Software Engineering (ECE 452) - Spring 2019
Group #3

Restaurant Automation Codename Adam

URL: https://github.com/sa2423/SE-2019

Date Submitted: May 5, 2019

Team Members:
Seerat Aziz
Lieyang Chen
Christopher Gordon
Alex Gu
Yuwei Jin
Christopher Lombardi
Arushi Tandon
Taras Tysovskyi
Hongpeng Zhang

Table of Contents

Contributions Breakdown

Everyone contributed equally towards this report.

Summary of Changes

Section 2: Glossary of Terms

- Terms updated to reflect changes made to Ingredient Prediction System

Section 3: Functional Requirements Specification

- Fully dressed descriptions written for all other use cases
- System sequence diagrams added for all other use cases

Section 6: Domain Analysis

- System operation contracts written for all other use cases

Section 7: Interaction Diagrams

- Design patterns described for each interaction diagram

Section 10: Algorithms & Data Structures

- Recommendation system algorithm description updated to reflect what was implemented
- Facial recognition system description added

Section 11: User Interface Design & Implementation

- Updated to include UI screens that were implemented on apps and website

Section 12: Design of Tests

- Updated to add unit tests implemented since Demo 1

Section 14: Project Management

- Updated to include changes such as our ticketing system (Asana) and messaging systems.

Section 1: Customer Problem Statement

1.1: Problem Statement

Restaurant automation is a daunting task, as there are many variables to consider in the automation process. Moreover, it is important to maintain a balance between efficiency and hospitality as most customers associate restaurants with their ambience as opposed to mechanized efficiency. Many restaurants nowadays have started to automate their tasks, such as Panera Bread and TGIF, by streamlining the ordering and payment processes by employing tablets with specialized apps. Spyce, a restaurant in Boston, has automated its cooking process by replacing human chefs with a robot [4]. However, even the owners of Spyce seek to maintain the human touch by attempting to tailor the experience for each customer.

For this reason our project aims to automate basic tasks that customers, waiting staff and chefs have to perform in restaurants, without outright replacing them, in order to maintain the human touch that the restaurant experience depends on. This, in turn, will decrease the time it takes from customers coming in and placing their order to actually being served their food, resulting into better dining experience. Additionally, it will help the restaurant owners manage their business better through detailed statistics and analysis of trends. It will provide a way to easily see all the placed orders as well as a way to predict the amount of ingredients that will be needed for any given day, (inferred from historical data). This allows the restaurant owners to dramatically cut down the leftovers and shape their menu accordingly to what sells best. Moreover, our project will improve customer experience by recommending meals.

While the proposed solution requires additional investment in the form of electronic devices for customers and servers, as of 2018 lower-end tablets running Android operating system can be purchased for under $50 dollars, especially in larger quantities. As such, this is well within the realm of financial feasibility for medium and even small, family owned restaurants. Although our prototype is tailored towards Android operating systems, it can be implemented on the iOS operating system if restaurants do not mind paying more.

Problem: Helping customers decide on a meal to eat

A common problem that customers face when going to eat a restaurants, whether they have visited the restaurant before or not, is deciding on what food to eat. This problem is often worsened by menus with food items with very “foreign” sounding names or unfamiliar menus in general. Such problems can discourage customers from visiting the same restaurant, so we find that assisting customers with choosing meals will cause them to maintain their relationship with our establishment. As a result, this will cause our customer base to increase over time.

The Codename Adam Solution:

We seek to solve this problem by implementing a meal recommendation system, which will be enhanced by using responses from the review system. In order to implement the recommendation system, the user will be given the choice to take a short quiz before ordering their meal. This quiz will ask questions about their meal

preferences (if they want a meal with cheese, rice, meat, etc.) to recommend a list of appropriate meals. This quiz component will incorporate active learning by taking the user's choices (what meal they select from the given list) into account for future recommendations. Likewise, the user will be allowed to "favorite" meals on the application, which will also be taken into account for future meal recommendations. After a user finishes their meal, they will be given a short questionnaire to rate the meal and restaurant service. This responses from this short questionnaire (the review system) will also be taken into account when recommending meals to future customers. If the customer returns, we can further streamline the ordering process, using an account system with facial recognition. During this procedure, all the customer needs to do is take a seat and look at the camera, avoiding any inconveniences such as user login. Of course, to avoid any concerns about data collection and invasion of privacy users are informed of their information being stored and can opt out of this feature at any time.

Problem: Providing customers with efficient and careful service

With the advancement of internet technologies, customers want information delivered to their screens instantly. For restaurants in particular, they would like to see today's menu, reserve a table or even order food in a matter of seconds, without ever leaving their homes or talking to people. Along with customer service, restaurants have consider their customers' dietary concerns. This includes taking religious dietary laws (such as Halal and Kosher) and fad diets (Keto and Vegan) into account. Moreover, as food allergies have become more prevalent in today's era, it has become more important for restaurants to be vigilant with their food preparation efforts.

The Codename Adam Solution:

Codename Adam provides an ecosystem of websites and mobile apps that allow customers to interact with restaurants remotely and effortlessly. Checking today's specials, getting food recommendations, reserving a table and making an order can now be done in a matter of seconds. In order for the restaurant staff to be able to keep up with the features offered online, we will also offer apps for servers and chefs to increase their productivity by being able to receive orders instantly and be notified immediately when an order is done cooking and is ready to be served. Furthermore, the app specialized for the waiting staff will provide information about the meals, such as ingredients used, calorie information, and any possible allergy warnings. Likewise, the restaurant website (with admin privileges and the ingredient prediction system) will help restaurant owners decrease operating costs and increase profits.

Problem: Make server spend their time more efficiently

A server has a lot of things to do when they are working. For example, they need to serve for new-coming customers and stand by them to listen to their decisions on meals, they need to bring a finished meal from chef to a customer who is waiting, they need to go helping those customers who is calling a server for some specific questions. They have a lot things to do in their work, so an efficient way to use their time is really desired so that all customers can enjoy their time by being fully served.

The Codename Adam Solution:

Actually, there are some unnecessary actions that can be avoid by making our application more advanced. For example, if we can allow the customers to order their food on the table app, the time server used to spend on record customers' decisions can be saved. If we can allow customers to order food online instead of making phone calls, then the time severs use to talk to customers on phone can be saved. If we can allow customers to use the application to write down their specifications on each meal, then the overhead severs spend in transmitting the information from customers to chefs can be saved. With time saved, servers can spend more time on those customers who requests a server calling so they can be helped at once.

Problem: Maximizing profits and streamlining process for the Chef

The kitchen is the engine of a restaurant. Every customer's order must pass through the kitchen, which means that the efficiency and quality at which the kitchen operates heavily impacts the performance of the restaurant itself. As a result, it is in the restaurant's best interest to promote the efficiency of its chefs and the quality of its ingredients and to reduce wastage in its kitchen. If this can be accomplished, chefs will have more time and fresher ingredients, allowing them to produce a larger quantity of meals of higher quality, not only improving the customer's dining experience, but also increasing the restaurant's profit potential.

The Codename Adam Solution:

Codename Adam's ingredient prediction system helps to solve many of these problems that restaurants face. Based off of records of which menu items were ordered on the same day of the week in previous weeks, the ingredient prediction system will calculate an estimated amount of each ingredient that will be required on that specific day. This will allow chefs to determine how much of each ingredient to prepare for the the day's dishes, reducing food wastage, preventing chefs from wasting time preparing unnecessary ingredients, and ensuring that dishes use only the freshest of ingredients. In the beginning, the chef may want to prepare a little more than what is estimated, but over time, the estimate should become more refined and more accurate, as outliers are identified and pruned from the estimation. This will be accomplished by performing the RANSAC algorithm on the amount of ingredients needed for the k most recent weeks. The estimate may also be refined by trends in dish reviews and favorites and pending reservations. An extension of this feature may further divide the ingredient prediction by time of day, to maximize the freshness of ingredients and efficient use of the chefs' time.

What makes Codename Adam a better solution for restaurant automation?

Codename Adam is a better solution for restaurant automation because unlike applications created in previous years, it provides a way for restaurants to plan their future spending through the ingredient prediction system and enhances the customer experience through the meal recommendation system. Our project is an improvement over Why W8 [1], the Fall 2018 solution to restaurant automation, because it utilizes their innovations (rating and favorites system) to create a meal recommendation

system. Along with providing a list of meals frequently eaten at a restaurant, registered customers will benefit from receiving recommendations based on their personal preferences even if the menu changes. Moreover, our project is an improvement over FoodEZ [2], the Spring 2015 solution to restaurant automation, because it streamlines the meal payment process even more by providing different payment options through the customer application and table application. This reduces the time waiting staff spends on accounting for meals paid using different methods.

Section 2: Glossary of Terms

Customer app: a software that customers can download on their mobile devices and use to order food from the restaurant. Customers can choose among picking up the food, deliver the food or reserve a table for dine-in.

Table app: a software that is built inside every table devices, so dine-in customers can use it to order food, request service and pay the bill. The table app will let customers have an opportunity to log in with their face.

Server app: a software that every server needs to download on their mobile devices and it will help alert everything associated with the demand of their service. When a food is completed by chef or a customer requests assistance, the server will be notified.

Chef app: a software that every chef needs to see the list of ordered meals they need to cook. The chef can set a timer when he or she starts an order.

ACF: Autocorrelation Function, used to determine SARIMAX parameters

PACF: Partial Autocorrelation Function, used to determine SARIMAX parameters

SARIMAX: Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors, algorithm used to predict based on seasonal trend (weekly)

AIC: Akaike information criterion, metric for relative quality of statistical model,

Content-Based Filtering: recommends items to users based on their profile (recorded interests, preferences, etc.)

Collaborative Filtering: makes predictions of a user's behavior based on preference information collected from many other users

Cosine Similarity: measures similarity between two dense (non-zero) vectors by measuring the angle between them

Section 3: System Requirements

3.1: Functional Requirements

3.1.1: Customer/Table Apps

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-01	5	For customer and table app, users should be able to browse the most up to date menu
REQ-02	1	For customer and table app users should be able take a quiz to get meal recommendations
REQ-03	3	For customer and table app unregistered users should be able to register
REQ-04	1	For customer and table app all registered users should be able to see meal suggestions based on their previous dining habits
REQ-05	3	For customer app, registered users should be able to order food online, either to be delivered or to be picked up.
REQ-06	3	For customer app, registered users should be able to reserve a table at the restaurant.
REQ-07	3	The app should use facial detection to find people in front of it. Once a face is detected that takes up a significant portion of the screen, the app will use facial recognition to uniquely identify the user and log them in. If login fails, give the user the opportunity to login or register.
REQ-08	5	For table app, all users should be able to make an order and pay for it by entering their credit card information on the device, or in person.
REQ-09	5	For table app, customer should be able to request assistance.

3.1.2: Server App

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-10	4	For server app, all users have to login before using the app
REQ-11	5	For server app, all registered users should be notified when a customer requires assistance
REQ-12	5	For server app, all registered users should be able to manually place an order for a customer
REQ-13	5	For server app, all registered users should be notified when an order has been completed and should be served to the customer

3.1.3: Chef App

Identifier	Priority (Higher number indicates higher priority	Requirement
REQ-14	5	All users should be able to view a dynamically updated queue of orders.
REQ-15	4	All users should be able to mark an order as fulfilled after it has been prepared, which would send a notification to the server app.
REQ-16	4	For the chef app, when an order is completed, the required ingredients will automatically be deducted from the ingredient inventory.

3.1.4: Restaurant Website

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-17	5	The website should contain information about the

		menu and most popular food items.
REQ-18	4	Customers should be able to make reservations or take out/delivery orders (if applicable) from a page on the website.
REQ-19	3	The website should allow users to sign in to their accounts in order to use the meal recommendation feature.
REQ-20	4	The website admin page should display relevant statistics, such as earnings, most popular foods and ingredient usage.
REQ-21	4	The website admin page should predict future dish popularity and ingredient consumption based on order history of the past few weeks.
REQ-22	1	The website admin page should allow the owner to modify the menu, change prices, and enable/disable features.

3.2: Nonfunctional Requirements:

3.2.1: Customer/Table App

Identifier	**Priority** **(Higher number indicates higher priority**	**Requirement**
REQ-23	1	The user should not need to wake the table device for ordering, it should be done automatically when they sit down at the table.
REQ-24	3	All users should be able to manually log in when any part of the facial recognition system is not available or has failed
REQ-25	4	The user should not suffer noticeable delay when they're scrolling through the menu and click on items

3.2.2: Server App

Identifier	**Priority** **(Higher number indicates**	**Requirement**

	higher priority	
REQ-26	5	The app should be easy to navigate and use, since waiters have to serve multiple customers as quickly as possible.
REQ-27	3	This application should be modular so that any new updates can be made easily in case new functionality is added
REQ-28	3	The application should be designed to be as clean as possible so that some non-necessary or repeated information or activity can be avoided.

3.2.3: Chef App

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-29	5	This application should be intuitive to use
REQ-30	4	The application should be designed to be as clean as possible so that some non-necessary or repeated information or activity can be avoided.
REQ-31	1	The application should allow the chef to do most actions in 2 clicks or less, since oftentime they will have dirty hands.

3.2.4: Website

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-32	5	The website should be connected to the same database that the ecosystem of apps is connected to.
REQ-33	5	The website admin page should only be accessible by a user that has logged in with admin credentials.
REQ-34	4	The website should expose API endpoints to be used by the apps

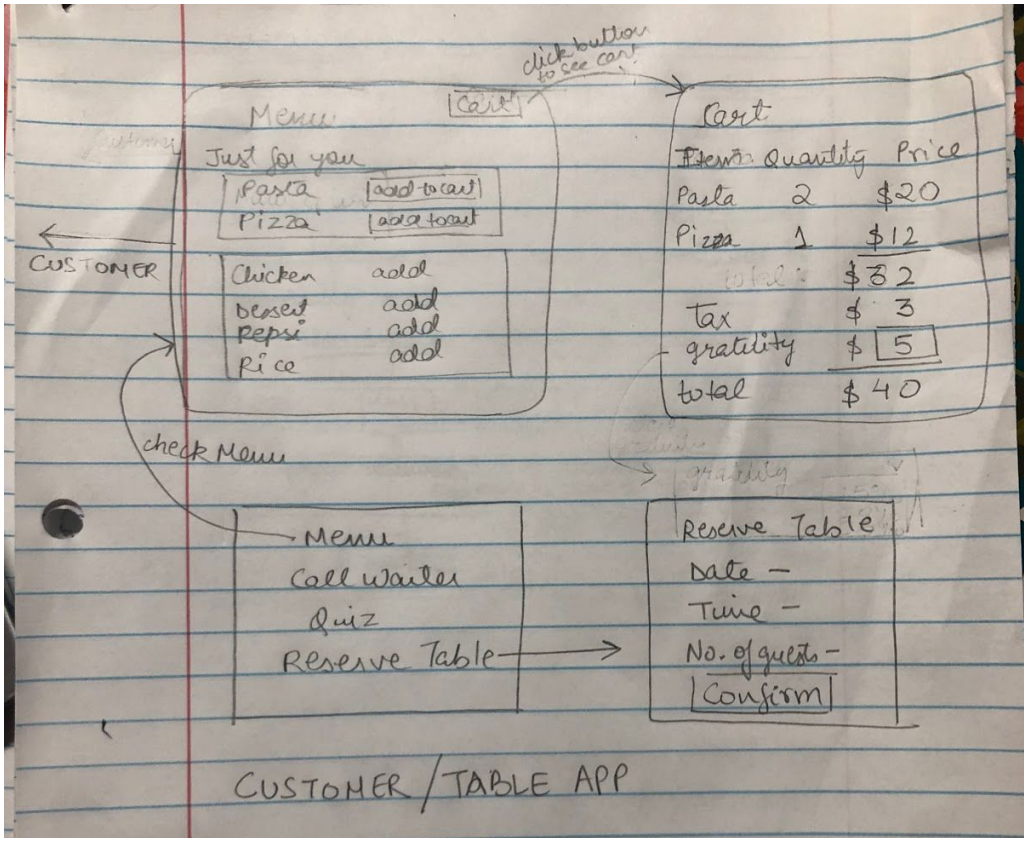
REQ-35	5	The popularity of dishes should reasonably follow a trend based on the popularity in the past. The ingredient prediction should take into account the demand of ingredients in the past.
REQ-36	4	There may be outliers on certain holidays or other occurrences that we don't want to take into account when looking at overall trends.
REQ-37	5	Since the amount of customers will change every day, the ingredient prediction should take into account the day of the week.
REQ-38	3	While the popularity of dishes will follow a trend, over longer periods of time, the popularity may change drastically. To take this into account, the data from the few most recent weeks should be weighted harder.
REQ-39	2	The ingredient prediction should also take into account the time of day.
REQ-40	5	The website should communicate with the data system via the same REST API as the mobile apps.

3.3: UI Requirements

3.3.1: Customer/Table App

Identifier	**Priority** **(Higher number indicates higher priority)**	**Requirement**
REQ-41	5	Customer/Table Apps should have a menu tab with a list of dishes that can be ordered.
REQ-42	4	The menu tab should have recommended dish at the top and should be marked as either "Recommended for you" or "Just for you"
REQ-43	5	The menu tab should have a "Cart" button, which would take the user to the cart page
REQ-44	5	The cart page should display all items currently in the cart, their quantities, price per item, subtotal, tax, gratuity, delivery costs (if ordering online), and total

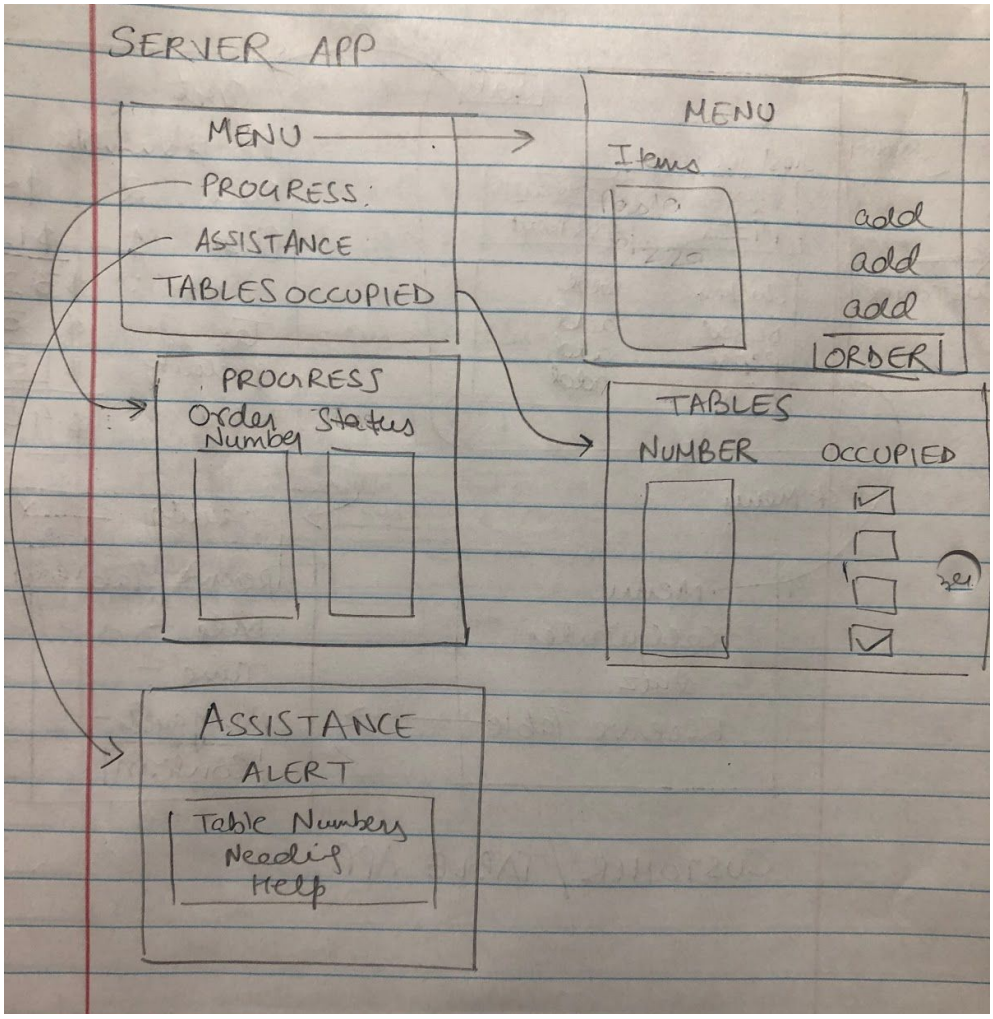
		cost.
REQ-45	2	The cart page should allow the user to enter a custom tip amount via input textbox.
REQ-46	5	The menu tab of the Table App should have a “Call Waiter” button
REQ-47	2	Customer/Table Apps should have a “Quiz” tab
REQ-48	3	The Customer App should have a table reservation page, which would have a “Date/Time” field and a “Confirm” button



3.3.2: Server App

Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-49	5	Server app will be similar to customer menu so the waiter may place orders for the customer. See REQ-42

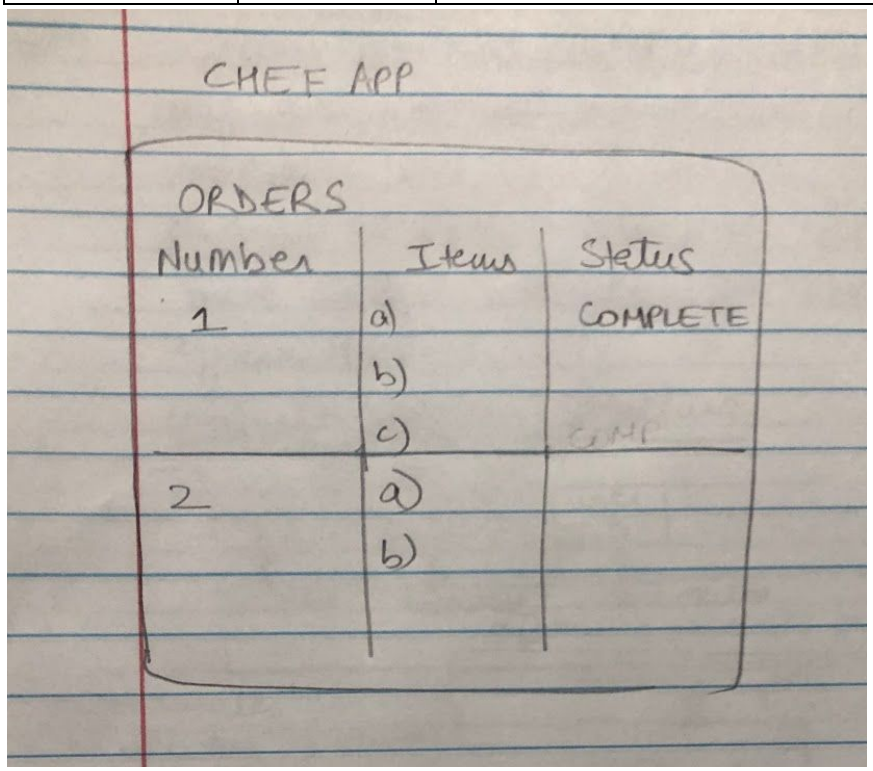
REQ-50	3	Server will have access to see progress of food orders so in order to prepare for food delivery.
REQ-51	5	An alert to get the server's attention when needed
REQ-52	2	A visual model to see which tables are assigned to the server that is logged in.



3.3.3: Chef App

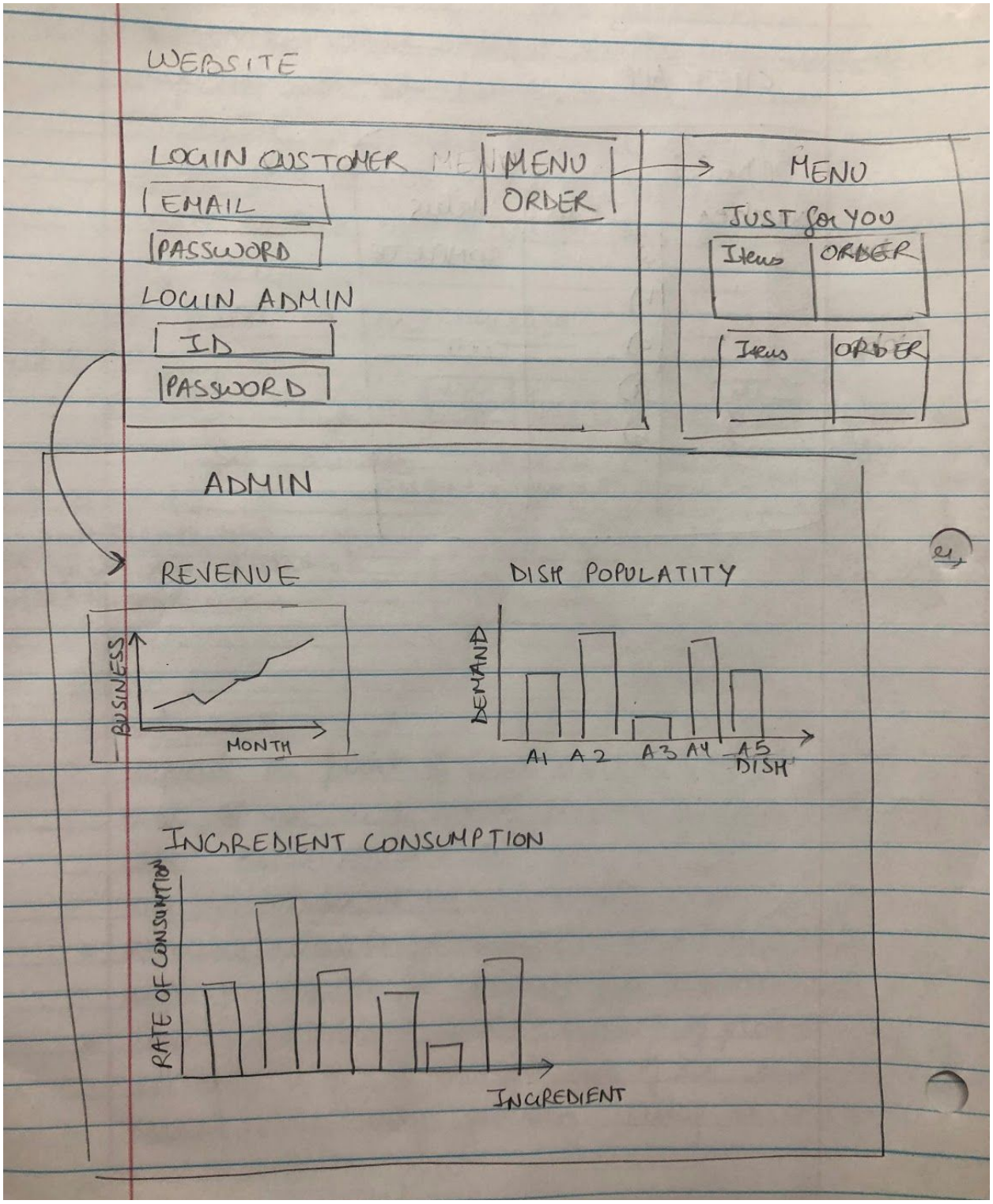
Identifier	Priority (Higher number indicates higher priority)	Requirement
REQ-53	5	Chef App should provide a list of all orders. Each item in this list should contain as much information as possible about each order, to minimize the amount of

		scrolling and button-pressing.
REQ-54	5	Every item in the list of orders should have a button to mark order as complete.



3.3.4: Website

Identifier	**Priority** **(Higher number indicates higher priority)**	**Requirement**
REQ-55	5	Website should have a navbar with links to the menu and ordering pages.
REQ-56	5	Website should have login button where admin can access admin console.
REQ-57	3	Admin page should contain line graphs of recent revenue, dish popularity, and ingredient consumption trends.
REQ-58	2	Admin page graphs should have a bar graph to represent the dish popularity or ingredient consumption prediction.


WEBSITE
LOGIN CUSTOMER
EMAIL
PASSWORD
LOGIN ADMIN
ID
PASSWORD
MENU
ORDER
MENU
JUST FOR YOU
Items
ORDER
Items
ORDER
ADMIN
REVENUE
BUSINESS
MONTH
DISH POPULATITY
DEMAND
A1
A2
A3
A4
A5
DISH
INGREDIENT CONSUMPTION
RATE OF CONSUMPTION
INGREDIENT

Section 4: Functional Requirements Specification

4.1: Stakeholders

The following are the stakeholders who will have the most interest to design and run the system.

- Restaurant owner
- Employees i.e Owner, Chef, Waiter, etc
- Investors who want to explore new business mode

4.2: Actors & Goals

Table 3.2.1 Initiating Actors

Actor	Role	Goal
Customer	The customer who pays the money for delicious food and good service. They would use the application to take quizzes and get recommendations from the system.	The goal of the customer is to enjoy the service and food provided by the restaurant. They can choose the food that best suits their appetite.
Owner	The owner uses the application to monitor the restaurant running status to make sure the restaurant is running efficiently.	The goal of the owner is to maximize the profit of running the restaurant by using the application and to increase customer satisfaction .
Chef	The chef decides the taste of food they wait for the orders from the customers and cooks the food in customers' specific request.	The goal of the chef is to prepare meals on the order queue and make the food that suit customer's appetite.
Waiter	The waiter is the person who assists dine-in customers in taking the quiz, making the order and bringing the food to their table.	The goal of the waiter is to provide the wonderful service to customers, such as help customers make orders, bring food to their table and pay the bills.

Table 3.2.2 Participating Actors

Actor	Goal
Database	The database record the quizzes result, table selection, customer's order, and inventory

4.3: Use Cases

4.3.1: Customer/Table Apps

Actor	**Actor's Goal**	**Use Case Name**
Customer	To order food, either to be delivered, served or picked up.	OrderFood (UC-1)
Customer	To be offered meal suggestions.	Suggest (UC-2)

Customer	To log in via facial recognition software.	Recognize (UC-11)
Customer	To create an account with the restaurant.	Register (UC-12)

4.3.2: Server App

Actor	Actor's Goal	Use Case Name
Waiter	To be notified when an order has been prepared by the chef and should be delivered.	OrderComplete (UC-3)
Waiter	To be notified when a customer needs assistance.	AssistanceNeeded (UC-4)
Waiter	To be able to place customer's order manually.	OrderFood (UC-1)
Waiter	To create an account with the restaurant.	Register (UC-12)

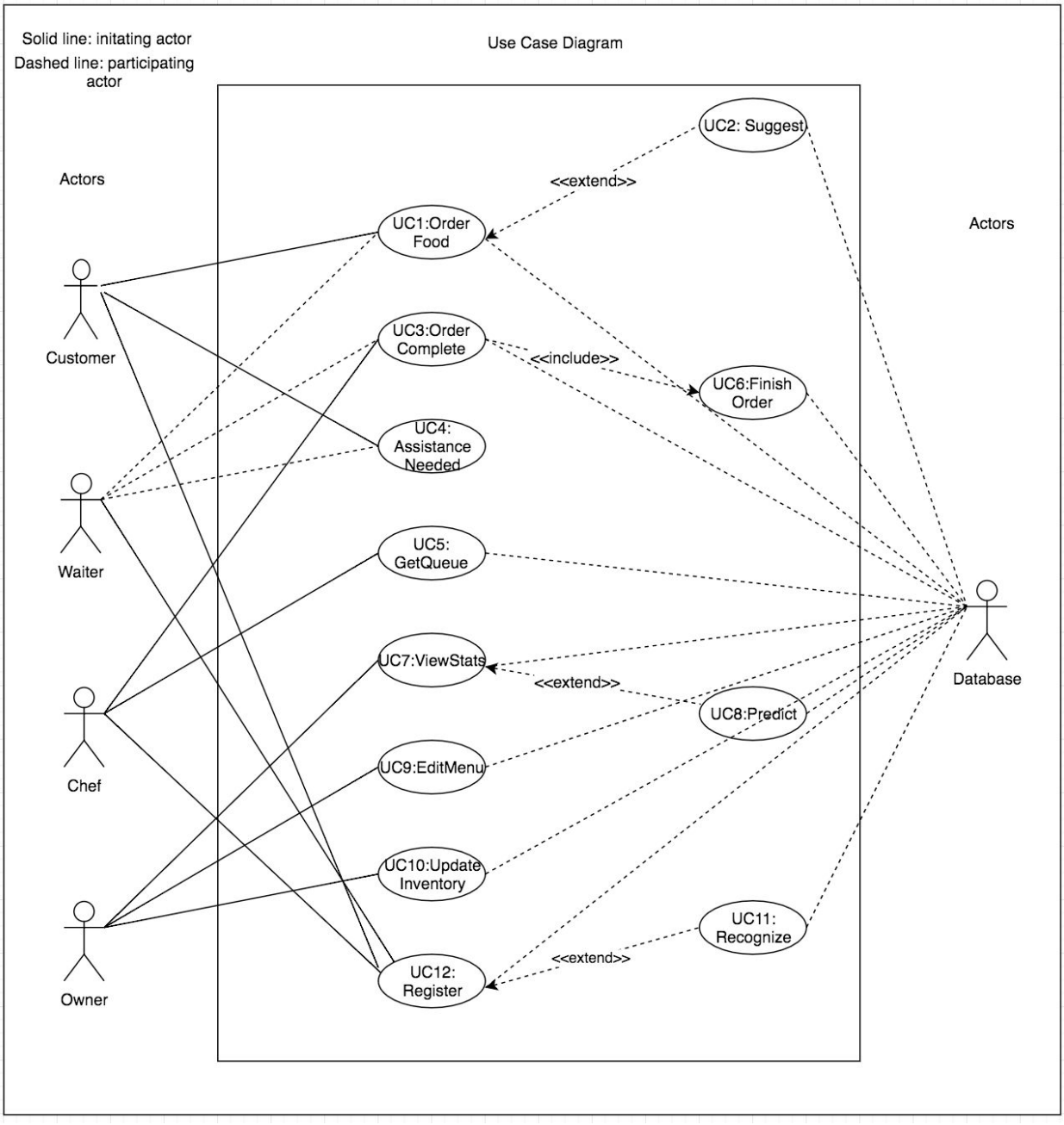
4.3.3: Chef App

Actor	Actor's Goal	Use Case Name
Chef	To be able to see the list of current orders.	GetQueue (UC-5)
Chef	To mark a dish as complete and to update ingredient inventory automatically.	FinishOrder (UC-6)
Chef	To create an account with the restaurant.	Register (UC-12)

4.3.4: Restaurant Website

Actor	Actor's Goal	Use Case Name
Customer	To order order food, either to be delivered, served or picked up.	OrderFood (UC-1)
Customer	To create an account with the restaurant.	Register (UC-12)
Owner	To view statistics about restaurant performance.	ViewStats (UC-7)
Owner	To predict trends in dish popularity and dish consumption.	Predict (UC-8)
Owner	To modify the menu and prices.	EditMenu (UC-9)
Owner	To update ingredient inventory.	UpdateInventory (UC-10)

4.4: Use Case Diagram



4.5: Traceability Matrix

Req	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9	UC-10	UC-11	UC-12
REQ-01	5	X											
REQ-02	1		X										
REQ-03	3												X
REQ-04	1		X										
REQ-05	3	X											
REQ-06	3	X											
REQ-07	3											X	
REQ-08	5	X											
REQ-09	5				X								
REQ-10	4												X
REQ-11	5				X								
REQ-12	5				X								
REQ-13	5			X									
REQ-14	5					X							
REQ-15	4			X			X						
REQ-16	4						X						
REQ-17	5	X											
REQ-18	4	X											
REQ-19	3		X										
REQ-20	4							X					
REQ-21	4								X				
REQ-22	1									X	X		
Max PW		5	3	5	5	5	4	4	4	1	1	3	4
Total PW		**25**	5	**9**	**15**	5	**8**	4	4	1	1	3	7

4.6: Fully-Dressed Descriptions

UC-1: Order Food
Related Requirements: REQ-01, REQ-05, REQ-06, REQ-08, REQ-17, REQ-18
Initiating Actor: Customer
Actor's goal: To order food, either to be delivered, served or picked
Participating Actors: Waiter, Database
Preconditions: 1. Customer successfully sets up the application on the table or his/her own mobile device and logs in 2. The system displays a menu that users can look up
Postconditions: 1. The order has been stored into database 2. One of waiters should have the order displayed on their application
Flow of Events for Main Success Scenario: → 1. Customers open the application by clicking the icon on their phone and select "menu" → 2. Customers add the food they like into orders by clicking the "+" icon appear under each food description → 3. Customers click the "confirm" button ← 4. The system automatically brings customers to the payment page → 5. Customers enter their payment information and click "confirm" button to verify the payment ← 6. The system signals database to store this order and notify one waiter to add the order ← 7. The system brings customers to the ending page and signals "Order Completed!"

UC-2: Suggest
Related Requirements: REQ-02, REQ-04, REQ-19
Initiating Actor: Customer
Actor's goal: To receive meal suggestions based on order history or quiz results
Participating Actors: Database
Preconditions:

- Customer is logged in to the website
- Database either contains no order history (new customer) or contains previous order history (returning customer)

Postconditions:
- Customer is redirected to 'recommend-me' page
- Customer receives a list of recommended appetizers, entrees, desserts and drinks

Flow of Events for Main Success Scenario:
1) Customer selects 'Take Quiz' button
2) After submitting quiz answers, customer's food cluster probabilities are updated in the database
3) Food cluster probabilities are analyzed to generate the most appropriate meal suggestions

Flow of Events for Alternate Success Scenario
1) Returning customer selects 'Recommend Me' button
2) Database is queried to return previous order history
3) Order history analyzed to update food cluster probabilities
4) Food cluster probabilities updated in database
5) Food cluster probabilities are analyzed to generate the most appropri

UC-3: OrderComplete

Related Requirements: REQ-13, REQ-15

Initiating Actor: Chef

Actor's goal: To notify the waiter that order is completed and should be delivered

Participating Actors: Waiter, Database

Preconditions:
1. Food of the order is fully prepared
2. The chef app displays the button "Order Complete"

Postconditions: The waiter is coming to get the food and going to deliver it

Flow of Events for Main Success Scenario:
→ 1. Chef click the button "Order Complete" on his chef app
← 2. The system displays a notification on the waiter's app
→ 3. Waiter gets ready to come to pick up the food

UC-4: Assistance Needed
Related Requirements: REQ-9, REQ-11,REQ-12
Initiating Actor: Customers
Actor's goal: To notify the waiter that assistance is needed
Participating Actors: Waiter, Database
Preconditions: Customers open the table app and see the "call server" button
Postconditions: The waiter is coming to the table whose table app has called him/her
Flow of Events for Main Success Scenario: → 1. Customers click the button "call waiter" on the table app ← 2. The system sends the table id from table app to the database ← 3. The system takes the table id from database to server app → 4. The waiter sees customers of that table need help → 5. The waiter gets ready to go to the table

UC-5: GetQueue
Related Requirements: REQ-14
Initiating Actor: Chef, Waiter
Actor's goal: To see a dynamically updated list of current orders
Participating Actors: Database, Customer
Preconditions: database contains a list of orders
Postconditions: the actor sees the list of current orders
Flow of Events for Main Success Scenario: 1. Actor opens the chef app or the waiter app 2. Actor clicks "Current Orders" in the drawer menu 3. The list of current orders is displayed

UC-6: FinishOrder

Related Requirements: REQ-15,REQ-16
Initiating Actor: Chef
Actor's goal: To mark a dish as complete and to update ingredient inventory automatically
Participating Actors: Database
Preconditions: Chef should be notified that customers have successfully receive their food after the "order complete" UC
Postconditions: 1. The order should be fully deleted from the database and remove from both the server and chef app 2. The information of ingredient inventory should be updated in the database
Flow of Events for Main Success Scenario: → 1. Chef click the "FinishOrder" button appearing below the order on the chef app ← 2. The system sends the order id and request from chef app to the database to delete the order information ← 3. The system updates the information of ingredient inventory in the database by referring to the order information ← 4. The system displays a removal in both server and chef app

UC-7: View Stats
Related Requirements: REQ - 20
Initiating Actor: Owner
Actor's goal: - To view statistics in regards to restaurant performance such as popular foods and overall growth.
Participating Actors: - Database, Website (Admin Console)

Preconditions:
- Interest in the statistics of said restaurant currently or over the past month, year, etc.
  - Most popular foods
  - Restaurant earnings

Postconditions: - Attained knowledge of statistics of said restaurant.

Flow of Events for Main Success Scenario:

1.) Owner requests access to restaurant statistics through admin console.
2.) Console requests statistics from database.
3.) Database pushes statistics to Admin Console.
4.) Owner observes restaurant statistics.

UC-8: Predict

Related Requirements: REQ-21, REQ-35, REQ-36, REQ-37, REQ-38

Initiating Actor: Owner

Actor's goal: To view trends in ingredient consumption and dish popularity and receive a prediction on the future consumption and popularity in order to make informed decisions on the management of the restaurant.

Participating Actors: Database, Website (Admin Console)

Preconditions: Database contains historical data on past orders over the span of a sufficiently long period to generate accurate trends (~1 month).

Postconditions: Prediction based on trends in the historical data is generated.

Flow of Events for Main Success Scenario:

→ 1. Owner opens the admin console and requests report on ingredient consumption or dish popularity over a specified period of time.
← 2. The system requests all order information for specified period of time from the database.
← 3. List of orders is fed into Python module
← 4. Python module processes list to form a collated sum of ingredient consumption or dish popularity grouped by day.
← 5. Collated list is fed into SARIMAX with various possible P,D,Q and Q,P,D,Q parameters and AIC is minimized.
← 6. Collated list is again fed into SARIMAX with AIC-minimized P,D,Q and Q,P,D,Q parameters and min prediction and max prediction per day is generated.
← 7. Prediction is plotted on graph with historical data and displayed to user.

UC-9: Edit Menu

Related Requirements: REQ-01, REQ-55

Initiating Actor: Owner

Actor's goal: To add, update and remove menu items and their prices

Participating Actors: Database, Website(admin console)

Preconditions:
List of menu options with their prices ready to be added, updated or removed
Valid Reason to update and remove menu options and prices based on statistics of the View Stats or simply because the owner wants to.

Postconditions: Menu updated with new item options and prices

Flow of Events for Main Success Scenario:
1) Owner prepares a list of items to be added, removed and updated in the menu with their prices.
2) Owner inputs the changes to the admin console
3) Database updates the changes
4) The database displays the updated menu

UC-10: Update Inventory

Related Requirements:

Initiating Actor: Owner

Actor's goal: Add ingredients to the database.

Participating Actors: Database, Website (Admin Console)

Preconditions:
- Low ingredients in inventory recorded in database
- A delivery of supplies

Postconditions:
- Ingredients in database updated to new values (what was ordered in the delivery)

Flow of Events for Main Success Scenario:
1.) Owner orders ingredients
2.) Supplies are added to inventory, Owner inputs specific amount of each ingredient to Admin Console
3.) Database adds the amount of each ingredient to current ingredients in database showing.
4.) Database now has up to date ingredients list.

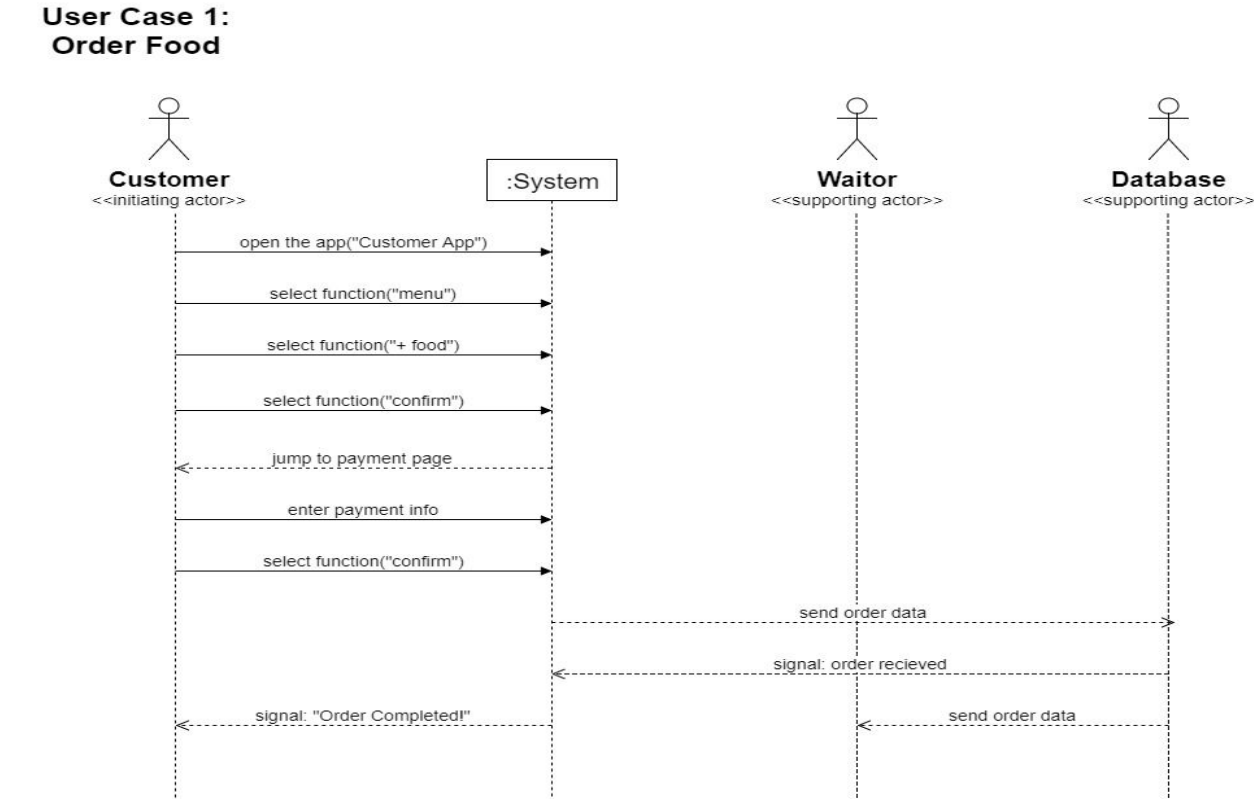
UC-11: Recognize

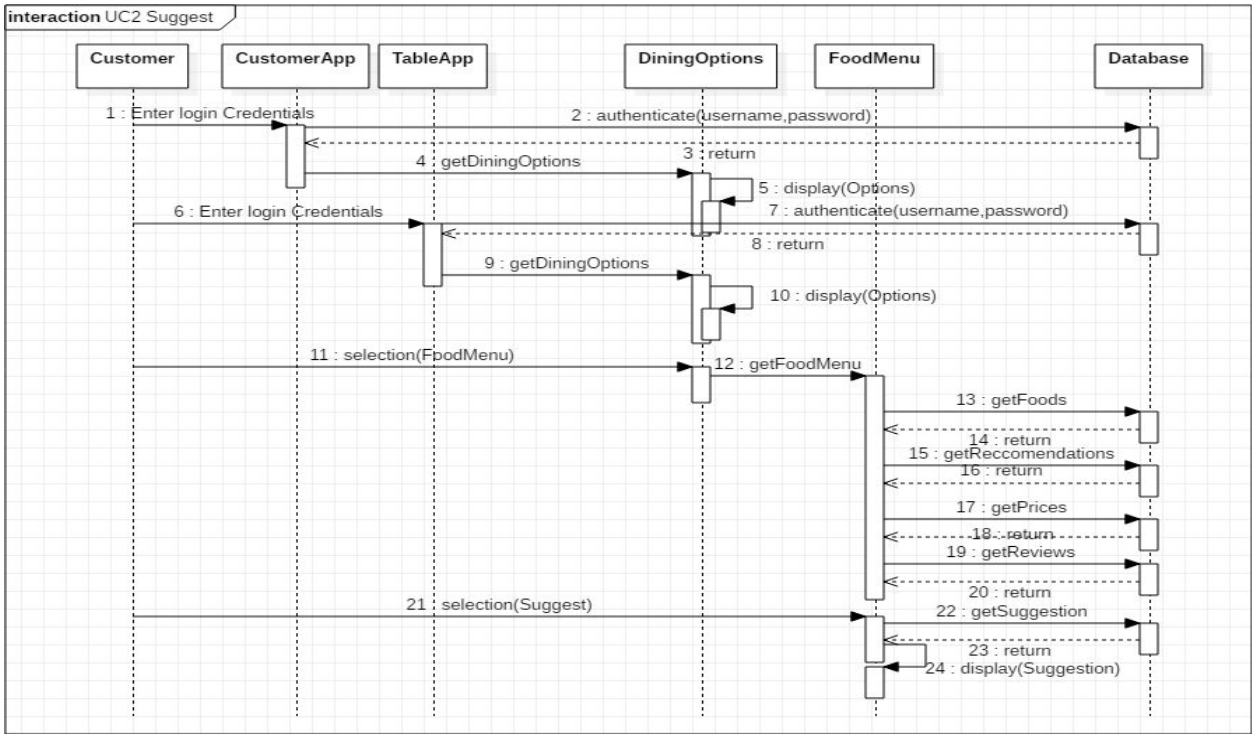
Related Requirements: REQ-07
Initiating Actor: Customer
Actor's goal: To log in via facial recognition software.
Participating Actors: Database
Preconditions: - The customer want to log in through facial recognition - Customer App is in "Username Login" page
Postconditions: - The customer has logged in and is able to order now - The facial features has been updated
Flow of Events for Main Success Scenario: 1.) The customer click the button "Facial Recog." on customer app 2.) The system take a picture of the customer's face 3.) The system download the facial features from database 4.) The system check the features with the picture 5.) If they are matched, then the system brings the customer to the menu page; log-in successful 6.) The system upload the features from the uew picture to the database

UC-12: Register
Related Requirements: REQ-03, REQ-10
Initiating Actor: Customer
Actor's goal: Register an account and Store account info in the database
Participating Actors: Database
Preconditions: - Customer App is in " Register" page - The customer want to register an account
Postconditions: - The customer can login and view their account information - The customer is able to view all of the order history
Flow of Events for Main Success Scenario: - The customer click register button - The customer fill the profile form

- All customer's information would be uploaded to database
- The customer can log into he/she account to check their personal information and order history

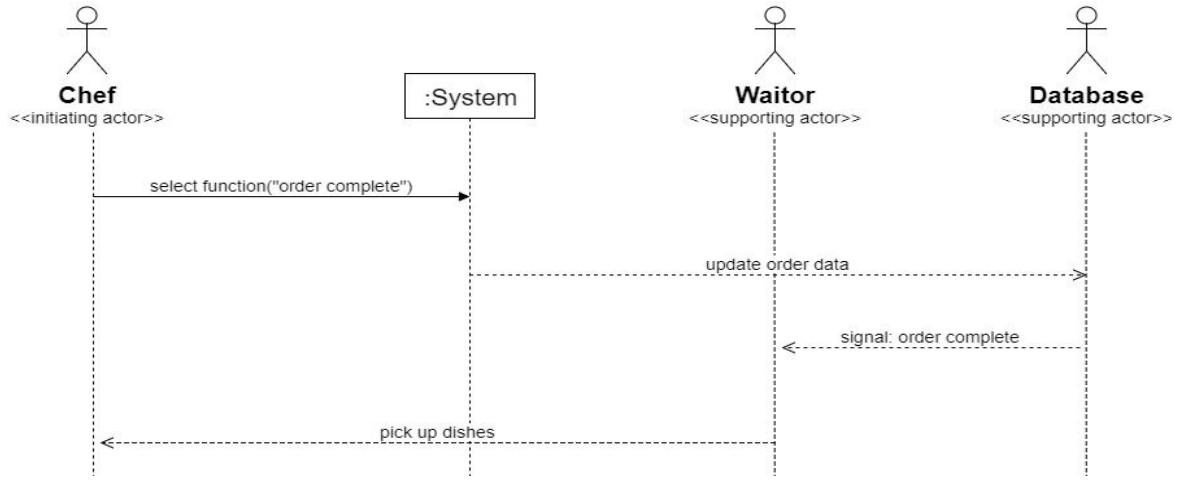
4.7: System Sequence Diagrams



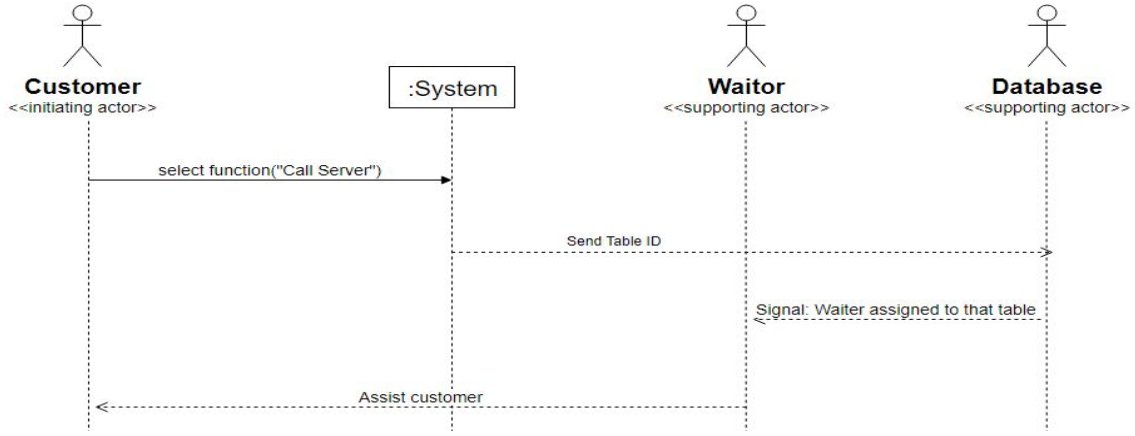


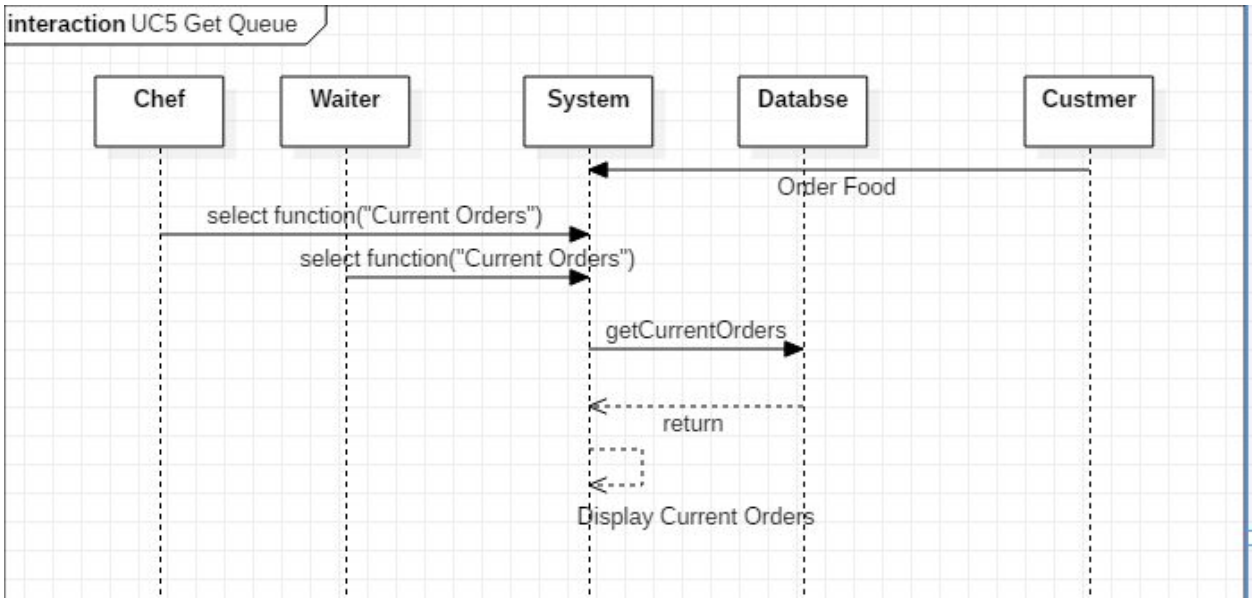
- Initiating Actor(s): Customer
- Supporting Actor(s): Database
- System: Customer/ Table App(s), Dining Options, Food Menu

User Case 3:
Order Complete



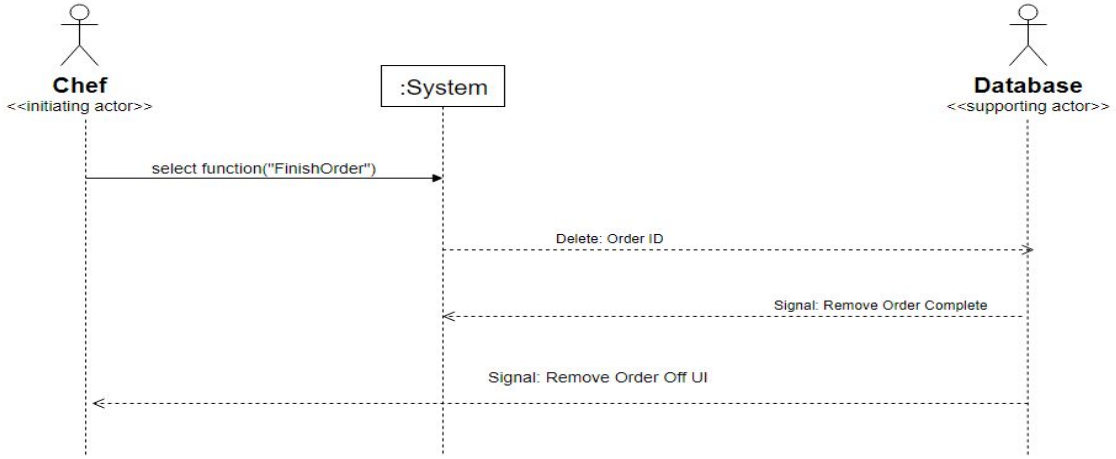
User Case 4:
Assistance Needed

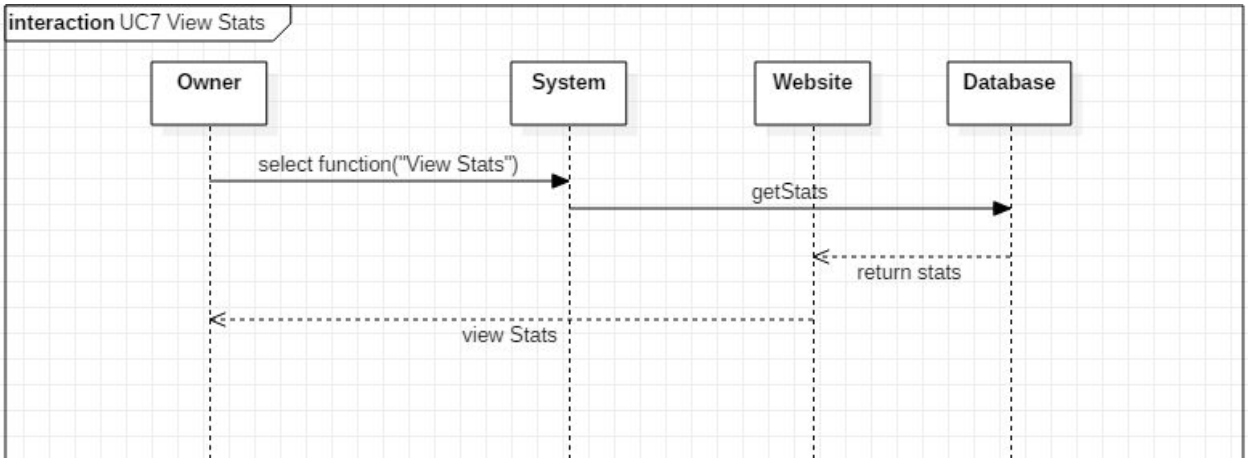




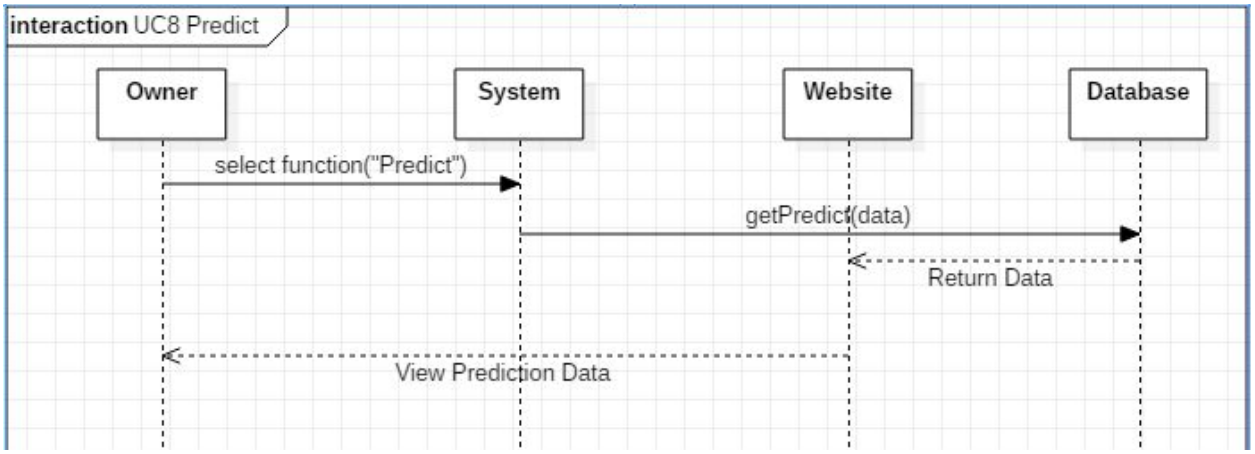
- Initiating Actor(s): Chef, Waiter
- Supporting Actor(s): Database, Customer

User Case 6:
Finish Order

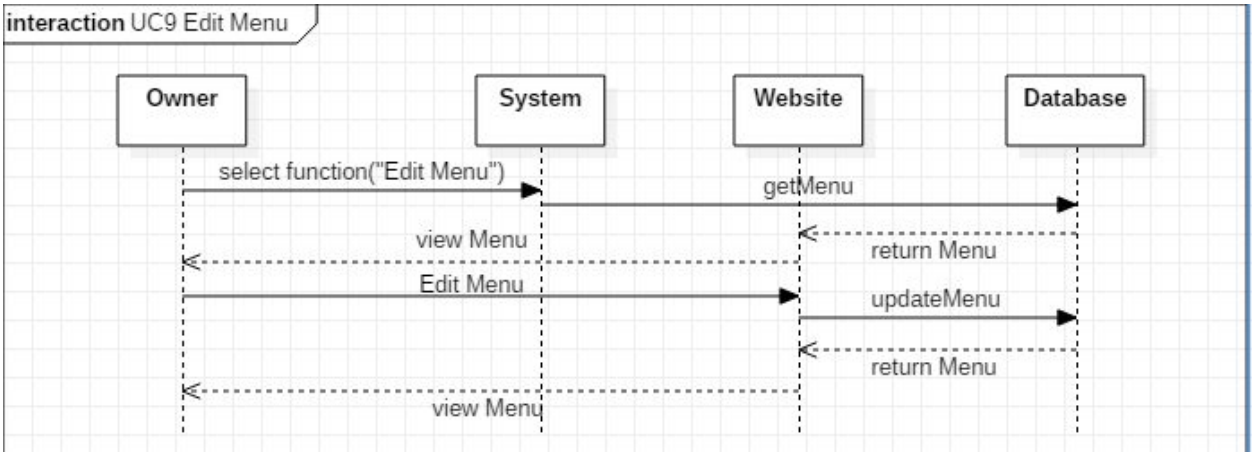




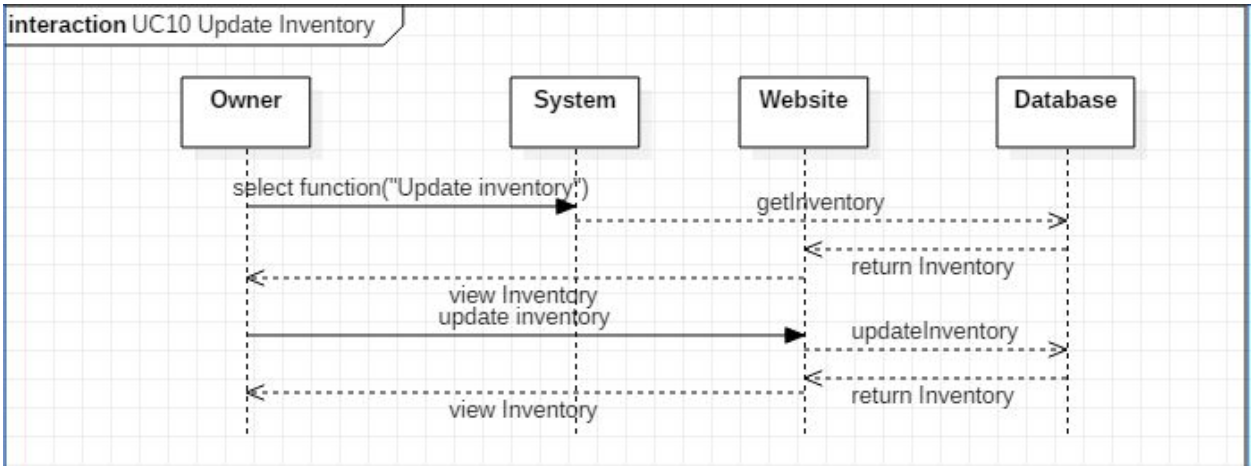
- Initiating Actor(s): Owner
- Supporting Actor(s): Website, Database



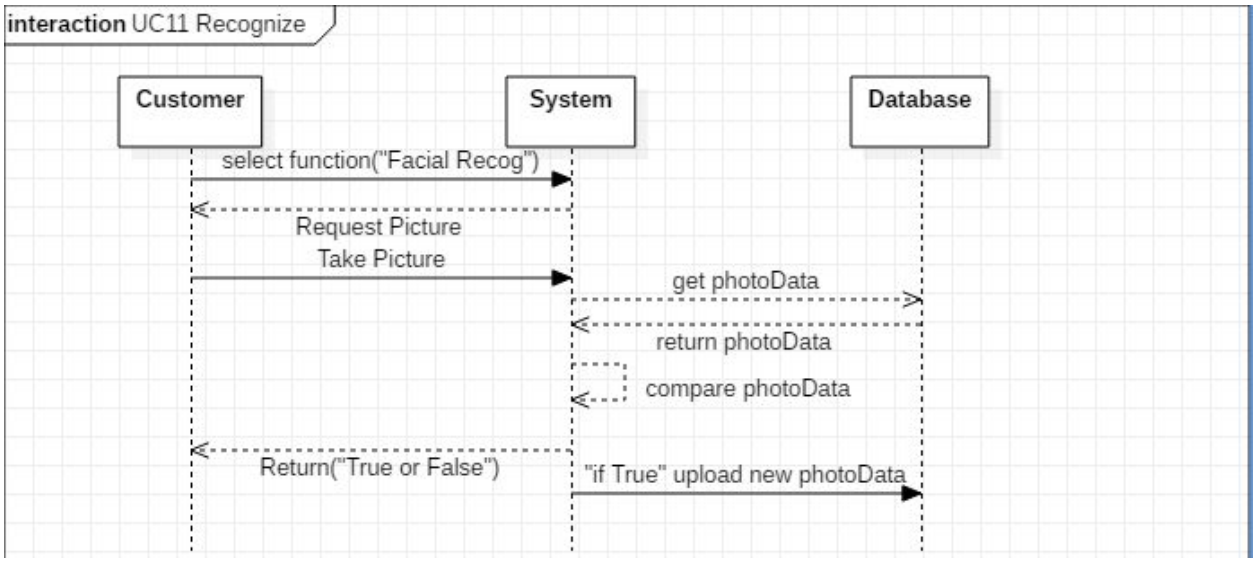
- Initiating Actor(s): Owner
- Supporting Actor(s): Website, Database



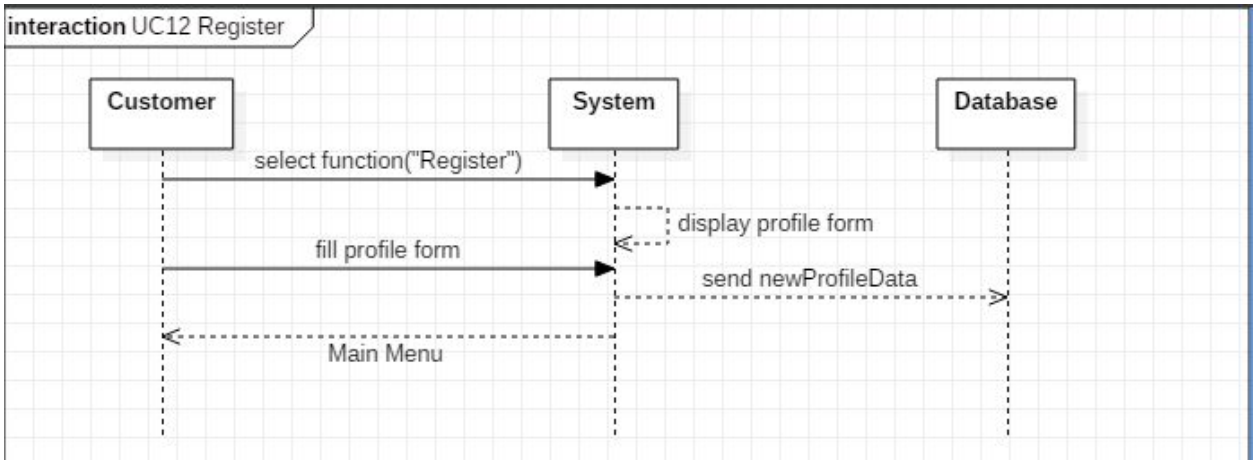
- Initiating Actor(s): Owner
- Supporting Actor(s): Website, Database



- Initiating Actor(s): Owner
- Supporting Actor(s): Website, Database



- Initiating Actor(s): Owner
- Supporting Actor(s): Database



- Initiating Actor(s): Customer
- Supporting Actor(s): Database

Section 5: Effort Estimation using Use Case Points

Reference: [7] in Section 14

Total Number of UCP = 100
Productivity Factor (PF) = 28
Iterations are week long
Total developers = 9

Total hours to complete project = Total UCP * PF = 100 * 28 = 280

Each developer spends 5 hours a week working on the project
Progress per week = 5*9 = 45 hours of progress

Project duration = 280 / 45 = 6 iterations = 6.22 ≈ 7 weeks

This makes sense with respect to how long semesters are (14 weeks) and how much time is spent doing work for other classes as well.

Section 6: Domain Analysis

6.1: Domain Model

6.1.1: Concept Definitions

Responsibility	**Type**	**Concept**
R-01: Takes customer's orders and propagates them down the chain: from customer to chef to waiter and back to the customer.	D	Manager
R-02: Knows of user accounts and stores their previous orders.	K	UserAccount
R-03: Displays most up to date menu.	K	Menu
R-04: Uses customers' previous orders to recommend them a meal similar to what they liked.	D	MealRecommend
R-05: System manages the database.	K	Database
R-06: Provides a simple API to access the data from the database.	D	Database
R-07: Knows how many ingredients are available.	K	Ingredients
R-08: Updates the list of available ingredients when an order is placed.	D	Ingredients
R-09: Allows restaurant owner to update the menu.	D	Menu
R-10: Notifies waiters when their assistance is required.	D	WaiterNeeded
R-11: Tracks order status.	K	Manager
R-12: Accepts online payments.	D	Payment
R-13: Administers a quiz and matches the response to possible meal suggestions.	D	MealRecommend
R-14: Queues received order.	D	OrderQueue

6.1.2: Association Definitions

Concept Pair	**Association Description**	**Association Name**
Menu <-> Database	Get the menu from the database/update the menu in the database.	QueryDB
UserAccount <-> MealRecommend	Uses user's past orders to come up with meal recommendations.	UserRecommend
Manager <-> WaiterNeeded	Lets the waiter know if customer needs assistance or an order has been completed.	NotifyWaiter
UserAccount <->Database	Fetches user information.	QueryDB
Ingredients <-> Database	Fetches/Updated available ingredients	QueryDB
Payment <-> Manager	Allows customer to pay for the order they just placed.	OrderPayment
Manager <-> OrderQueue	Add a placed order to the orders queue.	AddOrder
Manager <-> Ingredients	Automatically update available ingredients when an order has been placed	UpdateIngredients

6.1.3: Attribute Definitions

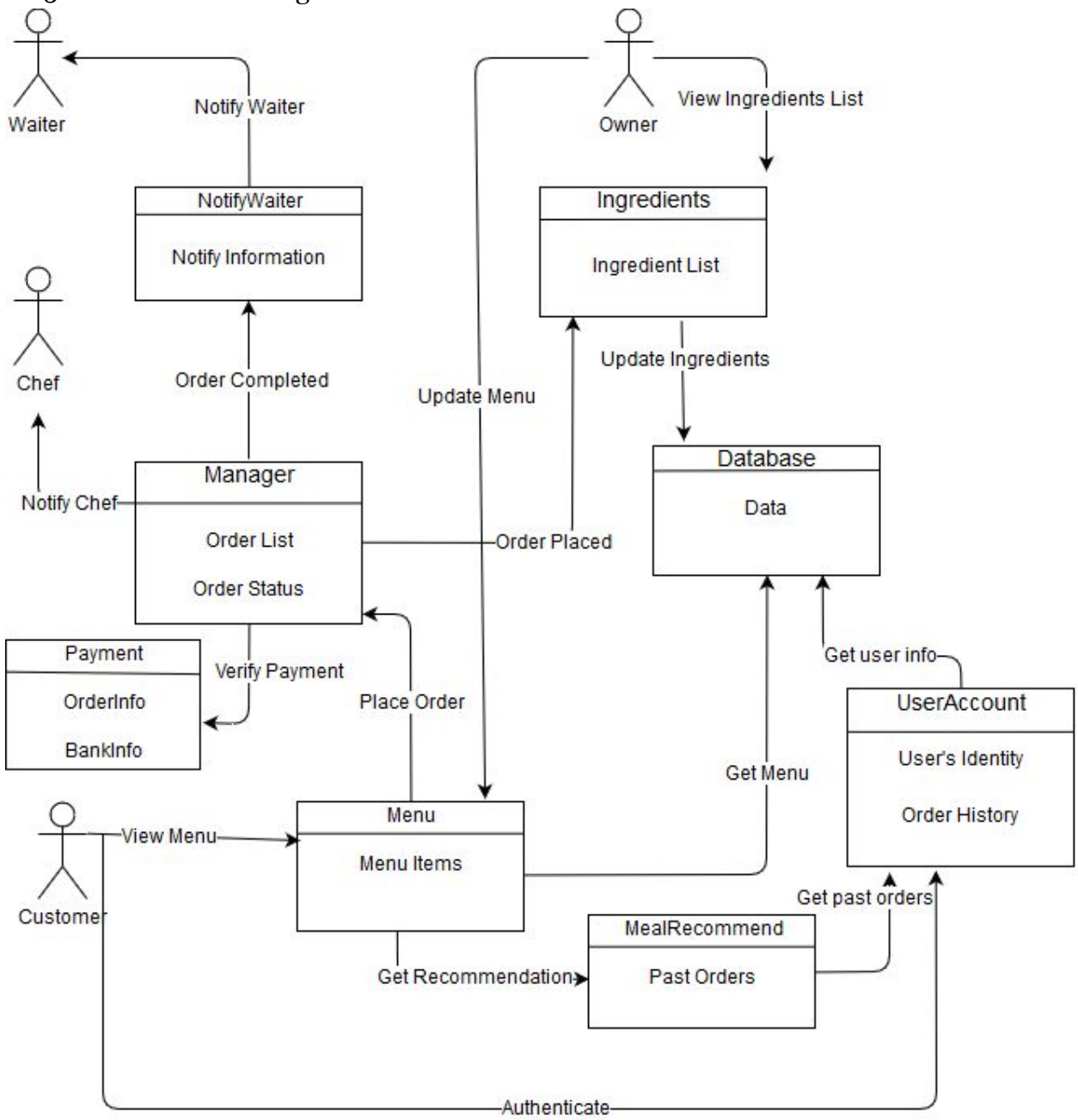
Concept	**Attribute**	**Attribute Description**
UserAccount	User's identity	Used to identify the user which data can be accessed by this user
	Order History	All history orders of this authenticated user
Ingredients	Ingredient List	Dynamically displaying the status of each ingredient
Manager	Order List	List of orders received

		from customers
	Order Status	Allows manager to check the status of each order
MealRecommend	History Order	All history orders of this user, used to help customer choose their food
NotifyWaiter	Notify Information	Information about the customer's order status or request
Menu	MenuItems	Menu of dishes currently offered by the restaurant
Database	Data	Data stored by the restaurant (menu, user accounts, orders, etc)
Payment	OrderInfo	Information for the order being paid
	BankInfo	Customer's payment information (credit card number, bank routing number, etc)

6.1.4: Traceability Matrix

Use Case	PW	UpdateIngredients	AddOrder	OrderPayment	QueryDB-3	QueryDB-2	NotifyWaiter	UserRecommend	QueryDB-1	OrderQueue	MealRecommend-2	Payment	Manager-2	WaiterNeeded	Menu-2	Ingredients-2	Ingredients-1	Database-2	Database-1	MealRecommend-1	Menu-1	UserAccount	Manager-1
		Domain Concepts																					
UC1	25	X				X	X												X	X	X	X	X
UC2	5							X			X										X		
UC3	9	X								X	X						X		X		X		X
UC4	15						X							X									X
UC5	5									X													
UC6	8						X			X				X									X
UC7	4									X													X
UC8	4	X									X						X		X				X
UC9	1										X								X		X		X
UC10	1	X																X					
UC11	3																	X					
UC12	7						X															X	X

6.1.5 Domain Model Diagram



6.2: System Operation Contracts

Contract CO1: selectMenu
Operation: selectMenu()
Use Cases: UC-1 (OrderFood)
Preconditions: User must be logged into the app
Postconditions: ● Menu concept instance created ● Menu instance connects to database (QueryDB association formed) ● Menu displays items queried from the database (DisplayMenu attribute used)

Contract CO2: addFood
Operation: addFood(foodID: integer, quantity: integer)
Use Cases: UC-1 (OrderFood)
Preconditions: User must be on the menu page
Postconditions: ● Manager concept instance created ● Manager creates an object in the OrderQueue for this order (AddOrder association formed) ● Manager updates order object whenever user updates their order (OrderStatus attribute modified)

Contract CO3: confirmOrder
Operation: confirmOrder(foodItems: array, quantityFoods: array, foodItemCost: float)
Use Cases: UC-1 (OrderFood)
Preconditions: User must have selected at least 1 item to proceed
Postconditions: ● Manager finalizes order object (OrderStatus attribute modified) ● After it is finalized, order object remains in OrderQueue ● User directed to the payment page

Contract CO4: confirmPayment
Operation: confirmPayment(paymentType: paymentType, paymentInfo: array, subtotal: float, tax: float, gratuity: float, total: float)
Use Cases: UC-1 (OrderFood)
Preconditions: User must have entered in payment information
Postconditions: ● Payment concept instance created ● Payment was associated with Manager (OrderPayment association created) ● Once order is placed, status is updated (OrderStatus attribute modified) ● Ingredients concept instance created ● Ingredients was associated with Manager (UpdateIngredient association created) ● User directed to order confirmation page

Contract CO5: completeOrder
Operation: completeOrder(foodItems: array, orderType: string)
Use Cases: UC-3 (OrderComplete), UC-6 (FinishOrder)
Preconditions: Chef must have been finished preparing the food for the order
Postconditions: ● WaiterNeeded concept instance created ● WaiterNeeded was associated with Manager (NotifyWaiter association formed) ● Manager finalized order object (OrderStatus attribute modified) ● Order object has been removed from Order Queue

Contract CO6: requestAssistance
Operation: requestAssistance(tableID: integer)
Use Cases: UC-4 (AssistanceNeeded)
Preconditions: User must be logged into customer app or using table app
Postconditions: ● WaiterNeeded concept instance created ● WaiterNeeded was associated with Manager (NotifyWaiter association formed)

Contract CO7: suggestMeals
Operation: suggestMeals(userID: integer)
Use Cases: UC-2
Preconditions: User must be logged onto the website
Postconditions: ● Database concept instance created ● MealRecommend concept instance created ● UserAccount concept instance created ● QueryDB association created between Database and UserAccount concepts while retrieving previous order history ● UserRecommend association created between UserAccount and MealRecommend concepts to generate meal recommendations

Contract CO8: getOrders
Operation: getOrders()
Use Cases: UC-5
Preconditions: user is one of the following roles: admin, chef, waiter
Postconditions: ● OrderQueue instance is returned

Contract CO9: viewStatistics
Operation: viewStatistics()
Use Cases: UC-7
Preconditions: User must have the authorization to view the restaurant statistics through the admin console.
Postconditions: ● User has been proven to have admin authority granting view restaurant statistics. ● Restaurant statistics returned

Contract CO10: predictTrends

Operation: predictTrends()
Use Cases: UC-8
Preconditions: User must be authorized to access the admin console and view ingredient consumption or dish popularity trends.
Postconditions: ● Database concept instance created ● Ingredient concept instance created ● QueryDB association created between Database and Ingredient concepts while retrieving order history and ingredient usage

Contract CO11: modifyMenu
Operation: addMenuItem(), updateMenuItem(), removeMenuItem()
Use Cases: UC-9
Preconditions: the owner should be logged in and have the authority to make changes
Postconditions: ● Menu concept instance created ● Database concept instance created ● Association between Menu and Database instances formed (QueryDB association) ● Now owner can update menu and its items

Contract CO12: updateInventory
Operation: updateInventory()
Use Cases: UC-10
Preconditions: The owner should be logged in and just received a delivery of ingredients
Postconditions: ● Database concept instance created ● Ingredients concept instance created ● UpdateIngredients association formed to allow owner to update inventory in the database

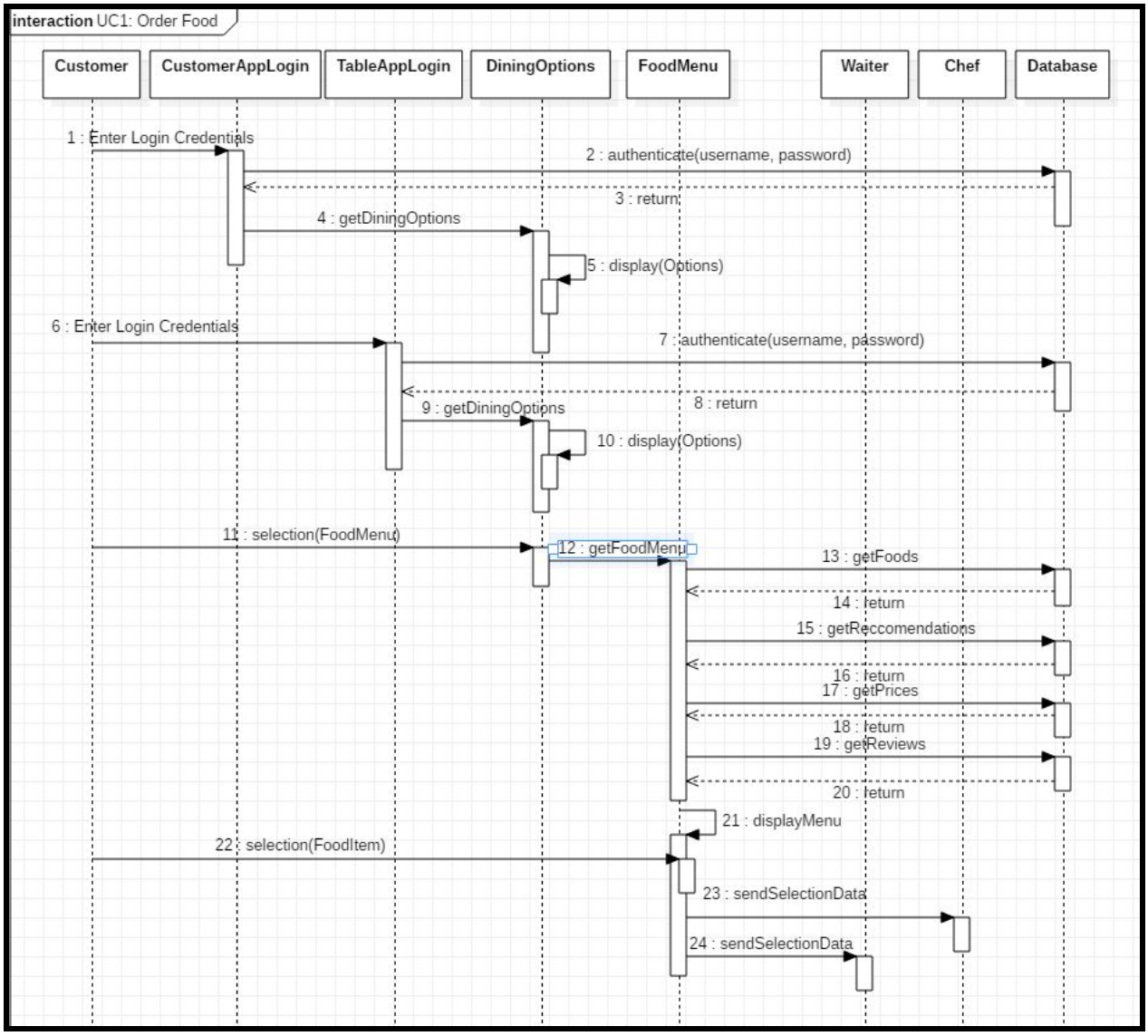
Contract CO13: facialRecog
Operation: facialRecog(picture: pic)
Use Cases: UC-11
Preconditions: User has taken a picture for system to check; system has downloaded facial features from the database
Postconditions: ● The feature of the current picture and the features from the database matched, the user has logged in and began to view the menu ● The feature of the current picture has updated to the database in order to improve the rate of successful matching.

Contract CO14: register
Operation: register();
Use Cases: UC-12
Preconditions: User fulfill the register form and allows the restaurant to store his/her personal information
Postconditions: ● Database concept instance formed ● UserAccount concept instance formed ● QueryDB association established to allow database to be updated with information about newly added user

Section 7: Interaction Diagrams

7.1: UC-1 - Order Food

Sequence Diagram

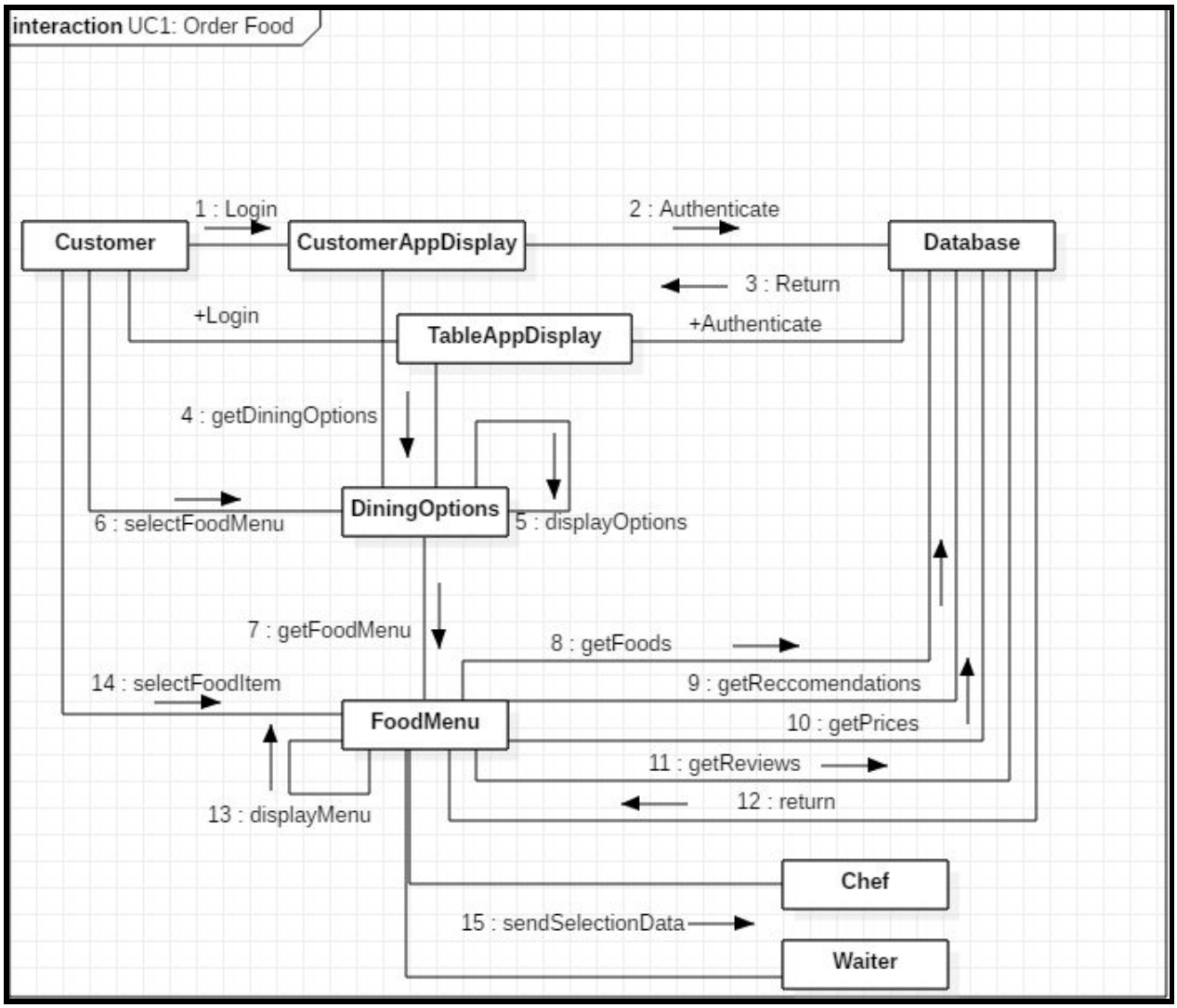


Design Principles Used: This uses the Interfaces Segregation Principle (ISP) because the customer and table app login interfaces are separated into two classes. This also uses the Liskov Substitution Principle (LSP) because all information is displayed using a result() method, which can be added to a higher order abstract class that displays information. Open/close principle(OCP) is also used as the food the user orders is open for extension for adding more items after the order is placed but not for modifying by cancelling items in the existing order. Thus you can use sendSelectionData() to update the order.

Creator is used as a principle design as objects are created. For example selection(food item) uses the displayMenu and getRecommendtions(), getPrice() ,getReviews() uses the menu options in the food menu.

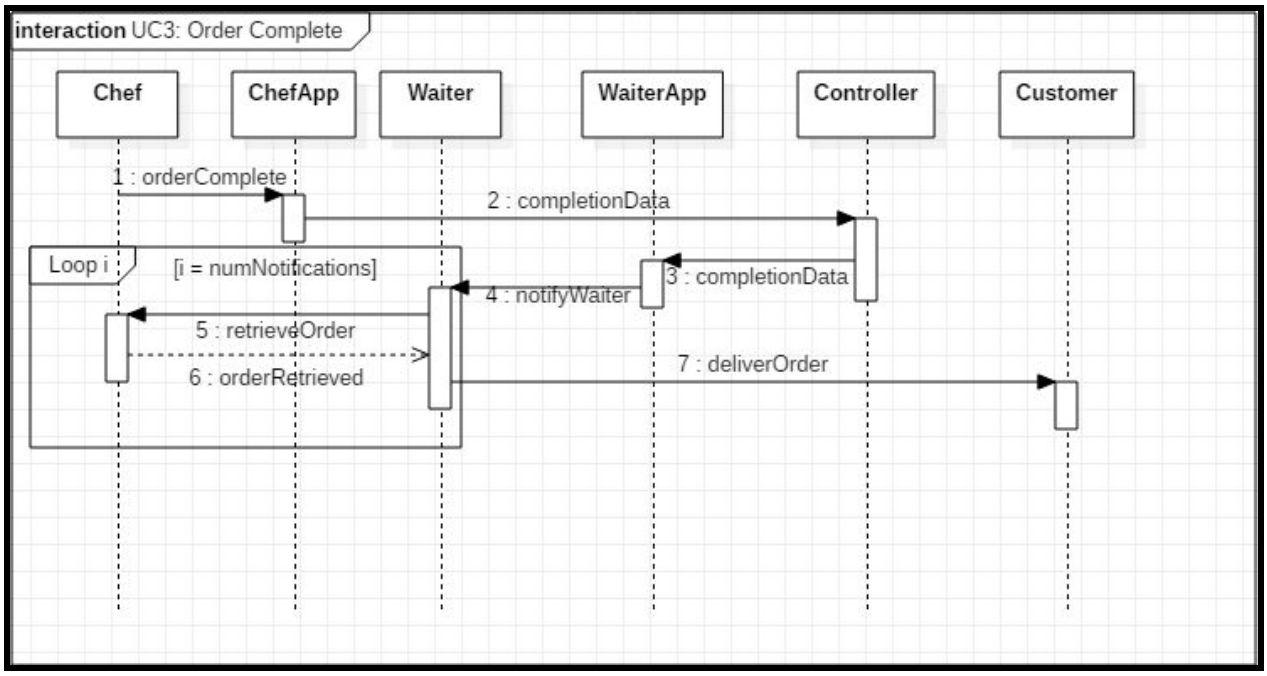
Design Pattern Used: This used the Decorator Pattern because all decorators know next subject and they all have some interfaces as real subjects. Also, each of them contributes to processing a specific case and forwards requests to the next subject so that a requests chain exists.

Communication Diagram



7.2: UC-3 - Order Complete

Sequence Diagram



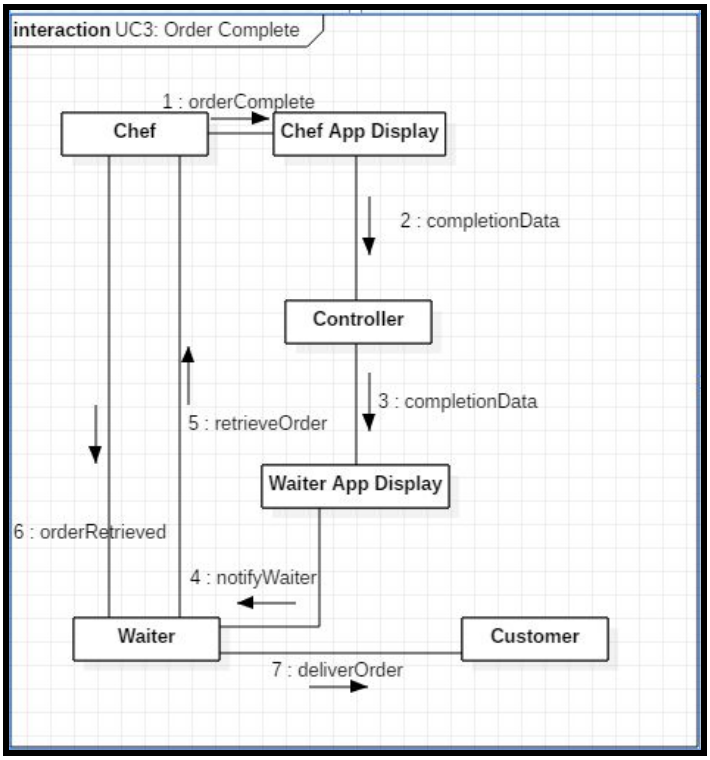
Design Principles Used:

In line with the Dependency Inversion Principle, the Chef and the Waiter Apps do not communicate directly with the database, and instead have their requests routed through the Controller class. This also follows Interface Segregation Principle(ISP), because we only use four interfaces here. High cohesion is also used as responsibilities of these given elements are highly focused and strongly related. Therefore, minimum number of dependencies is achieved.

Design Pattern Used:

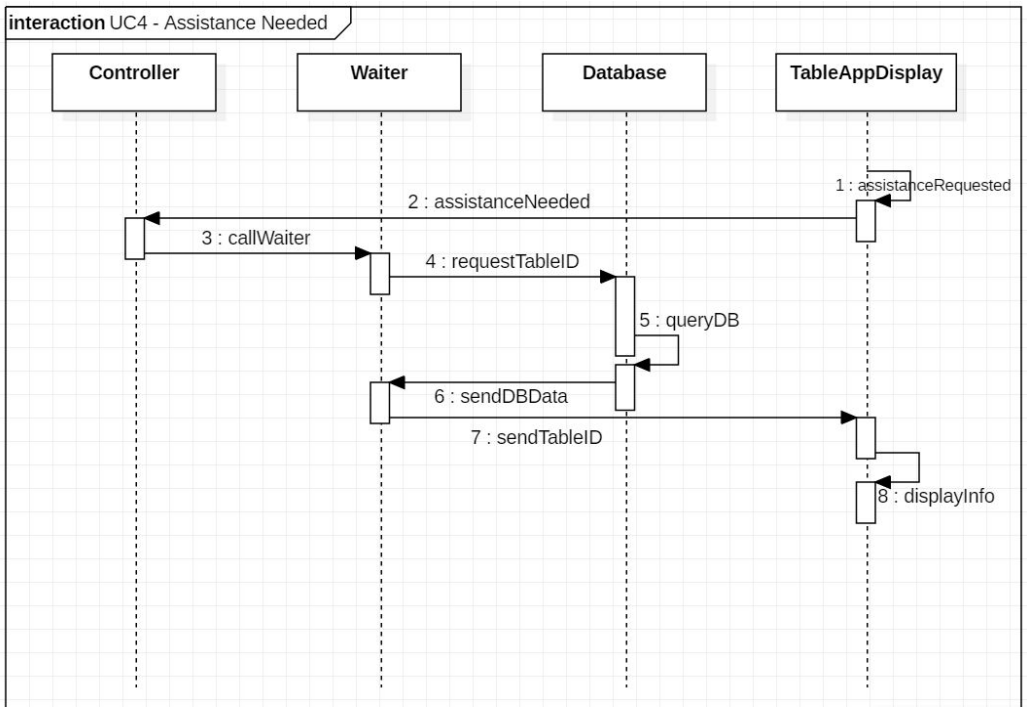
This also used the Decorator Pattern because all decorators know next subject and they all have some interfaces as real subjects. Also, each of them contributes to processing a specific case and forwards requests to the next subject so that a requests chain exists.

Communication Diagram



7.3: UC-4 - Assistance Needed

Sequence Diagram

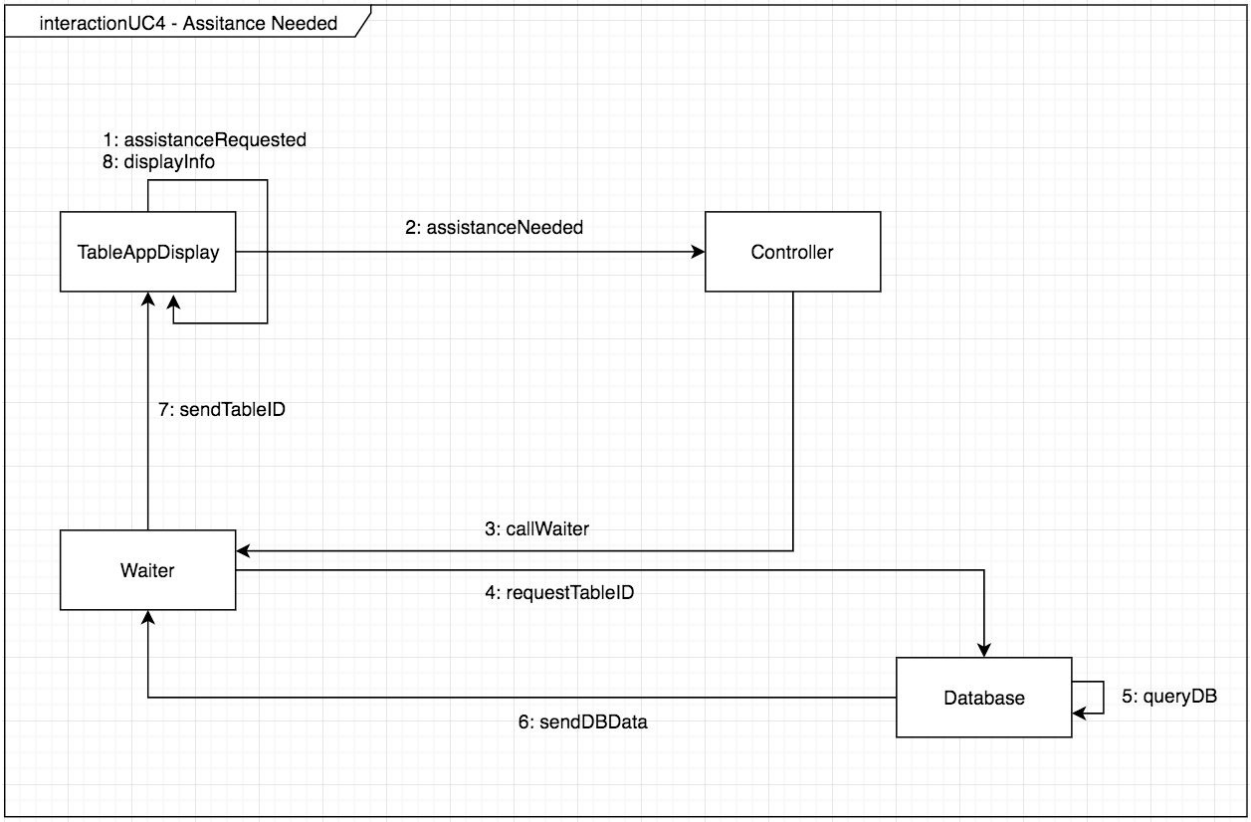


Design Principles Used: This follows the Dependency Inversion Principle (DIP) because the app has to go through the controller class in order get the correct information from the database. This also follows the Liskov Substitution Principle (LSP) because the user interfaces will only contain information that customers and waiting staff needs (buttons and table id queues for assistance). The controller pattern just uses the call waiter to call the waiter for assistance rather than having different classes for the types of assistance.

Design Pattern Used:

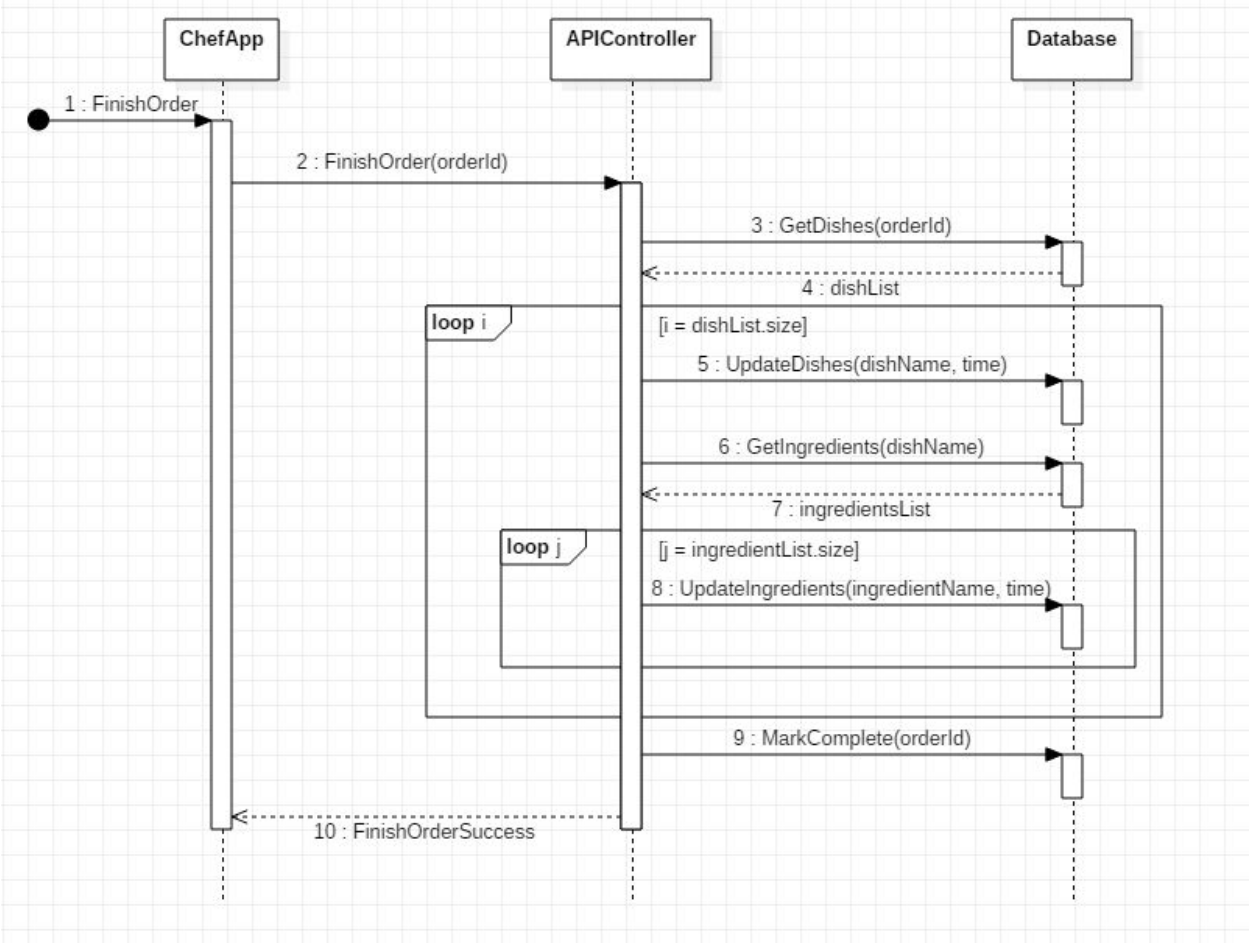
This uses Proxy pattern because it knows the real subject of requests and has some interface as real subject. It is responsible for intercepting and preprocessing requests, ensuring safe, efficient and correct access to the real subject.

Communication Diagram



7.4: UC-6 - Finish Order

Sequence Diagram



Design Principles Used:

Following the Dependency Inversion Principle, the Chef App sends its request to the API controller, which in turn handles the individual requests to the Database.

Following the Principle of Least Knowledge, or the Law of Demeter, the APIController has limited knowledge and must fetch all information about the orders, dishes, and ingredients from the database.

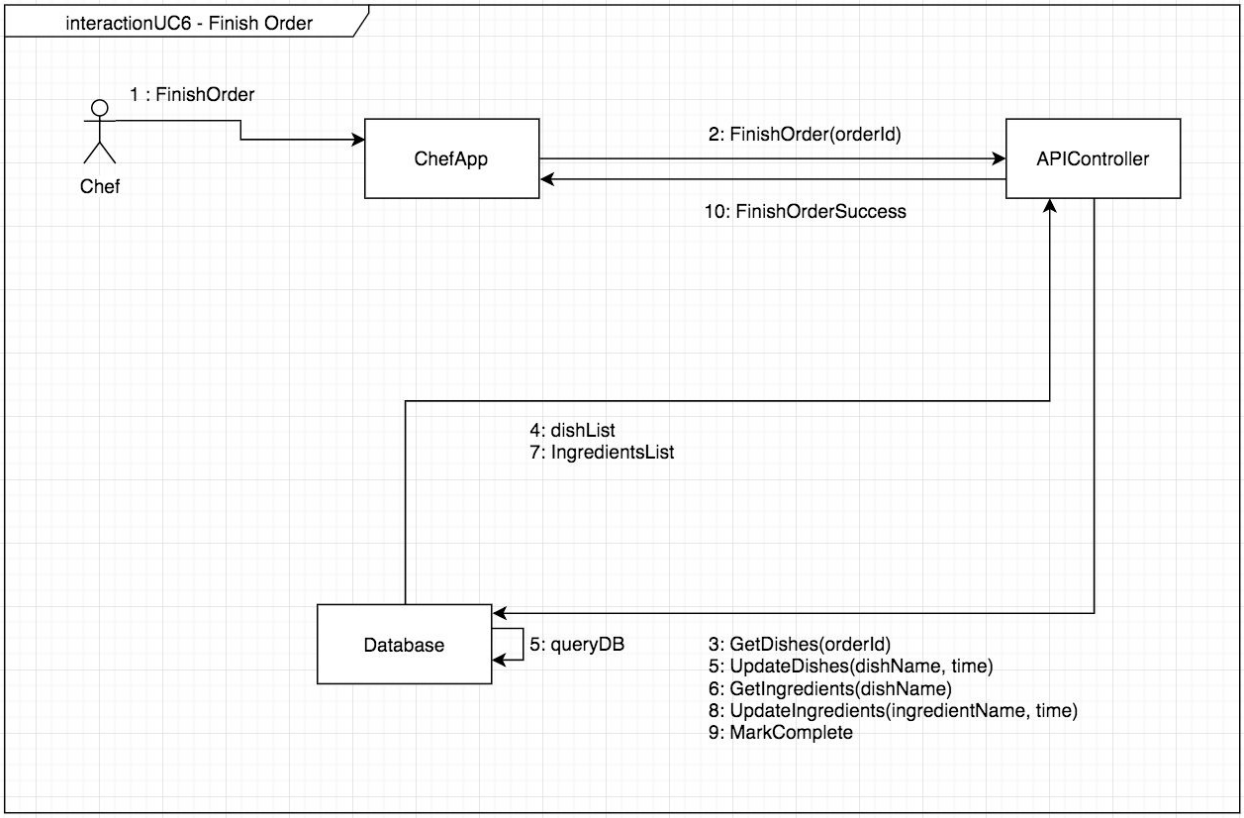
Open/Close principle(OCP) is used as orders can be extended as more orders are added but the orders marked completed can not be modified to not completed.

Creator design pattern is used as markCompleted() aggregates instances of getDishes() and orderID().

Design Pattern Used:

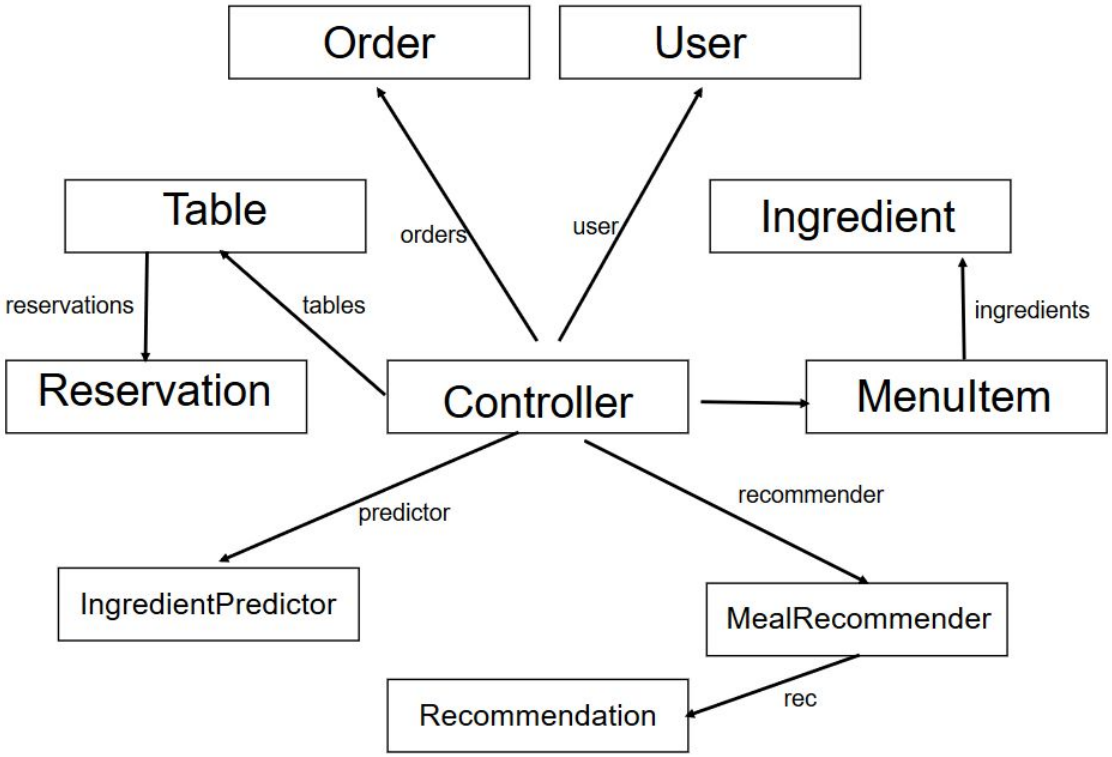
This uses Command Pattern because all subjects know receiver of action request. All command subjects execute an action.

Communication Diagram

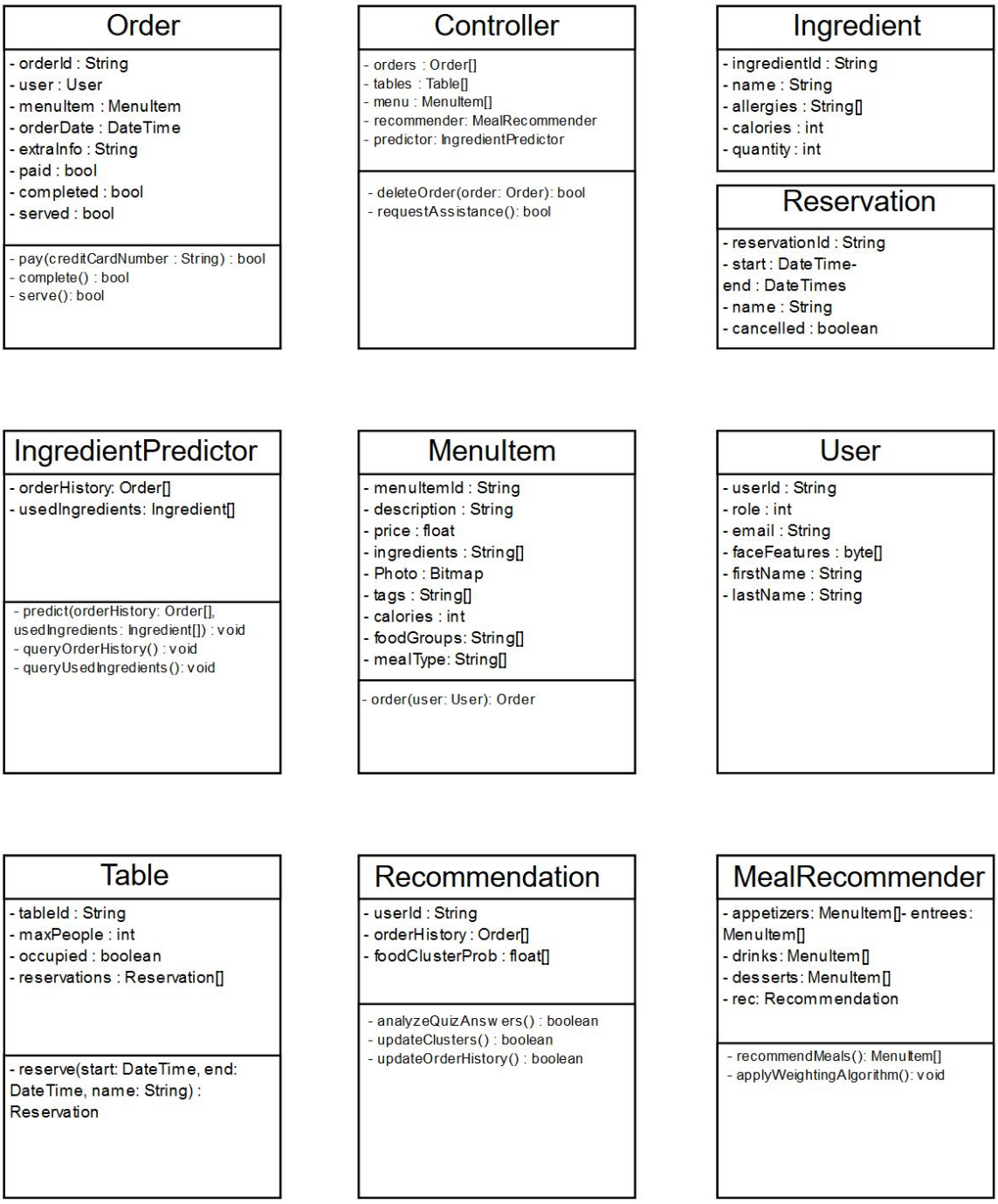


Section 8: Class Diagram & Interface Specification

8.1: Class Diagram



8.2: Data Types and Operation Signatures



Definitions:
Order: used to hold order-related information: who make the order, what food is inside the order, detail of the order and is order paid. This class should be in the same package as User and menuItem so it can create their objects.

Controller: class responsible for managing the app interactions. Contains a list of all current orders, MenuItems and tables, as well as IngredientPredictor and MealRecommender to run ingredient prediction analysis and generate meal recommendations. Additionally it handles HTTP requests to communicate with the server. The exact Controllers implementation will differ slightly between all the apps, but the Controller class shown on the diagram includes all the combined features from the specific implementations.

User: used to store user-related information: username, userId, role to determine what permission the current user has (customer, server or cheif). In addition it will also store face capture information for the user.

Ingredient: used to store ingredient-related information: its name, how many calories it has, whether it can cause allergies, and how much of it the restaurant has remaining.

MealRecommender: class to generate suggestions based on user's past ordering habits.

Recommendation: a class to store and manipulate information necessary for MealRecommender to generate suggestions.

Table: stores table related information, such as the maximum amount of people it can host and whether or not a table is currently occupied. Also stores the reservations for this table.

Reservation: class to store reservation information. When the reservation is made, who made it and whether or not it's still active or cancelled.

MenuItem: represents an item from the restaurant's menu.

IngredientPredictor: class to predict ingredient usage based on past orders.

8.3: Traceability Matrix

Concept/Class	*Order*	*Controller*	*Ingredient*	*MenuItem*	*User*	*Table*	*Rec**	*Res***	*Pred****
Manager	X	X		X					
UserAccount	X				X		X		
Menu		X		X					
MealRecommend							X		
Database	X	X	X	X	X	X	X	X	X
Ingredients			X	X					X
WaiterNeeded	X	X							
Payment	X								
OrderQueue	X	X							

*Recommendation
**Reservation
***IngredientPredictor

Manager: this domain concept was meant to propagate each customer's order to the waiters and chefs along with keeping track of each order's status. Order class contains the order information, Controller has the list of all orders, and MenuItem is responsible for creating an order.

UserAccount: this domain concept intended to keep track of user information and a list of a user's previous orders. Therefore an Order class had to be defined to keep track of each order, a User class had to be created to keep track of basic user information, and a Recommendation Class had to be created because it depends on users' previous orders to update their food cluster probabilities.

Menu: this concept intended to display the most up-to-date menu and allow the restaurant owner to update menu as needed. As a result, the MenuItem class was created to store information about each item in the menu. Moreover, the Controller class contains an array of MenuItem objects. This array can be edited to reflect the establishment's most current menu.

MealRecommend: this concept uses each customers' order history to recommend items along with administering quizzes to determine what foods to recommend. Therefore, the Recommendation class had to be created, which would keep track of each customer's

food cluster probabilities (determined by quizzes) and a users' previous orders (database will be queried to retrieve orders placed in the last two weeks).

Database: this domain concept provides a simple API to access the database and manages the database. Since the purpose of this concept is so broad, and the attributes of all classes need to be stored somewhere, the database concept contributed to the creation of all necessary classes.

Ingredients: this concept keeps track of what ingredients are available and updates that list of ingredients as orders are placed. This required the creation of the Ingredient class and the MenuItem class, which keeps track of ingredients required per food item.

WaiterNeeded: this concept is used to communicate the request for assistance between the customer/table apps and the waiter app. The Controller class will handle the request and facilitate communication between the multiple applications. The communication can also be triggered by the Order class, when an order is marked as completed using the complete() method, indicating that it has to be served by the waiter.

Payment: this concept allows the customers to pay for their orders. Because payment of orders is tightly coupled with the contents of the order itself, this concept is included in the Order class.

OrderQueue: Since this domain concept kept track of received orders, it necessitated the creation of the Controller class, which stores an array of Order class items, and the Order class, which stores relevant information about each order.

**Although no domain concept with a *very specific function* resulted in the creation of the Reservation class, it had to be created because the Controller class had to keep track of tables. The Table class kept track of reservations made on each table. Therefore, a Reservation class had to be made to keep track of all reservations made by customers.

8.4: Design Patterns

The design pattern we used for "Order Food" and "Order Complete" use case is the Decorator Pattern because all decorators know next subject and they all have some interfaces as real subjects. Also, each of them contributes to processing a specific case and forwards requests to the next subject so that a requests chain exists. For the "Assistant Needed" use case, we use Proxy pattern because it knows the real subject of requests and has some interface as real subject. It is responsible for intercepting and

preprocessing requests, ensuring safe, efficient and correct access to the real subject. We use Command Pattern for “Finish Order” use case, because all subjects know receiver of action request. All command subjects execute an action. In addition, We use Singleton design pattern for Network Manager in for apps and Publisher and Subscriber design pattern for Chef and Waiter app.

8.5: Object Constraint Language (OCL) Contracts

Order Class Contracts

Context: pay(creditCardNumber: String): bool
Pre: this.user.length != null (user is logged in) && this.menuItem.length != 0
Post: this.orderDate = new DateTime() this.paid = false this.completed = false this.served = false return true

Context: complete(): bool
Pre: order != null
Post: order.complete = true return true

Context: serve(): bool
Pre: order != null
Post: order.served = true return true

Controller Class Contracts

Context: deleteOrder(order: Order): bool
Pre: orders.length > 0 && order != null
Post: exists = orders.find(order) if (exists) orders.remove(order) return true

Context: requestAssistance(): bool
Pre: none
Post: sendRequest(tableID: Table.tableID) return true

IngredientPredictor Class Contracts

Context: predict(orderHistory: Order[], usedIngredients: Ingredient[]): dict
Pre: orderHistory.length > 0 || usedIngredients.length > 0
Post: return results.get_prediction(start = Date.now(), end = Date.now() + 7, dynamic=False)

Context: queryOrderHistory(start: Date, end: Date): Order[]
Pre: orders.length > 0
Post: return orders.loc(start, end)

Context: queryUsedIngredients(start: Date, end: Date): Ingredient[]
Pre: ingredients.length > 0
Post: return ingredients.loc(start, end)

MenuItem Class Contracts

Context: order(user: User): Order
Pre: user != null && user.length > 0

Post: tempOrder = new Order()
tempOrder.user = user
tempOrder.menuItem.append(this) //add selected MenuItem
return tempOrder

Table Class Contracts

Context: reserve(start: DateTime, end: DateTime, name: String) : Reservation
Pre: none
Post: tempReservation = new Reservation() tempReservation.start = start tempReservation.end = end tempReservation.name = name tempReservation.cancelled = false tempReservation.reservationId = randomInt().toString() return tempReservation

Recommendation Class Contracts

Context: analyzeQuizAnswers(): boolean
Pre: user is logged in

Post: tempArray = ones.(foodClusterProb.length)
for cluster in foodClusterProb
tempArray[cluster] += ((cluster.selected == true) : 1 ? 0) // if food cluster selected in quiz results, then increment value in array
this.foodClusterProb = tempArray
return true

Context: updateClusters(): boolean
Pre: user is logged in

Post: for order in this.orderHistory :
tempDishes = order.menuItem
foodClusters = []
for dish in tempDishes :
tempFoodGroups = dish.foodGroups
for fg in tempFoodGroups :
foodClusters.append(fg)
popularFoodGroups = topThreeClusters(foodClusters) //using a helper function to return values of food clusters or most ordered items
for cluster in foodClusterProb
if (foodClusterProb.name == cluster) foodClusterProb += 1
return true

Context: updateOrderHistory(): boolean

Pre: user is logged in

Post: this.orderHistory = []
tempOrders = queryOrdersDB(user: User) //returns most recent orders
this.orderHistory = tempOrders
return true

MealRecommender Class Contracts

Context: recommendMeals(): MenuItems[]
Pre: user is logged in

Post: popularFoodGroups = topThreeClusters(this.foodClusterProb)
dishes = []
for fg in popularFoodGroups :
tempDishes = queryDishesDB(fb)
dishes.append(tempDishes)
dishes = new Set(dishes) //remove duplicates
return dishes

Context: applyWeightingAlgorithm(): void
Pre: user is logged in
Post: tempRec = new Recommendation() tempRec.updateClusters() //takes care of updating clusters based on previous orders

Section 9: System Architecture & System Design

9.1: Architectural Styles

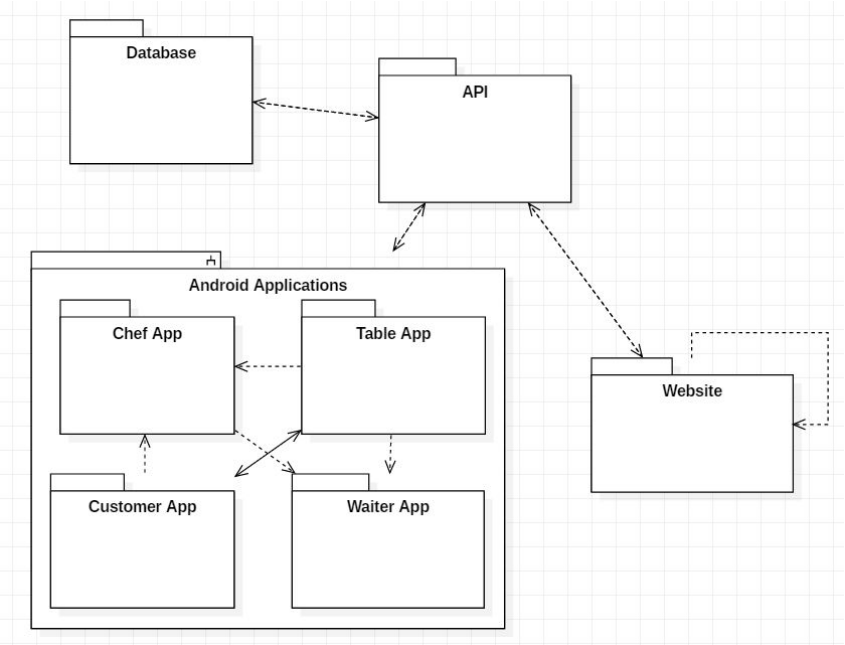
The server will expose a REST API for all communications to and from the apps, as well as the website via AJAX requests.

The apps will use the client-server model - they are essentially a UI wrapper that sends and receives information and do very little computation on their own. The server will do the majority of the work, from updating the database and sending notifications to chefs, waiters and customers to generating recommendations based on client's past dining habits.

The website will use the MVC architecture, having a template "view" that will be filled out with a "model" retrieved from the database by the "controller". In addition it will

perform AJAX requests to the REST API to update information without having to reload the entire page.

9.2: Identifying Subsystems



Our projects consists of multiple subsystems, namely:

- Four apps - Customer App, Table App, Chef App and Waiter App
- A website, both for customers and for the restaurant owner
- API server and the Database

The purpose of the apps is to simplify customer's dining experience, as well as to optimize the staff's workflow. The website serves two purposes - one is to allow customers to to browse the menu and make orders, similar to the customer app. Second purpose is to allow the restaurant owners to manage their restaurants and perform administrative tasks. The API Server acts as a central hub for the project. It's the entity the manages the database end exposes an interface for other subsystems to interact with the database.

Based on the UML package diagram, the website is able to edit itself because of the admin console. Likewise, the API can edit the database, the website, and Android applications. The Chef App can send notifications to the Waiter App. The Customer App sends orders to the Chef App and is closely linked with the Table App due to customers being able to sign on to and using both. Similarly, customers can send orders from the Table App, which are sent to the Chef App.

9.3: Mapping Subsystems to Hardware

The mapping of subsystems to hardware is fairly straightforward. The website/API server as well as the database will run on a remote x86 machine, more specifically an

EC2 instance hosted by Amazon Web Services. The apps are designed to run on any mobile device running Android 5.0 operating system or higher. When designing the customer app we will assume that it will primarily run on customer's private mobile phones with an average screen size of about 5" - 5.5". All the other apps will be designed assuming the will be run on larger tablet devices with the screen size of around 10", that will be provided by the restaurant owner.

9.4: Persistent Data Storage

We will be using MongoDB as out persistent storage. We decided to go with MongoDB versus a more widespread SQL databases due to two main reasons. First is that MongoDB stores information in JSON format, which integrates easily with NodeJS, a JavaScript framework. In some cases we can retrieve data from the database, and send it in the response, without even modifying it in any way. Second is that MongoDB's NoSQL nature allows us to modify our tables on the go, adding new fields as we need them without having to redefine the entire Schema. This is something we believe will be useful during the prototype phase of or project, when we discover we need extra fields we haven't thought about before.

9.5: Network Protocol

Since we're using a RESTful API, all our communications with the server will be done via the HTTP protocol. The apps will send HTTP GET and POST requests, and then parse the received response and display it to the user.

9.6: Global Control Flow

As a service based around node.js and REST API, our service innately has an event driven control flow. Upon startup of the server, Node maintains an event loop and listens for various events. Once an event, such as an API call to a specific endpoint, has fired, Node will trigger the corresponding event listener callback function to execute the requisite code.

Apart from timers that are part of the node.js control flow, our service does not include any additional timers.

Since the framework we are using for our web/API server, node.js, is single threaded, there should be minimal concurrency issues. However, node does make use of asynchronous callback functions that execute within the event loop, so there may be some edge cases within shared objects, such as the order queue, that we will need to synchronize using standard thread synchronization procedure. Nonetheless, as a single threaded application, much of the risk resulting from non-synchronized threads is mitigated significantly.

9.7: Hardware Requirements

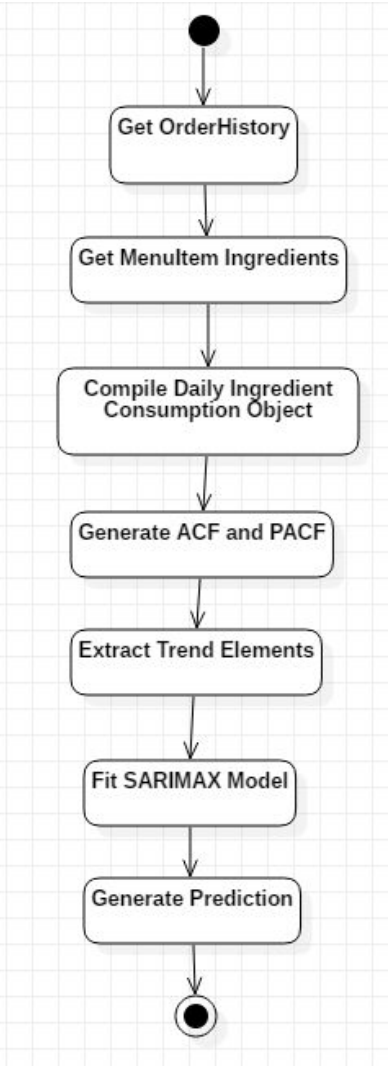
The apps will run on any mobile device running Android 5.0 or higher.
The website should work on any modern web browser (however we will only test it (and ensure it runs properly) on Microsoft Edge, Mozilla Firefox and Google Chrome).
The server should run on any hardware capable of running NodeJS, however we assume it will be run on an x86 machine running Linux operating system.

Section 10: Algorithms & Data Structures

10.1: Algorithms

10.1.1: Ingredient Prediction System

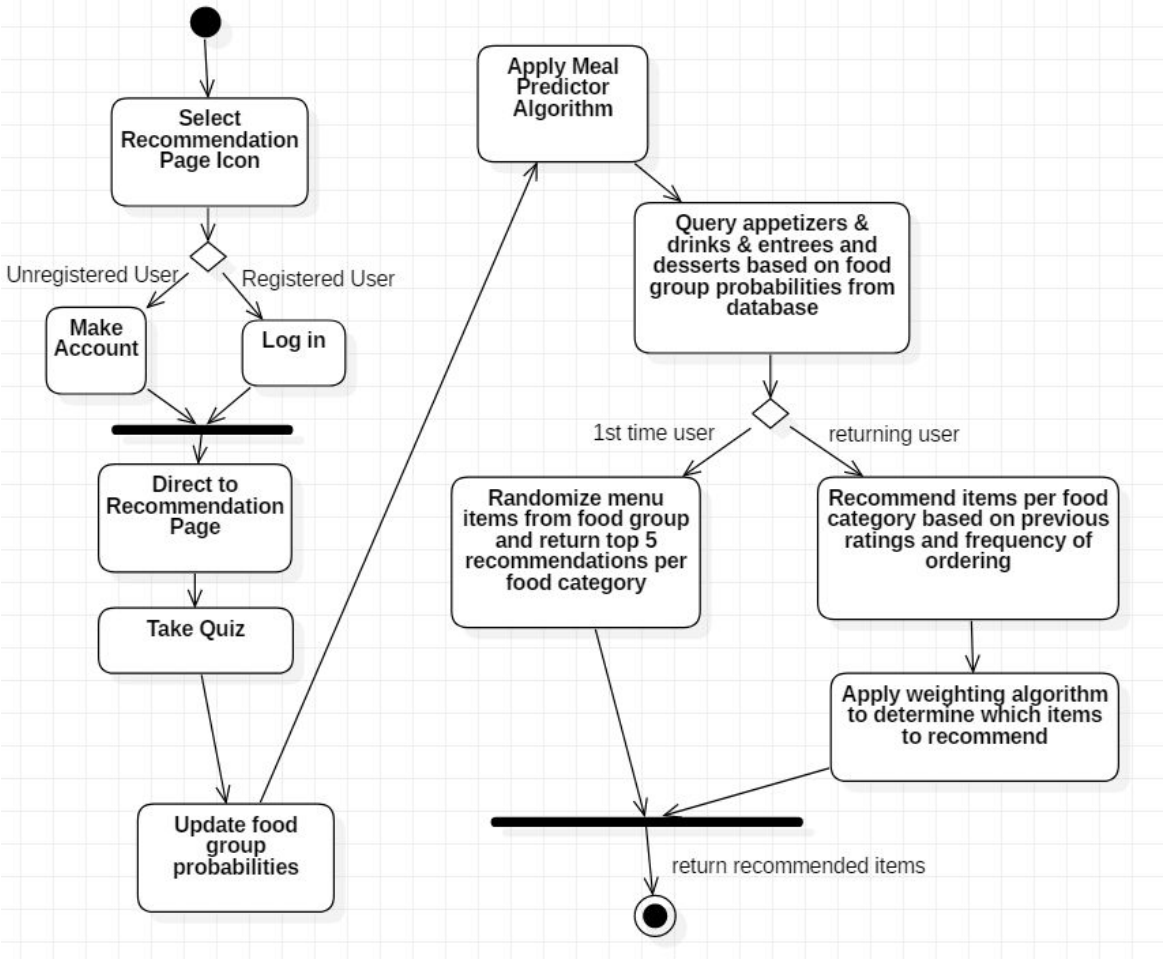
To perform ingredient prediction, the first step is to pull all of the relevant information about dish order history from the previous several weeks, and the component ingredients of the dishes from the database. This information will come in the form of a JSON object containing a list of all orders over the previous few weeks. Once this is completed, the data must be processed by adding together ingredient consumptions that occur within the same day and adding together ingredient consumptions from different dishes that share the same component ingredients. This processed data should be in a 2D array structure, in which the outer array corresponds to the day, and the inner array corresponds to the ingredients used. Using the autocorrelation function (ACF) and the partial autocorrelation function (PACF) functions, overall trend elements and seasonal trend elements over the course of the day should be extracted from the data. Once this is complete, the Seasonal Autoregressive Integrated Moving-Average with Exogenous Regressors (SARIMAX) model can be used to fit the data and forecast the ingredient consumption over the next few days.


Get OrderHistory
Get MenuItem Ingredients
Compile Daily Ingredient Consumption Object
Generate ACF and PACF
Extract Trend Elements
Fit SARIMAX Model
Generate Prediction

10.1.2: Recommendation and Rating System

Method 1: Content Based Filtering

Case 1: Registered user wants to use quiz answers to generate meal recommendations



How Quiz Answers are Analyzed:

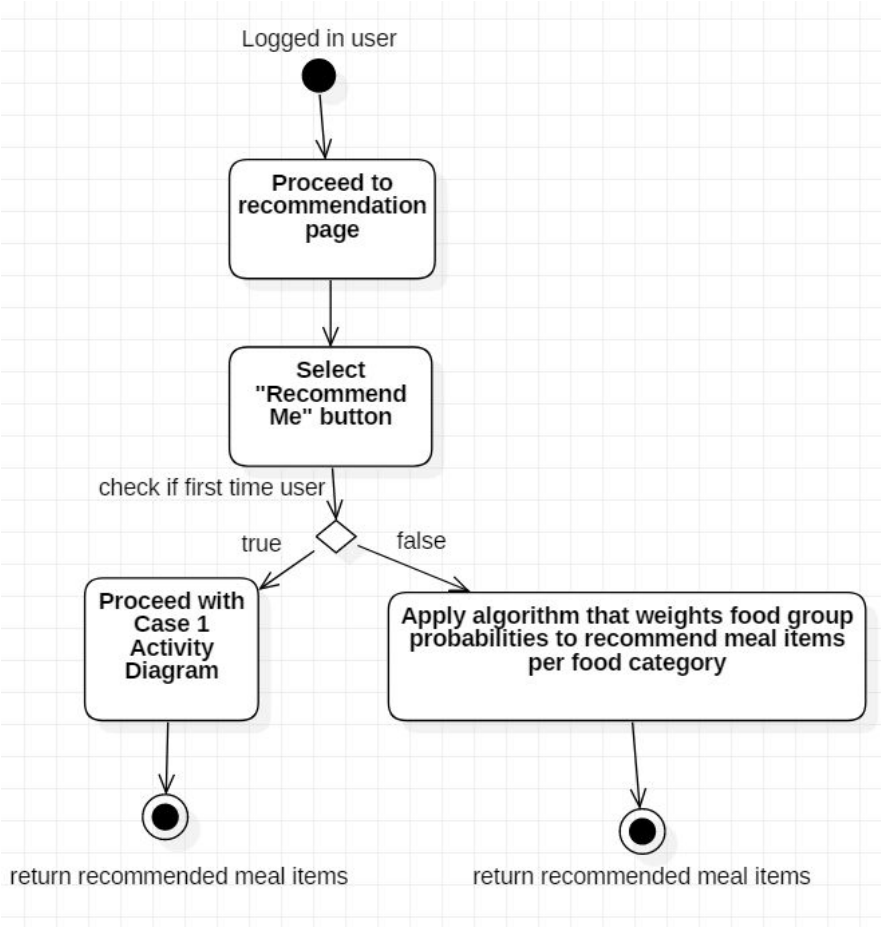
Menu items will be divided into a few supergroups (dairy, meats, vegetarian, drinks, carbohydrates etc.), which will be further subdivided into groups such as red meats, seafood, pasta, rice. The food group probabilities array will be initialized to have the same probability for each subgroup (each subgroup will have the same amount of points). After storing a customer's quiz answers, probabilities will be adjusted as needed (points will be added/subtracted) and will be normalized. Once all the probabilities are normalized, the weighting algorithm will be used if the customer is a returning customer. Otherwise, menu items for supergroups with the highest probabilities will be randomized and 5 recommendations per supergroup/meal type will be outputted.

user_food_preferences = <P(dairy), P(meats), P(carbs), … , P(vegetarian)>

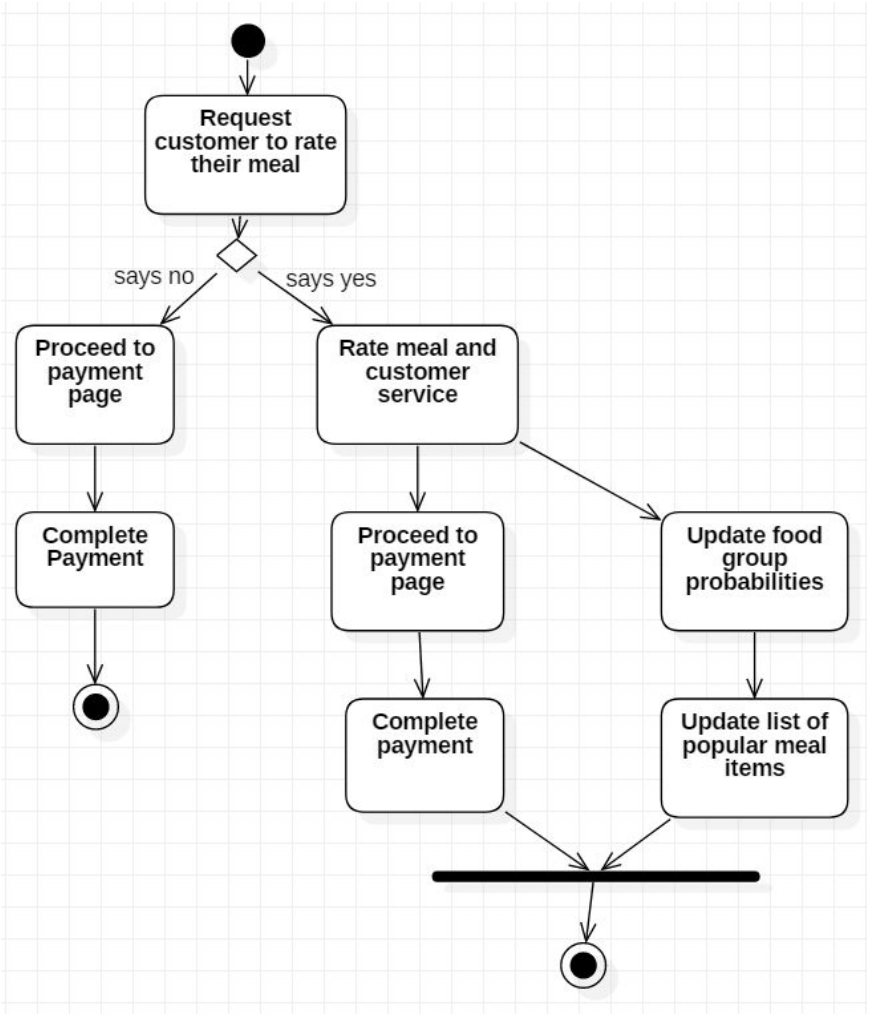
Description of Weighting Algorithm:

For users who have eaten at the restaurant more than once, information about how frequently order specific meal items and their ratings for each item will be stored in the database. This information will be used to give each meal time a point value. A user's "points" will be summed and normalized to generate probabilities. Food groups associated with most frequently eaten meals will have with higher point values, so those food groups will be given priority when used to recommend meal items. This can help recommend items similar but still different from previously eaten meal items.

Case 2: Registered user does not want to use quiz to generate meal predictions

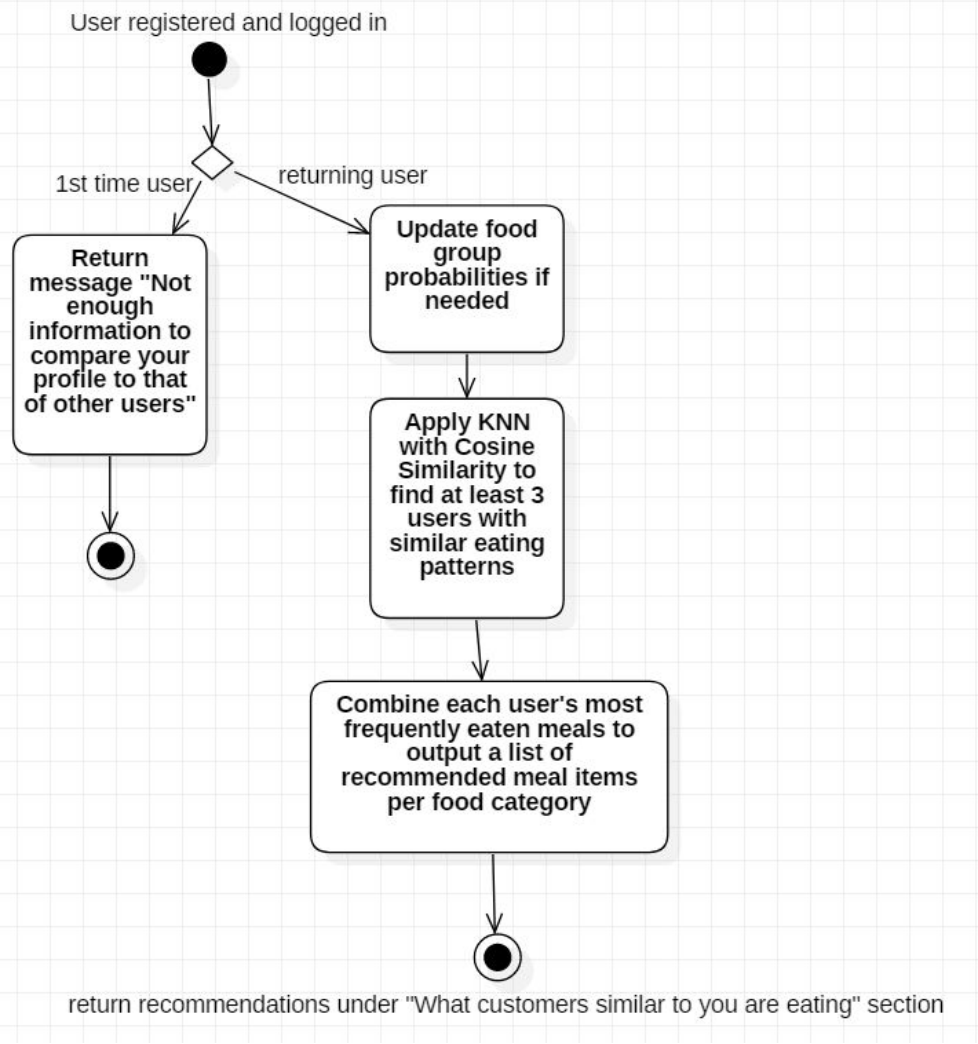


Case 3: Update user's meal recommendation profile using answers from rating system



Method 2: Collaborative Filtering

Case 1: Registered user can view meal recommendations based on eating patterns of similar users



Description of KNN/Cosine Similarity Algorithm

Once a user's food group probabilities array is updated, a simple search algorithm is used to find three closest food group probabilities of other users that are most similar to the user that is using the recommendation system. Once the userID's of customers whose eating patterns resemble the current user's the most are outputted, the application will combine their most frequently eaten meals to output a list of recommended meal items per food category. These meal recommendations will be displayed under the "What customers similar to you are eating" section.

Due to time constraints, this part of the recommendation system was not implemented by Demo 2 because fake order data and food group probabilities would have to generated for multiple users for the application to query the database, use cosine

similarity (compute-cosine-similarity npm package) and generate a list of recommendations based on other users' most eaten items.

Cosine similarity is described as follows:
A = user1_food_preferences = <P_1(dairy), P_1(meats), P_1(carbs), … , P_1(vegetarian)>
= <A_1, A_2, A_3, … , A_i>
B = user2_food_preferences = <P_2(dairy), P_2(meats), P_2(carbs), … , P_2(vegetarian)>
= <B_1, B_2, B_3, … , B_i>

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

10.1.3: Facial Recognition Algorithm

The goal of facial recognition system is to identify users using functions exist in our mobile devices, which means the algorithm need to support Android system. We first try to find some API inside Android to achieve it, but soon we find out the original API provided by android like FaceDetector class is not enough for our use. The similar problem occurs when we try to use some open sources SDKs like OpenCV: the functions are hard to modify to reach a satisfied confidence. Finally we choose a third party SDK provided by ArcSoft company which can satisfy our need well.

The SDK we used has its own engine to detect and identify the face. It stores the facial features in a List and store them in a txt file. The list includes the face's name and its features in bytes. To register a face, it first change the image to a Bitmap, then use AFD_FSDK_StillImageFaceDetection, which is a built-in function of the engine to detect the face. AFD_FSDKFace class holds the output of the detection. It includes the features as integers. When the matching rate is higher than a specific value, the user will be automatically logged in. To make our demo more fluently, we set the value to 0.6.

10.2: Data Structures

For the data storage on database, we are going to store all datas in JSON type. JSON is pretty readable and straightforward. Using JSON is beneficial when quickly creating domain objects in dynamic languages. JSON objects and code objects match so that it's is extremely easy to work with in some languages such as PHP and JavaScript. For example, when we store the ingredient information in the database, we can use following structure:

```
var greenPepper = {
```

```
        ingredientid: 1,
        name: “green pepper”,
        allergies: “None”,
        calories: 24,
        quantity:100
};
```

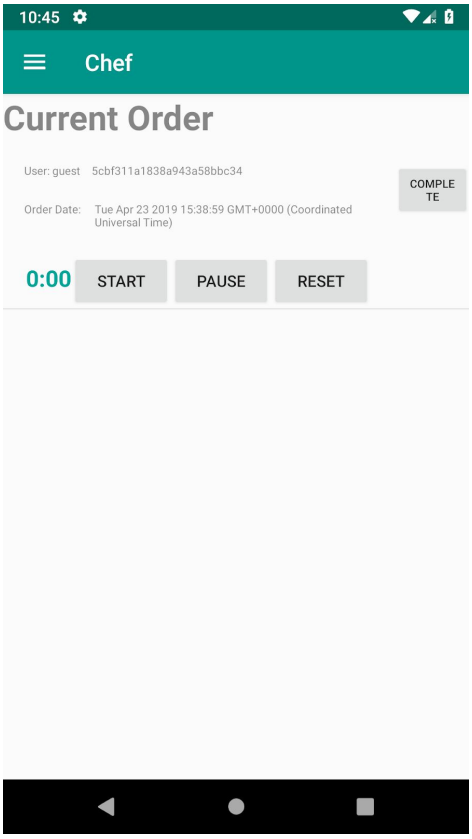
The above object may also be embedded in an ingredient object.

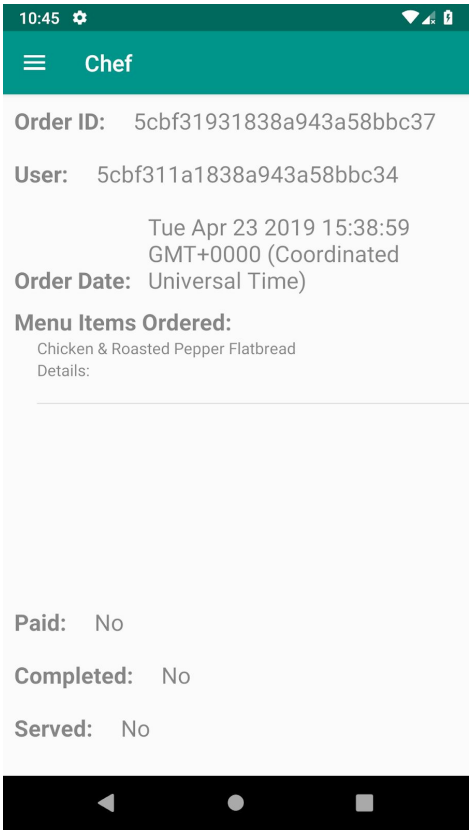
The Array List will be employed for storage of Ingredient, MenuItem, Table and User, including Manager, Chef and Waiters. ArrayList is an object oriented data structure, allowing developer dynamic adding and removing of elements. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically. For example, ArrayList allows manager keeps checking the usage of the ingredient, inserting and deleting elements in the arrayList. In addition, when doing the android development, an arrayList can be easily displayed by a listview which is more convenient for the application development. All ingredients, MenuItem, table and user information can be clearly displayed on the tablet screen.

Priority Queue will be mainly used for Order and Customer. Every customer comes in the restaurant will be assigned a table in First Come First Served order. Considering the size of tables may not in uniform and the amount of large table would be less than other normal table, when more than one big party come in, they may have to wait for more time and let some other small group get a table first even though those smaller group may be come in later. Priority Queue is the most suitable for this condition. When Chefs preparing dishes, it is time consuming for each time only doing one dish for one order. Chef can prepare more than one orders from the priority queue which have some procedures and materials in common to improve the working efficiency.

Section 11: User Interface Design & Implementation

11.1: Chef App

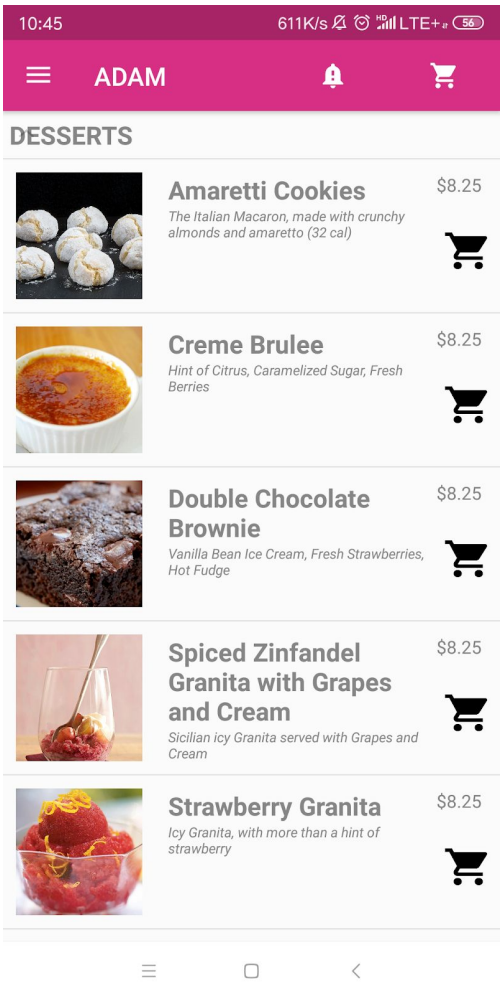




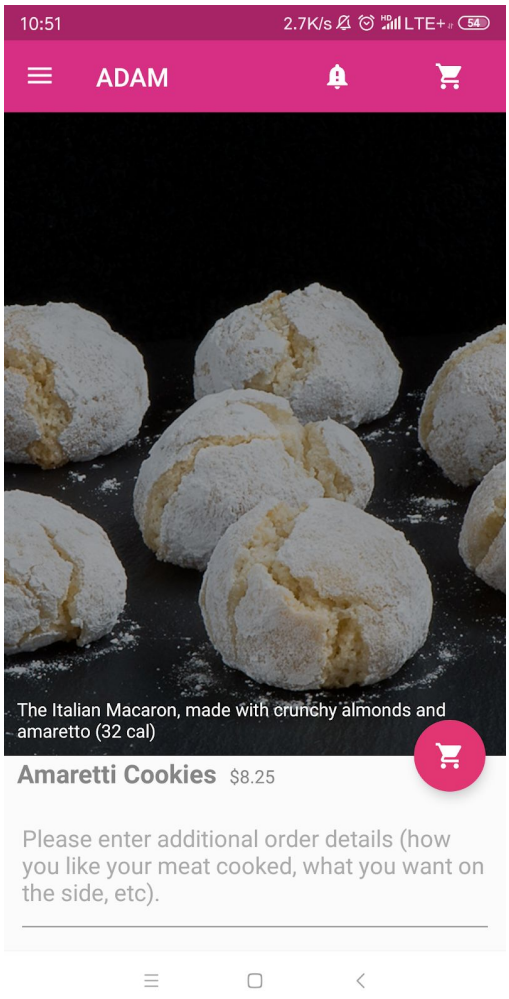
The chef app has two main screens - current orders list and orders detail, which can be viewed by pressing an order on the current orders list. Each order on the current orders list has a timer that can be started to help the chef keep track of order preparation time, as well as a button to mark the order completed. Order details views contains more information, such as the dishes ordered and order status.

11.2: Customer/Table Apps

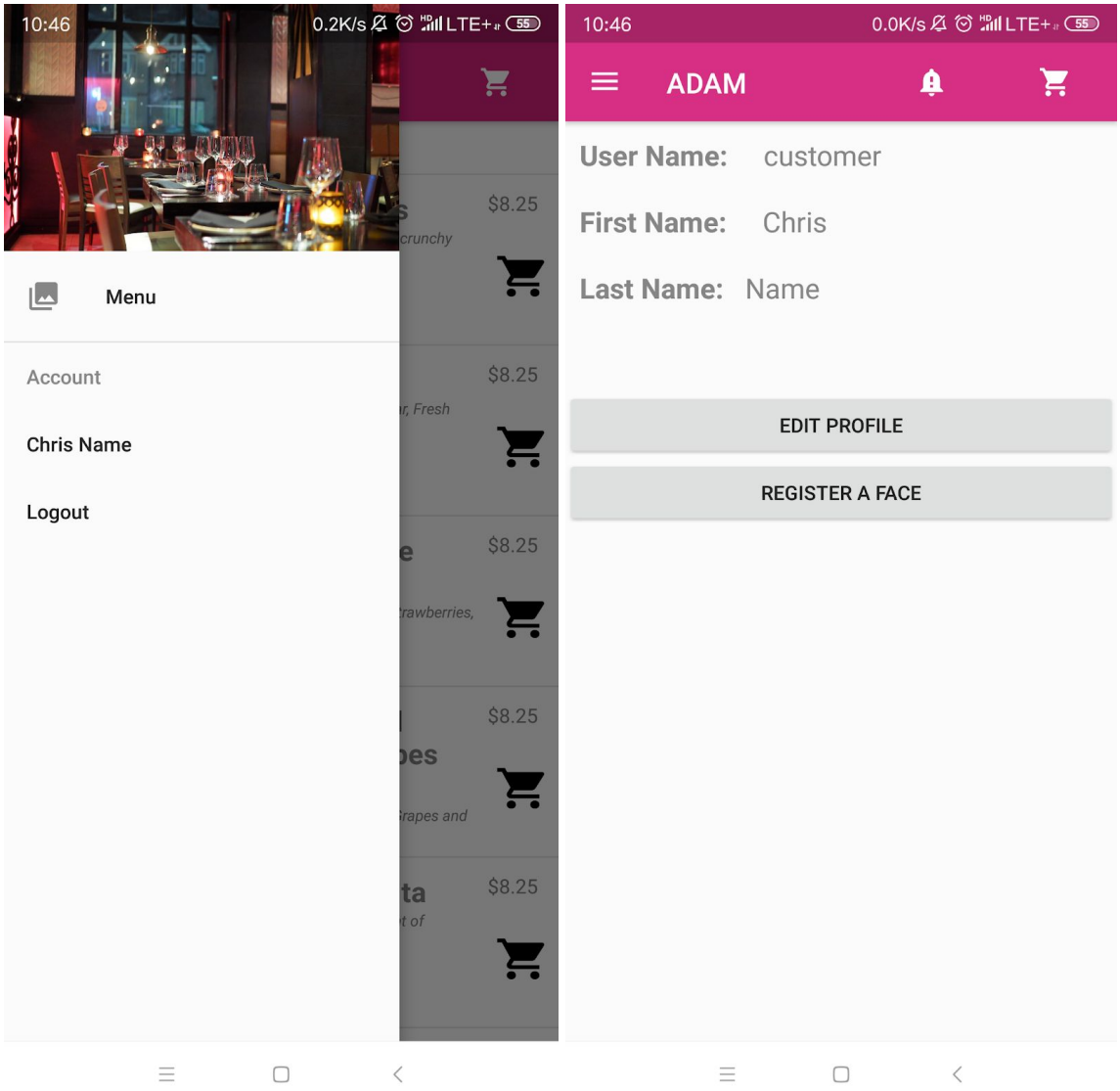
Customer Menu



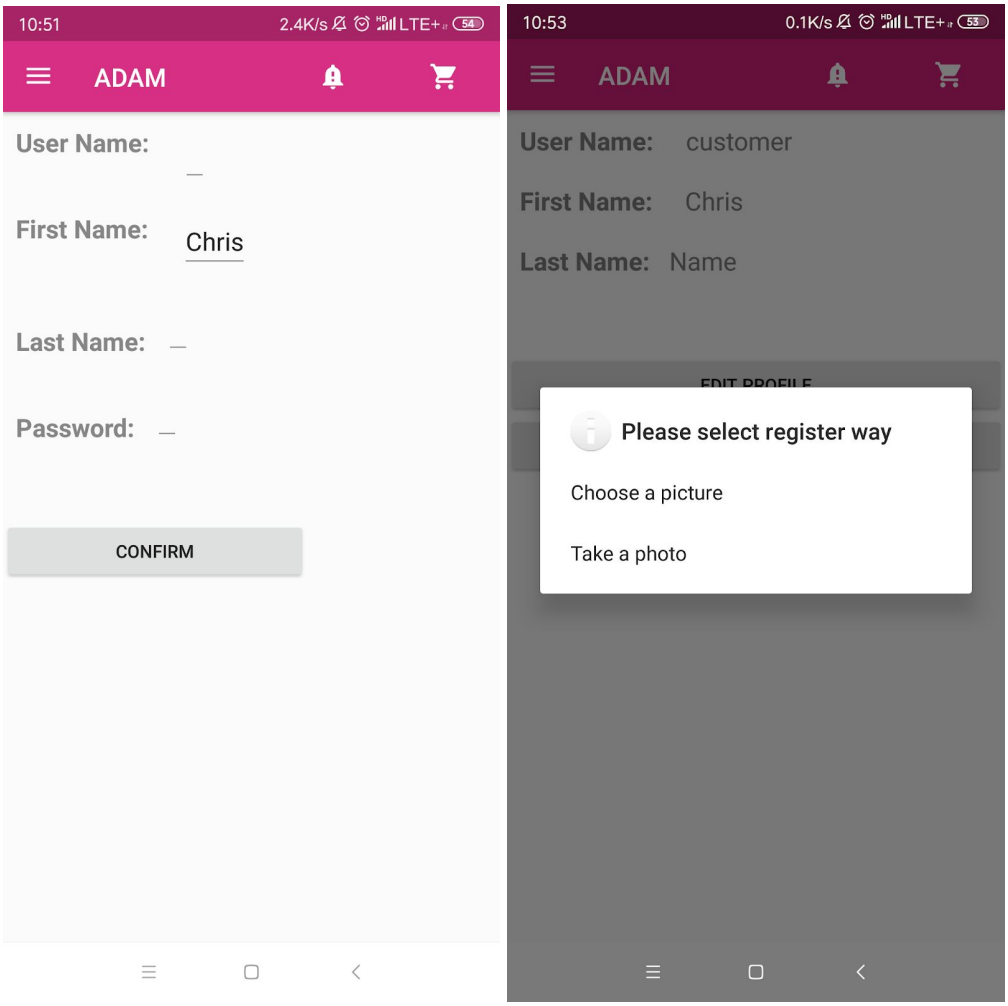
Customer Order Detail



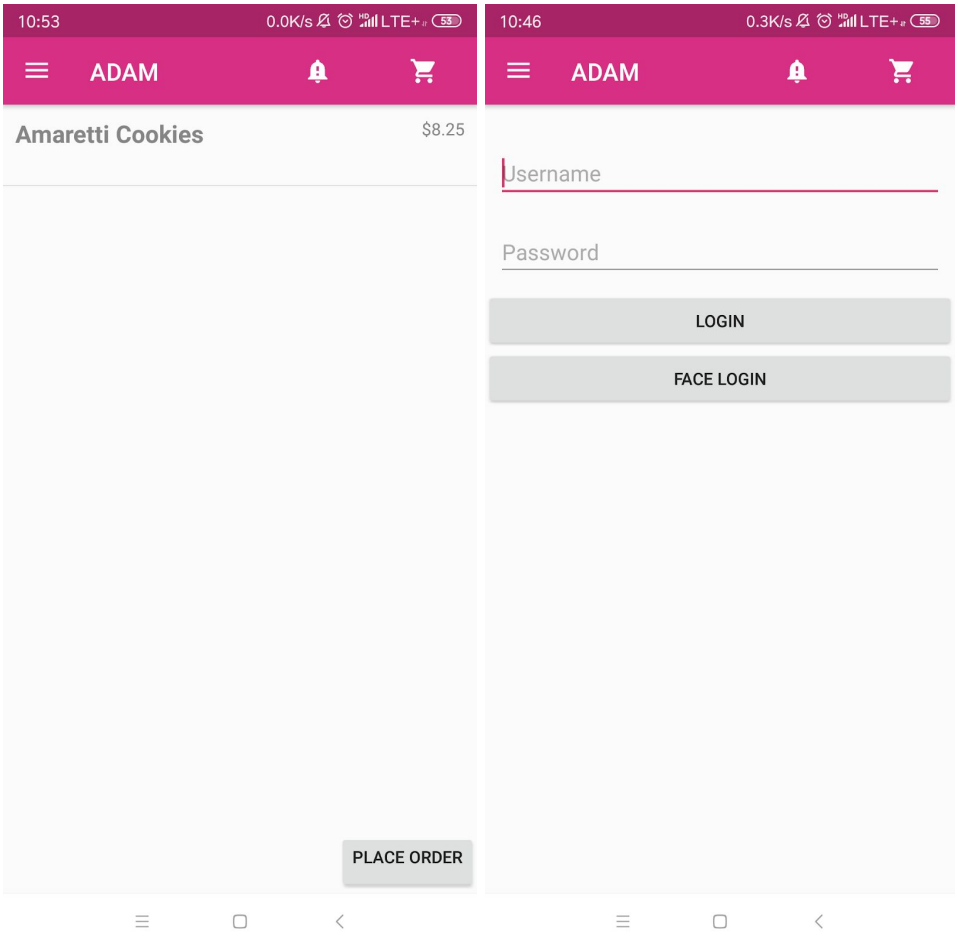
Main view of the customer app is the menu view. It shows the list of all dishes served by the restaurant. Each items has a dish name, description, photo and price, as well as a button to add it to card. Alternatively you can click on a dish to go to the details view, where the customer can see a longer description, as well as type in additional order details before adding it to cart.



The app has a slide out menu, where the user login, logout or edit their account details. To go to the edit detail view the user has to click on their name in the navigation drawer. The account overview has user details, as well as two buttons to edit the profile or register their face for face login.



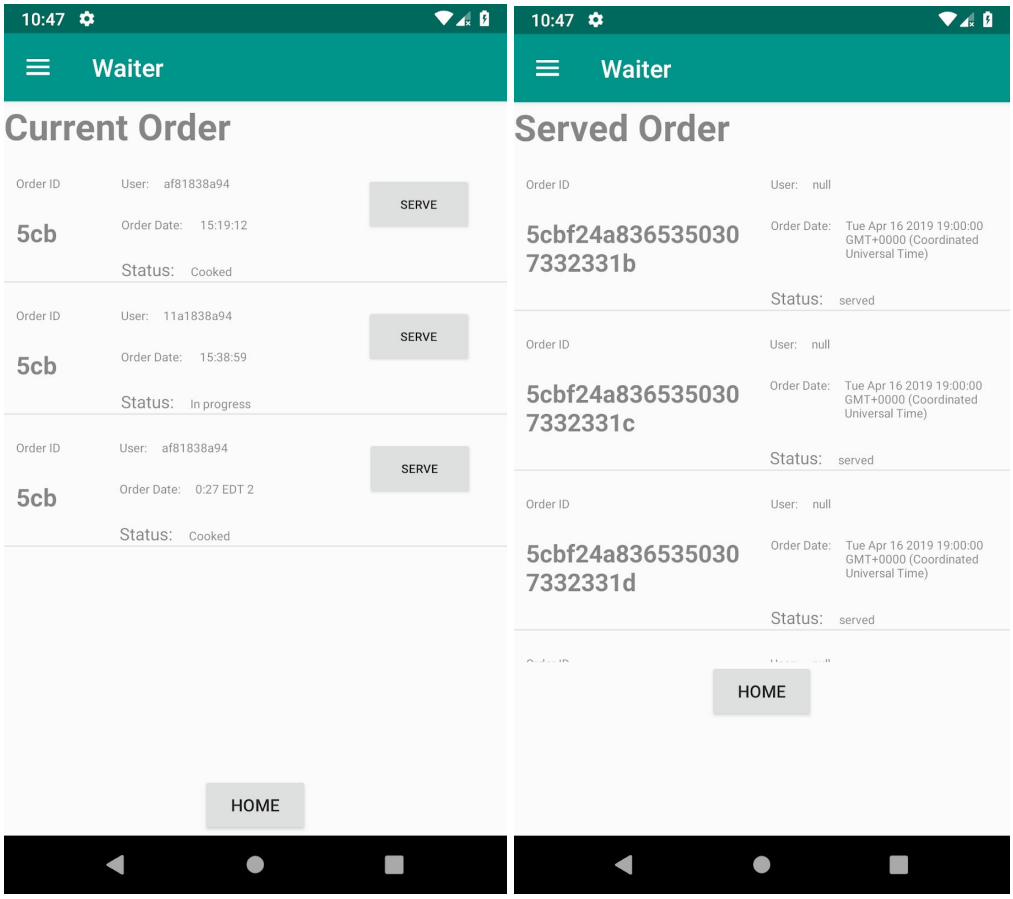
Pressing EDIT PROFILE on the profile info page will open a new view where the user can edit their profile information, such as username, First Name, Last Name and Password. Pressing Register a Face will prompt the user to either choose a picture already stored on their device or take a new picture of their face to be used for face login.



On any screen the user can press the cart button on the bar to access the cart view. It will have a list of all the items user has added to the cart and a button to place the order.

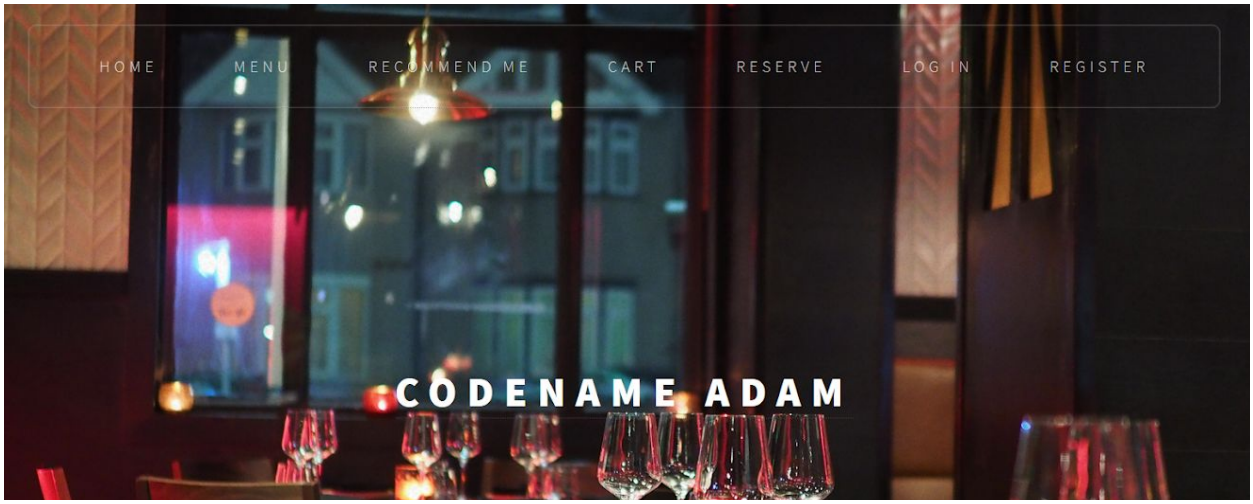
If the user is not logged in, they can access the login screen. From the user can enter their username and password and attempt to login by pressing the login button. Alternatively they can attempt to login by pressing FACE LOGIN. This will open the front camera feed and attempt to recognize any of the stored faces.

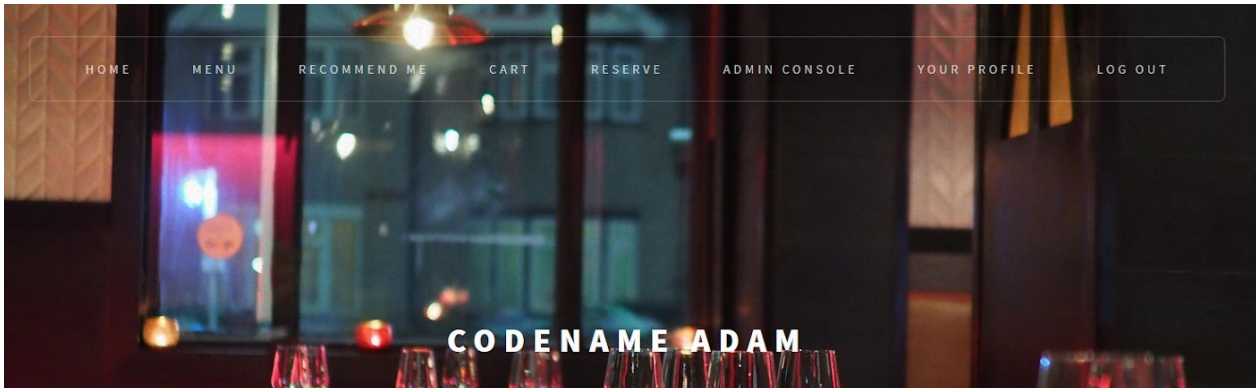
11.3: Waiter App



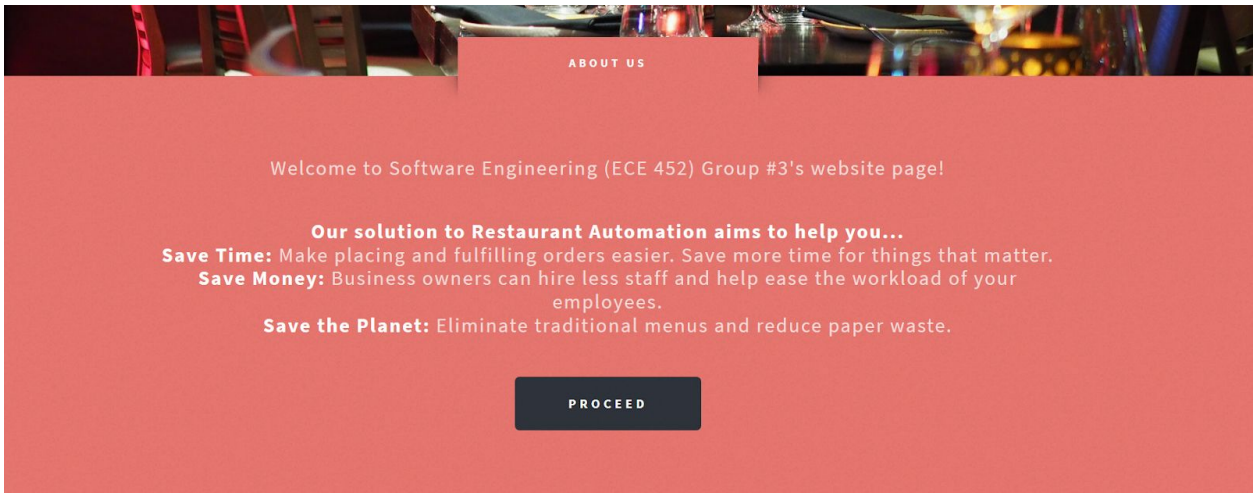
The waiter app has 2 main screens, Current Orders and Server Orders. Current Orders view has a list of all current orders. Each order has a status of the order (In Progress or Cooked) as well as a button to mark it served. When the order is marked as served it disappears from the Current Orders View and it's displayed in the Served Orders List.

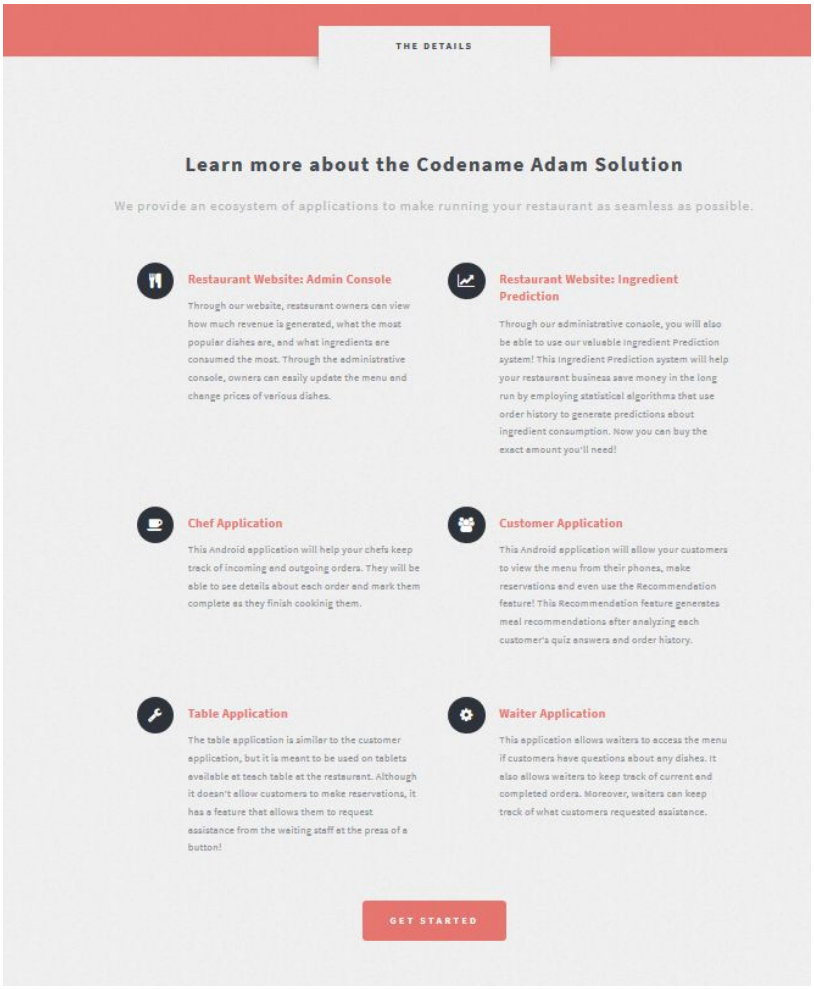
11.4: Restaurant Website



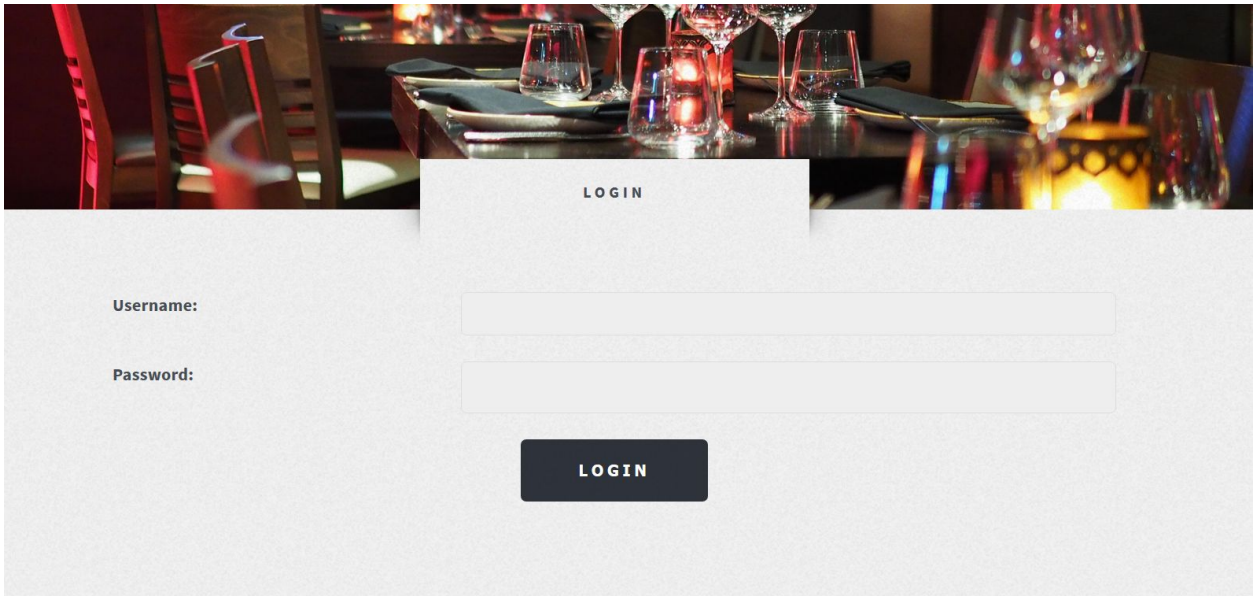


These are two views of the navbar to display its dynamic nature. The top picture is the navbar view from a non-logged in user and the bottom is when the restaurant owner is logged in (Admin Console option displayed).

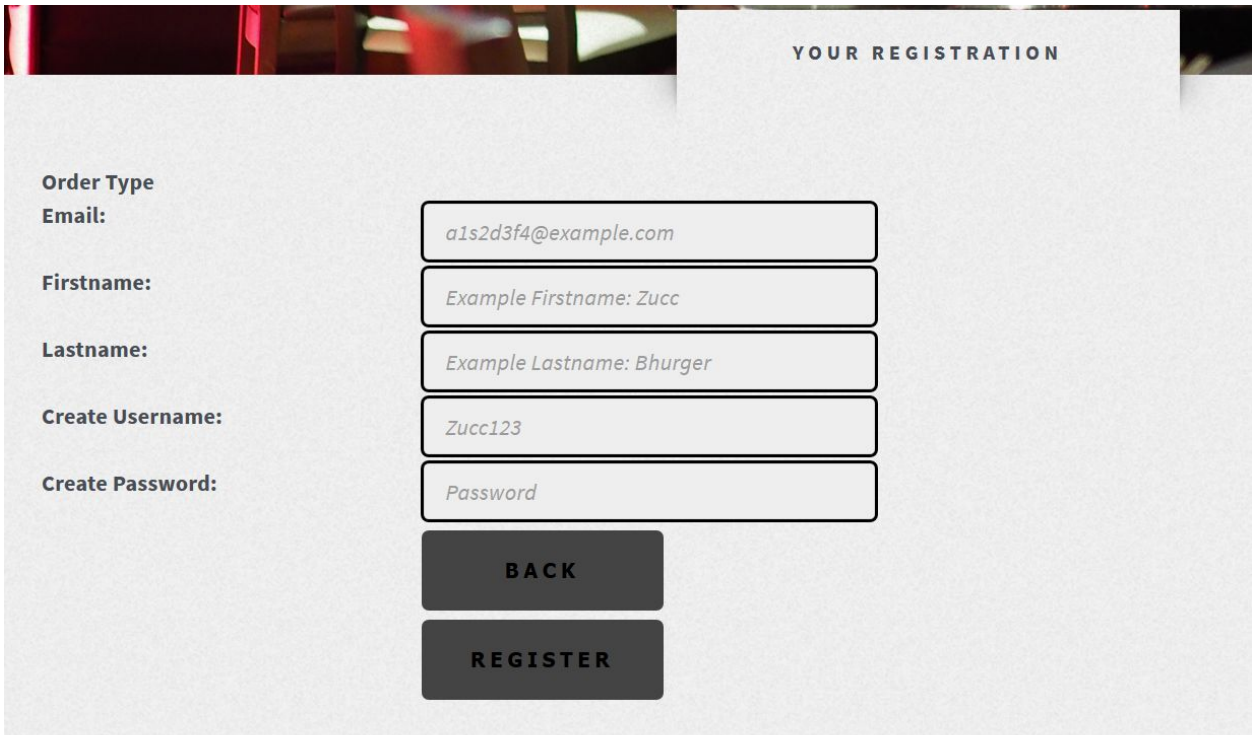




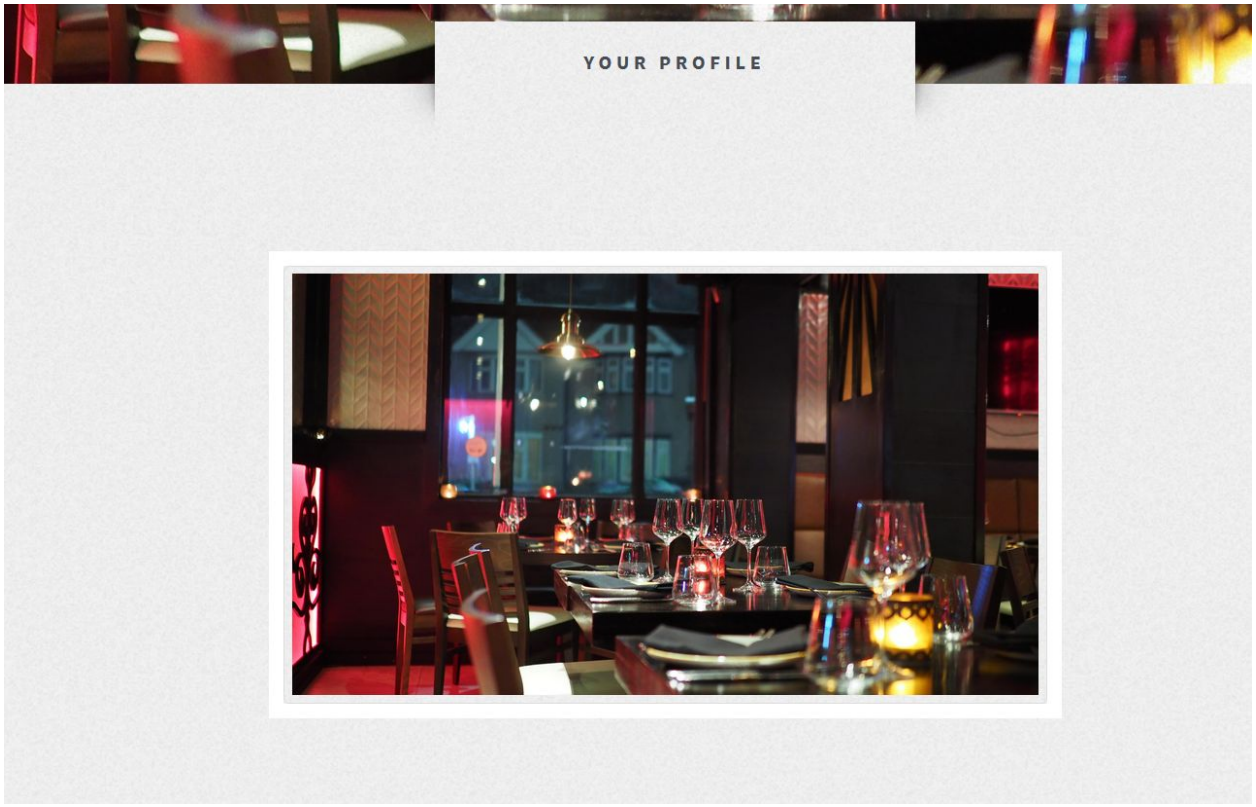
These two images are screenshots of the landing page. One section introduces the user to our group's project while the second section explains how our solutions make restaurant automation more efficient.

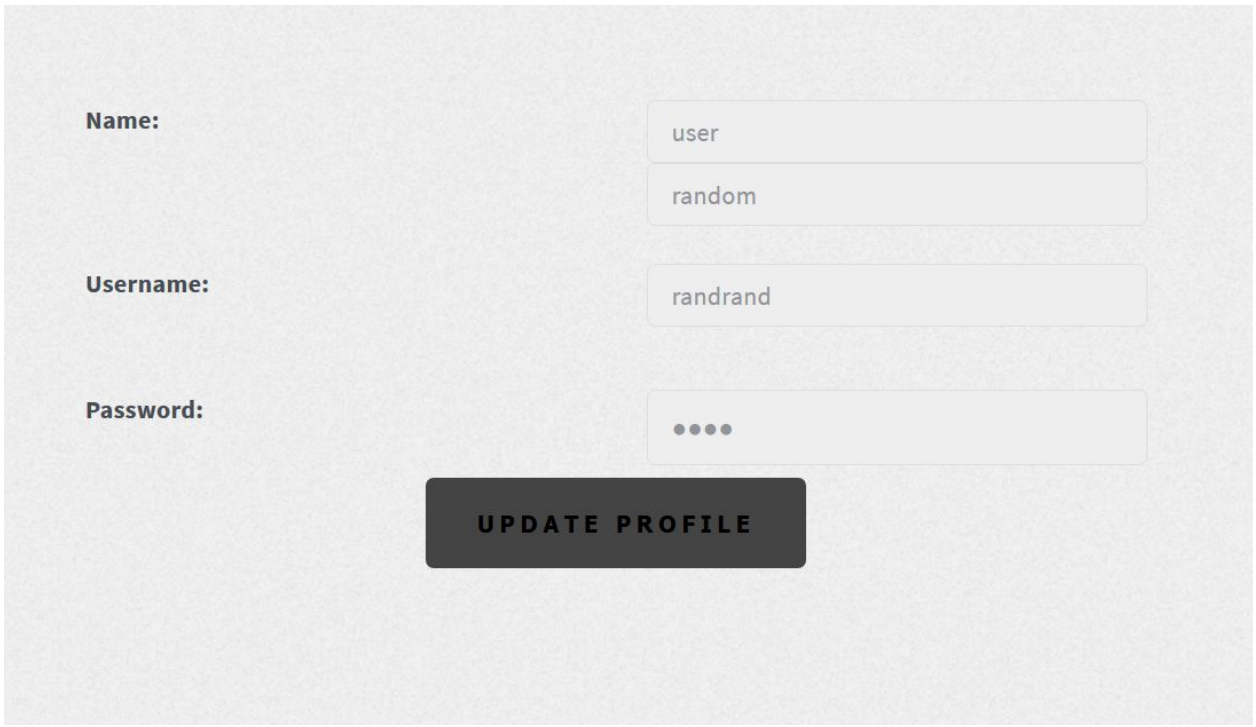


User is prompted to login when using recommendation feature, reservation system, placing orders and adding items to cart.

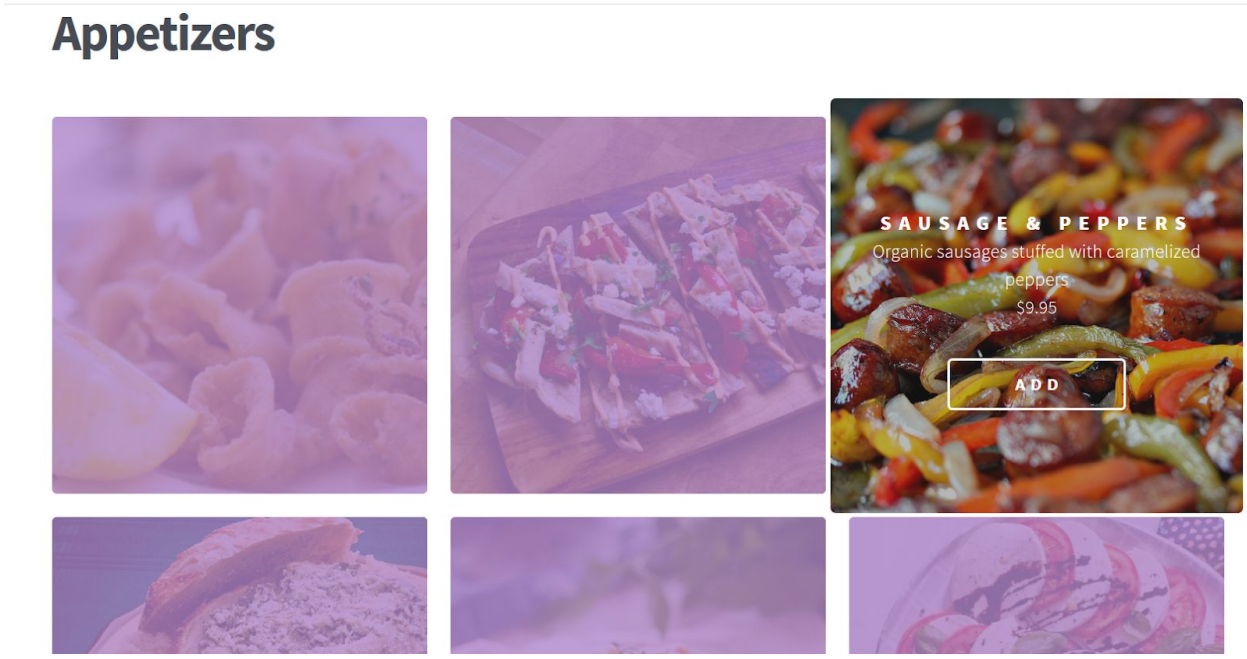


The registration form that allows new customers to make accounts.

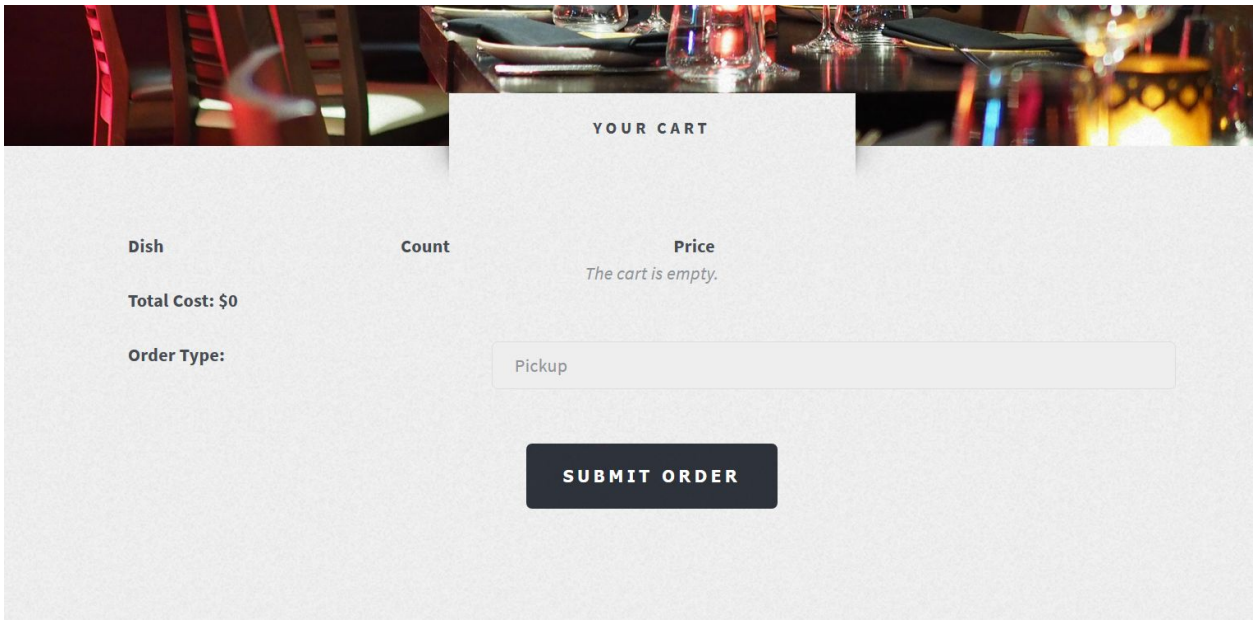


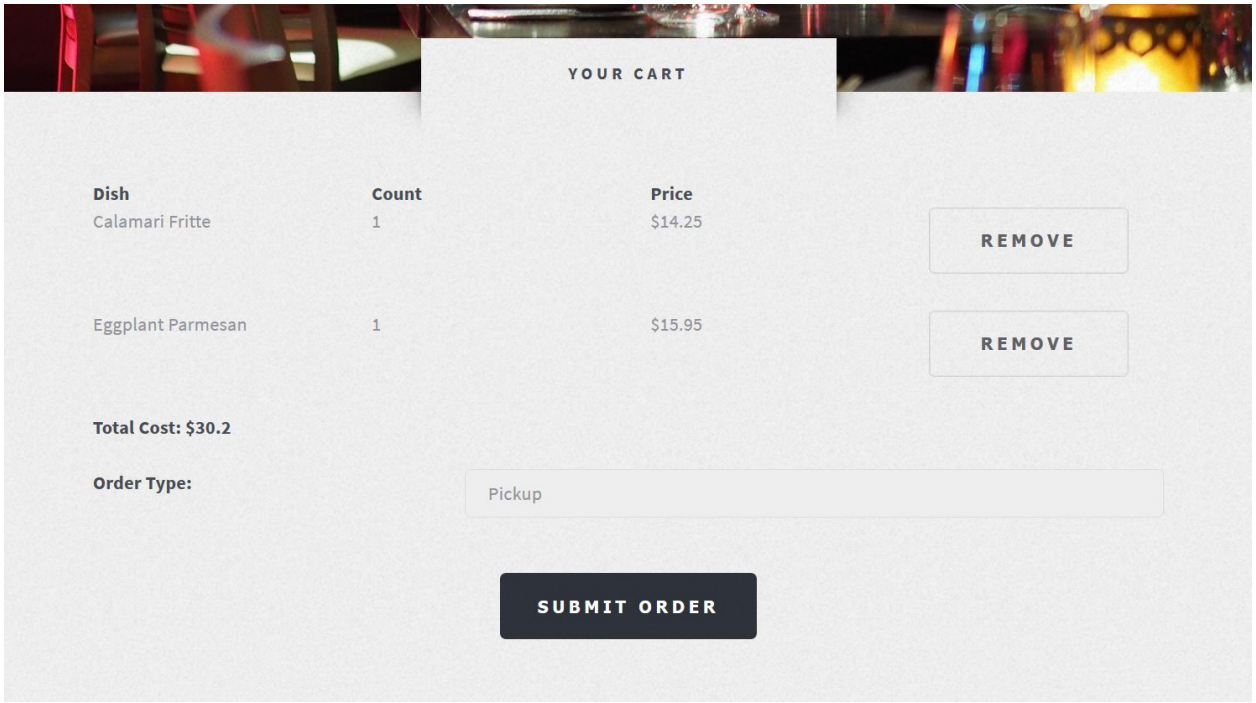


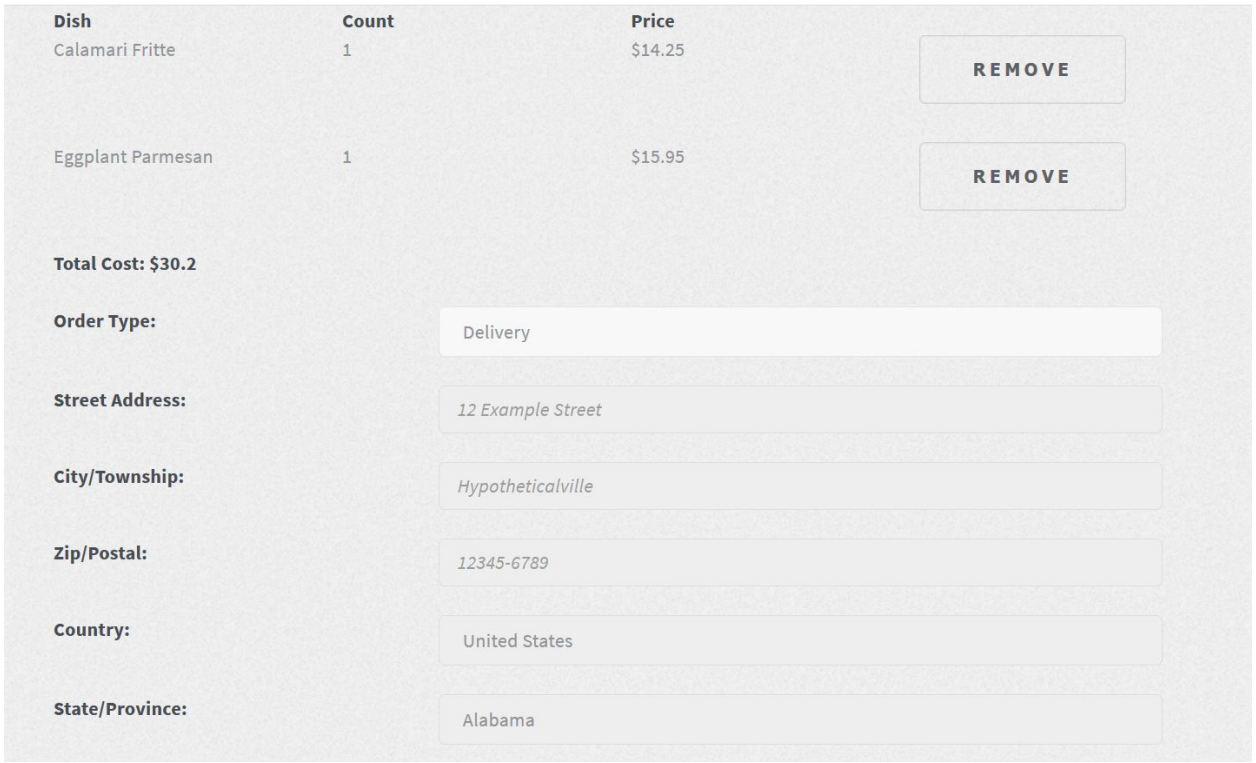
These two images comprise the user profile information page and allow users to update their information.



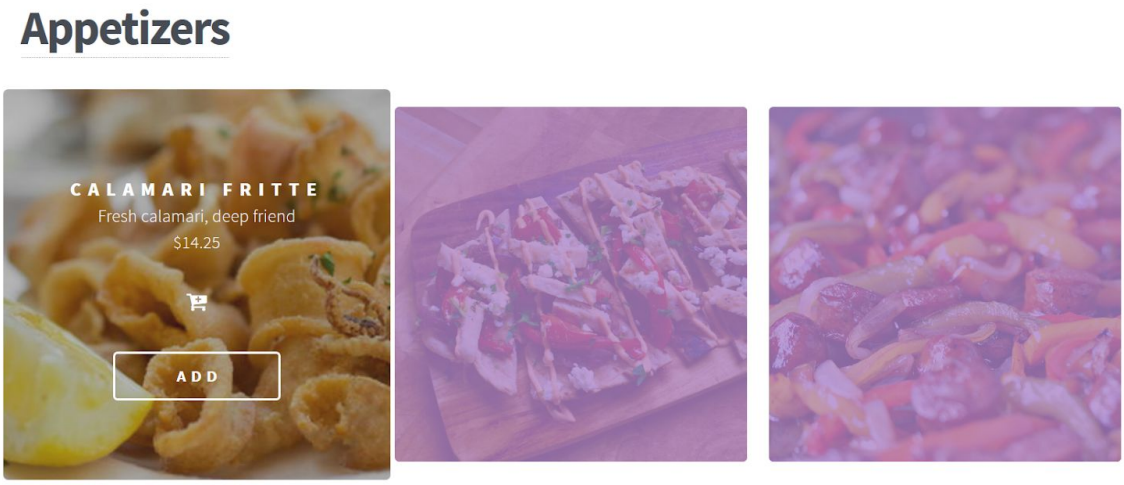
This is now dishes appear on the menu page. The ‘add’ button allows users to add them to the cart (which is empty in the picture below).



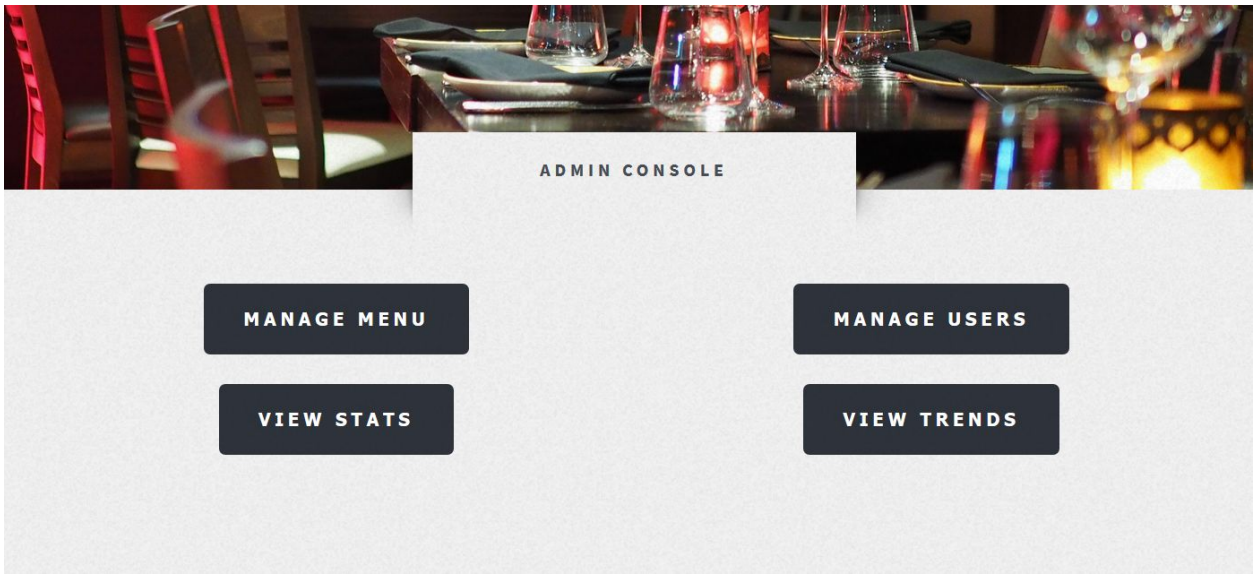




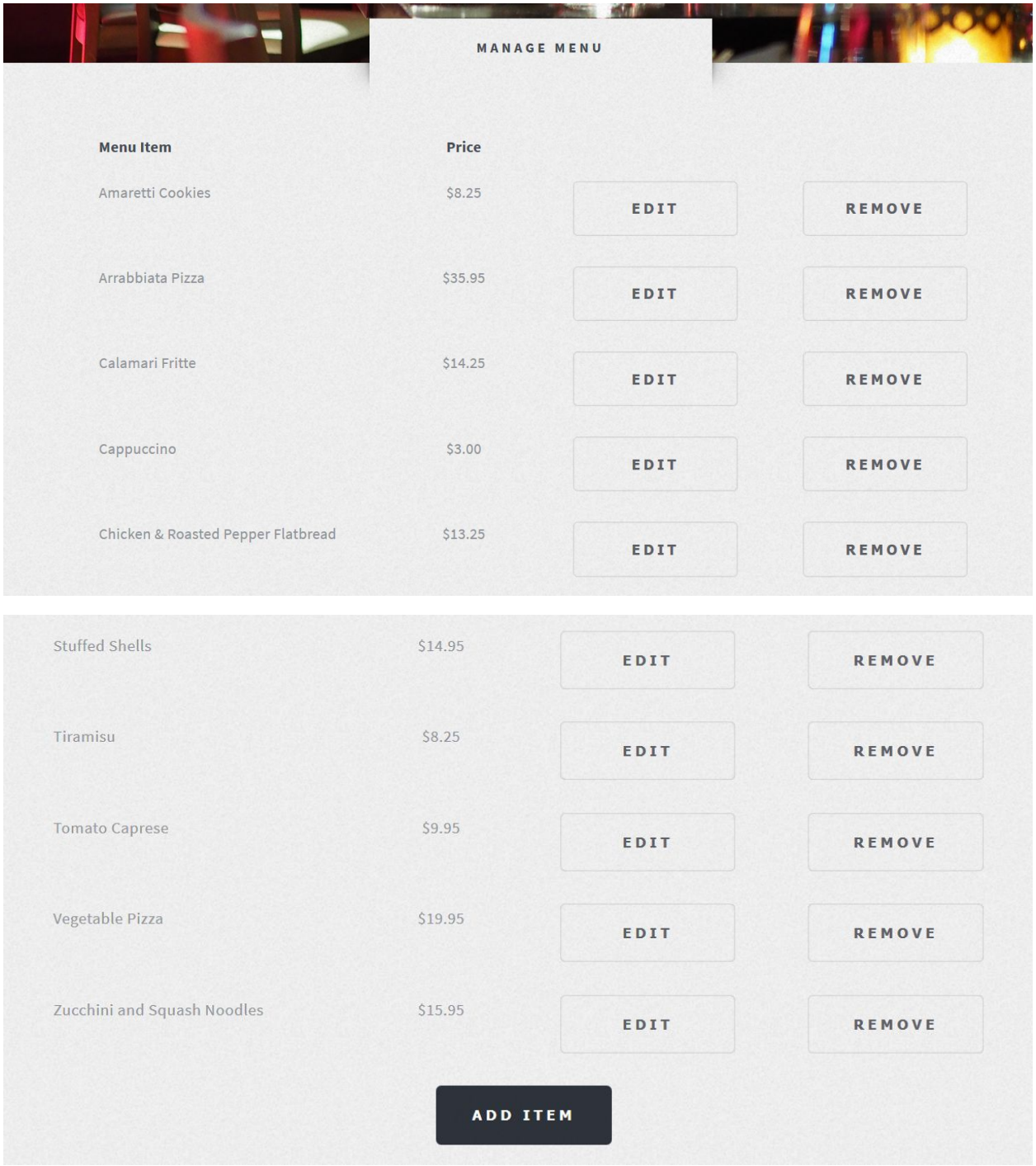
These are two different views of the cart. Customers are given the option to place orders for pick-up or delivery. Upon selecting delivery, the delivery form appears dynamically.



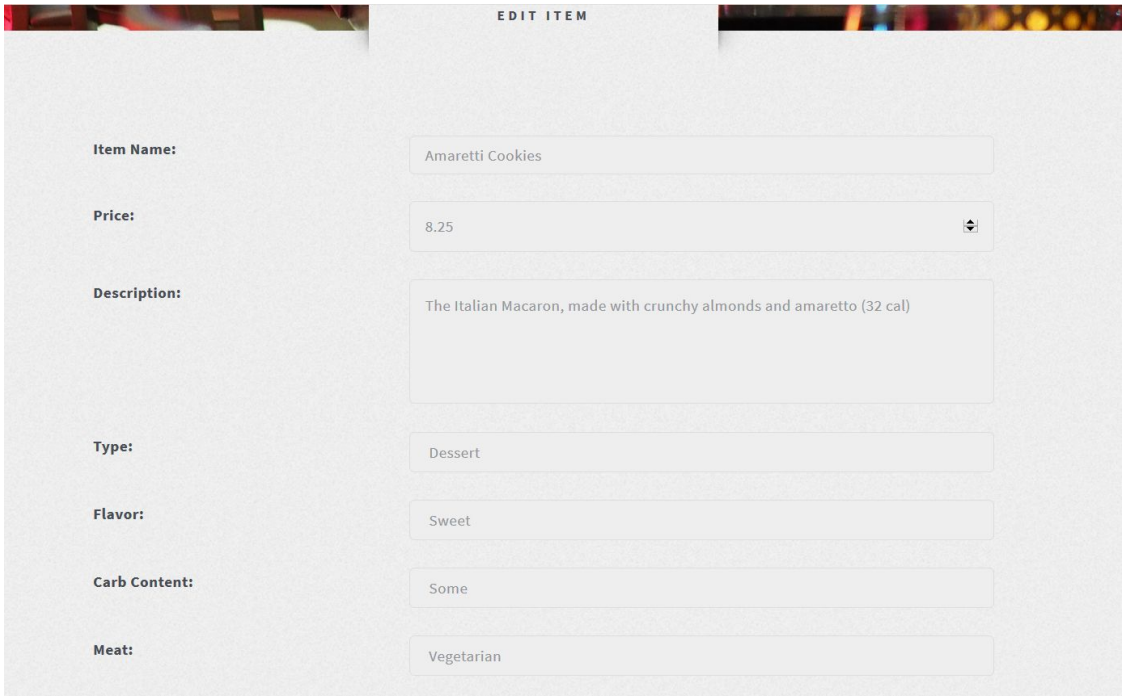
Once items are added to cart, this cart icon appears on the menu item on the menu page.



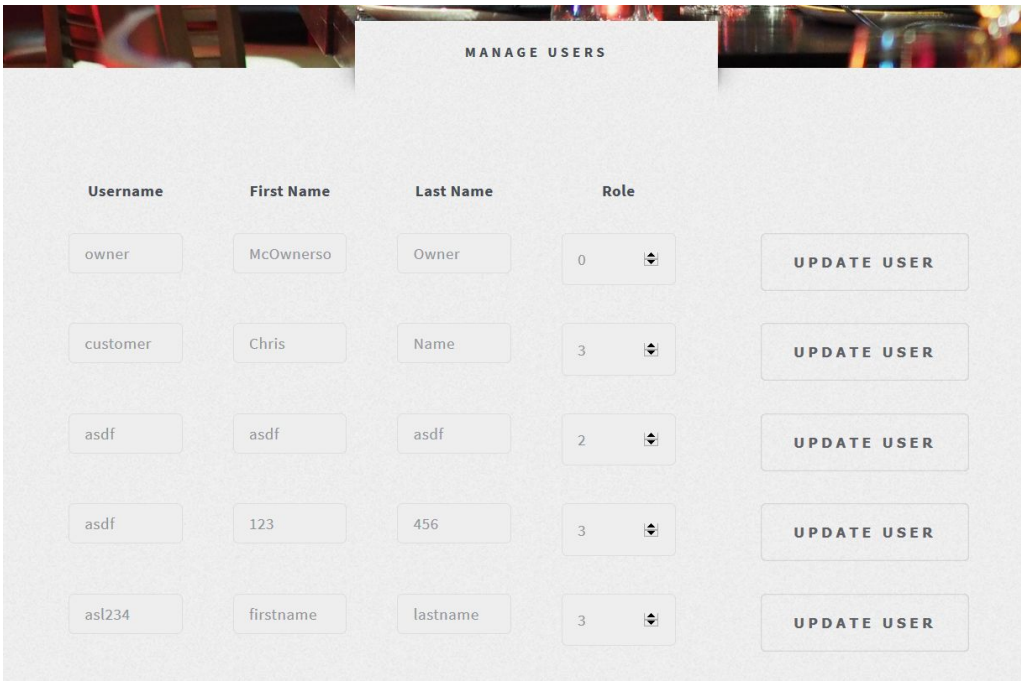
The admin console menu. Restaurant owners have the option to see these four submenus.



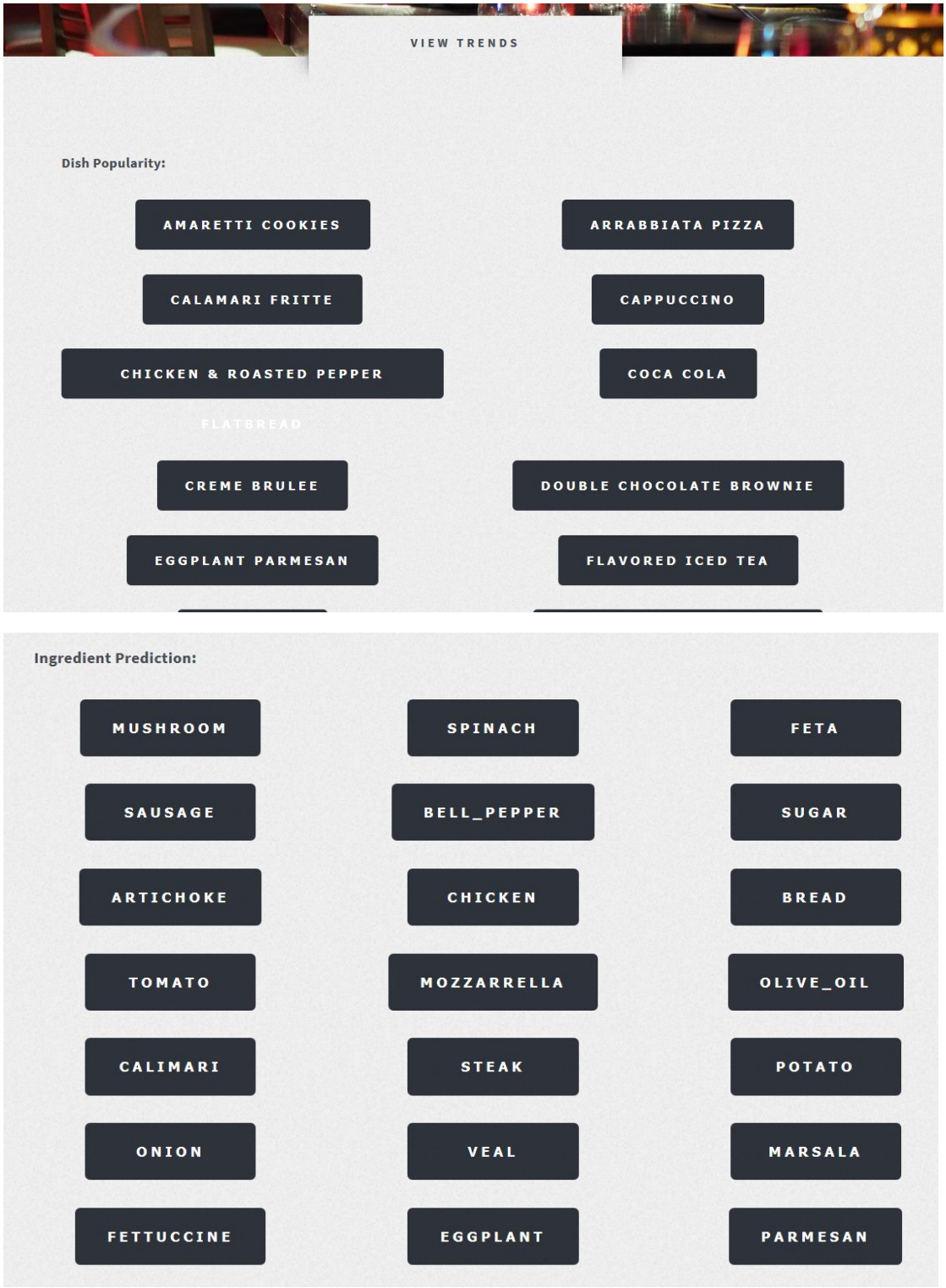
These two images comprise the 'Manage Menu' screen, which allows restaurant owners to update menu items as they wish.



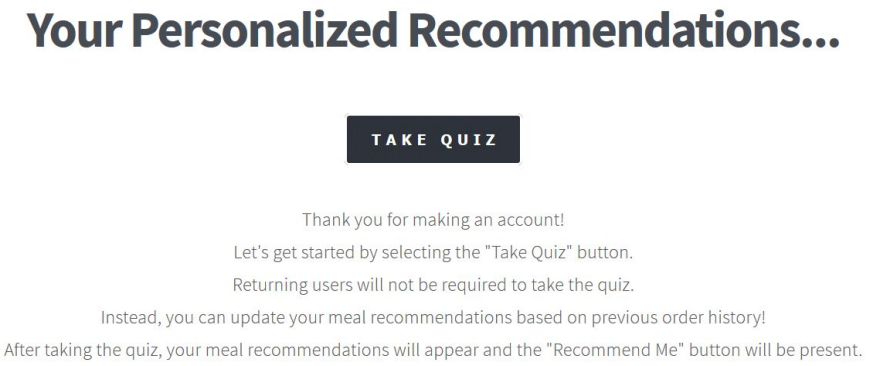
The edit menu item screen. Restaurant owners can edit names, prices, type of food and tags used for recommendation system.



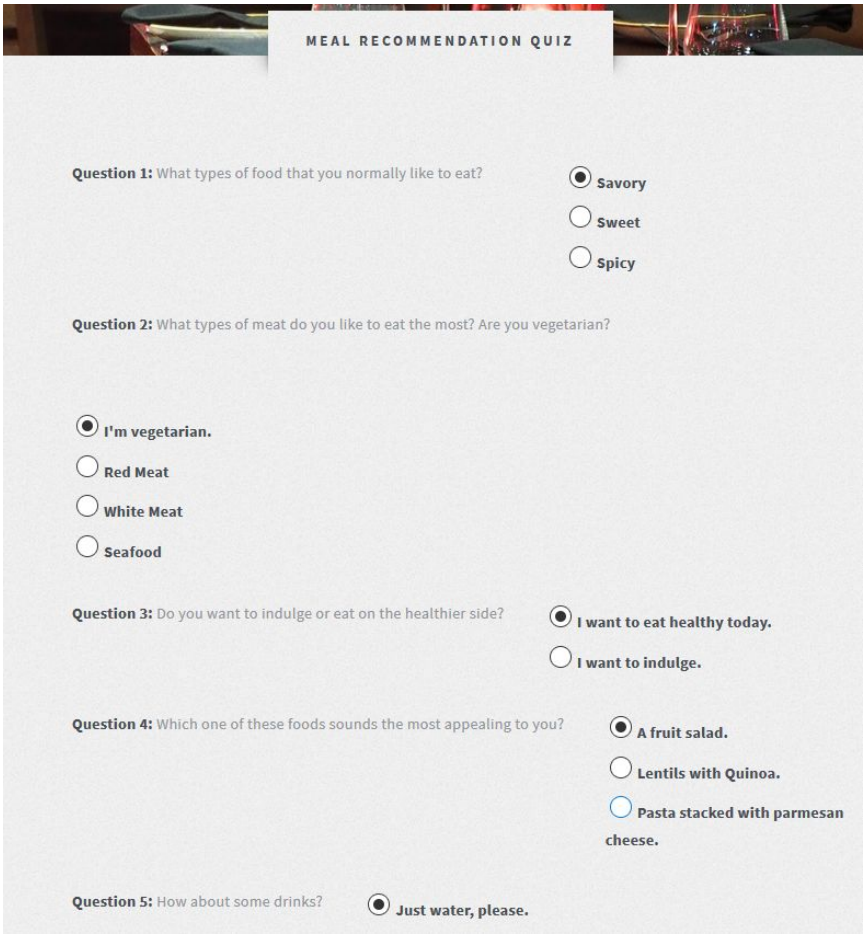
The manage users screen, which allows restaurant owners to update user permissions.



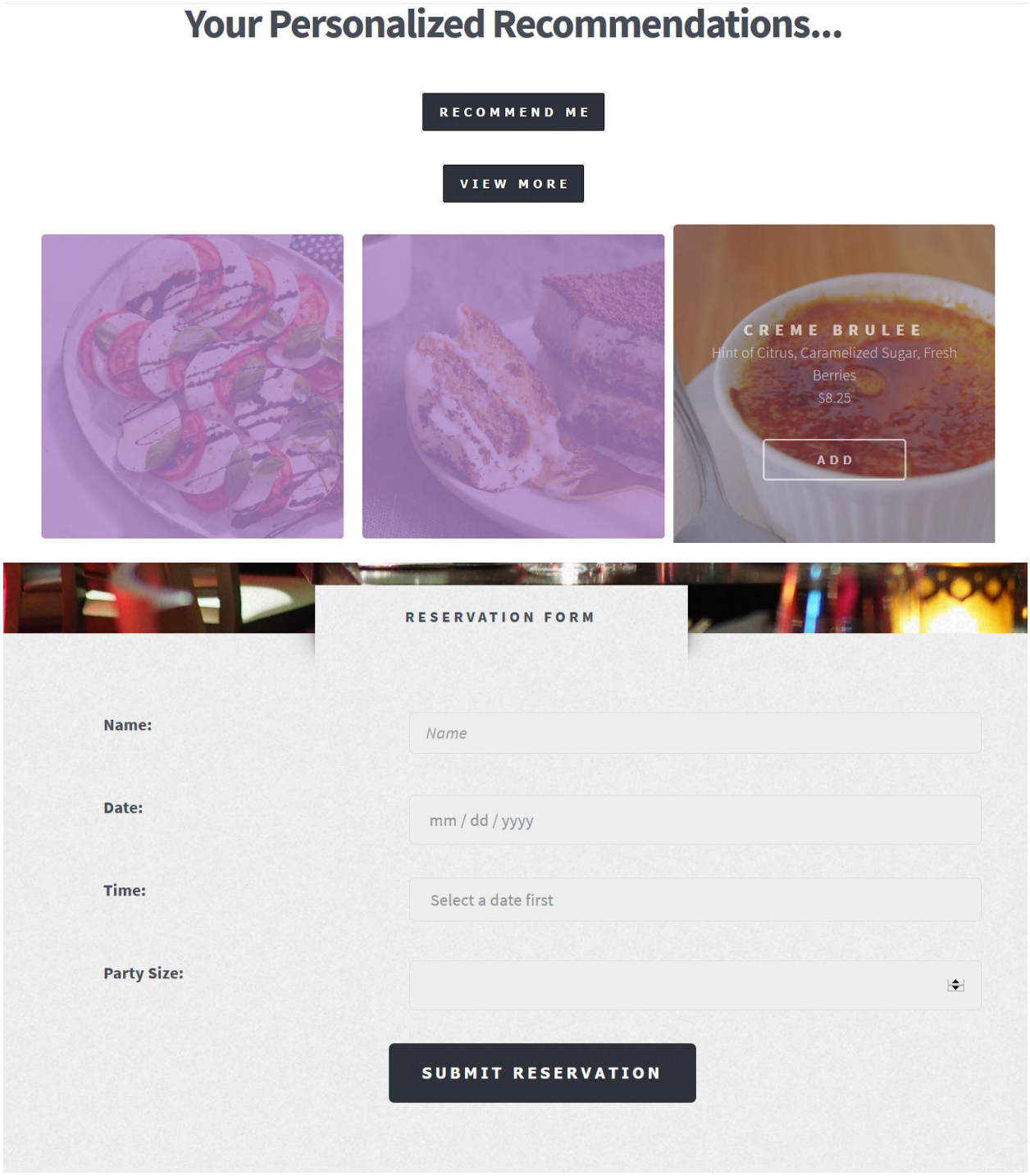
The view trends page, that allows restaurant owner to view relevant statistics per list item.



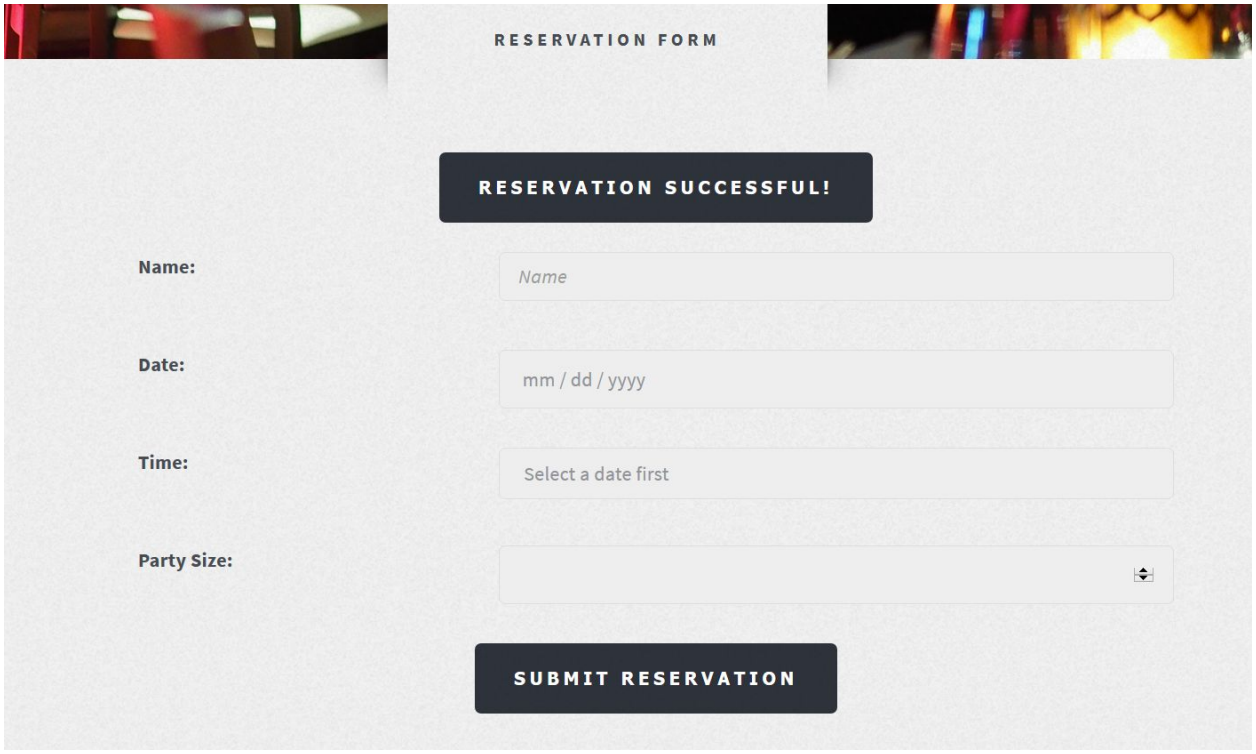
How the recommendation page appears to new users. Only new users have the option to take the quiz.



A snippet of the meal recommendation quiz page. Upon submitting, user is redirected to the view below.



A view of the reservation form. Customers must be logged in to fill it out.



How a successful reservation confirmation page looks like.

Section 12: Design of Tests

12.1: Unit Tests

On the server side, since we are using NodeJS as our back-end, we are testing using Mocha, which is a Javascript test framework. We will also use Mongo-Unit to perform tests for database related tasks. For the back-end, we have two primary types of tests: status tests and database tests. Status tests will ensure that every page we call will return with the status code 200. To ensure this we will have a test for every page and API endpoint that will make an HTTP request to the corresponding page and check the response status code. The database tests will ensure that we can add, delete, and edit information in our database properly. There are also several other tests that ensure proper responses to various inputs. The primary tests are outlined below:

```
describe('Orders Database Tests', function(){

    beforeEach(() => { //Before each test we empty the database
       return new Promise(function(resolve){
           MongoClient.connect(dbURL)
           .then(function(db){
```

```
                var collection = db.collection(ordersCollection);

                collection.remove({})
                resolve()
            })
        })});

    it('should return get no orders from empty orders collection',
function(done){
        request.get(base_url + 'orders', function(err, response, body){
            expect(response.body).to.equal('[]');
            done();
        });
    });

    it('should successfully place order', function(done){
         var d = new Date();
         var timestamp = d.toDateString() + d.toTimeString();

        var formData = {
            userId: "5c967e32d2e79f4afc43fdef",
            orderItems: ["Sprite"],
            date_placed: timestamp,
            paid: "false",
            completed: "false",
            served: "false"
        };

        request.post({url: base_url + 'placeOrder', formData: formData},
function(err, response, body){
            expect(response.statusCode).to.equal(200);
            done();
        });
    });

    it('should successfully place order', function(done){
         var d = new Date();
```

```
        var timestamp = d.toDateString() + d.toTimeString();

       var formData = {
           userId: "5c967e32d2e79f4afc43fdef",
           orderItems: ["Sprite"],
           date_placed: timestamp,
           paid: "false",
           completed: "false",
           served: "false"
       };

       request.post('http://52.39.140.122/submit-cart-website',
function(err, response, body){
           expect(response.statusCode).to.equal(302);
           done();
       });
   });

   it('should successfully get order from /orders after making order',
function(done){
       request.post('http://52.39.140.122/submit-cart-website',
function(err, response, body){
           expect(response.statusCode).to.equal(302);
           request.get(base_url + 'orders', function(err, response, body){
                expect(response.body).to.not.equal('[]');
                done();
             });
       });
   });

   it('should successfully get order from orders/id after making order',
function(done){
       request.post('http://52.39.140.122/submit-cart-website',
function(err, response, body){
           expect(response.statusCode).to.equal(302);
           request.get(base_url + 'orders', function(err, response, body){
                expect(response.body).to.not.equal('[]');
```

```
                var body = JSON.parse(response.body);
                request.get(base_url + 'orders/' + body[0]._id,
function(err, response, body){
                expect(response.statusCode).to.equal(200);
                var body = JSON.parse(response.body);
                done()
            });
            });
        });
    });

    it('should successfully set order complete', function(done){
        request.post('http://52.39.140.122/submit-cart-website',
function(err, response, body){
            expect(response.statusCode).to.equal(302);
            request.get(base_url + 'orders', function(err, response, body){
                expect(response.body).to.not.equal('[]');
                var body = JSON.parse(response.body);
                var id = body[0]._id
                request.get(base_url + 'orders/' + id + "/complete",
function(err, response, body){
                expect(response.statusCode).to.equal(200);
                var body = JSON.parse(response.body);
                expect(body.success).to.be.true;
                done()
            });
            });
        });
    });

    it('should successfully set order served', function(done){
        request.post('http://52.39.140.122/submit-cart-website',
function(err, response, body){
            expect(response.statusCode).to.equal(302);
            request.get(base_url + 'orders', function(err, response, body){
                expect(response.body).to.not.equal('[]');
                var body = JSON.parse(response.body);
```

```
                var id = body[0]._id
                request.get(base_url + 'orders/' + id + "/serve",
function(err, response, body){
                expect(response.statusCode).to.equal(200);
                var body = JSON.parse(response.body);
                expect(body.success).to.be.true;
                done()
            });
            });
       });
   });
});

describe('Assistance Requests Tests', function(){

    beforeEach(() => { //Before each test we empty the database
       return new Promise(function(resolve){
           MongoClient.connect(dbURL)
           .then(function(db){
               var collection =
db.collection(assistanceRequestsCollection);

               collection.remove({})
               resolve()
           })
       })});

    it('should return get no orders from empty assistanceRequests
collection', function(done){
       request.get(base_url + 'assistanceRequests', function(err,
response, body){
           expect(response.body).to.equal('[]');
           done();
       });
   });

   it('should successfully create assistance request', function(done){
```

```
        var options = {
            method: 'POST',
            uri: base_url + 'assistance',
            body: {
                user_id: "5c967e32d2e79f4afc43fdef",
                timestamp: "..."
            },
            json: true
        };

        request(options, function(err, response, body){
            expect(response.statusCode).to.equal(200);
            done();
        });
    });

    it('should successfully get order from /assistanceRequests after
creating request', function(done){

        var options = {
            method: 'POST',
            uri: base_url + 'assistance',
            body: {
                user_id: "5c967e32d2e79f4afc43fdef",
                timestamp: "..."
            },
            json: true
        };

        request(options, function(err, response, body){
            expect(response.statusCode).to.equal(200);
            request.get(base_url + 'assistanceRequests', function(err,
response, body){
                  expect(response.body).to.not.equal('[]');
                  done();
              });
```

```
        });
    });

    it('should successfully get order from assistanceRequests/id after
making order', function(done){

         var options = {
            method: 'POST',
            uri: base_url + 'assistance',
            body: {
                user_id: "5c967e32d2e79f4afc43fdef",
                timestamp: "..."
            },
            json: true
        };

        request(options, function(err, response, body){
            expect(response.statusCode).to.equal(200);
            request.get(base_url + 'assistanceRequests', function(err,
response, body){
                  expect(response.body).to.not.equal('[]');
                  var body = JSON.parse(response.body);
                  request.get(base_url + 'assistanceRequests/' +
body[0]._id, function(err, response, body){
                  expect(response.statusCode).to.equal(200);
                  var body = JSON.parse(response.body);
                  done()
              });
              });
        });
    });
});
```

For the apps, we will use JUnit to write unit tests. Most important ones are outlined below:

1)

```
public test OrderTest{
```

```
//Test to check that making an order for a valid menu items returns an
Order object
   @Test public void
       checkMenuItem_validId_success(){
           //MenuItem we want to order
           MenuItem item = new MenuItems();
           item.menuItemId = 1; //1 is an Id of a valid MenuItem

           //User placing an order
           User user = new User();

           Order order = item.order(user);

           //if order() is successful, it returns an Order instance
           AssertNotNull(order);
       }

   @Test public void
       checkMenuItem_inValidId_success(){
           //MenuItem we want to order
           MenuItem item = new MenuItems();
           item.menuItemId = -1; //-1 is not an id of a MenuItem in the
database

           //User placing an order
           User user = new User();

           Order order = item.order(user);

           //if order() is not successful, it should return null
           AssertNull(order);
       }
}

2)
public test RequestAssistanceTest{
   //Test to ensure that requesting assistance works
```

```
   @Test public void
       checkController_anyState_requestAssistance(){
           Controller controller = new Controller();

           //Request assistance should always return true, no matter the
controller state.
           //Otherwise something failed - unsuccessful HTTP request,
server error, etc
           boolean requestAssistanceResult =
controller.requestAssistance();

           assertEqual(requestAssistanceResult, true);
       }
}

3)
public class ReservationTest{
   //Test to check that making reservation for an available table returns
a reservation object

   @Test public void
       checkTable_validId_reserve{
           //Table we want to reserve
           Table table = new Table();
           table.tableId = 1; //1 is a valid id


           //User making reservation
           User user = new User();
           user.firstName = "user1";

           //Start datetime and end datetime entered by user
           DateTime start = new DateTime("03/17/2019", "3pm");
           DateTime end = new DateTime("03/17/2019", "4pm");

           Reservation reservation = table.reserve(start, end,
user.firstName);
```

```
            //if reserve() is successful, it returns a Reservation instance
            AssertNotNull(reservation);
        }

    @Test public void
        checkTable_inValidId_reserve{
        //Table we want to reserve
            Table table = new Table();
            table.tableId = -1; //-1 is not an id of a Table in the
database


            //User making reervation
            User user = new User();
            user.firstName = "user1";

            //Start datetime and end datetime entered by user
            DateTime start = new DateTime("03/17/2019", "3pm");
            DateTime end = new DateTime("03/17/2019", "4pm");

            Reservation reservation = table.reserve(start, end,
user.firstName);

            //if reserve() is not successful, it returns null
            AssertNull(reservation);
        }
    @Test public void
        checkTable_duplicateReservation_reserve{
        //Table we want to reserve
            Table table = new Table();
            table.tableId = 1; //1 is a valid id


            //User making reervation
            User user = new User();
            user.firstName = "user1";
```

```
            //Start datetime and end datetime entered by user
            DateTime start = new DateTime("03/17/2019", "3pm");
            DateTime end = new DateTime("03/17/2019", "4pm");

            Reservation reservation_one = table.reserve(start, end,
user.firstName);

            //Placing a reservation for a table when another reservation
for the same time already exists should retuen null
            Reservation reservation_two = table.reserve(start, end,
user.firstName);

            AssertNull(reservation_two);
        }
}

4)
public test OrderCompleteTest{
   //Test to ensure that completing a valid order returns true
   @Test public void
        checkOrder_validId_complete(){
            Order order = new Order();
            order.orderId = 1; //1 is a valid id (there is an order with id
1 in the database)

            boolean result = order.complete();

            assertEqual(result, true);
        }

   //Test to ensure that completing a valid order returns true
   @Test public void
        checkOrder_inValidId_complete(){
            Order order = new Order();
            order.orderId = -1; //-1 is not a valid id
```

```
            boolean result = order.complete();

            assertEqual(result, false);
        }
}
```

12.2: Test Coverage

The tests we write will cover all of the main parts we are aiming to get ready for the demo. Specifically the parts we need to implement our 4 most important use cases: UC-1: Order Food, UC-3: OrderComplete, UC-4: Assistance Needed and UC-6: FinishOrder. The tests will verify proper operation of both the back-end code, as well as the Android apps.

12.3: Integration Tests

Test-Case Identifier: TC - 1		
Use Case Tested: UC - 1,2 Pass/Fail Criteria: Test passes if 1.customer successfully views the menu, added food to the order and pay the money. 2. When customer views the order, food recommended will be listed at top. 3. server successfully receives the order and send the order to chef. 4. chef successfully receives the order from server. Test fails if any of these does not happen successfully. Input Data: Button Selection by clicking Payment information entered by user Item Selection by clicking		
Test Procedure	Expected Result	Actual Result
Step1: Customer open up the menu by clicking the “menu” item in the navigation side bar in the customer app Step2: Customer add or delete food by clicking the “+” or “-” contained in each food item. Step3: Customer enters the payment information and confirm the payment. Step4: Order is made and will be sent to server’s app and added to the server app. Step5:	Customer can view the menu and see the recommending food in the menu top. Customer can either add or delete the food. Customer can pay the order. Server can receive the order once it is paid and add it to server app. Server can notify the chef that an order is made. Chef can receive the notification from server and add this order to the chef’s app.	Success: Customer can view the menu and see the recommending food in the menu top. Customer can either add or delete the food. Customer can pay the order. Server can receive the order once it is paid and add it to server app. Server can notify the chef that an order is made. Chef can receive the notification from server and add this order to the chef’s app. Failure: One action listed above not happening will lead to failure.

Server sends this order to the chef app and the order will be added to the chef app		

Test-Case Identifier: TC - 2		
Use Case Tested: UC - 3,8 Pass/Fail Criteria: Test passes if the order is successfully removed in the server's app after customer received all the food they ordered. Test fails if the order is not removed. Input Data: Button Selection by clicking		
Test Procedure	Expected Result	Actual Result
Step1: chef has finished the last food in the order and click "finish" button and click "complete order" button on his or her chef app. Step2: a server will receive a notification by saying that all food is finished Step3: server complete the order by clicking the "complete order" button on the server app	Chef send the notification to the server and successfully remove the order from its app. Server receive the notification sent by the chef that the last piece of food in the order is complete. Server successfully deleted the order from its own server app.	Success: Chef sends out the notification. Chef remove the order from chef app. Server receives the notification. Server removes the order from server app. Failure: Each action listed above that does not happen will lead to the failure.

Test-Case Identifier: TC - 3		
Use Case Tested: UC - 4 Pass/Fail Criteria: Test passes if a notification is added to server's app after customer click the "assistance" button on the table app. Test fails if no notification is sent to server app. Input Data: Button Selection by clicking		
Test Procedure	Expected Result	Actual Result
A user requests service by clicking the "assistance" button	After clicking, a notification which includes the table number will be popped out in server's app who is assigned to this table.	Success: A notification which includes the table number will be popped out in server's app who is assigned to this table. Failure: Nothing shows in the server app.

Section 13: History of Work, Status & Future Work

We started by creating a server in NodeJS and hosting it on Amazon Web Services EC2 instance. From there we started working on the Customer, Chef, Waiter apps, as well as the basic website in preparation for demo 1. Our goal was to have our 4 most important use cases done. These use cases were UC-1 (OrderFood), UC-3 (OrderComplete), UC-4 (AssistanceNeeded) and UC-6 (FinishOrder). Additionally, we

started prototyping our machine learning algorithm in python, but we did not integrate it into our project yet.

Before Demo2, we are working on adding more functionality to our apps and website. Most recently we added the ability to authenticate users. This allows us to restrict access to certain API endpoints based on user's role. In addition we worked on integrating the recommendation system with the server, and creating API endpoints to expose this functionality to the apps. We also worked on the admin console, which would allow the restaurant owner to monitor the restaurant's activity and update the menu. Moreover, we worked on to add a timer for each order inside chef app, so chefs can check how much time they have spent on a specific order. We also allow users, especially customers, to configure their profiles like changing password or something. We also worked on allowing users to reserve a table with a specific table ID and specific time slot and if a table is occupied, the app or website should notify the users that which table is occupied in which time slot. What's more, we worked on using Google Firebase to implement message sending function. For example, when customers of a table want to request assistance, then Google Firebase will take care of this. The most fancy feature we are going to add is that we would integrate facial recognition function into our customer app so that customers can login with their face if they allow the camera to do it. These are the things we want to have done before the second demo.

After Demo2, we have almost finished all the plans we were considering to work. All applications work well. In the future, we would like to rework the UI of the apps and eliminate the non-critical bugs we've found, but haven't had the chance to fix.

Section 14: Project Management

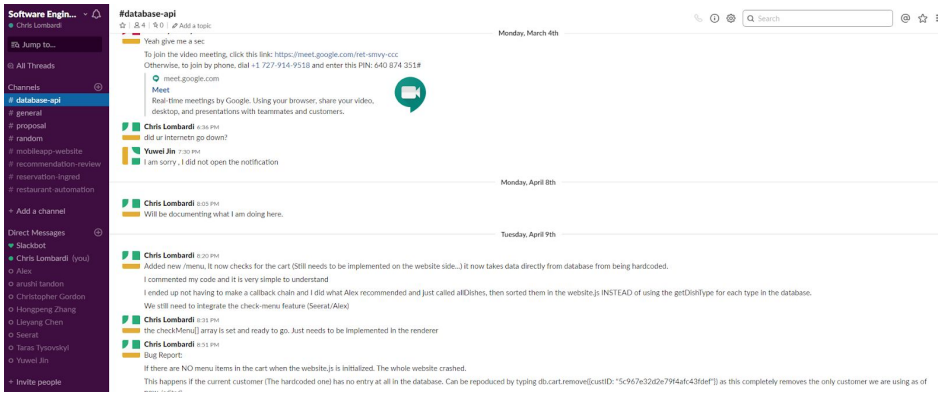
Our group was managed through through three main Applications.
Asana, Slack, and Groupme. Along with this, our group meets every week after class on Friday.

Asana is a ticket system that helps us keep track of what needs to be done. Each task can be assigned to a person and have a set due date. This helps us distribute the workload. Here is an example of our tickets that were created for this report:

Our next application we used was groupme, this was a chat messaging system where we can figure out when people can meet to resolve certain tasks. While this application was used plentiful, it had some problems since all messages go into the same chat room, which created some clutter.

Our final application we used was slack, this was a more organized chat messaging system. We made different “chat rooms” that were used for each platform. For example, we had one for the database, one for the web application, one for the chefs app, ect… This type of chat helped our chat be less cluttered of individuals trying to talk about different topics, while still allowing everyone to be able to read every message. (An issue that would occur if everyone was sending private messages.

Slack example,



To wrap everything up, on every Friday after class we would all meet together and create any new tickets for our new tasks. We would also make sure everyone is feeling like they are doing their part, without having to do too much. Overall, once we started to use of these applications together, our development became swift and efficient.

Section 15: References

1. Why W8 - Fall 2018 Restaurant Automation Project
2. FoodEZ - Spring 2015 Restaurant Automation Project

3. Professor Marsic's Restaurant Automation Project Description (https://www.ece.rutgers.edu/~marsic/books/SE/projects/Restaurant/)
4. Spyce (Boston restaurant with a robotic chef) - https://www.greenbiz.com/article/full-service-automation-restaurants-changing-food-industry
5. Professor Marsic's Lecture 10 Notes (Object Oriented Design II)
6. UML tool: https://staruml.io
7. https://www.mountaingoatsoftware.com/articles/estimating-with-use-case-points (to help with section 5)