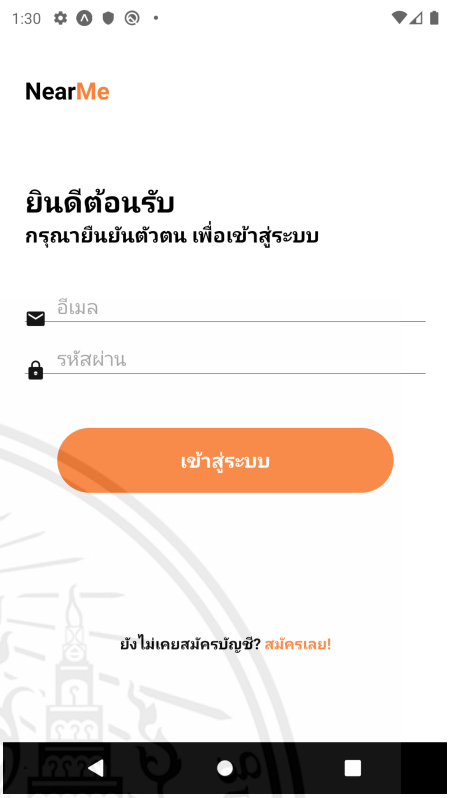

Figure 5.19: Login screen

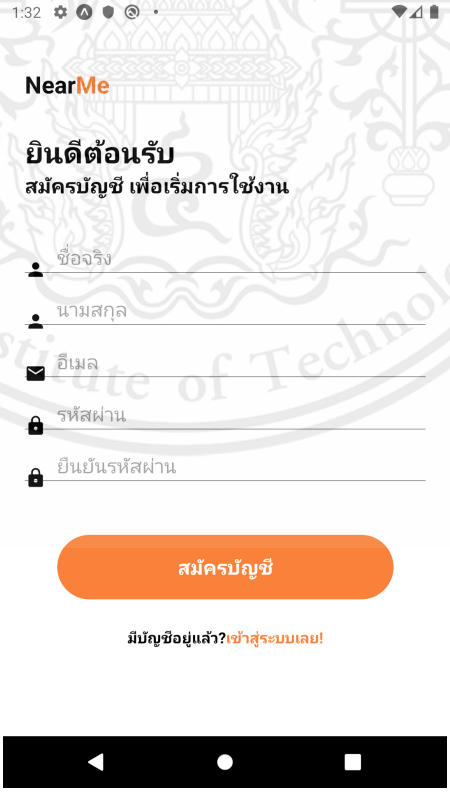

Figure 5.20: Registration screen

This section allows the shop owner to view and customize their profile, such as their shop profile picture, address, shop description, phone number, and the shop category. These details will be shown to customers when they navigate to the shop owner's store, therefore, it is vital for each shop owner to clarify their identity to the customers with these details.

Chapter 6
Conclusion

6.1 What we have done

We have implemented mobile applications for each type of user, where a shop owner and a customer can interact with each other to either display their food product through the system (shop owner) or place an order for a delivery (customer).

For the server we have implemented Django web application that can handle multiple role of users and be able to serve client-side through API endpoints. The continuous integration is also been adopted by connecting Github and Heroku together. With this feature, redeployment can be done easily and quickly. For the database, we have properly hosted on Amazon web services using Heroku add-ons feature. For the images, we have registered for Cloudinary cloud service and connect it to the web application static files.

A React Native Javascript user interfaces have been created to display, get and post information to/from the server-side. However, some features are left uncompleted and not fully integrated due to sub-par time management. These features include fully integrating the third role (Market admin), some data fields meant to be displayed being left out, among others.

6.2 Problem we face

Some of the problems that we have faced are time management, resource management and sub-par collaboration effort. By the time our API endpoints have been deployed, working properly, and with all necessary methods included; the time frame available for any integration efforts are already too limited to complete all that's need to be done. This resulted in work rush, leading to the client-side interface of the shop owner not being as polish as we wish it would have been (both in terms of styling and some features being left out in the end).

6.3 Possible improvement

Features that could be added to our current system includes a more in-depth analytic feature for shop owners. This include detailing the sales report in a more descriptive way, such as making it possible to filter sales according to certain amount of weeks, months, or annually, and provide feature to analyze it using machine learning. In addition, a more systematic and descriptive way to locate a delivery location could be implemented, similar to how google map display a user location. Finally, a better coordination effort and time management could help us achieve a more completed product in the future.

Bibliography

[1] Christie, T. (2020). Django REST Framework. Home - Django REST framework. https://www.django-rest-framework.org/.

[2] Django Software Foundation. (2020). Django documentation. Django. https://docs.djangoproject.com/en/3.1/.

[3] Facebook Open Source. (2020, October 29). Introduction · React Native. React Native. https://reactnative.dev/docs/getting-started.

[4] Grab SG. (2020, April 21). #AskGrab: Where does the merchant commission go? https://www.grab.com/sg/blog/askgrab-where-does-the-merchant-commission-go/.

[5] Grab. (2020). Learn how to use the features in your GrabFood Merchant App. Grab Help Center. https://help.grab.com/merchant/en-my/360027631612-Learn-how-to-use-the-features-in-your-GrabFood-Merchant-App.

[6] Hack Reactor. (2018, October 18). What is JavaScript Used For? Hack Reactor. https://www.hackreactor.com/blog/what-is-javascript-used-for.

[7] Moneymax. (2020, November 19). Foodpanda vs GrabFood: Food Delivery Service Battle. Moneymax. https://www.moneymax.ph/personal-finance/articles/foodpanda-vs-grabfood.

[8] PostgreSQL Tutorial. (2020). What is PostgreSQL? PostgreSQL Tutorial. https://www.postgresqltutorial.com/what-is-postgresql/.

[9] Rao, T. (2018, December 4). Building FoodPanda 3.0 and reinventing the food discovery experience. Medium. https://medium.com/designbytanya/designing-a-better-food-discovery-user-experience-while-building-the-new-foodpanda-app-9b13270236e.

[10] Rouse, M. (2020, September 22). What is REST API (RESTful API)? SearchAppArchitecture. https://searchapparchitecture.techtarget.com/definition/RESTful-API.

[11] Singh, A. (2020, October 21). Food On Demand : Business Models of Meal Delivery Startups. JungleWorks. https://jungleworks.com/food-on-demand-business-models-of-meal-delivery-startups/.

[12] Stein, A. (2020, May 8). How QR Codes Work and Their History. QR Code Generator. https://www.qr-code-generator.com/blog/how-qr-codes-work-and-their-history/.

[13] Viktor. (2020, November 21). The Food Delivery Business Model – A Complete Guide. productmint. https://productmint.com/the-food-delivery-business-model-a-complete-guide/.

[14] W3Schools. (2020). What is React? What is React. https://www.w3schools.com/whatis/whatis_react.asp.

[15] Welcome to Python. Python.org. (2020). https://www.python.org/.

[16] What Is React Native and How Is It Used? (2021, March 15). https://www.netguru.com/what-is-react-native.

[17] Apache Software Foundation. (2021). What is elasticsearch? https://www.elastic.co/what-is/elasticsearch.

[18] The PostgreSQL Global Development Group. (2021, February 11). Full Text Search. PostgreSQL Documentation. https://www.postgresql.org/docs/9.5/textsearch.html.

[19] Ryan Browne.(2021, June 28) What you need to know about the European food delivery giant that beat Uber to a deal with Grubhub.https://www.cnbc.com/2020/06/11/what-is-just-eat-takeaway-all-you-need-to-know.html