BMIT3173 Integrative Programming

ASSIGNMENT 202505

Student Name : Tai Jun An

Student ID : 24WMR09007

Programme : Bachelor of Information Technology in Information Security

Tutorial Group : 5

System Title : DineEase Food Ordering System

Modules : Ordering and Cart Module

Declaration

- I confirm that I have read and complied with all the terms and conditions of Tunku Abdul Rahman University of Management and technology's plagiarism policy.
- I declare that this assignment is free from all forms of plagiarism and for all intents and purposes is my own properly derived work.

Plagiarism Statement Form

I, Name (Block Capitals) **TAI JUN AN** Student ID **24WMR09007** Programme **Bachelor of Information Technology in Information Security** Tutorial Group **5** confirm that the submitted work are all my own work and is in my own words.

I Tai Jun An acknowledge the use of AI generative technology.

Signature: [illegible]

Date: **14/9/2025**

Table of Contents

1. Introduction to the System 4
2. Module Description 5
    2.1 Explanation of Ordering and Cart Module 6
3. Entity Classes 12
4. Design Pattern 17
    4.1 Description of Design Pattern 17
    4.2 Implementation of Design Pattern 18
        4.2.1 State Pattern Implementation in Order State Management 18
5. Software Security 29
    5.1 Potential Threat/Attack 29
    5.2 Secure Coding Practice 31
6. Web Services 35
7. Index 41

1. Introduction to the System

The **DineEase Food Ordering System** is an online platform created for **Kiaty Western Sdn Bhd**, a restaurant looking to improve the dining experience for customer and make internal operations more efficient. Currently, the restaurant uses traditional methods where customer wait for staff to take their orders at the table. These orders are either written down or typed into the POS system, and payments are made at the counter or at the table after the meal. This method lead to delays, mistake in orders, and a lot of work for staff, especially during busy time.

To fix these problem, the DineEase system introduces a **QR-based digital ordering system**. When customers sit down, they scan a QR code on their table that takes them to a mobile-friendly website where they can browse the menu, place their order, and pay all from their phone. Customer can log in as members to earn and redeem loyalty point, or create a new account. The system allow customer to order directly, reducing the need for staff involvement in taking order. This speeds up the process, reduces mistake, and allows the restaurant to serve more customers in less time. The system also give staff access to an **admin panel** to manage user accounts, update the menu, and keep track of order.

The system also makes **payment** and **rewards management** easier. Once the customer finishes their order, they can pay through cash or e-wallet, and the system automatically generate a receipt. The restaurant table is mark as cleared when the payment is done. The loyalty program reward customer with points for every purchase, which can used for discounts or other rewards. This encourage customers to visit again. Overall, DineEase help to create a **faster and more personalized dining experience** for customer, while making operations smoother and more efficient for restaurant staff.

2. Module Description

Ordering and Cart Module:

The **Ordering and Cart Module** is design to make it easy for customer to order food from their phone. When customer sit at their table and scan the QR code, they are redirect to the restaurant menu. The menu show them different food and drink choices. They can look at the food name, price, description, and options like how spicy it is or how big the portion are. Customers can add things to the cart and change what they order if they don't like it. This makes ordering food faster and easier.

Customer can manage they cart by adding or removing things before they finish they order. When they happy with their order, they can submit it directly to the kitchen. This saves time for the staff because they don't need to write down orders, and it also help reduce mistake that might happen. The module also shows the customer what happen to their order, like if it's waiting, cooking, or ready to serve. This lets customers know what's going on without asking the staff every time.

By making the ordering process automatic, this module makes the whole experience faster and more easy for both customer and staff. There is no need for the staff to take each order by hand, so this reduce work for them and make everything goes faster, especially when the restaurant is busy. The system also makes sure the orders are clear and accurate, so there is less chance of mistake or confusion. All in all, the **Ordering and Cart Module** makes ordering food more convenient, helps reduce mistakes, and makes the whole service faster and smoother for everyone.

2.1 Explanation of Ordering and Cart Module

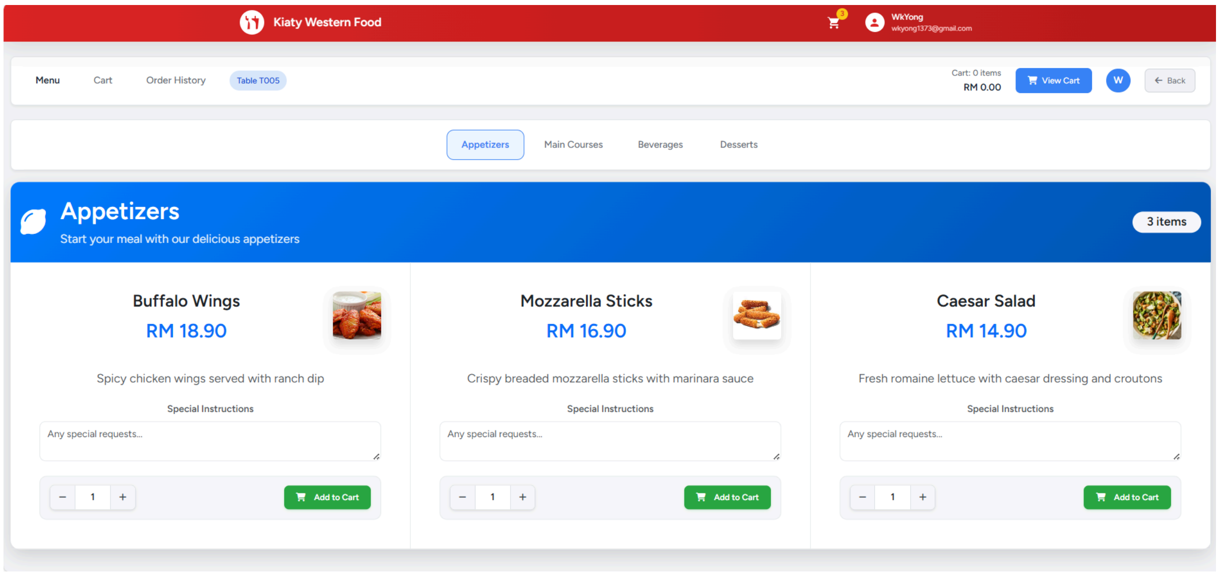


Figure 2.1 Menu Page

Figure 2.1 show the **Menu Page**, where customer can look at all food items to order. The page has sections like **Appetizers**, and every item has name, description, and price. Customer can click on the "Add to Cart" button to add things to the cart. They can also put special request for the dish if they want. This page lets customer see all the food options, their prices, and pick what they want to order.

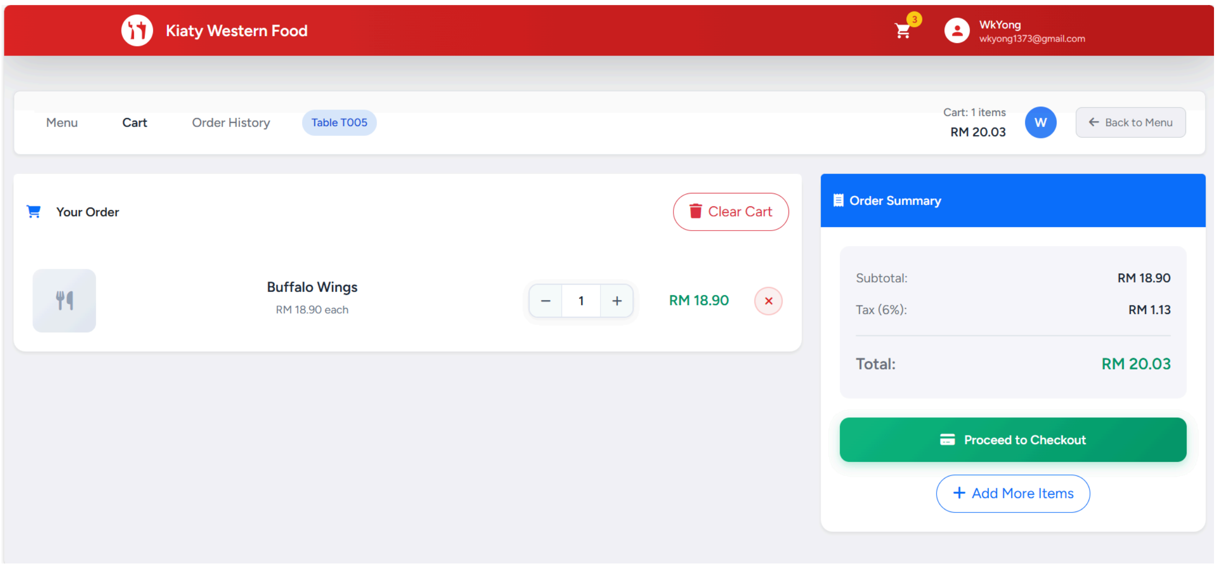
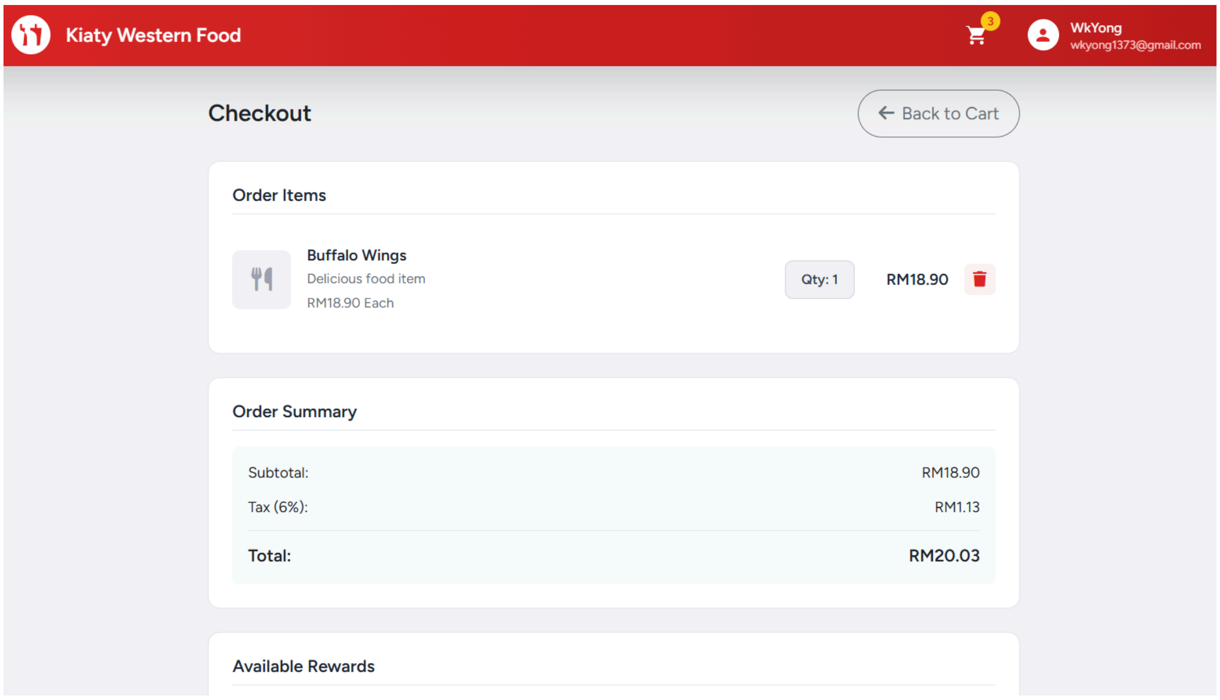


Figure 2.2 Cart Page

Figure 2.2 is the **Cart Page**. It shows the things the customer added to their cart. It shows the **item name**, how many items of it, and the total price for each one. The customer can change how many they want or even clear the cart if they don't want anything. On the side, it shows an **Order Summary**, where the customer can see the subtotal, tax, and total. The customer can check everything before going to the checkout.



Order Items

Buffalo Wings
Delicious food item
RM18.90 Each
Qty: 1
RM18.90

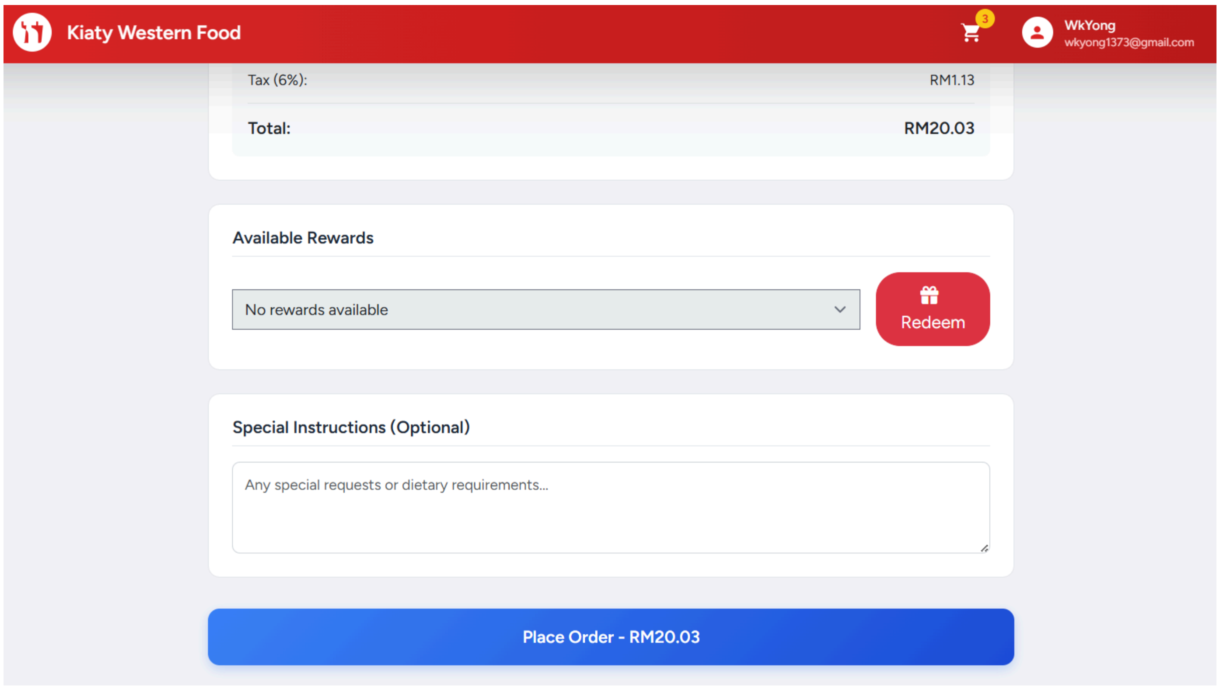
Order Summary

Subtotal:	RM18.90
Tax (6%):	RM1.13
Total:	**RM20.03**

Available Rewards

Figure 2.3 Checkout Page

Figure 2.3 show the **Checkout Page**, where the customer looks at their final order. It shows everything in the cart and the **total price**. The **Order Summary** also shows again with the **subtotal** and **tax**. The customer can also check if they have any rewards to use for discount. This page is where the customer confirm the order before paying.



Tax (6%):	RM1.13
Total:	**RM20.03**

Available Rewards

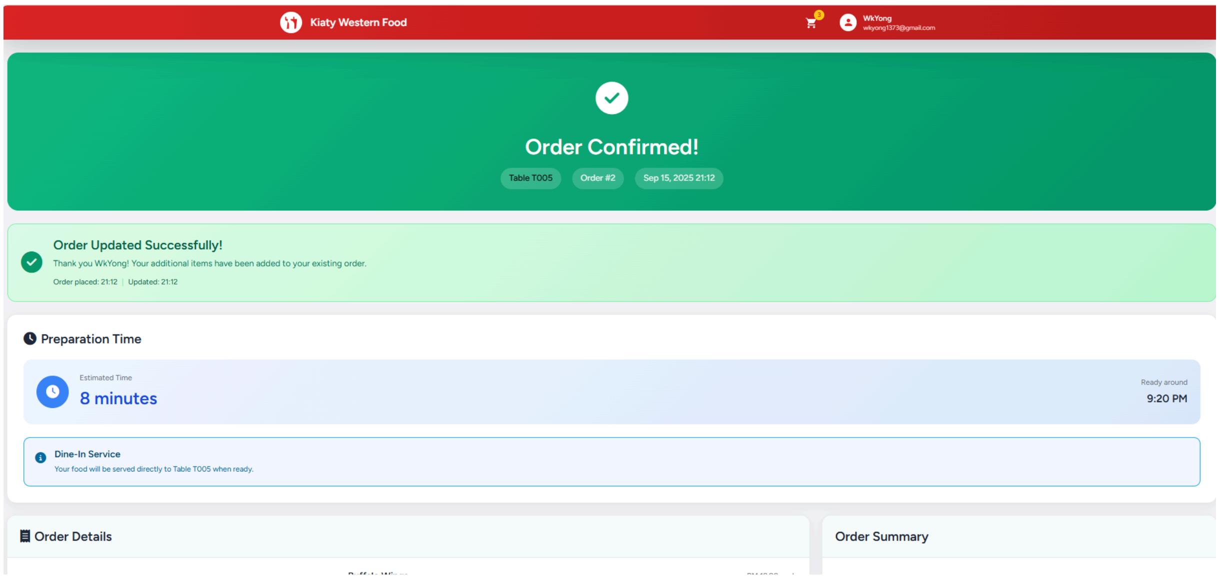
No rewards available

Special Instructions (Optional)

Any special requests or dietary requirements...

Figure 2.4 Checkout Page (with Rewards)

Figure 2.4 is the continuation of Checkout Page, the customer can use rewards to get discounts. The customer can see how many rewards they have and apply them to reduce the price. There's also a place to add any special instructions. After checking everything, the customer can click "Place Order" button to place their order.


Kiaty Western Food

WkYong
wkyong1373@gmail.com

Order Confirmed!

Table T005 | Order #2 | Sep 15, 2025 21:12

Order Updated Successfully!

Thank you WkYong! Your additional items have been added to your existing order.

Order placed: 21:12 | Updated: 21:12

Preparation Time

Estimated Time
8 minutes

Ready around
9:20 PM

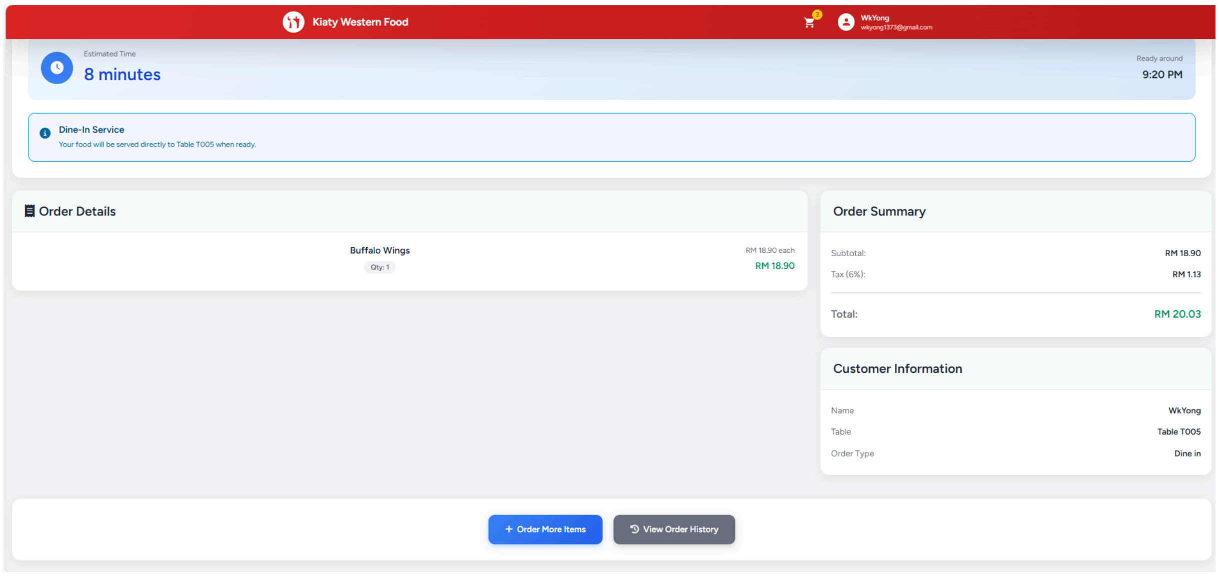
Dine-In Service
Your food will be served directly to Table T005 when ready.

Order Details

Order Summary

Figure 2.5 Order Confirmed Page

Figure 2.5 shows the **Order Confirmed Page**. After the customer places the order, a green message shows to say the order was successfully placed. It also shows the **preparation time** and order details, including when the order was placed. This page confirms to the customer that their order was received and is now being processed.



Estimated Time
8 minutes

Ready around
9:20 PM

Dine-In Service
Your food will be served directly to Table T005 when ready.

Order Details

Buffalo Wings
Qty: 1
RM 18.90 each
RM 18.90

Order Summary

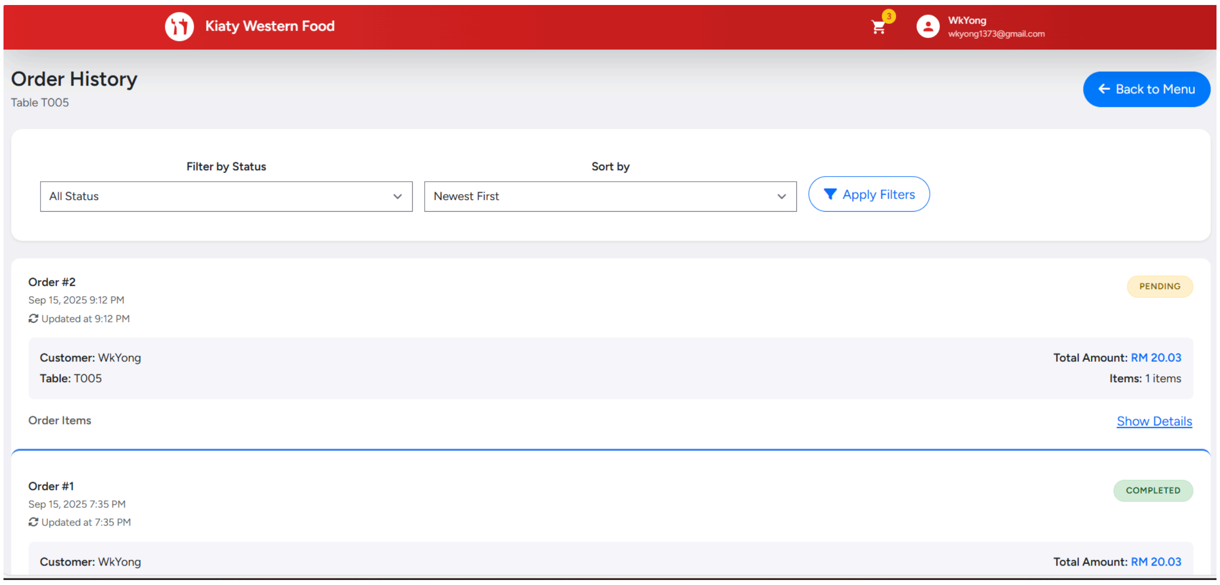
Subtotal:	RM 18.90
Tax (6%):	RM 1.13
Total:	RM 20.03

Customer Information

Name	WkYong
Table	Table T005
Order Type	Dine in

Figure 2.6 Order Confirmed Page

Figure 2.6 is the continuation of Order Confirmed Page. It shows the order details, like what items were ordered and their prices. It also shows preparation time and order summary with info about food and payment. The customer can click to Order New Items or View Order History to keep using the system.



Order History
Table T005

Filter by Status: All Status
Sort by: Newest First

Order #2
Sep 15, 2025 9:12 PM
Updated at 9:12 PM
PENDING

Customer: WkYong
Table: T005
Total Amount: RM 20.03
Items: 1 items

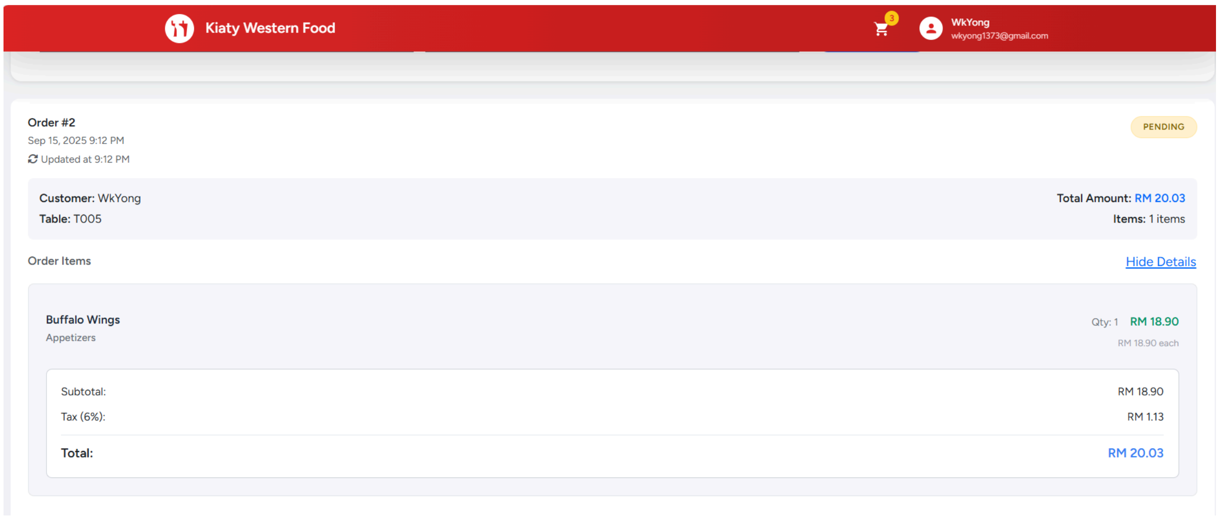
Order Items

Order #1
Sep 15, 2025 7:35 PM
Updated at 7:35 PM
COMPLETED

Customer: WkYong
Total Amount: RM 20.03

Figure 2.7 Order History Page

In this figure, we see the **Order History Page**. It shows a list of all orders that the customer made before. The customer can **filter** orders by **status** and sort them by the newest first. The list shows order number, total price, and **status** (like Pending, Completed). The customer can click on any order to see more details.


Kiaty Western Food
WkYong
wkyong1373@gmail.com

Order #2
Sep 15, 2025 9:12 PM
Updated at 9:12 PM
PENDING

Customer: WkYong
Table: T005
Total Amount: RM 20.03
Items: 1 items

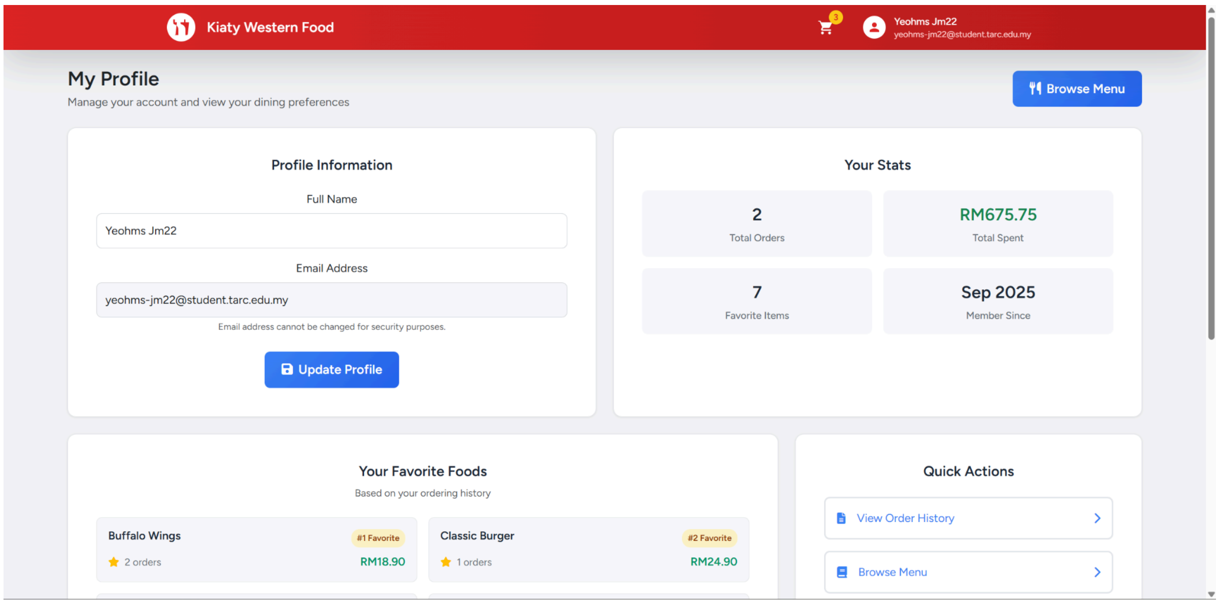
Order Items
Hide Details

Buffalo Wings
Appetizers
Qty: 1 RM 18.90
RM 18.90 each

Subtotal:	RM 18.90
Tax (6%):	RM 1.13
Total:	RM 20.03

Figure 2.8 Order History Page

Figure 2.8 is a details look at an order from the Order History Page. It shows more details about the order, like the items, their prices, and the total amount. The customer can also see tax and any other charges. This page helps the customer check what they have ordered and how much they need pay and what they have paid before.


Kiaty Western Food
Yeohms Jm22
yeohms-jm22@student.tarc.edu.my

My Profile
Manage your account and view your dining preferences
Browse Menu

Profile Information
Full Name: Yeohms Jm22
Email Address: yeohms-jm22@student.tarc.edu.my
Email address cannot be changed for security purposes.
Update Profile

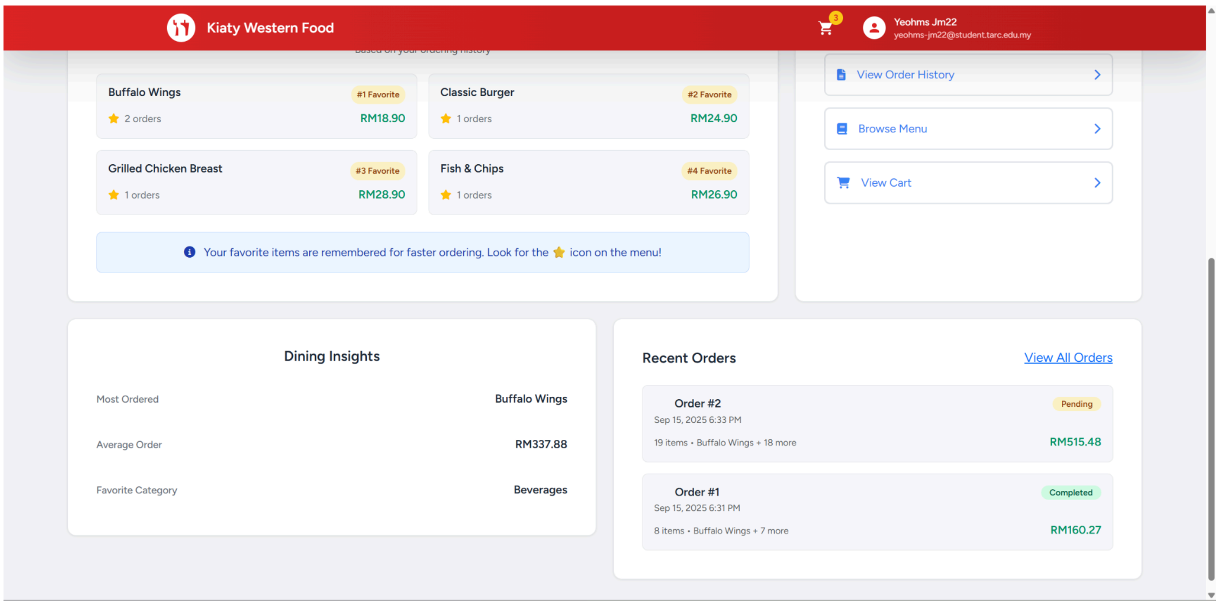
Your Stats
2 Total Orders
RM675.75 Total Spent
7 Favorite Items
Sep 2025 Member Since

Your Favorite Foods
Based on your ordering history
Buffalo Wings #1 Favorite 2 orders RM18.90
Classic Burger #2 Favorite 1 orders RM24.90

Quick Actions
View Order History
Browse Menu

Figure 2.9 Profile Page

The **Profile Page** shows the customer's information, like their **name**, **email**, and order details. It also shows how much they've spent in total, how many orders they've made, and what foods they like the most. The customer can change their profile or look at their **order history** and browse the menu.



Kiaty Western Food

3

Yeohms Jm22
yeohms-jm22@student.tarc.edu.my

Buffalo Wings	#1 Favorite	
2 orders	RM18.90	
Classic Burger	#2 Favorite	
1 orders	RM24.90	
Grilled Chicken Breast	#3 Favorite	
1 orders	RM28.90	
Fish & Chips	#4 Favorite	
1 orders	RM26.90	

Your favorite items are remembered for faster ordering. Look for the ⭐ icon on the menu!

View Order History

Browse Menu

View Cart

Dining Insights

Most Ordered	Buffalo Wings
Average Order	RM337.88
Favorite Category	Beverages

Recent Orders — View All Orders

Order #2 — Pending
Sep 15, 2025 6:33 PM
19 items • Buffalo Wings + 18 more — RM515.48

Order #1 — Completed
Sep 15, 2025 6:31 PM
8 items • Buffalo Wings + 7 more — RM160.27

Figure 2.10 Profile Page

Figure 2.10 is a continuation of **Profile Page**. It shows the customer's **favorite items**, recent orders, and details like the most ordered food. The page also shows **order statuses** and **prices** for the recent orders. This page helps the customer see their preferences and spending habits, making it easy to reorder their favorite food.

3. Entity Classes

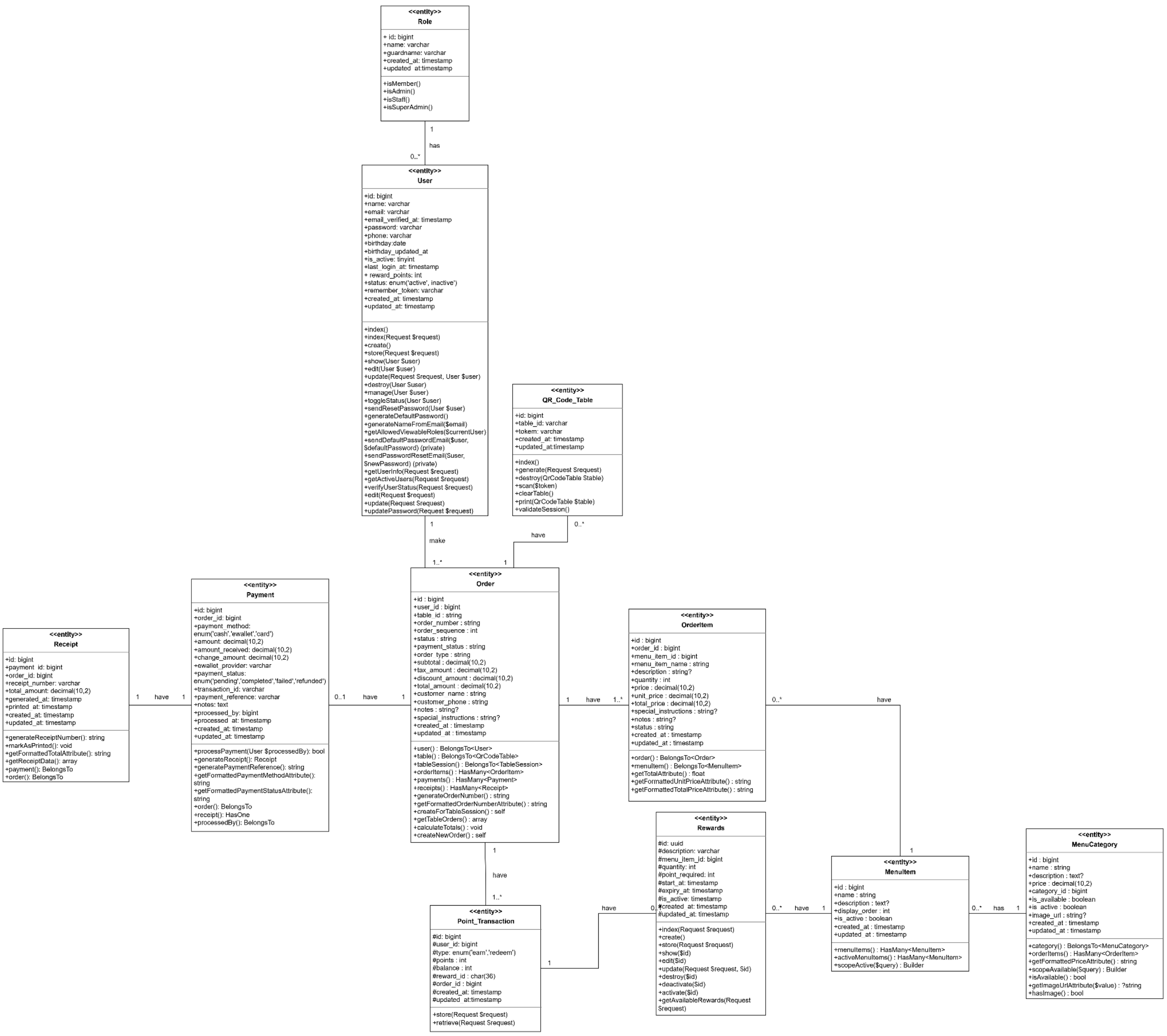


Diagram 3.1.1 Entity Class Diagram

Link:
https://drive.google.com/file/d/1g9-2sgluakqzYeD9Fs3JF-or1W6gndyq/view?usp=sharing

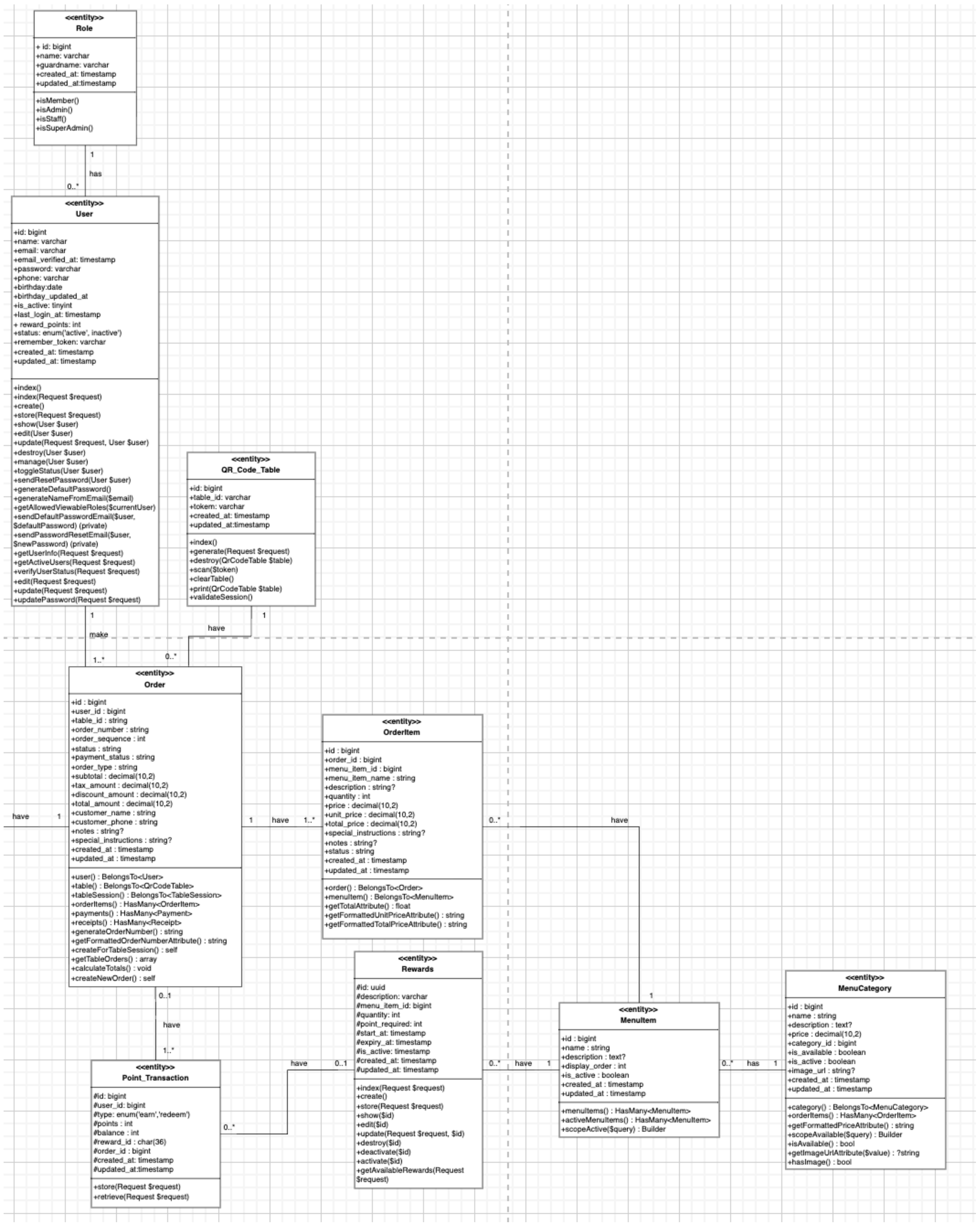


Diagram 3.1.2 Entity Class Diagram

1. User

The User class represent peoples who using the system, like customer or admin. They can signup, login, make orders, and check their profile. Some part of User class is:

- id: Special number that identify each user.
- name: Name of user, help to know who order food.
- email: Email for login, help to contact user.
- timestamps: Date when account was created or updated.

Relation with Order: User can make many order, so User and Order is connect. One user can have many order, but each order only belongs to one user.

2. Order

The Order class is for customer's order in the system. It keep track about what customer order. Some part of Order class are:

- id: Special number for each order.
- total_amount: Total money that customer need to pay for order.
- status: State of order, like waiting, cooking, or finish.
- customer_name: Name of person who make the order.
- address: Where food should be deliver.
- timestamps: When order made or update.

Relation with User: Order belongs to one user only, but user can have many order. So, User and Order is one-to-many relation.

Relation with OrderItem: One order can have many item, so Order and OrderItem is connected. Each order can contain many items.

3. OrderItem

The OrderItem class is for each item in the customer's order. Order can have many item. Some part of OrderItem class is:

- id: Special number for each order item.
- item_name: Name of food or drink that was ordered.
- quantity: How many of this item the customer order.
- price: Price of each unit of the item.
- special_instructions: Special request like no onions, extra spicy.

Relation with Order: Each OrderItem is part of one order, but one order can have many items. So, OrderItem and Order have one-to-many relation.

Relation with MenuItem: Each OrderItem is connected to one MenuItem, showing which menu item it came from. Many OrderItem can refer to the same MenuItem.

4. MenuItem

The MenuItem class is for the food or drink that customer can order. It got details like name, description, price, availability_status, and category_id.

- name: Name of the menu item.
- description: Tell what inside the menu item.
- price: Price of the item.
- availability_status: Whether item available or not.
- category_id: Tell which category this menu item belong to, like appetizers or drinks.

Relation with OrderItem: MenuItem is related to OrderItem, means each item in the order refer to one menu item. One MenuItem can have many OrderItem.

Relation with MenuCategory: MenuItem belongs to one MenuCategory, mean it's part of a category like appetizers or drinks. One category can have many menu items, but an item can belong to one category.

5. MenuCategory

The MenuCategory class is for group menu items into section like appetizers, main courses, desserts, drinks. This help customers find food easier. It got parts like:

- id: Unique number for each category.
- name: Name of the category like appetizers or drinks.
- description: Explain what this category contain.

Relation with MenuItem: MenuCategory can have many MenuItems in it, so one category can have many items, but each item belong to just one category. This is a one-to-many relation.

6. QR_Code_Table

The QR_Code_Table class mean the QR codes for each table. When customer scan the QR code at table, they can see the menu and order. It contain attributes like id, qr_code, table_number, and timestamps.

- id: Unique number for QR code.
- qr_code: Actual data of the QR code.
- table_number: The number of the table.
- timestamps: The date when QR code was created or updated.

Relation with User: QR_Code_Table help customer to connect to their menu, but it don't directly link to users. It just make the ordering process easier by linking customer to their table.

7. Rewards

The Rewards class is for loyalty points that customer earn for ordering food. It store info about points, how much balance customer got, and rules to use points. Some parts of Rewards class are:

- reward_value: Total points customer earned.
- balance: How many points left for user.
- redemption_rules: Rules for when customer can use their points.

Relation with User: Every user has own rewards, so Rewards class link to User in one-to-one relation. Each user has only one rewards account.

8. Point_Transaction

The Point_Transaction class track the point movements, like when points are earned or spent. It records things like id, transaction_type, points, and timestamp.

- transaction_type: Whether points earned or used.
- points: Number of points added or subtracted.
- timestamp: The date and time when transaction happened.

Relation with User: Each Point_Transaction linked to one user, so it's clear which user earned or spent points.

Relation with Rewards: Point_Transaction affect the user's reward balance, so it's linked to Rewards class, showing transaction impact on points.

9. Relationships and Associations

There's a connection between User and Order. A user can make many orders, but each order belongs to only one user. Order and OrderItem are connected, mean every order can have many items, but each item belong to only one order. OrderItem refers to MenuItem, so every item in an order is connected to one menu item. MenuItem and MenuCategory are linked, so every menu item is part of a category. User is connected to Rewards, meaning every user has their own reward balance, and Point_Transaction and Rewards is connected, showing each point transaction change the user's points.

4. Design Pattern

The **State pattern** is implemented in the Ordering and Cart Module to handle different order states and their transitions. The State pattern allows an object to alter its behavior when its internal state changes, making the order workflow management more organized and maintainable. In this case, the Ordering and Cart Module works like a State Machine that manages all the complicated state transitions for Order Status, Payment Processing, and Order Workflow management. Instead of the system needing to know about all the state rules, it just calls the Order State Module, which takes care of everything else. This makes the system more organized, easier to use, keep, and change in future.

4.1 Description of Design Pattern

The **Order Management Module** in my system uses the **State pattern** to make it easier for the system to handle order state transitions. It gives a simple way for the system to manage order states without having to know about all the complex state transition rules happening behind the scenes. It hides the **OrderState**, **PaymentState**, and **WorkflowState** parts of the system. For example, the **processPayment()**, **confirmOrder()**, and **cancelOrder()** functions are used by the system. These functions do a lot of work but the system does not need to know about that. The **Order** manages the food orders, the Payment handles the payment processing, and the Workflow takes care of order status management, but the user does not need to worry about all the **Order State Module**. This makes the system simpler for the user and easier to change if needed in future. It also helps when new features or order states are added in future, because the system does not need to worry about the changes.

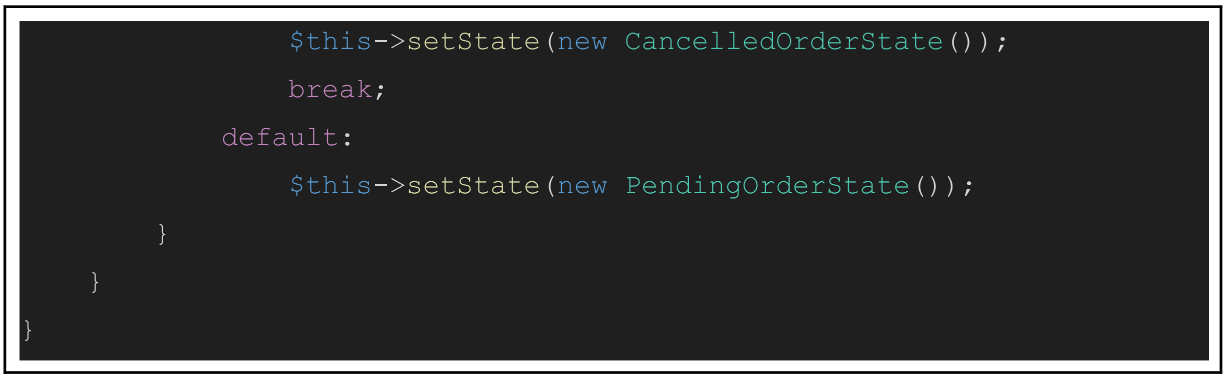
4.2 Implementation of Design Pattern

4.2.1 State Pattern Implementation in Order State Management

The **State pattern** is used with **State classes** that manage complex order state transitions and behaviors by providing a simple interface. The **State layer** works as a single point where the **Controllers, Models, Business Logic**, and **State Transitions** are connected. This makes everything simpler because instead of directly dealing with all the complicated state management, you just use the **State** layer to do everything. The example code shows how different **state facades** are used to handle complicated state transitions in a way that is more organized and easier to work with.

```php
<?php
// STATE PATTERN IMPLEMENTATION - Different order state strategies
interface OrderStateInterface
{
    public function processPayment(Order $order): string;
    public function startPreparation(Order $order): string;
    public function markReady(Order $order): string;
    public function completeOrder(Order $order): string;
    public function cancelOrder(Order $order): string;
    public function getStatusName(): string;
    public function getAllowedTransitions(): array;
}

// Concrete State Classes
class PendingOrderState implements OrderStateInterface
{
    public function processPayment(Order $order): string
    {
        // Customer pays after eating - transition to preparing
        $order->payment_status = 'completed';
        $order->status = 'preparing';
        $order->setState(new PreparingOrderState());
        $order->save();

        return 'Payment completed. Kitchen will now prepare your
order.';
    }

    public function startPreparation(Order $order): string
    {
        throw new Exception('Cannot start preparation without payment
confirmation');
    }

    public function markReady(Order $order): string
    {
        throw new Exception('Cannot mark order ready from pending
state');
    }

    public function completeOrder(Order $order): string
    {
        throw new Exception('Cannot complete unpaid order');
    }

    public function cancelOrder(Order $order): string
    {
        $order->status = 'cancelled';
        $order->setState(new CancelledOrderState());
        $order->save();

        return 'Order cancelled successfully';
    }

    public function getStatusName(): string
    {
        return 'pending';
    }

    public function getAllowedTransitions(): array
    {
        return ['preparing', 'cancelled'];
    }
}

class PreparingOrderState implements OrderStateInterface
{
    public function processPayment(Order $order): string
    {
        throw new Exception('Order already paid');
    }

    public function startPreparation(Order $order): string
    {
        return 'Order is already being prepared';
    }

    public function markReady(Order $order): string
    {
        $order->status = 'ready';
        $order->setState(new ReadyOrderState());
        $order->save();

        return 'Order is ready for serving';
    }

    public function completeOrder(Order $order): string
    {
        throw new Exception('Cannot complete order that is still being
prepared');
    }

    public function cancelOrder(Order $order): string
    {
        // Can cancel if kitchen hasn't started yet
        $order->status = 'cancelled';
        $order->setState(new CancelledOrderState());
        $order->save();

        return 'Order cancelled. Refund will be processed.';
    }

    public function getStatusName(): string
    {
        return 'preparing';
    }

    public function getAllowedTransitions(): array
    {
        return ['ready', 'cancelled'];
    }
}

class ReadyOrderState implements OrderStateInterface
{
    public function processPayment(Order $order): string
    {
        throw new Exception('Order already paid');
    }

    public function startPreparation(Order $order): string
    {
        throw new Exception('Order already prepared');
    }

    public function markReady(Order $order): string
    {
        return 'Order is already ready';
    }

    public function completeOrder(Order $order): string
    {
        $order->status = 'completed';
        $order->setState(new CompletedOrderState());
        $order->save();

        return 'Order completed. Customer has finished dining.';
    }

    public function cancelOrder(Order $order): string
    {
        throw new Exception('Cannot cancel order that is ready for
serving');
    }

    public function getStatusName(): string
    {
        return 'ready';
```

```php
    }

    public function getAllowedTransitions(): array
    {
        return ['completed'];
    }
}

class CompletedOrderState implements OrderStateInterface
{
    public function processPayment(Order $order): string
    {
        throw new Exception('Order already completed');
    }

    public function startPreparation(Order $order): string
    {
        throw new Exception('Order already completed');
    }

    public function markReady(Order $order): string
    {
        throw new Exception('Order already completed');
    }

    public function completeOrder(Order $order): string
    {
        return 'Order already completed';
    }

    public function cancelOrder(Order $order): string
    {
        throw new Exception('Cannot cancel completed order');
    }

    public function getStatusName(): string
    {
        return 'completed';
    }
    public function getAllowedTransitions(): array
    {
        return [];
    }
}

class CancelledOrderState implements OrderStateInterface
{
    public function processPayment(Order $order): string
    {
        throw new Exception('Cannot process payment for cancelled
order');
    }

    public function startPreparation(Order $order): string
    {
        throw new Exception('Cannot prepare cancelled order');
    }

    public function markReady(Order $order): string
    {
        throw new Exception('Cannot mark cancelled order as ready');
    }

    public function completeOrder(Order $order): string
    {
        throw new Exception('Cannot complete cancelled order');
    }

    public function cancelOrder(Order $order): string
    {
        return 'Order already cancelled';
    }

    public function getStatusName(): string
    {
        return 'cancelled';
    }
    public function getAllowedTransitions(): array
    {
        return [];
    }
}

// Context Class - Order Model with State
class Order extends Model
{
    protected $state;

    public function setState(OrderStateInterface $state)
    {
        $this->state = $state;
    }

    public function getState(): OrderStateInterface
    {
        if (!$this->state) {
            $this->loadStateFromStatus();
        }
        return $this->state;
    }

    // State-dependent methods
    public function processOrderPayment(): string
    {
        return $this->getState()->processPayment($this);
    }

    public function startOrderPreparation(): string
    {
        return $this->getState()->startPreparation($this);
    }

    public function markOrderReady(): string
    {
        return $this->getState()->markReady($this);
    }

    public function completeOrderStatus(): string
    {
        return $this->getState()->completeOrder($this);
    }

    public function cancelOrderStatus(): string
    {
        return $this->getState()->cancelOrder($this);
    }

    public function getCurrentStatus(): string
    {
        return $this->getState()->getStatusName();
    }

    public function getAvailableTransitions(): array
    {
        return $this->getState()->getAllowedTransitions();
    }

    // Load state from database status
    public function loadStateFromStatus()
    {
        switch ($this->status) {
            case 'pending':
                $this->setState(new PendingOrderState());
                break;
            case 'preparing':
                $this->setState(new PreparingOrderState());
                break;
            case 'ready':
                $this->setState(new ReadyOrderState());
                break;
            case 'completed':
                $this->setState(new CompletedOrderState());
                break;
            case 'cancelled':
```



```
                $this->setState(new CancelledOrderState());
                break;
            default:
                $this->setState(new PendingOrderState());
        }
    }
}
```

Figure 4.2.1 State Pattern in Order State System

Explanation of State Pattern Implementation:

The State classes are used as contexts to manage different order status transitions and their specific behaviors. The **State Pattern** helps handle complex order lifecycle management by encapsulating state-specific logic, so the Order model doesn't have to manage all the transition rules. This makes the system easier to update and change later.

- **PendingOrderState**: This state handles initial order operations including payment processing, order cancellation, but restricts direct confirmation without payment.
- **PreparingOrderState**: This state controls orders in preparation by allowing completion to ready status but preventing cancellations once cooking has started.
- **ReadyOrderState**: This state handles completed orders ready for pickup/delivery by allowing final completion but restricting cancellations.
- **CompletedOrderState**: This state manages finished orders by preventing any further modifications or state changes.
- **CancelledOrderState**: This state handles cancelled orders by preventing any operations on cancelled orders.

The state pattern manages the proper order lifecycle and status transitions while the system is running, so controllers don't need to deal with complex state validation rules. This also makes it easy to add new order states to the system later on.

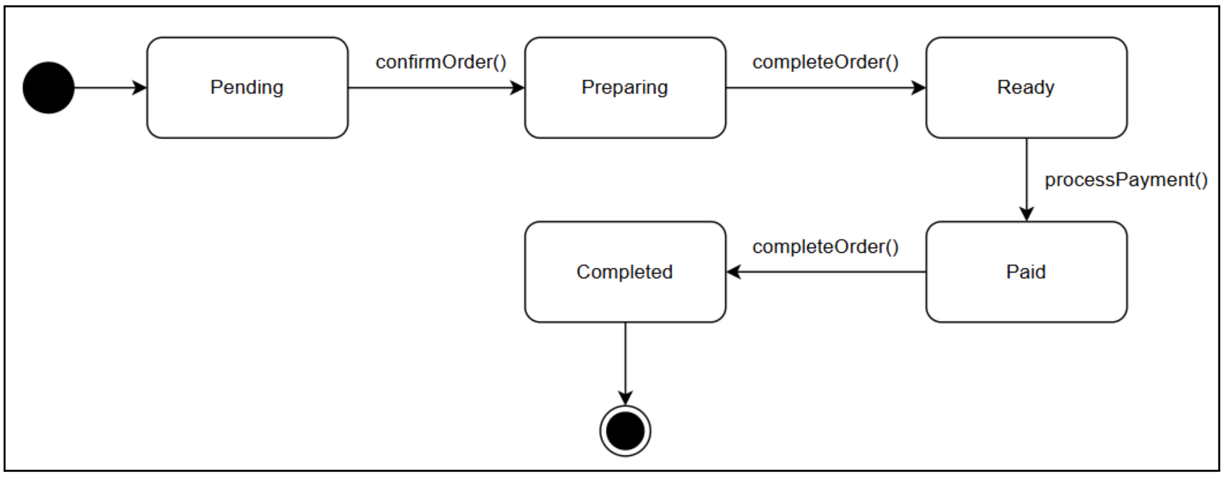


Figure 4.2.2: State Diagram

Explanation of the State Diagram:

1. Order State Flow (State Design Pattern):
This is the **State Pattern** that help to manage the order through different states like **Pending**, **Preparing**, **Ready**, **Paid**, and **Completed**. It use some functions like **confirmOrder()**, **completeOrder()**, and **processPayment()**. The system follow this pattern to make sure the order move smooth from one state to another without too much confusion.

2. Pending:
In **Pending**, the customer just put the order but the system didn't do anything yet. The order just **sit there**. It need to be confirmed before it can go to **Preparing**. **confirmOrder()** is the function that makes it move to **Preparing** state, then the kitchen can start the cooking or whatever.

3. Preparing:
When the order is in **Preparing**, it mean the kitchen is busy making the food or drink. The food not ready yet, so customer cannot pick up. The system use **completeOrder()** to say when the food is ready. It also use **calculatePrepTime()** to tell how long it gonna take to finish the food. After that, it move to **Ready** when the food is done.

4. Ready:
Ready is when the food is done and it sit there waiting to be pick up or served to the customer. The system can use **serveOrder()** to hand the food to the customer. It also use **notifyCustomer()** to tell the customer that their food is ready. Then, it wait for the customer to **pay**, and once they pay, the system move it to **Paid** state.

5. Paid:
In **Paid**, the customer already pay for the order. The food already been served. It move to **Completed** state after payment is done, meaning the order is fully finished.

6.Completed:
In the **Completed** state, the order done. The customer already paid, the food is eaten, and nothing else need to be done with the order. This is the last state, and no more action can be done.

How it Works:

- The **user** (restaurant staff or system) interact with the **Order State Management** pattern.
- The **State Pattern** helps move the order from **Pending** to **Preparing**, then to **Ready**, **Paid**, and finally **Completed**.
- **Pending** is where the order is waiting to be confirmed.
- **Preparing** is when the food is cooking and not yet ready.
- **Ready** is when the food is finished and waiting for customer to pick up.
- **Paid** is when the customer has paid, and the order is complete.
- **Completed** is the last state where everything is done, and no further steps are needed.

5. Software Security

5.1 Potential Threat/Attack

Threat 1: Session-Based Cart Manipulation & Price Tampering

What is the vulnerability?
The problem happen because the **cart system** store the **pricing info** in the session that user can control, without checking it with the real prices in the **database**. So, this let attackers change the data in the session, like:

- Changing **item prices** to very low amounts (e.g., $0.01) or even negative values.
- Modifying **quantities** of items beyond the allowed limits.
- Adding **items** to the cart that either do not exist or are inactive.
- **Bypassing** restrictions in the business logic (rules) of the system.

How can it be exploited?

```
<?php
// In CartController.php - Line 61-66, the cart stores user-controlled data:
$cart[$cartKey] = [
    'menu_item_id' => $menuItem->id,
    'name' => $menuItem->name,
    'price' => $menuItem->price,  // Stored in session - can be tampered
    'formatted_price' => $menuItem->formatted_price,
    'quantity' => $request->quantity,
    'special_instructions' => $request->special_instructions,
];
```

Here’s how an attacker might take advantage of this vulnerability:
- A customer normally adds an item to the cart, say **RM 25.00**.
- The attacker **intercepts the session** (using tools like browser **dev tools**) and changes the **price** in the cart to **RM 0.01**.
- The attacker then proceeds to **checkout** with the manipulated price, leading to **incorrect total calculations** based on the altered session data.

Where the issue exists:
- In the CartController::updateQuantity(), the system **trusts the session data** without validating it against the real prices.
- In OrderController::store(), the system directly uses the cart's tampered prices for **order creation**, **without checking** the prices on the server side.

Why it's a problem:
This issue allows **price manipulation** and **bypassing business rules**, potentially resulting in **loss of revenue** and **incorrect orders**.

Threat 2: Order History & Profile Information Disclosure

What is the vulnerability?
The issue happens because the **order history** and **profile pages** use the **table_id** from **sessions** without checking if it's valid. This allows an attacker to **view other customers' sensitive information** by manipulating the session data.

How can it be exploited?

Order History Page Attack:

```
<?php
// In OrderHistoryController.php - Line 18-31
$tableId = Session::get('table_id');  //  User-controlled
$tableToken = Session::get('table_token');

$query = Order::where('table_id', $tableId)  //  Direct use of user input
    ->with(['orderItems.menuItem', 'orderItems.menuItem.category', 'user']);
```

Profile Page Attack:

```
<?php
// In ProfileController.php - Line 141-144
$favoriteItems = OrderItem::whereHas('order', function ($query) use ($tableId) {
    $query->where('table_id', $tableId);  // User-controlled tableId
})
```

Attack Steps:

1. Scan legitimate QR code for Table-A1
2. Navigate to order history page
3. Use browser dev tools to modify session: table_id = "Table-B5"
4. Refresh order history page → now sees Table-B5's complete order history
5. Navigate to profile page → sees Table-B5's dining statistics and preferences
6. Can access order confirmation pages for other tables' orders

What Sensitive Data Gets Exposed:

1. Complete order history of other tables
2. Customer names and email from orders
3. Order totals and payment status
4. Popular menu items and ordering patterns
5. Special dietary instructions and preferences
6. Order preparation times and dining habits

Order Confirmation Page Vulnerability:

```
<?php
// In OrderController.php confirmation method
private function canAccessOrder(Order $order): bool
{
    $sessionTableId = Session::get('table_id');
    return $order->table_id === $sessionTableId;  // Bypassable with session manipulation
}
```

5.2 Secure Coding Practice

Threat 1: Session-Based Cart Manipulation & Price Tampering

Secure Practice #1: Database-First Validation
Implementation:

```
<?php
// In MenuController.php - Line 70
$menuItem = MenuItem::findOrFail($request->menu_item_id); // Always fetch fresh data
```

What it does:

- findOrFail(): Queries the database directly to get current item details (price, availability, etc.)
- Fresh Data Guarantee: Never trusts user-submitted data for critical information like prices
- 404 on Invalid ID: Automatically returns 404 error if item doesn't exist

Why it's secure:

- Prevents price tampering by using authoritative database values
- Stops stale data attacks where users manipulate cached information
- Ensures availability checks against current menu status
- Provides audit trail of what was actually ordered vs. what user claimed

Secure Practice #2: Quantity Limits Enforcement
Implementation:

```
<?php
// In CartController.php - Lines 56-59
$request->validate([
    'cart_key' => 'required|string',
    'quantity' => 'required|integer|min:1|max:10', // Business rule enforcement
]);
```

What it does:

- Minimum Quantity (1): Prevents zero or negative quantity orders
- Maximum Quantity (10): Stops bulk ordering abuse and inventory hoarding
- Integer Type: Ensures whole numbers only (no fractional items)

Why it's secure:

- Prevents resource exhaustion attacks through massive orders
- Stops negative quantity exploits that could break calculations
- Enforces business rules consistently across all operations
- Protects inventory integrity from unrealistic orders

Secure Practice #3: Cart Item Existence Validation

Implementation:

```
<?php
// In CartController.php (Cart Page Module) - Lines 63-82
$cart = Session::get('cart', []);
$cartKey = $request->cart_key;

if (isset($cart[$cartKey])) {
    $cart[$cartKey]['quantity'] = $request->quantity;
    Session::put('cart', $cart);

    // Calculate new totals
    $subtotal = array_sum(array_map(function($item) {
        return $item['price'] * $item['quantity'];
    }, $cart));

    // ... return success response
}

return response()->json([
    'success' => false,
    'message' => 'Item not found in cart'  // Prevents manipulation of non-existent items
], 404);
```

What it does:

- Cart Key Validation: Verifies that the item being modified actually exists in the cart
- Existence Check: Uses isset($cart[$cartKey]) to ensure cart item is legitimate
- Error Response: Returns 404 if someone tries to manipulate non-existent cart items
- Controlled Updates: Only allows modifications to valid, existing cart entries

Why it's secure:

- Manipulation Prevention: Stops attackers from adding fake cart items through direct API calls
- Data Integrity: Ensures only legitimate cart items can be updated or modified
- Attack Surface Reduction: Limits manipulation to items that were legitimately added to cart
- Business Logic Protection: Prevents creation of phantom cart items with manipulated prices

Threat 2: Order History & Profile Information Disclosure

Secure Practice #1: Session Validation
Implementation:

```
<?php
// In OrderHistoryController.php - Lines 21-25
// Validate table session
if (!$tableId || !$tableToken) {
    return redirect('/')->with('error', 'Please scan a table QR code first.');
}
```

What it does:

- Existence Check: Verifies both table ID and token are present in session
- Graceful Redirect: Sends users to scan QR code instead of showing errors
- Session Dependency: Ensures users follow proper authentication flow

Why it's secure:

- Prevents direct access to sensitive pages without proper authentication
- Stops session bypass attempts by requiring valid session state
- Enforces workflow security by requiring QR code scanning
- Provides user guidance instead of cryptic error messages

Secure Practice #2: Database Transaction Safety
Implementation:

```
<?php
// In OrderController.php - Lines 134-184
try {
    DB::beginTransaction();

    // Create order using the new table-based method
    $order = Order::createForTable($tableId, Auth::id(), $cart);

    // Add notes and special instructions if provided
    if ($request->notes || $request->special_instructions) {
        $order->update([
            'notes' => $request->notes,
            'special_instructions' => $request->special_instructions,
        ]);
    }

    // Clear cart after successful order creation
    Session::forget('cart');

    DB::commit();
} catch (\Exception $e) {
    DB::rollBack();
    // ... error handling
}
```

What it does:

- Atomic Operations: All database changes happen together or not at all

- Rollback on Failure: If any step fails, all changes are undone
- Consistent State: Database never left in partial/broken state
- Session Cleanup: Cart cleared only after successful order creation

Why it's secure:
- Prevents data corruption from partial updates
- Stops race condition exploits during concurrent operations
- Ensures financial integrity (orders and payments stay synchronized)
- Provides error recovery without leaving orphaned records

Secure Practice #3: Table Ownership Verification
Implementation:

```
<?php
// In OrderHistoryController.php (Order History Module) - Lines 101-115
// Get table information from session
$tableId = Session::get('table_id');
$tableToken = Session::get('table_token');

// Validate table session
if (!$tableId || !$tableToken) {
    return redirect('/')->with('error', 'Please scan a table QR code first.');
}

// Ensure the order belongs to the current table
if ($order->table_id !== $tableId) {
    abort(403, 'Unauthorized access to order');  // Blocks cross-table access
}

$order->load(['orderItems.menuItem', 'orderItems.menuItem.category', 'user']);
```

What it does:
- Session Validation: Checks both table ID and token are present in session
- Ownership Verification: Compares $order->table_id with session $tableId
- Access Control: Returns 403 Forbidden if table IDs don't match
- Secure Loading: Only loads order details after authorization passes

Why it's secure for Order History Module:
- Cross-Table Prevention: Stops users from viewing other tables' order history
- Session Integrity: Validates that session data corresponds to actual order ownership
- Information Leak Prevention: Blocks unauthorized access to sensitive customer data
- Authorization Before Data: Checks permissions before loading any sensitive information

6. Web Service

6.1 Ordering and Cart Module Web Services

6.1.1 Add Item to Cart

Webservice Mechanism

Key	**Description**
Protocol	RESTFUL
Function Description	Adds menu item to shopping cart with quantity and instructions
Source Module	Cart Management
Target Module	Menu Module, Order Processing, Session Management
URL	POST /menu/add-to-cart
Function Name	addToCart

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
menu_item_id	Integer	Mandatory	ID of the menu item	Can only contain numbers
quantity	Integer	Mandatory	Quantity of items	Range: 1-10
special_instructions	String	Optional	Customer special requests	Max 500 characters

Web Services Response Parameter (Consume):

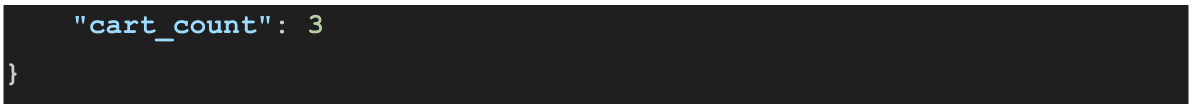
Field Name	Field Type	Mandatory/Optional	Description	Format
success	Boolean	Mandatory	Status of the operation	true/false
message	String	Mandatory	Response message	Success/error message
cart_count	Integer	Mandatory	Total items in cart	Updated cart count

Example Request:

```
POST /menu/add-to-cart
{
    "menu_item_id": 1,
    "quantity": 2,
    "special_instructions": "Extra spicy please"
}
```

Example Request:

```
{
    "success": true,
    "message": "Item added to cart successfully!",
```



```
    "cart_count": 3
}
```

Figure 6.1.1: POST /menu/add-to-cart API results

6.1.2 Get Cart Summary

Webservice Mechanism

Key	Value
Protocol	RESTFUL
Function Description	Retrieves cart summary with totals and item count
Source Module	Cart Management
Target Module	Order Processing, Checkout Module, UI Updates
URL	GET /menu/cart-summary
Function Name	getCartSummary

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
-	-	-	No parameters required	Session-based

Web Services Response Parameter (Consume):

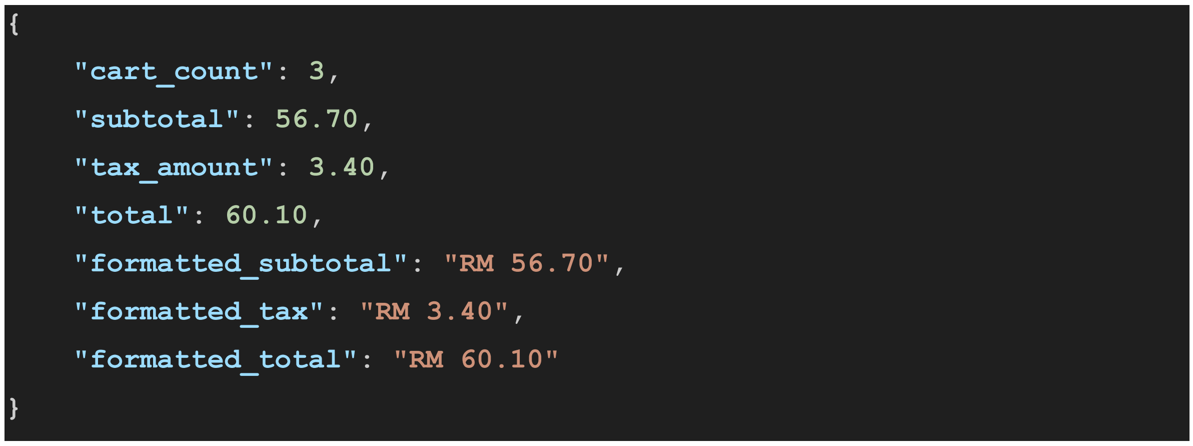
Field Name	Field Type	Mandatory/Optional	Description	Format
cart_count	Integer	Mandatory	Total items in cart	Number of items

subtotal	Decimal	Mandatory	Subtotal amount	Decimal format (XX.XX)
tax_amount	Decimal	Mandatory	Tax amount (6% SST)	Decimal format (XX.XX)
total	Decimal	Mandatory	Total amount	Decimal format (XX.XX)
formatted_total	Decimal	Mandatory	Formatted total with currency	Currency format (RM XX.XX)

Example Request:

```
GET /menu/cart-summary
```

Example Response:



```
{
    "cart_count": 3,
    "subtotal": 56.70,
    "tax_amount": 3.40,
    "total": 60.10,
    "formatted_subtotal": "RM 56.70",
    "formatted_tax": "RM 3.40",
    "formatted_total": "RM 60.10"
}
```

Figure 6.1.2: GET /menu/cart-summary API results

6.1.3 Update Cart Item

Webservice Mechanism

Key	Value
Protocol	RESTFUL

Function Description	Updates quantity of items in cart or removes items
Source Module	Cart Management
Target Module	Session Management, UI Updates, Order Processing
URL	POST /menu/update-cart
Function Name	updateCart

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
cart_key	String	Mandatory	Unique identifier for cart item	Cart item key
quantity	Integer	Mandatory	New quantity (0 to remove)	Range: 0-10

Web Services Response Parameter (Consume):

Field Name	Field Type	Mandatory/Optional	Description	Format
success	Boolean	Mandatory	Status of the operation	true/false
message	String	Mandatory	Response message	Success/error message

cart_count	Integer	Mandatory	Updated total items in cart	Updated cart count

Example Request:

```
POST /menu/update-cart
{
    "cart_key": "1_abc123",
    "quantity": 3
}
```

Example Response:

```
{
    "success": true,
    "message": "Cart updated successfully!",
    "cart_count": 5
}
```

Figure 6.1.3: POST /menu/update-cart API results

6.1.4 Remove Item from Cart

Webservice Mechanism

Key	Value
Protocol	RESTFUL
Function Description	Removes specific item from shopping cart
Source Module	Cart Management
Target Module	Session Management, UI Updates

URL	POST /menu/remove-from-cart
Function Name	removeFromCart

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
cart_key	String	Mandatory	Unique identifier for cart item	Cart item key

Web Services Response Parameter (Consume):

Field Name	Field Type	Mandatory/Optional	Description	Format
success	Boolean	Mandatory	Status of the operation	true/false
message	String	Mandatory	Response message	Success/error message
cart_count	Integer	Mandatory	Updated total items in cart	Updated cart count

Example Request:

```
POST /menu/remove-from-cart
{
    "cart_key": "1_abc123"
}
```

Example Response:

```
{
    "success": true,
    "message": "Item removed from cart successfully!",
    "cart_count": 2
}
```

Figure 6.1.4: POST /menu/remove-from-cart API results Figure

6.1.5 Clear Cart

Webservice Mechanism

Key	Value
Protocol	RESTFUL
Function Description	Removes specific item from shopping cart
Source Module	Cart Management
Target Module	Session Management, UI Updates
URL	POST /menu/remove-from-cart
Function Name	removeFromCart

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
cart_key	String	Mandatory	Unique identifier for cart item	Cart item key

Web Services Response Parameter (Consume):

Field Name	Field Type	Mandatory/Optional	Description	Format
success	Boolean	Mandatory	Status of the operation	true/false
message	String	Mandatory	Response message	Success/error message
cart_count	Integer	Mandatory	Updated total items in cart	Updated cart count

Example Request:

```
POST /menu/clear-cart
```

Example Response:

```
{
    "success": true,
    "message": "Cart cleared successfully!"
}
```

6.1.5: POST /menu/clear-cart API results

6.1.6 Create Order

Webservice Mechanism

Key	Value
Protocol	RESTFUL
Function Description	Creates order from cart items with customer details
Source Module	Order Management
Target Module	Cart Module, Payment Processing, Kitchen Management
URL	POST /order
Function Name	store

Web Services Request Parameter (Provide):

Field Name	Field Type	Mandatory/Optional	Description	Format
notes	String	Optional	Order notes	Max 500 characters
special_instructions	String	Optional	Special cooking instructions	Max 500 characters

Web Services Response Parameter (Consume):

Field Name	Field Type	Mandatory/Optional	Description	Format
success	Boolean	Mandatory	Status of the operation	true/false
redirect_url	String	Mandatory	Redirect URL after order creation	URL path
order_number	String	Mandatory	Generated order number	Format: T001-O001

Example Request:

```
POST /order
{
    "notes": "Table 5 order",
    "special_instructions": "No onions in any dish"
}
```

Example Response:

```
{
    "success": true,
    "redirect_url": "/order/123/confirmation",
    "order_number": "T001-O003"
}
```

Figure 6.1.6: POST /order API results

7. Index

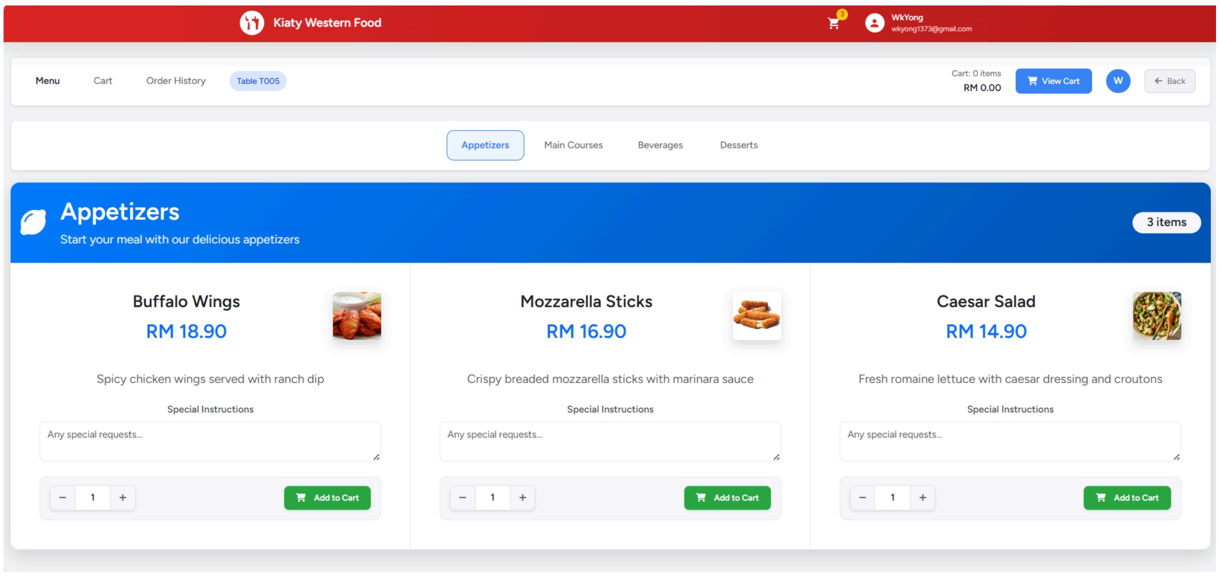


Figure 7.1 Menu Page

1. Source Path: '/menu/browse' (Customer Interface → Menu Browsing)
2. Application Route: Route::get('/menu/browse', [MenuController::class, 'browse'])
3. Controller: app\Http\Controllers\MenuController.php
4. View: resources\views\menu\browse.blade.php

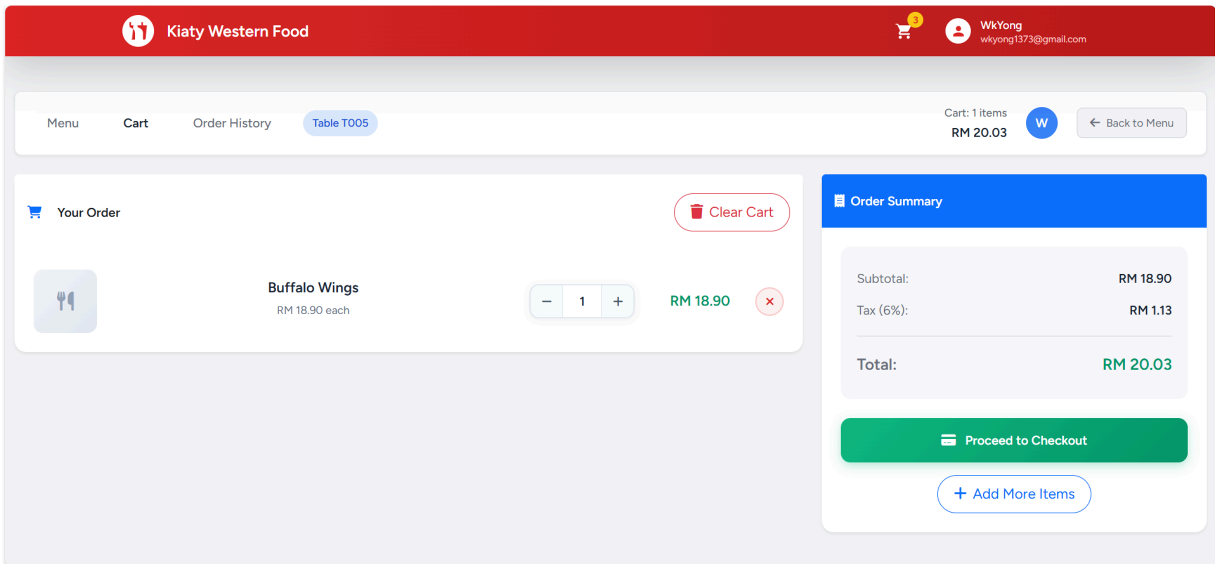
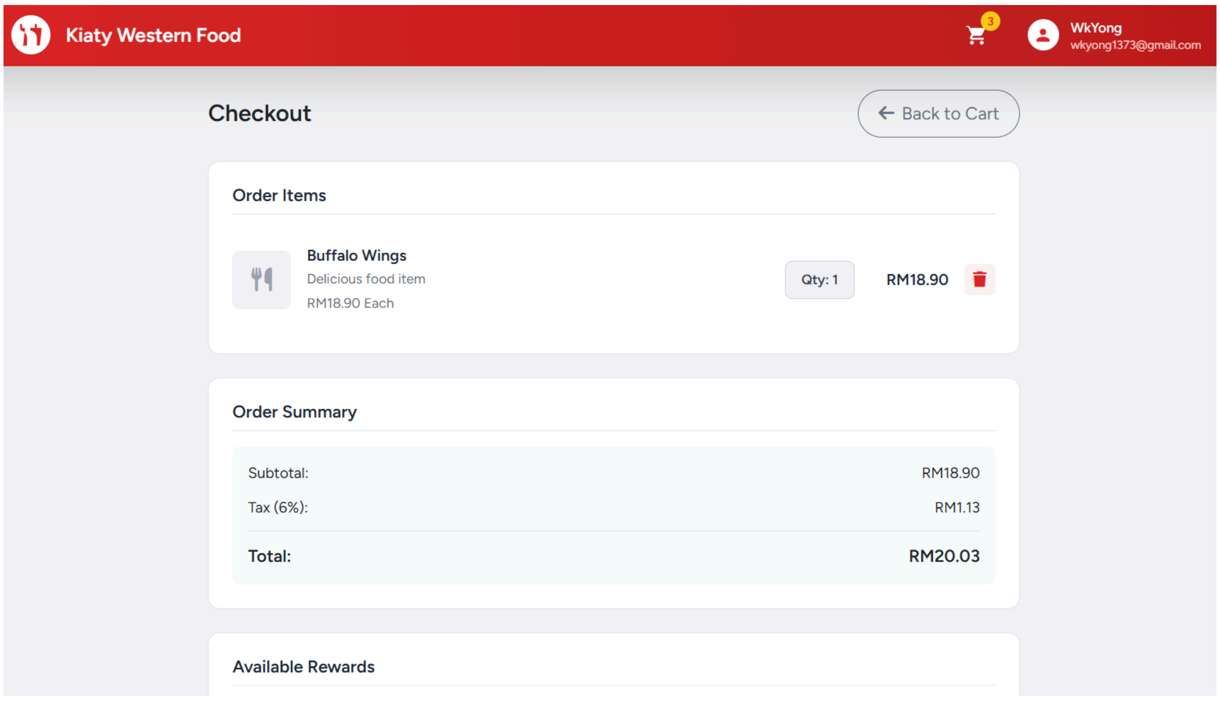


Figure 7.2 Cart Page

1. Source Path: '/cart' (Customer Interface → Shopping Cart)
2. Application Route: Route::get('/cart', [CartController::class, 'index'])
3. Controller: app\Http\Controllers\CartController.php
4. View: resources\views\cart\index.blade.php


Kiaty Western Food

WkYong
wkyong1373@gmail.com

Checkout

← Back to Cart

Order Items

Buffalo Wings
Delicious food item
RM18.90 Each

Qty: 1 RM18.90

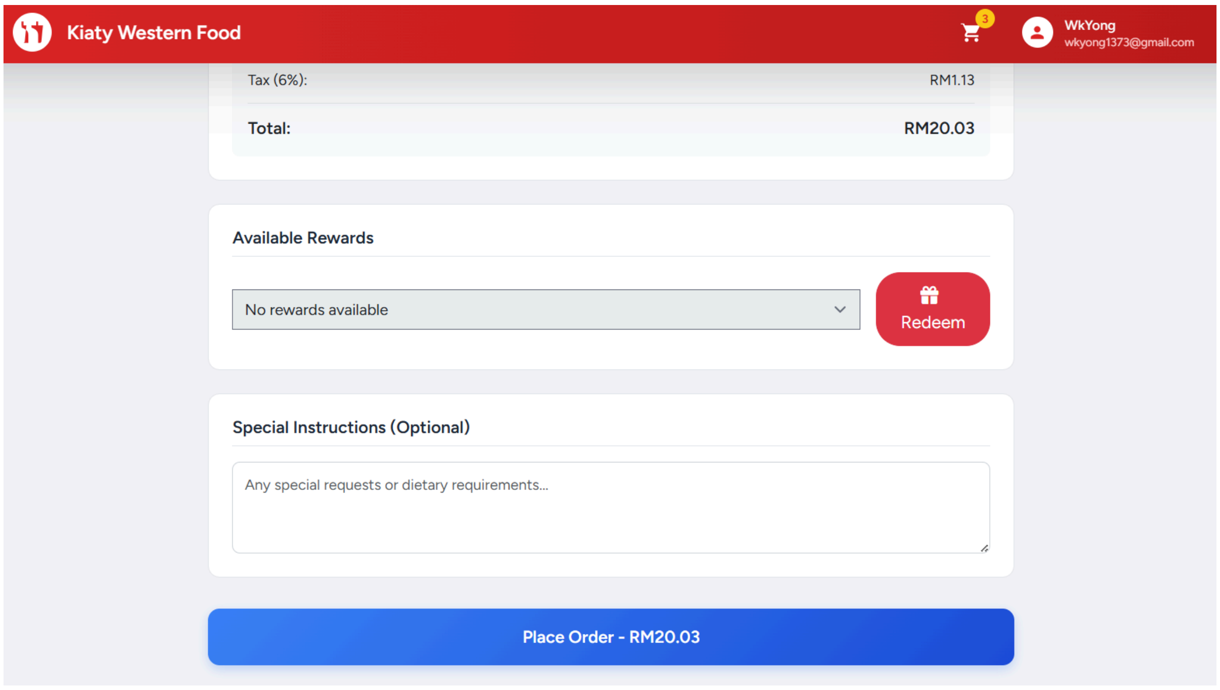
Order Summary

Subtotal:	RM18.90
Tax (6%):	RM1.13
Total:	RM20.03

Available Rewards

Figure 7.3 Checkout Page

1. Source Path: '/order/create' (Customer Interface → Order Creation)
2. Application Route: Route::get('/order/create', [OrderController::class, 'create'])
3. Controller: app\Http\Controllers\OrderController.php
4. View: resources\views\order\create.blade.php


Kiaty Western Food

WkYong
wkyong1373@gmail.com

Tax (6%):	RM1.13
Total:	RM20.03

Available Rewards

No rewards available

Redeem

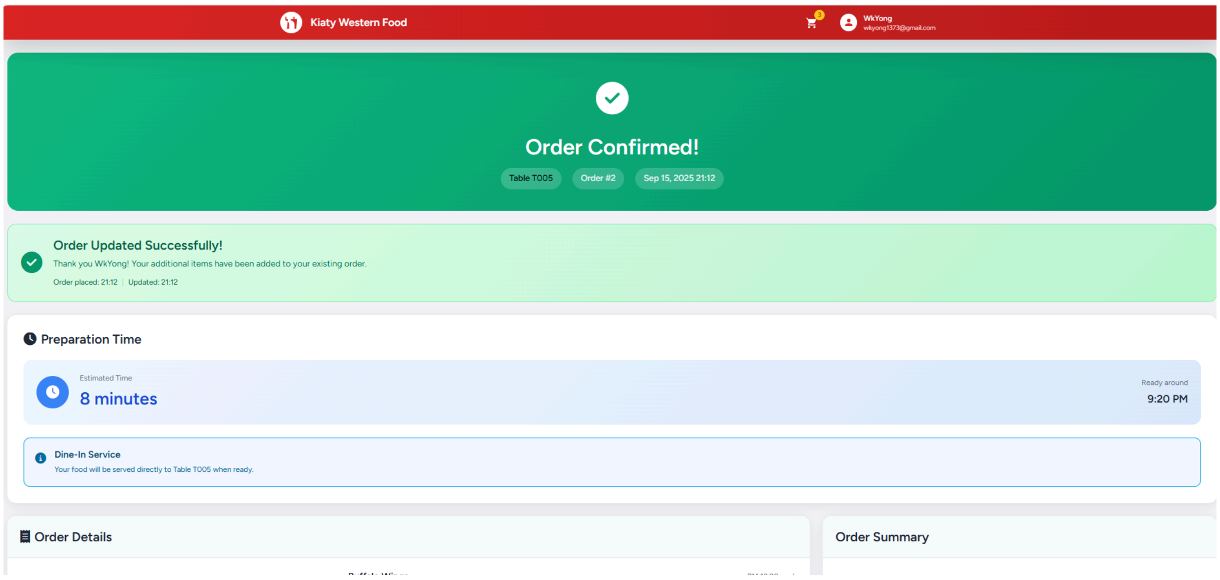
Special Instructions (Optional)

Any special requests or dietary requirements...

Place Order - RM20.03

Figure 7.4 Checkout Page

1. Source Path: '/order/store' (Order Processing → Place Order)
2. Application Route: Route::post('/order/store', [OrderController::class, 'store'])
3. Controller: app\Http\Controllers\OrderController.php
4. View: resources\views\order\create.blade.php


Kiaty Western Food
WkYong
wkyong1373@gmail.com

Order Confirmed!
Table T005 | Order #2 | Sep 15, 2025 21:12

Order Updated Successfully!
Thank you WkYong! Your additional items have been added to your existing order.
Order placed: 21:12 | Updated: 21:12

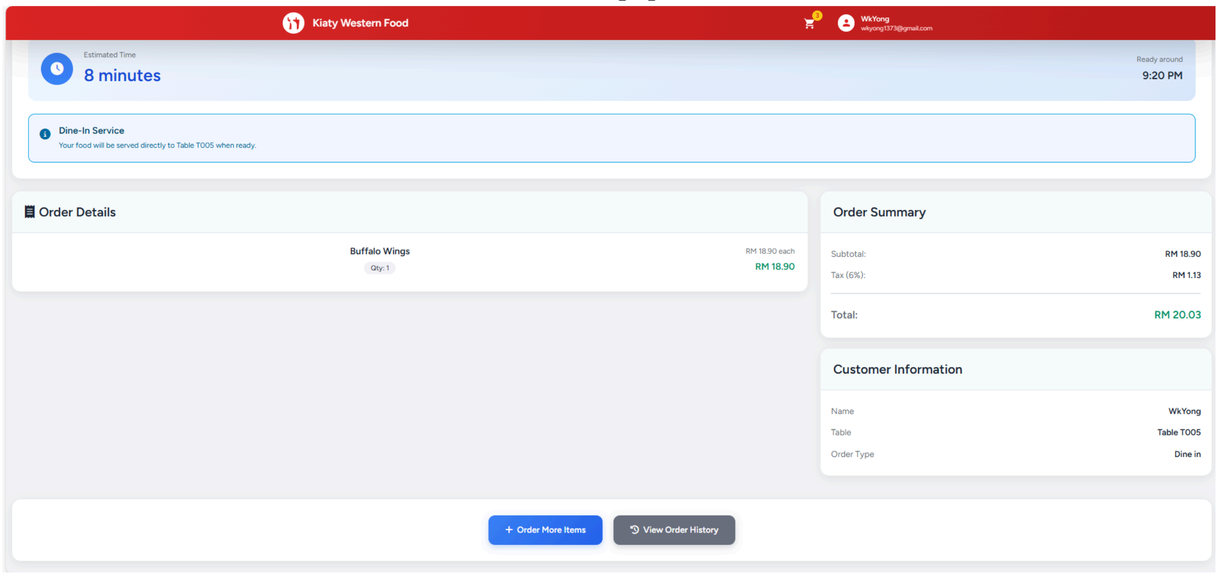
Preparation Time
Estimated Time
8 minutes
Ready around
9:20 PM

Dine-In Service
Your food will be served directly to Table T005 when ready.

Order Details
Order Summary

Figure 7.5 Order Confirmed Page

1. Source Path: '/order/confirmation/{order}' (Order Management → Confirmation Display)
2. Application Route: Route::get('/order/confirmation/{order}', [OrderController::class, 'confirmation'])
3. Controller: app\Http\Controllers\OrderController.php
4. View: resources\views\order\confirmation.blade.php


Kiaty Western Food
WkYong
wkyong1373@gmail.com

Estimated Time
8 minutes
Ready around
9:20 PM

Dine-In Service
Your food will be served directly to Table T005 when ready.

Order Details
Buffalo Wings
Qty: 1
RM 18.90 each
RM 18.90

Order Summary

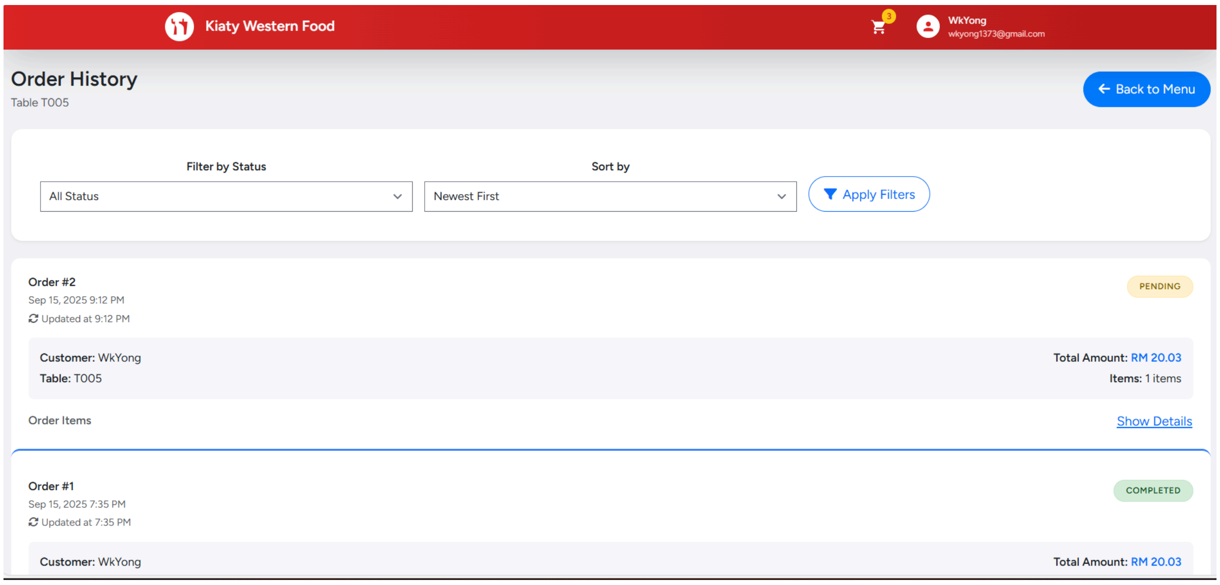
Subtotal:	RM 18.90
Tax (6%):	RM 1.13
Total:	RM 20.03

Customer Information

Name	WkYong
Table	Table T005
Order Type	Dine in

Figure 7.6 Order Confirmed Page

1. Source Path: '/order/confirmation/{order}' (Order Management → Confirmation Display)
2. Application Route: Route::get('/order/confirmation/{order}', [OrderController::class, 'confirmation'])
3. Controller: app\Http\Controllers\OrderController.php
4. View: resources\views\order\confirmation.blade.php



Order History
Table T005

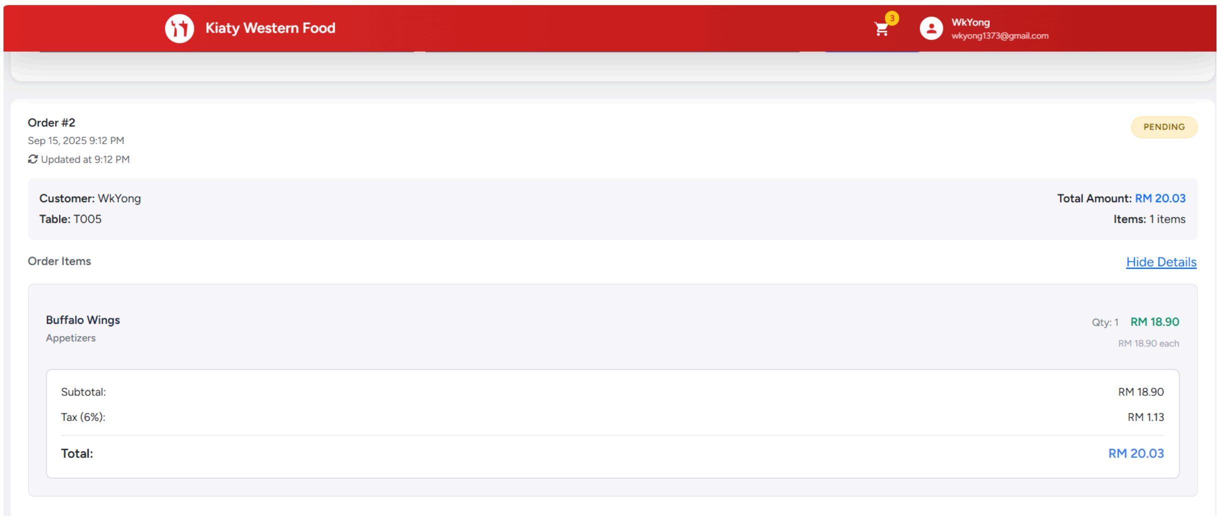
Filter by Status: All Status
Sort by: Newest First

Order #2
Sep 15, 2025 9:12 PM
Updated at 9:12 PM
Customer: WkYong
Table: T005
Total Amount: RM 20.03
Items: 1 items
Order Items

Order #1
Sep 15, 2025 7:35 PM
Updated at 7:35 PM
Customer: WkYong
Total Amount: RM 20.03

Figure 7.7 Order History Page

1. Source Path: '/order/history' (Customer Interface → Order History)
2. Application Route: Route::get('/order/history', [OrderHistoryController::class, 'index'])
3. Controller: app\Http\Controllers\OrderHistoryController.php
4. View: resources\views\order\history.blade.php



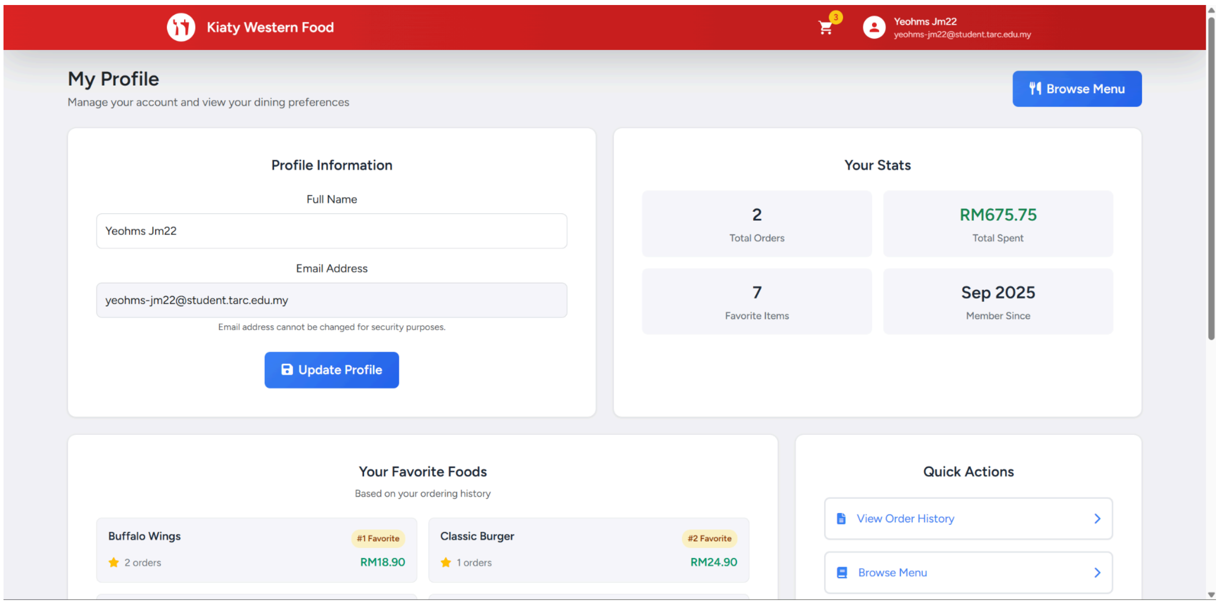
Order #2
Sep 15, 2025 9:12 PM
Updated at 9:12 PM
Customer: WkYong
Table: T005
Total Amount: RM 20.03
Items: 1 items
Order Items

Buffalo Wings
Appetizers
Qty: 1 RM 18.90
RM 18.90 each

Subtotal: RM 18.90
Tax (6%): RM 1.13
Total: RM 20.03

Figure 7.8 Order History Page

1. Source Path: '/order/history/{order}' (Order Management → Order Details)
2. Application Route: Route::get('/order/history/{order}', [OrderHistoryController::class, 'show'])
3. Controller: app\Http\Controllers\OrderHistoryController.php
4. View: resources\views\order\details.blade.php



Kiaty Western Food

Yeohms Jm22
yeohms-jm22@student.tarc.edu.my

My Profile

Manage your account and view your dining preferences

Browse Menu

Profile Information

Full Name: Yeohms Jm22

Email Address: yeohms-jm22@student.tarc.edu.my

Email address cannot be changed for security purposes.

Update Profile

Your Stats

2 Total Orders	RM675.75 Total Spent
7 Favorite Items	Sep 2025 Member Since

Your Favorite Foods

Based on your ordering history

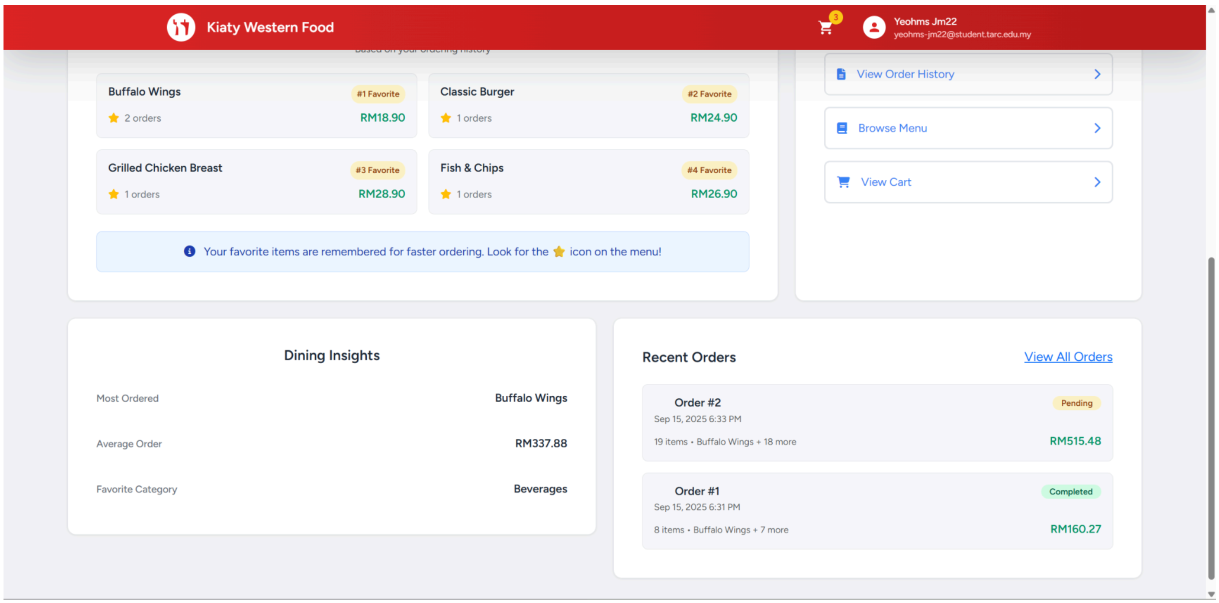
Buffalo Wings	#1 Favorite	2 orders	RM18.90
Classic Burger	#2 Favorite	1 orders	RM24.90

Quick Actions

- View Order History
- Browse Menu

Figure 7.9 Profile Page

1. Source Path: '/profile' (Customer Interface → Profile Management)
2. Application Route: Route::get('/profile', [ProfileController::class, 'show'])
3. Controller: app\Http\Controllers\ProfileController.php
4. View: resources\views\profile\show.blade.php



Kiaty Western Food

Yeohms Jm22
yeohms-jm22@student.tarc.edu.my

Buffalo Wings	#1 Favorite	2 orders	RM18.90
Classic Burger	#2 Favorite	1 orders	RM24.90
Grilled Chicken Breast	#3 Favorite	1 orders	RM28.90
Fish & Chips	#4 Favorite	1 orders	RM26.90

Your favorite items are remembered for faster ordering. Look for the ⭐ icon on the menu!

- View Order History
- Browse Menu
- View Cart

Dining Insights

Most Ordered	Buffalo Wings
Average Order	RM337.88
Favorite Category	Beverages

Recent Orders — View All Orders

Order #2	Sep 15, 2025 6:33 PM	19 items • Buffalo Wings + 18 more	Pending	RM515.48
Order #1	Sep 15, 2025 6:31 PM	8 items • Buffalo Wings + 7 more	Completed	RM160.27

Figure 7.10 Profile Page

1. Source Path: '/profile/edit' (Customer Interface → Profile Settings)
2. Application Route: Route::get('/profile/edit', [ProfileController::class, 'edit'])
3. Controller: app\Http\Controllers\ProfileController.php
4. View: resources\views\profile\edit.blade.php