

Real-time web food ordering system  
implemented with preparation  
process optimisation using time  
recommendation.

Kuay Tang Zheng  
19022570

UXCFXK-30-3  
Digital Systems Project

# Abstract

The number of seated restaurant diners due to COVID-19 in the United Kingdom in 2022 has increased gradually compared to 2021 and 2020 (Statista, 2021). The outbreak of Covid-19 has pushed many fast-food takeaways and restaurants to a higher level. Traditional ordering methods such as handwriting or manual collection time selection are inefficient if there are many customers and time allocations have no regular pattern.

High-quality services and fewer waiting times are vital points to improving customer satisfaction. The Restaurant web application allows users to read the menu through mobile devices without touching the physical menu that many other customers have used.

In this paper, the proposed system will focus on the algorithm that compares the new order contents product with existing orders content and calculates the best time slot with the product's preparation time and categories—shortening the time consumption for each ordering process and improving customer service.

# Acknowledgements

I would first like to offer thanks to my supervisor, Adam Gorine, for reviewing and giving suggestion throughout the project.

I would also thanks to Dr Martin, without whom this project would not be initiated.

Lastly, I would like to thank my family who run the restaurant business at Wales, the project would not be success without their supports.

# Table of Contents

Abstract .....	1
Acknowledgements .....	2
Table of Contents.....	3
Table of Tables.....	6
Table of Figures.....	8
Introduction .....	10
Literature Review.....	10
Current Restaurant Issues .....	10
Recommended System .....	10
Process Scheduling and estimation.....	11
Restaurant Management/Ordering system .....	11
One-Stop Restaurant Service Application .....	12
Requirements .....	13
Requirements Gathering.....	13
Project Aims and Objective .....	13
Functional Requirements .....	14
Database .....	14
Staff.....	15
Kitchen .....	16
Non-functional Requirements.....	18
Methodology .....	19
Agile Methodology.....	19
Design .....	20
Development Tools .....	20
User stories .....	20
Use Case Diagram .....	21
Sequence diagram .....	22
Flowchart .....	23
Network architecture.....	24
State diagram.....	24
Order Taking Class diagrams .....	25
Database Design .....	26
Algorithm design.....	28

Time recommendation.....	28
Test Design.....	29
Implementation .....	29
Project Set-Up and Database (Sprint 1) .....	29
Database .....	29
Functional requirement .....	30
Landing Page (Spring 2).....	30
Promise .....	31
Function Requirement .....	32
Staff Dashboard Page (Sprint 3) .....	32
jQuery .....	32
AJAX .....	33
Backend .....	34
Functional Requirement .....	34
Product Management Page (Spring 4) .....	34
Functional Requirement .....	36
Staff Management Page (Spring 5) .....	36
Password Hashing .....	36
Functional Requirements .....	36
Order System Page (Sprint 6) .....	37
Session .....	38
Functional Requirements .....	38
Kitchen Display Page (Spring 7).....	39
Functional Requirements .....	40
Login and Authentication System (Sprint 8).....	40
Passport.js.....	40
Functional Requirements .....	41
Set Up Socket for bi-direction communication (Spring 9) .....	41
Socket.io .....	41
Functional Requirements .....	42
Implement Collection Time Recommendation (Sprint 9) .....	42
Functional Requirement .....	45
Testing (Sprint 10).....	45
Testing .....	45
Mocha.js .....	45
User Acceptance test .....	45

Testing Reflection .....	46
Project Evaluation .....	46
Research .....	46
Requirements .....	46
Design .....	47
Methodology .....	47
Implementation .....	47
Testing .....	47
Limitation and Future Work .....	48
Feedback .....	48
Conclusion .....	48
References .....	49
Appendix A: First Appendix .....	50
User Acceptance Test .....	50
Landing and Login page .....	50
Dashboard page .....	50
Product Management Page .....	50
Staff Management Page .....	51
Ordering System Page .....	51
Kitchen Display System Page .....	52

# Table of Tables

Table 1 shows the functional requirements for Database. ....	15
Table 2 shows the functional requirements for staff. ....	16
Table 3 shows the functional requirements for kitchen. Customer .....	17
Table 4 shows the functional requirements for customer. ....	18
Table 5 shows the non-functional requirements for the system. ....	19
Table 6 shows the staff user stories.....	21
Table 7 shows the restaurant owner user stories. ....	21
Table 8 shows the design for user acceptance tests. ....	29
Table 9 shows the requirements fulfilled in the sprint 1. ....	30
Table 10 shows the requirements fulfilled in sprint 2.....	32
Table 11 shows the requirements fulfilled in sprint 3.....	34
Table 12 shows the requirements fulfilled in sprint 4.....	36
Table 13 shows the requirements fulfilled in sprint 5.....	36
Table 14 shows the requirements fulfilled in sprint 6.....	38
Table 15 shows the requirements fulfilled in sprint 7.....	40
Table 16 shows the requirements fulfilled in sprint 8.....	41
Table 17 shows the requirements fulfilled in sprint 9.....	42
Table 18 shows the Order And similarity value example, the colour indicates different cooking types. .....	44
Table 19 shows the utensils swap time result for each collection time. ....	44
Table 20 shows the Order And similarity value example, the colour indicates different cooking types. .....	44
Table 21 shows the utensils swap time result for each collection time. ....	44
Table 22 shows the requirements fulfilled in sprint 10.....	45
Table 23 shows the result of the user acceptance tests. ....	46





# Table of Figures

Figure 1 Hybrid Scheduling (Nonaka et al.,2018) .....	11
Figure 2 Foody product scheduling features (Vindya et al.,2018).....	12
Figures 3 & 4 Interface of the restaurant management system (Singhal and Konguvel, 2022) .....	12
Figure 4 shows the Gantt chart for the project.....	19
Figure 5 Network Architecture diagram .....	24
Figure 6 shows State Diagram for Order Taking.....	25
Figure 7 shows the take Order Class Diagram.....	26
Figure 8 shows the Food Ordering System Database Design. ....	27
Figure 9 shows the Recommendations algorithm for the collection time state diagram. ....	28
Figure 10 shows the FOS Database table diagram using MySQL Workbench. ....	29
Figure 11 shows the MySQL create pool connection in Node.js .....	30
Figure 12 shows the environment file for the MySQL connection .....	30
Figure 13 shows the FOS Landing Page. ....	31
Figure 14 shows how the products and categories are fetched from the database rendered to the users. ....	31
Figure 15 shows how easily a can pug use for loop to assign a variable to an HTML tag while using JavaScript to declare a variable within it.....	32
Figure 16 shows the page where the staff will make most of the interaction with the orders.....	32
Figure 17 shows how jQuery does the event listener in the frontend. ....	33
Figure 18 & 18 shows the selection for managing the order. ....	33
Figure 19 shows how the GET request is sent to the server. ....	33
Figure 20 shows the controller for viewing a specific order .....	34
Figure 21 shows one of the order details examples.....	34
Figure 22 shows the product management page for FOS. ....	35
Figures 23 & 24 show the addition and modification page for the category. ....	35
Figure 24 and Figure 25 show the addition and modification page for the product. ....	35
Figure 25 shows the staff management page. ....	36
Figure 26 shows the registration process code.....	36
Figure 27 shows the Ordering System page with a categories list. ....	37
Figure 28 shows the Chicken category products list. ....	37
Figure 29 shows the collection time selection window. ....	37
Figure 30 shows the Order Cart module. ....	38
Figure 31 shows the session connection code. ....	38
Figure 32 shows the kitchen display page.....	39
Figure 33 shows the done option for Order number 9. ....	39
Figure 34 Login Page for staff and customers. ....	40
Figure 35 shows how the verification process. ....	40
Figure 36 shows how passport.js serialises and deserialise user. ....	40
Figure 37 shows how passport.js checks user authentication. ....	41
Figure 38 shows the usage of the middleware isAuthenticated. ....	41
Figure 39 shows the socket.io flowchart.....	42
Figure 40 cancellation request sent from dashboard page user. ....	42
Figure 41 if the user is accepted in Figure 40.....	42
Figure 42 if the user declined in Figure 40 .....	42
Figure 43 shows the function to generate a recommended time. ....	43

Figure 44 the algorithm to get the similarity value between orders.....	43
Figure 45 shows the alert recommends time for the user .....	44
Figure 46 shows the alert recommends time for the user. ....	44
Figure 47 shows the unit testing for the dashboard page.....	45
Figure 48 shows an integration tests example.....	45
Figure 49 number of unit tests and results. ....	45

# Introduction

Over the last few years, the restaurant industry has been heavily affected by the covid pandemic. Most customers are afraid of showing up in the public space where the virus can be spread swiftly through physical contact. Therefore, the fast-food takeaway has been booming compared to the last few years. A small-medium business that runs both restaurant and takeaway in Wales has difficulty handling the business.

This project is to build a web application by reviewing the past papers regarding the application for restaurant management and what algorithms could potentially increase the overall efficiency of the business.

## Literature Review

### Current Restaurant Issues

According to a study by Jelle (2018), the investigation result of 94,404 customers visiting a restaurant shows that longer waiting time impacts the customers reneging behaviour and lowers the restaurant's total revenue by 15%. This statistic has proven the importance of minimising the error time gap given to the customers for product pick-up can increase customer satisfaction. The restaurant's customers who were provided with the wrong collection time have to wait an additional time for their order. On the other hand, orders done too early than the collection time could also induce the customers to get their already cold products.

In another paper, Kanyan (2015) believes that the causes of slow service are insufficient staff, lack of equipment, and poor food schedule. Proposed the solution by optimising the processes in every procedure; this could be taking orders from customers, products preparation, and collection. In contrast, Christopher (1999) provides a solution to reducing the service-cycle time by improving quick production in the kitchen, quicker guest check, and faster table reset. He believes that this can eventually benefit restaurants by increasing utility with the production rate.

### Recommended System

Using a recommended system can enhance the user experience in the restaurant ordering system and potentially shorten the service cycle. However, this project does not have sufficient data to perform a recommendation algorithm; reviewing past papers that use less or no customer information is conducted in this section.

Several papers advocate it by implementing a recommendations system into their restaurant applications. For instance, Ashutosh et al. (2020) use the PCY algorithm to find the most frequently bought product sets; this algorithm can suggest to customers additional products that meet the set criteria, and only the frequency of the order data is used. Vindya et al. (2018) utilised this system by showing the recommended products in the first few rows of the category and providing a unique menu to each customer.

Collaborative filtering recommendation system - compares the current user and finds the most related users' group, suggesting the group favours products. Unfortunately, this approach might require user data composed of age, gender, ratings, and Ordering frequency, which is out of consideration for this project. However, Greg et al. (2003) described a solution on how they use

item-to-item collaborative filtering on Amazon.com without requiring other users' information. Instead of finding the similarity between users, item-to-item collaborative filtering checks the most-similar items purchased by a user. Regardless of the number and accuracy of reviews on a particular product, this recommendations process will mitigate most cold start issues stated by Bobadilla et al. (2012).

## Process Scheduling and estimation

For every customer ordered through either phone calls or devices for takeaway, the restaurant must provide the estimated waiting time or collection time to allow the customers to manage their time. However, the error or inaccuracy of the time supplied to the customers usually happens, causing the customers to wait longer for their order than they should. The foods get cold because the food preparation is finished too early. A scheduling algorithm that can produce an accurate estimation time is required to mitigate these situations.

Nonaka (2019) proposed three cooking and serving methods scheduling with the consideration of multiple same items at a time, allowing the algorithm to configure dynamically. It consists of a floor arrangement schedule. The products are prepared once the customer places the order with the First in First Out methodology—on the other hand, Cooking Scheduling schedules the order to the specific zone that is already expected to prepare the same product type. The final method compensates for both the above-described processes by configuring dynamically (Nonaka et al., 2018).

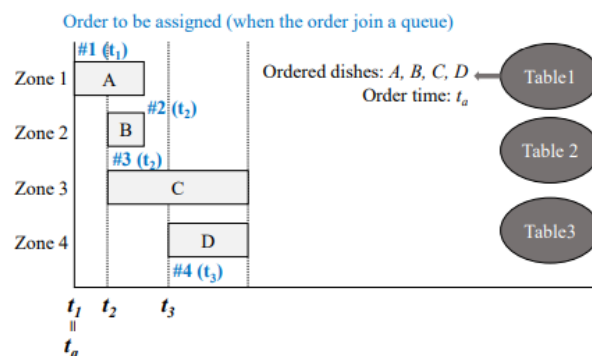


Figure 1 Hybrid Scheduling (Nonaka et al., 2018)

However, the proposed solution does not have the features to include the same categories type of products, decreasing the preparation time. Moreover, the paper's proposed algorithms have significant performance on efficiency but lack clarification on how the dynamic configurations will be performing (Nonaka et al., 2018).

## Restaurant Management/Ordering system

To avoid reinventing something that has been done in the past and increase the development efficiency of this project. This section will review the existing system—systems designed for restaurant management with additional functionality to enhance the overall service quality.

Foody (2018) composed multiple restaurant components into a single system. It allows customers to perform table reservations through mobile applications. It gives a unique experience to each customer by giving them the menu and food recommendation system based on their preference

(Vindya et al., 2018). Furthermore, Foody allows customers to check the product's ingredients through google; this can help customers with allergy restrictions.

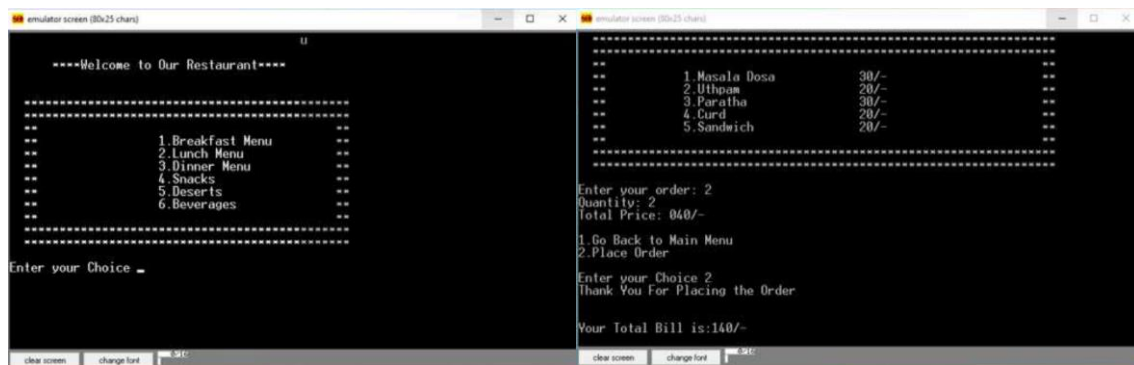
Assume	Chicken Pasta	Chicken Pizza
Preparation time	15 min	10min
Constant Value	10	7

qtn- quantity of order n  
PTn- preparation time of order nth order  
C.V n- constant value of n food item

Figure 2 Foody product scheduling features (Vindya et al.,2018)

However, the system task allocation and scheduling methods are not entirely designed for dynamic and real-time situations. For instance, the algorithm for scheduling only considered the quantity, preparation time, and contestant value of a specific product and the first in first out methodology is used.

Singhal and Konguvel (2022) proposed a restaurant management system whose objective is to slow the spread of COVID-19 between customers and employee staff by creating a digital menu. Furthermore, customers can check the product's availability and cancel the order. However, the proposed system is implemented with EMU8086 which the user interface and user experience are not friendly.



Figures 3 & 4 Interface of the restaurant management system (Singhal and Konguvel, 2022)

## One-Stop Restaurant Service Application

This paper has similar aims and objectives compared to the previous articles. On top of that, Kitsiri proposed application can manage customer queues, display the order status, and provide customers multilingual menu selection. Customers are allowed to take the line and will be automatically assigned a table when it is empty. If the customers choose not to show up, the staff can either arrange the specific customer to the back of the queue or delete it.

Even though the paper stated that the proposed application provided service to the customer and the restaurant service, customers are not allowed to do anything about ordering food but taking the queue and survey.

# Requirements

## Requirements Gathering

A face-to-face conversation with a restaurant owner, chef, and desk staff was conducted to get the most proper and realistic problem and requirement. Specific questions surrounded the conversation's subject:

- What system do you think can help you better in your position?
- What features or functionality would you like to include in the system?
- In what way you would consider using the system instead of using the traditional method?

The outcome of the conversation:

Restaurant Owner (Food pre-preparation):

- Kitchen display system that can sort based on the nearest order collection time without dealing with a chunk of paper orders.
- Highlighted or notified to prepare some products that required longer preparation time, such as frozen appetisers or steam products.
- Able to communicate directly to the front desk staff and make food available or unavailable to be ordered.

Kitchen Chef (Food processing):

- Allows the same category products to cluster together to minimise the cleaning and utensils swapping times.
- Limit the number of the same types of products needed to be prepared simultaneously, such as 8 Sweet and Sour Chicken, which is the maximum amount to be prepared in one process.

Staff:

- Able to manage products such as price, name, and category.
- Able to add additional ingredients to a customer's specific products.

## Project Aims and Objective

This project aims to increase the efficiency of the kitchen food preparation process and minimise the estimated waiting time provided to the customers while also including some basic functionality for the restaurant management system. The current target users are a Chinese restaurant in Wales named Bamboo House.

The project objectives are:

- Create a real-time ordering system for restaurants and takeaway.
- Create a QR for the users to enter the web application.
- Create an optimisation algorithm for collection time recommendation.
- Create a dashboard that shows all the orders status.
- Create a page for staff management.
- Create an order system for taking orders from customers.

- Create a kitchen display system for product preparation.

## Functional Requirements

Due to many functional requirements, four categories of functional requirements are created: database, staff, kitchen, and customer. These requirements are defined with different levels with the MoSCoW methodology. The reason to use the Moscow method is to determine the requirements' priority because there are optional functionalities to include in this project if the time allows. For the sake of clarification, justification of each MoSCoW is first provided:

**Must-Do:** This system is essential to all the must-do functional and non-functional requirements. This project is considered a failure if one of the most functional requirements is not met.

**Should-do:** All the should-do functional and non-functional requirements are optional to this system but should be implemented to improve the system's overall quality. These requirements should be viewed as additional points to the project.

**Could-do:** All the could-do functional and non-functional requirements will only be done after completing all the must-do and should-do.

These requirements are considered the third priority and implementing them is the best scenario for this project.

**Won't-do:** Due to the time constraints and complexity, all the won't-do functional and non-functional requirements will not be carried out in this project. These requirements can be considered a further improvement but are not needed in this project.

## Database

A database is required to keep the information needed for the restaurant management system. The following requirements are required to create, read, update, and delete (CRUD) database functionality.

No.	Description	Moscow	Justification
DF1	A database is needed to store all the data.	Must	Essential Feature
DF2	A database table to store all the information of products with CRUD functionality.	Must	Essential Feature
DF3	A database table to store all the categories and with CRUD functionality.	Must	Essential Feature
DF4	A database table stores the relationship between category,	Must	Essential Feature

	products, and CRUD functionality.		
<b>DF5</b>	A database table to store all the registered user information with CRUD functionality.	Must	Essential Feature
<b>DF6</b>	A database table to store all the sessions of the login staff and order cart.	Must	Keep track of user activity and identity.
<b>DF7</b>	A database table to store all the orders with CRUD functionality.	Must	Essential feature
<b>DF8</b>	A database table to store all the transactions with CRUD functionality.	Must	Essential feature
<b>DF9</b>	A database table to store all the order items with CRUD functionality.	Must	Record the order items make by the order data.
<b>DF10</b>	A database table to store the products' cooking type and CRUD functionality.	Must	This is needed for calculating the collection time recommendation.
<b>DF11</b>	A database table to store the products' ingredients with CRUD functionality.	Could	This is needed for calculating the food recommendation.

*Table 1 shows the functional requirements for Database.*

## Staff

No.	Description	Moscow	Justification
<b>SF1</b>	The system must allow the users to log in as a staff.	Must	Essential feature
<b>SF2</b>	The system must provide a frontend and backend for the user to view or change all the order statuses such as paid, collected, and done.	Must	This can allow the staff to view and keep track of the current order status.
<b>SF3</b>	The system must restrict access to non-staff users to access a staff only page.	Must	This is for the system security concern.
<b>SF4</b>	The system must provide front and backend for users to make a change or add products.	Must	Essential feature



<b>SF5</b>	The system must provide a front backend for users to take orders, produce a collection time for customers, and send it to the kitchen display system.	Must	Essential feature
<b>SF6</b>	The system should generate the optimal collection time for the specific order.	Must	This is one of the objectives of the project.
<b>SF7</b>	The system must allow the user to send a request to the kitchen for order cancellation.	Must	This is part of the objectives that minimise the activity requiring staff to cancel the order, such as asking for cancellation by walking into the kitchen.
<b>SF8</b>	The system should allow users to make a change or add additional ingredients.	Could	This is part of the food recommendation requirements.
<b>SF9</b>	The system would allow users to change the layout of the ordering user interface.	Won't	This requires more design aesthetics, and it is not the primary concern of this project.
<b>SF10</b>	The system must allow users to add, modify, and remove staff accounts.	Must	Essential feature

*Table 2 shows the functional requirements for staff.*

## Kitchen

Code	Description	Moscow	
<b>KF1</b>	The system must provide frontend and backend for the user to view all the accepted orders with a walk-in text or collection time.	Must	Essential feature
<b>KF2</b>	The system must provide a frontend for the user to view the details of the products, such as quantity, name, and additional message of the product.	Must	Essential feature

<b>KF3</b>	The system must allow the user to change the status of the ordered product to do.	Must	This corresponds to <b>SF2</b> , where the staff can view the order's status.
<b>KF4</b>	The system Could notify or highlight specific products in an order that required a long time to prepare.	Could	These products can be easily spotted by the users, which does not necessary to implement them.
<b>KF5</b>	The system must remove the order once the user has finished preparing.	Must	Essential feature
<b>KF6</b>	The system should allow the user to print out the order.	Could	This requirement can only be tested and fulfilled with the printer hardware.
<b>KF7</b>	The system would allow the user to communicate with the staff through the same system.	Should	This requirement is between “ <b>must</b> ” and “ <b>should</b> ”, but because there might be some complexity within it, a should-do MoSCoW level is given.
<b>KF8</b>	The system would automatically refresh if a new order came in.	Must	The kitchen must keep updating the newest order status.

*Table 3 shows the functional requirements for kitchen.  
Customer*

Code	Description	Moscow	Justification
<b>CF1</b>	The system must provide a frontend for users to view the menu.	Must	Essential featured
<b>CF2</b>	The system must allow users to ask staff for table service.	Could	Due to the project's primary objective is for the restaurant operations, customer requirements are not heavily considered.
<b>CF3</b>	The system should allow users to make orders.	Could	Same as above
<b>CF4</b>	The system should recommend products to users.	Could	Same as above
<b>CF5</b>	The system should allow users to register and log in to an account.	Could	Same as above

<b>CF6</b>	The system would allow users to modify the order sent into the kitchen.	Would	Same as above
------------	---	-------	---------------

*Table 4 shows the functional requirements for customer.*

## Non-functional Requirements

Non-functional Requirements focus on the performance and set the minimum system requirement.

Code	Description	Moscow	Justification
<b>NF1</b>	The system must handle three simultaneous requests for the order to send to the kitchen display system.	Must	This is to avoid the system crashing at the restaurant's peak time.
<b>NF2</b>	The system must be able to produce a recommended estimated collection time for each order within 2 seconds.	Must	This avoids customers waiting for their order collection time for too long.
<b>NF3</b>	The system must provide a clear and straightforward user interface.	Must	Essential feature
<b>NF4</b>	The system must run smoothly with no crashes. All the ordering input responses must be produced within a second.	Must	Essential feature
<b>NF5</b>	The system must be available 24/7, 365 days a year.	Must	Essential feature
<b>NF6</b>	The system must be able to handle and store up to 25 categories and 200 products.	Must	This will be tested with NF4 to ensure that large amounts of data stored in the system will not affect the system's operations.
<b>NF7</b>	The system must be able to store all the taken order data into the database, which can be reaccessed if the page is accidentally closed or corrupt.	Must	This is to avoid the website being accidentally closed or refreshed and the order page content being lost.
<b>NF8</b>	The system must be able to handle more than ten customers using it simultaneously.	Must	Essential feature

<b>NF9</b>	The system must be able to run on multiple browsers.	Must	This is to ensure that the JavaScript used for the frontend must be compatible with most browsers in the market.
<b>NF10</b>	The system Would design to adapt to a mobile application potentially.	Would	Due to the web application is not the only location to deploy this system, a design which can be easily adapted to the mobile application screen is considered.

Table 5 shows the non-functional requirements for the system.

## Methodology

The system development process consists of multiple stages such as literature review, requirement gathering, design, and testing. It is advised to follow the process to produce and deliver the entire system on time. Therefore, a methodology is needed.

### Agile Methodology

I chose Agile instead of Waterfall methodology because the initial of this project is considered late which started around 2022 of January. There is not much time to thoroughly do the literature review and design chapter. The waterfall model is suitable for stable system developments and accurately predicts the design stage's flaws (Mike 2012). On the other hand, due to the Agile model iterative process, the design and requirements in this project can be changed after each sprint when facing stagnation on specific tasks.

To perform agile methodology, tasks must be split into different sprints. This allows me to allocate different times to each sprint based on the difficulty of the tasks. A week before the deadlines will be used as the completion of the project day to avoid abrupt interruption to the project.

#### Gantt Chart

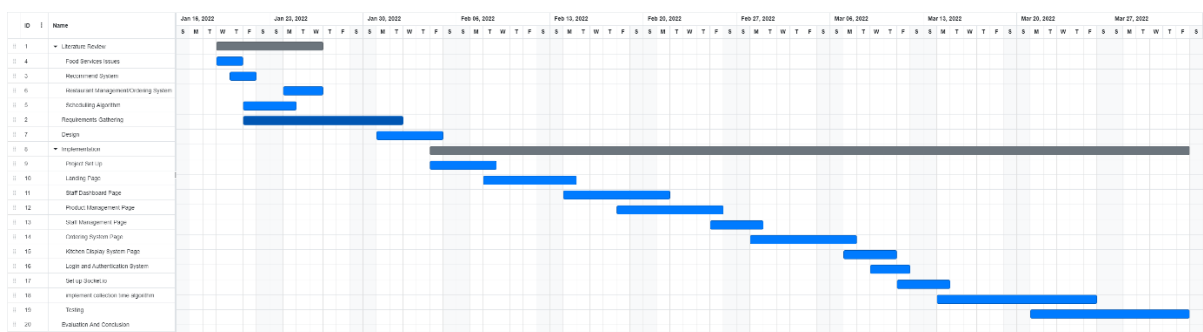


Figure 4 shows the Gantt chart for the project.

# Design

## Development Tools

The following tools are used for building the web application and design:

- **Lucid Chart**  
All the low-level and high-level system design figures described in this chapter will be using a Lucid chart. It provided multiple templates to make the designing process quickly with no fee of charge. Not only that, but there are also lots of icon or image resources available.
- **MySQL**  
All user data for this system must be well-formatted, and data-to-data must have a strong relationship, such as products under a category. It is database management system using SQL.
- **Node.js**  
Node.js is a backend web server runtime environment; the performance compared to PHP is memory efficient, fast, and can run responses to multiple clients with one thread due to its asynchronous runtime. It can use the NPM command to get all the external libraries that would assist in the building of the fundamental functionality of this system.
- **Express.js**  
Express.js is a web application framework for Node.js that can be used to deploy a web application quickly and easily.
- **Socket.io**  
Socket.io can connect all the clients in a room to allow real-time bi-directional communication without losing data. Additionally, it is well documented and has many tutorials online.
- **JQUERY**  
jQuery helps to speed shorten the JavaScript code required for frontend functionality. On top of that, it is required to perform Ajax.
- **AJAX**  
Asynchronous JavaScript and XML minimise the times for page loading by performing all the get post delete methods asynchronously.
- **Passport.io**  
passport.io can be used to authenticate and verify users to avoid non-staff users entering the prohibited page.
- **Pug**  
Pug is a replacement for HTML code. Which can easily add JavaScript code within it, and shorter syntax makes the code more manageable and readable.
- **Mocha.js**  
Finally, Mocha.js is used for all the testing phases in the implementation part. It is a test framework for node.js in the browser and checks for most of the asynchronous issues (Mocha, 2022).

## User stories

With the listed functional requirements, details of the user stories would help identify and ease the further design works. User stories would help identify and detail the functionality requirement with the requirements gathered and listed.

<b>As a staff</b>	I want to manage product details to keep the product's detail up to date.
	I want to view all orders to let the customer know their order status.
	I want to take customers' orders and send their orders to the kitchen for preparation.
	I want to view the order history to check the order details if the customer has a question.
	I want to set the order status to paid so that other staff can know it.
	I want to be able to send a cancellation request, so the order can be void if the order is not yet processing.

*Table 6 shows the staff user stories.*

**As a kitchen staff/Restaurant Owner** I want to manage the staff list so that new/old staff can be added or removed.

	I want to be the only one to manage the staff list so that the other staff cannot access it.
	I want to view the order created by the staff so that I can know the order details and prepare it in the first place.
	I want to change the status of the ordered product to know what has been done and what has not.
	I want to change the order's status so that the staff can know whether the order is prepared or delayed.
	I want to view the order cancellation request immediately so that the staff can let the customer know the response in no time.
	I want the system to sort the order automatically or manually so that the order with the nearest collection time can be on top of the page.

*Table 7 shows the restaurant owner user stories.*

## Use Case Diagram

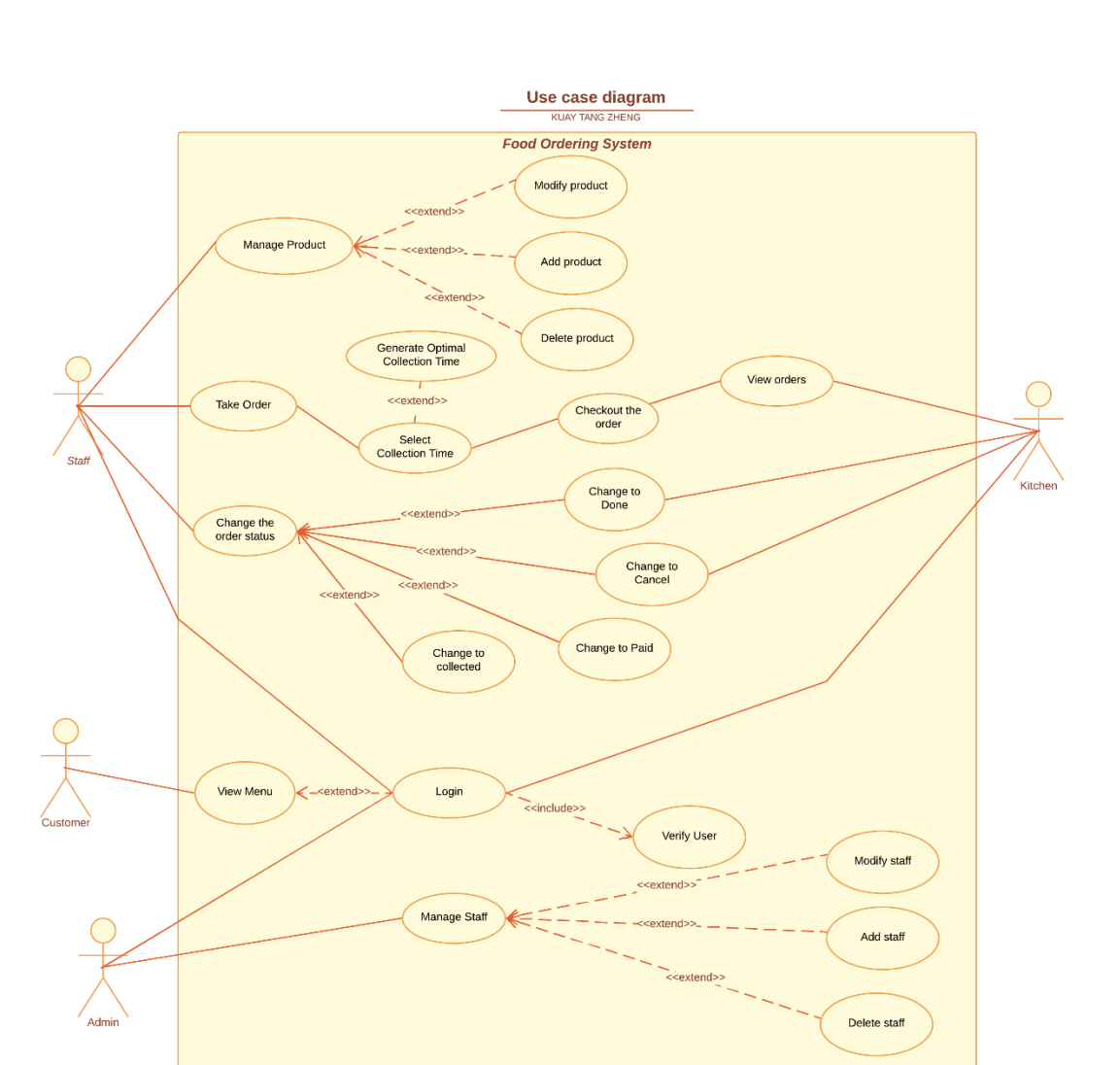


Figure 5 Food Ordering System High-level Use Case Diagram

This use case diagram includes most of the high-level functionality that have been planned to add to this system. The actor "Admin" has the highest privilege and can use all the features even though the only connection node is the staff management.

## Sequence diagram

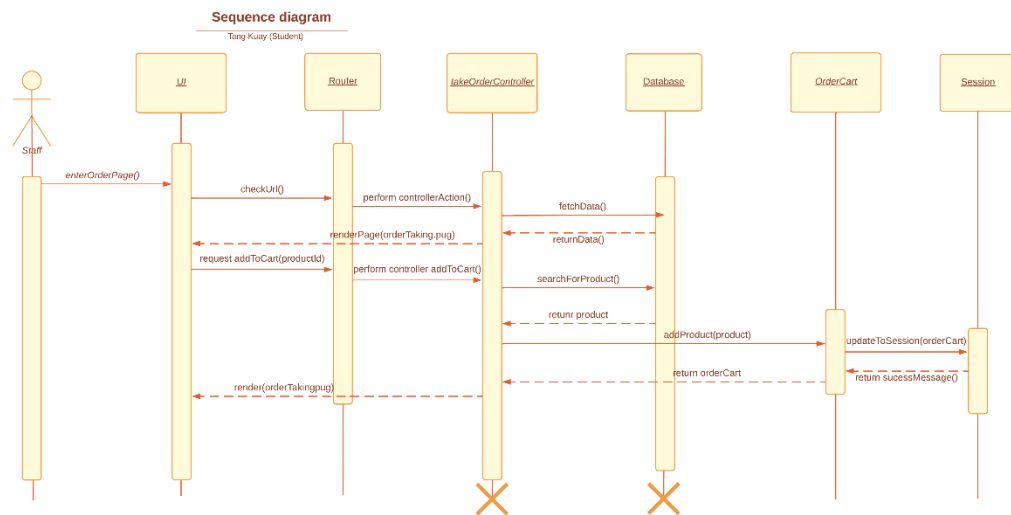


Figure 6 Low-level Sequence Diagram for taking orders

This sequence diagram shows how the staff interact with the system to add the product to the cart before sending it out. This sequence diagram uses Model View Control (MVC) structure which is how Nodejs works. The OrderCart instance is stored inside the session once it is created.

Router is where the code will handle all the URL request from the clients. After that, the router will call a business logic function from controller class. The action can be, add product, cancel, or checkout OrderCart. The controller then fetches the required products info from the database and add it in the OrderCart module. This module will be stored in the session until the user make an final action to the OrderCart.

## Flowchart

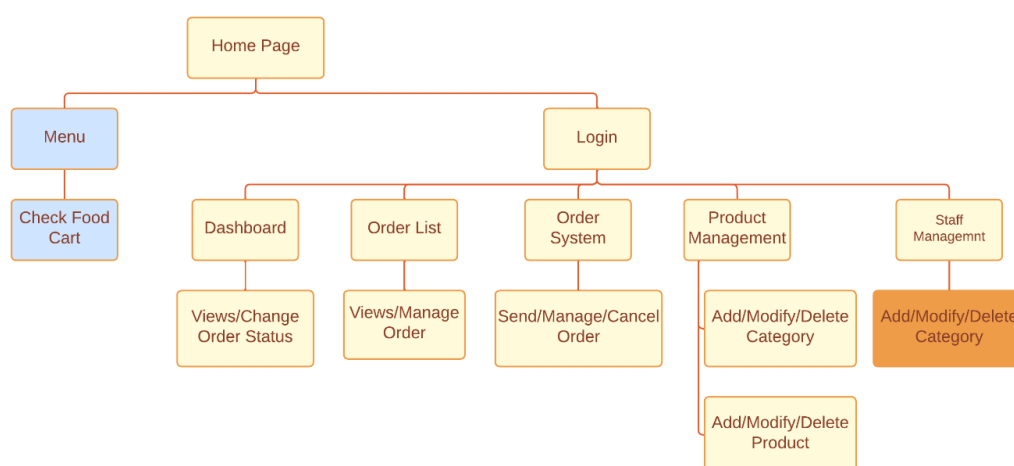


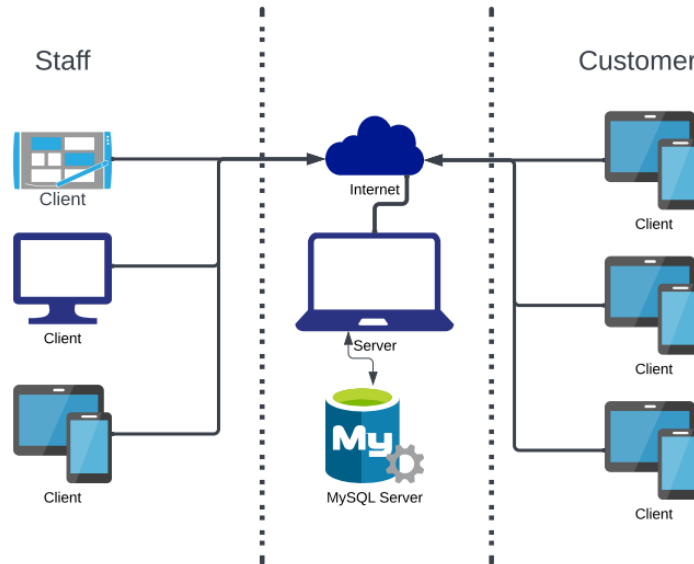
Figure 7 Food Ordering System Web pages flowchart

Figure 7 shows the web application pages hierarchy. The home page is the landing page for all users, and the user needs to be authenticated to enter the pages after the Login page. Furthermore, the customer can only view the menu and their food cart. To Clarify the design:



- blue colour pages - indicate that all users can access it
- Yellow colour pages - indicates only staff or admin can access it.
- The orange page indicates that both staff and admin can access it, but only the admin can use the functionality inside.

## Network architecture



*Figure 5 Network Architecture diagram*

This diagram shows the overview network architecture that will be used. In the testing section, this web application will be hosted locally. Because this system is built as a web application, users can use various devices with browser compatibility to access it. On the left side is the staff client side, and the right side is the customer client side. For the staff clients, considering multiple devices with different screen sizes need to be accessible to the web application, a responsive web design must be considered.

## State diagram

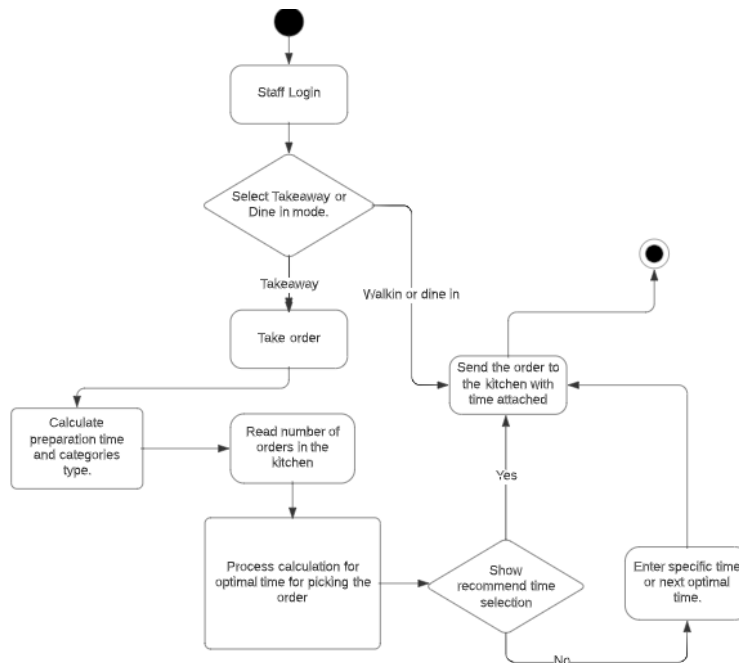


Figure 6 shows State Diagram for Order Taking.

Whether the customer walks in to make an order or by phone, the staff user has to log in to the system before taking the order. The staff can then select take away or restaurant mode, which does not share the same price and additional charges. For example, take away charge customers an extra 10p for a bag.

After the order is taken, the system will compare the existing and new orders. The calculation for the best collection time would be based on the type of the order products and preparation time. If the customer is satisfied with the collection time, the order can be accepted and sent directly to the kitchen display page.

## Order Taking Class diagrams

Because the Nodejs framework uses modules instead of classes, all the objects are created by fetching data from the MySQL database. The Class Diagram shown here is the hypothesis of the module that will be included in this project.

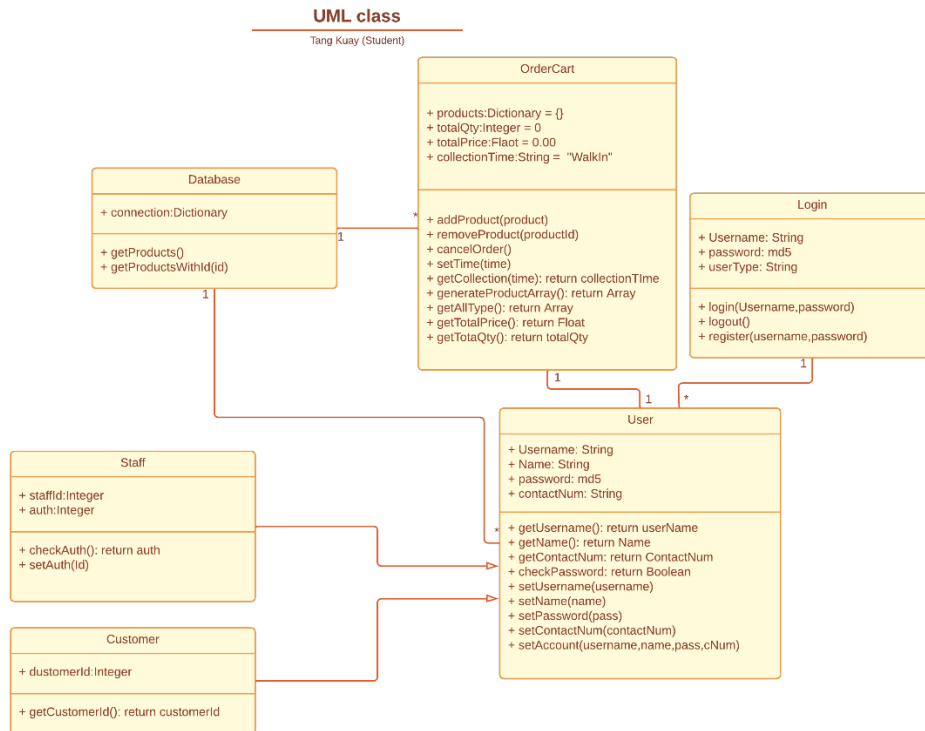


Figure 7 shows the take Order Class Diagram.

This class diagram is an example of the usage of order taking. Each user can only have one account and OrderCart. Therefore, the manipulation of the OrderCart can happen only if a user has login.

## Database Design

The database diagram consists of all the tables gathered from functional database requirements and is generated using MySQL Workbench.

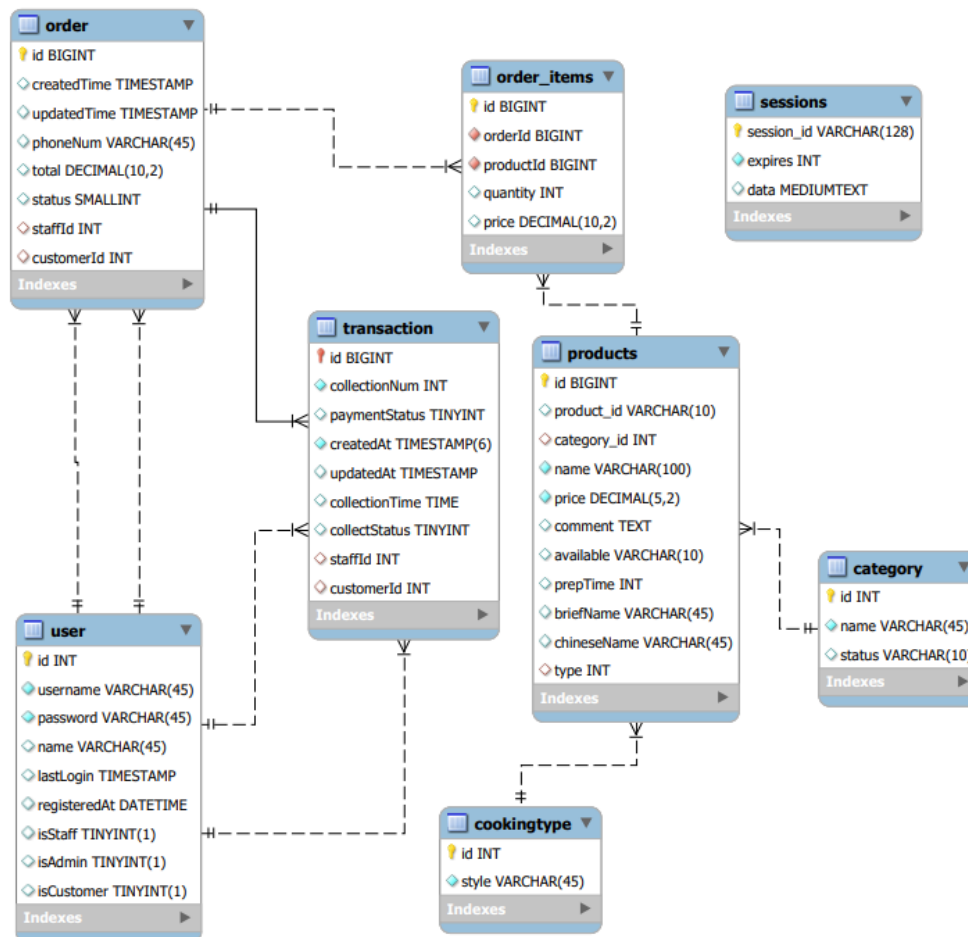


Figure 8 shows the Food Ordering System Database Design.

The User table has the most connected relationship with the table in the above figure. All the actions performed by the user that modify the database table must be recorded with the user `userId` and timestamp to make the system more traceable. For instance, the design choice of the database is to record the staff that changes the status of the order or transaction.

The second table that needs to mention is the cooking type, which only stores a String variable. It is used to distinguish the product cooking method, in which the chef can cook all the products that share the same cooking type without having to swap the utensils. Therefore, this table will be used to calculate the recommended collection time for each order.

Also, the session table stores the user session created by the Nodejs session library. It contains information, such as order cart, `userId`, and `customerId` but will be deleted once the user idles for a particular time or chooses to log out.

Finally, Order and Transaction tables share the same Id. Even though the table data structure looks similar, both tables have identical usage but for different actors. For instance:

- Order Table – It will only be used in the kitchen to update the order's status, such as Send in, Preparing, Done.
- A transaction table is used to record the transaction between staff and customers, such as Unpaid, Paid, Send Out, and Cancel.

# Algorithm design

## Time recommendation

Customers' orders might be varied; however, some orders consist of products that the existing order also has. Therefore, the objective of this project is to create an algorithm to produce a time that can help the kitchen minimise the time of swapping utensils by grouping the same products. The algorithm gets all the next 60 minutes of existing orders collection time. This ensures that the algorithm will not produce recommended time that needs customers to wait for long. After that, comparison, and calculation of the similarity of the orders will proceed. Once the candidate is found, the system will show the staff the same recommended collection time as the candidate order.

The below diagram shows the algorithm state diagram to generate a recommended collection time.

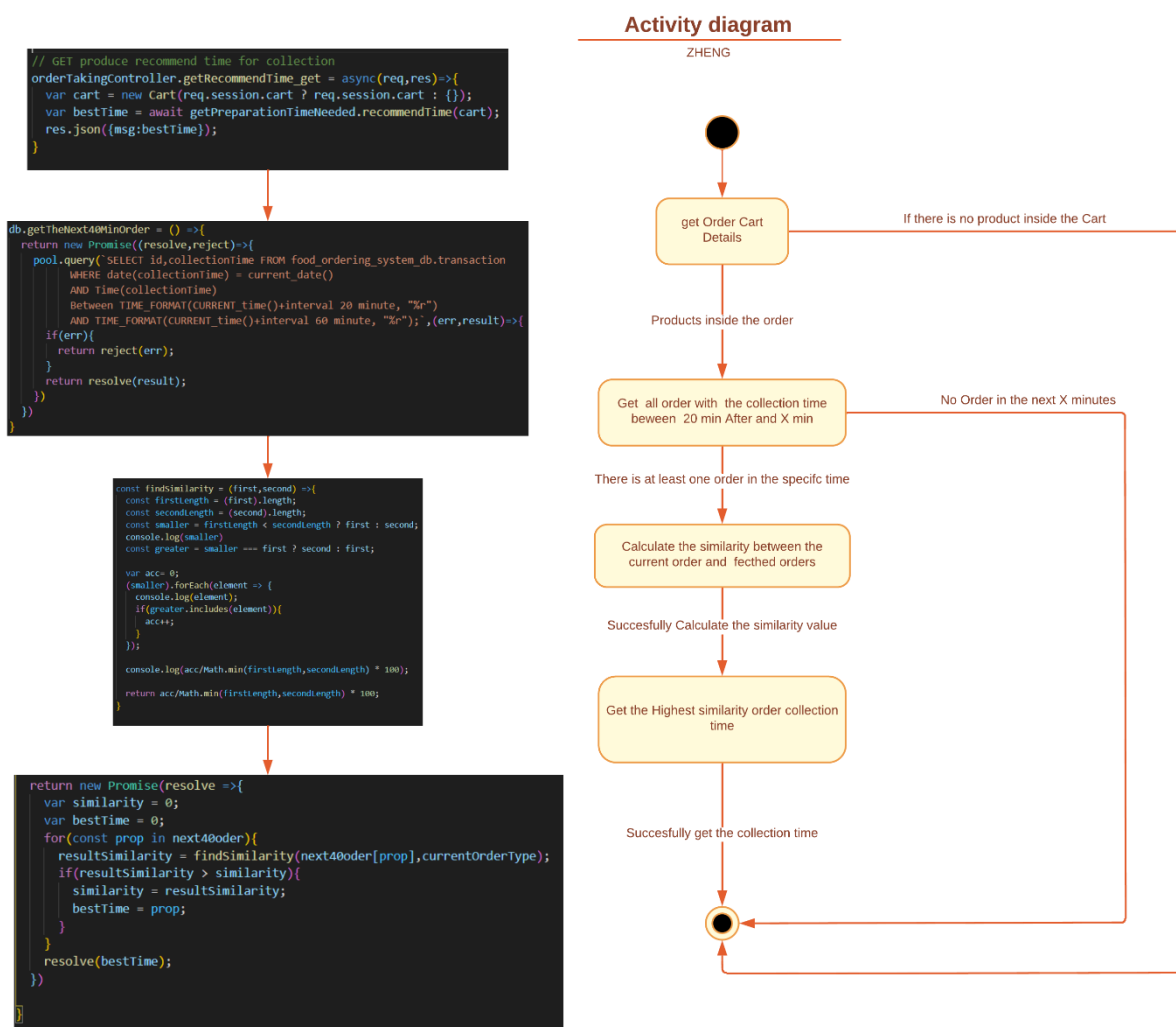


Figure 9 shows the Recommendations algorithm for the collection time state diagram.

Although the code shows the 20-60 minutes of time constraint for the order comparison, staff can modify the limit in the web application themselves.

## Test Design

After the completion of all sprints, user acceptance testing will be conducted. For complete coverage unit testing, some functional requirements will break down into multiple small user acceptance test cases. For example, the database CRUD requirements can break down into four different test cases. Although the target is to fulfil all the requirements, some tests might fail due to unknown circumstances and will consider solving them based on the MoSCoW priority.

The test format will be like the table below:

Test ID	Testing	Expected Outcome	Pass/Fail
UAT1	Enter the landing page.	Users can navigate to the landing page by given URL or QRCODE.	Pass
UAT2	Enter the login page.	Users can navigate to the login page.	<b>Fail</b> – user is unable to navigate to the login page.

*Table 8 shows the design for user acceptance tests.*

More user acceptance tests can be found in the appendix.

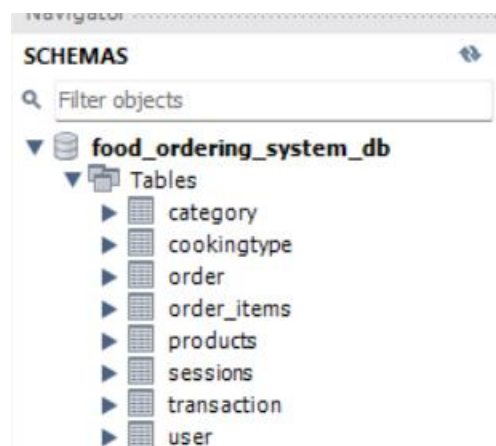
## Implementation

### Project Set-Up and Database (Sprint 1)

#### Database

Once the project finishes set up with Node.js and Express.js, the first sprint is to create a database with all the tables since this is essential for the following sprints. For instance, a product needs to be shown under its category, which requires a table foreign key to make the connection.

With MySQL Workbench, the database can be created without using SQL Query:



*Figure 10 shows the FOS Database table diagram using MySQL Workbench.*

```

require('dotenv').config();
const { response } = require('express');
const mysql = require('mysql2');

const pool = mysql.createPool({
  connectionLimit: 10,
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD
});

```

Figure 11 shows the MySQL create pool connection in Node.js

Figure 10 code shows how the pool connection between node.js and MySQL. The database access information is stored in an environment file and read with the dotenv library.

```

DB_PORT = 3306

DB_HOST=localhost
DB_USER=root
DB_NAME=food_ordering_system_db
DB_PASSWORD=password

```

Figure 12 shows the environment file for the MySQL connection

All the queries to the MySQL database can be done using the pool constant; it is limited to only ten connections to satisfy the NF8 non-functional requirements and avoid the device slowing down at the peak time.

## Functional requirement

Code	Status
DF1	Completed

The only requirement it satisfies in this sprint is DR1 which creates the database.

Table 9 shows the requirements fulfilled in the sprint 1.

## Landing Page (Spring 2)

The restaurant management and ordering system consist of multiple frontends, either for customers or staff. The landing page for every user is the home page which allows staff to navigate to the restaurant management page and customers to view all the menus. The home page is designed for customers to view the menu, consisting of a category navigations button and all the products' prices and names in their categories.

Back to the top page (Category) Admin Login

Bamboo House Menu	
Set Meals	Combination Platter
Dim Sum	Appetisers
Soup	Seafood Dishes
Prawn Dishes	Beef Dishes
Chicken Dishes	Pork Dishes
Duck Dishes	Noodles, Rice & Vermicelli
Additional Portions	Vegetarian Selection
Sweets	European Dishes
Childrens Menu	Drinks

Dim Sum	
Har Kau(Prawn Dumplings)	£3.95
Siu Mai(Prawn & Pork Dumplings)	£3.95
Steamed Char Siu Buns(x3)	£3.80
Fun Kori(Deep Fried Prawn Parcel)(x4)	£3.95
Vegetables Spring Roll(V)(x10)	£3.65
Spicy Vegetable Samosas(V)(x5)	£3.65
Crispy Won Ton(x8)	£3.95
Crispy Chicken Cheese Roll(x3)	£3.95
Crispy Duck Roll(x3)	£3.95
Wor Tips(Pan Fried Pork & Vegetable Dumplings)(x4)	£4.50
Appetisers	

Figure 13 shows the FOS Landing Page.

The page is created with Pug and designed with CSS. All the products and categories data are fetched directly from the database through the server. The page is rendered after the product is assigned to its category using a dictionary.

```
indexController.homepage_get = async (req,res) =>{
  let categories_dict = {};
  let allCategory = await db.getAllCategory();
  let allProduct = await db.getAllFood();
  allCategory.forEach(cat => {
    categories_dict[cat.categoryName] = [];
    allProduct.forEach(item => {
      if (item.category_id == cat.id){
        categories_dict[cat.categoryName].push(item);
      }
    });
  });
  res.render('index',{
    products: categories_dict
  })
}
```

Figure 14 shows how the products and categories are fetched from the database rendered to the users.

## Promise

Figure 13, the function uses an async header, especially when there are database process queries inside it because each database query returns a Promise object, which will include one of three states:

- Promise<pending> - The code action is not done when the return Promise object is pending.
- Promise<fulfil> - The code action is done and returns the value resolve value.
- Promise<reject> - The code executes failed when the return Promise object is rejected.

MySQL query will always return Promise<pending> and process the following code without using await keyword to wait until the database query is done and resolve it due to Node.js asynchronous runtime.



```

-for (key in products)
  h2.procat= key
  hr
  table
    - let items = products[key]
    each item in items
      tr
        td
          span.product=item.name
          -var foodPrice = "£"+item.price
        td.price
          span=foodPrice

```

Figure 15 shows how easily a can pug use for loop to assign a variable to an HTML tag while using JavaScript to declare a variable within it.

Most of the time in this sprint is, inserting the product and category data into the database using MySQL Workbench. Therefore, there will be no testing required in this sprint.

## Function Requirement

Code	Completion
CF1	Completed

Table 10 shows the requirements fulfilled in sprint 2.

## Staff Dashboard Page (Sprint 3)

The Staff Dashboard Page is where the staff will have most of the interaction with the orders. After a staff login, it will be a landing page for staff; therefore, it consists of multiple information about the restaurant operations. The staff can modify the status of the order or transaction.

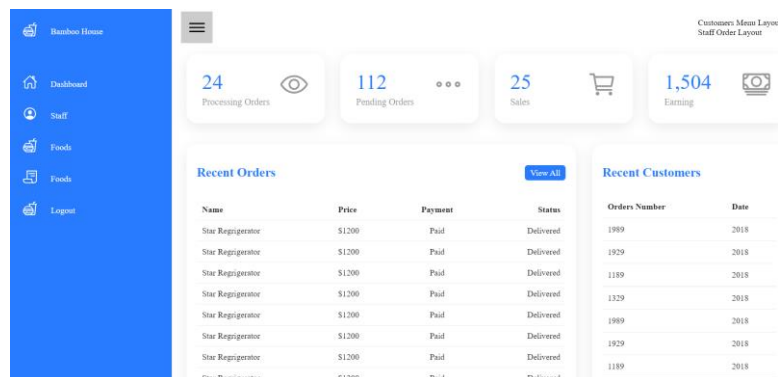


Figure 16 shows the page where the staff will make most of the interaction with the orders.

## jQuery

jQuery is a JavaScript library that helps the manipulation of HTML efficiently. For instance, each table row on this page holds an orderId within the HTML tag. The orderId is recorded, and a window will appear once the users click on the table row; this is done using the jQuery event handling feature.

```

$('.trOrderLink tr').on('click',function(){
    temp_order_id = $(this).attr('order_id');
    console.log(temp_order_id);
    $('#statusOption').addClass('hide');
    $('#orderOption').removeClass('hide');
});

```

Figure 17 shows how jQuery does the event listener in the frontend.

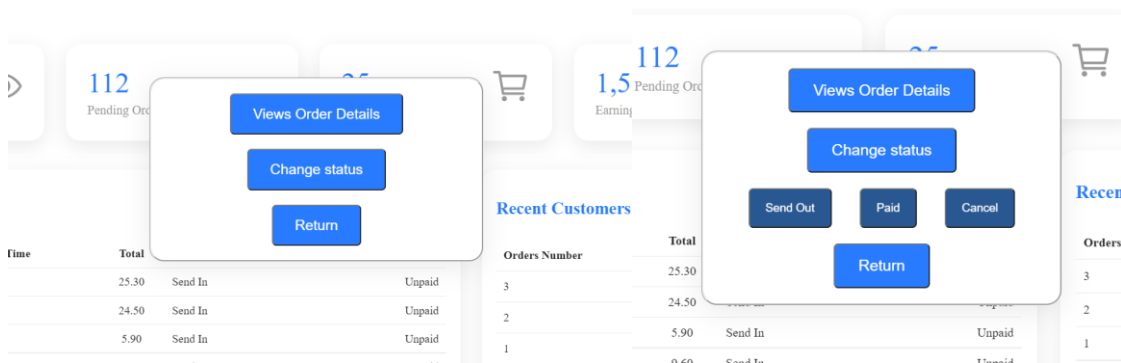


Figure 18 & 18 shows the selection for managing the order.

```

if(action == 'Paid'){
    $.ajax({
        url:`home/order/${temp_order_id}/1/updatePaymentStatus`,
        method:'post',
        success:(res)=>{
            if(res.msg=="success"){
                alert("Status Updated");
                let order = $('`[order_id="${temp_order_id}"]`');
                order.find('td.tdOrderStatus').text("Paid");
            }else{
                alert('some error occured try again');
            }
        },
        error:(res)=>{
            alert('server error occured');
        }
    })
}

```

Figure 19 shows how the GET request is sent to the server.

## AJAX

Furthermore, Ajax which stands for Asynchronous Javascript and XML, allows clients to request the server asynchronously without interfering with the current page. It is used in this project with the collaboration between Ajax and Jquery. For instance, In **Figure 18**, Ajax is responsible to send the post request and jQuery handle the text changing. So that the user can send the GET request to the server and change the status of the order without having to reload the whole page but the component of it.

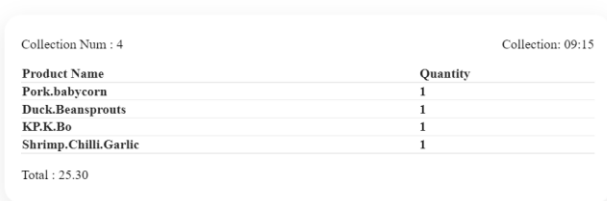
## Backend

```
dashboardController.getSpecificOrder_get = async(req,res)=>{
  let id = req.params.order_id;
  var order_items = await db.getOrderDetails(id);
  var order = await db.getOrder(id);
  var order_trans = await db.getOrderTrans(id);

  res.render('staff/orderDetails',{
    order_items : order_items,
    order : order[0],
    order_trans : order_trans[0]
  })
}
```

Figure 20 shows the controller for viewing a specific order

**Figure 19** shows where the server handles the HTTP request controller. For instance, the controller will be triggered with an HTTP request via Ajax and jQuery if a user selects a specific order to view details. After that, the dashboardController will render a new page with provided orderId.



Collection Num : 4 Collection: 09:15

Product Name	Quantity
Pork.babycorn	1
Duck.Beansprouts	1
K.P.K.Bo	1
Shrimp.Chilli.Garlic	1

Total : 25.30

Figure 21 shows one of the order details examples.

## Functional Requirement

Code	Completion
SF2	Completed

Table 11 shows the requirements fulfilled in sprint 3.

## Product Management Page (Spring 4)

A Pug layout file is created because the sidebar will be used on multiple pages. Therefore, all the pages that extend or import the layout file will have the sidebar included as default. This can help front end modification more quickly when an extra page button is required.



## Functional Requirement

Code	Completion
SF4	Completed
DF5	Completed

Table 12 shows the requirements fulfilled in sprint 4.

## Staff Management Page (Spring 5)

The staff management page allows the admin to add, modify, and delete staff. A similar development process is used in the last sprints.

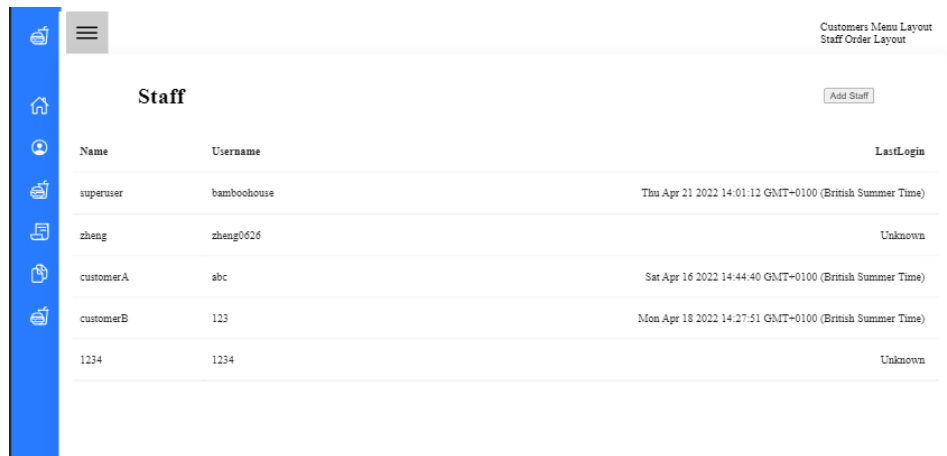


Figure 25 shows the staff management page.

## Password Hashing

The author supervisor suggests that the user password in the registration process must be hashed with the md5 hashing library. Instead of storing the raw password, the hashed value is stored in the database for each registration. When the user is trying to log in, the controller will hash the password input and compare. Therefore, other users cannot know each other passwords when accessing the database.

```
authenticateController.register_post = async (req,res)=>{
  let name = req.body.name_field;
  let username = req.body.username_field;
  let password = req.body.password_field;
  password = md5(password);

  await db.addUser(name,username,password,0);
  res.redirect('/');
}
```

Figure 26 shows the registration process code.

## Functional Requirements

Code	Completion
SF10	Completed
DF5	Completed

Table 13 shows the requirements fulfilled in sprint 5.

## Order System Page (Sprint 6)

Starting from this sprint will be the main objective of the project. Order System Page is where staff can the order from customers and select collection time.

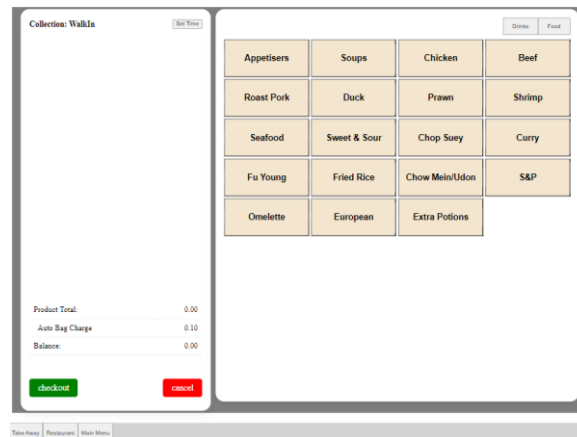


Figure 27 shows the Ordering System page with a categories list.



Figure 28 shows the Chicken category products list.

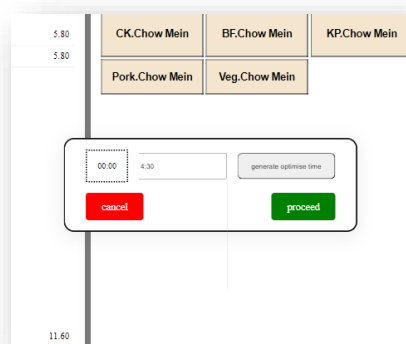


Figure 29 shows the collection time selection window.

Figure 27 has three sections: order list, category and product list, and navigation bar.

- **Order List**  
Where the staff can view the number of products added to the current order, staff can choose to check out or save the order in the database. Cancellation is to remove all the products from the list. Each order can select its collection time.
- **Category and Product List**  
Where all the categories' buttons will lead the user to their products list, this is where the staff can choose the product to add to the order.
- **Navigation Bar**  
For changing the order type to the restaurant, takeaway mode, and navigate back to the staff home page.

## Session

It is not memory efficient to call SQL query each time the staff add a new product to the cart. Sometimes the customer even intends to cancel the order before placing it. Hence, a module for maintaining a temporary order cart is required.

```
module.exports = function OrderCart(initProducts){
  this.products = initProducts.products || {};
  this.totalQty = initProducts.totalQty || 0;
  this.totalPrice = initProducts.totalPrice || '0.00';
  this.collectionTime = initProducts.collectionTime || "WalkIn";
}
```

Figure 30 shows the Order Cart module.

```
app.use(session({
  key: "hello.session.key",
  name: process.env.SESSION_NAME,
  resave: false,
  saveUninitialized: false,
  store: sessionStore,
  secret: process.env.SESSION_SECRET,
  cookie: {
    maxAge: 1000 * 60 * 60 * 12, //12hours
    sameSite: true,
  }
})))
```

Figure 31 shows the session connection code.

Additionally, a session is used to maintain the OrderCart module throughout the event. Once the staff add the first product into the OrderCart module, a token will be stored as a cookie on the client-side, and the session data is stored in the server-side database.

## Functional Requirements

Code	Completion
DF6	Completed
DF7	Completed
DF8	Completed
DF9	Completed
DF10	Completed
SF5	Completed

Table 14 shows the requirements fulfilled in sprint 6.

## Kitchen Display Page (Spring 7)

The kitchen display System page allows the kitchen staff to view the existing order. It is created for food preparation in the kitchen. Each order will include its collection time, number, and products. Besides, the only orders shown here are the order with the status Send in.

8:09:28			
7	09:45	9	10:00
1 Duck Rolls		1 Duck Beansprouts	
1 Pork Gin Snions		3 KP Gin Snions	
		1 Shrimp Gin Snions	
		1 Veg FR	
		2 KP Chow Mein	
		1 Shrimp Chow Mein	
14	WalkIn	15	06:30
1 BBQ Ribs		1 BBQ Ribs	
1 CK Corn Soup		1 Cant Ribs	
1 Ck Blk Pepper		1 Duck Rolls	
1 Ck Cant		2 KP Cashewnuts	
1 Ck Onion			
1 Bf Blps			
1 KP K Bo			
1 Shrimp Gin Snions			
16	06:40	17	06:30
1 Duck Bam Wnuts		1 Shrimp Beansprouts	
1 KP Pineapple		1 SeaFood Cashewnuts	
1 Veg Curry			
1 S&P Chips			
18	06:50	19	06:50
1 SP Chop Suey		1 KP Pineapple	
		1 S&P Squid	
20	06:50	21	07:00
1 Duck Bam Wnuts		1 Shrimp Oyster	
		1 CK Chow Mein	
Take Away Restaurant Main Menu			

Figure 32 shows the kitchen display page.

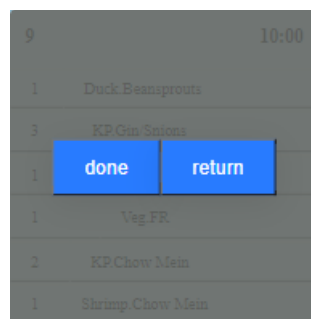


Figure 33 shows the done option for Order number 9.

Once the user selects the done button, the order is declared done and removed from the screen. The options window is shown after the user clicks the order window.



## Functional Requirements

Code	Completion
KF1	Completed
KF2	Completed
KF3	Completed
KF5	Completed

Table 15 shows the requirements fulfilled in sprint 7.

## Login and Authentication System (Sprint 8)

### Passport.js

Passport.js is authentication middleware for Node.js. It is responsible for the system login system. After verifying the user account successfully, passport.js serialises the user into the session in a set amount of time. Therefore, users do not need to verify each time they enter a restricted page.

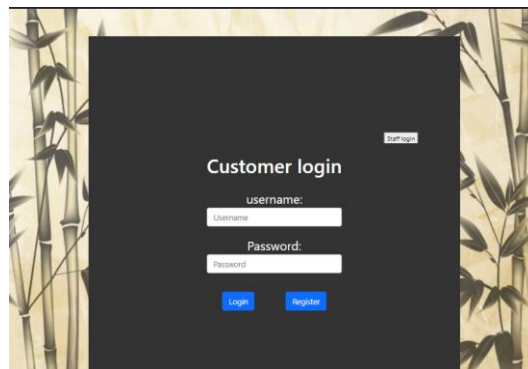


Figure 34 Login Page for staff and customers.

```
const verifyCallback = (username,password,done)=>{
  var x = (new Date()).getTimezoneOffset() * 60000;
  var timestamp = new Date(Date.now() - x).toISOString().slice(0, 19).replace('T', ' ');
  db.getUserByUsername(username).then((user)=>{
    if(!user) {return done(null,false)}

    if(md5(password) == user.password){
      db.updateLoginUserTime(user.id,timestamp).then(()=>{
        return done(null,user);
      });
      return done(null,user);
    }else{
      return done(null,false);
    }
  })
  .catch((err)=>{
    done(err);
  })
}
```

Figure 35 shows how the verification process.

```
passport.serializeUser((user,done)=>{
  done(null,user.id);
});

passport.deserializeUser((userId,done)=>{
  db.getUserById(userId).then((user)=>{
    done(null,{id:user[0].id,name:user[0].username,isStaff:user[0].isStaff});
  })
  .catch(err=>done(err))
});
```

Figure 36 shows how passport.js serialises and deserialise user.

```

module.exports.isAuthenticated = (req,res,next) =>{
  if(req.isAuthenticated() && req.user.isStaff){
    next();
  }else{
    res.status(400).redirect('/');
  }
}

```

Figure 37 shows how passport.js checks user authentication.

Figure 37 is the middleware to restrict the non-staff user from accessing the restaurant management pages. It is exported and used in the router file for authentication and will redirect the user to the landing page if a non-staff user tries to access it with the URL.

```

router.get('/logout', authenticateController.logout_get);
router.get('/takeOrder', isAuthenticated, takeOrderController.takeOrder_get);
router.get('/takeOrder/add-to-cart/:id', isAuthenticated, takeOrderController.addToCart_get);
router.get('/takeOrder/cancelOrder', isAuthenticated, takeOrderController.cancelOrder_get);
router.post('/takeOrder/setCollectionTime', isAuthenticated, takeOrderController.setCollectionTime_post);
router.get('/takeOrder/checkout', isAuthenticated, takeOrderController.checkout_get);
router.get('/takeOrder/getRecommendTime', isAuthenticated, takeOrderController.getRecommendTime_get);
router.post('/manage-product/addCategory', isAuthenticated, productController.addCategory_post);
router.get('/orders', isAuthenticated, kitchenDisplayController.kitchenDisplayPage_get);

```

Figure 38 shows the usage of the middleware isAuthenticated.

## Functional Requirements

Code	Completion
DF6	Completed
SF1	Completed
SF3	Completed
CF5	Completed

Table 16 shows the requirements fulfilled in sprint 8.

## Set Up Socket for bi-direction communication (Spring 9)

### Socket.io

This sprint is to build a socket connection between clients with the Socket.io library. Socket.io is a real-time bi-directional connection that is used widely in chat applications. It is the tool to satisfy the KF7 requirement. Socket.io has a client-side responsible for sending messages and accepting them, and the server-side is responsible for manipulating the message direction.

Each page that requires communication must include socket.io in the page JavaScript as an instance.

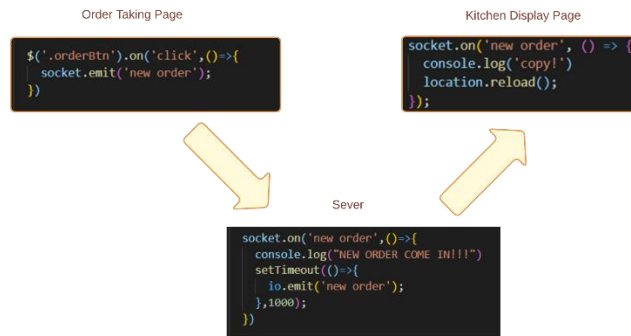


Figure 39 shows the socket.io flowchart.

In Figure 39, once an order is checked out on order taking page, it will emit a message to the server, and the server will direct the message back to all clients that are listening to this message. In this case, the kitchen display system will refresh after each order comes in, ensure that the page is real-time and keep updating for the newest order list.

The exact process is implemented for order cancellation requests, where the staff can request it from the dashboard page to the kitchen display page. Therefore, a confirm window will pop up on the kitchen display page, and the user can confirm or decline the request.

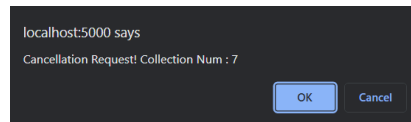


Figure 40 cancellation request sent from dashboard page user.



Figure 41 if the user is accepted in Figure 40



Figure 42 if the user declined in Figure 40

## Functional Requirements

Code	Completion
SF7	Completed
KF8	Completed

Table 17 shows the requirements fulfilled in sprint 9.

## Implement Collection Time Recommendation (Sprint 9)

This sprint is to create an algorithm that generates a recommended collection time for **sprint six** (Ordering System). A minor modified Cosine similarity function calculates the similarity between

orders with cooking type attributes created in the database products. Therefore, at least one order is required to generate time.

```
getPreparationTimeNeeded.recommendTime = async (orderCart) =>{
  currentOrderType = [...new Set(orderCart.getAllType())];
  var next40Order = {};
  var next40Trans = await db.getTheNext40MinOrder();
  for(const order of next40Trans){
    orderItems = [];
    orderDetails = await getOrderType(order.id);
    orderDetails.forEach(orderType => {
      if(orderType.type != null){
        orderItems.push(orderType.type)
      }
    });
    next40Order[order.collectionTime] = [... new Set(orderItems)];
  }
  return new Promise(resolve =>{
    var similarity = 0;
    var bestTime = 0;
    for(const prop in next40Order){
      resultsSimilarity = findSimilarity(next40Order[prop],currentOrderType);
      if(resultsSimilarity > similarity){
        similarity = resultsSimilarity;
        bestTime = prop;
      }
    }
    resolve(bestTime);
  })
}
```

Figure 43 shows the function to generate a recommended time.

In Figure 43, the order that needs to generate for a collection time is passed to this recommendTime(43rderCart) function. After that, get all the orders that collection times are between 20 minutes and 40 minutes after. With a set of current types and a list of other order types, the similarity value is calculated with the function in Figure 44. Finally, the order with the highest similarity value returns its collection time.

```
const findSimilarity = (first,second) =>{
  const firstLength = (first).length;
  const secondLength = (second).length;
  const smaller = firstLength < secondLength ? first : second;
  console.log(smaller)
  const greater = smaller === first ? second : first;

  var acc= 0;
  (smaller).forEach(element => {
    console.log(element);
    if(greater.includes(element)){
      acc++;
    }
  });

  console.log(acc/Math.min(firstLength,secondLength) * 100);

  return acc/Math.min(firstLength,secondLength) * 100;
}
```

Figure 44 the algorithm to get the similarity value between orders

The method to test the algorithm is to generate 2 random orders with different cooking type products and collection time. After that, perform calculation for the optimise collection time with the given order. The results will product a similarity value which define the collection time, and a table shows the different scenario outcome.

For instance:

Order (Collection Time)	New Order	Order A (7:00)	Order B (7:20)
Product 1	Salt & Pepper Wings	Salt & Pepper Chips	King Prawn Fried Rice

Product 2	Chicken Beansprout	Chicken Chow Mein	Special Foo Young
Product 3	Chicken Fried Rice	Shrimp Oyster Sauce	
Similarity Value		66.66	33.33

Table 18 shows the Order And similarity value example, the colour indicates different cooking types.

New Order Collection Time	7:00	7:20
Total utensils swap times (The smaller, the better)	5	6

Table 19 shows the utensils swap time result for each collection time.

The highest similarity value is order B, and the collection time of 7:00 is recommended to the user with an alert and replace the value automatically.

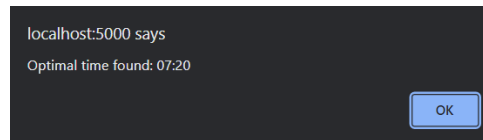


Figure 45 shows the alert recommends time for the user

Another critical example that shows the best scenario of the algorithm usage:

Order (Collection Time)	New Order	Order A (13:20)	Order B (13:30)
Product 1	Sweet & Sour Chicken	Steak with black pepper sauce	Chicken with Satay Sauce
Product 2	Steak with Satay Sauce	Duck Fried Rice	Beef with Cashew nuts
Product 3	Shrimp with Black beans	Salted and Pepper Squid	Sweet & Sour Duck
Product 4	Chicken Oyster Sauce	King Prawn Chow Mein	Beef with Black beans sauce.
Similarity Value		66.66	33.33

Table 20 shows the Order And similarity value example, the colour indicates different cooking types.

New Order Collection Time	13:20	13:30
Total utensils swap times (The smaller, the better)	8	4

Table 21 shows the utensils swap time result for each collection time.

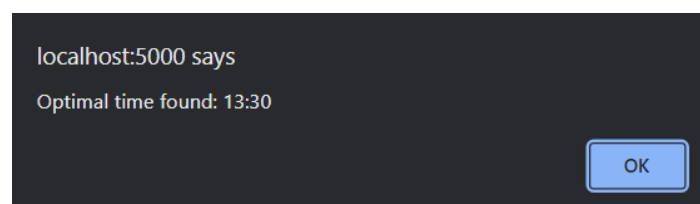


Figure 46 shows the alert recommends time for the user.

## Functional Requirement

Code	Completion
SF6	Completed

Table 22 shows the requirements fulfilled in sprint 10.

## Testing (Sprint 10)

### Testing

Testing is essential to the backend code to identify errors that have not been encountered before deploying it in the real world. There are two types of testing in this sprint:

- Unit Testing – is a testing method where a unit is a small chunk of code that acts as a function and checks if the return value is expected.
- Integration Testing checks whether multiple units collaborate and deliver the expected behaviour or value.

There is a total of 40-unit testing for all different controllers and pages. Moreover, the integration will be included in the user acceptance testing.

### Mocha.js

Mocha.js is a JavaScript test framework for Node.js to test browsers and make asynchronous testing. It will be used to test the backend for each sprint that uses the backend.

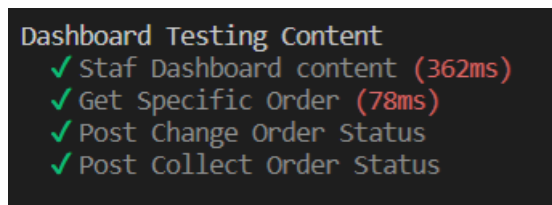


Figure 47 shows the unit testing for the dashboard page.

```
it('Post Change Order Status', function(done) {
  request('http://localhost:5000/user/home/order/4/Paid/updatePaymentStatus', async function(error, response, body) {
    var order = await db.getOrderTrans(4);
    expect(order.paymentStatus == "1");
    done();
  });
});
it('Post collect Order Status', function(done) {
  request('http://localhost:5000/user/home/order/4/collected/updateCollectStatus', async function(error, response, body) {
    var order = await db.getOrderTrans(4);
    expect(order.status == "2");
    done();
  });
});
```

Figure 48 shows an integration tests example.

40 passing (10s)

Figure 49 number of unit tests and results.

### User Acceptance test

All the user acceptance tests details can be found in the appendix chapter. Only all the functional requirements that have been fulfilled will be tested.

Code	Pass/Fail	Code	Pass/Fail	Code	Pass/Fail	Code	Pass/Fail
------	-----------	------	-----------	------	-----------	------	-----------

UA1	Pass		UA11	Pass		UA21	Pass		UA31	Pass	
UA2	Pass		UA12	Pass		UA22	Pass		UA32	Pass	
UA3	Pass		UA13	Pass		UA23	Pass		UA33	Pass	
UA4	Pass		UA14	Pass		UA24	Pass		UA34	Pass	
UA5	Pass		UA15	Pass		UA25	Pass		UA35	Pass	
UA6	Fail – Not implemented		UA16	Pass		UA26	Pass		UA36	Pass	
UA7	Fail - Not implemented		UA17	Pass		UA27	Pass				
UA8	Pass		UA18	Pass		UA28	Pass				
UA9	Pass		UA19	Pass		UA29	Pass				
UA10	Pass		UA20	Pass		UA30	Pass				

Table 23 shows the result of the user acceptance tests.

## Testing Reflection

Even though the last sprint is about testing, some tests are already done in different sprints to ensure the basic features are satisfied before moving to the next sprint. The user acceptance test results shown in the table have been modified for several iterations to pass most tests. Hence, it has proven its usefulness in helping identify the issues that are not encountered. However, some tests were failures because the implementation did not cover all the functional requirements; hence the user acceptance tests cannot be verified.

The approach for doing tests again recommended collection time algorithm is by giving the system multiple real-world scenarios where some randomly generated orders are provided. Comparing human and system calculation results have helped uncover the algorithm's defects. These results can be found in **Table 1** and **Table 2**.

Even though the number of unit testing increases over time, Mocha.js can test the full coverage of the system will only one command. It helps to identify errors in variables passing through different controllers.

# Project Evaluation

## Research

The research in this project was successful as I was able to find some relevant field papers. The papers have shown a different way to create a restaurant management system using a variety of algorithms to enhance it. However, it was challenging searching for articles that discuss takeaway and collection time optimisation. Therefore, I have tried to look for a similar field that would be useful if implemented in this project. For instance, Amazon's item-to-item collaborative algorithm allows me to find similarities between items and their features. As a result, the algorithm produced the best collection time with the highest similarity value.

## Requirements

Requirements gathering is the most successful aspect of this project, in which requirements were gathered from the different actors of the system. Even though the number of functional requirements was huge, it increased the project's complexity and made it more challenging. For instance, **SF7** required socket.io to create a bi-directional connection. **SF8** required an algorithm to optimise service-routine.

## Design

The design in this project was also successful as I could produce a variety of different level diagrams. The network architecture diagram was suggested by my supervisor Adam Gorine; it shows the network routing between clients and the server-side and helped me realise that different frontend design is required for devices with various screen sizes.

However, because JavaScript use dynamic types to define a variable, the class diagram was challenging to produce. Moreover, because of the MySQL database usage, there were no classes/modules, but one used for storing the temporary order cart. Therefore, I overcame this by creating a hypothesis class diagram for the project. The class diagram shows the relationship between each other and what value should a class store. Even though this Class Diagram was not implemented in this project, it helps design the database entity diagram.

## Methodology

Even though this project was developed using agile methodology, some sprints got held on until another sprint was made due to information required from another sprint. For instance, to have the Dashboard page working, it needs an order system page to produce order samples. The main issue I had with this was losing track of the tasks. Luckily, I quickly found the Kanban board on GitHub and deployed it to manage all the tasks.

With Agile methodology, the time for each sprint was planned. It improved the flexibility of the project development by adding additional features if there was extra time left after the sprint. However, unforeseen circumstances happened, and I was forced to use lesser time as a plan for the sprints behind. This issue could have been solved by setting the hand-in date a few weeks or a month earlier.

## Implementation

All the **"must"** the **"should"** requirements were implemented successfully. However, because this is my first time building a web application, starting implementation without knowing the basic knowledge of Model View Control (MVC) and Document Object Model (DOM) was the wrong decision. I am forced to recode and redesign the backend each time I find a better solution. For example, instead of keeping all code in the same folder and file, an ideal solution is to divide the code into three different MVC folders. Moreover, creating JavaScript code for manipulating the frontend was time-consuming until jQuery was imported. These all issues can be avoided by conducting more research on the tools that were planned to use.

## Testing



All the unit test for the system conducts smoothly. However, because this project is not intended to build as an API, the integration testing is challenging to perform with a tool like Postman or Mocha.js. Moreover, it is challenging to create integration testing that needs to run multiple times where there is only one database exists. For example, the integration testing for product deletion will require different product IDs in each run.

Therefore, the integration testing is part of the user acceptance testing, where the user is the one that does all the tests. Both unit testing and user acceptance testing went well. However, if the project was to start again, redesigning the web application towards the rendering and sending REST API format will be considered.

## Limitation and Future Work

Most limitations were considered implemented in the project, but they are included in future works due to time constraints. The main limitation of this system can be found easily, such as being unable to print a paper-based order receipt; this is not included in the project due to the lack of a compatible printer.

Secondly, the system could be improved by allowing more interaction with the system for customers. For instance, customers can take a queue for the restaurant table, order through the system, and make payments.

Thirdly, add ingredients information for different products for customers. To avoid allergy issues and a step further would be using the ingredients feature to build a product recommendation for customers who previously liked similar dishes.

Finally, add review feature to collect customer feedbacks. This would help the restaurant owners improve their services and the user experience of the application.

## Feedback

Adam, my supervisor, pointed out the issue with my objective writing. The most helpful advice is to use the md5 hashing technique for storing the user password, which I implemented in **sprint 5**. Therefore, the customer password will not be revealed, even with access to the database.

## Conclusion

The project overall met all the objectives and the aims, which is to produce a Food ordering system that optimised the process of the restaurant. The application allows the users to take an order, change the order status, and manage products and staff. Each order can generate an optimised collection time, benefit both customers and the kitchen by giving more accuracy and shortening the food process time.

# References

- Bobadilla, J., Ortega, F., Hernando, A. and Bernal, J. (2012) A Collaborative Filtering Approach to Mitigate the New User Cold Start Problem. Knowledge-based Systems [online]. 26, pp. 225-238. [Accessed 27 April 2022].
- Dhiman, K. and Phansikar, M. (2021) Online Food Ordering Management System. International Journal For Research in Applied Science and Engineering Technology [online]. 9, pp. 2096-2107. [Accessed 27 April 2022].
- Gursale, A., Navale, K., Bhosale, K. and Hanamant, B. (2020) Restaurant Ordering System Using Ar, Website Development and Recommender System Using ML. International Journal of Computer Trends and Technology [online]. 68 (4), pp. 139-144. [Accessed 27 April 2022].
- Hollingsworth, B., Dawson, P.J. and Maniadakis, N. (1999) Efficiency Measurement of Health Care: A Review of Non - parametric Methods and Applications. Health Care Management Science [online]. 2 (3), pp. 161-172. [Accessed 27 April 2022].
- Singhal, N. and Konguvel, E. (2022) An Approach For Restaurant Management System During Covid-19. In 2022 International Conference on Computer Communication and Informatics [online]., pp. 1-6. [Accessed 27 April 2022].
- Statista (2022) Year-over-year daily change in seated restaurant diners due to the coronavirus (COVID-19) pandemic in the United Kingdom (UK) from February 24, 2020 to February 20, 2022 [online]. Available from: <https://www.statista.com/statistics/1104991/coronavirus-restaurant-visitation-impact-united-kingdom-uk/> [Accessed 27 April 2022].
- Kanyan, A., Ngana, L. and Voon, B.H. (2016) Improving the Service Operations of Fast-food Restaurants. Procedia-social and Behavioral Sciences [online]. 224, pp. 190-198. [Accessed 27 April 2022].
- Liyanage, V., Ekanayake, A., Premasiri, H., Munasinghe, P., and Thelijjagoda, S. (2018) Foody-smart Restaurant Management and Ordering System. In 2018 Ieee Region 10 Humanitarian Technology Conference (R10-htc) [online]., pp. 1-6. [Accessed 27 April 2022].
- Linden, G., Smith, B. and York, J. (2003) Amazon. Com Recommendations: Item-to-item Collaborative Filtering. Ieee Internet Computing [online]. 7 (1), pp. 76-80. [Accessed 27 April 2022].
- McCormick, M. (2012) Waterfall Vs. Agile Methodology. Mpcs. [online] [Accessed 27 April 2022].
- Nonaka, T., Nobutomo, T. and Mizuyama, H. (2018) A Model of Dynamic Scheduling of Restaurant Operations Considering the Order and Timing of Serving Dishes [online]. : Springer International Publishing. [Accessed 27 April 2022].
- Noor, M.Z.H., Rahman, A.A.A., Saaid, M.F., Ali, M.S.A.M. and Zolkapli, M. (2012) The Development of Self-service Restaurant Ordering System (Sros) [online]. 2012 IEEE Control and System Graduate Research Colloquium: IEEE. [Accessed 27 April 2022].

# Appendix A: First Appendix

## User Acceptance Test

### Landing and Login page

Test ID	Testing	Expected Outcome
UAT1	Enter to the page.	Users can navigate to the landing page by given URL or QRCODE.
UAT2	Enter to the login page.	Users can navigate to the login page.
UAT3	User Enters the wrong password or username.	Redirect users to the same page.
UAT4	A user login in as a customer.	Passport.js recorded the customer in the session.
UAT5	A user login in as a staff.	Passport.js recorded the staff in the session.
UAT6	A user adds a product in to Cart.	The cart number increased on the landing page and the backend.
UAT7	A user checkout the cart.	The cart emptied and sent the order to the dashboard.

### Dashboard page

Test ID	Testing	Expected Outcome
UAT8	A user changes an order transaction status to Paid.	The order status changed in both the front end and the database.
UAT9	A user changes an order status to Send out.	The order status changed in the database, and the order will be moved to the order history table.
UAT10	A user views an order detail.	Direct the user to a new page with the order details.
UAT11	A user requests an order cancellation.	A confirm window to pop up on the kitchen display system page with the order number.

### Product Management Page

Test ID	Testing	Expected Outcome
UAT12	Add a product with some input that is left blank.	Product add failed; ask the user to enter all the information.
UAT13	Add a new product.	Product added in both front end and database.
UAT14	Delete a product.	Product removed in both front end and database.

UAT15	Modify a product but with some inputs left blank.	Product modified fail, ask the user to enter all the information.
UAT16	Modify a product.	Product modified in both front end and database.
UAT17	Add a new category with empty input.	Category add fail, ask the user to enter the name input.
UAT18	Add a new category.	Category added in both front end and database.
UAT19	Delete a category.	Category deleted in both front end and database.
UAT20	Modify a category with empty input.	Category modifies fail, ask the user to enter the name input.
UAT21	Modify a category.	Category modified in both front end and database.

### Staff Management Page

Test ID	Testing	Expected Outcome
UAT22	A user adds a staff with empty input.	Staff account add failed; ask the user to enter all the inputs field.
UAT23	A user adds a new staff account.	Staff account added in both front end and database.
UAT24	Delete a staff account.	Staff account removed in both front end and database.
UAT25	Modify a staff account with empty input.	Staff account modification failed; ask the user to enter all the inputs field.
UAT26	Modify a staff account.	Staff account modified in both front end and database.

### Ordering System Page

Test ID	Testing	Expected Outcome
UAT27	A user adds a product to the cart.	The product added to the cart in the backend, and the product shows in the cart front end.
UAT28	The user cancels the order cart.	The order cart is removed from the session, and the front end shows an empty product.
UAT29	User checkout the order cart with the empty product inside.	The checkout failed, alerting the user that there was no product inside the order cart.
UAT30	User checkout the order cart with the product inside.	The order cart is saved in the database and removed from the session, and the front end shows an empty product.
UAT31	User selects a collection time.	The collection time is changed in the order cart.
UAT32	User removes a collection time.	The collection time is changed to the default value "WalkIn".

UAT33	User generates a recommended collection time.	If success is generated, alert the user to recommend time and replace the collection time value; otherwise, alert the user failed.
-------	---	--

### Kitchen Display System Page

Test ID	Testing	Expected Outcome
UAT34	User changes the order status to Done.	The order status changed to done in the database and removed the order from the page.
UAT35	User accepts the cancellation request.	The order status changed to cancel in the database and removed the order from the page.
UAT36	User declines the cancellation request.	The order status remained unchanged, and the order remained on the page.